

Introduction

Helloworld

Pour démarrer l'apprentissage d'un programme, rien de tel que l'écriture du fameux HelloWorld.

Fichier [Helloworld.java](#)

```
public class Helloworld {  
  
    /**  
     * point d'entrée des programmes java. Il est nécessaire de respecter  
     * exactement la signature  
     * de cette méthode afin que le programme puisse s'exécuter.  
     *  
     * @param args tableau d'arguments passés lors du lancement du  
     * programme  
     */  
    public static void main(String[] args) {  
        System.out.println("Hello world !");  
    }  
}
```

Pour compiler le programme, il est nécessaire de se positionner dans le repertoire **src**

```
cd src/main/java  
javac Helloworld.java  
java Helloworld
```

En java, toute ligne d'instructions est terminée par un **point virgule**

Depuis Java 23, il est possible de créer une méthode *main* sans être dans une classe [JEP 477](#)

Déclaration d'une variable

Pour utiliser une variable, il est nécessaire de la déclarer et de lui assigner une valeur. Lors de la déclaration, on définit le type de la variable (primitif, classe, record, enum) suivi du nom de la variable.

Fichier [Variables1.java](#)

```
public class Variables1 {  
  
    public static void main(String[] args) {  
        /**  
         * Declaration et assignement d'une variable
```

```
    */
    int magicNumber = 62;

    /**
     * Une chaine de caractères peut être concaténée à tout type
     d'éléments
     */
    System.out.println("Le nombre magique est " + magicNumber);

    /*
     * L'utilisation de la méthode printf permet d'écrire des chaines
     de caractères
     * en formattant la chaine de chaine de caratères.
     */
    System.out.printf("Le nombre magique est %d", magicNumber);

    // des variable
    short year = 2024;
    char a = 'a';

    double montant = 3_000;

    long longNumber = 5_678_232;
    long longNumber2 = 5_678_232l;

    LocalDateTime now = LocalDateTime.now();

    boolean yes = true;
}
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Variables1
Le nombre magique est 62
Le nombre magique est 62
```

Il est tout à fait possible de déclarer puis, par la suite, d'assigner une valeur à cette variable.

Fichier **Variables2.java**

```
public class Variables2 {

    public static void main(String[] args) {
        /**
         * Declaration puis assignement
         */
        String name;
        name = "Java";
        System.out.printf("Un langage toujours au top : %s", name);
    }
}
```

```
    }  
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Variables2  
Un langage toujours au top : Java
```

tant qu'une variable, déclarée dans le corps d'une méthode, n'est pas assignée à une valeur, elle ne peut pas être utilisée.

Fichier **Variables3.java**

```
public class Variables3 {  
  
    public static void main(String[] args) {  
        /**  
         * Declaration sans assignement  
         */  
        int ageDuCapitaine;  
        System.out.printf("L'age du capitaine est %d", ageDuCapitaine);  
    }  
}
```

```
javac Variables3.java  
Variables3.java:8: error: variable ageDuCapitaine might not have been  
initialized  
        System.out.printf("L'age du capitaine est %d", ageDuCapitaine);
```

Les primitives

Java n'est pas un langage purement objet. 8 primitives sont mises à disposition au développeur.

Catégorie	Type	Taille	Signé	Min	Max
Entier	byte	2 ⁸	signed	-128	127
	char	2 ¹⁶	unsigned	0	2 ¹⁶ -1
	short	2 ¹⁶	signed	-2 ¹⁵	2 ¹⁵ -1
	int	2 ³²	signed	2 ²	2 ³¹ -1
	long	2 ⁶⁴	signed	-2 ⁶³	2 ⁶³ -1
Decimaux	float	2 ³²	signed	-2 ⁻¹⁴⁹	(2-2 ⁻²³) * 2 ¹²⁷
	double	2 ⁶⁴	signed	-2 ⁻¹⁰⁷⁴	(2-2 ⁻⁵²) * 2 ²¹⁰²³

Catégorie	Type	Taille	Signé	Min	Max
Autre	boolean				

Les opérateurs d'assignement permettent de modifier la valeur d'une primitive.

Operateur	Nom	Exemple
=	Assigne une valeur	int i = 18;
+=	Ajoute la valeur de droite	i = 1;i += 56 // i vaut 57
-=	Soustrait la valeur de droite	i = 10;i -= 2 // i vaut 8
/=	Division par la valeur de droite	i = 10;i /= 2 // i vaut 5
*=	Multiplication par la valeur de droite	i = 10;i *= 2 // i vaut 20
%=	Application du modulo	i = 10;i %= 2 // i vaut 0

Les opérateurs mathématiques sont utilisés sur les primitives pour réaliser des opérations.

Operateur	Nom	Exemple
+	Addition	1 + 5
-	Soustraction	1 – 3
*	Multiplication	45 * 90
/	Division	1 / 10
%	Modulo (Reste de la division Euclidienne)	65 % 3
++	Incrémente de 1	i++
--	Décrémente de 1	i--

Il est possible de convertir un paramètre d'un certain type (de primitif) en un autre.

Attention, il peut y avoir des pertes de precision lorsqu'on utilise un type dont l'intervalle de definition est plus petit.

```
public class Variable5 {  
  
    public static void main(String[] args) {  
        //Explicit  
        int value = 56;  
        short castValue = (short) value;  
        System.out.println(castValue);  
  
        //Implicit  
        short valueS = 56;  
        int valueI = valueS;  
        System.out.println(valueI);  
    }  
}
```

```
//Perte de precision
int number = (int) 34.78; // numb
System.out.println(number);
}
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Variable5
56
56
34
```

Combinaison de type

Calcul	Resultat	Règle
2/3	0	(int, int) -> (int)
2/3.0	0,6666666666	(int, double) -> (double)
2.0/3.0	0,6666666666	(double, double) -> (double)
2.0 + "3.0"	2.03.0	(double, string) -> (string)

Règle :

String > double > int > char > boolean

Et si on parlait de *null*

Une variable peut :

- ne pas être définie.
- être associée à une valeur numerique, booleen, chaine de caractères, énumération, objet.
- être associée à **null**

null représente l'absence de valeur. il n'est ni un objet, ni un type. Il représente une valeur spéciale. Une variable associée à **null** mal utilisée peut entrainer l'exception **NullPointerException**.

null ne peut pas être assigné à une primitive

Le mot clé *var*

Depuis Java5, il est possible d'éviter de déclarer le type d'une variable grâce au mot clé **var**.

Dans ce cas, il est nécessaire d'assigner une valeur afin que le compilateur infère le type de la variable. Une fois le type inféré, il n'est plus possible d'assigner à la variable à une valeur d'un autre type.

Fichier **Variable4.java**

```
public class Variables4 {

    public static void main(String[] args) {
        var birthday = 1995;
        var author = "James Gosling";

        System.out.printf("birthday : %s, author : %s\n",
            ((Object)birthday).getClass().getName(), author.getClass().getName());

        var message = "Java est apparu en " + birthday + " et un des auteurs est " + author;
        System.out.println(message);
    }
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Variables4
birthday : java.lang.Integer, author : java.lang.String
Java est apparu en 1995 et un des auteurs est James Gosling
```

Loop et conditions

Avant : les opérateurs logiques

Opérateur	Nom	Exemple
&&	Et	x == y && x < z
	Ou	x == y x < z
!	Not	!(x == y && x < z)

if / else if / else

Le mot clé **if** permet d'exécuter un bloc d'instructions si la condition est vraie.

Pour comparer des primitives, on peut utiliser un des comparateurs suivants :

Comparateur	Description
==	Egalité
!=	Différent
<	Plus petit
<=	Plus petit
>	Plus grand
>=	Plus grand ou égal

Pour définir un bloc, on utilise les accolades { }

Fichier **Condition1.java**

```
public class Condition1 {  
  
    public static void main(String[] args) {  
        int age1 = 25;  
        int age2 = 43;  
  
        if(age1 < age2){  
            System.out.printf("La variable age1 (%d) est à une valeur  
inférieure à la variable age2  (%d) ", age1, age2);  
        }  
    }  
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Condition1  
La variable age1 (25) est à une valeur inférieure à la variable age2
```

Le mot clé **else** permet d'exécuter le second bloc d'instructions dans le cas où la condition du **if** n'est pas réalisée.

Fichier **Condition2.java**

```
public class Condition2 {  
  
    public static void main(String[] args) {  
        int nombre = 19;  
        if(nombre % 2 == 0){  
            System.out.printf("Le nombre (%d) est un nombre pair ",  
nombre);  
        }else{  
            System.out.printf("Le nombre (%d) est un nombre impair ",  
nombre);  
        }  
    }  
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Condition2  
Le nombre (19) est un nombre impair
```

En utilisant **else if**, il est possible d'enchaîner des tests conditionnels en associant un bloc d'instructions à chaque test. Dès qu'un test est vrai, le bloc d'instructions est exécuté. Les conditions suivantes seront

ignorées.

Fichier **Condition3.java**

```
public class Condition3 {  
  
    public static void main(String[] args) {  
        int angle = 85;  
        if(angle == 90) {  
            System.out.println("Angle droit");  
        }else if(angle == 0) {  
            System.out.println("Angle plat");  
        }else if(angle > 90) {  
            System.out.println("Angle obtus");  
        }else {  
            System.out.println("Angle aigu");  
        }  
    }  
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Condition3  
Angle aigu
```

dans le cas d'une succession de **else if**, le mot clé seul **else** sera toujours à la fin de cette succession.

switch ... case

Dans sa forme de base, le switch case ressemble au **if**. Il permet de tester plusieurs alternatives.

```
switch(variable) {  
    case constante1:  
        // bloc d'instructions  
    case constante2:  
        // bloc d'instructions  
    //...  
    default:  
        // bloc d'instructions  
}
```

Attention : dès qu'un case est vraie, le bloc d'instructions est évalué. Si ce bloc ne se termine pas par le mot clé **break**, les cases situées en dessous seront évaluées jusqu'au prochain **break** ou jusqu'à la fin du **switch**.

on ne peut pas avoir 2 cases avec la même constante

Depuis Java 12, le switch évolue afin d'intégrer le pattern matching.

for

Le mot clé **for** permet d'exécuter plusieurs fois un même bloc d'instructions.

Fichier **Looping1.java**

```
public class Looping1 {  
  
    public static void main(String[] args) {  
        for(int index=0; index<5; index++){  
            System.out.printf("Hello %d ! \n", index);  
        }  
    }  
}
```

Le bloc de code contenant l'instruction `System.out.printf("Hello %d ! \n", index);` est exécuté 5 fois (index pouvant varier de 0 à 4)

```
mvn --quiet compile exec:java -Dexec.mainClass=Looping1  
Hello 0 !  
Hello 1 !  
Hello 2 !  
Hello 3 !  
Hello 4 !
```

Si on décompose les parties de **for**

statement	description
int index=0	bloc de déclaration. La variable index est déclarée et sa valeur est initialisée à 0
index<5	bloc d'évaluation. A chaque itération, la condition est évaluée et tant qu'elle est vraie, le bloc d'instructions sera évalué
index++	bloc d'instruction. A chaque itération, les instructions dans cette partie sont évaluées. C'est ici que l'on trouve le plus souvent une incrementation. Ici, à chaque passage la valeur de la variable index s'incrémente de 1

on utilise la boucle **for** lorsque l'on peut déterminer à l'avance le nombre d'itérations.

L'opérateur ternaire ?

Il est possible d'évaluer sur une même ligne et d'assigner une valeur spécifique en fonction de la réalisation d'une condition. On utilise l'opérateur ternaire ?

Voici un exemple long avec le mot clé **if**

Fichier **Looping2.java**

```
public class Looping2 {  
  
    public static void main(String[] args) {  
        for(int nombre=0; nombre<5; nombre++){  
            if(nombre % 2 == 0){  
                System.out.printf("Le nombre %d est un nombre pair.\n",  
nombre);  
            }else{  
                System.out.printf("Le nombre %d est un nombre impair.\n",  
nombre);  
            }  
        }  
    }  
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Looping2  
Le nombre 0 est un nombre pair.  
Le nombre 1 est un nombre impair.  
Le nombre 2 est un nombre pair.  
Le nombre 3 est un nombre impair.  
Le nombre 4 est un nombre pair.
```

il est possible d'utiliser l'opérateur ternaire à la place du mot clé **if**

Fichier **Looping3.java**

```
public class Looping3 {  
  
    public static void main(String[] args) {  
        for(int nombre=0; nombre<5; nombre++){  
            var type = nombre % 2 == 0 ? "pair": "impair";  
            System.out.printf("Le nombre %d est un nombre %s.\n", nombre,  
type);  
        }  
    }  
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Looping3  
Le nombre 0 est un nombre pair.  
Le nombre 1 est un nombre impair.
```

```
Le nombre 2 est un nombre pair.  
Le nombre 3 est un nombre impair.  
Le nombre 4 est un nombre pair.
```

While

Le mot clé **while** permet d'itérer tant que la condition associée au **while** est vérifiée. while est souvent utilisé lorsque le nombre d'itérations ne peut être déterminé.

Fichier [Looping4.java](#)

```
public class Looping4 {  
  
    public static void main(String[] args) {  
        var total = 0;  
        var inc = new SecureRandom().nextInt(10);  
        while(total<20){  
            total += inc;  
        }  
        System.out.printf("Inc : %d, Total : %d", inc, total);  
    }  
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Looping4  
Inc : 6, Total : 24
```

il existe également l'instruction **do ...while()**. Le bloc d'instructions sera toujours évalué **au moins une fois**

Les méthodes

Une méthode est une fonction réalisation un traitement spécifique.

Une méthode :

- porte un nom
- prend 0 à n arguments
- peut retourner une valeur.
- possède des accolades. Les accolades vont introduire le bloc d'instructions qui sera évalué lors de l'appel de la méthode.
- est déclarée à l'intérieur d'une classe.

Il existe 2 types de méthodes :

- **les méthodes de classe** Ces méthodes, en fonction de leur portée, peuvent être appelé n'importe où dans le code en utilisant directement la classe.

- **les méthodes d'instance.** Ces méthodes manipulent les propriétés d'un objet et altèrent le comportement de ce dernier. Ces méthodes ne peuvent pas être appelées à partir du nom de la classe.

Depuis **Java 23**, il est possible de créer la méthode **main** en dehors d'une classe. [JEP - 477](#)

Squelette d'une méthode :

```
<scope> (static) <return type> methodName(<parameters>...) {  
    ....  
}
```

Les méthodes de classe

Le mot clé **static** permet de définir une méthode de classe.

Une méthode peut ne rien retourner. Dans ce cas, on utilisera le mot clé **void**

Fichier [Methode1.java](#)

```
public class Methode1 {  
  
    public static void displayHello(String name){  
        System.out.println("Hello " + name + " !");  
    }  
  
    public static void main(String[] args) {  
        displayHello("Bob");  
    }  
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Methode1  
Hello Bob !
```

Elle peut également retourner une donnée. Donc il faut indiquer dans la signature le type (primitif ou classe) que va retourner la méthode

Fichier [Methode2.java](#)

```
public class Methode2 {  
  
    public static int sum(int a, int b){  
        return a + b;  
    }  
}
```

```
public static void main(String[] args) {
    System.out.printf("Somme 10 + 7 : %d", sum(10, 17));
}
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Methode2
Somme 10 + 7 : 27
```

Depuis Java5, il est possible de déclarer un nombre infini de paramètres d'un même type. On parle alors de **varargs**. Un seul vararg est autorisé par méthode et il doit être le dernier paramètre.

Pour introduire un paramètre **vararg**, il suffit d'ajouter ... après le type du paramètre suivi du nom de celui-ci. Le paramètre sera considéré comme un tableau.

Fichier **Methode3.java**

```
public class Methode3 {

    public static int subtract(int... numbers){
        int ret = 0;
        for(var number: numbers){
            ret -= number;
        }
        return ret;
    }

    public static void main(String[] args) {
        System.out.printf("Soustraction : %d\n", subtract(1,3,4,5));
        System.out.printf("Soustraction : %d", subtract(9, 8));
    }
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Methode3
Soustraction : -13
Soustraction : -17
```

Les exemples ci-dessus utilisent la notion de vararg. En effet, la méthode **System.out.printf** est une méthode avec un vararg. Le premier paramètre est la chaîne de caractères, suivie de plusieurs paramètres qui seront utilisés pour remplacer les éléments introduits par %.

La classe System

La classe `System` fournit des méthodes pour interagir avec les entrées/sorties standards, récupérer les variables/propriétés d'environnements et quelques méthodes utilitaires. La classe ne peut être instanciée.

Sortie console : `System.out`

La propriété `System.out` permet de réaliser des sorties console. La propriété est de type **`PrintStream`** possédant des nombreux méthodes pour écrire dans le terminal.

Méthode	Description
<code>println()</code>	réalise une sortie console du paramètre en ajoutant un retour à la ligne
<code>print()</code>	réalise une sortie console du paramètre
<code>printf(str, ...args)</code>	réalise une sortie en console en utilisant une chaîne formatée. Les arguments seront intégrés à la chaîne

la propriété `System.err` ressemble à la propriété `System.out` mais réalise la sortie d'erreurs

Entrée standard : `System.in`

La propriété `System.in` permet de lire les éléments saisis en console. La propriété est de type **`InputStream`**.

La classe **`InputStream`** fournit des méthodes de base pour lire des octets. Pour faciliter la récupération des éléments saisis, nous utilisons la classe [`java.util.Scanner`](#)

```
import java.util.Scanner;

public class Main1 {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Saisissez un mot : ");
        var mot = scanner.next();
        System.out.println("Résultat : " + mot);

        System.out.print("Saisissez un nombre : ");
        var nb = scanner.nextInt();
        System.out.println("Résultat : " + nb);

        //pour forcer le scanner à lire jusqu'au retour charriot
        scanner.useDelimiter(System.getProperty("line.separator"));
        System.out.print("Saisissez une phrase : ");
        var phrase = scanner.next();
        System.out.println("Résultat : " + phrase);
    }
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Main1
Saisissez un mot : java
Résultat : java
Saisissez un nombre : 8
Résultat : 8
Saisissez une phrase : Java dans le top 3 des langages !
Résultat : Java dans le top 3 des langages !
```

Les tableaux

Un tableau est une structure de données permettant de stocker une liste d'une longueur finie d'éléments de même type.

Un tableau est un objet possédant la propriété **length**. Cette propriété length permet d'obtenir la taille du tableau

Les éléments d'un tableau sont accessibles à partir d'un index. Le premier élément est à l'index 0 (Zero based index origin)

Déclaration

La déclaration est identique à celle utilisé dans le langage **C**.

Syntaxe :

```
type[] name; // syntaxe préconisée `
type name[];
```

- type : type des éléments stockés dans le tableau
- Bracket [] symbole indiquant que les éléments vont être stockés dans un tableau

Fichier [Exemple1.java](#)

```
public class Exemple1 {
    public static void main(String[] args) {
        // Tableau d'entiers
        int[] arrayOfInt;
        // Tableau de float
        float distances[];
        // Tableau de double
        double[] values;
        // tableau de chaines de caractères
        String[] name;
    }
}
```

Instanciation

Pour instancier un tableau, il est nécessaire d'utiliser le mot clé **new** suivi du type et entre crochets, il faut préciser la taille du tableau.

Fichier [Exemple2.java](#)


```
public class Exemple2 {  
  
    public static void main(String[] args) {  
        // Tableau d'entiers  
        int[] arrayOfInt = new int[10];  
        // Tableau de float  
        float distances[] = new float[5];  
        // Initialisation directe avec valeur  
        double[] values = {1.0, 2, 3.0};  
        // tableau de chaines de caractères  
        String[] name;  
        name = new String[5];  
    }  
}
```

Lors de l'initiation, chaque cellule du tableau sera initialisé par la valeur par défaut associée au type.

- numerique (primitive) = 0
- boolean = false
- les objets = null

Initialisation des cellules

Pour accéder au cellule du tableau, on utilise les crochets en spécifiant l'index.

Dans le cas où on fait suite du symbole `=`, on va procéder à l'initialisation de la cellule par la valeur se trouvant à droite de ce symbole.

Dans le cas où il n'y a pas de symbole `=`, on récupère la valeur présent dans la cellule.

Fichier **Exemple3.java**

```
public class Exemple3 {  
    public static void main(String[] args) {  
        int[] arrayOfInt = new int[5];  
        arrayOfInt[0] = 10;  
        arrayOfInt[1] = 20;  
        arrayOfInt[3] = 40;  
        arrayOfInt[4] = 50;  
        arrayOfInt[2] = 30;  
        System.out.format("Valeur à l'index 3 : %d", arrayOfInt[3]);  
    }  
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Exemple3  
Valeur à l'index 3 : 40
```

Parcourir un tableau

for c-like

Le mot clé for en prenant 3 instructions permet de parcourir le tableau en exploitant la taille du tableau.

Fichier [Exemple4.java](#)

```
public class Exemple4 {  
  
    public static void main(String[] args) {  
        String[] languages = {"Java", "Python", "Rust", "C++"};  
        for(int index = 0; index < languages.length; index++){  
            System.out.printf("Langage à l'index %d : %s \n", index,  
languages[index]);  
        }  
    }  
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Exemple4  
Langage à l'index 0 : Java  
Langage à l'index 1 : Python  
Langage à l'index 2 : Rust  
Langage à l'index 3 : C++
```

for each

Depuis Java 5, il est possible de parcourir des éléments "itérables" avec le for.

La syntaxe est la suivante :

```
for(<type> variable : elementsIterable)
```

Fichier [Exemple5.java](#)

```
public class Exemple5 {  
  
    public static void main(String[] args) {  
        String[] languages = {"Java", "Python", "Rust", "C++"};  
        for(String language:languages){  
            System.out.printf("Langage : %s \n", language);  
        }  
    }  
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Exemple5
Langage : Java
Langage : Python
Langage : Rust
Langage : C++
```

La classe *java.util.Arrays*

La classe `java.util.Arrays` contient un ensemble de méthodes statiques pour manipuler des tableaux.

La méthode `toString`

Cette méthode permet d'afficher le contenu d'un tableau de manière lisible. Cette méthode va parcourir chaque élément et invoquer la méthode **`String.valueOf`** sur cet élément.

```
import java.util.Arrays;

public class Exemple6 {
    public static void main(String[] args) {
        Object[] items = {1, 2, "EPSI", Math.PI, 7.8, null};
        System.out.println(Arrays.toString(items));
    }
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Exemple6
[1, 2, EPSI, 3.141592653589793, 7.8, null]
```

Les méthodes `sort`

Les méthodes **`sort`** permettent de trier les éléments du tableau.

Attention : le tableau est directement modifié.

```
import java.util.Arrays;

public class Exemple7 {
    public static void main(String[] args) {
        double[] notes = {4, 18, 12.5, 20, 15};
        Arrays.sort(notes);
        System.out.println(Arrays.toString(notes));
    }
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Exemple7  
[4.0, 12.5, 15.0, 18.0, 20.0]
```

Il est également possible de trier les éléments en passant en 2 eme paramètre un implementation de l'interface **Comparator**.

```
import java.util.Arrays;  
  
public class Exemple8 {  
    public static void main(String[] args) {  
        String[] languages = {"Java", "Python", "Rust", "C++"};  
        Arrays.sort(languages, (a,b) -> a.length() - b.length());  
        //Comparator.comparing(String::length)  
        System.out.println(Arrays.toString(languages));  
    }  
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Exemple8  
[C++, Java, Rust, Python]
```

Les chaînes de caractères

Représentation

En java, la représentation une chaîne de caractères s'effectue par les guillemets.

2 manières d'utilisation :

- sur une ligne avec le guillemet simple
- sur plusieurs lignes avec l'utilisation de 3 guillemets.

Fichier [Exemple1.java](#)

```
public class Exemple1 {  
  
    public static void main(String[] args) {  
        var phrase = "une phrase sur une ligne";  
  
        var paragraphe = ""  
            Java a été conçu pour être facile à apprendre et à  
            utiliser efficacement pour les  
            programmeurs professionnels.  
            Il est basé sur C++, ce qui permet de tirer parti de vos  
            connaissances existantes,  
            mais il élimine de nombreuses sources de complexité et de  
            problèmes propres à C++.  
            "";  
    }  
}
```

Au niveau programmatique, une chaîne de caractères **n'est pas une primitive** mais **un objet de type `java.lang.String`**.

Fichier [Exemple2.java](#)

```
public class Exemple2 {  
  
    public static void main(String[] args) {  
        var phrase = "une phrase sur une ligne";  
  
        System.out.println("Type de la variable phrase : " +  
            phrase.getClass());  
    }  
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Exemple2
Type de la variable phrase : class java.lang.String
```

La classe String

La classe String possède de nombreuses méthodes permettant de manipuler la chaîne de caractères ([documentation](#)).

Méthode	Description
<code>int length()</code>	Retourne le nombre de caractères
<code>char charAt(int index)</code>	Retourne le caractère à la position « index »
<code>String substring(int beginIndex)</code>	Permet d'extraire une partie de la chaîne de caractères
<code>String substring(int beginIndex, int endIndex)</code>	Permet d'extraire une partie de la chaîne de caractères
<code>String concat(String str)</code>	Permet de concaténer une chaîne de caractère
<code>String replace(char oldChar, char newChar)</code>	Permet de remplacer un caractère dans la chaîne de caractères
<code>String replaceAll(String regex, String replacement)</code>	Permet de remplacer la partie de chaîne de caractères vérifiant l'expression régulière
<code>String trim()</code>	Permet de supprimer les espaces en début et fin de chaîne
<code>String toLowerCase()</code>	Permet de mettre une chaîne de caractères en minuscule
<code>String toUpperCase()</code>	Permet de mettre une chaîne de caractères en majuscule

Fichier [Exemple3.java](#)

```
public class Exemple3 {

    public static void main(String[] args) {
        String string = "Java c'est plutôt sympa";
        System.out.println(string.length());

        String[] words = string.split(" ");
        System.out.println(Arrays.toString(words));

        String replaceWord = string.replace("Java", "JavaScript");
        System.out.println(replaceWord);

        String extractWord = string.substring(18);
        System.out.println(extractWord);

        String upper = string.toUpperCase();
        System.out.println(upper);
    }
}
```

```
}  
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Exemple3  
23  
[Java, c'est, plutôt, sympa]  
JavaScript c'est plutôt sympa  
sympa  
JAVA C'EST PLUTÔT SYMPA
```

Manipulation des caractères

Une chaîne de caractères, bien que représentée en interne par un tableau de caractères, n'est pas un tableau de caractères.

Pour manipuler caractère par caractère, les méthodes **charAt**, **toCharArray**, **chars** permettent de manipuler caractère par caractère

Fichier [Exemple4.java](#)

```
public class Exemple4 {  
  
    public static void main(String[] args) {  
        var phrase = "parcourons les caractères";  
        for(int index=0; index<phrase.length(); index++){  
            System.out.printf("%c ", phrase.charAt(index));  
        }  
        System.out.println("");  
        var tableauChar = phrase.toCharArray();  
        for(var car: tableauChar){  
            System.out.printf("%c ", car);  
        }  
        System.out.println("");  
        phrase.chars().forEach(c -> System.out.printf("%c ", c));  
    }  
}
```

```
p a r c o u r o n s   l e s   c a r a c t è r e s  
p a r c o u r o n s   l e s   c a r a c t è r e s  
p a r c o u r o n s   l e s   c a r a c t è r e s
```

L'immutabilité

Une chaîne de caractères, une fois initialisée, ne peut plus être modifiée.

Ainsi, les propriétés internes de l'objet (de la chaîne de caractères) ne sont pas altérables. Les méthodes de ces types d'objet retournent le plus souvent une nouvelle objet de la même classe.

On parle alors d'**immuabilité**. C'est le cas de la classe **String**

Fichier **Exemple5.java**

```
public class Exemple5 {  
  
    public static void main(String[] args) {  
        String chaineEnMinuscules = "une chaine en minuscules."  
  
        String chaineEnMajuscules = chaineEnMinuscules.toUpperCase()  
  
        System.out.println("Variable chaineEnMinuscules : " +  
chaineEnMinuscules );  
        System.out.println("Variable chaineEnMajuscules : " +  
chaineEnMajuscules );  
    }  
}
```

```
Variable chaineEnMinuscules : une chaine en minuscules.  
Variable chaineEnMajuscules : UNE CHAINE EN MINUSCULES.
```

On constate qu'aucune modification n'a été apportée à la variable **chaineEnMinuscules**

Comparaison

En java, il n'est pas possible de vérifier l'égalité d'un objet avec **==**. Une comparaison de valeur avec **==** n'est possible uniquement avec les primitives.

Pour des objets, l'utilisation de **==** est possible, mais dans ce cas on ne compare pas l'état (et le type) des objets mais uniquement leur adresses en mémoires.

Une chaîne de caractères étant une instance de la classe **String** est un objet et donc si on souhaite vérifier l'égalité de 2 chaînes de caractères, il faut utiliser la méthode **equals**.

Fichier **Exemple6.java**

```
public class Exemple6 {  
  
    public static void main(String[] args) {  
        String chaine1= "2 chaines égales";  
        String chaine2= "2 chaines égales";  
  
        if(chaine1.equals(chaine2)){  
            System.out.println("Les 2 chaines sont égales");  
        }  
    }  
}
```



```
}  
}
```

Les 2 chaines sont égales

Attention, il se pourrait que vous constatiez que l'utilisation de l'égalité `==` fonctionne. Son fonctionnement repose uniquement sur le fait que Java utilise un mécanisme de cache afin d'optimiser la création des chaines de caractères. La modification du cache ou la manière dont est créée la chaine pourrait ne plus faire fonctionner l'égalité avec `==`

Concaténation

En java, la concaténation de chaines s'effectue directement avec le symbole `+`. Il est possible de concaténer une chaine de caractères avec n'importe quel type (primitif, objet).

Fichier [Exemple7.java](#)

```
public class Exemple7 {  
    public static void main(String[] args) {  
        int nb1 = 10;  
        int nb2 = 89;  
        String chaine = "La somme de " + nb1 + " et " + nb2 + "est égale à "  
        + (nb1+nb2);  
        System.out.println(chaine);  
    }  
}
```

La somme de 10 et 89est égale à 99

Performance

La concaténation en Java en utilisant le symbole `+` n'est pas **performante**. En effet, lors la concaténation, 1 objet intermédiaire est créé en memoire afin de réaliser la nouvelle chaine.

Pour optimiser le traitement, il est conseillé d'utiliser les classes :

- **java.util.StringBuilder** , le plus utilisé et le plus performant
- **java.util.StringBuffer** , plus lente que la première car elle peut être utilisée dans un contexte multi-threadé

Ces 2 classes possèdent :

- des méthodes **append** afin d'ajouter des éléments à ajouter à la chaine finale.
- la méthode **toString** permettant de générer la chaine de caractères

Fichier **Exemple8.java**

```
import java.time.Duration;

public class Exemple8 {

    static String[] items = {"Une", "chaine", "de", "caracteres"};

    static int MAX_ITERATION = 100000;

    public static void calculateDuration(String type, Runnable execute) {
        long start = System.nanoTime();
        execute.run();
        long stop = System.nanoTime();

        var duration =
Duration.ofNanos(stop).minus(Duration.ofNanos(start)).toString();
        System.out.println(type + " : " + duration);
    }

    public static void concateWithPlus() {
        String ret = "";
        for (int i = 0; i < MAX_ITERATION; i++) {
            for (var item : items) {
                ret += item + " ";
            }
        }
    }

    public static void concateWithStringBuilder() {
        StringBuilder builder = new StringBuilder();
        for (int i = 0; i < MAX_ITERATION; i++) {
            for (var item : items) {
                builder.append(item).append(" ");
            }
        }
        var ret = builder.toString();
    }

    public static void main(String[] args) {
        calculateDuration("StringBuilder : ",
Exemple8::concateWithStringBuilder);
        calculateDuration("Concaténation classique : ",
Exemple8::concateWithPlus);
    }
}
```

```
StringBuilder : : PT0.019722319S  
Concaténation classique : : PT41.799873016S
```

On constate que la concaténation avec `StringBuilder` est 2200 fois plus rapide que la concaténation traditionnelle.

Programmation Orientée Objet

La programmation Orientée Objet est un des concepts principaux du langage Java.

La réalisation de classes est la principale possibilité offerte par le langage pour définir de nouveaux types.

Qu'est ce qu'une classe ?

Une classe peut être assimilée à un plan logiciel ou à un prototype pour les objets. Elle définit les éléments qui donneront une existence à un objet.

On y trouvera :

- des propriétés
- des méthodes pour manipuler ces propriétés.

Les propriétés seront un ensemble de variables typées qui posséderont des valeurs variant tout le long du cycle de vie de l'objet. **Les valeurs des propriétés représentent l'état de l'objet**

Les méthodes permettent d'accéder aux propriétés, de modifier leur valeur ou de réaliser un comportement en manipulant l'ensemble des valeurs des propriétés.

La classe en tant que telle n'a pas d'existence réelle. Pour donner vie à une classe et l'attribuer à des variables, il faut créer une instance de celle-ci, c'est à dire créer un **objet**.

Plane
+String brand +String model +int capacity +double speed +double elevation
+takeoff() +landing() +accelerate(double speed) +deccelerate(double speed) +addPassagers(int number)

Définir une classe

Pour définir une classe,

- une classe est dans un fichier portant le même nom
- son nom commence par une majuscule et respecte le format pascalCase.

- elle est définie par le mot clé **class**
- elle possède des propriétés, des constructeurs et des méthodes.

```
public class AirPlane {  
  
    // 1 déclaration des attributs / propriété  
    private String brand;  
    private String model;  
    private int capacity;  
    private double speed;  
    private double elevation;  
  
    //Les méthodes  
    public void decelerate(double speedValue) {  
        if (this.speed - speedValue > 0) {  
            this.speed -= speedValue;  
        }  
    }  
  
    public void accelerate(double speedValue) {  
        this.speed += speedValue;  
    }  
}
```

Les variables

Élément « nommé » permettant de stocker des informations sur la classe ou sur l'instance de la classe (objet)

Le tableau suivant présente les types de variables que vous pouvez rencontrer dans une classe.

Variables locales	Variables d'instance	Variables de classe
<ul style="list-style-type: none">• Variables déclarées à l'intérieur des méthodes de la classe ou entre accolades {...}• Ces variables ne sont accessibles uniquement à l'intérieur des méthodes	<ul style="list-style-type: none">• Variables déclarées à l'intérieur d'une classe mais pas à l'intérieur d'une méthode de la classe.• La valeur des variables représente l'état de la classe à un instant t.• La valeur de ces variables est spécifique à l'instance de la classe	<ul style="list-style-type: none">• Variables déclarées à l'intérieur d'une classe avec le mot clé static• La valeur de ces variables sont partagées entre toutes les instances de la classe• On utilise ces variables pour représenter des constantes.• En fonction de la portée, il est possible d'accéder à ces variables avec la classe

Concernant les variables d'instances, chaque instance possédera « ses » propres valeurs pour chaque propriété

Pour créer une instance, il faut utiliser le mot clé **new**

Les constructeurs

Un constructeur est un bloc d'instructions permettant d'initialiser une nouvelle instance d'une classe.

Un constructeur :

- porte le **même nom que la classe**
- retourne aucun type.
- accède au mot clé **this** qui représente l'instance de la classe
- sert à configurer l'état initial de l'instance

Une classe peut ne pas avoir de constructeur. Dans ce cas, Java configure automatiquement un constructeur par défaut ne réalisant aucun traitement. Les propriétés de l'instance seront initialisées par les valeurs par défaut.

Dès qu'un constructeur est défini, le constructeur par défaut n'est plus accessible.

On peut définir autant de constructeurs que l'on souhaite. La différence se fera sur le nombre et le type des paramètres les définissant.

Pour notre classe **AirPlane**, le constructeur suivant initialisera la marque et le modèle. Quant aux autres propriétés (speed...), elles seront initialisées par les valeurs par défaut.

```
public AirPlane(String brand, String model) {  
    this.brand = brand;  
    this.model = model;  
}
```

le mot clé **this** permet d'accéder à l'instance de la classe c'est à dire à l'objet que nous sommes en train d'initialiser. Suivi du `.`, il permet d'accéder aux propriétés et aux méthodes de l'instance. le mot clé **this** sera disponible dès que l'on accède à des méthodes d'instance.

La création d'une instance se fera par l'utilisation du mot clé **new** suivi du constructeur souhaité.

L'exemple suivant permet de définir 2 variables : a310 et b777. Ces 2 variables sont du type **AirPlane**. Puis, nous initialisons les variables en appelant le constructeur prenant la marque et le modèle.

```
public class Main1 {  
  
    public static void main(String[] args) {  
        AirPlane a310 = new AirPlane("Airbus", "A310");  
        AirPlane b777 = new AirPlane("Boeing", "B777");  
    }  
}
```

Le mode debug de l'IDE permet de visualiser ces 2 instances avec leurs propriétés respectives.

Evaluate expression (🔗) or add a

```

🔗 args = {String[0]@762} []
▼ 📄 a310 = {AirPlane@763}
  > 🔗 brand = "Airbus"
  > 🔗 model = "A310"
    🔗 capacity = 0
    🔗 speed = 0.0
    🔗 elevation = 0.0
▼ 📄 b777 = {AirPlane@764}
  > 🔗 brand = "Boeing"
  > 🔗 model = "B777"
    🔗 capacity = 0
    🔗 speed = 0.0
    🔗 elevation = 0.0

```

Le tableau suivant synthétise les 3 types de constructeurs :

Constructeur par défaut

- En l'absence de constructeur, la classe possède, par défaut, un constructeur initialisant les variables d'instance à leur valeur par défaut
- Dans le cas où un constructeur est défini dans la classe, le constructeur par défaut ne sera plus disponible

Constructeur sans paramètre

- Constructeur définie dans la classe et ne prenant aucun paramètre

```

public AirPlane() {
    System.out.println("Initialisation")
    this.model = "inconnu";
}

```

Constructeur avec paramètres

- Constructeur dans la classe permettant et prenant des paramètres

```

public AirPlane(String brand, String
model) {
    this.brand = brand;
    this.model = model;
}

```

Il est tout à fait possible de déclarer autant de constructeurs que l'on souhaite. Il s'agit du principe de l'overloading.

En ajoutant un constructeur avec uniquement la marque, nous pouvons créer des instances à partir de ce dernier.

```

public AirPlane(String brand) {
    this.brand = brand;
}

```

```

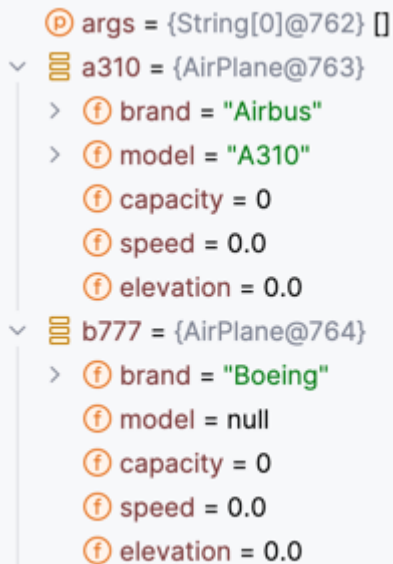
public class Main2 {

    public static void main(String[] args) {

```

```
AirPlane a310 = new AirPlane("Airbus", "A310");  
AirPlane b777 = new AirPlane("Boeing");  
  
}  
  
}
```

Le mode debug permet de visualiser que la variable *b777* ne possède que la marque.



```
args = {String[0]@762} []  
a310 = {AirPlane@763}  
  > brand = "Airbus"  
  > model = "A310"  
  capacity = 0  
  speed = 0.0  
  elevation = 0.0  
b777 = {AirPlane@764}  
  > brand = "Boeing"  
  model = null  
  capacity = 0  
  speed = 0.0  
  elevation = 0.0
```

Les méthodes

Les méthodes d'instance vont permettre d'accéder à notre objet afin qu'il réalise un traitement. Ce traitement pourra modifier l'état de notre objet via la modification de la valeur des propriétés d'instance.

L'accès à ces méthodes se fera en utilisant le point.

L'exemple présente 2 méthodes qui vont agir sur la vitesse de l'instance.

```
public void decelerate(double speedValue) {  
    if (this.speed - speedValue > 0) {  
        this.speed -= speedValue;  
    }  
}  
  
public void accelerate(double speedValue) {  
    this.speed += speedValue;  
}
```

```
public class Main3 {  
  
    public static void main(String[] args) {  
        AirPlane a310 = new AirPlane("Airbus", "A310");  
        a310.accelerate(200);  
        a310.decelerate(20);  
    }  
}
```



```

    }
}

```

Le tableau montre l'évolution de la vitesse de l'instance **a310**.

Etat 1	Etat 2	Etat 3
<pre> a310 = {AirPlane@763} > brand = "Airbus" > model = "A310" capacity = 0 speed = 0.0 elevation = 0.0 </pre>	<pre> args = {String[0]@761} [] a310 = {AirPlane@763} > brand = "Airbus" > model = "A310" capacity = 0 speed = 200.0 elevation = 0.0 </pre>	<pre> args = {String[0]@761} [] a310 = {AirPlane@763} > brand = "Airbus" > model = "A310" capacity = 0 speed = 180.0 elevation = 0.0 </pre>

Les accesseurs

La POO repose sur le principe d'encapsulation. Ce principe permet de masquer les détails d'un objet à un client. Afin d'accéder aux propriétés, des méthodes d'accès sont créées afin de retourner ou de modifier les propriétés de l'instance

Ainsi,

- Les propriétés sont **privée** le plus souvent. Lors de leur déclaration, les propriétés de la classe sont préfixés par **private**.
- Les propriétés privées ne sont accessible que les méthodes de l'instance
- L'accès et la modification des valeurs des propriétés seront le plus souvent réalisé par des méthodes appelées **getter** et **setter**.

En complétant la classe **AirPlane**, l'ajout d'un getter et d'un setter sur la propriété **model** permet de lui associer une valeur et d'y accéder.

```

public void setModel(String model) {
    this.model = model;
}

public int getCapacity() {
    return capacity;
}

```

```

public class Main4 {

    public static void main(String[] args) {
        AirPlane b777 = new AirPlane("Boeing");
        b777.setModel("B777");
    }
}

```

```
        System.out.println("L'instance b777 est associée au modele : " +
b777.getModel());
    }
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Main4
L'instance b777 est associée au modele : B777
```

Une erreur serait de mettre des setter dans toutes les classes. Hors, certaines propriétés ne doivent pas être modifiées sans contrôle. Par exemple, la propriété **speed** ne doit être modifiée directement vu qu'elle est contrôlée par les méthodes **accelerate** et **decelerate**.

On ne permettra que la récupération de la valeur de la propriété **speed**.

```
public double getSpeed() {
    return speed;
}
```

```
public class Main5 {

    public static void main(String[] args) {
        AirPlane a310 = new AirPlane("Airbus", "A310");
        a310.accelerate(200);
        a310.decelerate(20);
        System.out.printf("Avion %s %s a une vitesse de %.2f \n",
a310.getBrand(), a310.getModel(), a310.getSpeed());
    }
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Main5
Avion Airbus A310 a une vitesse de 180,00
```

L'héritage

L'héritage est la capacité d'une classe d'être créée à partir d'une autre classe en l'étendant. Ce principe permet d'éviter de dupliquer le code et de factoriser le code en commun dans une classe dite mère.

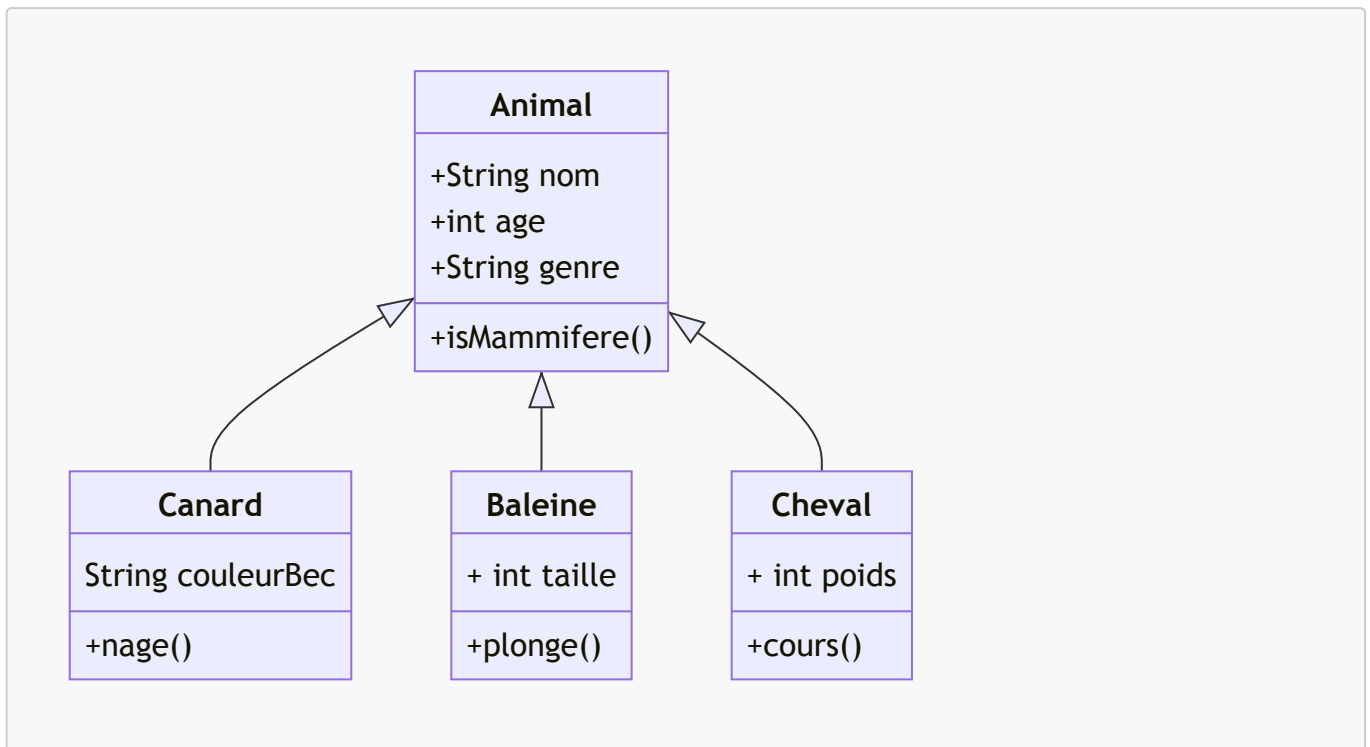
Les classes héritantes d'une autre classe héritent des comportements de la classe héritée.

Les classes héritante sont dites des classes filles et peuvent définir :

- de nouvelles propriétés
- de nouvelles méthodes.

Les classes fille peuvent également modifier le comportement de la classe mère en redefinissant les méthodes de la classe mère (**overriding**)

Exemple d'héritage :



Comment hériter ?

En Java, une classe peut hériter d'une autre classe en utilisant le mot clé **extends**. Le mot clé **extends** se place après le nom de classe et est suivi du nom de classe mère.

Attention, une classe **ne peut hériter que d'une seule classe !**

Dans le cas des animaux :

```
public class Animal {
    private String nom;
    private int age;
    private String genre;

    public boolean isMammifere(){
        return false;
    }
}

public class Baleine extends Animal{
    private int taille;

    public void plonge(){

    }
}
```

Ici, nous constatons que la classe **Baleine** représente un mammifère. La méthode **isMammifere** de la classe doit être redéfinie dans la classe **Baleine**.

Ainsi, la classe **Baleine** devient :

```
public class Baleine extends Animal{

    private int taille;

    public void plonge(){
    }

    @Override
    public boolean isMammifere() {
        return true;
    }
}
```

Ici, vous voyez apparaître l'annotation `@Override`. Les annotations sont énormément utilisées dans le cadre de développement d'applications d'entreprise. Elles permettent d'ajouter des comportements transverses sans que vous soyez obligés de coder. L'annotation `@Override` est purement indicative et n'a aucun apport lors de l'exécution.

Le mot clé super

Au sein des classes fille, nous pouvons redéfinir les méthodes. Cette redéfinition peut cependant s'appuyer/utiliser ce qui a été initialement défini dans la classe mère.

Le mot clé **super** permet d'appeler les méthodes de la classe mère au sein d'une classe fille,

Par exemple, ajoutons une méthode **getNom** dans la classe **Animal**. Cette méthode permettra de fournir le nom de l'animal.

Dans le cas de la classe **Baleine**, nous souhaitons que la méthode **getNom** retourne le nom de la baleine en préfixant pas *Baleine*.

En recodant, nous obtenons :

```
public abstract class Animal {

    private String nom;
    private int age;
    private String genre;

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

    public String getNom() {
```

```
        return nom;
    }
}

public class Baleine extends Mammifere{

    private int taille;

    public Baleine(String nom, int age) {
        super(nom, age);
    }

    @Override
    public String getNom() {
        return "Baleine " + super.getNom();
    }

    public void plonge(){
    }
}
```

```
import animal.Baleine;

public class Main10 {
    public static void main(String[] args) {
        var baleine = new Baleine("Moby Dick", 173);
        System.out.println(baleine.getNom());
    }
}
```

Baleine Moby Dick

Le mot clé **abstract**

Dans l'exemple ci dessus, aucune obligation n'a été imposé quant à la redéfinition de la méthode **isMammifere**. La classe **Balein** aurait très bien pu ne pas redéfinir cette méthode et d'un point de vue compilation et exécution, aucune erreur n'aurait été rencontrée.

Et pourtant, conceptuellement, une **baleine** est un mammifère.

Le moté clé **abstract** permet de forcer la redefinition des méthodes dites abstraites par les classes héritantes. Il doit être positionné avec le mot clé **class** de la classe mère puis peut être utilisé lors de la déclaration de certaines méthodes.

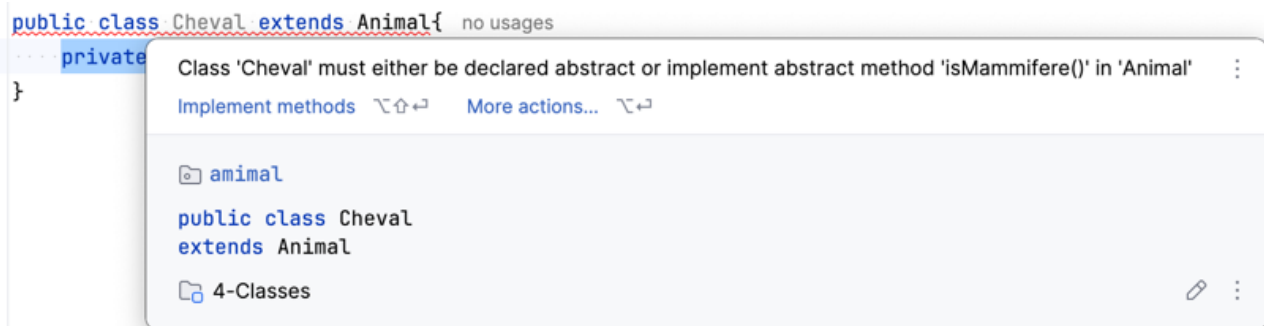
Ces méthodes abstraites ne fourniront aucun bloc de code dans la classe. Seule la siganture sera définie (ce que la méthode retourne et ce qu'elle prend en paramètre).

Les classes **filles** devront implémenter les méthodes abstraites. C'est à dire fournir un bloc d'instructions.

Voici la nouvelle déclaration de la classe **Animal** :

```
public abstract class Animal {  
  
    private String nom;  
    private int age;  
    private String genre;  
  
    public abstract boolean isMammifere();  
  
}
```

Si nous ne faisons aucune autre modification, le projet ne compile plus.



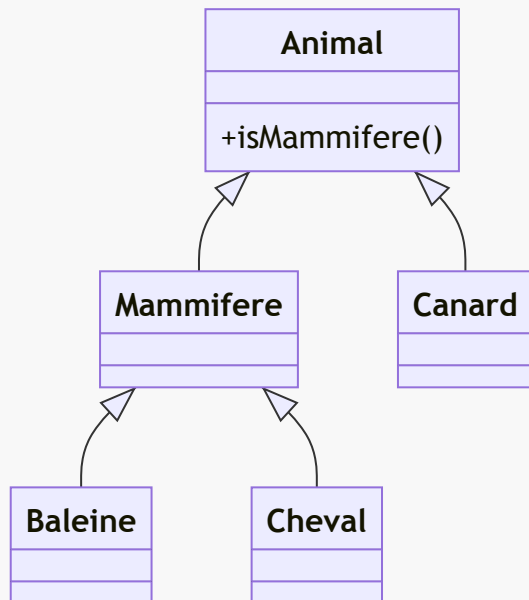
Il est nécessaire de définir une implémentation de cette méthode dans les classes **Cheval** et **Canard**

```
public class Cheval extends Animal{  
    private int poids;  
  
    @Override  
    public boolean isMammifere() {  
        return true;  
    }  
}  
  
public class Canard extends Animal {  
  
    private String couleurBec;  
  
    public void nage() {  
    }  
  
    @Override  
    public boolean isMammifere() {  
        return false;  
    }  
}
```

Le mot clé final.

La méthode **isMammifere** va devoir être implémentée dans l'ensemble des classes représentant des animaux. Dans notre exemple, les classes **Cheval** et **Baleine** représentent des animaux tous 2 des mammifères et potentiellement partageant des caractéristiques communes.

Ainsi, une nouvelle classe **Mammifere** peut être réalisée afin de regrouper les mammifères entre eux.



Le nouveau diagramme de classe présente le nouvel arbre d'héritage.

Dés lors, la méthode **isMammifere** sera définie dans la classe **Mammifere** et ne devra plus être redéfinie par les classes héritant de la classe **Mammifere**.

Pour éviter qu'une méthode ne soit redéfinie, il suffit de la faire précéder par le mot clé **final**. Le mécanisme d'**overriding** ne pourra plus avoir lieu.

La classe **Mammifere** devient :

```
public class Mammifere extends Animal{
    @Override
    public final boolean isMammifere() {
        return true;
    }
}
```

Si nous changeons l'héritage sur la classe **Baleine** en laissant la méthode **isMammifere**.

```
public class Baleine extends Mammifere{ no usages
```

```
... private int taille; no usages
```

```
... public void plonge(){ no usages  
... }
```

```
... @Override no usages
```

```
... public boolean isMammifere(){
```

```
... return true
```

```
... }
```

'isMammifere()' cannot override 'isMammifere()' in 'animal.Mammifere'; overridden method is final

Make 'Mammifere.isMammifere()' not final More actions...

© animal.Baleine

public boolean isMammifere()

Overrides: isMammifere in class Mammifere

4-Classes

La classe **Baleine** doit devenir :

```
public class Baleine extends Mammifere{  
  
    private int taille;  
  
    public void plonge(){  
    }  
  
}
```

La classe *java.lang.Object*

Toute classe étend directement ou indirectement de la classe *java.lang.Object* même si aucun héritage n'est indiqué.

Ainsi, les méthodes de la *java.lang.Object* sont disponibles.

Object

```
+Object clone()
+int hashCode()
+boolean equals(Object obj)
+String toString()
+void finalize()
+Class getClass()
+void notify()
+void notifyAll()
+void wait()
+void wait(long timeout)
+void wait(long timeout, int nanos)
```

Attardons nous sur les méthodes principales.

La méthode toString

Cette méthode permet de proposer une représentation de l'objet sous la forme d'une chaîne de caractères.

L'implémentation de la classe **Object** est très sommaire. Elle ne fournit que le nom de la classe complète de la valeur retournée par la méthode **hashCode**.

```
import animal.Baleine;

public class Main6 {
    public static void main(String[] args) {
        var baleine = new Baleine("Moby Dick", 173);
        System.out.println(baleine);
    }
}
```

```
animal.Baleine@506e6d5e
```

la méthode `System.out.println` invoque directement la méthode **toString** lorsqu'elle reçoit un objet en paramètre.

En redéfinissant la méthode **toString**, il serait plus intéressant d'afficher le nom et l'âge de la baleine.

Pour cela, redéfinissons la méthode **toString** dans la classe **Animal**.

```
package animal;
```

```
import java.util.StringJoiner;

public abstract class Animal {

    private String nom;
    private int age;
    private String genre;

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

    public abstract boolean isMammifere();

    @Override
    public String toString() {
        return new StringJoiner(", ", this.getClass().getSimpleName() + "
", " ")
            .add("age=" + age)
            .add("nom='" + nom + "'")
            .toString();
    }
}
```

En executant le programme précédent, l'affichage est désormais le suivant :

```
mvn --quiet compile exec:java -Dexec.mainClass=Main6
Baleine[age=173, nom='Moby Dick']
```

La méthode equals

Cette méthode est fréquemment utilisée pour vérifier que 2 objets sont égaux.

Par défaut, la méthode fournie pour la classe **java.lang.Object** vérifie uniquement que les 2 objets sont la même instance.

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

Si nous laissons l'implémentation, par défaut, le code suivant indique que les 2 baleines sont différents. Hors, elles semblent cependant être identiques.

:

```
public class Main7 {  
    public static void main(String[] args) {  
        var baleine1 = new Baleine("Moby Dick", 173);  
        var baleine2 = new Baleine("Moby Dick", 173);  
        System.out.printf("Les baleines sont elle les mêmes ? Reponse :  
%b", baleine1.equals(baleine2));  
    }  
}
```

Résultat :

```
Les baleines sont elle les mêmes ? Reponse : false
```

L'égalité entre 2 objets dans la réalité n'est pas uniquement basée sur le fait que ce soit la même instance. Il s'agit le plus souvent d'une comparaison entre les valeurs des propriétés caractérisant chaque instance.

Par exemple, dans la classe **Animal**, on pourrait considérer que 2 objets sont égaux :

- soit les objets sont la même instance
- soit ils sont issus de la même classe et ont le même nom et le même age.

Ainsi, la classe **Animal** doit redéfinir la méthode **equals**.

```
public abstract class Animal {  
  
    private String nom;  
    private int age;  
    private String genre;  
  
    public Animal(String nom, int age) {  
        this.nom = nom;  
        this.age = age;  
    }  
  
    public abstract boolean isMammifere();  
  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (o == null || getClass() != o.getClass()) return false;  
        Animal animal = (Animal) o;  
        return age == animal.age && Objects.equals(nom, animal.nom);  
    }  
  
    //Autres méthode  
}
```

En exécutant la méthode **main** précédente, on constate que l'égalité devrait être vraie.

```
Les baleines sont elle les mêmes ? Reponse : true
```

La méthode hashCode

Cette méthode est utilisée par des algorithmes de hachage avant de vérifier que 2 objets sont égaux.

Par exemple, la classe **HashSet** fréquemment utilisée lors de l'utilisation de **Set** utilise cette méthode avant de comparer les objets. Ainsi, des objets identiques doivent avoir en premier lieu le même résultat au niveau du Hashcode.

un set est une ensemble d'objets ne contenant pas de doublon.

Si nous conservons l'implémentation de la méthode **equals** mais que nous ne redéfinissons pas la méthode **hashCode**, nous constatons que le set contient **2 éléments**. Or, les 2 baleines étant les mêmes, un seul élément ne devrait être présent dans la set.

```
public class Main8 {
    public static void main(String[] args) {
        var baleine1 = new Baleine("Moby Dick", 173);
        var baleine2 = new Baleine("Moby Dick", 173);

        Set<Animal> set = new HashSet<>();
        set.add(baleine1);
        set.add(baleine2);

        System.out.printf("Nombre d'éléments : %d", set.size());
    }
}
```

```
Nombre d'éléments : 2
```

Il est donc nécessaire de redéfinir la méthode **hashCode**. L'implémentation de ce type méthode repose sur la réalisation d'une empreinte numérique en fonction des propriétés utilisées dans l'égalité. Pour la classe **Animal**, l'implémentation serait la suivante :

```
public abstract class Animal {

    private String nom;
    private int age;
    private String genre;

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

```
        this.age = age;
    }

    public abstract boolean isMammifere();

    // methode equals

    @Override
    public int hashCode() {
        return Objects.hash(nom, age);
    }

    // methode toString

}
```

Si nous relançons le main précédent, nous obtenons bien le résultat attendu :

```
Nombre d'éléments : 1
```

L'agrégation / La composition / L'association

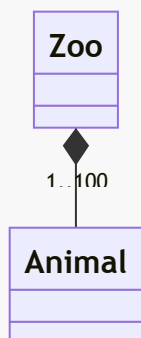
Comme décrit une classe permet de définir de nouveaux type, ces nouveaux types peuvent ainsi être utilisés pour déclarer des propriétés.

L'association est le fait qu'un objet soit lié à un objet d'une autre classe.

L'agrégation et la composition sont le fait qu'un objet soit lié à plusieurs objets d'une autre classe. La composition implique en plus une appartenance. La destruction de l'objet contenant détruit les objets liés.

Créons la classe **Zoo** qui va contenir un ensemble d'animaux. Cet ensemble est limité au maximum à 100 animaux.

Au niveau UML,



Au niveau de la classe **Zoo** :

```

public class Zoo {
    private Animal[] animaux;

    public Zoo() {
        this.animaux = new Animal[100];
    }

    public void addAnimal(Animal animal, int index){
        this.animaux[index] = animal;
    }

    @Override
    public String toString() {
        return new StringJoiner(", ", Zoo.class.getSimpleName() + "[",
            "]")
            .add("animaux=" + Arrays.stream(animaux)
                .filter(Objects::nonNull)
                .map(Object::toString)
                .collect(Collectors.joining(", ")))
            .toString();
    }
}

```

La méthode **addAnimal** permet d'ajouter des animaux au sein de la classe **Zoo**.

```

import animal.Baleine;
import animal.Canard;
import animal.Zoo;

public class Main9 {
    public static void main(String[] args) {
        Zoo zoo = new Zoo();
        zoo.addAnimal(new Baleine("Moby Dick", 173), 0);
        zoo.addAnimal(new Canard("Donald Duck", 90, "jaune"), 1);
        System.out.println(zoo);
    }
}

```

```

Zoo[animaux=Baleine[age=173, nom='Moby Dick'],Canard[age=90, nom='Donald
Duck']]

```

Vous pouvez constater que la méthode **addAnimal** prend 2 paramètres dont le premier est un paramètre de type **Animal**. Nous appelons cette méthode en passant une instance de la classe **Baleine** puis une instance de la classe **Canard**. Aucune erreur est levée ce qui est tout à fait normal, les classes **Baleine** et **Canard** héritent toutes 2 de la classe **Animal**. Donc, toute instance de ces classes sont, par héritage, des instances de la classe **Animal**. On parle ici de polymorphisme.

Les interfaces

Introduction

Une interface est un prototype de classe définissant des méthodes devant être implémentées par les classes réalisant ce prototype.

En java, une interface est introduite par le mot clé **interface**.

Une interface peut contenir :

- Des signatures de méthodes
- Des variables avec *public static final*
- Des méthodes statiques

Une classe peut implémenter plusieurs interfaces via le mot clé **implements**.

Une interface peut étendre plusieurs interfaces via le mot clés **extends**

Declaration

Fichier [AutoRoute](#)

```
public interface AutoRoute {  
  
    /**  
     * retourne la categorie afin de calculer le prix du péage  
     *  
     * @return  
     */  
    int getCategorie();  
}
```

Fichier [PoidsLourd](#)

```
public class PoidsLourd extends Vehicule  
    implements AutoRoute {  
  
    public PoidsLourd(String modele, int nbEssieux) {  
        super(modele, nbEssieux);  
    }  
  
    @Override  
    public int getCategorie() {  
        return 2;  
    }  
}
```

Fichier `Voiture`

```
public class Voiture extends Vehicule
    implements AutoRoute {

    public Voiture( String modele, int nbEssieux) {
        super(modele, nbEssieux);
    }
    @Override
    public int getCategorie() {
        return 1;
    }
}
```

Ainsi, les classes **Voiture** et **PoidsLourd** implémentent l'interface **AutoRoute**. On a l'obligation de définir la méthode **getCategorie()**.

Les méthodes par défaut

Depuis Java 8, on peut avoir des méthodes **avec une implémentation** exploitant les méthodes de l'interface.

La méthode doit être préfixée par le mot clé **default**.

Si nous reprenons l'interface **AutoRoute**, nous pouvons ajouter la méthode **calculerPrixPeage** qui sera une méthode par défaut et utilisera la catégorie du vehicule afin de calculer le prix.

```
public interface AutoRoute {

    /**
     * retourne la categorie afin de calculer le prix du péage
     *
     * @return
     */
    int getCategorie();

    /**
     * calcule le prix du péage.
     *
     * @return prix du péage
     */
    default double calculerPrixPeage(){
        return getCategorie() * 15;
    }
}
```

Les classes **PoidsLourd** et **Voiture** ne sont pas obligées de fournir une implémentation à la méthode **calculerPrixPeage**

Ainsi, il est désormais possible d'ajouter de nouvelles méthodes sur des interfaces sans que cela n'impacte la compilation des applications. Ce nouveau concept a ainsi permis d'intégrer la philosophie des streams au niveau des collections tout en conservant la rétro compatibilité des classes existantes liées aux collections.

Les collections

Qu'est ce qu'une collection ?

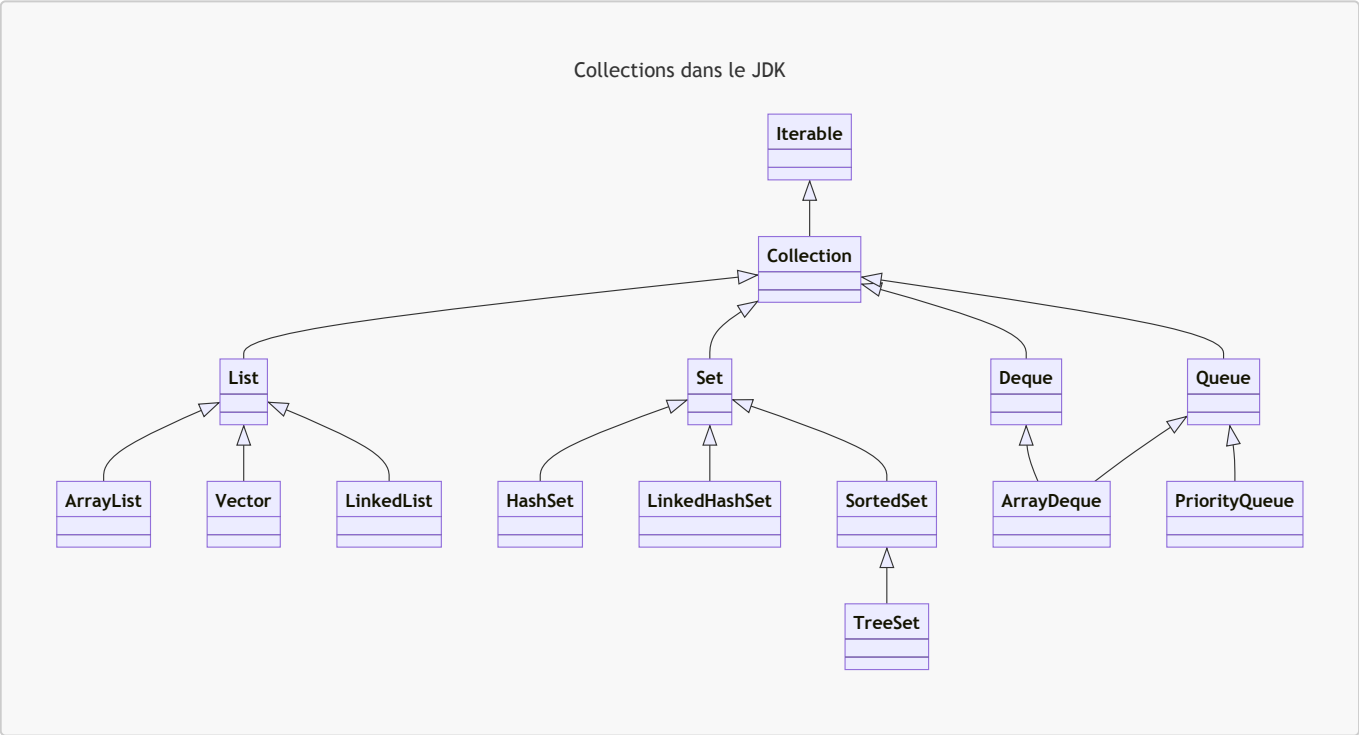
Une collection est une structure de données permettant de regrouper un ensemble d'objets.

Une collection peut etre comparée à un tableau avec des avantages indéniables :

- aucune contrainte de taille. La liste s'adapte automatiquement en cachant la complexité sous jacente.
- de nombreuses méthodes permettant de manipuler les éléments de la collection.
 - Ajout, suppression, recherche
 - Parcours
 - Tri

Les classes permettant de créer des collections sont essentiellement dans le package *java.util*.

Le JDK fournit diverses implémentations répondant à des besoins différents et adaptés à des contextes d'utilisation.



Les grandes familles de collections

List	Set	Queue
<i>java.util.List</i>	<i>java.util.Set</i>	<i>java.util.Queue</i>
Liste ordonnée d'objets	Liste non ordonnée d'objets	Liste ordonnée d'objets type FIFO
Possibilité d'avoir des doublons	Aucun doublon	Possibilité d'avoir desdoublons

List	Set	Queue
Possibilité de placer un élément à un index précis de la liste	Pas possibilité de placer un élément à l'endroit souhaité	Tout nouvel élément est placé à la fin de la liste
Accès possible à un élément par son index		Accès uniquement au premier élément

L'interface List

3 implémentations principalement utilisées :

- ArrayList
 - Bonne performance en accès get / set
 - **Classe la plus utilisée**
- LinkedList
 - Bonne performance en accès add / remove
 - Performance médiocre en accès get / set
- Vector
 - Toutes les méthodes sont synchronisées
 - Performance médiocre
 - **Classe à ne plus utiliser**

Quelques méthodes fréquemment utilisées :

Methode	Description
<code>boolean add(E e)</code>	Permet d'ajouter un élément
<code>void add(int index, E element)</code>	Permet d'ajouter un élément à l'index
<code>E get(int index)</code>	Permet de récupérer un élément dans la liste à partir de son index
<code>boolean remove(Object o)</code>	Permet de supprimer un élément.
<code>int size()</code>	Retourne la taille de la liste
<code>boolean isEmpty()</code>	Indique si la liste est vide
<code>void sort(Comparator<? super E> c)</code>	Permet de trier la liste en se basant sur unComparator

Fichier [Main1](#)

```
import java.util.ArrayList;
import java.util.List;
```

```
public class Main1 {  
  
    public static void main(String[] args) {  
        List<String> list = new ArrayList<>();  
        list.add("pomme");  
        list.add("melon");  
        list.add("orange");  
        list.add("cerise");  
        list.add("fraise");  
        String pomme = list.get(0);  
        System.out.println(pomme);  
        System.out.printf("Taille : %d\n", list.size());  
        String orange = list.remove(2);  
        System.out.println(orange);  
        System.out.printf("Taille : %d", list.size());  
    }  
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Main1  
pomme  
Taille : 5  
orange  
Taille : 4
```

L'interface **List** possède des méthodes statiques *List.of* permettant facilement d'initialiser des listes d'éléments. Attention, la liste créée est immutable. C'est à dire que vous ne pouvez ni ajouter, ni supprimer des éléments.

L'exemple suivant exploite la méthode *List.of* afin d'initialiser une liste de fruits directement remplie.

```
import java.util.List;  
  
public class Main2 {  
    public static void main(String[] args) {  
        List<String> list = List.of("pomme", "melon", "orange", "cerise",  
        "fraise");  
        System.out.println(list);  
    }  
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Main2  
[pomme, melon, orange, cerise, fraise]
```

L'interface Set

Un set ressemble fortement à une liste. Cependant, les sets n'ont pas de doublons.

- HashSet
 - Classe les éléments à partir de la valeur du hashCode.
 - Seulement 1 élément « null »
- LinkedHashSet
 - Comme le HashSet mais en conservant l'ordre d'insertion
- TreeSet
 - Classe les éléments en réalisant un tri avec un ordre ascendant
 - Pour optimiser le tri
 - soit les éléments implémentent Comparable
 - soit un comparateur est passé au constructeur de la classe TreeSet

Pour supprimer les doublons, Java va tester l'égalité des objets en utilisant la méthode **equals**. Il est donc important de correctement définir l'égalité des objets d'une même classe.

Le test d'égalité sur chaque objet du set pouvant être couteux en mémoire, les implémentations de l'interface Set utilisent la méthode hashCode pour calculer l'empreinte avant de comparer l'égalité.

Ainsi, dès que vous insérerez un objet dans le set, lors de l'insertion, la méthode hashCode est appelée afin d'obtenir une empreinte numérique. A partir de cette empreinte, l'implémentation va rechercher la présence d'un objet ayant la même empreinte, si c'est le cas, il y aura un appel à la méthode equals. Si la méthode retourne vraie, l'objet inséré remplacera l'objet présent. Si non, l'objet sera ajouté à la liste.

Fichier [Main3](#)

```
import java.util.HashSet;
import java.util.Set;

public class Main3 {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        set.add("pomme");
        set.add("pomme");
        set.add("melon");
        set.add("orange");
        set.add("cerise");
        set.add("fraise");
        System.out.println(set.isEmpty());
        System.out.printf("Taille : %d\n", set.size());
    }
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Main3
false
Taille : 5
```

L'interface **Set** possède également des méthodes statiques `Set.of` permettant facilement d'initialiser un set d'éléments.

```
import java.util.List;
import java.util.Set;

public class Main4 {
    public static void main(String[] args) {
        Set<String> set = Set.of("pomme", "melon", "orange", "cerise",
"fraise");
        System.out.println(set);
    }
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Main4
[fraise, melon, orange, cerise, pomme]
```

L'interface Queue

- `PriorityQueue`
 - Permet de récupérer les éléments triés après leur insertion
- `ConcurrentLinkedQueue`
 - Classe thread-safe
 - Adapté pour des accès dans un environnement multi-threadé
- `ArrayBlockingQueue`
 - Permet de stocker des éléments avec une taille finie
 - Éléments stockées en mode FIFO

Méthode	Description
<code>boolean add(E e)</code>	Permet d'ajouter un élément mais lève une exception en cas de rejet de l'ajout
<code>boolean offer(E e)</code>	Permet d'ajouter un élément mais ne lèvera pas d'exception en cas de rejet de l'ajout
<code>E remove()</code>	Permet de supprimer un élément de la queue. Lève une exception en cas de problème

Méthode	Description
<code>E poll()</code>	Permet de supprimer un élément de la queue. Ne lève pas d'exception en cas de problème
<code>E element()</code>	Permet de récupérer un élément sans le retirer de la queue. Lève une exception en cas de problème
<code>E peek()</code>	Permet de récupérer un élément sans le retirer de la queue. Ne lève pas d'exception en cas de problème

Parcourir une collection

Utilisation du for

Depuis Java 5, le parcours des éléments itérables a été facilité par l'utilisation du for ("for each").

Un tableau est également considéré comme un élément itérable. La syntaxe ci-dessous est également utilisable avec les tableaux.

Syntaxe :

```
for(<Type> <variable> : <iterable>){  
    // instructions  
}
```

```
import java.util.ArrayList;  
  
public class Main5 {  
  
    public record Country(String name, String capital){};  
  
    public static void main(String[] args) {  
        var countries = new ArrayList<Country>();  
        countries.add(new Country("France", "Paris"));  
        countries.add(new Country("Allemagne", "Berlin"));  
        countries.add(new Country("Anglaterre", "Londres"));  
        countries.add(new Country("Belgique", "Bruxelles"));  
  
        for(var country:countries){  
            System.out.println(country.name.toUpperCase() + " a pour  
capitale "+ country.capital);  
        }  
    }  
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Main5  
FRANCE a pour capitale Paris
```

```
ALLEMAGNE a pour capitale Berlin
ANGLATERRE a pour capitale Londres
BELGIQUE a pour capitale Bruxelles
```

Les streams et foreach

Depuis Java 8, la programmation fonctionnelle a fait son apparition au sein du langage Java.

La programmation fonctionnelle a pour objectif d'appliquer des transformations (via l'utilisation des fonctions) sur des éléments afin d'obtenir d'autres éléments.

Les streams et les lambdas ont ainsi été intégrés dans le JDK pour mettre en place ce concept.

Ainsi, les collections possèdent une méthode *stream* permettant d'initialiser la stream de la collection.

Sur la stream obtenue, la méthode **foreach** est disponible et sera appelée sur chaque élément constituant la collection.

```
import java.util.ArrayList;

public class Main6 {
    public record Country(String name, String capital){};

    public static void main(String[] args) {
        var countries = new ArrayList<Main5.Country>();
        countries.add(new Main5.Country("France", "Paris"));
        countries.add(new Main5.Country("Allemagne", "Berlin"));
        countries.add(new Main5.Country("Anglaterre", "Londres"));
        countries.add(new Main5.Country("Belgique", "Bruxelles"));

        countries.stream()
            .forEach(country ->
                System.out.println(country.name().toUpperCase() + " a pour capitale " +
                    country.capital()));
    }
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Main6
FRANCE a pour capitale Paris
ALLEMAGNE a pour capitale Berlin
ANGLATERRE a pour capitale Londres
BELGIQUE a pour capitale Bruxelles
```


Les streams & Les lambdas

Depuis Java 8, la programmation fonctionnelle a fait son apparition au sein du langage Java.

La programmation fonctionnelle a pour objectif d'appliquer des transformations (via l'utilisation des fonctions) sur des éléments afin d'obtenir d'autres éléments.

Les streams et les lambdas ont ainsi été intégrées dans le JDK pour mettre en place ce concept.

Les lambdas

Depuis Java 8, les lambdas sont la fonctionnalité la plus importante depuis Java 5

- Evite de faire des classes anonymes
- Permet de réaliser de la programmation fonctionnelle
- Evite d'écrire du code inutile.

Syntaxe :

```
(parameter) -> //unique instruction

(parameter) -> {
    //instructions
}
```

Les instructions d'une lambda doivent être très courtes. Le plus souvent, une seule ligne d'instruction constitue le corps de la lambda.

Les accolades ne sont pas obligatoires. Dans ce cas, une seule ligne d'instruction, ne se terminant pas par un point-virgule et dont le résultat de l'instruction est considéré comme la valeur de retour

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.StringJoiner;
import java.util.stream.Collectors;

public class Main1 {

    static class Personne{
        String nom;
        LocalDate dateNaissance;

        public Personne(String nom, LocalDate dateNaissance) {
            this.nom = nom;
            this.dateNaissance = dateNaissance;
        }
    }
}
```

```

    }

    @Override
    public String toString() {
        return nom + " - " + dateNaissance;
    }

    public LocalDate getDateNaissance() {
        return dateNaissance;
    }

    public String getNom() {
        return nom;
    }
}

public static void main(String[] args) {

    Personne donald = new Personne("donald", LocalDate.parse("1934-01-01"));
    Personne mickey = new Personne("mickey", LocalDate.parse("1928-01-01"));
    Personne dingo = new Personne("dingo", LocalDate.parse("1932-05-25"));
    List<Personne> personnages = Arrays.asList(donald, mickey, dingo);

    Comparator<Personne> compare = (p1, p2) ->
p1.dateNaissance.compareTo(p2.dateNaissance);
    personnages.sort(compare);

    System.out.println(personnages);
}
}

```

Les méthodes d'inférence

Il est fréquent qu'une lambda n'est simplement que l'application d'une transformation, d'une comparaison sur une propriété. Il est possible de passer uniquement que la fonction. Celle-ci sera appliquée sur tous les éléments.

Pour ne passer qu'une méthode, on indique uniquement où elle se trouve en utilisant `::` au lieu du point.

```

import java.time.LocalDate;
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

public class Main2 {
    public static void main(String[] args) {
        Main1.Personne donald = new Main1.Personne("donald",

```

```
LocalDate.parse("1934-01-01"));
    Main1.Personne mickey = new Main1.Personne("mickey",
LocalDate.parse("1928-01-01"));
    Main1.Personne dingo = new Main1.Personne("dingo",
LocalDate.parse("1932-05-25"));
    List<Main1.Personne> personnages = Arrays.asList(donald, mickey,
dingo);

    Comparator<Main1.Personne> compare =
Comparator.comparing(Main1.Personne::getDateNaissance);

    personnages.sort(compare);

    System.out.println(personnages);
}
}
```

La méthode static **Comparator.comparing** prend en paramètre une méthode dont le but est de fournir une valeur pour la comparer.

```
public static <T, U extends Comparable<? super U>> Comparator<T>
comparing(
    Function<? super T, ? extends U> keyExtractor)
```

L'interface `Function` est une interface fonctionnelle prenant un élément en entrée et fournissant une valeur en retour.

Ainsi, nous passons la méthode **Main1.Personne::getDateNaissance**.

A l'exécution, la JVM va :

- invoquer la méthode `getDateNaissance` sur les éléments de la liste afin de récupérer la date de naissance
- comparer des couples de date de naissance afin de classer les objets.

Les streams

Un stream ;

- Permet de mieux interagir avec des collections de données
- Permet de chaîner des transformations sur une collection
- S'appuie sur un nouveau concept : `lambda`
- Support la programmation fonctionnelle
 - Transformer des objets
 - Éviter les effets de bord

Un stream c'est 3 éléments :

- Une source de données (la collection ...)
- Des transformations
- Une opération terminale



```
import java.util.ArrayList;

public class Main6 {
    public record Country(String name, String capital){};

    public static void main(String[] args) {
        var countries = new ArrayList<Main5.Country>();
        countries.add(new Main5.Country("France", "Paris"));
        countries.add(new Main5.Country("Allemagne", "Berlin"));
        countries.add(new Main5.Country("Anglaterre", "Londres"));
        countries.add(new Main5.Country("Belgique", "Bruxelles"));

        countries.stream()
            .forEach(country ->
System.out.println(country.name().toUpperCase() + " a pour capitale " +
country.capital()));
    }
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Main6
FRANCE a pour capitale Paris
ALLEMAGNE a pour capitale Berlin
ANGLATERRE a pour capitale Londres
BELGIQUE a pour capitale Bruxelles
```

Une transformation applique la transformation sur chaque élément de la stream et retourne une autre stream

Transformation	Description
filter	Permet de filtrer les données
map	
mapToInt	
mapToLong	Permet de réaliser une transformation de la données
mapToDouble	

Transformation	Description
reduce	Réaliser une réduction sur les éléments de la streams
distinct	Supprime tous les éléments en double
skip	Permet de sauter un certain nombre d'éléments
limit	Permet de réduire le nombre d'éléments de la Stream
sorted	Permet de trier les éléments

Par exemple, reprenons l'exemple précédent, et créons la chaîne de caractères avant la sortie console

La transformation avec map

```
import java.util.ArrayList;

public class Main7 {
    public record Country(String name, String capital){};

    public static void main(String[] args) {
        var countries = new ArrayList<Main5.Country>();
        countries.add(new Main5.Country("France", "Paris"));
        countries.add(new Main5.Country("Allemagne", "Berlin"));
        countries.add(new Main5.Country("Anglaterre", "Londres"));
        countries.add(new Main5.Country("Belgique", "Bruxelles"));

        countries.stream()
            .map(country -> country.name().toUpperCase() + " a pour
capitale " + country.capital())
            .forEach(str -> System.out.println(str));
    }
}
```

Le filtre

Le filtre introduit par **filter** permet d'extraire les éléments dont le filtre retourne **true**.

```
import java.util.ArrayList;

public class Main8 {
    public record Country(String name, String capital){};

    public static void main(String[] args) {
        var countries = new ArrayList<Main5.Country>();
        countries.add(new Main5.Country("France", "Paris"));
        countries.add(new Main5.Country("Allemagne", "Berlin"));
        countries.add(new Main5.Country("Anglaterre", "Londres"));
        countries.add(new Main5.Country("Belgique", "Bruxelles"));
    }
}
```

```

        countries.stream()
            .filter(country -> country.name().startsWith("A"))
            .map(country -> country.name().toUpperCase() + " a pour
capitale " + country.capital())
            .forEach(str -> System.out.println(str));
    }
}

```

Le tri

Le tri est réalisé par la méthode **sorted**. En ne passant de paramètres, les éléments seront triés si ils sont comparable.

Il est également possible de passer une instance de la classe `Comparator` afin de permettre de comparer les éléments entre eux.

```

import java.util.ArrayList;

public class Main9 {
    public record Country(String name, String capital){};

    public static void main(String[] args) {
        var countries = new ArrayList<Main5.Country>();
        countries.add(new Main5.Country("France", "Paris"));
        countries.add(new Main5.Country("Allemagne", "Berlin"));
        countries.add(new Main5.Country("Anglaterre", "Londres"));
        countries.add(new Main5.Country("Belgique", "Bruxelles"));

        countries.stream()
            .sorted((c1, c2) ->
c1.name().compareToIgnoreCase(c2.name()))
            .map(country -> country.name().toUpperCase() + " a pour
capitale " + country.capital())
            .forEach(str -> System.out.println(str));
    }
}

```

Les terminaisons

Les terminaisons ne retournent pas de streams.

Le chainage d'altération est interrompu

Methode	Description
<code>findFirst</code>	Retourne le 1er éléments de la stream
<code>allMatch</code>	Retourne true si tous les éléments de la stream valide le predicat

Methode	Description
noneMatch	Retourne true si au moins un des éléments de la stream valide le predicat
anyMatch	Retourne true si aucun éléments de la stream valide le predicat
count	Retourne le nombre d'éléments de la stream
max	Retourne l'élément maximal de la stream (utilisation de comparateur)
min	Retourne l'élément minimal de la stream (utilisation de comparateur)
toArray	Convertit les éléments de la stream en un tableau
collect	Permet de collecter les éléments de la stream et de les récupérer sous la format d'une liste, d'un set...

```
import java.util.ArrayList;

public class Main10 {
    public record Country(String name, String capital){};

    public static void main(String[] args) {
        var countries = new ArrayList<Main10.Country>();
        countries.add(new Main10.Country("France", "Paris"));
        countries.add(new Main10.Country("Allemagne", "Berlin"));
        countries.add(new Main10.Country("Anglaterre", "Londres"));
        countries.add(new Main10.Country("Belgique", "Bruxelles"));

        var firstItem = countries.stream()
            .sorted((c1, c2) ->
                c1.name().compareToIgnoreCase(c2.name()))
            .findFirst();

        if(firstItem.isPresent()){
            var country = firstItem.get();
            System.out.println("Pays : " + country.name + ", Capitale : "
+ country.capital());
        }
    }
}
```

Les enum

Les énumérations sont un nouveau type apparu depuis Java 5. Elles permettent de représenter une liste finie d'éléments.

Les caractéristiques

Les énum peuvent être déclarées à l'intérieur ou à l'extérieur des classes. Elle est définie par le mot clé **enum**.

Une énumération contient une liste de valeurs figées lors de la compilation. Chaque valeur représente une instance de l'énum.

Une énumération ne peut être altérée lors de l'exécution. Elle est implicitement **static** et **final**.

Resemblant à une classe, les constructeurs présents à l'intérieur de l'énumération ne sont pas accessibles depuis l'extérieur. Uniquement les valeurs définies dans l'énumération pourront accéder aux divers constructeurs.

Une énumération est **immutable**

Déclaration

Fichier [Saison](#)

```
public enum Saison {  
    PRINTEMPS,    // représente les différentes valeurs de l'énum Saison  
    ETE,  
    AUTOMNE,  
    HIVER  
}
```

Les énumérations possèdent la méthode **values()** permettant de récupérer un tableau des valeurs constituant l'énumération.

Les valeurs de l'énumération sont des constantes pointant chacune vers un emplacement mémoire spécifique qui ne changera pas pendant l'exécution. Ainsi, on peut comparer les énumérations directement avec le comparateur **==**

Fichier [Exemple1](#)

```
public class Exemple1 {  
    public static void main(String[] args) {  
        var saisons = Saison.values();  
        for (Saison s : saisons) {  
            System.out.println(s);  
        }  
    }  
}
```



```
        var saison = Saison.AUTOMNE; //resolution pqr une valeur de
l'énumération

        var automne = Saison.valueOf("AUTOMNE"); // résolution par le nom

        if(saison == Saison.AUTOMNE) {
            System.out.println("C'est bien l'automne");
        }
    }
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Exemple1
PRINTEMPS
ETE
AUTOMNE
HIVER
C'est bien l'automne
```

Personnalisation d'une enum

Une énumération peut avoir des propriétés

L'initialisation des propriétés est réalisée via un constructeur

Afin de respecter le principe de l'énum, ces propriétés :

- ne doivent pas être modifiable
- Accessible qu'en lecture

Un constructeur n'est utilisable qu'à l'intérieur de l'enum.

Fichier [Exemple2](#)

```
public class Exemple2{

    enum ColorPanel {
        RED("#FF0000"), GREEN("#FF0000"),
        LIGHT_GREEN("#90EE90"), BLUE("#0000FF");
        private final String hexaCode;

        ColorPanel(String hexaCode) {
            this.hexaCode = hexaCode;
        }
        /**
         * @return the hexaCode
         */
        public String getHexaCode() {
            return hexaCode;
        }
    }
}
```

```
    }

    @Override
    public String toString() {
        return super.toString() + "["+ hexaCode + "]";
    }
}

public static void main(String[] args) {
    ColorPanel c = ColorPanel.BLUE;
    System.out.println(c.getHexaCode());
    System.out.println(c);
}

}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Exemple2
#0000FF
BLUE[#0000FF]
```

Utilisation du switch case

Les énumérations peuvent être utilisées avec le classique swith case.

Fichier [Exemple3](#)

```
public class Exemple3 {

    public static void main(String[] args) {
        var saison = Saison.ETE;

        switch (saison) {
            case AUTOMNE:
                System.out.println("C'est l'automne");
                break;
            case HIVER:
                System.out.println("C'est hivers");
                break;
            case PRINTEMPS:
                System.out.println("C'est le printemps");
                break;
            case ETE:
                System.out.println("C'est l'été");
                break;
        }
    }
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Exemple3  
C'est l'été
```

Les Exceptions

Introduction

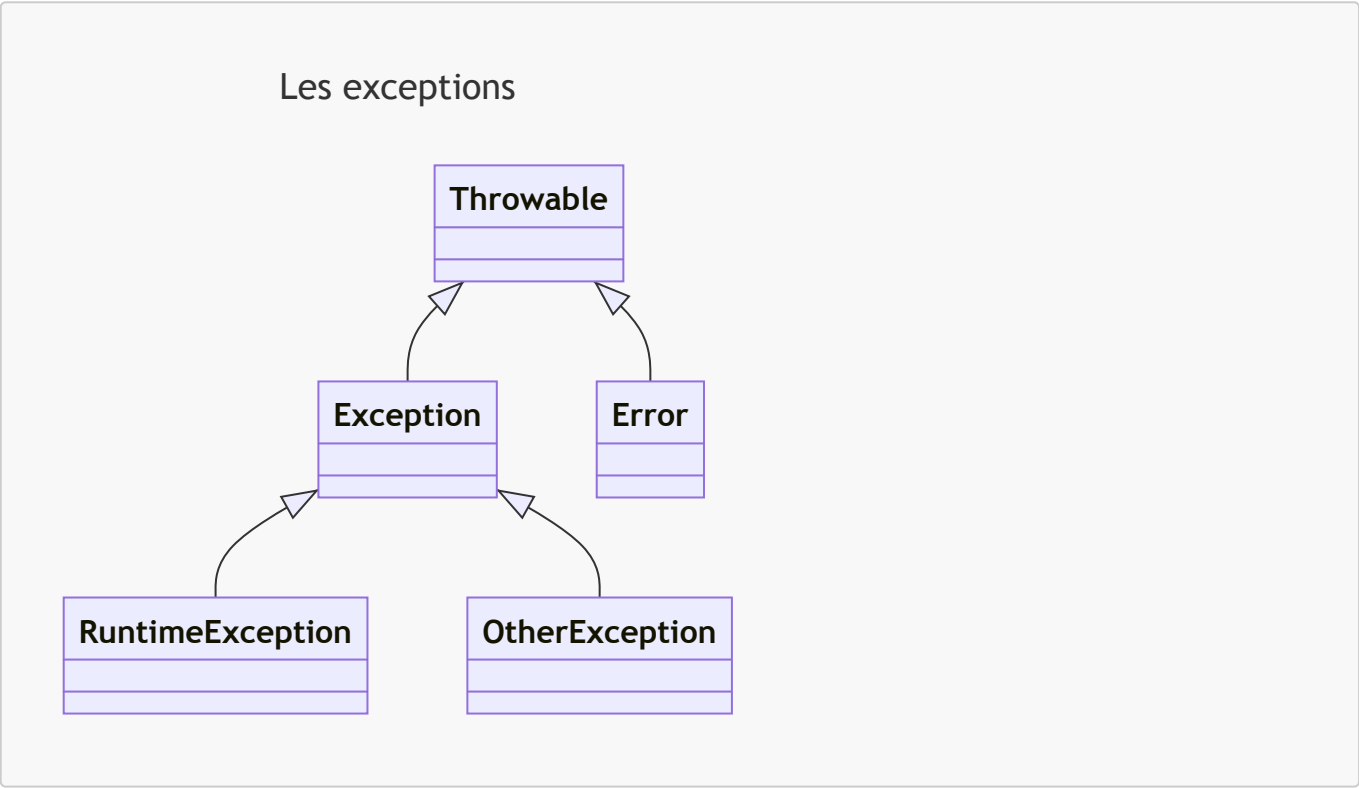
Événement intervenant lors de l'exécution d'un programme qui interrompt le flow d'instructions normales d'un programme.

Les exceptions entraine une interruption dans le programme

En java, une **Exception** est une classe étendant la classe **Throwable**

Classe Throwable

La classe **Throwable** est la classe de base de toutes les exceptions.



Les méthodes suivants sont disponibles :

Méthode	Description
String getMessage()	Retourne le message de l'exception
void printStackTrace()	Affiche l'exception avec l'état de la pile (« stack »).
Throwable getCause()	Retourne l'origine de l'erreur

Classe Error

La classe **Error** est la classe représentant une erreur grave en provenance de la JVM ou d'un de ses composants

Cette erreur entraine un arrêt du programme

Vous ne devez pas étendre de la classe `Error`

Vous ne devez pas intercepter ces exceptions. Lorsqu'une exception de ce type est levée, votre application aura de forte chance d'être totalement instable.

Classe `RuntimeException`

Le classe **`RuntimeException`** est la classe dont les exceptions ne sont pas obligatoirement interceptées par « try catch »

Ces exceptions peuvent être levées sans que la méthode l'indique via le mot clé « throws ».

On parle d'exception implicite

Exemple :

- `NullPointerException`
- `ArrayIndexOutOfBoundsException`

Classe **`Exception`**

La classe **`Exception`** est la classe représentant les erreurs classiques remontées le plus souvent par les méthodes

Ces exceptions doivent être interceptées par le bloc « try catch »

Créer son exception

Pour créer de nouvelles exceptions, il suffit d'étendre la classe **`Exception`**.

```
package fr.epsi.exception;

public class UserNotFoundException extends Exception{
    public UserNotFoundException() {
        super("Utilisateur non trouvé");
    }

    public UserNotFoundException(Long identifier) {
        super("Utilisateur %d non trouvé".formatted(identifier));
    }
}
```

Par convention, les exceptions sont suffixées par le mot « Exception »

Utilisation d'une exception : Le mot clé `throws`

Pour indiquer qu'une méthode peut « lever » une exception, il faut utiliser le mot clé **`throws`** suivi de l'ensemble des exceptions pouvant être levées. Les exceptions sont séparées par une virgule. Ce mot clé est à positionner à la suite des paramètres de la méthode.

Pour lever l'exception, le mot clé « `throw` » doit être utilisé suivi d'une instance de la classe d'exception.

La levée de l'exception entraîne une interruption de la méthode.

Dans le cas de l'exemple suivant, on constate que dans le cas où l'utilisateur n'est pas trouvé, l'exception **UserNotFoundException** sera levée par la méthode.

Fichier [UserService](#)

```
public class UserService {

    private Map<Long, User> users = Map.of(
        1L, new User(1L, "Donald", "Duck"),
        2L, new User(2L, "Mickey", "Mouse"),
        3L, new User(3L, "Peter", "Pan"),
        4L, new User(4L, "Mini", "Mouse")
    );

    public User findUserById(Long identifier) throws
    UserNotFoundException{
        User user = this.users.get(identifier);
        if(user == null) {
            throw new UserNotFoundException(identifier);
        }
        return user;
    }

}
```

Lorsque vous réalisez des classes d'exception, n'oubliez pas de passer dans le/les constructeurs des éléments qui permettront de comprendre facilement le problème.

Intercepter une exception

Toute exception n'étendant ni **RuntimeException**, ni **Error** doit être interceptée par un bloc **try catch**.

En appelant méthode **findUserById**, l'exemple suivant gère l'exception **UserNotFoundException**

Fichier [Main](#)

```
package fr.epsi;

import fr.epsi.exception.UserNotFoundException;
import fr.epsi.service.UserService;

public class Main {

    public static void main(String[] args) {
        var userService = new UserService();

        try {
            var user = userService.findUserById(1L);
        }
```

```

        System.out.println("Utilisateur : " + user);

        var otherUser = userService.findUserById(20L);
        System.out.println("Utilisateur : " + otherUser);
    } catch (UserNotFoundException e) {
        System.err.println(e.getMessage());
    }
}
}

```

Il est possible d'intercepter plusieurs exceptions de 2 manières différents.

Soit en appliquant plusieurs catch à la suite si l'on souhaite réaliser des traitements en fonction des exceptions levées.

Fichier [Main](#)

```

package fr.epsi;

import fr.epsi.exception.UserNotFoundException;
import fr.epsi.service.UserService;

public class Main2 {

    public static void main(String[] args) {
        var userService = new UserService();

        try {
            var user = userService.findUserById(1L);
            System.out.println("Utilisateur : " + user);

            var otherUser = userService.findUserById(20L);
            System.out.println("Utilisateur : " + otherUser);
        } catch (UserNotFoundException e) {
            System.err.println(e.getMessage());
        } catch (NullPointerException e){
            System.err.println("Exception qui n'aurait pas du être levée.
Message : " + e.getMessage());
        }
    }
}

```

```

mvn --quiet compile exec:java -Dexec.mainClass=fr.epsi.Main2
Utilisateur : User[identifiant=1, firstname=Donald, lastname=Duck]
Utilisateur 20 non trouvé

```

Soit en séparant les exceptions par un pipe |. On ne dissocie plus le traitement à réaliser en fonction de l'exception.

Fichier [Main](#)

```
package fr.epsi;

import fr.epsi.exception.UserNotFoundException;
import fr.epsi.service.UserService;

public class Main3 {

    public static void main(String[] args) {
        var userService = new UserService();

        try {
            var user = userService.findUserById(1L);
            System.out.println("Utilisateur : " + user);

            var otherUser = userService.findUserById(20L);
            System.out.println("Utilisateur : " + otherUser);
        } catch (UserNotFoundException|NullPointerException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=fr.epsi.Main3
Utilisateur : User[identifiant=1, firstname=Donald, lastname=Duck]
Utilisateur 20 non trouvé
```

Le mot clé **finally**

Il est parfois nécessaire de réaliser un traitement spécifique qu'une exception soit levée ou non.

Le mot clé **finally** permet ainsi d'introduire un bloc d'instructions qui sera **toujours** réalisé au sein d'un bloc **try...catch**.

Par exemple, imaginons que nous réalisons le parcours d'un fichier. Il sera nécessaire de fermer celui-ci en fin de traitement.

```
package fr.epsi;

import fr.epsi.domain.User;
import fr.epsi.exception.UserNotFoundException;
import fr.epsi.service.UserService;

public class Main4 {

    public static void main(String... args) throws Exception{
        UserService userService = new UserService();
```



```
        try {
            User user = userService.findUserById(20L);
        } catch (UserNotFoundException e) {
            System.out.println("Exception remontée");
            System.out.println(e.getMessage());
            throw e;
        } finally {
            System.out.println("Traitement finally en cours");
        }
    }
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=fr.epsi.Main4
Exception remontée
Utilisateur 20 non trouvé
Traitement finally en cours
Exception in thread "main" fr.epsi.exception.UserNotFoundException:
Utilisateur 20 non trouvé
    at fr.epsi.service.UserService.findUserById(UserService.java:20)
    at fr.epsi.Main4.main(Main4.java:12)
```

L'interface AutoCloseable

Comme vu en introduction, la classe **Object** propose la méthode **finalize** appelée lors destruction de l'instance.

L'appel à cette méthode est totalement imprévisible. L'appel sera réalisée uniquement lors d'un passage du garbage collector lorsque la jvm détecte la nécessité de libérer de la mémoire (dans la seconde ou voir plusieurs minutes après l'utilisation de l'instance).

Afin de contrôler# Les Exceptions

Introduction

Événement intervenant lors de l'exécution d'un programme qui interrompt le flow d'instructions normales d'un programme.

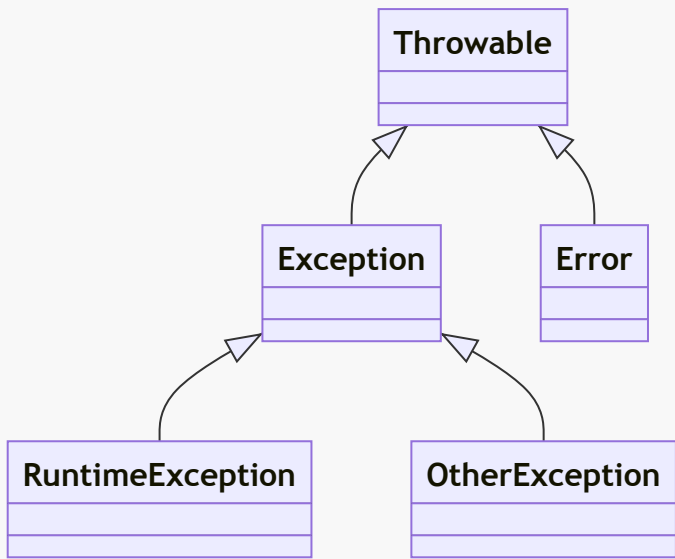
Les exceptions entraîne une interruption dans le programme

En java, une **Exception** est une classe étendant la classe **Throwable**

Classe Throwable

La classe **Throwable** est la classe de base de toutes les exceptions.

Les exceptions



Les méthodes suivants sont disponibles :

Méthode	Description
String getMessage()	Retourne le message de l'exception
void printStackTrace()	Affiche l'exception avec l'état de la pile (« stack »).
Throwable getCause()	Retourne l'origine de l'erreur

Classe Error

La classe **Error** est la classe représentant une erreur grave en provenance de la JVM ou d'un de ses composants

Cette erreur entraine un arrêt du programme

- Vous ne devez pas étendre de la classe Error
- Vous ne devez pas intercepter ces exceptions. Lorsqu'une exception de ce type est levée, votre application aura de forte chance d'être totalement instable.

Classe RuntimeException

Le classe **RuntimeException** est la classe dont les exceptions ne sont pas obligatoirement interceptées par « try catch »

Ces exceptions peuvent être levées sans que la méthode l'indique via le mot clé « throws ».

On parle d'exception implicite

Exemple :

- NullPointerException

- `ArrayIndexOutOfBoundsException`

Classe **Exception**

La classe **Exception** est la classe représentant les erreurs classiques remontées le plus souvent par les méthodes

Ces exceptions doivent être interceptées par le bloc « try catch »

Créer son exception

Pour créer de nouvelles exceptions, il suffit d'étendre la classe **Exception**.

```
package fr.epsi.exception;

public class UserNotFoundException extends Exception{
    public UserNotFoundException() {
        super("Utilisateur non trouvé");
    }

    public UserNotFoundException(Long identifier) {
        super("Utilisateur %d non trouvé".formatted(identifier));
    }
}
```

Par convention, les exceptions sont suffixées par le mot « Exception »

Utilisation d'une exception : Le mot clé throws

Pour indiquer qu'une méthode peut « lever » une exception, il faut utiliser le mot clé **throws** suivi de l'ensemble des exceptions pouvant être levées. Les exceptions sont séparées par une virgule. Ce mot clé est à positionner à la suite des paramètres de la méthode.

Pour lever l'exception, le mot clé « throw » doit être utilisé suivi d'une instance de la classe d'exception.

La levée de l'exception entraîne une interruption de la méthode.

Dans le cas de l'exemple suivant, on constate que dans le cas où l'utilisateur n'est pas trouvé, l'exception **UserNotFoundException** sera levée par la méthode.

Fichier [UserService](#)

```
public class UserService {

    private Map<Long, User> users = Map.of(
        1L, new User(1L, "Donald", "Duck"),
        2L, new User(2L, "Mickey", "Mouse"),
        3L, new User(3L, "Peter", "Pan"),
        4L, new User(4L, "Mini", "Mouse")
    );
}
```

```
    public User findUserById(Long identifier) throws
    UserNotFoundException{
        User user = this.users.get(identifier);
        if(user == null) {
            throw new UserNotFoundException(identifier);
        }
        return user;
    }
}
```

Lorsque vous réalisez des classes d'exception, n'oubliez pas de passer dans le/les constructeurs des éléments qui permettront de comprendre facilement le problème.

Intercepter une exception

Toute exception n'étendant ni **RuntimeException**, ni **Error** doit être interceptée par un bloc **try catch**.

En appelant méthode **findUserById**, l'exemple suivant gère l'exception **UserNotFoundException**

Fichier [Main](#)

```
package fr.epsi;

import fr.epsi.exception.UserNotFoundException;
import fr.epsi.service.UserService;

public class Main {

    public static void main(String[] args) {
        var userService = new UserService();

        try {
            var user = userService.findUserById(1L);
            System.out.println("Utilisateur : " + user);

            var otherUser = userService.findUserById(20L);
            System.out.println("Utilisateur : " + otherUser);
        } catch (UserNotFoundException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

Il est possible d'intercepter plusieurs exceptions de 2 manières différents.

Soit en appliquant plusieurs catch à la suite si l'on souhaite réaliser des traitements en fonction des exceptions levées.

Fichier [Main](#)

```
package fr.epsi;

import fr.epsi.exception.UserNotFoundException;
import fr.epsi.service.UserService;

public class Main2 {

    public static void main(String[] args) {
        var userService = new UserService();

        try {
            var user = userService.findUserById(1L);
            System.out.println("Utilisateur : " + user);

            var otherUser = userService.findUserById(20L);
            System.out.println("Utilisateur : " + otherUser);
        } catch (UserNotFoundException e) {
            System.err.println(e.getMessage());
        } catch (NullPointerException e){
            System.err.println("Exception qui n'aurait pas du être levée.
Message : " + e.getMessage());
        }
    }
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=fr.epsi.Main2
Utilisateur : User[identifiant=1, firstname=Donald, lastname=Duck]
Utilisateur 20 non trouvé
```

Soit en séparant les exceptions par un pipe |. On ne dissocie plus le traitement à réaliser en fonction de l'exception.

Fichier [Main](#)

```
package fr.epsi;

import fr.epsi.exception.UserNotFoundException;
import fr.epsi.service.UserService;

public class Main3 {

    public static void main(String[] args) {
        var userService = new UserService();

        try {
            var user = userService.findUserById(1L);
```

```
        System.out.println("Utilisateur : " + user);

        var otherUser = userService.findUserById(20L);
        System.out.println("Utilisateur : " + otherUser);
    } catch (UserNotFoundException|NullPointerException e) {
        System.err.println(e.getMessage());
    }
}
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=fr.epsi.Main3
Utilisateur : User[identifiant=1, firstname=Donald, lastname=Duck]
Utilisateur 20 non trouvé
```

Le mot clé **finally**

Il est parfois nécessaire de réaliser un traitement spécifique qu'une exception soit levée ou non.

Le mot clé **finally** permet ainsi d'introduire un bloc d'instructions qui sera **toujours** réalisé au sein d'un bloc **try...catch**.

Par exemple, imaginons que nous réalisons le parcours d'un fichier. Il sera nécessaire de fermer celui-ci en fin de traitement.

Fichier [Main4](#)

```
package fr.epsi;

import fr.epsi.domain.User;
import fr.epsi.exception.UserNotFoundException;
import fr.epsi.service.UserService;

public class Main4 {

    public static void main(String... args) throws Exception{
        UserService userService = new UserService();
        try {
            User user = userService.findUserById(20L);
        } catch (UserNotFoundException e) {
            System.out.println("Exception remontée");
            System.out.println(e.getMessage());
            throw e;
        } finally {
            System.out.println("Traitement finally en cours");
        }
    }
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=fr.epsi.Main4
Exception remontée
Utilisateur 20 non trouvé
Traitement finally en cours
Exception in thread "main" fr.epsi.exception.UserNotFoundException:
Utilisateur 20 non trouvé
    at fr.epsi.service.UserService.findById(UserService.java:20)
    at fr.epsi.Main4.main(Main4.java:12)
```

L'interface AutoCloseable

Comme vu en introduction, la classe **Object** propose la méthode **finalize** appelée lors destruction de l'instance.

L'appel à cette méthode est totalement imprévisible. L'appel sera réalisée uniquement lors d'un passage du garbage collector lorsque la jvm détecte la nécessité de libérer de la mémoire (dans la seconde ou voir plusieurs minutes après l'utilisation de l'instance).

Imaginons que l'instance libère un fichier, cette libération sera réalisée que plus tard après usage si nous décidions de le réaliser dans la méthode **finalize**.

Une ancienne technique consiste à réaliser cette libération au sein du bloc d'instructions **finally**.

Une autre technique consiste à déclarer l'instance entre parenthèses au niveau du **try**. La classe de l'instance doit dans ce cas implémentée l'interface **AutoCloseable** et ainsi fournir une implémentation à la méthode **close**. La JVM appellera automatiquement la méthode **close**

Fichier [MyResource](#)

```
package fr.epsi;
public class MyResource implements AutoCloseable{
    public void doStuff(){
        System.out.println("Je travaille....");
    }
    @Override
    public void close() {
        System.out.println("Good bye ! J'ai libéré des ressources");
    }
}
```

Fichier [Main5](#)

```
package fr.epsi;

public class Main5 {

    public static void main(String[] args) {
        try (var resource = new MyResource()) {
```

```
        resource.doStuff();
    }
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=fr.epsi.Main5
Je travaille....
Good bye ! J'ai libéré des ressources
```