

Javalisons la vie !!

PRÉLUDE

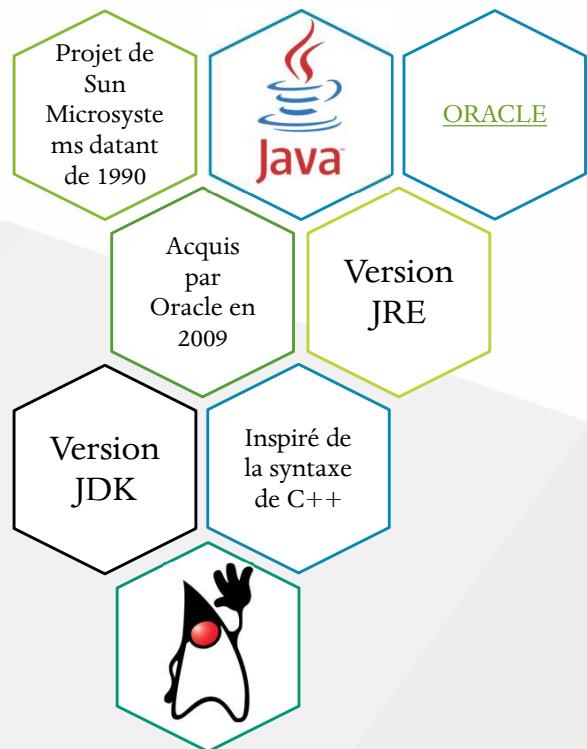
Nous allons aborder le langage de votre formation : JAVA.

Nous verrons de nombreux éléments, notamment les termes qui définissent Java ainsi que les concepts qui l'accompagnent.

N'hésitez pas à noter ceux qui vous paraissent compliqués, car ils font partie de la culture du développement que vous devrez être capables de citer.

C'est parti...

APERÇU



Robuste et sûr: met l'accent principalement sur la vérification des erreurs.



Indépendant de la machine employée pour l'exécution: le compilateur génère un format de fichier neutre en architecture.



Très performant: Grâce à l'utilisations de compilateurs Just-In-Time (JIT).



Compilé, multi-tâches et dynamique: compilé en bytecode indépendante de la plate-forme, multithread, s'adapte à un environnement en évolution.



Simple, orienté objet et familier: facile à apprendre, tout est objet



Mécanisme de ramasse-miettes
Garbage Collector



ENVIRONNEMENT DE DÉVELOPPEMENT MISE EN PLACE ET PRISE EN MAIN

ENVIRONNEMENT DE DÉVELOPPEMENT

[option] Téléchargement de la JDK

La JDK pour Java Développer Kit est le kit permettant d'avoir toute la librairie Java pour développer des applications en Java.

Nous travaillerons avec la JDK 21 : [OpenJdk](#)

<https://jdk.java.net/archive/>

[Pourquoi ? java-se-support-roadmap](#)

1. Tout d'abord, nous allons télécharger la JDK et l'installer sur notre poste de travail
2. Décompresser l'archive à l'emplacement voulu et mettre à jour la variable d'environnement PATH pour y inclure le chemin vers le dossier bin de la JDK

Installation à partir de l'IDE : IntelliJ

<https://www.jetbrains.com/idea/>

1. S'inscrire chez JetBrains avec votre email afpa-dev-pompey.fr
2. Choisir la licence Pack Etudiant
3. Télécharger et installer
4. Puis lancer l'IDE et s'identifier avec votre compte



ORGANISATION BIEN STRUCTURER SON CODE

LA NOTION DE DYNAMIQUE

Hier en algorithme

Avec l'algorithme, on a travaillé avec deux types de variables :

Statiques : les variables qu'on utilise tout au long de notre programme.

Automatiques : lors de la création de procédures ou de fonctions, on crée des variables le temps de l'exécution de l'appel.

- Dans les deux cas , ces variables sont déclarées en début de programme...
- Ces variables sont définies avec un indicateur et un type

demain avec Java

Une application doit gérer un grand nombre de données, variable dans le temps, difficilement estimable en quantité donc en ressources.

C'est pourquoi, avec des langages évolués vient l'idée de **variables dynamiques**.

- Ces variables seront créées quand on en a besoin d'où l'apparition de la notion **d'objets**.
- **Un objet va pouvoir contenir tous un ensemble d'informations et de méthodes utilisables selon nos souhaits.**

UN PROGRAMME JAVA

Tout programme Java commence par **une classe principale** qui contiendra **une méthode main**, point de départ de l'application.

Le compilateur Java ira chercher ce point d'entrée de votre application afin d'en exécuter la 1ère instruction.

```
public class App {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        System.out.println("Hello World!");  
    }  
}
```

ORGANISATION : BONNE PRATIQUE

Eviter de mettre une logique de code dans le main.

Une bonne pratique est de faire appel à un objet de la classe ici App, objet qui fera ensuite appel à une méthode qui contiendra la logique de l'application.

Ne rien garder dans votre fonction main qui puisse être extrait vers une méthode.

```
④ /**
 * @author jboeb
 *
 */
public class App {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub

        // etape 1
        System.out.println("Hello World!");

        // etape 2 : pour sortir la logique de code du main
        App app = new App();
        app.start();
    }

    public void start()
    {
        System.out.println("Démarrage du programme");
    }
}
```

UN PROGRAMME JAVA

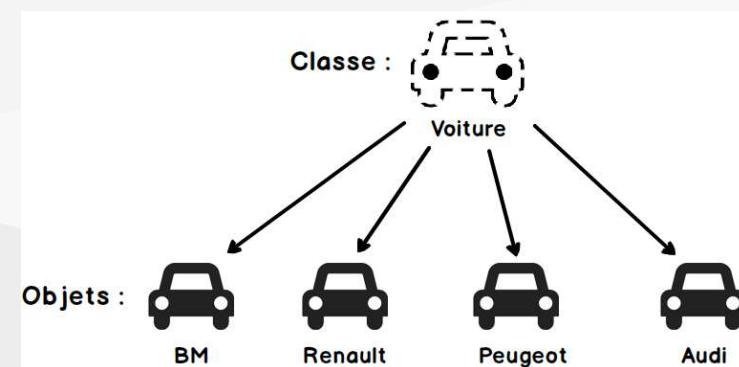
- Un programme Java peut être défini comme une collection de classes qui vont nous permettre de créer une collection d'objets qui communiquent via l'invocation des méthodes.
- **Classe**
 - Une classe peut être définie comme un modèle qui décrit les comportements ou les états de l'objet. Son extension est `.java`
 - La 1^{ère} classe de toute application java contient un méthode `main` qui sera le point d'entrée de l'application.
- **Objet**
 - Les objets ont des propriétés et des comportements. **Un objet est une instance d'une classe.**

Méthodes

- Une classe peut contenir plusieurs méthodes.
 - Une méthode est un comportement.
 - C'est dans les méthodes où les traitements sont écrits, les données sont manipulées et toutes les actions sont exécutées.

Variables d'instance

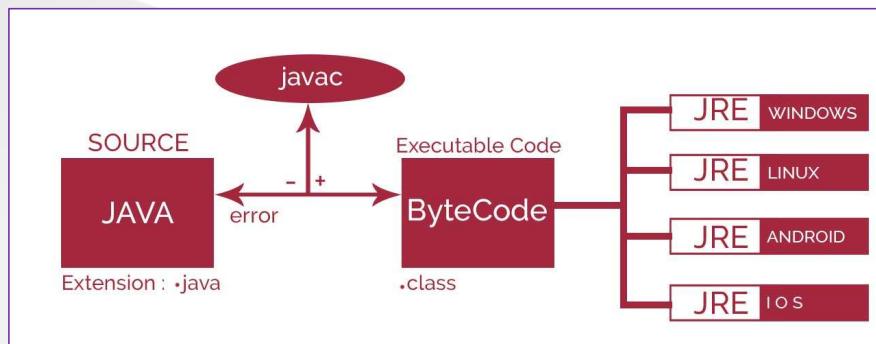
- Chaque objet possède une variable d'instance. L'état d'un objet est créé par les valeurs affectées à ces variables d'instance.
- Ces variables d'instance peuvent être modifiées par des méthodes.



COMPILEATION

Deux étapes sont importantes pour qu'un programme Java se lance : la compilation et l'interprétation.

1. Le compilateur intervient en amont pour prendre le code du développeur et le transformer en **byteCode** (ou code binaire – langage machine).
2. Puis l'interpréteur (JIT) traduit le **byteCode** en instructions pour exécuter le programme.



EXEMPLE DE COMPILEATION

Vous n'avez pas forcément besoin d'un IDE pour coder et compiler un projet Java

1. Pour pouvoir exécuter ce programme, nous allons devoir le compiler. Pour cela, nous devons utiliser le programme **javac** (Java Compiler) dont c'est la fonction.
2. Cela nous donne un fichier Bytecode en **.class**
3. Ensuite avec le programme **java**, nous pouvons lancer le programme

```
/**  
 * Ce programme n'est pas très intéressant  
 */  
public class PremierProgramme {  
  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

Javac PremierProgramme.java

```
%javac -d . PremierProgramme.java  
%javac -d . PremierProgramme.java  
%java PremierProgramme  
Hello World!
```

java PremierProgramme

```
$ java PremierProgramme  
Hello World!
```

LIAISON DYNAMIQUE

La **liaison dynamique** implique qu'un programme Java est une collection de fichiers.

- Ces fichiers peuvent se trouver à différents endroits dans le système de fichiers. Il faut donc un mécanisme pour permettre de les localiser.
- En Java, on utilise le **classpath** : le chemin des classes.
- *On peut par exemple spécifier un ou plusieurs chemins avec le paramètre **-classpath** aux commandes **java** et **javac** indiquant les répertoires à partir desquels il est possible de trouver des fichiers class.*

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/JAVA
$ java -classpath JavaTuto/exemple/ PremierProgramme
Hello World!
```

Java ne supporte que la liaison dynamique.

- Cela signifie que chaque fichier compilé donnera un fichier **class**.
- Cela signifie également qu'un programme Java est en fait une collection de plusieurs fichiers class.

Si votre programme est dépendant d'une bibliothèque tierce en Java, vous devez également fournir les fichiers de cette bibliothèque au moment de l'exécution.

Note : un programme Java a forcément une dépendance.

- Avec la classe **System**.
- Avec la classe **Objet**
 - Ces classes sont fournies avec la JDK et c'est donc la JVM qui va trouver le code sources de ces classes. C'est pourquoi on n'a pas besoin de les nommer explicitement dans notre exemple précédent. **C'est grâce à la liaison Dynamique.**



BIBLIOTHÈQUES JAVA LES FICHIERS JAR

LES FICHIERS JAR

Java est une collection de fichiers class et qu'il n'est pas rare qu'un programme ait besoin de centaines voire de milliers de ces fichiers alors on se rend vite compte qu'il n'est pas très facile de distribuer un programme Java sous cette forme.

Pour pallier ce problème, on peut utiliser des fichiers **jar**.

JAR signifie **Java ARchive** : il s'agit d'un fichier zip contenant un ensemble de fichiers class mais qui a l'extension **.jar**.

Java fournit l'utilitaire jar pour créer une archive :

- `jar -cf monappli.jar App.class`

Depuis l'IDE, si l'on souhaite distribuer son projet, on peut, par exemple, produire un fichier JAR.

Si l'on souhaite utiliser dans son projet, une librairie JAR pour ajouter une nouvelle fonctionnalité, il suffit d'inclure à son projet une librairie JAR.

Procédure:

1. Ouvrez votre projet IntelliJ IDEA installé et
2. Allez dans **Fichier > Structure du projet**
3. Sélectionnez **Modules** dans le panneau de gauche et sélectionnez l'onglet **Dépendances** .
4. Sélectionnez l'icône + et sélectionnez l'option **1 JAR ou Répertoires** .
5. sélectionnez votre fichier JAR ou vous pouvez sélectionner les répertoires.
6. Cliquez sur le bouton **OK**

<https://stacklima.com/comment-ajouter-un-fichier-jar-externe-a-un-projet-intellij-idea/>



ORGANISATION EN PACKAGE

ORGANISATION : PACKAGE

Dans le cycle de vie d'un projet : Le temps consacré au "débugage" et à la maintenance est largement supérieur au temps de développement.

Les sources du projet doivent donc :

- Suivre les mêmes règles de syntaxe et de présentation.
- Être correctement structurées.
- Être suffisamment documentées.

➤ Respect des conventions de noms

Les packages

Un package est une bibliothèque d'objets, c'est-à-dire une collection de classes rangées ensemble (dans un même répertoire).

Il y a trois règles pour faire des packages :

- **Structuration des projets** : chaque package est relié à un répertoire : Organisation arborescente
- **Conflits de noms** : les packages définissent **des espaces de nommage**. (avoir deux classes de même nom mais d'un package différent)
- **Affiner la visibilité** : la notion de package est associée à un niveau de visibilité intermédiaire.

ORGANISATION : PACKAGE

Pour créer un package :

- Créer un répertoire
- Le nom de répertoire, **en minuscule**, est le nom du package.

Pour rattacher une classe à un package :

- On écrit sur **la 1ère ligne** du fichier contenant la définition de la classe le mot-clé package suivi de son nom.
- **Les classes d'un même package sont toutes rangées dans un même répertoire.**

Pour éviter les conflits lors de la distribution des classes :

- On utilise comme racine une URL inversée suivie des répertoires de l'arborescence
 - **Ex : package fr.perso.projet.hello;**

Attention : toutes les classes d'un projet doivent être affectées à un package.

Par défaut, une classe a accès à l'espace de nom de son propre package et du package **java.lang**.

Lorsque l'on utilise un membre (de classe ou d'instance) d'une classe A dans une méthode d'une classe B, le compilateur va chercher la classe A :

1. dans la classe B (notion de classe interne) ;
2. parmi toutes les classes appartenant au même package que B
3. parmi toutes les classes du package **java.lang**.

Si la classe A n'est pas définie dans l'un de ces trois ensembles, vous devez aider le compilateur à la trouver en indiquant :

- soit à chaque utilisation de la classe A ;
- soit une fois pour toute au début de la classe B [**IMPORT**]

EXEMPLE

- 1 Se trouve dans le même package ou dans java.lang
- 2 Se trouve en dehors du package et pas dans java.lang donc import

The screenshot shows an IDE interface with a project named "tutoriel". The project structure on the left includes ".idea", "out", "src" (which contains "fr.afpa.dev.pompey" package with "App" and "Senseo" classes), ".gitignore", "tutoriel.iml", "External Libraries", and "Scratches and Consoles". The right pane displays the code for "App.java".

```
1 package fr.afpa.dev.pompey;
2
3
4 // importation de la classe Array provenant du package java.util
5 import java.util.Arrays;
6
7 public class App {
8     /**
9      * @param args
10     */
11    public static void main(String[] args) {
12        System.out.println("Hello world!");
13
14        // TYPE Object
15        String chaine; // chaîne de caractères
16        Arrays tableau;
17
18        Senseo senseo = new Senseo();
19
20    }
21}
```



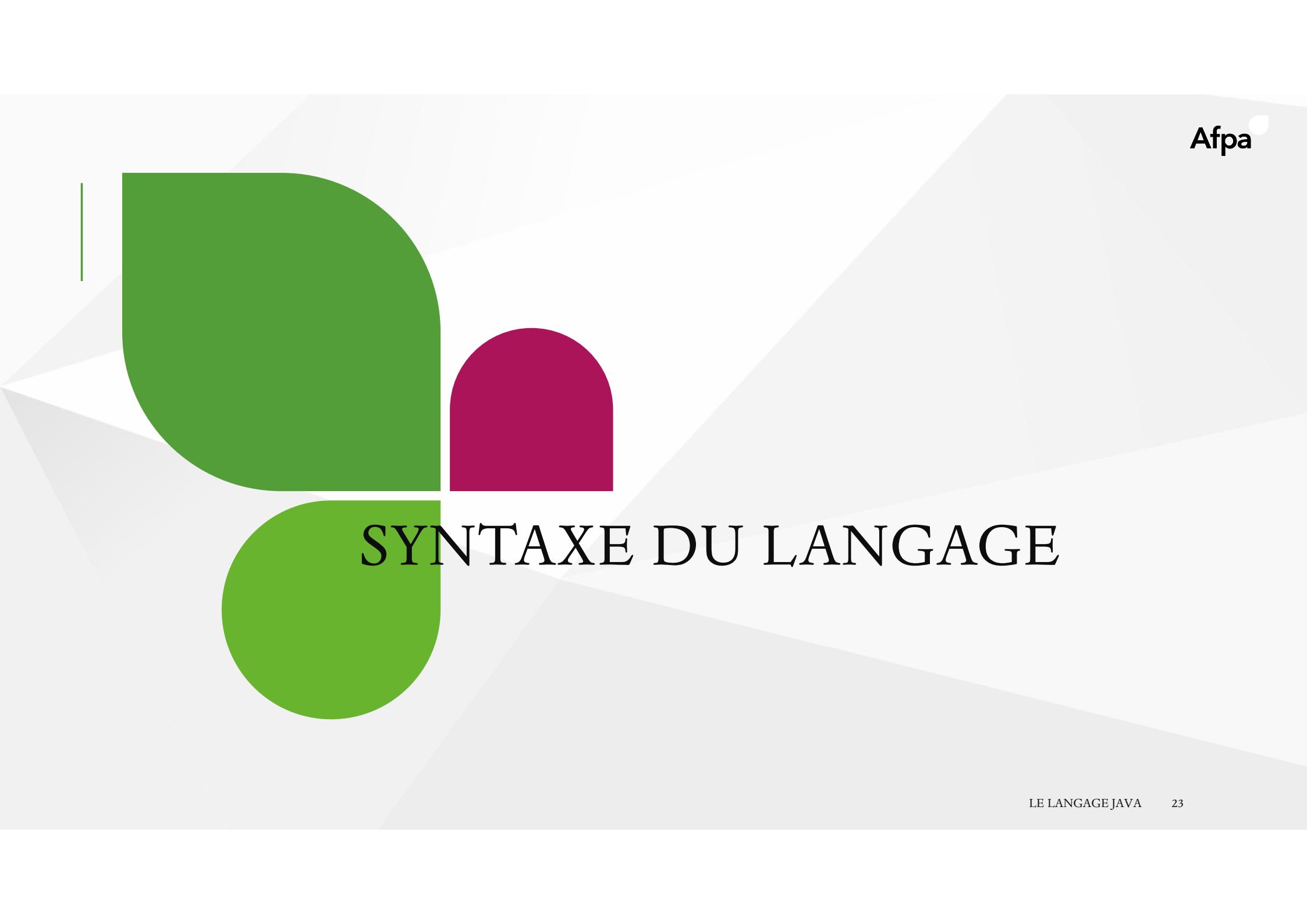
CONVENTIONS DE NOMMAGE RÈGLES

CONVENTIONS DE NOMMAGE

Type	Convention	Exemple
Packages	Un nom de package s'écrit toujours en minuscule. L'utilisation d'un _ est tolérée pour représenter une séparation.	java.utils com.company.extra_utils
Classes et interfaces	Le nom des classes et des interfaces ne doit pas être des verbes. La première lettre de chaque mot doit être en majuscule (écriture dromadaire).	MyClass SuppressionClientOperateur
Annotations	La première lettre de chaque mot doit être une majuscule (écriture dromadaire). Il est toléré d'écrire des sigles intégralement en majuscules.	@InjectIn @EJB
Méthodes	Le nom d'une méthode est le plus souvent un verbe. La première lettre doit être en minuscule et les mots sont séparés par l'utilisation d'une majuscule (écriture dromadaire).	run() runFast() getWidthInPixels()

CONVENTIONS DE NOMMAGE

Type	Convention	Exemple
Variables	<p>La première lettre doit être en minuscule et les mots sont séparés par l'utilisation d'une majuscule (écriture dromadaire).</p> <p>En Java, les développeurs n'ont pas pour habitude d'utiliser une convention de nom pour différencier les variables locales des paramètres ou même des attributs d'une classe.</p> <p>Le nom des variables doit être explicite sans utiliser d'abréviation. Pour les variables « jetables », l'utilisation d'une lettre est d'usage (par exemple i, j ou k)</p>	widthInPixels clientsInscrits total
Constantes	<p>Le nom d'une constante s'écrit intégralement en lettres majuscules et les mots sont séparés par _.</p>	LARGEUR_MAX INSCRIPTIONS_PAR_ANNEE



SYNTAXE DU LANGAGE

LES TYPES

- **une conversion implicite** : la modification est réalisée automatiquement par le compilateur, si la conversion du type A vers le type B peut se faire sans perte quelle que soit la valeur du type A
- **une conversion explicite** via l'opérateur **cast** sous la forme **(type_souhaité)variable** est nécessaire.

Java est un langage fortement typé. Il est **obligatoire de préciser son type**.

Primitifs

Nom	Taille en octets lors des calculs	Valeur par défaut	Valeurs possibles
boolean	Un seul bit suffit, mais on réserve souvent un octet pour les stocker.	false	true , false
byte	1	0	entiers compris entre -128 et +127 (-2 ⁷ et 2 ⁷ -1)
short	2	0	entiers compris entre -32 768 et 32 767 (-2 ¹⁵ et 2 ¹⁵ -1)
int	4	0	entiers compris entre -2 147 483 648 et +2 147 483 647 (-2 ³¹ et 2 ³¹ -1)
long	8	0	entiers compris entre -9 223 372 036 854 775 808 et 9 223 372 036 854 775 807 (-2 ⁶³ et 2 ⁶³ -1)
char	2	'\u0000'	Ensemble des valeurs Unicode (valeurs de U+0000 à U+FFFF, 4 chiffres obligatoires après '\u') Les 128 premiers caractères sont les codes ASCII et se notent entre apostrophes : 'a' , '1' , '\'' , '\n' .
float	4	0.0	Ensemble des nombres [-3,402 823 47 × 10 ³⁸ .. -1,402 398 46 × 10 ⁻⁴⁵] , 0, [1,402 398 46 × 10 ⁻⁴⁵ .. 3,402 823 47 × 10 ³⁸]
double	8	0.0	Ensemble des nombres [-1,797 693 134 862 315 70 × 10 ³⁰⁸ .. -4,940 656 458 412 465 44 × 10 ⁻³²⁴] , 0, [4,940 656 458 412 465 44 × 10 ⁻³²⁴ .. 1,797 693 134 862 315 70 × 10 ³⁰⁸]
Object	Dépendant de la machine virtuelle	null	

Références

CONVERSION IMPLICITE

Java est très strict du point de vue des données et il est parfois intéressant de convertir une donnée d'un **type primitif vers un autre type primitif**.

Il y a deux types de conversion : [conversion sans pertes d'informations](#) et [conversion avec pertes d'informations](#)

Important : on ne peut pas convertir une donnée Boolean vers un autre type

Il existe également une [conversion par affectation](#) (ex: `x = y` où `y` sera convertie dans le type de `x` avant affectation). Cette conversion n'est possible que sans pertes d'informations *[sinon Erreur de compilation.](#)*

De Vers

<code>byte</code>	<code>short, int, long, float, double</code>
<code>short</code>	<code>int, long, float, double</code>
<code>char</code>	<code>int, long, float, double</code>
<code>int</code>	<code>long, float, double</code>
<code>long</code>	<code>float, double</code>
<code>float</code>	<code>double</code>

Sans perte

De Vers

<code>short</code>	<code>byte, char</code>
<code>char</code>	<code>byte, short</code>
<code>int</code>	<code>byte, short, char</code>
<code>long</code>	<code>byte, short, char, int</code>
<code>float</code>	<code>byte, short, char, int, long</code>
<code>double</code>	<code>byte, short, char, int, long, float</code>

Avec perte

CONVERSION EXPLICITE

Il existe des conversions avec perte d'informations dont nous n'aurons pas le choix de les réaliser de par leur utilité.

- Pour cela, on doit faire un **cast**, aussi appelé conversion explicite.

Ce **cast** se réalise à l'aide l'opérateur de cast qui va pouvoir convertir une donnée d'un type dans le type spécifié soit avec ou sans perte d'informations.

```
// conversion explicite CAST
x = 292;

f = (float) x;      // Conversion sans perte d'information
b = (byte) x;       // Conversion avec perte d'information
c = (char) b;        // Conversion d'un byte vers un char

System.out.println (f);
System.out.println (b);
System.out.println (c);
System.out.println ((byte) c);
```

```
292.0
36
$
36
```



LES CLASSES ENVELOPPÉES WRAPPER CLASS

LES CLASSES ENVELOPPÉES

Comme les types primitifs ne sont pas des classes, l'API standard de Java fournit également des classes qui permettent d'envelopper la valeur d'un type primitif : on parle de [wrapper class](#).

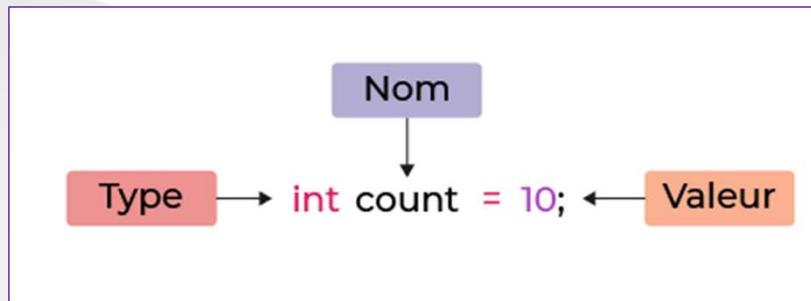
Pourquoi avoir créé ces classes ? Cela permet d'offrir un emplacement facile à mémoriser à des méthodes utilitaires.

```
boolean b = Boolean.parseBoolean("true");
byte by = Byte.parseByte("1");
short s = Short.parseShort("1");
int i = Integer.parseInt("1");
long l = Long.parseLong("1");
float f = Float.parseFloat("1");
double d = Double.parseDouble("1");
// enfin presque toutes car Character n'a pas cette méthode
```

Type	Classe associée
boolean	java.lang.Boolean
char	java.lang.Character
int	java.lang.Integer
byte	java.lang.Byte
short	java.lang.Short
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double

SYNTAXE

Déclaration de variables



En Java une instruction est délimitée par un **point-virgule**.

Rappels conventions

Le nom doit être explicite, lisible et compréhensible => **camelCase**.

Java est un langage à typage fort : on doit spécifier directement son type à une variable.

Tant qu'on ne lui affecte pas de valeur, la variable n'est pas utilisée.

```
int count; // sans affectation  
  
int count = 10; // avec affectation
```

SYNTAXE

Constante

Utilisation du mot-clé : final

Convention de nommage : **UPPER_SNAKE_CASE**

```
// déclaration d'une constante
final double PI = 3,14159;
```

Blocs de code

Java permet de structurer le code en bloc. Un bloc est délimité par des accolades.

{ ... }

- Un bloc permet d'isoler par exemple le code conditionnel à la suite d'un si..alors.. mais il est également possible de créer des blocs anonymes.
- Un bloc de code n'a pas besoin de se terminer par un point-virgule.

JAVADOC ET COMMENTAIRES

JAVADOC

A chaque fois qu'on crée une classe, vous êtes censé la documenter dans des marqueurs `/**` et `*/` à placer en début de déclaration d'une méthode, de variables etc...

Cela permet de générer automatiquement une page de documentation HTML.

Exemple de javadoc pour String : [String](#)

<https://docs.oracle.com/en/java/javase/21/docs/api/index.html>

Les commentaires

Dans votre futur code, vous devez également commenter votre logique.

Pour commenter deux types de marqueurs sont disponibles :

// commentaire sur une ligne

/*

* Bloc de commentaires

* ...

*/

SYNTAXE

En Java, une classe est déclaré par le mot-clé class suivi de son nom.

Le mot-clé public précise la portée (scope) de la définition de cette classe.

Ensute on définit le bloc de codes.

Dans cette classe, on peut contenir :

- Des attributs.
- Des méthodes.
- Des constantes.
- Des énumérations.
- Des classes internes.

```


    /**
     * Classe Modèle
     * @author jboeb
     * Décrit la composition d'une classe Java
     */
public class Model {

    /**
     * Déclaration des constantes
     */

    /**
     * Déclaration des éléments static
     */
        // Attributs
        // Méthodes
        // Getter
        // Setter

    /**
     * Déclaration des éléments d'instance
     */
        // Attributs
        // constructeurs
        // Méthodes d'instance

        // Public
        // Protected
        // Private

        // Méthodes abstract

        // Getter
        // Setter
}


```



LES OPÉRATEURS OPÉRATIONS SUR LES VARIABLES

LES OPÉRATEURS ARITHMÉTIQUES

Operator	Name	Description	Example
+	Addition	Adds together two values	$x + y$
-	Subtraction	Subtracts one value from another	$x - y$
*	Multiplication	Multiplies two values	$x * y$
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	$x \% y$
++	Increment	Increases the value of a variable by 1	$++x$
--	Decrement	Decreases the value of a variable by 1	$--x$

PREFIX ET POSTFIX

Prefix : ++x

Si la variable commence par ++, alors elle est appelée pré-incrémantation.

Exemple :

```
int x = 3;
```

```
Int a = ++x; // a = 4, x = 4
```

Postfix : x++

Si la variable se termine par ++, alors elle est appelée post-incrémantation.

Exemple :

```
int x = 3;
```

```
Int a = x++; // a = 3, x = 4
```

LES OPÉRATEURS DE COMPARAISONS

Operator	Name	Example
<code>==</code>	Equal to	<code>x == y</code>
<code>!=</code>	Not equal	<code>x != y</code>
<code>></code>	Greater than	<code>x > y</code>
<code><</code>	Less than	<code>x < y</code>
<code>>=</code>	Greater than or equal to	<code>x >= y</code>
<code><=</code>	Less than or equal to	<code>x <= y</code>

LES OPÉRATEURS LOGIQUES

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	<code>x < 5 && x < 10</code>
	Logical or	Returns true if one of the statements is true	<code>x < 5 x < 4</code>
!	Logical not	Reverse the result, returns false if the result is true	<code>!(x < 5 && x < 10)</code>

LES OPÉRATEURS

Ternaire

L'opérateur ternaire permet d'affecter une valeur suivant le résultat d'une condition.

Exp boolèenne ? Valeur si vrai : valeur si faux

Par exemple

```
String s = age >= 18 ? "Majeur" : "Mineur";
```

Nombre	Décimale (base 10) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0	00
1	01
2	02
3	03
4	04
...	...
9	09
10	10

Nombre	Binaire (base 2) [0, 1]
0	0000
1	0001
2	0010
3	0011
4	0100
...	...
9	1001
10	1010

Bitwise

Les opérateurs *bitwise* permettent de manipuler la valeur des bits d'un entier.

~ négation binaire

& Et binaire

^ Ou exclusif

| ou binaire

```
int i = 0b1;
i = 0b10 | i; // i vaut 0b11
i = 0b10 & i; // i vaut 0b10
i = 0b10 ^ i; // i vaut 0b00
i = ~i; // i vaut -1
```

LES OPÉRATEURS

Décalage

Les opérateurs de décalage s'utilisent sur des entiers et permettent de déplacer les bits vers la gauche ou vers la droite.

<< décalage vers la gauche

>> décalage vers la droite avec préservation du signe

>>> décalage vers la droite sans préservation du signe

```
int i = 1;
i = i << 1 // i vaut 2
i = i << 3 // i vaut 16
i = i >> 2 // i vaut 4
```

Transtypage

Il est parfois nécessaire de signifier que l'on désire passer d'un type vers un autre au moment de l'affectation.

Java étant un langage fortement typé, il autorise par défaut uniquement les opérations de transtypage qui sont sûres.

```
int i = 1;
Long l = i; // Ok
short s = (short) l; // cast obligatoire
```

LES OPÉRATEURS D'AFFECTATIONS

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
=	x = 3	x = x 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

L'OPÉRATEUR .

L'opérateur . permet d'accéder aux attributs et aux méthodes d'une classe ou d'un objet à partir d'une référence.

```
String s = "Hello the world";
int length = s.length();
System.out.println("La chaîne de caractères contient " + length + " caractères");
```

```
int length = "Hello the world".length();
```

```
String name = int.class.getName();
```



PRIORITÉS ET ASSOCIATIVITÉS DES OPÉRATEURS

PRIORITÉS ET ASSOCIATIVITÉS DES OPÉRATEURS

Priorités :

Le tableau suivant montre tous les opérateurs Java à partir de la priorité la plus élevée (1) à la priorité la plus faible (15).

Associativités :

La plupart des opérateurs sont **associatifs de gauche à droite**, sauf ceux des niveaux de priorité 2, 3, 14 et 15 qui sont associatifs de droite à gauche.

Bonne pratique : Utiliser des parenthèses.

Niveau	Opérateur	Description	Exemple
1	<code>++</code>	incrémentation postfixe	<code>x++</code>
	<code>--</code>	décrémentation postfixe	<code>x--</code>
2	<code>++</code>	incrémentation préfixe	<code>++x</code>
	<code>--</code>	décrémentation préfixe	<code>--x</code>
	<code>-</code>	changement de signe	<code>-x</code>
	<code>+</code>	signe +	<code>+x</code>
	<code>!</code>	NON logique	<code>! x</code>
4	<code>*</code>	multiplication	<code>x * y</code>
	<code>/</code>	division	<code>x / y</code>
	<code>%</code>	modulo	<code>x % y</code>
5	<code>+</code>	addition	<code>x + y</code>
	<code>-</code>	soustraction	<code>x - y</code>
7	<code><</code>	plus petit que	<code>x < y</code>
	<code><=</code>	plus petit ou égal à	<code>x <= y</code>
	<code>></code>	plus grand que	<code>x > y</code>
	<code>>=</code>	plus grand ou égal à	<code>x >= y</code>
8	<code>==</code>	égal	<code>x == y</code>
	<code>!=</code>	différent	<code>x != y</code>
10	<code>^</code>	OU logique exclusif	<code>x ^ y</code>
12	<code>&&</code>	ET logique	<code>x && y</code>
13	<code> </code>	OU logique inclusif	<code>x y</code>
14	<code>? :</code>	opérateur conditionnel	<code>x ? y : z</code>
15	<code>=</code>	affectation	<code>x = y</code>
	<code>+=, -=, *=, /=, %=</code>	affectations composées	<code>x += y</code>

Figure 15. Priorité de quelques opérateurs Java.

EXERCICES

Priorités des opérateurs

Eliminer les parenthèses superflues dans les expressions suivantes (l'ordre de calcul devant rester le même) :

1. $(a + b) - (2 * c)$
2. $(2 * x) / (y * z)$
3. $(x + 3) * (n \% p)$
4. $(-a) / (-(b + c))$
5. $(x / y) \% (-z)$

Corrections

Selon le tableau des priorités précédemment : A

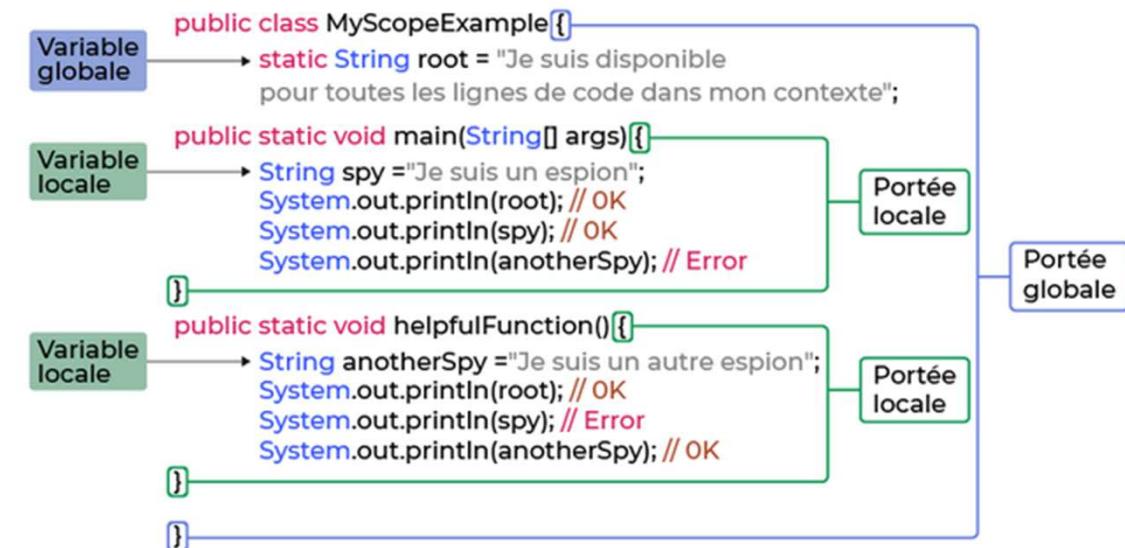
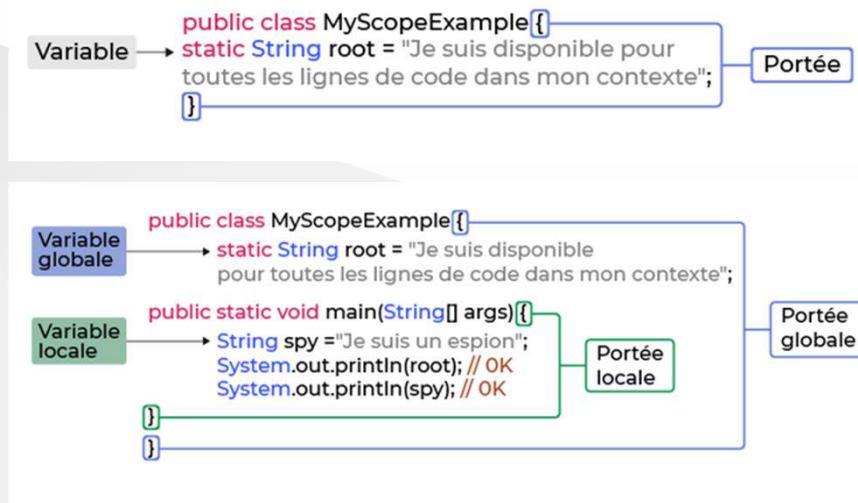
1. $a + b - 2 * c$
2. $2 * y / (y * z)$
3. $(x + 3) * (n \% p)$ $- *$ et $\%$ même priorité
4. $-a / -(b + c)$
5. $x / y \% -z$



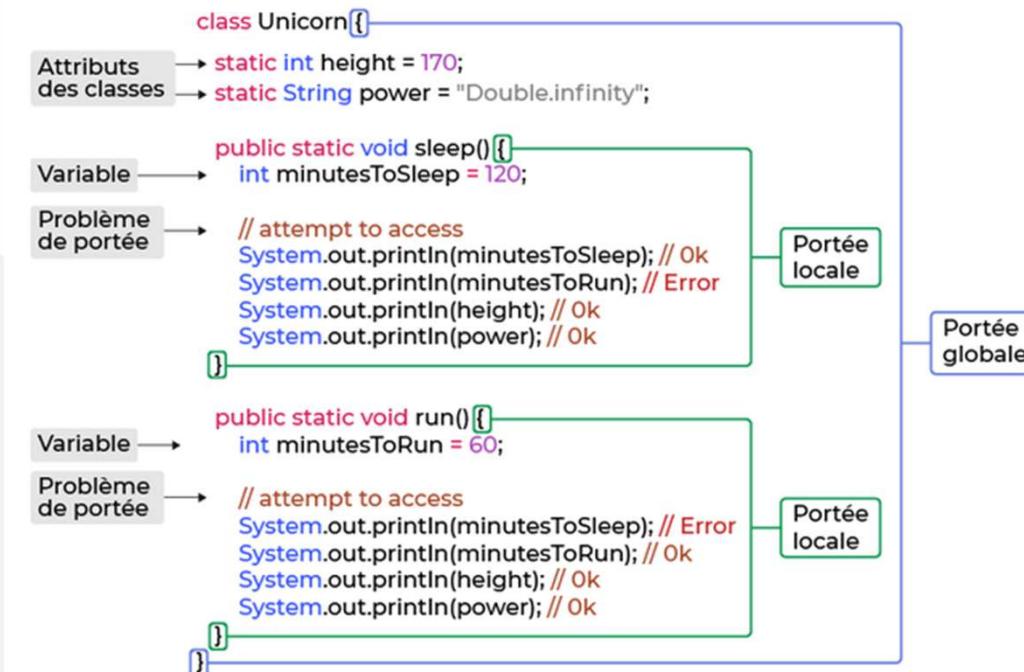
PORTEE DES VARIABLES

QUAND ? OÙ ? COMMENT ?

PORTEE DES VARIABLES



PORTEE DES VARIABLES





LES STRUCTURES DE CONTÔLES CONDITIONNEZ VOTRE CODE !!

LES STRUCTURES DE CONTROLES : FOR / WHILE

```
1 for (initialisation; terminaison; increment) {  
2 // code à répéter  
3 }
```

java

```
1 for (int i=0; i<5;i++) {  
2     System.out.println("Clap your hands!");  
3 }
```

```
1 while (logicalExpression) {  
2 // liste de déclarations  
3 }
```

```
1 int numberOfTrees = 0;  
2 while (numberOfTrees < 10) {  
3     numberOfTrees += 1;  
4     System.out.println("I planted " + numberOfTrees + " trees");  
5 }  
6  
7 System.out.println("I have a forest!");
```

LES STRUCTURES DE CONTROLES : DO WHILE / IF ELSE IF

```
1 do {  
2 // instructions  
3 } while(logicalExpression);
```

```
1 int pushUpGoal = 10;  
2 do{  
3     print ("Push up!");  
4     pushUpGoal -= 1;  
5 } while(pushUpGoal > 0);
```

```
1 if(condition1) {  
2 // instructions  
3 }  
4 else if(condition2) {  
5 // instructions  
6 }  
7 else {  
8 // instructions  
9 }
```

LES STRUCTURES DE CONTRÔLES : SWITCH

```
1 public static void main(String[] args) {
2     switch(args.length) {
3         case 0: // aucun argument n'a été envoyé
4             sayHelloTo("world");
5             break;
6         case 1: // l'utilisateur a fourni un argument dans le terminal
7             sayHelloTo(args[0]);
8             break;
9         case 2: // l'utilisateur a fourni 2 arguments
10            sayHelloTo(args[0] + "-" + args[1]);
11            break;
12        default: // l'utilisateur a fourni plus d'arguments qu'on peut en gérer !
13            System.out.println("Sorry, I don't know how to manage more than 2 names!");
14    }
15 }
```

LES STRUCTURES DE CONTROLES : CONTINUE / BREAK

continue

```
1 for ( int i=0; i <10; i++) {  
2 // déclarations exécutées à chaque itération  
3     if(i == 2 ||i == 5) {  
4         continue;  
5     }  
6 System.out.println("Valeur de i : " + i + ".");  
7 }
```

break

```
1 int [] myArray = { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 };  
2  
3 for (int i =0; i<myArray.length;i++) {  
4     if (myArray[i] == "50") {  
5         System.out.println ("J'ai trouvé mon " +basket[i]+ " !");  
6         break;  
7     }  
8 System.out.println ("J'en suis à " +basket[i]+ " ...");  
9 }
```

LES STRUCTURES

For amélioré : For-each

Pour que cette expression compile, il faut que la variable désignant la collection à droite de : implémente le type Iterable ou qu'il s'agisse d'un tableau.

Il faut également que la variable à gauche de : soit compatible pour l'assignation d'un élément de la collection.

```
short arrayOfShort[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

for (int k : arrayOfShort) {
    System.out.println(k);
}
```

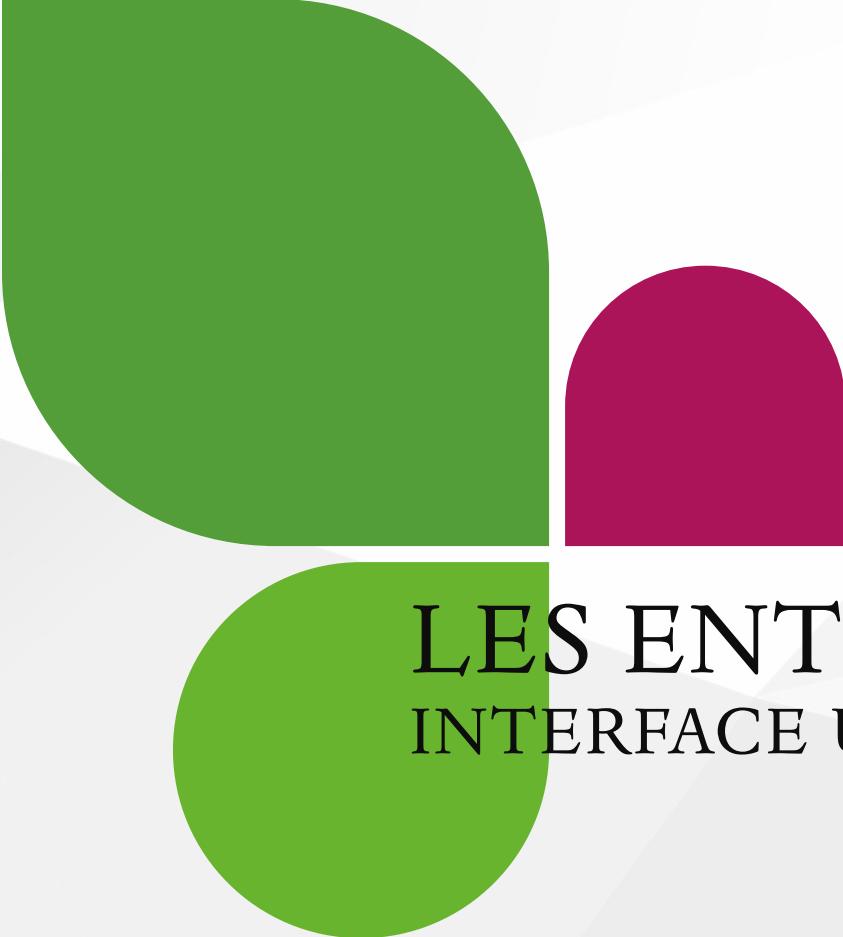
Libellé

Il est possible de mettre un libellé avant une expression for ou while. Par défaut, break et continue n'agissent que sur le bloc d'itération dans lequel ils apparaissent. En utilisant un libellé, on peut arrêter ou continuer sur une itération de niveau supérieur :

```
int m = 0;

boucleDeCalcul:
for (int i = 0; i < 10; ++i) {
    for (int k = 0; k < 10; ++k) {
        m += i * k;
        if (m > 500) {
            break boucleDeCalcul;
        }
    }
}

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



LES ENTRÉES ET SORTIES INTERFACE UTILISATEURS

LA CLASSE SYSTEM

Cette classe permet l'interfaçage entre votre programme Java et le système d'exploitation utilisé.

Vous pouvez notamment y manipuler les principaux flux de caractères de votre programme :

- `System.in`,
- `System.out`
- `System.err`
- les propriétés d'environnement de votre programme
- le temps système (en millisecondes et microsecondes)
- ...

Grâce à cette classe, nous allons pouvoir commencer à mettre en place une interaction avec nos applications, certes console mais utiles.

- Avec `System.out`, nous allons pouvoir afficher des messages
- Avec `System.in`, nous allons pouvoir récupérer des informations saisies par l'utilisateur. Pour cela, il faut utiliser la classe `Scanner` qui permet de lire la saisie de notre clavier.

```
public class TestScanner {  
  
    public static void main(String[] args) {  
        Scanner scanner = new Scanner(System.in);  
  
        System.out.print("Saisissez une chaîne de caractères : ");  
        String chaine = scanner.nextLine();  
  
        System.out.print("Saisissez un nombre : ");  
        int nombre = scanner.nextInt();  
  
        System.out.print("Saisissez les 8 caractères de votre identifiant : ");  
        String identifiant = scanner.next(".{8}");  
  
        System.out.println("Vous avez saisi :");  
        System.out.println(chaine);  
        System.out.println(nombre);  
        System.out.println(identifiant);  
    }  
}
```



LES TABLEAUX CHOUETTE DES TABLEAUX !!

LES TABLEAUX

Les tableaux à 1 dimension

```
public class Tableau {

    // déclaration de tableaux
    int tabInt[] = new int [10];

    /* autre écriture :
     * int [] tabInt = new int [10];
     */

    int [] tab1, tab2;
    int [] tab = { 1, 2, 3};

    // affectation d'une valeur dans tabInt
    tabInt[indice] = valeur;

    // récupération d'une valeur de tab
    nb = tab[indice];

    // récupération de la taille du tableau
    taille = tab.length;

}
```

Les tableaux à 2 dimensions

```
public class Matrice {

    // déclaration d'une matrice
    int mat [][] = new int[m][n];

    // remplissage d'une matrice
    int[][] mat = {
        {23,98,89},
        {34,34,76},
        {34,34,90}
    };

    // parcours d'une matrice
    for (int i = 0; i < mat.length; i++) {
        for (int j = 0; j < mat.length; j++) {
            System.out.print(mat[i][j] + "\t");
        }
        System.out.println();
    }
}
```

LES TABLEAUX

Quelques exemples.

```
int[] tableau = {1, 2, 3, 4, 5};

int premierElement = tableau[0];
int dernierElement = tableau[tableau.length - 1];

System.out.println(premierElement); // 1
System.out.println(dernierElement); // 5

for (int i = 0, j = tableau.length - 1; i < j; ++i, --j) {
    int tmp = tableau[j];
    tableau[j] = tableau[i];
    tableau[i] = tmp;
}
```

```
int[] tableau = {1, 2, 3, 4, 5};

for (int v : tableau) {
    System.out.println(v);
}
```

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

int[][][] tableauTroisDimensions = {{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}};

System.out.println(tableauDeuxDimensions[0][1]);
System.out.println(tableauTroisDimensions[0][1][0]);
```

LES TABLEAUX

Egalité de deux tableaux

En Java, il n'est pas possible d'utiliser l'opérateur `==` pour comparer deux objets.

- En effet, cet opérateur compare la référence des variables.
- Cela signifie qu'il indique `true` uniquement si les deux variables référencent le même objet.

Pour comparer deux objets, il faut utiliser la méthode `equals`. La classe outil `java.util.Arrays` fournit des méthodes de classe `equals` pour comparer des tableaux en comparant un à un leurs éléments.

```
int[] tableau = {1, 5, 4, 3, 2};  
java.util.Arrays.sort(tableau);  
System.out.println(java.util.Arrays.toString(tableau));
```

```
String[] tableau = {"premier", "deuxième", "troisième", "quatrième"};  
java.util.Arrays.sort(tableau);  
System.out.println(java.util.Arrays.toString(tableau));
```

1

2

Tri & recherche

Le tri et la recherche sont des opérations courantes sur des tableaux de valeurs. La classe outil `java.util.Arrays` offrent un ensemble de méthodes de classe pour nous aider dans ces opérations.

1. Tout d'abord, `java.util.Arrays` fournit plusieurs méthodes `sort`. Celles prenant un tableau de primitives en paramètre trient selon l'ordre naturel des éléments.
2. Il est également possible de trier certains tableaux d'objets. Par exemple, il est possible de trier des tableaux de chaînes de caractères.

COPIE DE TABLEAUX

Comme il n'est pas possible de modifier la taille d'un tableau, la copie peut s'avérer une opération utile.

`java.util.Arrays` fournit des méthodes de classe `copyOf` et `copyOfRange` pour réaliser des copies de tableaux.

```
int[] tableau = {1, 2, 3, 4, 5};

int[] nouveauTableau = java.util.Arrays.copyOf(tableau, tableau.length - 1);
System.out.println(java.util.Arrays.toString(nouveauTableau)); // [1, 2, 3, 4]

nouveauTableau = java.util.Arrays.copyOf(tableau, tableau.length + 1);
System.out.println(java.util.Arrays.toString(nouveauTableau)); // [1, 2, 3, 4, 5, 0]

nouveauTableau = java.util.Arrays.copyOfRange(tableau, 2, tableau.length);
System.out.println(java.util.Arrays.toString(nouveauTableau)); // [3, 4, 5]
```

CONVERSION DE TABLEAUX

La plupart des API Java utilisent des collections plutôt que des tableaux. Pour transformer un tableau d'objets en liste, on utilise la méthode `java.util.Arrays.asList`.

La liste obtenue possède une taille fixe.

Par contre le contenu de la liste est modifiable, et toute modification des éléments de cette liste sera répercutee sur le tableau.

```
String[] tableau = {"Bonjour", "le", "monde"};
java.util.List<String> liste = java.util.Arrays.asList(tableau);

liste.set(0, "Hello");
liste.set(1, "the");
liste.set(2, "world");

// Le tableau a été modifié à travers la liste
System.out.println(java.util.Arrays.toString(tableau)); // [Hello, the, world]
```

JAVA.UTIL.ARRAYS

Les méthodes de la classe Arrays sont des méthodes statiques, ce qui signifie que vous les utilisez avec le nom de la classe sans instancier un objet Arrays.

Méthode	Description
static int binarySearch(type[] a, type key)	Rechercher dans le tableau spécifié la valeur de key spécifiée à l'aide de l'algorithme de recherche binaire.
static boolean equals(type[] a, type[] a2)	Renvoyer true si les deux tableaux du même type spécifiés sont égaux
static void fill(type[] a, type val)	Affecter la valeur spécifiée à chaque élément du tableau spécifié
static void sort(type[] a)	Trier le tableau spécifié en ordre croissant
static void sort(type[] a, int fromIndex, int toIndex)	Trier la plage spécifiée du tableau en ordre croissant
static void parallelSort(type[] a)	Trier le tableau spécifié en ordre croissant à l'aide d'un tri en parallèle.
static type[] copyOf(type[] originalArray, int newLength)	Copier le tableau spécifié, en tronquant ou en remplaçant avec la valeur par défaut (si nécessaire) afin que la copie ait la longueur spécifiée.
static String toString(type[] originalArray)	renvoyer une représentation sous forme de chaîne du contenu du tableau spécifié.
Etc...	



LES ÉNUMÉRATIONS UN PEU, BEAUCOUP, ...

LES ÉNUMÉRATIONS

Très utile de pouvoir représenter des listes finies d'éléments.

```
// Utilisation
Level myVar = Level.MEDIUM;

switch(myVar) {
    case LOW:
        System.out.println("Low level");
        break;
    case MEDIUM:
        System.out.println("Medium level");
        break;
    case HIGH:
        System.out.println("High level");
        break;
}
```

```
// déclaration d'une matrice
enum Level {
    LOW,
    MEDIUM,
    HIGH
}
```

CLASSE ENUM

Méthodes

`valueOf(String)` : permet de convertir une chaîne de caractères en une énumération.

```
Level level = Level.valueOf("HIGH");
```

`Values` : retourne un tableau contenant tous les éléments de l'énumération dans l'ordre de leur déclaration.

```
for(Level l : Level.values()) {  
    System.out.println(l);  
}
```

Méthodes

`Name` : retourne le nom de l'élément sous forme d'une chaîne de caractères.

```
String state = Level.HIGH.name(); // "HIGH"
```

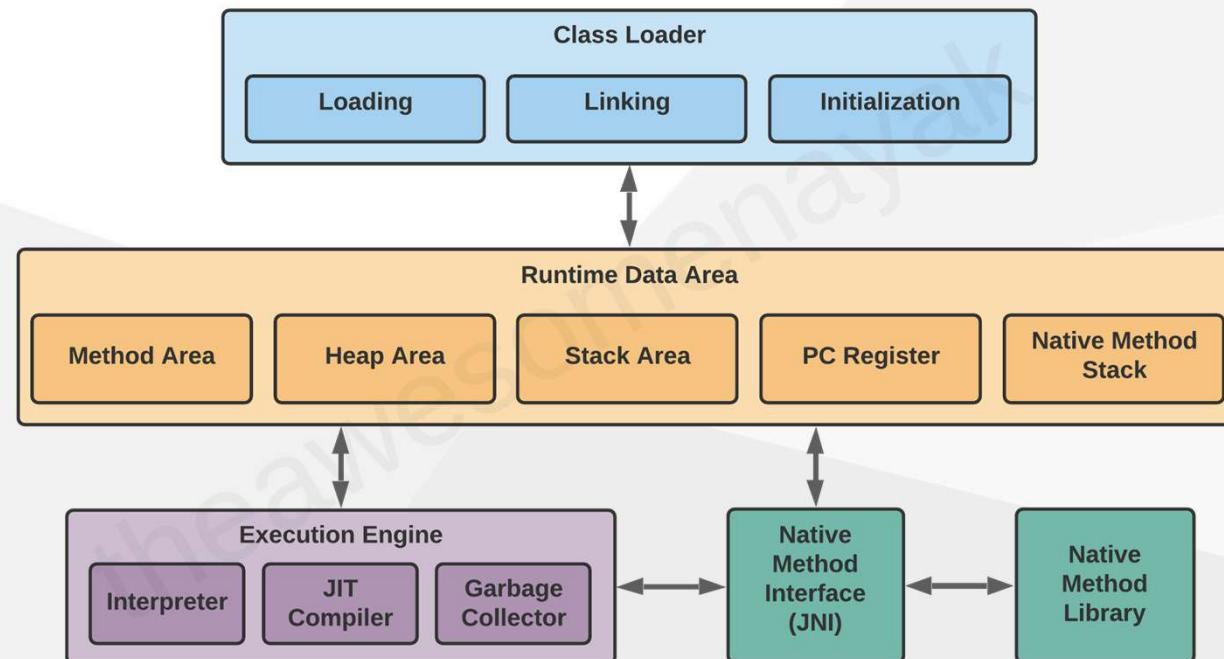
`Ordinal` : retourne le numéro d'ordre d'un élément (1^{er} élément à 0)

```
Int ordre = Level.HIGH.ordinal(); // 2
```



LA MÉMOIRE DE LA JVN DESCRIPTION ET ORGANISATION

SCHÉMA DE LA MÉMOIRE DE LA JVM



COMPOSITION DE LA MÉMOIRE.

* Un thread est une unité d'exécution faisant partie d'un programme.

La mémoire de la JVM

Plusieurs zones de mémoire sont utilisées par la JVM :

- les registres (register)
 - ces zones de mémoires sont utilisées par la JVM exclusivement lors de l'exécution des instructions du byte code.
- une ou plusieurs piles ([stack](#))
- un tas ([heap](#))
- une zone de méthodes ([method area](#))

Description

Chaque thread* possède sa propre pile

- Variables locales, paramètres, valeur de retours
- Seules des données de types primitif et des références à des objets peuvent être stockés.

Le tas est partagé par tous les threads.

- Elle stocke toutes les instances des objets créés.
- Les tableaux sont par exemple stockés dans cette zone mémoire.

Method Area est partagé par tous les threads.

- Elle stocke les définitions des classes et interfaces, des constructeurs et des méthodes, les constantes, les variables de classes ...

LE RAMASSE-MIETTES

Le ramasse-miettes (garbage collector) est un processus léger (thread) qui est créé par la JVM et qui s'exécute régulièrement pour contrôler l'état de la mémoire.

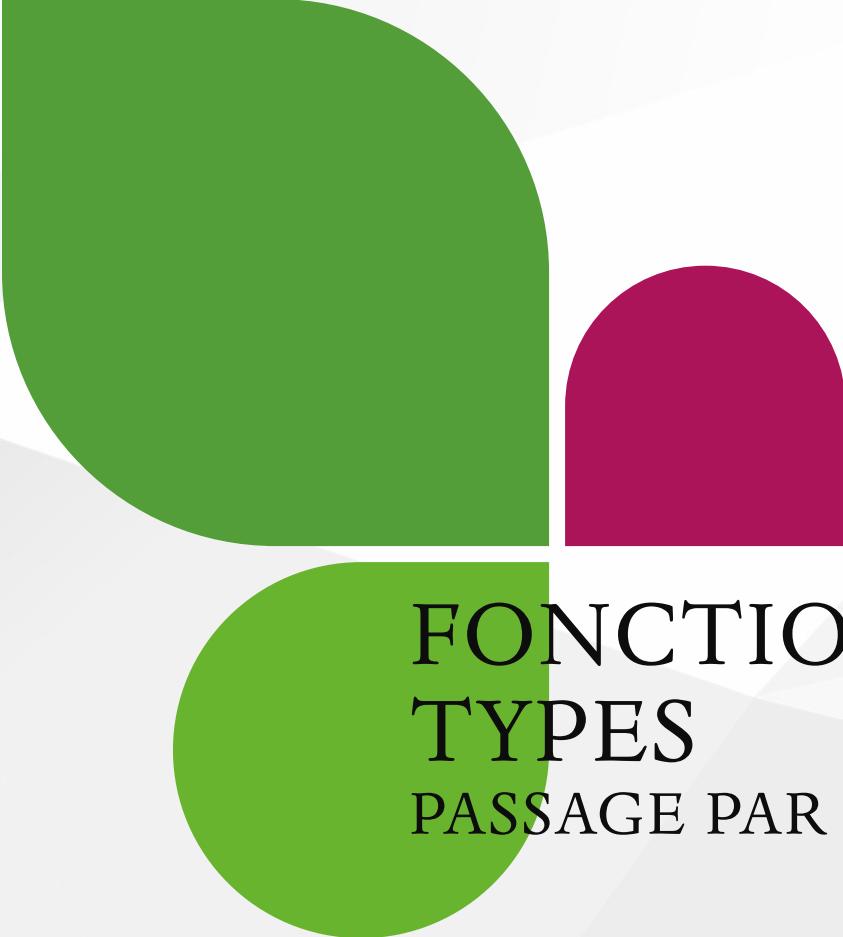
S'il détecte que des portions de mémoire allouées ne sont plus utilisées, il les libère afin que l'application ne manque pas de ressource mémoire.

Il permet :

- d'éviter de devoir demander explicitement la libération de la mémoire.
- de vérifier périodiquement si les objets sont référencés.

Si un développeur souhaite qu'un objet soit détruit et son espace mémoire récupéré, il doit s'assurer que plus aucune référence n'existe vers cet objet.

- Il suffit d'affecter la valeur `null` aux variables et aux attributs qui réfèrent cet objet.

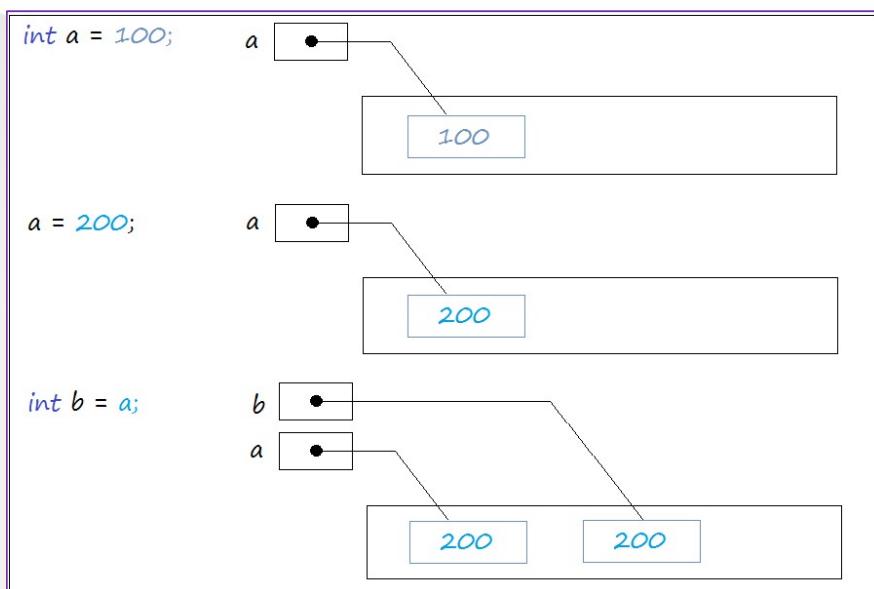


FONCTIONNEMENT DES TYPES

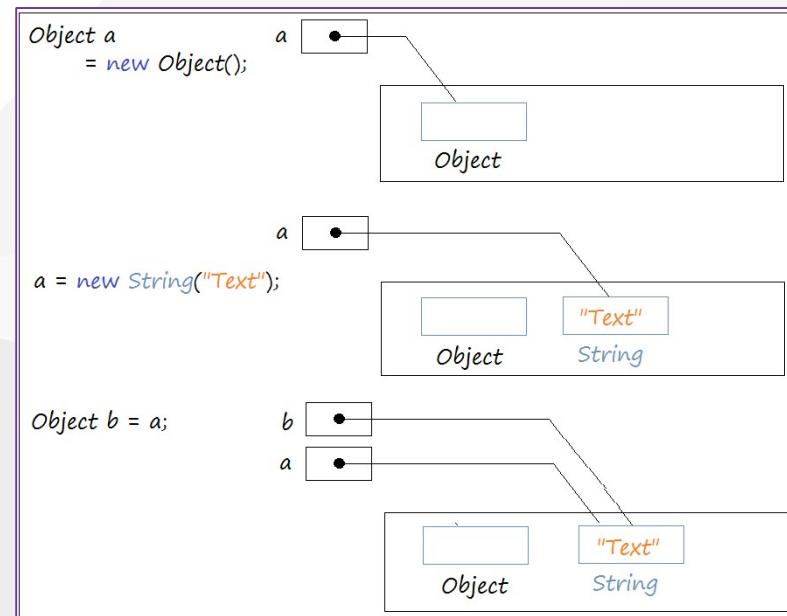
PASSAGE PAR TYPE OU RÉFÉRENCE

FONCTIONNEMENT DES TYPES

Type primitif

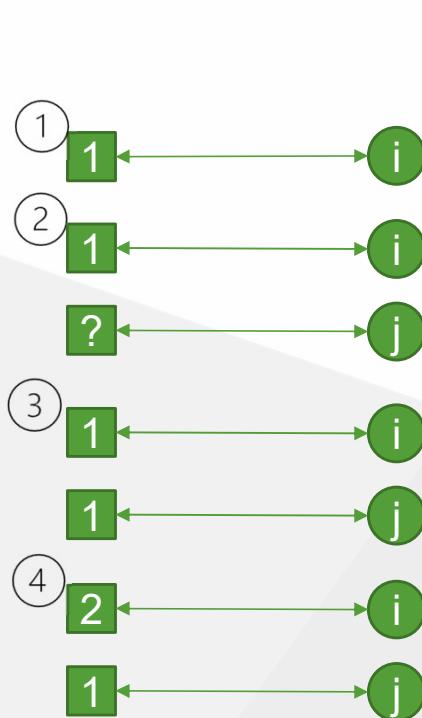


Type référence (les objets)



PASSAGE PAR VALEUR

```
int i = 1;    ①
int j;        ②
j = i;        ③
i = 2;        ④
```



1. Création et initialisation d'une variable i
 - Un espace mémoire du type demandé est associé à i où est encodé l'entier 1.
 2. Création d'une variable j
 - Un espace mémoire du type demandé est associé à j.
 3. Affectation de la valeur de i à j
 - Le bout de l'espace mémoire associé à i est recopié dans celui de j
 4. Affectation de la valeur 2 à i
 - i prends la valeur 2
 - j ne bouge pas.
- Conclusion : Il n'y a pas de lien entre i et j

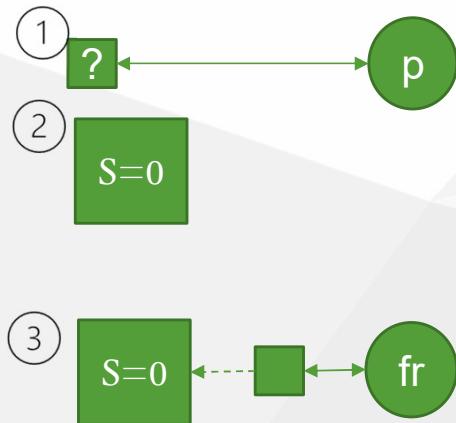
PASSAGE PAR RÉFÉRENCE

Pays p ①

new Pays() ②

Pays fr = new Pays(); ③

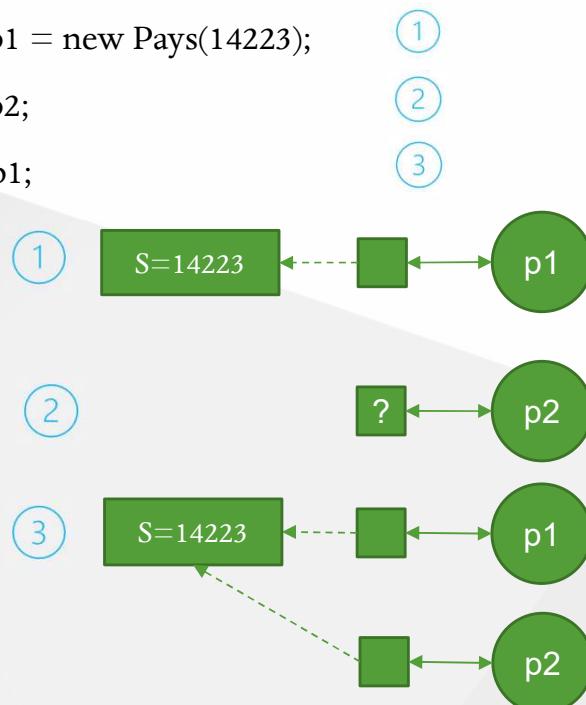
```
class Pays {
    public double superficie;
    public Pays() {
        superficie = 0;
    }
    public Pays(double superficie) {
        this.superficie = superficie;
    }
}
```



1. Aucun objet (aucune instance) de la classe Pays n'est créé
 - L'espace mémoire nécessaire à stocker un Pays n'est même pas réservé.
2. Instanciation d'un objet. Il se passe 2 choses :
 - Un espace mémoire est alloué permettant de stocker un objet de type Pays
 - Un constructeur est appelé pour initialiser l'objet.
3. Cette fois-ci, 3 étapes
 - Déclaration d'une référence
 - Création de l'objet à l'aide de new
 - Affectation de la référence vers l'emplacement mémoire de l'objet créé.
 - **On dit que l'objet est référencé par la variable fr**

AFFECTATION DE RÉFÉRENCES

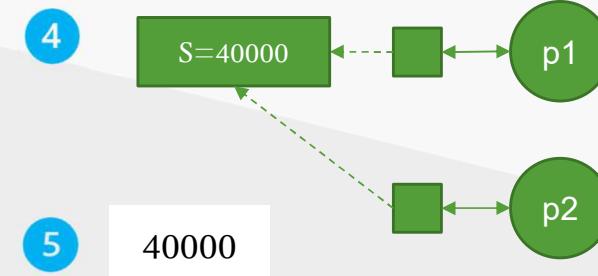
```
Pays p1 = new Pays(14223);  
Pays p2;  
p2 = p1;
```



- Création d'une instance Pays référencé par p1 et initialisée par le constructeur.
- Déclaration d'une variable p2 avec réservation d'un espace mémoire sans affectation.
- La valeur de la référence de p1 est recopiée dans l'espace mémoire correspondant à p2.

Conclusion : 1 seul objet a été créé et cet objet est référencé par les deux variables p1 et p2 et non une copie d'objet.

- `p2.surface = 40000;`
- `System.out.println(p1.surface);`





LES CHAÎNES DE CARACTÈRES

LA CLASSE STRING

LA CLASSE STRING

[HTTPS://DOCS.ORACLE.COM/E
N/JAVA/JAVASE/17/DOCS/API/J
AVA.BASE/JAVA/LANG/STRING.
HTML](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/String.html)

La classe string est une classe fondamentale de JAVA permettant de gérer les chaînes de caractères. Un objet String est un tableau de char.

String txt1 = "bonjour";

String texte = new String("bonjour");

Opérateur de concaténation : +

Méthodes de la classe String : replace(), replaceFirst() et
replaceAll(),
toUpperCase(), toLowerCase(), trim() etc...

Propriétés de la classe String : length

Égalité : equals("texte")

LA CLASSE STRING EN MÉMOIRE

String Pool

La String Pool est la zone spéciale de la mémoire pour stocker les objets de type String.

Si nous utilisons des guillemets pour créer un objet String, il est stocké dans le String Pool.

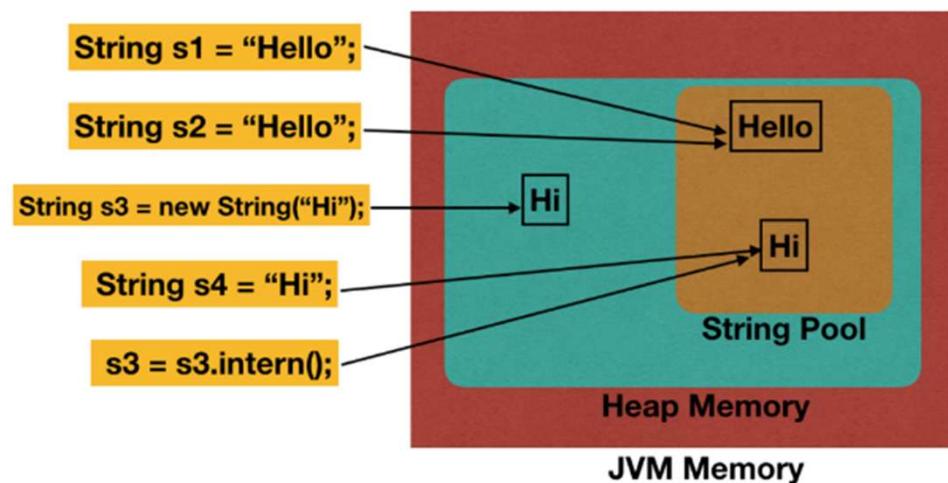
Si nous utilisons l'opérateur new pour créer une String, celle-ci est créée dans le tas (Heap Memory).

Comment fonctionne la String Pool

- Lorsque nous créons une String, il est stocké dans la String Pool.
- S'il existe déjà une String avec la même valeur dans la String Pool, le nouvel objet n'est pas créé. La référence à l'objet est renvoyée.
- La String Pool est un cache d'objet String car la String est immuable (n'est pas sujet au changement).
- Si j'utilise new String, l'objet String est créé dans le tas. On peut le déplacer dans la String Pool avec la méthode intern()

EXEMPLE

1. Lorsque nous créons la première chaîne s1, il n'y a pas de chaîne avec la valeur "Hello" dans le pool de chaînes. Ainsi, une chaîne "Hello" est créée dans le pool et sa référence est affectée à s1.
2. Lorsque nous créons la deuxième chaîne s2, il existe déjà une chaîne avec la valeur "Hello" dans le pool de chaînes. La référence de chaîne existante est affectée à s2.
3. Lorsque nous créons la troisième chaîne s3, elle est créée dans la zone de tas car nous utilisons le nouvel opérateur.
4. Lorsque nous créons la quatrième chaîne s4, une nouvelle chaîne avec la valeur "Hi" est créée dans le pool et sa référence est affectée à s4.
5. Lorsque nous comparons s1 == s2, il revient true car les deux variables font référence au même objet chaîne.
6. Lorsque nous comparons s3 == s4, il renvoie false car ils pointent vers les différents objets de chaîne.
7. Lorsque nous appelons la intern() sur s3, elle vérifie s'il existe une chaîne dans le pool avec la valeur "Hi" ? Comme nous avons déjà une chaîne avec cette valeur dans le pool, sa référence est renvoyée et affectée à s3.
8. s3 == s4 Renvoie maintenant true car les deux variables font référence aux mêmes objets de chaîne.



AVANTAGES DE LA STRING POOL

1. La String Pool permet la mise en cache des objets String. Cela économise beaucoup de mémoire pour la JVM qui peut être utilisée par d'autres objets.
2. La String Pool contribue à de meilleures performances de l'application en raison de la réutilisabilité. Cela permet de gagner du temps pour créer une nouvelle chaîne s'il existe déjà une chaîne présente dans le pool avec la même valeur.



STRINGBUILDER ET STRINGBUFFER

UNE AUTRE MANIÈRE DE CRÉER DES CHAÎNES DE CARACTÈRES.

STRINGBUFFER ET STRINGBUILDER

StringBuffer

La classe Java StringBuffer est utilisée pour créer une chaîne modifiable. La classe StringBuffer en Java est identique à la classe String, sauf qu'elle peut être modifiée.

Remarque ! La classe Java StringBuffer est sécurisée pour les threads, c'est-à-dire que plusieurs threads ne peuvent pas y accéder simultanément. Donc, il est sûr

StringBuilder

StringBuilder en Java représente une séquence de caractères mutable.

Étant donné que la classe String en Java crée une séquence de caractères immuable, la classe StringBuilder fournit une alternative à la classe String, car elle crée une séquence mutable de caractères.

STRINGBUFFER

Résultat du code :

Taille : 26

Capacité : 42

String après append : Developpement Informatique.com
String après insert : Developpement- Informatique.com
String après reverser : moc.euqitamrofnI -tnemeppoleveD
String après delete(0, 5) : uqitamrofnI -tnemeppoleveD
String après deleteCharAt(7) : uqitamrfnI -tnemeppoleveD
String après replace : MeknesmrfnI -tnemeppoleveD

```
public class Test {  
  
    public static void main(String args[]) {  
        StringBuffer s = new StringBuffer("Developpement Informatique");  
  
        System.out.println("Taille : " + s.length());  
  
        System.out.println("Capacité : " + s.capacity());  
        s.append(".com");  
  
        System.out.println("String après append : " + s);  
  
        s.insert(13, "-");  
  
        System.out.println("String après insert : " + s);  
  
        s.reverse();  
  
        System.out.println("String après reverser : " + s);  
        s.delete(0, 5);  
        System.out.println("String après delete(0, 5) : " + s);  
        s.deleteCharAt(7);  
        System.out.println("String après deleteCharAt(7) : " + s);  
        s.replace(0, 5, "Meknes");  
        System.out.println("String après replace : " + s);  
    }  
}
```

STRINGBUILDER

Résultat du code :

String = AAAABBCCCC

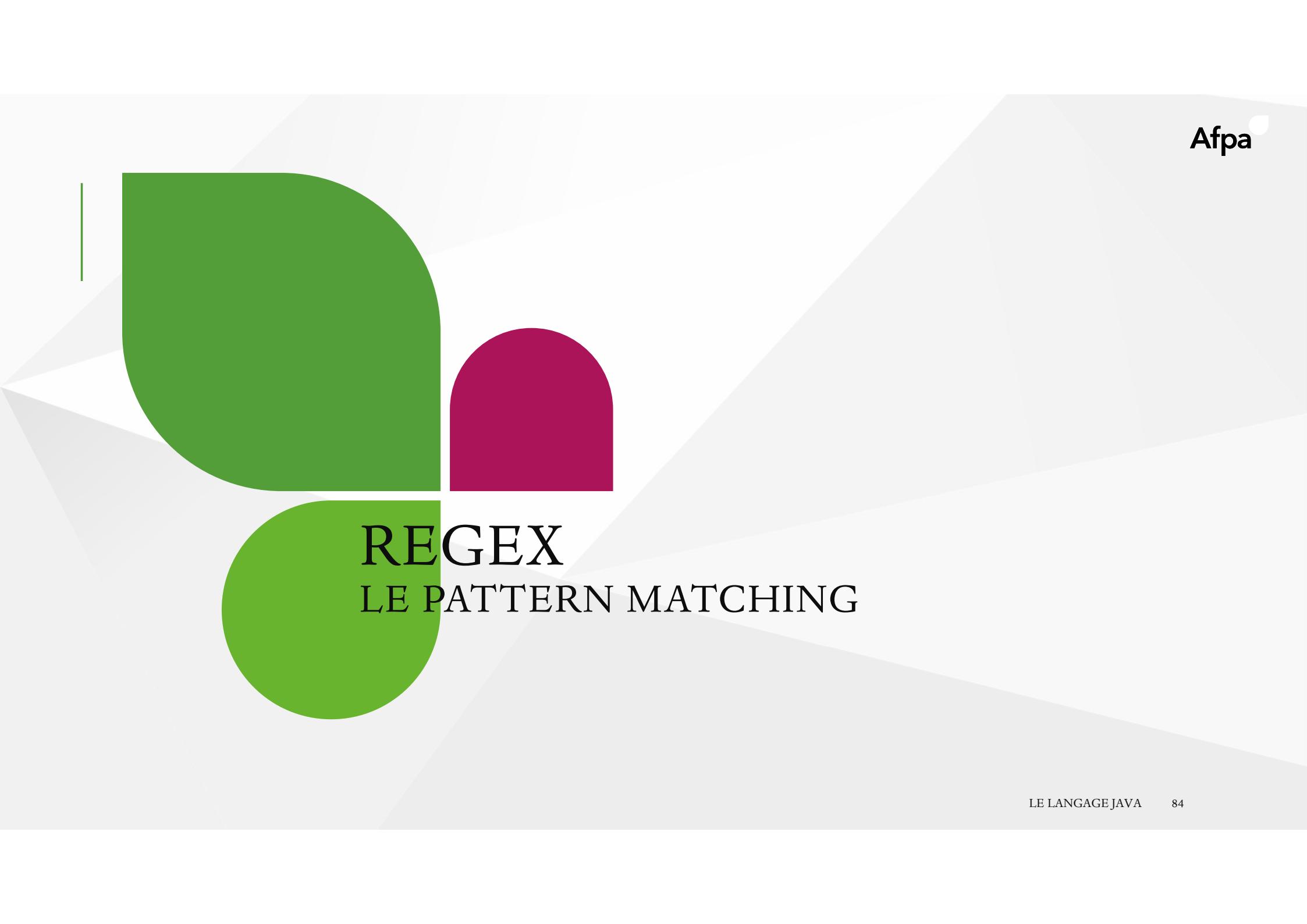
Reverse String = CCCCCBBBAAAA

Modified StringBuilder = CCCCCBBBAAAA,

StringBuilder = CCCCCBBBAAAA,

Capacity of StringBuilder = 27

```
public class test {  
  
    public static void main(String[] argv)  
    {  
        // create a StringBuilder object with a String pass as parameter  
        StringBuilder str = new StringBuilder("AAAABBCCCC");  
        // print strin  
        System.out.println("String = " + str.toString());  
  
        // reverse the string  
        StringBuilder reverseStr = str.reverse();  
  
        // print string  
        System.out.println("Reverse String = " + reverseStr.toString());  
  
        // Append ', '(44) to the String  
        str.appendCodePoint(44);  
  
        // Print the modified String  
        System.out.println("Modified StringBuilder = " + str);  
  
        // get capacity  
        int capacity = str.capacity();  
  
        // print the result  
        System.out.println("StringBuilder = " + str);  
        System.out.println("Capacity of StringBuilder = " + capacity);  
    }  
}
```



REGEX

LE PATTERN MATCHING

LES EXPRESSIONS RÉGULIÈRES EN JAVA

`java.util.regex` : package pour la correspondance de motifs (Pattern matching) avec des expressions régulières.

Une expression régulière est une séquence spéciale de caractères qui vous aide à faire correspondre ou à trouver d'autres chaînes ou ensembles de chaînes, en utilisant une syntaxe spécialisée conservée dans un motif.

Elles peuvent être utilisées pour rechercher, éditer ou manipuler du texte et des données.

Expressions régulières :

<https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html>

La-puissance-des-regex : <https://buzut.net/la-puissance-des-regex/>

REGEX

Avec les REGEX, nous allons pouvoir contrôler les saisies des utilisateurs, effectuer des recherches de pattern rapidement, etc...

Certaines méthodes de la classe String acceptent comme paramètres des "motif"

Par exemple la méthode `String.matches` prend en paramètre de type String, une expression régulière.

Quelques exemples :

```
boolean match = "hello".matches("hello");
System.out.println(match); // true
```

```
String s = "hello";
System.out.println(s.matches("."*)); // true
System.out.println(s.matches("."+)); // true
System.out.println(s.matches("X?hel+oW?")); // true
System.out.println(s.matches("."+l{2}o")); // true
System.out.println(s.matches("[eh]{0,2}l{1,100}o")); // true
```

```
String s = "hello";
System.out.println(s.replaceAll("[aeiouy]", "\^_\^")); // h^\_ll^\_
```

```
String s = "hello the world";
// ["hello", "the", "world"]
String[] tab = s.split("\W");

// ["hello", "world"]
tab = s.split(" the ");

// ["he", "", "", "the w", "r", "d"]
tab = s.split("[ol]");
```

EXEMPLE

Quelques exemples d'utilisation

```
// CONSTANTES
final String REGEXTELINT = "^\\+(:[0-9] ?){6,14}[0-9]$";
final String REGEXTELFR = "^(0|\\+33|0033)[1-9][0-9]{8}$";
final String REGEXEMAIL = "[\\w._-]+@[\\w._-]+\\.\\.[a-z]{2,}$";

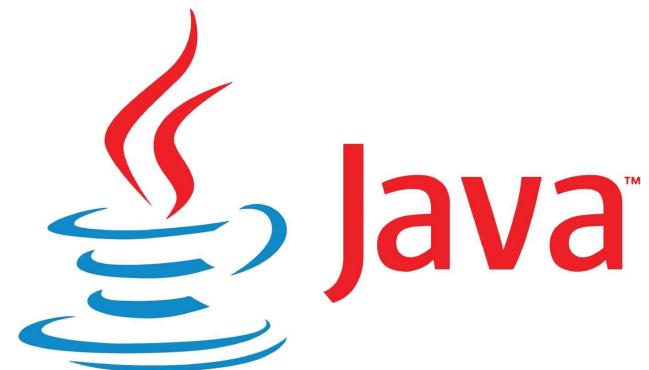
String phone = "8930";
String email = "test@test.fr";

// création de mon pattern
Pattern pattern = Pattern.compile("afpa", Pattern.CASE_INSENSITIVE);
Matcher matcher = pattern.matcher("Visite l'afpA !");
// recherche de mon pattern à partir de mon matcher
boolean matchFound = matcher.find();

// test
if (matchFound) {
    System.out.println("afpa trouvé");
} else {
    System.out.println("afpa non trouvé");
}

pattern = Pattern.compile(REGEXEMAIL, Pattern.CASE_INSENSITIVE);
matcher = pattern.matcher(email);
if (!matcher.find()) {
    System.out.println("ça matche pas !");
} else {
    System.out.println("ça matche !");
}

// autre méthode
if (!phone.matches(REGEXTELFR) && !phone.matches(REGEXTELINT)) {
    System.out.println("ça matche pas !");
} else {
    System.out.println("ça matche !");
}
```



Tout est objet !!

PROGRAMMATION ORIENTÉE OBJET

POO / OPP
TOUT EST OBJET !!

LES PILIERS DE LA POO

La programmation par objet est un paradigme de programmation informatique.

Elle consiste en la définition et l'interaction de briques logicielles appelées objets

- un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre.

Pour faire simple, la POO est une approche de la programmation informatique et du traitement des solutions aux problèmes (paradigme) en considérant les éléments logiciels comme des objets.

Comme tout concept qui se respecte, la POO est basé sur des piliers qui renforcent son adoption et voici ces 5 piliers essentiels:

1. **Objet (et classe)** : Nous l'avons déjà abordé en début de Java. Une classe est un moule à partir duquel on va créer un objet. Un objet est une instance de la classe.
2. **Encapsulation** : Ce pilier est un mécanisme consistant à cacher les données et certaines implémentations de l'objet et à avoir accès aux données que par les services proposés par l'objet (méthodes dites publiques) *Garantir l'intégrité des données et masquer l'implémentation de certaines méthodes de la classe.*
3. **Héritage** : Ce principe consiste donc à créer une classe (classe fille) qui partagera les caractéristiques d'une autre classe (dite classe mère).
4. **L'abstraction** : permet à un programmeur de mieux concevoir en pensant en termes général plutôt qu'en termes spécifiques les différents comportements d'une classe.
5. **Polymorphisme** : *poly* qui signifie plusieurs et *morphe* qui signifie forme. En héritage, cela qui consiste à créer une méthode dans la classe mère qui sera polymorphe : *Ceci voudra dire que cette méthode aura donc plusieurs implementations en fonction des classes filles qui vont les implémenter.*

QU'EST-CE QU'UN OBJET ?

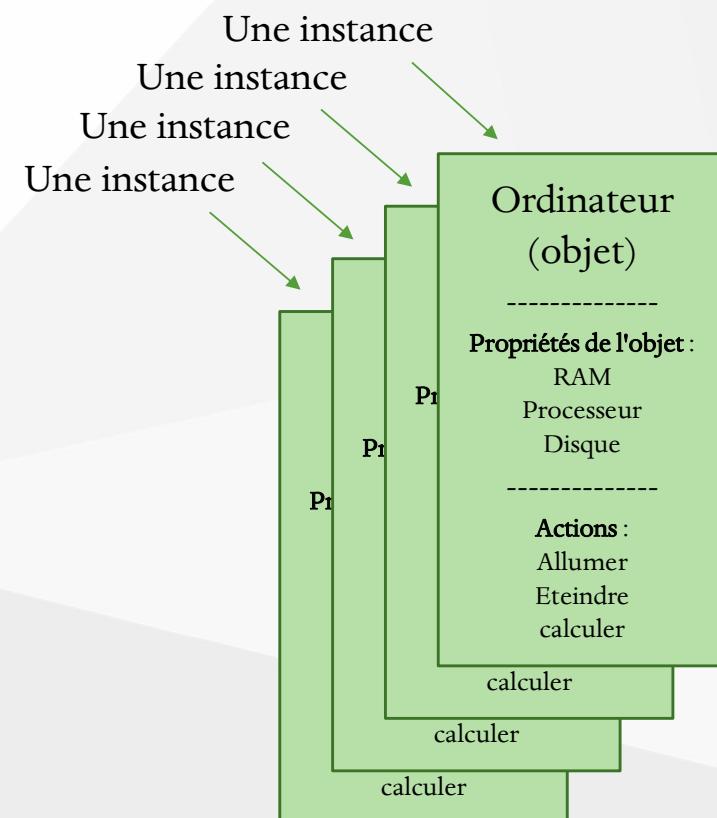
En POO, tout est objet et c'est pareil dans la vie réelle.

Par exemple - L'ordinateur est un objet :

- Il possède des propriétés, caractérisées sous forme de données. (RAM, processeur, disque etc..)
- Il peut réaliser des actions (s'allumer, s'éteindre, calculer, etc...)
- Il peut interagir avec d'autres objets. (un écran, un humain etc...)

La logique veut que l'on puisse avoir plusieurs objets, mais qui auront certainement des caractéristiques différentes.

- On parle alors d'**instance**.



COMPRENDRE LA NOTION DE CLASSE

En POO, un objet né de l'instance d'une classe.
(la base de la POO)

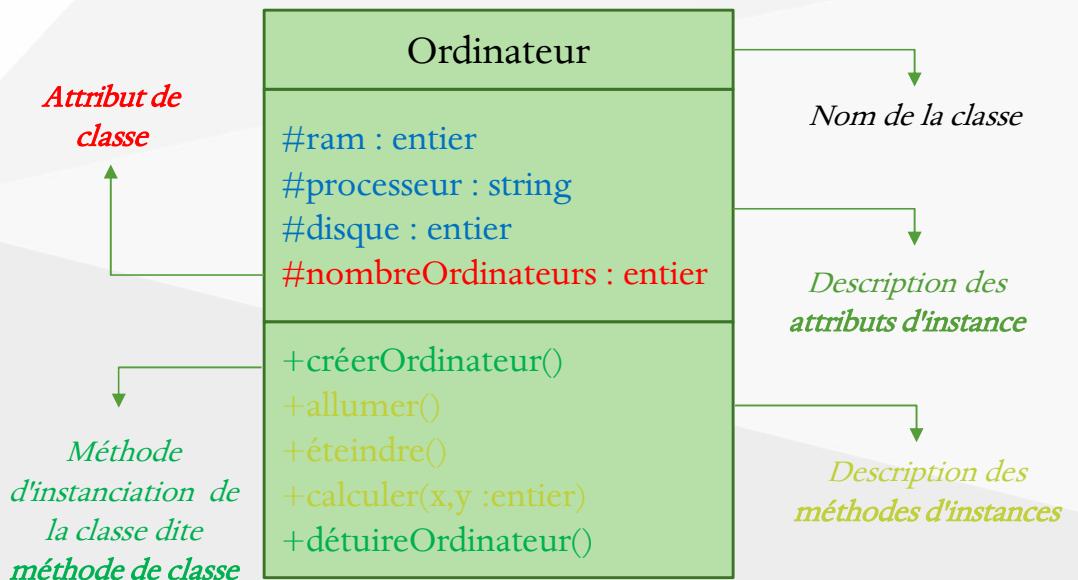
Une classe, c'est le schéma ou le plan qui nous permet d'instancier notre objet.

On peut instancier autant d'objets à partir d'une classe mais un objet généré ne peut changer de classe en cours de route.

En résumé, la classe est un catalogue de variables (propriétés) et d'actions internes (méthodes) qui interagissent entre eux pour donner un résultat.

Un objet instancié est mis en mémoire : il est référencé

Il est possible d'utiliser des propriétés dans vos méthodes et vous pouvez appeler des méthodes dans d'autres méthodes.



COMPRENDRE LA NOTION DE CLASSE

Attributs d'instance

Les attributs d'instance sont déclarés dans une classe, mais en dehors d'une méthode, d'un constructeur ou d'un bloc.

Les attributs d'instance sont créés lorsqu'un objet est instancié avec l'utilisation du mot-clé `new` et détruites lorsque l'objet est détruit.

Les attributs d'instance sont accessibles directement en appelant le nom de variable à l'intérieur de la classe. Dans les méthodes statiques, elles doivent être appelées en utilisant le nom complet : `Objet.VariableName`

Les attributs d'instance contiennent des valeurs qui doivent être référencées par plusieurs méthodes, constructeurs ou blocs, ou des parties essentielles de l'état d'un objet qui doivent être présentes dans toute la classe.

Attributs de classe

Les attributs de classe sont déclarés avec le mot clé `static` dans une classe mais en dehors d'une méthode, d'un constructeur ou d'un bloc.

Les attributs de classe sont créés au démarrage du programme et détruites à l'arrêt du programme.

Les attributs de classe sont accessibles en appelant avec le nom de classe.

- `ClassName.VariableName`

Il n'y aura qu'une seule copie de chaque attribut de classe par classe, quel que soit le nombre d'objets créés à partir de celle-ci.

COMPRENDRE LA NOTION DE CLASSE

Méthodes d'instance

Une méthode d'instance est une méthode associée à un objet dans une classe.

Par conséquent, les méthodes non statiques sont appelées à l'aide d'un objet de la classe dans laquelle la méthode est définie. Elle peut accéder aux membres non statiques ainsi qu'aux membres statiques d'une classe.

En Java, quand la méthode non statique est appelée, l'objet qui a invoqué la méthode est transmis en tant qu'argument implicite (la référence 'this'). Ainsi, dans la méthode, ce mot clé peut être utilisé pour faire référence à l'objet qui a appelé la méthode.

```
MyClass objMyClass = new MyClass();  
objMyClass.MyInstanceMethod();
```

Méthodes de classe

En POO, la méthode statique est une méthode associée à une classe.

Par conséquent, les méthodes statiques n'ont pas la capacité d'opérer sur une instance particulière d'une classe.

Les méthodes statiques peuvent être appelées sans utiliser d'objet de la classe contenant la méthode statique.

Le statique doit être utilisé lors de la définition d'une méthode statique en Java.

```
MyClass.MyStaticMethod();
```

EXEMPLE DE DÉCLARATION DE LA CLASSE ORDINATEUR

Ici, nous déclarons simplement notre classe.

```
public class Ordinateur {  
  
    public int ram;  
    public String processeur;  
    public int disque;  
    public int nombreOrdinateurs;  
  
    public void allumer() {  
  
    }  
  
    public void eteindre() {  
  
    }  
  
    public void calculer(int x, int y) {  
        System.out.println("Calcul en cours...");  
    }  
}
```

LE MOT CLÉ STATIC

UTILISATION ET UTILITÉ

LE MOT CLÉ STATIC

Le mot clé static en Java est utilisé pour indiquer que le membre d'une classe (variable, méthode, ou bloc) appartient à la classe elle-même, *plutôt qu'aux instances de cette classe.*

Examinons les différentes utilisations et les implications du mot clé static en Java :

1. Variables statiques :

- Une variable statique est partagée par toutes les instances de la classe.
- Elle est initialisée une seule fois, au moment où la classe est chargée en mémoire

2. Méthodes statiques :

- Une méthode statique peut être appelée sans créer une instance de la classe.
- Elle ne peut pas accéder directement aux membres non statiques (variables d'instance ou méthodes non statiques) de la classe.

3. Blocs statiques :

- Un bloc statique est utilisé pour initialiser les variables statiques.
- Il est exécuté une seule fois, lorsque la classe est chargée.

UTILITÉ DU MOT CLÉ STATIC

Tout au long de votre codage, le mot clé static peut avoir un rôle différent :

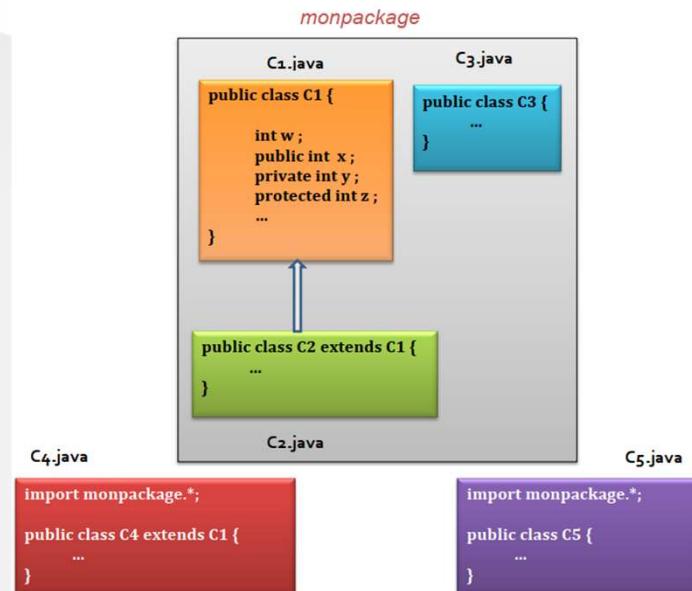
1. Partage de données :
 - Les variables statiques sont utilisées pour partager des données entre toutes les instances d'une classe. Par exemple, un compteur qui doit être commun à toutes les instances.
2. Méthodes utilitaires :
 - Les méthodes statiques sont souvent utilisées pour les fonctions utilitaires qui ne dépendent pas de l'état de l'instance. Par exemple, les méthodes de la classe Math comme Math.sqrt().
3. Initialisation :
 - Les blocs statiques sont utilisés pour l'initialisation complexe des variables statiques, là où une simple initialisation inline ne suffit pas.

NIVEAUX DE VISIBILITÉ

JE TE VOIS ! JE TE VOIS PLUS !

NIVEAUX DE VISIBILITÉ DE JAVA

- ✓ *C1, C2 et C3* sont 3 classes **publiques** appartenant à *monpackage*.
- ✓ Dans *monpackage*, la classe *C2* hérite de *C1*.
- ✓ Les classes *C4* et *C5* importent le package *monpackage* mais n'appartiennent pas à ce package.
- ✓ La classe *C4* hérite de *C1*.



Accès à	accès package	public	private	protected
w	oui	oui	oui	oui
C1	oui	oui	non	oui
C2	oui	oui	non	oui
C3	oui	oui	non	oui
C4	non	oui	non	oui
C5	non	oui	non	non

Le tableau résume les différents mode d'accès des membres d'une classe.

Modificateur du membre	private	aucun	protected	public
Accès depuis la classe	Oui	Oui	Oui	Oui
Accès depuis une classe du même package	Non	Oui	Oui	Oui
Accès depuis une sous-classe	Non	Non	Oui	Oui
Accès depuis toute autre classe	Non	Non	Non	Oui

NIVEAUX DE VISIBILITÉ DE JAVA

- **public** : accessible de n'importe où - La classe aura accès à ce membre mais aussi n'importe quelle autre classe.
 - *Par exemple, notre méthode main est public afin que la JVM puisse l'invoquer depuis l'extérieur de la classe car il n'est pas présent dans la classe courante.*
- **protected** : un membre marqué dans une classe comme protégé peut être manipulé :
 - Dans la classe qui définit ce membre
 - Dans les classes qui dérivent de la classe considérée.
 - Dans toutes les classes définies dans le même package que celle qui définit le membre protégé.
- **package private** : mode de visibilité par défaut, en l'absence de mot clé. Un membre sera considéré comme étant en "package private". Ce membre sera visible dans tout code dans le même package.
- **private** : Dans une classe, la déclaration d'un membre en private indiquera que seule la classe pourra y accéder.
- **final** : selon le contexte,
 - Pour une variable, indique que sa valeur ne peut être changée, une fois initialisée.
 - Pour une méthode, indique que la méthode ne pourra pas être redéfinie par une classe héritant de la classe ayant définie la méthode.
 - Pour une classe, il empêche la classe d'être héritée par une autre classe.

CONTRÔLE DE L'ACCÈS AUX PROPRIÉTÉS HÉRITÉS

private

Dans la classe de base, mettre **private** rendra **inaccessible** même pour les sous-classes.

Une classe dérivée n'a pas plus de priviléges du fait de l'héritage.

- *Par exemple, la classe mère définit ses setters en private pour interdire à la classe enfant de pouvoir redéfinir ses setters.*

protected

Protected appliqué à un attribut ou une méthode d'une classe mère permet de conserver une protection semblable à celle de **private**, tout en rendant possible l'accès à cet attribut ou cette méthode dans les classes filles ou les classes du même package.

ENCAPSULATION

CACHER L'IMPLÉMENTATION

LE PRINCIPE D'ENCAPSULATION

L'**encapsulation** est l'un des principes non négligeables de la POO.

Mais en quoi cela consiste ?

C'est relativement simple en fait : cela consiste à cacher le fonctionnement interne de votre objet en imposant à l'utilisateur de l'objet de passer par vos méthodes.

Cela permet notamment 2 choses :

- La sécurisation des données de votre objet, car l'utilisateur est obligé d'utiliser les méthodes que vous lui mettez à disposition
- Réaliser des traitements internes à l'objet sans que l'utilisateur ne le sache. Car oui certains traitements n'ont pas besoin d'être visibles pour l'utilisateur de l'objet, car soit cela n'a pas d'intérêt ou parce que tout simplement il s'en fout. C'est ce qu'on appelle **l'abstraction**.

MODIFICATION DE NOTRE CLASSE

- Nos attributs d'instance et de classe deviennent privés par le mot-clé **private**
- Du fait de la mise en place de l'encapsulation, la création de méthodes d'accès à nos attributs sont mis en place au travers **d'accesseurs** et de **mutateurs** (**accessors/mutators**) appelés **getter** et **setter**

Ce sont dans ces méthodes que nous metterons en place toute la logique de contrôle des données dit les tests métiers.

*Pour rappel, le mot-clé **this** fait référence à l'objet qui fait appel à la méthode.*

Donc, gardez cet automatisme : si un utilisateur veut changer un attribut de ma classe, il doit passer par une méthode.

```
public class Ordinateur {  
  
    private int ram;  
    private String processeur;  
    private int disque;  
    private static int nombreOrdinateurs;  
  
    public void allumer() {  
        this.changementBios();  
    }  
  
    private void eteindre() {}  
  
    private void changementBios() {  
        this.changementOS();  
    }  
  
    private void changementOS() {}  
  
    public int getRam() {  
        return ram;  
    }  
  
    public void setRam(int ram) {  
        this.ram = ram;  
    }  
  
    public String getProcesseur() {  
        return processeur;  
    }  
}
```

LE PRINCIPE D'ENCAPSULATION

GETTER - ACCESSEURS - ACCESSORS

En Java, le getter est une méthode utilisée pour protéger les données et faire en sorte de rendre le code sûr.

Le getter retourne une valeur du type de l'attribut concerné.

La convention veut que la méthode suive la syntaxe
`public type getNomAttribut()`

Cette méthode est public et l'attribut est private, forçant ainsi à passer par cette méthode d'accesseur pour accéder à l'attribut de l'instance.

SETTER - MUTATEURS - MUTATORS

En Java, le setter est une méthode utilisée pour protéger les données et faire en sorte de rendre le code sûr.

Le setter fixe et met à jour la valeur de l'attribut concerné. Il permet de mettre en place la logique de contrôle sur l'attribut.

La convention veut que la méthode suive la syntaxe
`public void setNomAttribut(paramètres..)`

Cette méthode est public et l'attribut private, forçant ainsi à passer par cette méthode mutateur pour fixer ou mettre à jour l'attribut de l'instance.

MODIFICATION DE NOTRE CLASSE

Le principe de l'abstraction est ici appliqué en indiquant certaines méthodes en **private**.

Ici, c'est la méthode public `allumer()` qui appelle la méthode private `chargementBios()` qui appelle elle même une méthode private `chargementOS()`

L'utilisateur se fiche complètement de l'ordre d'allumage, ni de ce qu'il se passe réellement : ce qui l'intéresse c'est le résultat.

```
public void setProcesseur(String processeur) {
    this.processeur = processeur;
}

public int getDisque() {
    return disque;
}

public void setDisque(int disque) {
    this.disque = disque;
}

public static int getNombreOrdinateurs() {
    return nombreOrdinateurs;
}

public static void setNombreOrdinateurs(int nombreOrdinateurs) {
    Ordinateur.nombreOrdinateurs = nombreOrdinateurs;
}
```

THIS

Le mot-clé **this** désigne dans une classe, **l'instance courante** de la classe elle-même.

Il est utilisé à différentes fins :

Rendre univoque :

- Rendre le code explicite et non ambigu
 - Exemple différence entre paramètre et attribut

S'auto-désigner comme référence :

- Fait référence à l'instance elle-même comme paramètre d'une méthode

Désigner l'instance de la classe qui encadre :

- Dans le cas de classes imbriquées, c'est-à-dire qu'une classe interne utilise l'instance de la classe externe, le mot-clé this préfixé du nom de la classe externe permet de désigner l'instance de la classe externe. S'il n'est pas préfixé, il désigne l'instance de la classe interne.

Appeler un autre constructeur de la classe :

- Un constructeur peut appeler un autre constructeur de la classe en utilisant le mot-clé this
 - `This("neo", "Moi");`

LES MÉTHODES

PRINCIPE ET CARACTÉRISTIQUES

SYNTAXE D'UNE MÉTHODE

```
// en-tête de la méthode
modificateur returnType nomMethode(ListeParamètres) {
    // corps ou contenu de la méthode
}
```

Chaque méthode doit inclure les deux parties décrites dans la syntaxe :

- L'en-tête de la méthode
- Le corps de la méthode

- **L'en-tête de la méthode** fournit des informations sur la manière dont les autres méthodes peuvent interagir avec elle. *Une en-tête de méthode est également appelé une déclaration.*
- Entre deux accolades, **le corps de la méthode** contient les instructions permettant d'effectuer le travail de la méthode : **son implémentation.**
 - Techniquement, une méthode n'est pas obligée de contenir des instructions dans son corps : **un stub.**
- **Modificateur** : le modificateur d'accès pour une méthode Java peut être l'un des mot-clés suivants : **public, private, protected** ou si non spécifié, package par défaut.
 - Le plus souvent, les méthodes sont accessibles en public.
- La **signature** d'une méthode est la combinaison du nom de la méthode et du nombre, des types et de l'ordre des arguments.
 - Par conséquent, vous pouvez dire qu'un appel de méthode doit correspondre à la signature de la méthode appelée.

MÉTHODES

STATIC

c'est un mot-clé qui, lorsqu'il est associé à une méthode, en fait une méthode liée à une classe.

La méthode `main()` est statique afin que JVM puisse l'invoquer sans instancier la classe.

Cela évite également le gaspillage de mémoire inutile qui aurait été utilisé par l'objet déclaré uniquement pour appeler la méthode `main()` par la JVM.

VOID

C'est un mot-clé utilisé pour spécifier qu'une méthode ne retourne rien.

Comme la méthode `main()` ne renvoie rien, son type de retour est void .

Dès que la méthode `main()` se termine, le programme java se termine également.

Par conséquent, cela n'a aucun sens de revenir de la méthode `main()` car JVM ne peut rien faire avec la valeur de retour de celle-ci.

MÉTHODES : DIFFÉRENCE ENTRE ARGUMENTS ET PARAMÈTRES

Arguments

- Lorsqu'une méthode est appelée, les valeurs transmises lors de l'appel sont appelées arguments.
- Ceux-ci sont utilisés dans l'instruction d'appel de méthode pour envoyer une valeur de la méthode appelante à la méthode appelée.
- Pendant l'appel, chaque argument est toujours attribué au paramètre dans la définition de la méthode.
- Ils sont aussi appelés paramètres réels.

Paramètres

- Les valeurs qui sont écrites lors de la définition de la méthode sont appelées paramètres.
- Ceux-ci sont utilisés dans l'en-tête de la méthode appelée pour recevoir la valeur des arguments.
- Les paramètres sont des variables locales auxquelles sont attribués les arguments lorsque la méthode est appelée.
- Ils sont aussi appelés paramètres formels.

PASSAGE DE PARAMÈTRES : TYPES PRIMITIFS

Considérons la fonction suivante et un appel de cette fonction :

```
static void augmenter(int i) {  
    i++;  
}  
  
public void test() {  
  
    int j=1;          // (1)  
    augmenter(j);    // (2)  
    System.out.println(j); // (3)  
}
```

1. Nous avons déjà vu ce que cela donne.

*Lors de l'appel de la fonction, le passage de paramètre s'effectue par valeur. Cela signifie qu'une variable *i* locale à la fonction est créée et que l'on recopie dans l'espace mémoire associé à cette variable la valeur associée à *j**

2. *i* est incrémenté de 1
3. On sort de la fonction. La variable locale est effacée et le résultat affiché est 1 (soit la valeur de la variable *j*).

Conclusion : si on passe un paramètre correspondant à un type primitif à une méthode, sa valeur ne peut pas être modifiée par la méthode.

PASSAGE DE PARAMÈTRES : TYPES RÉFÉRENCES

Prenons l'exemple ci-dessous :

```
public class Test {

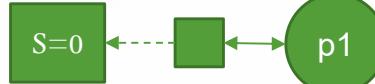
    class Pays {
        private double surface;

        public Pays() {
            this.surface = 0;
        }

        static void augmenterSurface(Pays p) {
            p.surface=p.surface + 1;          // 3
        }
    }

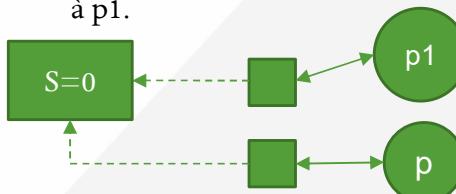
    Pays p1 = new Pays();                // 1
    augmenterSurface(p1);               // 2
    System.out.println(p1.surface);      // 4
}
```

- Création et instanciation de l'objet p1

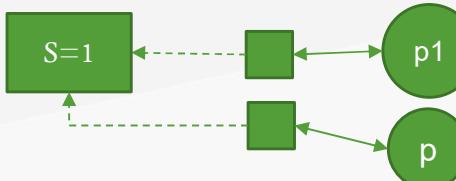


- Lors de l'appel de la méthode, le passage de paramètres s'effectue par valeur **mais cette fois-ci, c'est la référence qui est passée par valeur.**

- Une variable locale p est déclaré et la valeur de la référence correspond à p1.



- Dans la méthode, surface est augmentée de 1.



- A la sortie de la méthode, la variable locale est effacée et le résultat affiché est 1.

Conclusion : Si on passe un paramètre correspondant à une référence à une méthode, l'objet référencé peut être modifié par la méthode.

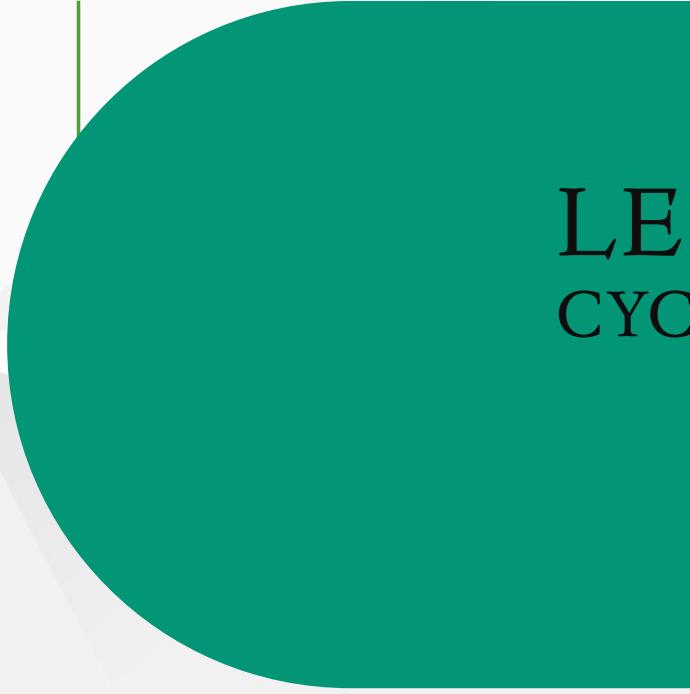
MÉTHODES : SURCHARGE

Principe

La surcharge permet à différentes méthodes d'avoir le même nom, mais des signatures différentes où la signature peut différer en fonction du nombre de paramètres d'entrée, du type de paramètres d'entrée ou des deux.

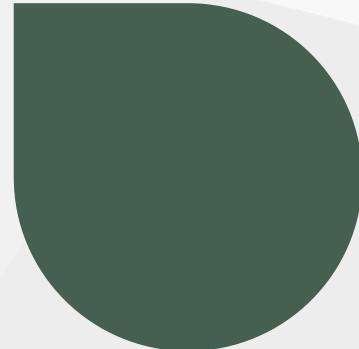
Quelques règles

1. **On ne peut pas surcharger par type de retour.**
2. **On peut surcharger des méthodes statiques.**
3. **On ne peut pas surcharger la méthode main.**



LES CONSTRUCTEURS

CYCLE DE VIE D'UN OBJET



RÔLE DU CONSTRUCTEUR

Il est possible de déclarer des méthodes particulières dans une classe que l'on nomme **constructeurs**.

Un constructeur a pour objectif d'initialiser un objet nouvellement créé afin de garantir qu'il est dans un état cohérent avant d'être utilisé.

Un constructeur a la signature suivante :

[portée] [nom de la classe]([paramètres]) {...}

Un constructeur se distingue d'une méthode car il n'a jamais de type de retour (pas même **void**). De plus un constructeur a obligatoirement **le même nom que la classe**.

Lorsqu'une voiture est créée par l'application avec l'opérateur new comme avec l'instruction suivante :

`Ordinateur ordi = new Ordinateur();`

Alors, la JVM crée l'espace mémoire nécessaire pour le nouvel objet de type Voiture, puis elle appelle le constructeur et enfin elle assigne la référence de l'objet à la variable voiture.

Donc le constructeur permet de réaliser une initialisation complète de l'objet selon les besoins des développeurs.

LES CONSTRUCTEURS

Chaque classe a un constructeur. Si nous n'écrivons pas explicitement un constructeur pour une classe, le compilateur Java construit **un constructeur par défaut** pour cette classe.

Chaque fois qu'un nouvel objet est créé, au moins un constructeur sera appelé. La règle principale de constructeurs, c'est qu'ils doivent avoir le même nom que la classe. Une classe peut avoir plusieurs constructeurs.

Par exemple, pour notre classe Ordinateur, le constructeur par défaut sera :

public Ordinateur() { }

Si votre classe ne contient qu'un seul constructeur sans paramètre dont le corps est vide, alors vous pouvez supprimer cette déclaration car le compilateur le générera automatiquement.

On peut avoir plusieurs constructeurs dans une même classe (**surcharge**).

*Public Ordinateur(int ram, String Processeur) { ...
}*

Pour rappel :

- *Déclaration - une déclaration de variable avec un nom et un type d'objet*
- *Instanciation - le mot-clé "new" est utilisé pour créer l'objet*
- *Initialisation - le mot-clé "new" suivi de l'appel au constructeur initialise le nouvel objet.*

AJOUT DU CONSTRUCTEUR À NOTRE CLASSE

Ajout de deux constructeurs symbolisant la surcharge des méthodes dans la classe.

Ici deux constructeurs ayant une signature différente.

A noter que les constructeurs ne retournent pas de type d'où la non-présence du mot-clé void

C'est également dans le constructeur que nous allons gérer notre attribut de classe nombreOrdinanteurs.

Il existe une méthode destroy() pour détruire un objet :

```
public void destroy() {  
    Ordinateur.nombreOrdinateurs--;  
}
```

```
public class Ordinateur {  
  
    private int ram;  
    private String processeur;  
    private int disque;  
    private static int nombreOrdinateurs;  
  
    public Ordinateur(int ram, String processeur, int disque) {  
        // constructeur de la classe  
        this.setRam(ram);  
        this.setProcesseur(processeur);  
        this.setDisque(disque);  
        // incrémentation du nom d'instance créée  
        Ordinateur.nombreOrdinateurs++;  
    }  
  
    public Ordinateur(int ram, String processeur) {  
        // constructeur de la classe  
        this.setRam(ram);  
        this.setProcesseur(processeur);  
        // incrémentation du nom d'instance créée  
        Ordinateur.nombreOrdinateurs++;  
    }  
}
```

LES CONSTRUCTEURS

Constructeur privé

Il est tout à fait possible d'interdire l'instantiation d'une classe en Java.

Pour cela, il suffit de déclarer tous ses constructeurs avec une portée private.

Un cas d'usage courant est la création d'une classe outil :

- Une classe outil ne contient que des méthodes de classe. Il n'y a donc aucun intérêt à instancier une telle classe. Donc, on déclare un constructeur privé pour éviter une utilisation incorrecte.

Appel d'un constructeur dans un constructeur

Certaines classes peuvent offrir différents constructeurs à ses utilisateurs.

Souvent ces constructeurs vont partiellement exécuter le même code.

Pour simplifier la lecture et éviter la duplication de code, un constructeur peut appeler un autre constructeur en utilisant le mot-clé `this` comme nom du constructeur.

Cependant, un constructeur ne peut appeler qu'un seul constructeur et, s'il le fait, cela doit être sa première instruction.

LES CONSTRUCTEURS

Appel d'une méthode dans un constructeur

Il est tout à fait possible d'appeler une méthode de l'objet dans un constructeur.

Cela est même très utile pour éviter la duplication de code et favoriser la réutilisation.

Attention cependant au statut particulier des constructeurs. Tant qu'un constructeur n'a pas achevé son exécution, l'objet n'est pas totalement initialisé.

La méthode finalize

Si un objet souhaite effectuer un traitement avant sa destruction, il peut implémenter la méthode finalize.

Cette méthode a la signature suivante :

```
protected void finalize() { }
```

Notes : Dans la pratique cette méthode n'est utilisée que pour des cas d'implémentation très avancés.

AGRÉGATION ET COMPOSITION

AGRÉGATION ET COMPOSITION

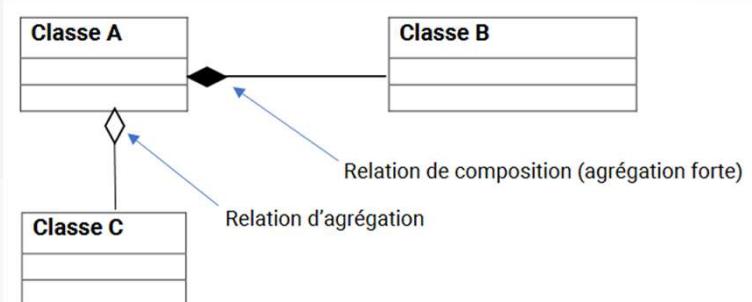
Le type d'association, entre classes, a un impact majeur lors de l'implémentation, c'est-à-dire lors de la programmation.

Au moment de l'élaboration du diagramme de classes, l'identification du type d'association est un travail qui est rigoureux et qui demande une analyse approfondie.

En effet, l'agrégation est un cas particulier d'association exprimant une relation de contenance.

L'agrégation exprime le fait qu'une classe est composée d'une ou plusieurs autres classes. Il existe deux types d'agrégation.

1. Agrégation faible
2. Agrégation forte appelée Composition.



L'agrégation est modélisée par un trait et à l'extrémité un losange vide. La composition est modélisée en UML par un trait et un losange noir plein à l'extrémité

AGRÉGATION

Soit l'exemple suivant modélisant un produit appartenant à une catégorie.



Un produit appartient à une seule catégorie et une catégorie peut avoir un ou plusieurs produits.

Cette association peut être lue de cette façon : **un produit fait partie d'une catégorie**. Aussi, cette association peut être lue de cette façon : **Une catégorie est composée d'un ou plusieurs produits**.

Lorsqu'il s'agit d'association « **fait partie** » ou « **est composé de** » ou « **avoir un** », nous parlons d'association d'agrégation.

Ce type d'association est modélisé par un trait et à l'extrémité un losange à côté du conteneur ou du composite.



Ici, le composite est la classe Catégorie et le composant est la classe Produit.

COMPOSITION

Une composition est une agrégation forte. La question qui se pose est que signifie agrégation forte.

Est-ce que la suppression d'une catégorie élimine automatiquement ces produits ?

NON ! Nous pouvons conclure que l'agrégation entre la classe Catégorie et la classe Produit n'est pas une agrégation forte, ce n'est pas une composition.

Une association de type composition se traduit par la suppression d'un objet composite élimine automatiquement tous les objets composants.

Ce qui exprime une agrégation forte.

Soit l'exemple suivant, modélisant un appartement appartient à un immeuble.



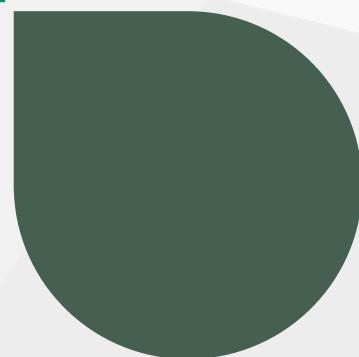
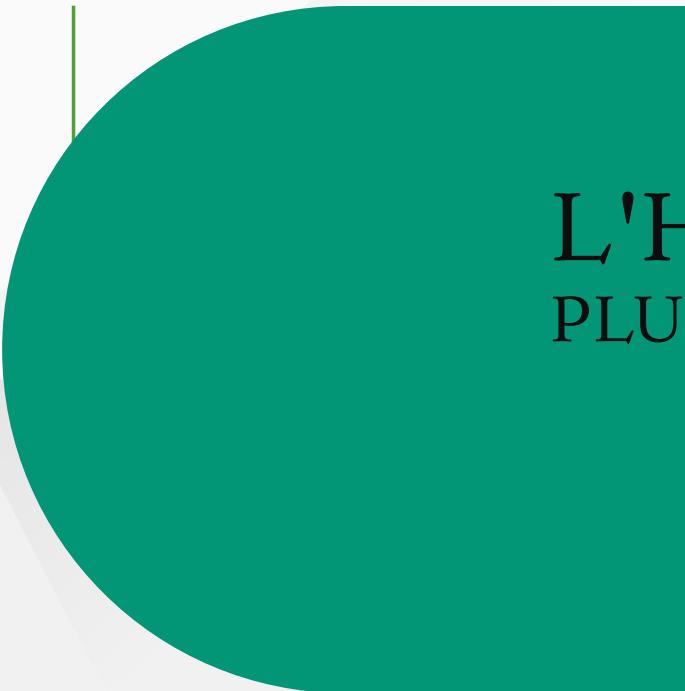
Ici, il s'agit d'une agrégation forte ou ce qu'on appelle **Composition**.

Pourquoi ?

La destruction de l'immeuble entraîne automatiquement la destruction des appartements associés.

La modélisation d'une composition est exprimée par un losange plein côté le composé.

Dans notre exemple, le composé (composite) est la classe Immeuble et le composant est la classe Appartement.



L'HÉRITAGE PLUS DE RICHESSES DANS JAVA !

HÉRITAGE AGRÉGATION FAIBLE COMPOSITION AGRÉGATION FORTE

Est-un (is-a)

Cette relation permet de créer une chaîne de relation d'identité entre des classes.

Elle indique qu'une classe peut être assimilée à une autre classe qui correspond à une notion plus abstraite ou plus générale.

On parle d'héritage pour désigner le mécanisme qui permet d'implémenter ce type de relation.

Superclasse = classe de base = Mère

Sous-classe = classe dérivé = Enfant

- La classe A est un enfant de la classe Mère

A un (has-a)

Cette relation permet de créer une relation de dépendance d'une classe envers une autre.

Une classe a besoin des services d'une autre classe pour réaliser sa fonction.

On parle également de relation de composition pour désigner ce type de relation.

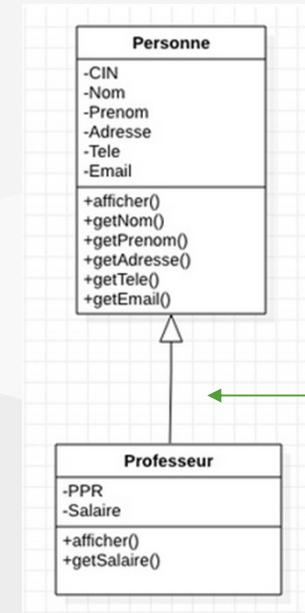
L'HÉRITAGE (IS-A)

Principe

En Java et dans tous les langages orientés objet, l'héritage est un mécanisme qui permet à une classe d'acquérir tous les comportements et attributs d'une autre classe, ce qui signifie que vous pouvez créer une nouvelle classe simplement en indiquant en quoi elle diffère d'une classe qui a déjà été développée et testée.

Lorsque vous créez une classe en la faisant hériter d'une autre classe, la nouvelle classe contient automatiquement les champs de données et les méthodes de la classe d'origine.

Exemple d'héritage



A noter : le sens de la flèche.
Une classe fille connaît sa classe mère mais pas l'inverse

LA COMPOSITION (HAS-A)

La situation “a un” décrit la composition.

- Par exemple, **Entreprise** contenant un tableau d'objets **Departments**.

- **un département est une entreprise**
- **une entreprise a des départements**



- Par conséquent, cette relation n'est pas un héritage ; c'est une forme de confinement appelée **composition** - relation dans laquelle une classe contient un ou plusieurs membres d'une autre classe, lorsque ces membres ne continueraient pas d'exister sans l'objet qui les contient.

- De même, chaque objet **Department** peut contenir un tableau d'objets **Employe**.

- **un employé est un département**
- **un département a des employés.**



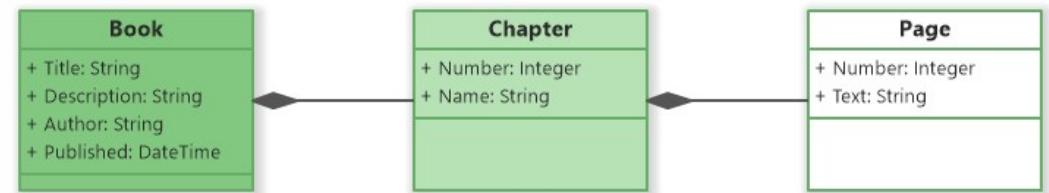
- Cette relation n'est pas non plus un héritage; il s'agit d'un type spécifique de confinement appelé **agrégation**: relation dans laquelle une classe contient un ou plusieurs membres d'une autre classe, lorsque ces membres continueraient d'exister sans l'objet qui les contient.

LA COMPOSITION (HAS-A)

La composition est le type de relation le plus souvent utilisé en programmation objet. Elle indique une dépendance entre deux classes.

L'une a besoin des services d'une autre pour réaliser sa fonction.

La composition se fait en déclarant des attributs dans la classe.



RÈGLES D'HÉRITAGE EN JAVA

Spécifique à Java :

- Une classe fille ne peut hériter que d'une seule classe mère : **Héritage simple**.
- Plusieurs classes filles peuvent hériter d'une même classe mère.
- Une classe fille peut elle-même être la classe mère d'une nouvelle classe. Il n'y a pas de limites dans les niveaux.
- On peut donc affirmer qu'en Java, toute instance de classe, quelle que soit sa classe d'appartenance est de type Object, en plus d'être du type de la classe avec laquelle elle a été instanciée.



LE CAST INDUIT PAR L'HÉRITAGE

En POO, une sous-classe peut être considérée comme une instance de la classe mère

ClasseMere objMere = new ClasseFille(); 

L'inverse n'est pas vraie

ClasseFille objFille = new ClasseMere(); 

L'héritage définit un cast implicite de ClasseFille vers ClasseMere

Dans l'exemple suivant, on part du principe que methodeRedef() existe dans la classe Mère et la classe Fille

```
ClasseMere objMere = new ClasseFille();
objMere.methodeMere();
objMere.methodeRedef(); // Quelle méthode va être appelée ?
```

Java considère toujours qu'il doit tenir compte de l'objet désigné par la référence : en conséquence, c'est bien la méthode de ClasseFille qui va être appelée. Ce comportement est logique : il est raisonnable que le traitement que l'on souhaite voir s'exécuter soit bien celui de l'objet désigné. Mais :

```
ClasseMere objMere = new ClasseFille();
objMere.methodeFille(); // Erreur de compilation
((ClasseFille) objMere).methodeFille(); // OK, le cast « rassure » le compilateur
```

Donc ici, les seules méthodes appelables directement sont celles de ClasseMere. Parmi ces méthodes, certaines peuvent avoir été redéfinies, auquel cas, ce sont les méthodes redéfinies qui seront appelées si l'on pointe sur une instance de la classe dérivée.

L'HÉRITAGE EN JAVA : EXTENDS ET INSTANCEOF

```
Public class Professeur extends Personne {  
    // code  
}
```

Chaque Professeur reçoit automatiquement les attributs et les méthodes du superclass Personne

Vous ajoutez ensuite de nouveaux attributs et méthodes à la sous-classe nouvellement créée.

```
Professeur prof = new Professeur();
```

L'objet prof a accès à toutes les méthodes de la classe Personne, ainsi qu'aux méthodes de sa propre classe.

Vous pouvez utiliser l'opérateur instanceof pour déterminer si un objet est un membre ou un descendant d'une classe.

Si prof est un objet de Professeur :

- prof instanceof Professeur ↗ true
- prof instanceof Personne ↗ true

Si p1 est un objet Personne :

- p1 instanceof Personne ↗ true
- p1 instanceof Professeur ↗ false

APPEL DE CONSTRUCTEURS PENDANT L'HÉRITAGE

Ce qui se passe chez l'enfant

Lorsque vous instanciez un objet membre d'une sous-classe, vousappelez à la fois le constructeur de la superclasse et le constructeur de la sous-classe. Lorsque vous créez un objet de sous-classe, le constructeur de la superclasse doit d'abord s'exécuter, puis le constructeur de la sous-classe est exécuté.

Constructeur par défaut

Si aucune définition de constructeur alors Java fournit et utilise le constructeur par défaut qui ne nécessite pas d'arguments.

Lorsqu'on définit un constructeur (qu'il soit par défaut ou avec des paramètres), alors Java utilise votre constructeur.

Si une superclasse contient :	Alors ses sous-classes :
Aucun constructeur définit	Ne nécessite pas de constructeurs
Un constructeur par défaut définit	Ne nécessite pas de constructeurs
Seuls les constructeurs autres que ceux par défaut	Doit contenir un constructeur qui appelle celui de la superclasse

APPEL DE CONSTRUCTEURS PENDANT L'HÉRITAGE

super(liste des paramètres)

Le mot-clé super fait toujours référence à la super-classe de la classe dans laquelle vous l'utilisez.

Remarque ! L'instruction super() doit être la première instruction de tout constructeur de sous-classe qui l'utilise. Même les définitions des attributs ne peuvent la précéder. On construit d'abord l'ascendant avant de construire l'objet.

Exemple :

```
class Personne {  
    String nom;  
    String cin;  
  
    public Personne(String nom, String cin) {  
        this.nom = nom;  
        this.cin = cin;  
    }  
  
    public void afficher() {  
        System.out.println("Nom : " + nom + " - cin : " + cin);  
    }  
  
    public class Professeur extends Personne {  
        int PPR;  
        double salaire;  
  
        public Professeur(String nom, String cin, int ppr, double salaire) {  
            super(nom, cin);  
            this.PPR = ppr;  
            this.salaire = salaire;  
        }  
    }  
}
```

Constructeur de la Mère

Constructeur de l'enfant avec
appel du constructeur de la
Mère

POO : COMPRENDRE LA SURCHARGE ET LA REDÉFINITION DE MÉTHODE EN JAVA

REDÉFINITION DE MÉTHODE

La programmation orientée objet repose sur le concept d'héritage, qui permet aux classes de dériver de classes existantes.

L'héritage permet aux classes de partager des propriétés et des méthodes communes, tout en offrant la possibilité de personnaliser et d'ajouter des fonctionnalités spécifiques à chaque classe.

Pour exploiter cette flexibilité et cette réutilisabilité, les développeurs ont recours à des concepts clés tels que **la redéfinition de méthode, la surcharge et le polymorphisme**.

La redéfinition de méthode, également appelée polymorphisme par substitution, se produit lorsque les sous-classes redéfinissent les méthodes héritées de leur classe parente. Cela permet aux sous-classes de personnaliser le comportement des méthodes héritées pour répondre à leurs propres besoins.

Prenons l'exemple d'une classe Voiture qui a une méthode "vitesseMaximale"

```
public class Voiture {  
  
    public void vitesseMaximale() {  
        System.out.println("Vitesse max voiture = 200 km/h");  
    }  
}
```

Maintenant, un programmeur qui hérite de la classe Voiture et souhaite redéfinir la méthode "vitesseMaximale" pour sa propre utilisation.

```
public class Enfant extends Voiture {  
  
    @Override  
    public void vitesseMaximale() {  
        System.out.println("Vitesse max voiture enfant = 20 km/h");  
    }  
}
```

SURCHAGE DE MÉTHODE

La surcharge de méthode se produit lorsqu'une classe a plusieurs méthodes avec le même nom, mais des paramètres différents.

Cela permet à une classe de traiter différents types de données et de fournir une interface commune pour les méthodes qui ont des fonctionnalités similaires.

Prenons l'exemple d'une classe Voiture qui a une méthode "vitesseMaximale" surchargée pour accepter un argument de type String :

```
public class Voiture {  
  
    public void vitesseMaximale() {  
        System.out.println("La vitesse maximale de la voiture est de 200 km/h");  
    }  
  
    public void vitesseMaximale(String conducteur) {  
        System.out.println("La vitesse maximale de la voiture pour " + conducteur + " est de 250 km/h");  
    }  
}
```

Dans cet exemple, la classe Voiture a deux méthodes avec le même nom "vitesseMaximale", mais la deuxième méthode prend un argument de type String qui représente le nom du conducteur.

Lorsqu'un objet de la classe Voiture est créé et que la méthode "vitesseMaximale" est appelée avec un argument, la méthode surchargée sera exécutée plutôt que la méthode originale qui ne prend pas d'argument.

POLYMORPHISME

RÉ-ÉCRITURE DES MÉTHODES

L'HÉRITAGE EN JAVA : POLYMORPHISME

Redéfinition des méthodes de la Mère

Lorsque vous créez une sous-classe en étendant une classe existante, la nouvelle sous-classe contient des données et des méthodes définies dans la superclasse d'origine.

Parfois, cependant, les attributs et les méthodes de la superclasse ne conviennent pas entièrement aux objets de la sous-classe.

Dans ces cas, vous souhaitez redéfinir les membres de la classe mère.

Polymorphisme

Utiliser le même nom de méthode pour indiquer différentes implémentations est appelé **polymorphisme** : **Même signature mais implémentation différente.**

Lorsqu'on souhaite redéfinir une méthode de classe mère dans une classe enfant, on peut insérer l'annotation **@Override** juste avant la signature de la méthode dans la classe fille.

Cela force le compilateur à émettre un message d'erreur si cela n'est pas fait !

Les trois types de méthodes que vous ne pouvez pas redéfinir dans une sous-classe sont :

- Méthodes statiques
- Méthodes finales
- Méthodes dans les classes finales

POLYMORPHISME PAR L'EXEMPLE

Le polymorphisme se réfère à la capacité des objets d'une classe à prendre plusieurs formes.

En Java, le polymorphisme est réalisé grâce à la redéfinition de méthode et à l'utilisation d'une référence de type de classe parente pour faire référence à une classe fille.

Prenons l'exemple d'une classe Animal qui a une méthode "crier"

```
public class Animal {

    public void crier() {
        System.out.println("L'animal crie");
    }
}
```

Maintenant, supposons que nous ayons deux classes qui héritent de la classe Animal, la classe Chien et la classe Chat :

```
public class Chien extends Animal {

    @Override
    public void crier() {
        System.out.println("Le chien aboie");
    }
}
```

```
public class Chat extends Animal {

    @Override
    public void crier() {
        System.out.println("Le chat miaule");
    }
}
```

Dans cet exemple, les classes Chien et Chat ont redéfini la méthode "crier" héritée de la classe Animal pour représenter les sons qu'ils font.

POLYMORPHISME PAR L'EXEMPLE

Maintenant, supposons que nous ayons une méthode "faireCrier" dans une autre classe qui prend un objet de la classe Animal en paramètre et appelle sa méthode "crier" :

```
public class RefugedeAnimaux {  
  
    public void faireCrier(Animal animal) {  
        animal.crier();  
    }  
  
}
```

Maintenant, si nous créons un objet Chien et un objet Chat et appelons la méthode "faireCrier" sur chaque objet, la méthode "crier" correspondante de chaque objet sera exécutée :

```
public static void main(String[] args) {  
  
    RefugedeAnimaux refugedeAnimaux = new RefugedeAnimaux();  
    Chien chien = new Chien();  
    Chat chat = new Chat();  
  
    refugedeAnimaux.faireCrier(chien); // affiche "Le chien aboie"  
    refugedeAnimaux.faireCrier(chat); // affiche "Le chat miaule"  
}
```

Dans cet exemple, l'utilisation de la référence de type de classe parente "Animal" nous permet d'appeler la méthode "crier" de la sous-classe appropriée en fonction de l'objet passé en paramètre.

LA CLASSE OBJECT

LA CLASSE À L'ORIGINE DE TOUT CHOSES

LA CLASSE OBJECT

Java est un langage qui ne supporte que l'héritage simple. L'arborescence d'héritage est un arbre **dont la racine est la classe Object**.

Si le développeur ne précise pas de classe parente dans la déclaration d'une classe, alors la classe hérite implicitement de Object.

La classe Object fournit des méthodes communes à toutes les classes.

Certaines de ces méthodes doivent être redéfinies dans les classes filles pour fonctionner correctement.

La méthode equals

- L'implémentation par défaut de equals fournie par Object compare les références entre elles. Si la simple égalité de référence ne suffit pas, **il faut alors redéfinir la méthode**.
Dans certaines classes, cette méthode est déjà redéfinie : pour la classe String par exemple.

La méthode hashCode

- La méthode hashCode est fournie pour l'utilisation de certains algorithmes, notamment pour l'utilisation de table de hachage. (**Utilisation très technique**)

La méthode toString

- C'est une méthode très utile, notamment pour le débogage et la production de log. Elle permet d'obtenir une représentation sous forme de chaîne de caractères d'un objet. Elle est implicitement appelée par le compilateur lorsqu'on concatène une chaîne de caractères avec un objet. **Il suffit de redéfinir cette méthode pour obtenir la représentation souhaitée.**

REDÉFINITION DE LA MÉTHODE EQUALS

L'implémentation par défaut de `equals` fournie par `Object` compare les références entre elles. L'implémentation par défaut est donc simplement :

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

Parfois, l'implémentation par défaut peut suffire mais si on souhaite tester de l'égalité d'un objet au-delà de la notion de référence, il faut redéfinir la méthode `equals`.

L'implémentation de `equals` doit être conforme à certaines règles pour s'assurer qu'elle fonctionnera correctement :

- Son implémentation doit être réflexive :
 - Pour `x` non nul, `x.equals(x)` doit être vrai
- Son implémentation doit être symétrique :
 - Si `x.equals(y)` est vrai alors `y.equals(x)` doit être vrai
- Son implémentation doit être transitive :
 - Pour `x`, `y` et `z` non nuls
 - Si `x.equals(y)` est vrai
 - Et si `y.equals(z)` est vrai
 - Alors `x.equals(z)` doit être vrai
- Son implémentation doit être consistante
 - Pour `x` et `y` non nuls
 - Si `x.equals(y)` est vrai alors il doit rester vrai tant que l'état de `x` et de `y` est inchangé.
- Si `x` est non nul alors `x.equals(null)` doit être faux.

LA CLASSE OBJECT

La méthode finalize

La méthode finalize est appelée par le ramasse-miettes avant que l'objet ne soit supprimé et la mémoire récupérée.

Redéfinir cette méthode donne donc l'opportunité au développeur de déclencher un traitement avant que l'objet ne disparaisse.

La méthode clone

La méthode clone est utilisée pour cloner une instance, c'est-à-dire obtenir une copie d'un objet.

Par défaut, elle est déclarée protected car toutes les classes ne désirent pas permettre de cloner une instance.

Pour qu'un objet soit clonable, sa classe doit implémenter l'interface marqueur [Cloneable](#).

ATTENTION : il effectue un clonage simple et non en profondeur donc si l'objet à cloner contient des références à des objets alors les attributs de ces objets ne seront pas clonés.

LA CLASSE OBJECT

La méthode getClass

La méthode `getClass` permet d'accéder à l'objet représentant la classe de l'instance.

Cela signifie qu'un programme Java peut accéder par programmation à la définition de la classe d'une instance.

Cette méthode est notamment très utilisée dans des usages avancés impliquant la réflexivité.

Les méthodes de concurrence

La classe `Object` fournit un ensemble de méthodes qui sont utilisées pour l'échange de signaux (thread) dans la programmation concurrente.

Il s'agit des méthodes `notify`, `notifyAll` et `wait`.

LES TYPES DE COMPARAISONS

`==`, `EQUALS`, `COMPARATOR`

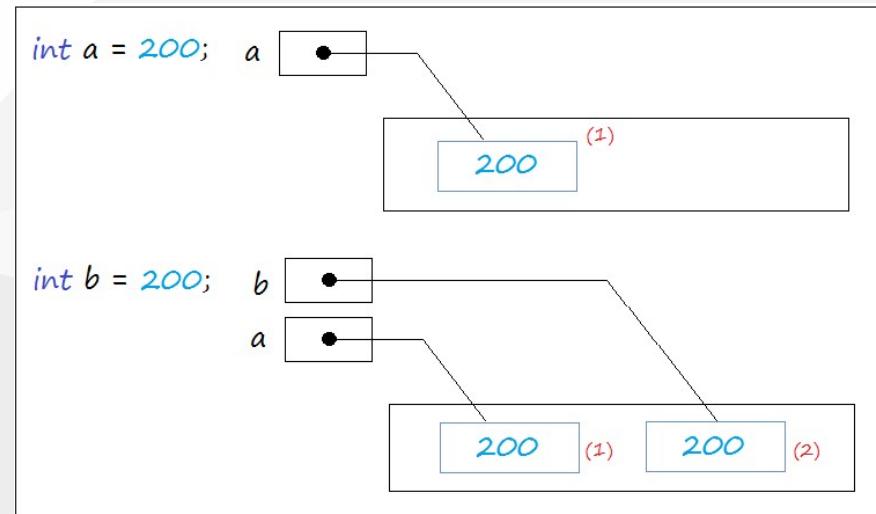
LES TYPES DE COMPARAISON EN JAVA

Comparaison des types primitifs

Avec le type primitif, nous avons seulement une façon de comparer par l'utilisation l'opérateur `==`

les types primitifs comparent entre eux via ses valeurs.

Exemple



COMPARAISON LES TYPES DE RÉFÉRENCE AVEC ==

L'utilisation l'opérateur `==` pour comparer des types de référence

l'opérateur `==` compare les références de l'objet pour voir si elles font référence à la même instance String.

- Si la valeur des deux références d'objet **fait référence à la même instance String**
 - le résultat de l'expression booléenne serait **True**.
- Si, en revanche, la valeur des deux références d'objet **fait référence à différentes instances même si les deux occurrences de String ont la même valeur.**
 - Le résultat de l'expression booléenne serait **False**.

```
public class ReferenceEeDemo {
    public static void main(String[] args) {
        // REMARQUE: pour String, deux façons d'initialiser l'objet ci-dessous ne sont pas les mêmes:
        String str1 = "String 1";
        String str2 = new String("String 1");
        // L'opérateur 'new' crée la zone de mémoire (1)
        // contient la chaîne (String) "This is text"
        // Et 's1' est une référence qui pointe vers la zone (1).
        String s1 = new String("This is text");
        // L'opérateur 'new' crée la zone de mémoire (2)
        // Contient la chaîne (String) "This is text"
        // Et 's2' est une référence qui pointe vers la zone (2)
        String s2 = new String("This is text");
        // Utilisez l'opérateur == pour comparer 's1' et 's2'.
        // Les résultats sont faux (false).
        // Il est évidemment différent de ce que vous pensez.
        // La raison en est le type de référence
        // L'opérateur == compare les positions auxquelles ils indiquent.
        boolean e1 = (s1 == s2); // false
        System.out.println("s1 == s2 ? " + e1);

        // Il n'y a aucun opérateur 'new'.
        // Java crée une référence appelée 'obj'.
        // Et pointant vers une zone de mémoire sur laquelle 's1' indique.
        Object obj = s1;

        // 2 référence 'obj' et 's1' indiquent toutes une zone de mémoire.
        // Le résultat renvoie vrai (true).
        boolean e2 = (obj == s1); // true
        System.out.println("obj == s1 ? " + e2);
    }
}
```

s1 == s2 ? false
 obj == s1 ? true

COMPARAISON LES TYPES DE RÉFÉRENCE AVEC EQUALS

L'utilisation l'opérateur `equals(..)` pour comparer des types de référence.

La méthode `equals()` compare la valeur des objets, indépendamment du fait que les deux références d'objet se réfèrent ou non à la même instance **si celle-ci a été redéfinie dans la classe (la classe String par exemple)**.

- Si les deux références d'objet de type String font référence à la même instance String, alors c'est parfait!
- Si les deux références d'objet font référence à deux occurrences String différentes, cela ne fait aucune différence. C'est la « valeur » (c'est-à-dire: le contenu de la chaîne de caractères).
- **Si ce n'est pas le cas alors il faut redéfinir la méthode equals**

```
public class StringComparationDemo {  
  
    Run | Debug  
    public static void main(String[] args) {  
  
        String s1 = new String(original: "This is text");  
  
        String s2 = new String(original: "This is text");  
  
        // s1 and s2 Comparison, use equals(..)  
        boolean e1 = s1.equals(s2);  
  
        // The result is true  
        System.out.println("first comparation: s1 equals s2 ? " + e1);  
  
        s2 = new String(original: "New s2 text");  
  
        boolean e2 = s1.equals(s2);  
  
        // The result is false  
        System.out.println("second comparation: s1 equals s2 ? " + e2);  
    }  
}
```

```
first comparation: s1 equals s2 ? true  
second comparation: s1 equals s2 ? false
```

REEMPLACER LA MÉTHODE EQUALS(OBJECT)

La méthode `equals(Object)` est donc une méthode disponible dans la classe `Object`.

Dans certaines situations, vous pouvez remplacer (polymorphisme) cette méthode dans les sous-classes.

```
public class NumberOfMedalsComparationDemo {
    public static void main(String[] args) {
        // La réussite de l'équipe américaine.
        NumberOfMedals american = new NumberOfMedals(goldCount: 40, silverCount: 15, bronzeCount: 15);

        // La réussite de l'équipe japonaise.
        NumberOfMedals japan = new NumberOfMedals(goldCount: 10, silverCount: 5, bronzeCount: 20);

        // La réussite de l'équipe sud-coréenne
        NumberOfMedals korea = new NumberOfMedals(goldCount: 10, silverCount: 5, bronzeCount: 20);

        System.out.println("Medals of American equals Japan ? " + american.equals(japan));

        System.out.println("Medals of Korea equals Japan ? " + korea.equals(japan));
    }
}
```

Medals of American equals Japan ? false
Medals of Korea equals Japan ? true

```
public class NumberOfMedals {
    // Le nombre de médailles d'or
    private int goldCount;

    // Le nombre de médailles d'argent.
    private int silverCount;

    // Le nombre de médailles de bronze
    private int bronzeCount;

    public NumberOfMedals(int goldCount, int silverCount, int bronzeCount) {
        this.goldCount = goldCount;
        this.silverCount = silverCount;
        this.bronzeCount = bronzeCount;
    }

    public int getGoldCount() {
        return goldCount;
    }

    public int getSilverCount() {
        return silverCount;
    }

    public int getBronzeCount() {
        return bronzeCount;
    }

    // Remplacez la méthode equals(Object) de la classe Objet
    @Override
    public boolean equals(Object other) {
        // Si other = null, le résultat renvoie faux (false).
        if (other == null) {
            return false;
        }
        // Si 'other' n'est pas le type de NumberOfMedals
        // le résultat renvoie faux (false).
        if (!(other instanceof NumberOfMedals)) {
            return false;
        }

        NumberOfMedals otherNoM = (NumberOfMedals) other;

        if (this.goldCount == otherNoM.goldCount && this.silverCount == otherNoM.silverCount
            && this.bronzeCount == otherNoM.bronzeCount) {
            return true;
        }
        return false;
    }
}
```

COMPARER DEUX OBJETS

INTERFACE COMPARABLE

INTERFACE COMPARABLE

L'interface Comparable est utilisée pour comparer un objet de la même classe avec une instance de cette classe, elle fournit l'ordre des données pour les objets de la classe définie par l'utilisateur.

La classe doit implémenter l'interface `java.lang.Comparable` pour comparer son instance, elle fournit la méthode `compareTo` qui prend un paramètre de l'objet de cette classe.



DES OBJETS PEUVENT ÊTRE COMPARÉS ENTRE EUX (COMPARABLE)

Vous écrivez une classe qui simule un acteur (Actor).

Vous souhaitez organiser l'ordre des acteurs selon le principe de comparer le nom de famille (lastName) d'abord, comparez le prénom (firstName)

Pour cela, vous devez implémenter l'interface **Comparable** dans votre classe Actor et surcharger la méthode **compareTo**

```
// Pour comparer les uns avec les autres,  
// la classe d'Actor doit implémenter une interface Comparable.  
public class Actor implements Comparable<Actor> {  
  
    private String firstName;  
    private String lastName;  
  
    public Actor(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
  
    public String getFirstName() {  
        return firstName;  
    }  
  
    public String getLastName() {  
        return lastName;  
    }  
  
    // Comparez cet Actor avec un autre actor.  
    // Selon le principe de comparaison du dernier nom d'abord,  
    // puis comparez FirstName.  
    @Override  
    public int compareTo(Actor other) {  
  
        // Comparez deux chaînes.  
        int value = this.lastName.compareTo(other.lastName);  
  
        // Si lastName de deux objets ne sont pas égaux  
        if (value != 0) {  
            return value;  
        }  
        // Si lastName de deux objets sont le même.  
        // Ensuite, comparez fistName.  
        value = this.firstName.compareTo(other.firstName);  
        return value;  
    }  
}
```

DES OBJETS PEUVENT ÊTRE COMPARÉS ENTRE EUX (COMPARABLE)

Ensute dans votre programme principal, il suffit de parcourir votre tableau d'acteurs et d'utiliser votre méthode compareTo

Gemma Arterton
Christian Bale
Mischa Barton
Joan Collins
Daniel Craig

```
public class ActorSortingDemo {  
  
    Run | Debug  
    public static void main(String[] args) {  
  
        Actor actor1 = new Actor("Mischa", "Barton");  
        Actor actor2 = new Actor("Christian", "Bale");  
        Actor actor3 = new Actor("Joan", "Collins");  
        Actor actor4 = new Actor("Gemma", "Arterton");  
        Actor actor5 = new Actor("Daniel", "Craig");  
  
        Actor[] actors = new Actor[] { actor1, actor2, actor3, actor4, actor5 };  
  
        // Utilisez un algorithme pour trier ce nouveau tableau ci-dessus.  
        // Triez les objets Actor en augmentant progressivement l'ordre.  
        for (int i = 0; i < actors.length; i++) {  
  
            for (int j = i + 1; j < actors.length; j++) {  
                // Si actors[j] < actors[i]  
                // Ensuite, échangez les positions les uns avec les autres.  
                if (actors[j].compareTo(actors[i]) < 0) {  
                    // Utilisez une variable temporaire.  
                    Actor temp = actors[j];  
                    actors[j] = actors[i];  
                    actors[i] = temp;  
                }  
            }  
        }  
        // Imprimez les éléments du tableau.  
        for (int i = 0; i < actors.length; i++) {  
            System.out.println(actors[i].getFirstName() + " " + actors[i].getLastName());  
        }  
    }  
}
```

TRIER PAR COMPARATEUR (COMPARATOR)

Dans les exemples ci-dessus, nous trions un tableau ou une liste, dont ses éléments sont capables de se comparer les uns aux autres (parce qu'il implémente une interface Comparable).

La question avec les classes qui n'implémente pas l'interface Comparable. Dans ce cas, vous devez fournir un comparateur. C'est une règle pour organiser des objets.

```
public class Person {  
  
    private int age;  
    private String fullName;  
  
    public Person(String fullName, int age) {  
        this.fullName = fullName;  
        this.age = age;  
    }  
  
    public int getAge() {  
        return age;  
    }  
  
    public String getFullName() {  
        return fullName;  
    }  
}
```

```
import java.util.Comparator;  
  
// Cette classe implémente l'interface Comparateur <Person>  
// Il peut être considéré comme la règle pour comparer les objets Person.  
public class PersonComparator implements Comparator<Person> {  
  
    // Remplacez (override) la méthode compare.  
    // Spécifiez les règles de comparaison entre 2 objets Person.  
    @Override  
    public int compare(Person o1, Person o2) {  
        // Si deux objets sont nuls, ils sont considérés comme égaux.  
        if (o1 == null && o2 == null) {  
            return 0;  
        }  
        // Si o1 est null, o2 est considéré plus grand.  
        if (o1 == null) {  
            return -1;  
        }  
        // Si o2 est null, o1 est considéré plus grand.  
        if (o2 == null) {  
            return 1;  
        }  
        // Principe:  
        // Trier progressivement par rapport à l'âge.  
        int value = o1.getAge() - o2.getAge();  
        if (value != 0) {  
            return value;  
        }  
        // Si l'âge est le même, comparez le nom complet (fullName).  
        // Comparez par Alphabet( les lettres).  
        value = o1.getFullName().compareTo(o2.getFullName());  
        return value;  
    }  
}
```

VARIANTE AVEC ARRAYS.SORT

Utiliser Arrays.sort(Object[]) pour trier la liste de l'exemple ci-dessus.

```
import java.util.Arrays;

public class ActorSortingDemo2 {
    Run | Débug
    public static void main(String[] args) {

        Actor actor1 = new Actor("Mischa", "Barton");
        Actor actor2 = new Actor("Christian", "Bale");
        Actor actor3 = new Actor("Joan", "Collins");
        Actor actor4 = new Actor("Gemma", "Arterton");
        Actor actor5 = new Actor("Daniel", "Craig");

        Actor[] actors = new Actor[] { actor1, actor2, actor3, actor4, actor5 };

        // Utilisez Arrays.sort(Object[]) pour trier.
        Arrays.sort(actors);

        // Imprimez les éléments.
        for (int i = 0; i < actors.length; i++) {
            System.out.println(actors[i].getFirstName() + " " + actors[i].getLastName());
        }
    }
}
```

TRIER UNE LISTE

Utilisation d'une ArrayList et de sa méthode sort()

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

public class ListSortingDemo {

    Run|Debug
    public static void main(String[] args) {

        Actor actor1 = new Actor("Mischa", "Barton");
        Actor actor2 = new Actor("Christian", "Bale");
        Actor actor3 = new Actor("Joan", "Collins");
        Actor actor4 = new Actor("Gemma", "Arterton");
        Actor actor5 = new Actor("Daniel", "Craig"),

        // Une liste d'éléments comparables.
        // (Comparable)
        List<Actor> actors = new ArrayList<Actor>();

        actors.add(actor1);
        actors.add(actor2);
        actors.add(actor3);
        actors.add(actor4);
        actors.add(actor5);

        // Utilisez la méthode Collections.sort(List)
        // afin de trier une liste (List)
        Collections.sort(actors);

        for (Actor actor : actors) {
            System.out.println(actor.getFirstName() + " " + actor.getLastName());
        }
    }
}
```

TRIER PAR COMPARATEUR (COMPARATOR)

Ensuite, que ce soit à partir d'un tableau ou d'une liste, il faut faire appel à la méthode

`sort(tableau_ou_collection,Comparator<? supers T>)`
comme le montre l'exemple :

```
Person: 18 / Mischa
Person: 20 / Christian
Person: 20 / Marry
Person: 21 / Daniel
Person: 21 / Tom
-----
Person: 18 / Mischa
Person: 20 / Christian
Person: 20 / Marry
Person: 21 / Daniel
Person: 21 / Tom
```

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

public class ComparatorSortingDemo {

    public static void main(String[] args) {
        Person person1 = new Person("Marry", 20);
        Person person2 = new Person("Tom", 21);
        Person person3 = new Person("Daniel", 21);
        Person person4 = new Person("Mischa", 18);
        Person person5 = new Person("Christian", 20);

        // Un tableau n'est pas trié.
        Person[] array = new Person[] { person1, person2, person3, person4, person5 };

        // Trier le tableau, utilisez: <T> Arrays.sort(T[],Comparator<? supers T>).
        // Et fournissez un Comparator (un Comparateur).
        Arrays.sort(array, new PersonComparator());

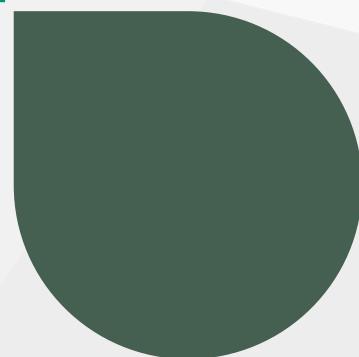
        for (Person person : array) {
            System.out.println("Person: " + person.getAge() + " / " + person.getFullName());
        }

        System.out.println(x: -----);

        // Pour la liste:
        List<Person> list = new ArrayList<Person>();
        list.add(person1);
        list.add(person2);
        list.add(person3);
        list.add(person4);
        list.add(person5);

        // Trier le tableau, utilisez:
        // <T> Collections.sort(List<T>, Comparator<? supers T>).
        // Et fournissez un Comparator (un Comparateur).
        Collections.sort(list, new PersonComparator());

        for (Person person : list) {
            System.out.println("Person: " + person.getAge() + " / " + person.getFullName());
        }
    }
}
```



COLLECTIONS

COLLECTIONNEZ-LES TOUS !!

LES COLLECTIONS

On a découvert les tableaux et on peut vite se rendre compte de leurs limites avec l'utilisation des objets.

- Leur taille fixe par exemple.

Les collections vont apporter une souplesse d'utilisation.

Parmi les collections, on trouve :

- Les listes ([lists](#))
- Les ensembles ([sets](#))
- Les tableaux associatifs ([maps](#))

Toutes ces collections sont génériques. On peut donc créer que des collections d'objets.

Si on veut créer une collection d'un type primitif. Il faut utiliser la classe enveloppe du type :

- [Integer](#) pour [int](#) par exemple

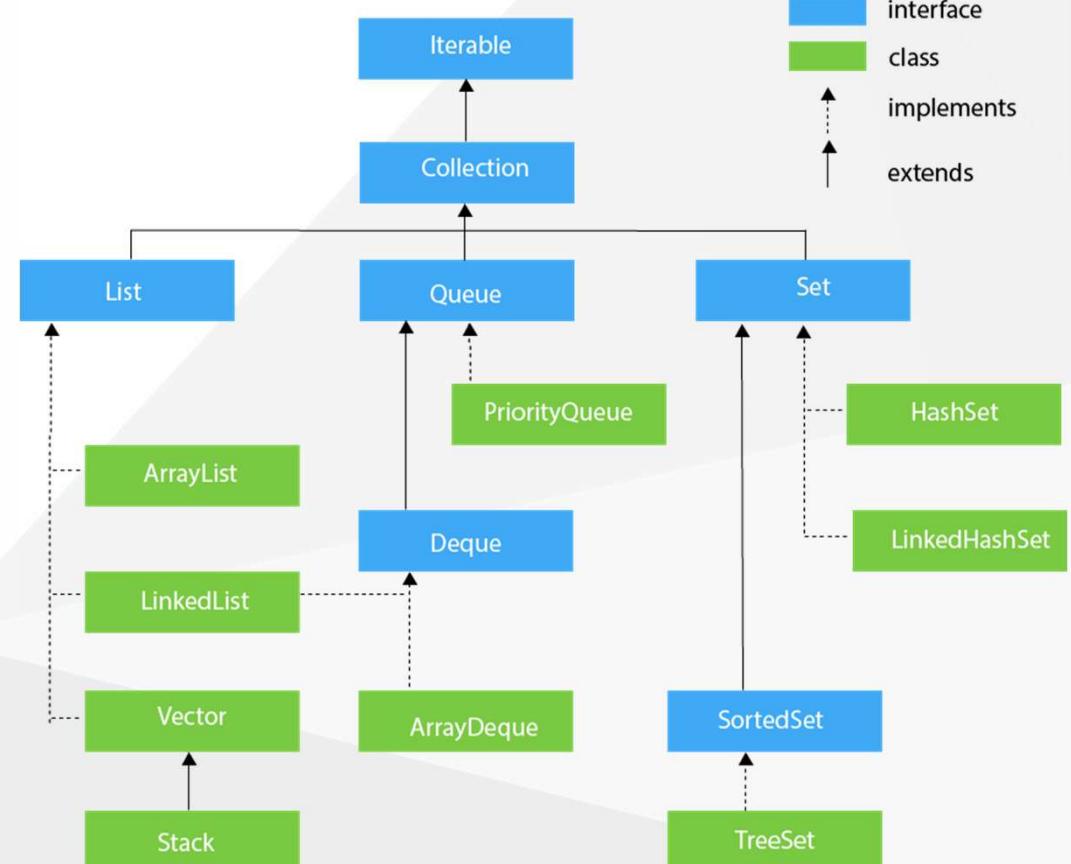
LISTS

Une liste est une collection ordonnée d'éléments.

Il existe différentes façons d'implémenter des listes dont les performances sont optimisées soit pour les accès aléatoires aux éléments soit pour les opérations d'insertion et de suppression d'éléments.

L'interface **Collection** dont hérite toutes les autres interfaces pour les listes, hérite elle-même de **Iterable**.

- Cela signifie que toutes les classes et toutes les interfaces servant à représenter des listes dans le Java Collections Framework peuvent être parcourues avec une structure de for amélioré (**foreach**).



LISTS

La classe ArrayList

La classe `java.util.ArrayList` est une implémentation de l'interface List.

Elle stocke les éléments de la liste sous la forme de blocs en mémoire.

- Cela signifie que la classe ArrayList est très performante pour les accès aléatoires en lecture aux éléments de la liste.
- Par contre, les opérations d'ajout et de suppression d'un élément se font en temps linéaire.
- Elle est donc moins performante que la classe `LinkedList` sur ce point.

La classe LinkedList

La classe `java.util.LinkedList` est une implémentation de l'interface List.

Sa représentation interne est une liste doublement chaînée.

- Cela signifie que la classe `LinkedList` est très performante pour les opérations d'insertion et de suppression d'éléments.
- Par contre, l'accès aléatoire en lecture aux éléments se fait en temps linéaire.
- Elle est donc moins performante que la classe `ArrayList` sur ce point.

JAVA.UTIL.ARRAYLIST

La classe `ArrayList` peut être utilisée pour créer des listes d'objets.

- redimensionnable dynamiquement, ce qui signifie que sa taille peut changer pendant l'exécution du programme.

```
ArrayList< String> names = new ArrayList< String>();
```

Une `ArrayList` peut contenir n'importe quel type d'objet

- L'ajout d'un type de données entre crochets fait que Java vérifie que vous affectez les types appropriés à une liste.

Le constructeur par défaut crée une `ArrayList` avec une capacité de 10 éléments mais :

```
ArrayList< String> noms = new ArrayList< String>(20);
```

la méthode `Collections.sort(collection)` permet de trier une `ArrayList`

Méthode	Description
<code>public void add(Object)</code> <code>public void add(int, Object)</code>	Ajouter un élément à une <code>ArrayList</code> ; la version par défaut ajoute un élément au prochain emplacement disponible; une version surchargée vous permet de spécifier une position à laquelle nous voulons ajouter l'élément
<code>public void remove(int)</code>	Supprimer un élément d'une <code>ArrayList</code> à un emplacement spécifié
<code>public void set(int, Object)</code>	Modifier un élément à un emplacement spécifié dans une <code>ArrayList</code>
<code>Object get(int)</code>	Récupérer un élément d'un emplacement spécifié dans une <code>ArrayList</code>
<code>public int size()</code>	Renvoyer la taille actuelle de <code>ArrayList</code>

PARCOURS D'UNE ARRAYLIST

Création et parcours d'un ArrayList et d'une List

```
/**  
 * Création d'une ArrayList et manipulation.  
 */  
List<String> liste = new ArrayList<>();  
  
ArrayList<String> couleurs = new ArrayList<>();  
couleurs.add("Red");  
couleurs.add("Blue");  
couleurs.add("Yellow");  
couleurs.add("Green");  
  
// ajout d'element dans la liste  
liste.add("une première chaîne");  
liste.add("une troisième chaîne");  
  
// demande de la taille de la liste  
System.out.println(liste.size()); // 2  
  
// insertion d'un élément  
liste.add( index: 1, element: "une seconde chaîne");  
  
// affichage de la taille  
System.out.println(liste.size()); // 3  
  
// Parcours de la liste  
for (String s : liste) {  
    System.out.println(s);  
}  
  
for(String s : couleurs) {  
    System.out.println(s);  
}
```

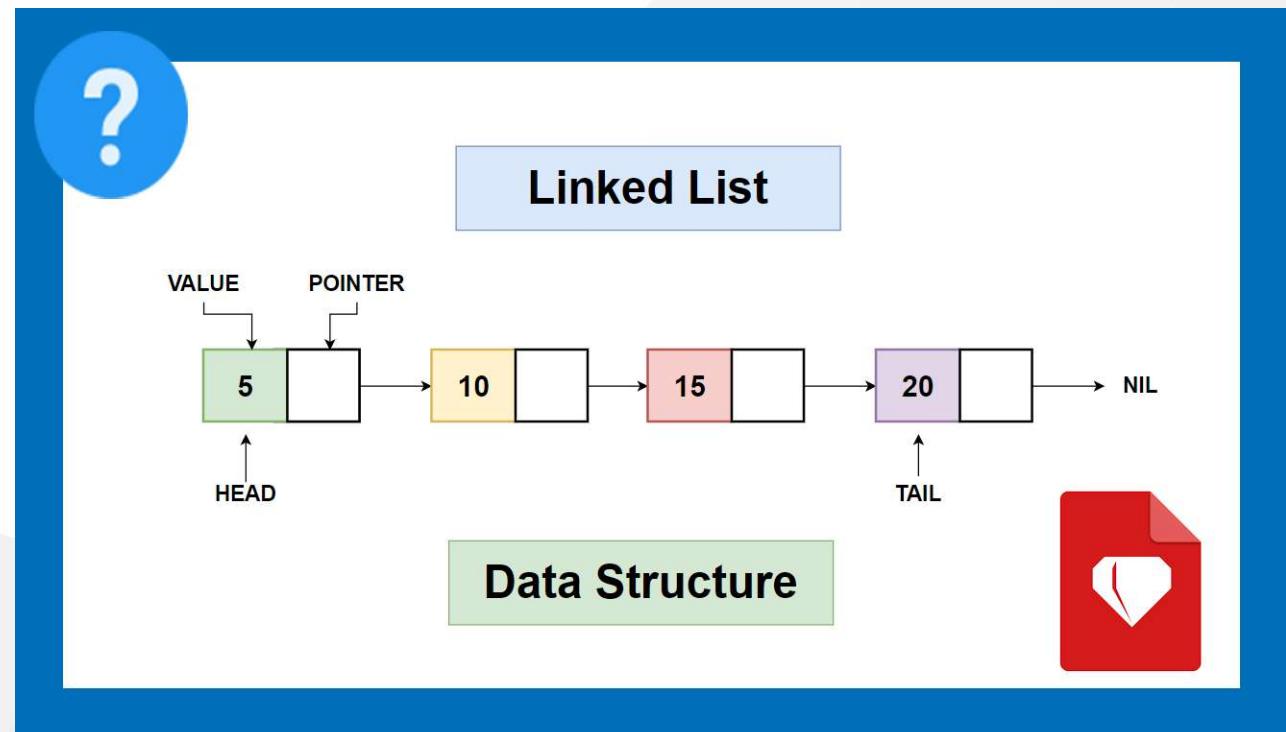
JAVA.UTIL.LINK EDLIST

LinkedList sont des structures de données linéaires où les éléments ne sont pas stockés dans des emplacements contigus et chaque élément est un objet séparé avec une partie de données et une partie d'adresse.

Les éléments sont liés à l'aide de pointeurs et d'adresses.

Chaque élément est appelé un nœud.

La classe **LinkedList** hérite de **AbstractSequentialList** et implémente l'interface **List**.



JAVA.UTIL.LINKEDLIST

Méthode	Description
void add(int index, Object element)	Cette méthode insère l'élément spécifié à la position spécifiée dans cette liste.
boolean add(Object o)	Cette méthode ajoute l'élément spécifié à la fin de cette liste.
boolean addAll(Collection c)	Cette méthode ajoute tous les éléments de la collection spécifiée à la fin de cette liste, dans l'ordre dans lequel ils sont renvoyés par l'itérateur de la collection spécifiée.
boolean addAll(int index, Collection c)	Cette méthode Insère tous les éléments de la collection spécifiée dans cette liste, en commençant à la position spécifiée
void addLast(Object o)	Cette méthode ajoute l'élément spécifié à la fin de cette liste.
Object remove()	Cette méthode récupère et supprime la tête (premier élément) de la liste.
int size()	Cette méthode renvoie le nombre d'éléments dans cette liste.
Etc...	

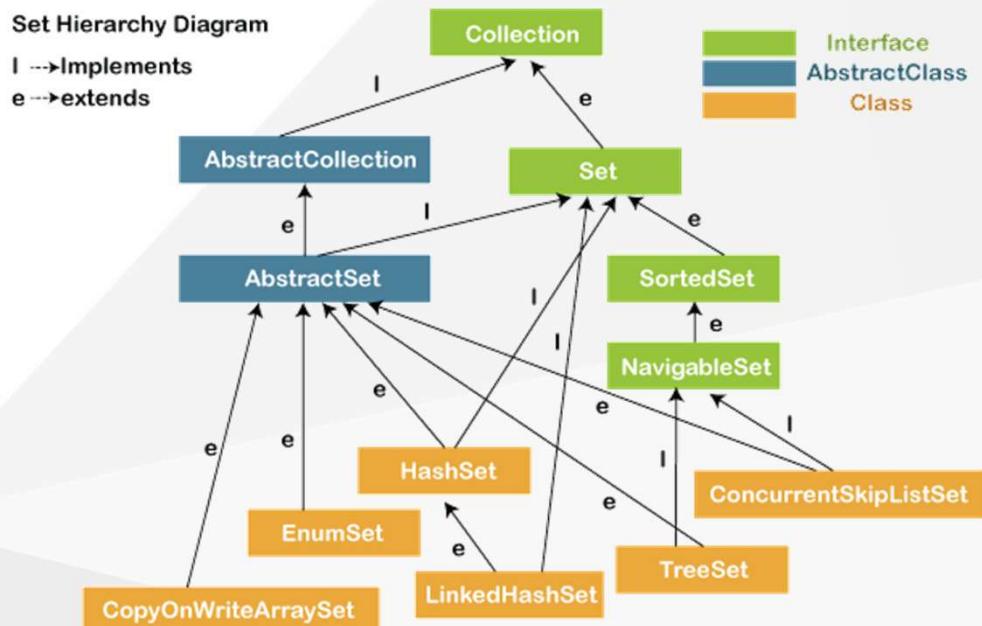
SET

Les ensembles (set) sont des collections qui ne contiennent aucun doublon. Deux éléments e1 et e2 sont des doublons si :

- `e1.equals(e2) == true` ou si `e1` vaut null et `e2` vaut null.

Pour contrôler l'unicité, 3 implémentations :

- [TreeSet](#)
 - Particularité : conserve toujours ses éléments triés.
- [HashSet](#)
 - utilise un code de hachage (hash code) pour contrôler l'unicité de ces éléments. Un code de hachage est une valeur associée à l'objet.
- [LinkedHashSet](#).
 - utilise en interne un code de hachage mais elle garantit en plus que l'ordre de parcours des éléments sera le même que l'ordre d'insertion.



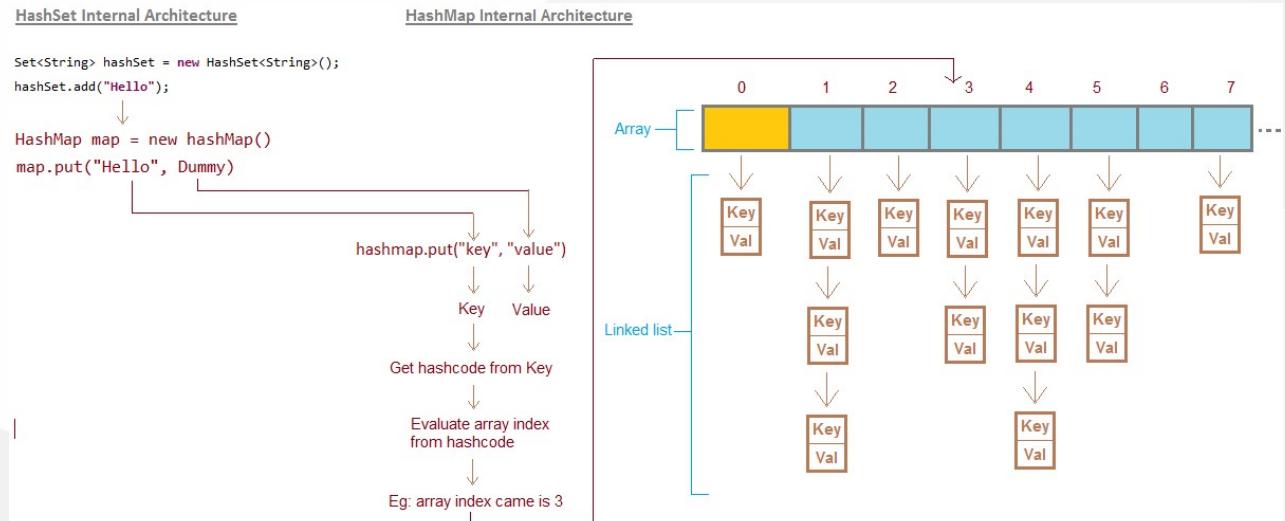
JAVA.UTIL.HASHSET

Une structure de données bien connue pour trouver des objets rapidement est la table de hachage (**HashTable**).

Une table de hachage calcule un entier, appelé le code de hachage pour chaque objet.

Un code de hachage est en quelque sorte dérivé des champs d'instance d'un objet, de préférence de telle sorte que les objets avec des données différentes génèrent des codes différents.

En Java, les tables de hachage sont implémentées en tant que tableaux de listes chaînées (**LinkedList**). Chaque liste s'appelle une alvéole (en anglais, buckets ou slots).



JAVA.UTIL.HASHSET

Méthode	Description
HashSet()	Ce constructeur construit un HashSet par défaut.
HashSet(Collection c)	Ce constructeur initialise le hash set en utilisant les éléments de la collection c.
boolean add(Object o)	Ajouter l'élément spécifié à HashSet s'il n'est pas déjà présent.
boolean isEmpty()	Retourner true si cet HashSet ne contient aucun élément.
Iterator iterator()	Retourner un itérateur sur les éléments de cet HashSet.
boolean remove(Object o)	Supprimer l'élément spécifié de cet HashSet s'il est présent.
int size()	Retourner le nombre d'éléments.
void clear()	Supprimer tous les éléments.

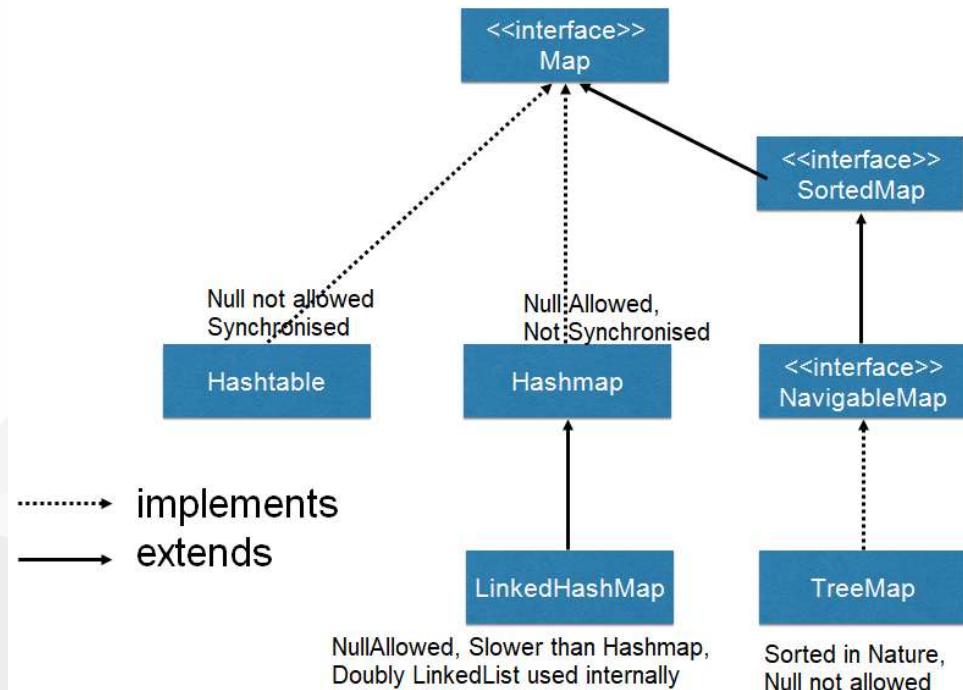
LES MAPS

Les *map* permet d'associer une clé à une valeur. Elle ne peut pas contenir de doublon de clés.

Le Java Collections Framework fournit plusieurs implémentations de tableaux associatifs :

- TreeMap,
- HashMap,
- LinkedHashMap.

Map Interface



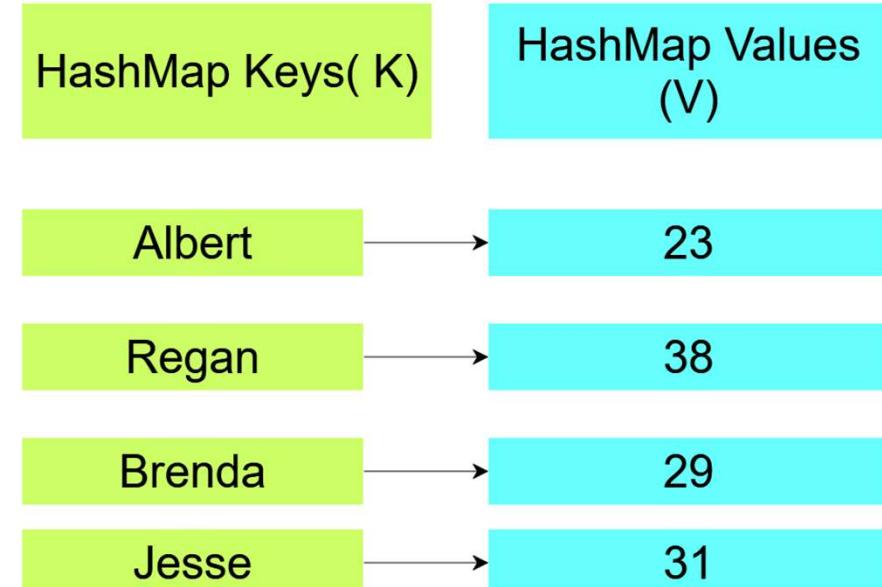
JAVA.UTIL.HASHMAP

En général, vous avez des informations clés et vous souhaitez rechercher l'élément associé.

La structure des données HashMap sert cet objectif.

Un HashMap stocke des paires clé / valeur.

Vous pouvez trouver une valeur si vous fournissez la clé.



www.TestingDocs.com

HashMap<K,V>

JAVA.UTIL.HASHMAP

Méthode	Description
HashMap<K, V>()	Ce constructeur construit un HashMap par défaut.
HashMap<K, V>(Map m)	Ce constructeur initialise un HashMap en utilisant les éléments de l'objet Map donné m.
HashMap<K, V>(int capacity)	Ce constructeur initialise la capacité de HAshMap sur la valeur entière donnée, capacity.
void clear()	Supprimer tous les mappages de cette map.
boolean containsKey(Object key)	Renvoyer true si cette map contient un mappage pour la clé spécifiée.
boolean containsValue(Object value)	Renvoyer true si cette map mappe une ou plusieurs clés sur la valeur spécifiée.
Object get(Object key)	Renvoyer la valeur à laquelle la clé spécifiée est mappée dans cette map, ou null si la carte ne contient aucune correspondance pour cette clé.
Object put(Object key, Object value)	Associer la valeur spécifiée à la clé spécifiée dans cette map.
boolean isEmpty()	Renvoyer true si cette map ne contient aucune correspondance clé-valeur.
Object remove(Object key)	Supprimer le mappage de cette clé de cette map si elle est présente..
int size()	Renvoyer le nombre de mappages clé-valeur dans cette map
Etc...	

LA CLASSE OUTIL COLLECTIONS

La classe `java.util.Collections` est une classe outil qui contient de nombreuses méthodes pour les listes, les ensembles et les tableaux associatifs.

Elle contient également des attributs de classes correspondant à une liste, un ensemble et un tableau associatif vides et immutables.

```
List<String> liste = new ArrayList<>();
Collections.addAll(liste, "un", "deux", "trois", "quatre");

// La chaîne a plus grande dans la liste : "un"
String max = Collections.max(liste);
System.out.println(max);

// Inverse l'ordre de la liste
Collections.reverse(liste);
// [quatre, trois, deux, un]
System.out.println(liste);

// Trie la liste
Collections.sort(liste);
// [deux, quatre, trois, un]
System.out.println(liste);

// Recherche de l'index de la chaîne "deux" dans la liste triée : 0
int index = Collections.binarySearch(liste, "deux");
System.out.println(index);

// Remplace tous les éléments par la même chaîne
Collections.fill(liste, "même chaîne partout");
// [même chaîne partout, même chaîne partout, même chaîne partout, même chaîne
// partout]
System.out.println(liste);

// Enveloppe la liste dans une liste qui n'autorise plus à modifier son contenu
liste = Collections.unmodifiableList(liste);

// On tente de modifier une liste qui n'est plus modifiable
liste.add("Test"); // ERREUR à l'exécution : UnsupportedOperationException
```

LA CLASSE DATE LES DATES EN JAVA

LES DATES

Java.util.Date

Elle est la première classe à être apparue pour représenter une date. Elle comporte de nombreuses limitations :

- Il n'est pas possible de représenter des dates antérieures à 1900
- Elle ne supporte pas les fuseaux horaires
- Elle ne supporte que les dates pour le calendrier grégorien
- Elle ne permet pas d'effectuer des opérations (ajout d'un jour, d'une année...)

L'API Date/Time

Depuis Java 8, une nouvelle API a été introduite pour représenter les dates, le temps et la durée. Toutes ces classes ont été regroupées dans le package [java.time](#).

Pour les dates, les classes [LocalDate](#), [LocalTime](#) et [LocalDateTime](#) permettent de représenter respectivement [une date](#), [une heure](#) et [une date et heure](#)

On peut facilement passer d'un type à une autre.

- Par exemple la méthode `LocalDate.atTime` permet d'ajouter une heure à une date, créant ainsi une instance de `LocalDateTime`. Toutes les instances de ces classes sont immutables.

Si on veut avoir l'information de la date ou de l'heure d'aujourd'hui, on peut créer une instance grâce à la méthode [now](#).

L'API DATE/TIME

Les classes Year et YearMonth

Les classes `Year` et `YearMonth` permettent de manipuler les dates et d'obtenir des informations intéressantes à partir de l'année ou du mois et de l'année.

```
Year year = Year.of( isoYear: 2004);

// année bissextile ?
boolean isLeap = year.isLeap();

// 08/2004
YearMonth yearMonth = year.atMonth(Month.AUGUST);

// 31/08/2004
LocalDate localDate = yearMonth.atEndOfMonth();
```

La classe Instant

La classe `Instant` représente un point dans le temps.

Contrairement aux classes précédentes qui permettent de représenter les dates pour les humains, la classe `Instant` est adaptée pour réaliser des traitements de données temporelles.

```
Instant maintenant = Instant.now();
Instant epoch = Instant.ofEpochSecond(0); // 01/01/1970 00:00:00.000

Instant uneMinuteDansLeFuture = maintenant.plusSeconds( secondsToAdd: 60);

long unixTimestamp = uneMinuteDansLeFuture.getEpochSecond();
```

L'API DATE/TIME

La Période

Il est possible de définir des périodes grâce à des instances de la classe [Period](#).

Une période peut être construite directement ou à partir de la différence entre deux instances de type [Temporal](#). Il est ensuite possible de modifier une date en ajoutant ou soustrayant une période.

```
YearMonth moisAnnee = Year.of(isoYear: 2000).atMonth(Month.APRIL); // 04/2000

// période de 1 an et deux mois
Period periode = Period.ofYears(1).plusMonths(monthsToAdd: 2);

YearMonth moisAnneePlusTard = moisAnnee.plus(periode); // 06/2001

Period periode65Jours = Period.between(LocalDate.now(),
    LocalDate.now().plusDays(daysToAdd: 65));
```

La durée

La durée est représentée par une instance de la classe [Duration](#). Elle peut être obtenue à partir de deux instances de [Instant](#).

```
Instant debut = Instant.now();

// ... traitement à mesurer

Duration duree = Duration.between(debut, Instant.now());
System.out.println(duree.toMillis());
```

FORMATAGE DES DATES

Pour formater une date pour l'affichage, il est possible d'utiliser la méthode `format` déclarée dans les classes `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`, `Year` et `YearMonth`.

Le format de représentation d'une date et/ou du temps est défini par la classe `DateTimeFormatter`.

```
// 01/09/2010 16:30
LocalDateTime dateTimeExemple = LocalDateTime.of(year: 2010,
    Month.SEPTEMBER, dayOfMonth: 1, hour: 16, minute: 30);

// En utilisant des formats ISO de dates
System.out.println(dateTimeExemple.format(DateTimeFormatter.BASIC_ISO_DATE));
System.out.println(dateTimeExemple.format(DateTimeFormatter.ISO_WEEK_DATE));
System.out.println(dateTimeExemple.format(DateTimeFormatter.ISO_DATE_TIME));

DateTimeFormatter datePattern = DateTimeFormatter.ofPattern("dd/MM/yyyy");
// 01/09/2010
System.out.println(dateTimeExemple.format(datePattern));

DateTimeFormatter dateTimePattern = DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm");
// 01/09/2010 16:30
System.out.println(dateTimeExemple.format(dateTimePattern));

// 1 septembre 2010
DateTimeFormatter frenchDatePattern = DateTimeFormatter
    .ofPattern(pattern: "d MMMM yyyy", Locale.FRANCE);
System.out.println(dateTimeExemple.format(frenchDatePattern));
```

LECTURES DES DATES

Pour transformer une chaîne de caractères en date (notamment pour obtenir une date à partir de la saisie d'un utilisateur ou de la lecture d'un fichier), il est possible d'utiliser la méthode `parse` déclarée dans les classes `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`, `Year` et `YearMonth`.

Le format d'une date et/ou du temps est défini par la classe `DateTimeFormatter`.

```
String exemple = "02/06/2010 12:35";

DateTimeFormatter dtf = DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm");

Year yearExemple = Year.parse(exemple, dtf);
System.out.println(yearExemple);

YearMonth yearMonthExemple = YearMonth.parse(exemple, dtf);
System.out.println(yearMonthExemple);

LocalDate localDateExemple = LocalDate.parse(exemple, dtf);
System.out.println(localDateExemple);

LocalDateTime localDateTimeExemple = LocalDateTime.parse(exemple, dtf);
System.out.println(localDateTimeExemple);
```

LES EXCEPTIONS

TRY...CATCH...FINALLY

LES EXCEPTIONS

La gestion des cas d'erreur représente un travail important dans la programmation. Les sources d'erreur peuvent être nombreuses dans un programme.

La **robustesse** d'une application est souvent comprise comme sa capacité à continuer à rendre un service acceptable dans un environnement dégradé, c'est-à-dire quand toutes les conditions attendues normalement ne sont pas satisfaites.

En Java, la gestion des erreurs se confond avec la gestion des cas exceptionnels. **On utilise alors le mécanisme des exceptions.**

Une **exception** est une classe Java qui représente un état particulier et qui hérite directement ou indirectement de la classe **Exception**.

Par convention, le nom de la classe doit permettre de comprendre le type d'exception et doit se terminer par **Exception**.

- **NullPointerException**
- **NumberFormatException**
- **IndexOutOfBoundsException**

Une exception est un événement inattendu survenant pendant l'exécution du programme.

- Entrée utilisateur non valide
- Échec d'un périphérique
- Perte de connexion réseau
- Limites physiques (mémoire insuffisante)
- Erreurs de code
- Ouvrir un fichier indisponible

HIÉRARCHIE DES EXCEPTIONS

la classe **Throwable** est la classe racine de la hiérarchie.

La liste des messages d'erreur après chaque tentative d'exécution est appelée **stacktrace**. La liste montre chaque méthode appelée lors de l'exécution du programme.

- Les programmes capables de gérer les exceptions de manière appropriée sont plus tolérants aux pannes et robustes.
- Les applications à tolérance de pannes sont conçues pour continuer à fonctionner, éventuellement à un niveau réduit, en cas de défaillance d'une partie du système.

La robustesse représente la capacité d'un système à résister aux contraintes et à continuer à fonctionner.

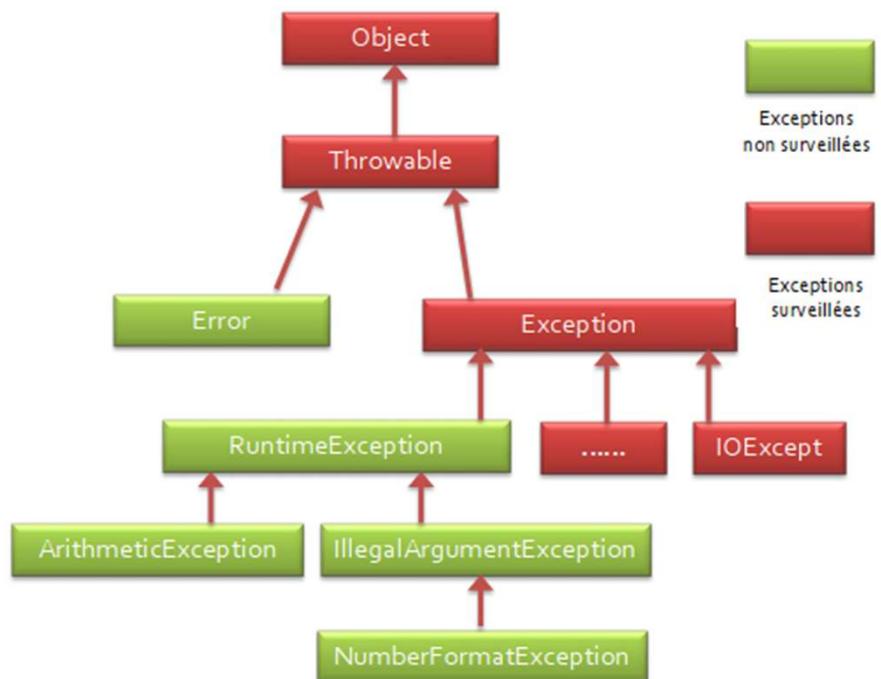


Figure 1 : Extrait de la hiérarchie des exceptions.

GESTION D'EXCEPTIONS EN JAVA

La classe Error

La classe **Error** représente des erreurs plus graves que votre programme ne peut généralement pas récupérer.

Par exemple, la mémoire.

La classe Exception

La classe **Exception** comprend les erreurs moins graves qui représentent des conditions inhabituelles survenant pendant l'exécution d'un programme et à partir desquelles le programme peut être restauré.

- Par exemple, l'indice d'un tableau dépassé.

Liste des exceptions Java

<https://programming.guide/java/list-of-java-exceptions.html>

HIÉRARCHIE DES EXCEPTIONS

Exceptions non surveillées

- Le compilateur Java ne vérifie pas les exceptions non surveillées.
- Elles se produisent pendant l'exécution et sont dues à des erreurs de logique du développeur (indice d'un tableau hors limites, division par zéro, méthode sur une référence null, ...) ou par exemple atteinte des limites des ressources du système.
 1. Elles n'ont pas l'obligation d'être gérées par un [catch](#).
 2. Les méthodes qui peuvent en lever ne doivent pas ([mais peuvent](#)) l'indiquer dans leur entête.
 3. Elles sont les exceptions du type [RuntimeException](#) et ses sous-classes

Exceptions surveillées

- Les exceptions surveillées sont vérifiées par le compilateur de Java.
- Les méthodes qui peuvent lever une exception surveillée doivent l'indiquer dans leur entête en précisant la clause [throws](#).
- Toutes les exceptions surveillées doivent explicitement être contrôlées avec un bloc [catch](#).
- L'exception remonte toute la pile d'appel de méthodes jusqu'à ce qu'un gestionnaire d'exception soit trouvé.
- Les exceptions surveillées comprennent toutes les exceptions du type [Exception](#) et ses sous-classes, *sauf la classe [RuntimeException](#) et ses sous-classes*.

GESTION D'EXCEPTIONS EN JAVA

Quelques méthodes utiles

Méthode	Description
public String getMessage()	Renvoie un message détaillé sur l'exception qui s'est produite. Ce message est initialisé dans le constructeur Throwable.
public String toString()	Renvoie le nom de la classe concaténée avec le résultat de getMessage () .
public Throwable getCause()	Renvoie la cause de l'exception représentée par un objet Throwable.

Capture et traitement des exceptions

Dans la terminologie orientée objet, vous essayez (**try**) une procédure pouvant provoquer une erreur. Une méthode qui détecte une condition d'erreur lève une exception (**throws**) et si vous écrivez un bloc de code qui traite l'erreur, ce bloc est dit intercepte l'exception (**catch**).

```
try{
    // traitements
}
catch(TypeException var){
    // gérer la condition d'erreur
}
```

TRY.. CATCH ET THROW

Try

Lorsque vous créez un segment de code dans lequel une exception peut survenir, vous placez le code dans un bloc `try {...}`

- ce bloc de code que vous essayez d'exécuter tout en reconnaissant qu'une exception pourrait se produire.

```
try{  
    // traitements  
}  
catch(TypeException var){  
    // gérer la condition d'erreur  
}
```

Catch... Throw

Un bloc `catch {...}` est un segment de code qui peut gérer une exception, peut être levée par le bloc `try` qui le précède.

L'exception pourrait être celle qui est lancée automatiquement, ou vous pourriez écrire explicitement une instruction `throw`.

- Une instruction `throw` est une instruction qui envoie un objet `Exception` à partir d'un bloc ou d'une méthode afin qu'il puisse être géré ailleurs.

EXEMPLE

Dans l'exemple ci-dessus,

- si la variable heros vaut null alors le traitement du bloc try est interrompu par une **NullPointerException**.
- Sinon le bloc continue à s'exécuter.
- Si la condition est vraie, le traitement du bloc est interrompu par le lancement d'une **FinDuMondeException** et le traitement reprend dans le bloc catch...

```
try {  
    if (heros == null) {  
        throw new NullPointerException("Le heros ne peut pas être nul !");  
    }  
  
    boolean victoire = heros.combattre(espritDuMal);  
    boolean planDejoue = heros.desamorcer(machineInfernale);  
  
    if (!victoire || !planDejoue) {  
        throw new FinDuMondeException();  
    }  
  
    heros.setPoseVictorieuse();  
}  
catch (FinDuMondeException fdme) {  
    // ...  
}
```

EXEMPLE

b étant un entier, lorsqu'une valeur illégale est tentée, une exception `InputMismatchException` est créée automatiquement et le bloc catch est exécuté.

Pour rappel, c'est une exception qui n'est pas capturé automatiquement par Java et donc qui nous incombe de traiter.

Ici, la demande de saisie s'effectue avec des String et donc si l'utilisateur envoie autre choses que des nombres alors l'addition ne sera pas possible, générant une exception de type `InputMismatchException`

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

public class Test {
    public static void main(String args[]) {
        int a, b;
        Scanner clavier = new Scanner(System.in);

        try {
            System.out.print("Saisir a : ");
            a = clavier.nextInt();

            System.out.print("Saisir b : ");
            b = clavier.nextInt();

            System.out.println("a+b = " + (a + b));
        } catch (InputMismatchException e) {
            System.out.println(e);
        }

        // fermer les ressources
        clavier.close();
    }
}
```

GESTION D'EXCEPTIONS EN JAVA

Multiples blocs catch

Vous pouvez placer autant d'instructions que nécessaire dans un bloc try et intercepter autant d'exceptions que vous le souhaitez.

Si vous essayez plusieurs instructions, seule la première instruction générant une erreur lève une exception.

Dès que l'exception se produit, la logique est transférée vers le bloc catch, ce qui laisse le reste des instructions du bloc try non exécutées.

Exemples

Exemple 1 : catch générique

```
try{  
    // traitements  
}  
catch(Exception e){  
    System.out.println(e);  
}
```

Exemple 2 : catch spécifique

```
try{  
    // traitements  
}  
catch(ArithmetricException, InputMismatchException e)  
{  
    System.out.println(e);  
}
```

LE BLOC FINALLY

Le code dans un bloc finally s'exécute que le bloc try précédent identifie une exception ou non. En règle générale, vous utilisez un bloc finally pour effectuer des tâches de nettoyage qui doivent être effectuées, que des exceptions se soient produites ou non.

```
try
{
    // instructions
}
catch(Exception e) {
    // actions si une exception a été lancée
}
finally
{
    // nettoyage
}
```

```
java.io.FileReader reader = new java.io.FileReader(filename);
try {
    int nbCharRead = 0;
    char[] buffer = new char[1024];
    StringBuilder builder = new StringBuilder();
    // L'appel à reader.read peut lancer une java.io.IOException
    while ((nbCharRead = reader.read(buffer)) >= 0) {
        builder.append(buffer, 0, nbCharRead);
    }
    // le retour explicite n'empêche pas l'exécution du block finally.
    return builder.toString();
} finally {
    // Ce block est obligatoirement exécuté après le block try.
    // Ainsi le flux de lecture sur le fichier est fermé
    // avant le retour de la méthode.
    reader.close();
}
```

TRY-WITH-RESSOURCES

La gestion des ressources peut également être réalisée par la syntaxe du try-with-resources.

Après le mot-clé try, on déclare entre parenthèses une ou plusieurs initialisations de variable.

Ces variables doivent être d'un type qui implémente l'interface [AutoCloseable](#) ou [Closeable](#).

Ces interfaces ne déclarent qu'une seule méthode : close.

Le compilateur ajoute automatiquement un bloc finally à la suite du bloc try pour appeler la méthode close sur chacune des variables qui ne valent pas null.

```
try (java.io.FileReader reader = new java.io.FileReader(filename)) {  
    int nbCharRead = 0;  
    char[] buffer = new char[1024];  
    StringBuilder builder = new StringBuilder();  
    while ((nbCharRead = reader.read(buffer)) >= 0) {  
        builder.append(buffer, 0, nbCharRead);  
    }  
    return builder.toString();  
}
```

PROPAGATION D'UNE EXCEPTION

Throws :

Si une méthode lève une exception mais qu'une méthode différente interceptera, vous devez créer une clause `throws` suivi du type d'exception dans l'entête de la méthode. (spécification d'exception)

```
public class listPrix {
    private static final double[] prix = {15.99, 27.88, 34.56, 45.89};
    public static void afficherPrix(int elem) throws IndexOutOfBoundsException
    {
        System.out.println("le prix est : " + prix[elem]);
    }
}
```

- Si vous générez **une exception vérifiée** à partir d'une méthode, vous devez effectuer l'une des opérations suivantes soit :
 - Attraper l'exception "catch" dans la méthode.
 - Spécifier l'exception dans la clause `throws` dans l'en-tête de la méthode.
- Si vous écrivez une méthode avec une clause `throws` dans l'en-tête, toutes méthodes utilisant votre méthode doit effectuer l'une des opérations suivantes soit :
 - Intercepter et gérer l'exception possible.
 - Déclarer l'exception dans sa clause `throws`. La méthode appelée peut alors renvoyer l'exception à une autre méthode qui pourrait l'attraper ou la lancer à nouveau.
- Si vous écrivez une méthode qui lève explicitement **une exception vérifiée** qui n'est pas interceptée dans la méthode, Java requiert que vous utilisez la clause `throws` dans l'en-tête de la méthode.
- Vous incluez la clause `throws` dans l'en-tête de méthode afin que les applications qui utilisent vos méthodes soient informées du risque d'exception.

GESTION D'EXCEPTIONS EN JAVA

Créez vos propres classes d'exception

Java : 40 catégories d'exceptions mais on ne peut pas tout prédire.

- La solution : Création de sa propre classe d'exceptions.

Votre classe **MaClasseException** doit étendre la classe **Exception**

La classe Exception

4 constructeurs :

- **Exception()** - Construit un nouvel objet **Exception** avec la valeur null comme message de détail.
- **Exception(String message)** - Construit un nouvel objet **Exception** avec le message de détail spécifié.
- **Exception(String message, Throwable cause)** - construit un nouvel objet **Exception** avec le message de détail spécifié et la cause.
- **Exception(Throwable cause)** - construit un nouvel objet **Exception** avec la cause spécifiée et un message détaillé de `cause.toString()`, qui contient généralement la classe et le message détaillé de `cause`, ou null si l'argument de la cause est null.

EXEMPLE

Création d'une classe `CompteException` qui contient une seule instruction qui transmet la description d'une erreur au constructeur Mère `Exception`. Ce message sera récupéré au travers de la méthode `getMessage()` avec un objet `CompteException`.

Conseils :

Vous ne devez pas créer un nombre excessif de types d'exception spéciaux pour vos classes, en particulier si l'environnement de développement Java contient déjà une classe `Exception` qui interceptera l'erreur.

Toutefois, lorsque cela est approprié, les classes `Exception` spécialisées constituent un moyen élégant de gérer les situations d'erreur.

```
class CompteException extends Exception {
    public CompteException() {
        super("le solde de votre compte est négatif");
    }
}

public class Gestion {
    public Gestion(double salaire) throws CompteException {
        if (salaire < 0) {
            // lever une exception
            throw (new CompteException());
        }
    }
}

public class Test {
    public static void main(String args[]) throws CompteException {
        Scanner clavier = new Scanner(System.in);
        System.out.print("Saisir votre salaire horaire : ");
        double salaire = clavier.nextDouble();
        try {
            Gestion cpt = new Gestion(salaire);
            System.out.println("le salaire est " + salaire);
        } catch (CompteException e) {
            System.out.println("Erreur : " + e.getMessage());
        }

        // sinon
        clavier.close();
    }
}
```

EXEMPLE DE GESTION DES EXCEPTIONS PAR L'EXEMPLE

```

public static void main(String[] args) {
    // TODO Auto-generated method stub

    DAO<Login> loginDAO = new LoginDAO();
    Login login;

    try {
        login = new Login(1, "Formateur", "@form");

        if ( loginDAO.create(login) ) {
            System.out.println("Login enregistré");
        } else {
            System.out.println("login non enregistré");
        }

    } catch (ReportException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

```

Comme c'est dans notre Main que nous avons positionnés le **try ... catch**, le catch capture l'exception et la traite selon nos instructions (affichage, log etc...)

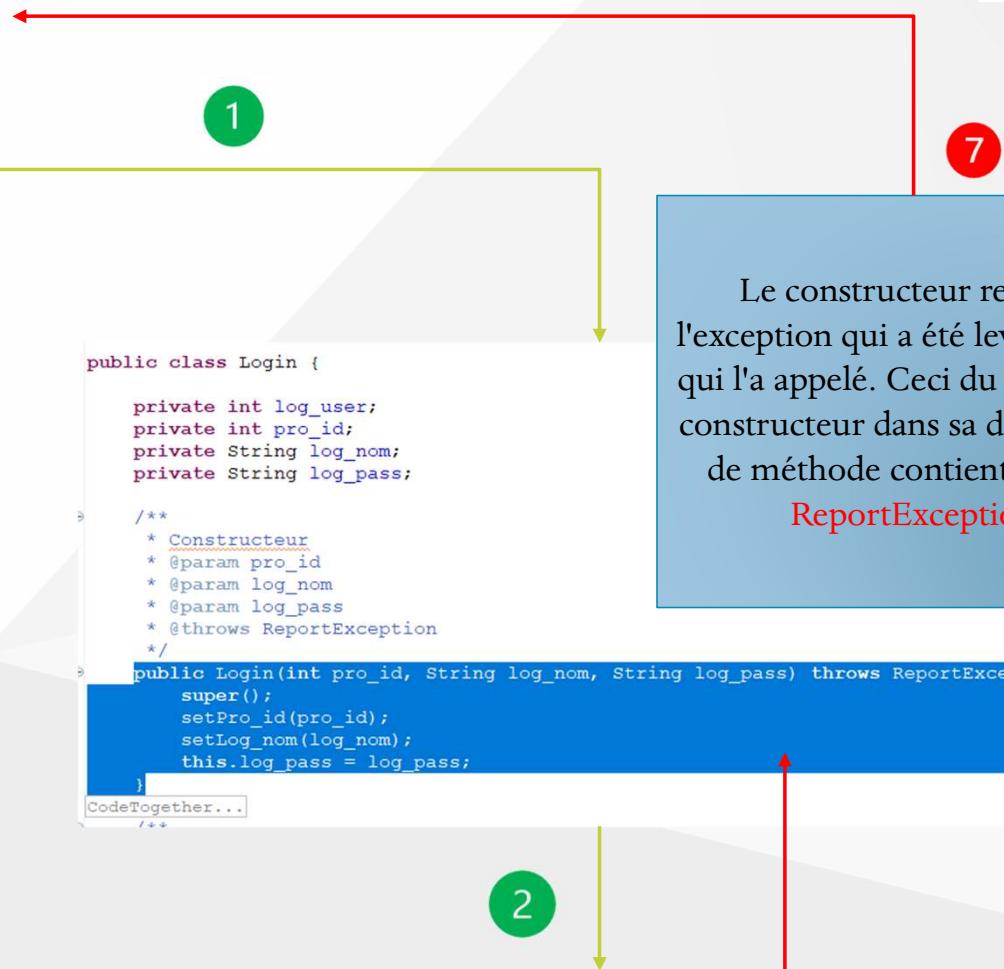
```

public class Login {

    private int log_user;
    private int pro_id;
    private String log_nom;
    private String log_pass;

    /**
     * Constructeur
     * @param pro_id
     * @param log_nom
     * @param log_pass
     * @throws ReportException
     */
    public Login(int pro_id, String log_nom, String log_pass) throws ReportException {
        super();
        setPro_id(pro_id);
        setLog_nom(log_nom);
        this.log_pass = log_pass;
    }
}

```



Le constructeur renvoie l'exception qui a été levée à celui qui l'a appelé. Ceci du fait que le constructeur dans sa déclaration de méthode contient **throws ReportException**

ReportException

TESTS SUR MES SETTER de la saisie avec possibilité de déclencher une Exception

3

```
/**
 *
 * @param log_nom
 * @throws ReportException
 */
public void setLog_nom(String log_nom) throws ReportException {

    if (log_nom == null || log_nom.isEmpty() ) {
        throw new ReportException("Problème de Saisie : Le nom ne peut être vide");
    }
    this.log_nom = log_nom;
}

/**
 *
 * @return
 */
public String getLog_pass() {
    return log_pass;
}

/**
 *
 * @param log_pass
 * @throws ReportException
 */
public void setLog_pass(String log_pass) throws ReportException {
    if (log_pass == null || log_pass.isEmpty() ) {
        throw new ReportException("Problème de Saisie : Le mot de passe ne peut être vide");
    } else if (!checkLogPass(log_pass) ) {
        throw new ReportException(
            "Problème de Saisie : les critères du mot de passe ne sont pas bonnes");
    }

    this.log_pass = hashPass(log_pass);
}
```

4



5

En cas de non-respect des règles, alors je lève une exception en la créant par **Throw new ReportException(...)** classe que j'ai créé pour gérer mes propres messages d'erreurs.

Cette exception créée est alors "lancée" par le setter à l'appelant donc ici le constructeur

6

GESTION DES FICHIERS

LECTURE ET ÉCRITURE DE FICHIERS

GESTION DES FICHIERS

Les données peuvent être stockés sur deux grands types de périphériques de stockage dans un SI

Vous pouvez classer les fichiers en fonction de la manière dont ils stockent les données

Le **stockage volatile** (temporaire) : les variables sont perdues à l'extinction du l'ordinateur (stocké sur la RAM).

Le **stockage non volatile** est un stockage permanent. Les données sont stockées sur des supports de stockages tels qu'un disque dur.

Les fichiers texte contiennent des données qui peuvent être lues dans un éditeur de texte car elles ont été codées à l'aide d'un schéma tel que ASCII ou Unicode.

Les fichiers binaires contiennent des éléments de données qui n'ont pas été codés en tant que texte. Leur contenu est au format binaire, ce qui signifie que vous ne pouvez pas les comprendre en les affichant dans un éditeur de texte. Les exemples incluent les images, la musique et les fichiers de programme compilés avec une extension .class

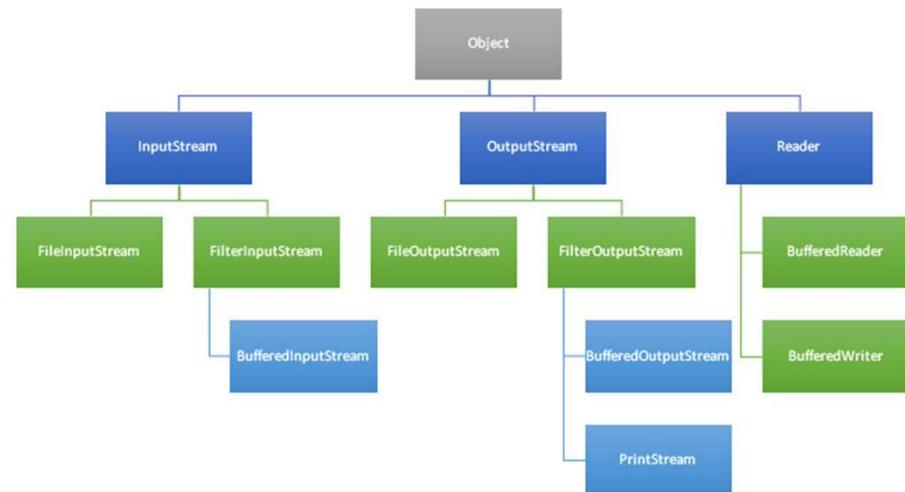
ENTRÉES / SORTIES

En Java, les E/S sont représentées par des objets de type `java.io.InputStream`, `java.io.Reader`, `java.io.OutputStream` et `java.io.Writer`.

Le package `java.io` définit donc l'ensemble de classes qui vont pouvoir être utilisées conjointement pour réaliser des traitements E/S

Les flux de données sont généralement attachés à des ressources système :

- Impératif de les fermer après utilisation : `close()`
- Gestion des `IOException` : encadrement des méthodes d'appel par un bloc `try...catch`



CLASSE INPUTSTREAM

La classe `InputStream` est une classe abstraite qui représente un flux d'entrée *de données binaires*.

- Elle déclare des méthodes `read` qui permettent de lire des données octet par octet ou bien de les copier dans un tableau.

Par exemple, `FileInputStream` permet d'ouvrir un flux de lecture binaire sur un fichier.

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;

public class TestFileInputStream {

    public static void main(String[] args) throws IOException {

        try (InputStream stream = new FileInputStream("/chemin/vers/mon/fichier.bin")) {
            byte[] buffer = new byte[1024];
            int nbRead;
            while ((nbRead = stream.read(buffer)) != -1) {
                // ...
            }
        }
    }
}
```

CLASSE OUTPUTSTREAM

La classe `OutputStream` est une classe abstraite qui représente un flux de sortie *de données binaires*.

- Elle déclare des méthodes `write` qui permettent d'écrire des données octet par octet ou bien de les écrire depuis un tableau. La classe `OutputStream` fournit également la méthode `flush` pour forcer l'écriture de la zone tampon.

Par exemple, `FileOutputStream` permet d'ouvrir un flux d'entrée binaire sur un fichier.

```
import java.io.FileOutputStream;
import java.io.IOException;

public class TestFileOutputStream {

    public static void main(String[] args) throws IOException {
        try (FileOutputStream stream = new FileOutputStream("chemin/vers/mon/fichierdesortie.bin")) {
            byte[] octets = "hello the world".getBytes();
            stream.write(octets);
        }
    }
}
```

CLASSE READER

La classe `Reader` est une classe abstraite qui permet de lire des flux de caractères.

- Comme `InputStream`, la classe `Reader` fournit des méthodes `read` mais qui acceptent en paramètre des caractères.

Par exemple, la classe `StringReader` permet de parcourir une chaîne de caractères sous la forme d'un flux de caractères.

La classe `FileReader` permet de lire le contenu d'un fichier texte.

```
import java.io.IOException;
import java.io.Reader;
import java.io.StringReader;

public class TestStringReader {

    public static void main(String[] args) throws IOException {
        Reader reader = new StringReader("hello the world");

        int caractere;
        while ((caractere = reader.read()) != -1) {
            System.out.print((char) caractere);
        }
    }

    import java.io.FileReader;
    import java.io.IOException;
    import java.io.Reader;

    public class TestFileReader {

        public static void main(String[] args) throws IOException {
            try (Reader reader = new FileReader("/le/chemin/du/fichier.txt")) {
                char[] buffer = new char[1024];
                int nbRead;
                while ((nbRead = reader.read(buffer)) != -1) {
                    // ...
                }
            }
        }
    }
}
```

CLASSE WRITER

La classe `Writer` est une classe abstraite qui permet d'écrire des flux de caractères.

- Comme `OutputStream`, la classe `Writer` fournit des méthodes `write` mais qui acceptent en paramètre des caractères.

Elle fournit également des méthodes `append` qui réalisent le même type d'opérations et qui retournent l'instance du `Writer` afin de pouvoir chaîner les appels. Il existe plusieurs classes qui en fournissent une implémentation concrète.

Par exemple, la classe `StringWriter` ou `FileWriter`

```
import java.io.IOException;
import java.io.StringWriter;

public class TestStringWriter {

    public static void main(String[] args) throws IOException {
        StringWriter writer = new StringWriter();

        writer.append("Hello")
            .append(' ')
            .append("the")
            .append(' ')
            .append("world");

        String resultat = writer.toString();

        System.out.println(resultat);
    }
}

import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;

public class TestFileWriter {

    public static void main(String[] args) throws IOException {
        try (Writer writer = new FileWriter("/chemin/vers/mon/fichier.txt", true)) {
            writer.append("Hello world!\n");
        }
    }
}
```

CLASSE SCANNER

La classe `java.util.Scanner` agit comme un décorateur pour différents types d'instance qui représentent une entrée.

- Elle permet de réaliser des opérations de lecture et de validation de données plus complexes que les classes du packages `java.io`.

```
import java.util.Scanner;

public class TestScanner {

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

        System.out.print("Saisissez une chaîne de caractères : ");
        String chaine = scanner.nextLine();

        Integer nombre = null;
        do {
            System.out.print("Saisissez un nombre : ");
            if (!scanner.hasNextInt()) {
                scanner.next();
                System.err.println("Ceci n'est pas un nombre valide");
                continue;
            }
            nombre = scanner.nextInt();
        } while (nombre == null);

        String identifiant = null;
        do {
            System.out.print("Saisissez les 8 caractères de votre identifiant : ");
            // On utilise une expression régulière pour vérifier le prochain token
            if (!scanner.hasNext(".{8}")) {
                scanner.next();
                System.err.println("Ceci n'est pas un identifiant valide");
                continue;
            }
            identifiant = scanner.next();
        } while (identifiant == null);

        System.out.println("Vous avez saisi :");
        System.out.println(chaine);
        System.out.println(nombre);
        System.out.println(identifiant);
    }
}
```

CLASSE FILE

Le package `java.io` fournit la classe `File` qui représente un fichier.

À travers, cette classe, il est possible de savoir si le fichier existe, s'il s'agit d'un répertoire... On peut également créer le fichier ou le supprimer.

- L'interface `Path` représente un chemin de fichier de manière générique.
- Les classes `Paths` et `FileSystem` servent à construire des instances de type `Path`.
- La classe `FileSystem` fournit également des méthodes pour obtenir des informations à propos du ou des systèmes de fichiers présents sur la machine.
- On peut accéder à une instance de `FileSystem` grâce à la méthode `FileSystems.getDefault()`.

```
import java.io.File;
import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.Path;
import java.nio.file.Paths;

public class TestPath {

    public static void main(String[] args) throws IOException {
        Path cheminFichier = Paths.get("home", "david", "fichier.txt");

        System.out.println(cheminFichier); // home/david/fichier.txt
        System.out.println(cheminFichier.getNameCount()); // 3
        System.out.println(cheminFichier.getParent()); // home/david
        System.out.println(cheminFichier.getFileName()); // fichier.txt

        cheminFichier = FileSystems.getDefault().getPath("home", "david", "fichier.txt");
        File fichier = cheminFichier.toFile();

        // maintenant on peut utiliser le fichier
    }
}
```

SÉRIALISATION D'OBJETS

CRÉER UNE PERSISTANCE DANS UN
FICHIER.

LA SÉRIALISATION D'OBJETS

Les classes `ObjectOutputStream` et `ObjectInputStream` permettent de réaliser la **sérialisation/désérialisation** d'objets :

- un objet (et tous les objets qu'il référence) peut être écrit dans un flux ou lu depuis un flux.

Cela peut permettre de sauvegarder dans un fichier un état de l'application.

La sérialisation d'objets a des limites :

- Le format est propre à Java.
- Très dépendant de la structure des classes.

Pour qu'un objet puisse être sérialisé, il faut que sa classe implémente l'interface marqueur `Serializable`.

- Si un objet référence d'autres objets dans ses attributs alors il faut également que les classes de ces objets implémentent l'interface `Serializable`.
- *Beaucoup de classes de l'API standard de Java implémentent l'interface Serializable, à commencer par la classe String.*

Pour qu'un objet ne puisse être sérialisé, il faut indiquer par le mot-clé `transient` devant l'objet concerné.

- `Private transient list<Person> children = new ArrayList<>;`
 - *Cela fera en sorte de ne pas sérialiser la liste des enfants.*

Les classes qui implémentent `Serializable` possèdent un numéro de version interne qui change à la compilation.

- Ce numéro de version est sérialisé avec les objets.
- Si des changements majeurs dans la classe ont eu lieu, il est facile de comparer les numéros de version à la désérialisation et ainsi d'indiquer que la classe n'est pas compatible avec la version actuelle.

EXEMPLE

Ici, nous implémentons la classe Person afin de la sérialiser.

Pour cela, la classe implémente **Serializable**.

Cette classe crée une personne avec une liste d'enfants.

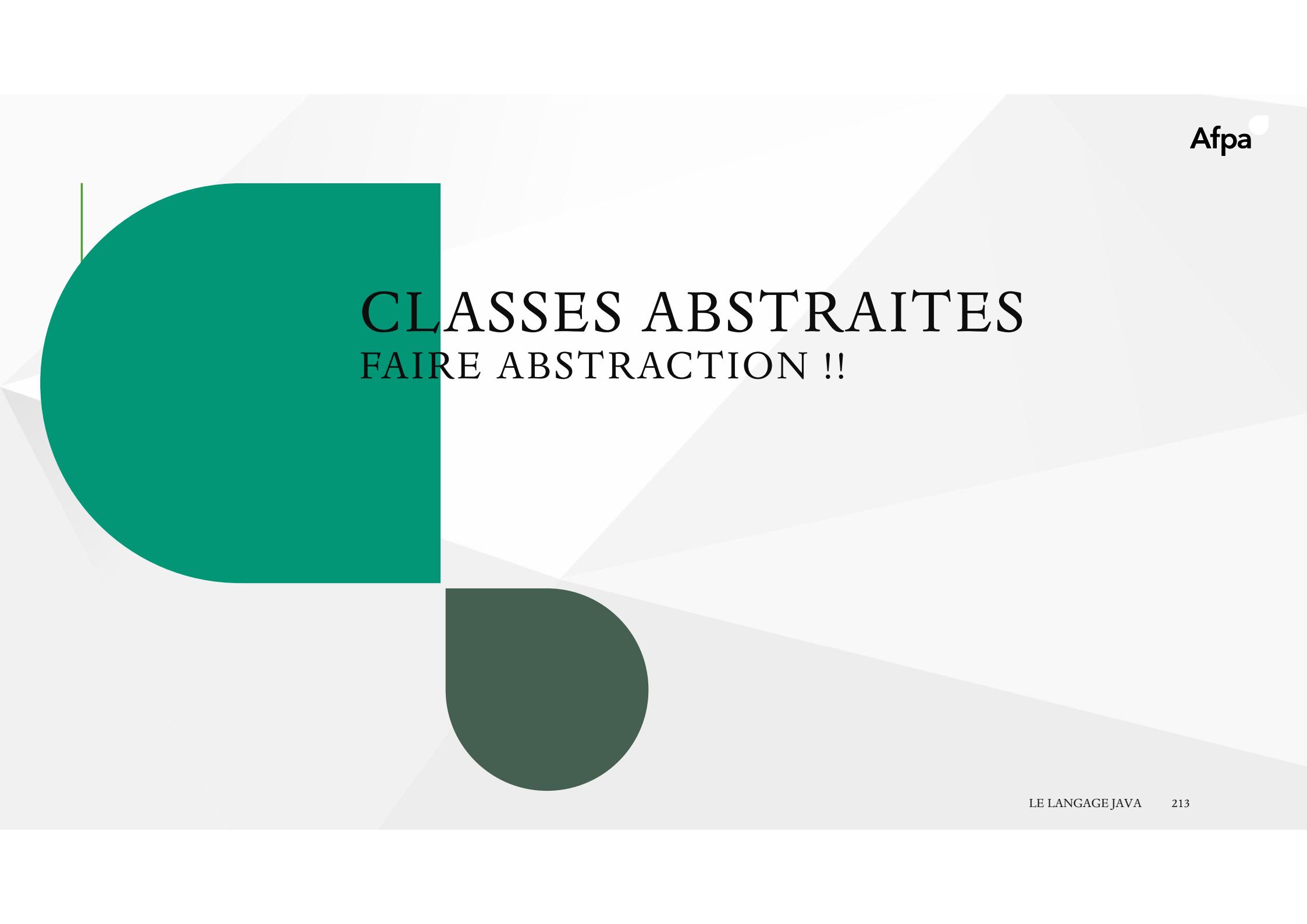
```
public class Person implements Serializable {  
    /**  
     *  
     */  
    private static final long serialVersionUID = 1L;  
  
    private String lastname;  
    private String firstname;  
    private List<Person> children = new ArrayList<>();  
  
    public Person(String prenom, String nom) {  
        this.firstname = prenom;  
        this.lastname = nom;  
    }  
  
    public String getNom() {  
        return lastname;  
    }  
  
    public String getPrenom() {  
        return firstname;  
    }  
  
    public void addChildren(Person... pchildren) {  
        Collections.addAll(this.children, pchildren);  
    }  
  
    public List<Person> getChildren() {  
        return children;  
    }  
  
    @Override  
    public String toString() {  
        return this.firstname + " " + this.lastname;  
    }  
}
```

EXEMPLE

Dans la classe Serial, deux méthodes sont implémentées :

- 1. Une méthode pour sérialiser
 - 1. Ouvre un flux d'écriture vers le fichier bin
 - 2. Création d'un `ObjectOutputStream` qui va nous permettre d'écrire dans le fichier.
 - 3. Ecriture des objets Person dans le flux de sortie
- Une méthode pour désérialiser.
 - 1. Ouvre un flux de lecture vers le fichier bin
 - 2. Création d'un `ObjectInputStream` qui va nous permettre de lire le contenu du fichier.
 - 3. Lecture du contenu du fichier avec enregistrement dans un objet Person.

```
public class Serial {  
    public static void setSerial() throws IOException {  
        Person person = new Person("Donald", "Duck");  
        person.addChildren(  
            new Person("Riri", "Duck"),  
            new Person("Fifi", "Duck"),  
            new Person("Loulou", "Duck"),  
            new Person("neo", "Jero")  
        );  
  
        OutputStream outputStream = Files.newOutputStream(Paths.get("ressources/genealogy.bin"));  
  
        ObjectOutputStream objectStream = new ObjectOutputStream(outputStream);  
        objectStream.writeObject(person);  
    }  
  
    public static void getSerial() throws IOException, ClassNotFoundException {  
        InputStream outputStream = Files.newInputStream(Paths.get("ressources/genealogy.bin"));  
  
        ObjectInputStream objectStream = new ObjectInputStream(outputStream);  
        Person person = (Person) objectStream.readObject();  
  
        System.out.println(person);  
        for (Person children : person.getChildren()) {  
            System.out.println(children);  
        }  
    }  
}
```



CLASSES ABSTRAITES

FAIRE ABSTRACTION !!

CLASSES ABSTRAITES EN JAVA

Définition

Une classe concrète est une classe à partir de laquelle vous pouvez instancier des objets.

Parfois, une classe est si générale que vous n'avez jamais l'intention de créer des instances spécifiques de la classe.

Le seul but de la création de classes abstraites est de permettre à d'autres classes de les hériter.

Si vous essayez d'instancier un objet à partir d'une classe abstraite, vous recevez un message d'erreur du compilateur indiquant que vous avez commis une erreur InstantiationException.

Utilisation

Vous utilisez le mot-clé `abstract` lorsque vous déclarez une classe abstraite.

Les classes abstraites peuvent inclure deux types de méthodes:

- `Les méthodes non abstraites`, comme celles que vous pouvez créer dans n'importe quelle classe, sont implémentées dans la classe abstraite et sont simplement héritées par ses enfants
- `Les méthodes abstraites` n'ont pas de corps (pas d'accolades ni de déclarations) et elles doivent être implémentées (corps donnés) dans des classes enfant.

CLASSES ABSTRAITES EN JAVA

Les classes abstraites contiennent généralement au moins une méthode abstraite.

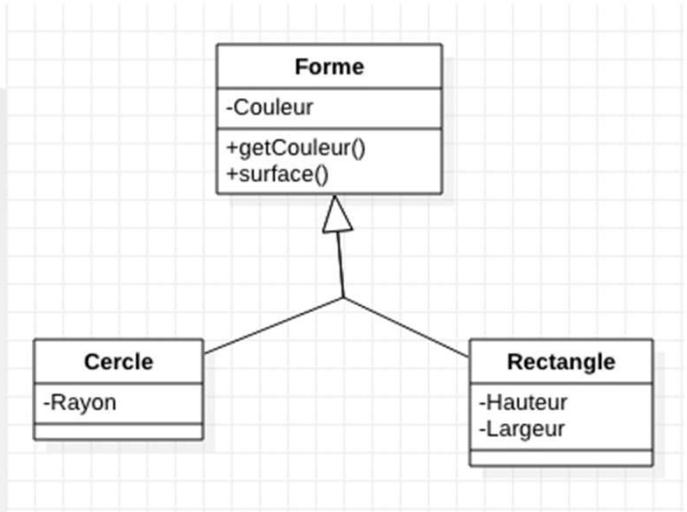
Lorsque vous créez une méthode abstraite, vous fournissez le mot clé **abstract** et le reste de l'en-tête de la méthode, y compris le type, le nom et les paramètres de la méthode.

Cependant, la déclaration se termine là. Vous ne fournissez aucune accolade ni aucune instruction dans la méthode, mais seulement un point-virgule à la fin de la déclaration.

Exemple : public abstract afficher();

- Si vous déclarez une classe abstraite, chacune de ses méthodes peut être abstraite ou non.
- Si vous déclarez une méthode abstraite, vous devez également déclarer sa classe abstraite.
- **Toutes classes héritant d'une classe mère abstraites doivent implémenter les méthodes déclarées abstraites au sein de la classe mère.**

EXEMPLE



```

public class Test {

    public static void main(String args[]) {
        Forme f1 = new Cercle("Rouge", 2.2);
        System.out.println("Surface : " + f1.surface());

        System.out.println("-----");

        Forme f2 = new Rectangle("Vert", 2, 4);
        System.out.println("Surface : " + f2.surface());
    }
}
  
```

```

abstract class Forme {
    String couleur;

    // méthode abstraite
    abstract double surface();

    // constructeur
    public Forme(String couleur) {
        System.out.println("Constructeur Forme");
        this.couleur = couleur;
    }

    // méthode concrète
    public String getcouleur() {
        return couleur;
    }
}
  
```

EXEMPLE

```

class Cercle extends Forme {
    double rayon;

    public Cercle(String couleur, double rayon) {
        // appeler le constructeur de la classe mère Forme
        super(couleur);
        System.out.println("Constructeur Cercle");
        this.rayon = rayon;
    }

    @Override
    double surface() {
        return Math.PI * Math.pow(rayon, 2);
    }
}

```

```

class Rectangle extends Forme {

    double hauteur;
    double largeur;

    public Rectangle(String couleur, double hauteur, double largeur) {
        // appeler le constructeur de la classe mère Forme
        super(couleur);
        System.out.println("Constructeur Rectangle");
        this.hauteur = hauteur;
        this.largeur = largeur;
    }

    @Override
    double surface() {
        return hauteur * largeur;
    }
}

public class Test {

    public static void main(String args[]) {
        Forme f1 = new Cercle("Rouge", 2.2);
        System.out.println("Surface : " + f1.surface());

        System.out.println("-----");

        Forme f2 = new Rectangle("Vert", 2, 4);
        System.out.println("Surface : " + f2.surface());
    }
}

```

CLASSES ABSTRAITES EN JAVA

Intérêt :

Le recours aux classes abstraites facilite largement la Conception Orientée Objet.

En effet, on peut placer dans une classe abstraite toutes les fonctionnalités dont on souhaite disposer pour toutes ses descendantes :

- soit sous forme d'une implémentation complète de méthodes (non abstraites) et de champs (privés ou non) lorsqu'ils sont communs à toutes ses descendantes,
- soit sous forme d'interface de méthodes abstraites dont on est alors sûr qu'elles existeront dans toute classe dérivée instanciable.

C'est cette certitude de la présence de certaines méthodes qui permet d'exploiter le **polymorphisme**, et ce dès la conception de la classe abstraite, alors même qu'aucune classe dérivée n'a peut-être encore été créée.

Notamment, on peut très bien écrire des canevas recourant à des méthodes abstraites.

INTERFACES INTERFACER !!

LES INTERFACES

Une interface permet de définir un ensemble de services qu'un client peut obtenir d'un objet.

- Une interface introduit une abstraction pure qui permet un découplage maximal entre un service et son implémentation.

On retrouve ainsi les interfaces au cœur de l'implémentation de beaucoup de bibliothèques et de frameworks.

- Le mécanisme des interfaces permet d'introduire également une forme simplifiée d'héritage multiple.

- Une interface se déclare avec le mot-clé **interface**.
- Une interface décrit un ensemble de méthodes en fournissant uniquement leur signature.
 - Les méthodes d'une interface sont par défaut **public** et **abstract**. Il n'est pas possible de déclarer une autre portée que **public**.
- **Une classe peut ensuite implémenter une ou plusieurs interfaces.**

LES INTERFACES EN JAVA

Propriétés des interfaces

4. Semblables aux classes, les interfaces peuvent hériter d'autres interfaces.
5. Par essence, les méthodes d'une interface sont abstraites.

```
interface A {  
    ...  
}  
interface B {  
    ...  
}  
Interface C extends A, B {  
    ...  
}
```

Interfaces et classes abstraites

Pourquoi les interfaces et pas une classe abstraite ?

L'utilisation d'une classe de base abstraite pour exprimer une propriété générique pose un problème majeur. Une classe ne peut hériter que d'une seule classe.

D'autres langages de programmation, en particulier C++, permettent à une classe d'avoir plusieurs super-classes. Cette fonctionnalité s'appelle l'héritage multiple.

Cela rend le langage très complexe (comme en C++) ou moins efficace (comme dans Eiffel).

Au lieu de cela, les interfaces offrent la plupart des avantages d'un héritage multiple tout en évitant les complexités et les inefficacités.

LES INTERFACES EN JAVA

Définition

Si l'on considère une classe abstraite **n'implantant aucune méthode** et aucun champ (hormis des constantes), on aboutit à la notion d'interfaces.

En Java, une interface n'est pas une classe mais **un ensemble d'exigences pour les classes qui souhaitent s'y conformer**.

```
public interface I
{
    void f(int n); // en-tête d'une méthode f (public abstract facultatif)
    void g();      // en-tête d'une méthode f (public abstract facultatif)
}
```

```
public class Professeur implements Comparable{
    private String nom;
    private double salaire;

    public Professeur(String nom, double salaire){
        this.nom=nom;
        this.salaire=salaire;
    }

    public int compareTo(Object AutreProf)
    {
        Professeur p= (Professeur)AutreProf;
        return Double.compare(salaire, p.salaire);
    }
}
```

Propriétés des interfaces

1. Toutes les méthodes d'une interface sont automatiquement publiques (pas besoin de l'indiquer) et n'ont pas d'implémentation.
2. **Vous ne pouvez jamais utiliser** l'opérateur **new** pour instancier une interface.
3. **Même si on ne peut instancier une interface**, on a le droit de créer des variables d'interface et d'y faire référence à un objet d'une classe qui implémente une interface.
 1. Comparable x;
 2. Comparable x = new Professeur("John",7000);
 3. If (x instanceof Comparable) {...}

LES INTERFACES EN JAVA

Méthodes statiques et privées

Depuis Java 8, vous pouvez ajouter des méthodes statiques aux interfaces.

Semblable à une classe, nous pouvons accéder aux méthodes statiques d'une interface en utilisant ses références.

`Interface.methodeStatic();`

Depuis Java 9, possible d'instancier des méthodes private et private static.

- Ce type de méthodes servent comme méthode d'aide qui fournissent le support à d'autres méthodes dans les interfaces.

Méthode par défaut (default)

Depuis Java 8, des méthodes avec implémentation ont été introduite dans une interface.

Pour déclarer les méthodes par défaut :

```
default void afficher() {  
    // corps de la méthode (implémentation)  
}
```

Si plusieurs classes implémentent une même méthode d'une interface, nous devrions alors revenir sur toutes nos classes en cas de modifications. **Ce serait fastidieux et source d'erreurs.**

RÉSOUDRE LES CONFLITS DE MÉTHODES PAR DÉFAUT

Que se passe-t-il si la même méthode est définie comme méthode par défaut dans une interface, puis à nouveau comme méthode d'une superclasse ou d'une autre interface ?

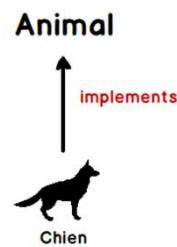
les règles en Java sont beaucoup plus simples. Les voici :

- Les superclasses gagnent. Si une super-classe fournit une méthode concrète, les méthodes par défaut portant le même nom et les mêmes types de paramètres sont simplement ignorées.
- Conflit d'interfaces. Si une interface fournit une méthode par défaut et qu'une autre interface contient une méthode avec le même nom et les mêmes types de paramètres (par défaut ou non), vous devez résoudre le conflit en surchargeant cette méthode.

CLASSE ABSTRAITE VS INTERFACE COMPARAISON

RAPPEL : INTERFACE VS CLASSE ABSTRAITE

Interface



Une interface peut être utilisé pour implémenter une classe.

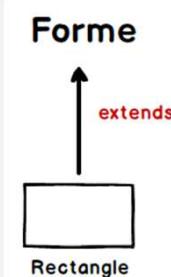
L'interface ne contient aucune méthode qui comporte du code. Toutes les méthodes d'une interface sont des méthodes abstraites.

Une interface ne peut pas être instanciée.

En revanche, les classes qui implémentent des interfaces peuvent être instanciées.

Les interfaces ne contiennent jamais de variables d'instance, mais peuvent contenir des variables « public static final ».

Classe abstraite



Une classe dont le mot clé est « abstract » est appelée une classe abstraite.

Les classes abstraites doivent avoir au moins une méthode abstraite, c'est-à-dire des méthodes sans corps. Il peut avoir plusieurs méthodes concrètes (méthodes qui ont du code).

Les classes abstraites vous permettent de créer des modèles pour des classes concrètes. Mais la classe qui hérite doit implémenter la méthode abstraite.

Les classes abstraites ne peuvent pas être instanciées.

DIFFÉRENCES ENTRE ABSTRAITE ET INTERFACE

Classe abstraite

Le mot-clé **abstract** en Java est utilisé pour créer ou déclarer une classe abstraite.

Une classe peut hériter des propriétés et méthodes d'une classe abstraite en utilisant le mot-clé `extends`.

Une classe abstraite peut avoir des méthodes abstraites ou non abstraites définies en elle.

- Les méthodes abstraites sont celles pour lesquelles aucune implémentation n'est fournie.

Une classe abstraite peut contenir des variables finales ou non finales (attributs de classe). Il peut également contenir des attributs statiques ou non statiques.

Interface

En Java, le mot-clé **interface** est utilisé pour créer ou déclarer une nouvelle interface.

Pour implémenter une Interface en Java, on peut utiliser le mot-clé `implements`

Une interface ne peut contenir que des méthodes abstraites. Nous ne pouvons fournir que la définition de la méthode mais pas son implémentation. Après Java 8, nous pouvons également avoir des méthodes par défaut et statiques dans les interfaces.

Une interface ne peut contenir que des membres statiques et finaux, et aucun autre type de membre n'est autorisé.

DIFFÉRENCES ENTRE ABSTRAITE ET INTERFACE

Classe abstraite

Une classe abstraite peut implémenter une interface et implémenter les méthodes de l'interface.

Une classe abstraite peut étendre d'autres classes et peut également implémenter des interfaces.

Java ne prend pas en charge les héritages multiples via des classes. Les classes abstraites, comme toute autre classe, ne prennent pas en charge les héritages multiples.

Les membres ou attributs de classe abstraits peuvent être privés, protégés ou publics.

Interface

Une interface ne peut étendre aucune autre classe et ne peut pas surcharger ou implémenter des méthodes de classe abstraite.

Comme discuté dans le point précédent, les interfaces ne peuvent pas étendre d'autres classes. Mais il n'y a aucune restriction dans l'implémentation d'une interface.

La prise en charge des héritages multiples en Java est fournie via les interfaces. C'est parce que les interfaces fournissent une abstraction complète.

Les attributs ou les membres d'une Interface sont toujours publics.

QUAND UTILISER LA CLASSE ABSTRAITE ET L'INTERFACE ?

- Les classes abstraites peuvent fournir une abstraction partielle ou complète.
- Une classe parent abstraite peut être créée pour quelques classes qui ont des fonctionnalités communes.
- Les classes abstraites sont également privilégiées si vous souhaitez plus de liberté d'action.
- Les interfaces, en revanche, fournissent toujours une abstraction complète.
- Les interfaces sont privilégiées lorsque l'on veut définir une structure de base. Le développeur peut alors construire n'importe quoi avec cette structure.
- Les interfaces prennent également en charge les héritages multiples. Ainsi, une seule classe peut implémenter plusieurs interfaces.

CLASSES INTERNE INNER CLASS

INNER CLASS

La plupart du temps, une classe en Java est déclarée dans un fichier portant le même nom que la classe avec l'extension .java.

- Cependant, il est également possible de déclarer des classes dans une classe.

On parle alors de classes internes ([inner classes](#)).

Cela est également possible, dans une certaine limite, pour les interfaces et les énumérations.

La déclaration des classes internes peut se faire dans l'ordre que l'on souhaite à l'intérieur du bloc de déclaration de la classe englobante.

Les classes internes [peuvent être ou non](#) déclarées **static**.

- Ces deux cas correspondent à deux usages particuliers des classes internes.

LES CLASSES INTERNE STATIC

Les classes internes déclarées static sont des classes pour lesquelles l'espace de noms est celui de la classe englobante.

Une classe interne **static** est souvent utilisée pour éviter de séparer dans des fichiers différents de petites classes utilitaires et ainsi de faciliter la lecture du code.

Pour une classe interne static :

- Son nom complet inclut le nom de la classe englobante (qui agit comme un package).
 - `package.ClasseEnglobante.ClasseInterne`
- La classe englobante et la classe interne partage le même espace privé.
 - Cela signifie que les attributs et les méthodes privés déclarés dans la classe englobante sont accessibles à la classe interne.
 - Réciproquement, la classe englobante peut avoir accès aux éléments privés de la classe interne.
- Une instance de la classe interne n'a accès directement qu'aux attributs et aux méthodes de la classe englobante qui sont déclarés static.

ILLUSTRATION

la classe Individu fournit publiquement une implémentation d'un Comparator qui permet de comparer deux instances en fonction de leur nom et de leur prénom.

```
Individu[] individus = {
    new Individu("John", "Eod"),
    new Individu("Annabel", "Doe"),
    new Individu("John", "Doe")
};

Arrays.sort(individus, new Individu.Comparateur());

System.out.println(Arrays.toString(individus));
```

```
import java.util.Comparator;

public class Individu {

    public static class Comparateur implements Comparator<Individu> {
        @Override
        public int compare(Individu i1, Individu i2) {
            if (i1 == null) {
                return -1;
            }
            if (i2 == null) {
                return 1;
            }
            int cmp = i1.nom.compareTo(i2.nom);
            if (cmp == 0) {
                cmp = i1.prenom.compareTo(i2.prenom);
            }
            return cmp;
        }
    }

    private final String prenom;
    private final String nom;

    public Individu(String prenom, String nom) {
        this.prenom = prenom;
        this.nom = nom;
    }

    @Override
    public String toString() {
        return this.prenom + " " + this.nom;
    }
}
```

CLASS INTERNE NON STATIC

Une classe interne qui n'est pas déclarée avec le mot-clé static est liée au contexte d'exécution d'une instance de la classe englobante.

Comme pour les classes internes static, le nom complet de classe interne inclut celui de la classe englobante et les deux classes partagent le même espace privé.

Mais surtout, une classe interne maintient une référence implicite sur un objet de la classe englobante.

Cela signifie que :

- une instance d'une classe interne ne peut être créée que par un objet de classe englobante : c'est-à-dire dans le corps d'une méthode ou dans le corps d'un constructeur de la classe englobante.
- une instance d'une classe interne a accès directement aux attributs de l'instance dans le contexte de laquelle elle a été créée.

Une classe interne est utilisée pour créer un objet qui a couplage très fort avec un objet du type de la classe englobante. On utilise fréquemment le mécanisme de classe interne lorsque l'on veut réaliser une interface graphique en Java avec l'API Swing.

ILLUSTRATION

L'exemple ci-dessus est un programme complet qui crée une boîte de dialogue contenant deux boutons qui permettent respectivement d'incrémenter et de décrémenter un nombre qui est affiché.

La classe JButton qui représente un bouton attend comme paramètre de construction une instance implémentant l'interface Action. Cette instance définit le libellé du bouton et l'action à réaliser lorsque l'utilisateur clique sur le bouton.

Ces classes sont des classes internes. Dans leur méthode actionPerformed, elles appellent soit la méthode incrementer soit la méthode decrementer. Ces deux méthodes sont définies par la classe englobante BoiteDeDialogue.

Donc les instances de ces classes d'action appellent ces méthodes sur l'instance de l'objet englobant qui les a créées. Ainsi, les classes internes possèdent une référence sur l'objet BoiteDeDialogue qui les a créées.

```

public class BoiteDeDialogue extends JDialog {

    private class IncrementerAction extends AbstractAction {
        public IncrementerAction() {
            super("Incrémente");
        }

        @Override
        public void actionPerformed(ActionEvent e) {
            incrementer();
        }
    }

    private class DecrementerAction extends AbstractAction {
        public DecrementerAction() {
            super("Décrémente");
        }

        @Override
        public void actionPerformed(ActionEvent e) {
            decrementer();
        }
    }

    private JLabel label;
    private int valeur;

    @Override
    protected void dialogInit() {
        super.dialogInit();
        this.setLayout(new FlowLayout());
        this.label = new JLabel(Integer.toString(this.valeur));
        this.add(this.label);
        this.add(new JButton(new IncrementerAction()));
        this.add(new JButton(new DecrementerAction()));
        this.pack();
    }

    private void incrementer() {
        label.setText(Integer.toString(++this.valeur));
    }

    private void decrementer() {
        label.setText(Integer.toString(--this.valeur));
    }

    public static void main(String[] args) {
        BoiteDeDialogue boiteDeDialogue = new BoiteDeDialogue();
        boiteDeDialogue.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
        boiteDeDialogue.setVisible(true);
    }
}

```

CLASSES ANONYMES

Une classe anonyme est une classe qui n'a pas de nom.

- Elle est déclarée au moment de l'instanciation d'un objet.
- Comme une classe anonyme n'a pas de nom, il n'est pas possible de déclarer une variable qui serait un type de cette classe.
- Une classe anonyme est donc utilisée pour créer à la volée une classe qui spécialise une autre classe ou qui implémente une interface.
- Pour déclarer une classe anonyme, on déclare le bloc de la classe au moment de l'instanciation avec new.

On retrouve beaucoup ce système de classes anonymes avec les évènements dans Swing !!

```
JButton Calculer = new JButton("Calculer");
Calculer.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        calculer();
    }
});
```

Dans ce code, lorsqu'on ajoute une action d'écoute sur un bouton, nous devons implémenter l'interface `ActionListener` et implémenter la méthode `actionPerformed(ActionEvent e)`.

Pour cela, nous créons une instance de l'interface et réécrivons la méthode pour réaliser l'action associée à notre bouton.



EXPRESSIONS LAMBDA L'ÉCRITURE SIMPLIFIÉE

LES EXPRESSIONS LAMBDAS

Définition

Les expressions lambda sont aussi nommées **closures ou fonctions anonymes** :

- leur but principal est de permettre de passer en paramètre un ensemble de traitements.

Principe

La syntaxe d'une expression lambda est composée de trois parties :

- un ensemble de paramètres, d'aucun à plusieurs
- l'opérateur `->`
- le corps de la fonction

Elle peut prendre deux formes principales :

- `(paramètres) -> expression;`
- `(paramètres) -> { traitements; }`

QUELQUES EXEMPLES

Exemple	Description
x -> x * 2	Accepter un nombre et renvoyer son double
(x, y) -> x + y	Accepter deux nombres et renvoyer leur somme
(int x, int y) -> x + y	Accepter deux entiers et renvoyer leur somme
(String s) -> System.out.print(s)	Accepter une chaîne de caractères et l'afficher sur la sortie standard sans rien renvoyer
c -> { int s = c.size(); c.clear(); return s; }	Renvoyer la taille d'une collection après avoir effacé tous ses éléments.
n -> n % 2 != 0;	Renvoyer un booléen qui précise si la valeur numérique est impaire
(char c) -> c == 'z';	Renvoyer un booléen qui précise si le caractère est 'z'
() -> { System.out.println("Hello World"); };	Afficher "Hello World" sur la sortie standard
(val1, val2) -> { return val1 >= val2; } (val1, val2) -> val1 >= val2;	Renvoyer un booléen qui précise si la première valeur est supérieure ou égale à la seconde
() -> { for (int i = 0; i < 10; i++) traiter(); }	Exécuter la méthode traiter() dix fois



LES THREADS PROGRAMMATION CONCURRENTE

DÉFINITION

Tous les systèmes d'exploitation modernes permettent l'exécution de tâches simultanées.

- Vous pouvez lire vos courriers électroniques tout en écoutant de la musique ou en lisant des nouvelles sur une page Web.

Nous pouvons dire qu'il s'agit d'une concurrence au niveau du processus.

- Mais dans un processus, nous pouvons également avoir plusieurs tâches simultanées.
- Les tâches simultanées exécutées dans un processus sont appelées threads.

Un thread :

- Unité d'exécution faisant partie d'un programme.
- Possède une priorité et un nom
 - par défaut "Thread-" suivi d'un numéro séquentiel incrémenté automatiquement.
- Autonome.
- Parallèle à d'autres threads.
- En Java, la JVM fonctionne aussi avec des threads
 - Un threads pour le ramassage-miettes
 - Un threads pour l'exécution de l'application...
- Consomme des ressources en mémoire et du temps processeur.
- Améliore les performances.

REPRÉSENTATION AVEC UML ET LE DIAGRAMME DE SÉQUENCE

Le 1er diagramme est sans multithreading. On constate que l'action 1.2 est en attente de la fin de l'action 1.1 et ainsi de suite...

Le 2ème diagramme est avec multithreading. Cette fois-ci, les deux actions se déroulent parallèlement et ne sont plus en attente de la fin d'une action.

Figure 17-3
Diagramme de séquence de la proie et du prédateur se désaltérant sans multithreading.

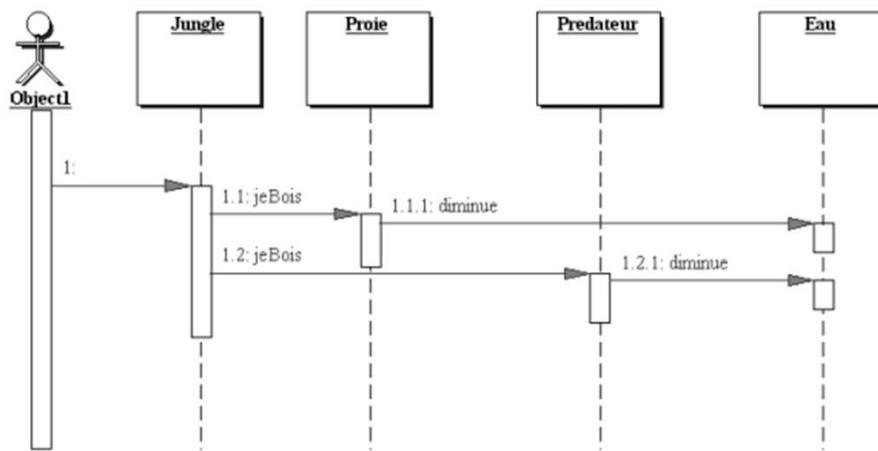
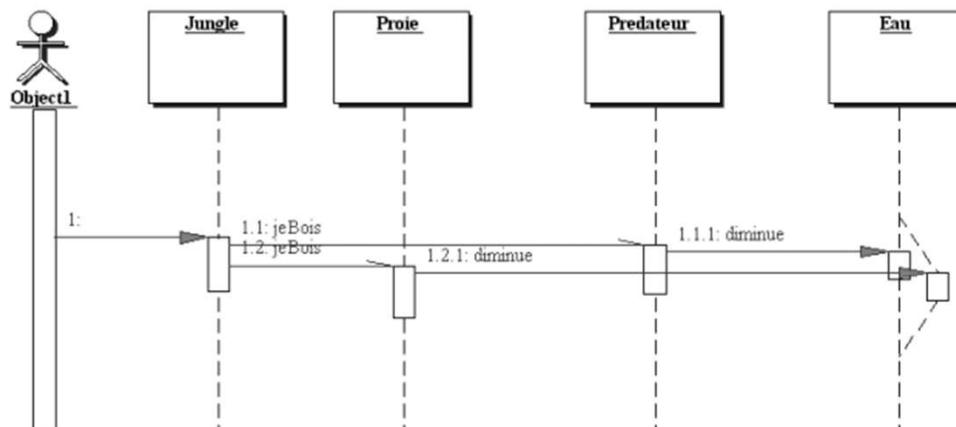


Figure 17-4
Diagramme de séquence de la proie et du prédateur se désaltérant avec multithreading.



CATÉGORIE DE THREADS

Thread utilisateur

Les Threads utilisateur sont ceux que nous mettons en place au travers des classes et interfaces mises à disposition par JAVA.

Démon (daemon threads)

Ce sont des threads qui s'exécutent indéfiniment. C'est l'arrêt de la JVM qui provoque leur fin.

- Le threads du ramasse-miettes est un bon exemple.

Pour préciser un démon, il faut invoquer sa méthode `setDaemon()` en lui passant la valeur `true` comme paramètre.

Cette méthode doit être invoquée avant que le thread ne soit démarré : une fois le thread démarré, son invocation lève une exception de type `illegalThreadStateException`.

DÉFINITION

L'interface Runnable

Cette interface doit être implémentée par toutes classes qui contiendra des traitements à exécuter dans un thread.

- Ces classes devront alors redéfinir une seule méthode run() pour contenir le code à exécuter dans le thread.

La classe Thread

La classe Thread est définie dans le package java.lang. Elle implémente l'interface Runnable.

Elle possède plusieurs constructeurs : un par défaut et plusieurs avec un ou plusieurs paramètres :

- le nom du thread (par défaut Thread-xx)
- L'objet qui implémente l'interface Runnable
- L'objet contenant les traitements du thread
- le groupe auquel sera rattaché le thread

CRÉATION, EXÉCUTION ET DÉFINITION

Comme pour tous les éléments du langage Java, les threads sont des objets.

Nous avons deux façons de créer un thread en Java :

- Hériter de la classe Thread et redéfinir la méthode `run()`.
- Construire une classe qui implémente l'interface `Runnable` et la méthode `run()`
 - puis créer un objet de la classe Thread en transmettant l'objet Runnable en tant que paramètre, c'est l'approche recommandée qui vous offre plus de flexibilité.

La classe Thread enregistre certains attributs d'informations pouvant nous aider à identifier un thread, à connaître son statut ou à contrôler sa priorité.

Ces attributs sont :

- `ID` : stocke un identifiant unique pour chaque thread.
- `Name` : stocke le nom du thread.
- `Priority` : stocke la priorité des objets Thread, priorité comprise entre 1 et 10, 1 la plus basse et 10 la plus élevée.
 - Il n'est pas recommandé de changer la priorité des threads.
- `Status` : stocke le statut d'un thread. En Java, six états définis dans `Thread.State` :
 - `NEW` : a été créé et il n'a pas encore commencé
 - `RUNNABLE` : est en cours d'exécution dans la machine virtuelle Java
 - `BLOCKED` : est bloqué et attend un moniteur
 - `WAITING` : attend un autre thread
 - `TIMED_WAITING` : attend un autre thread avec un délai d'attente
 - `TERMINATED` : a terminé son exécution

EXEMPLE

```
class Multithreading implements Runnable {
    public void run() {
        try {
            for (int i = 0; i < 5; i++)
                System.out.println(Thread.currentThread().getName() + " a le
control");
        } catch (Exception e) {
            System.out.println("Exception " + e.getMessage());
        }
    }
}
```

```
// classe principale
public class Multithread {
    public static void main(String[] args) {
        Thread objet = new Thread(new Multithreading());
        objet.start();
        Thread objet2 = new Thread(new Multithreading());
        objet2.start();
        for (int i = 0; i < 5; i++) {
            // Le controle passe au thread enfant apres 5 iteration
            Thread.yield();

            // Apres execution du thread enfant,
            // le thread main prend le relais
            System.out.println(Thread.currentThread().getName() + " a le
control");
        }
    }
}
```



```
Thread-0 a le controle
Thread-1 a le controle
Thread-1 a le controle
Thread-1 a le controle
Thread-1 a le controle
main a le controle
```

OPTIONS

Les groupes

Un groupe de threads permet de regrouper les threads selon différents critères et de les manipuler en même temps.

Les threads d'un même groupe peuvent se manipuler, se contrôler entre eux et peuvent contenir d'autres groupes de threads. (notion de parent)

Le groupe par défaut est le main.

Informations, état d'un thread

On peut obtenir des informations sur les thread (son identifiant, son contexte, sa priorité, daemon ?...) ou sur son état (en vie, interrompu...)

CYCLE DE VIE

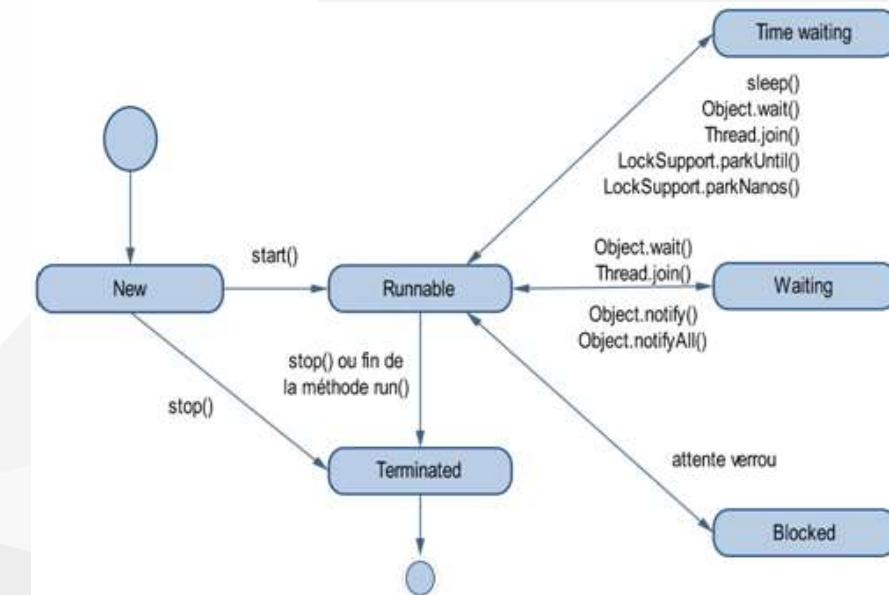
Un thread, encapsulé dans une instance de type classe Thread, suit un cycle de vie qui peut prendre différents états.

Le statut du thread est encapsulé dans l'énumération `Thread.State`

Une fois lancé par la méthode `start()`, plusieurs actions peuvent suspendre l'exécution d'un thread :

- invocation de la méthode `sleep()`, `join()` ou `suspend()`
- attente de la fin d'une opération de type I/O

L'invocation de certaines méthodes de la classe Thread peut lever une exception de type `IllegalThreadStateException` si cette invocation n'est pas permise à cause de l'état courant du thread.



SYNCHRONISATION ENTRE THREADS

Principe

Plusieurs threads - une seule ressource

- Possible problèmes de concurrence
 - Écriture dans un fichier par exemple.

Solution : la synchronisation

- 2 méthodes
 - Les moniteurs : mécanisme implicite, intégrée à la JVM
 - Les locks : mécanisme explicite via les classes de l'API du JDK
- En Java, concept appelé moniteurs
 - Chaque objet est associé à moniteur qui peut être verrouillé ou déverrouillé par un seul thread à la fois.

Notion de verrous

Pour verrouiller un thread, il faut utiliser le mot clé **synchronized**

- Soit on synchronise un ensemble d'instructions sur un objet
- Soit on synchronise l'exécution d'une méthode (statique et non statique).

LES MONITEURS

Utilisation

```
public static synchronized void maMethode() {
    // synchroniser toute une méthode
}

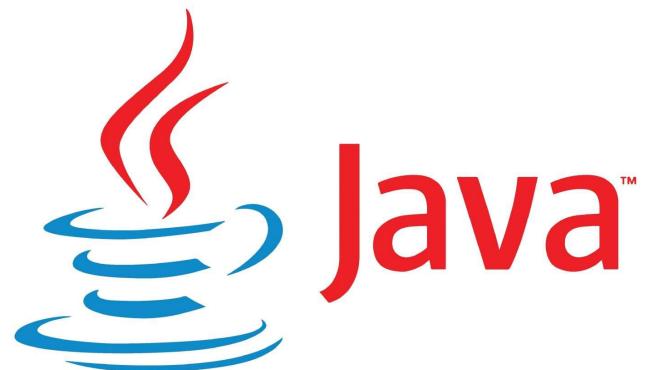
public static synchronized void maMethode() {
    // // synchroniser toute une méthode static
}

public void maMethode() {
    synchronized(this) {
        // synchroniser un bloc de code
        // en précisant l'instance
    }
}

public void maMethode() {
    synchronized(MaClasse.class) {
        // Synchroniser la classe
        // protéger les accès aux ressource static
    }
}
```

Verrou posé sur l'instance de classe Singleton

```
Singleton result = instance;
if (result != null) {
    return result;
}
synchronized(Singleton.class) {
    if (instance == null) {
        instance = new Singleton(value);
    }
    return instance;
}
```



JDBC et La porte d'accès aux bases de données.

JDBC

LA PORTE D'ACCÈS AUX BASES DE
DONNÉES.

ASTUCE JAVA-BDD

Dans le cadre de nos bases, une bonne pratique est d'utiliser les *Wrapper* au lieu des types primitif.

1. Primitif long number;
2. *Wrapper* Long number;

Pourquoi ? Un Wrapper est un objet puisqu'il est défini par une classe. De ce fait, l'objet étant capable d'accepter la valeur null, vous pouvez facilement récupérer toutes les valeurs null provenant de la BDD.

Enfin, un Wrapper vous permet d'avoir accès à des méthodes pour manipuler le type, ce qui est confortable dans nos différents développements.



EN L'OCCURRENCE, LES CHAMPS DE NOTRE TABLE SQL SE VOIENT TOUTES APPLIQUER UNE CONTRAINTE NULL, IL Y A DONC RISQUE DE VALEURS NULLES ET GÉNÉRATION D'EXCEPTIONS



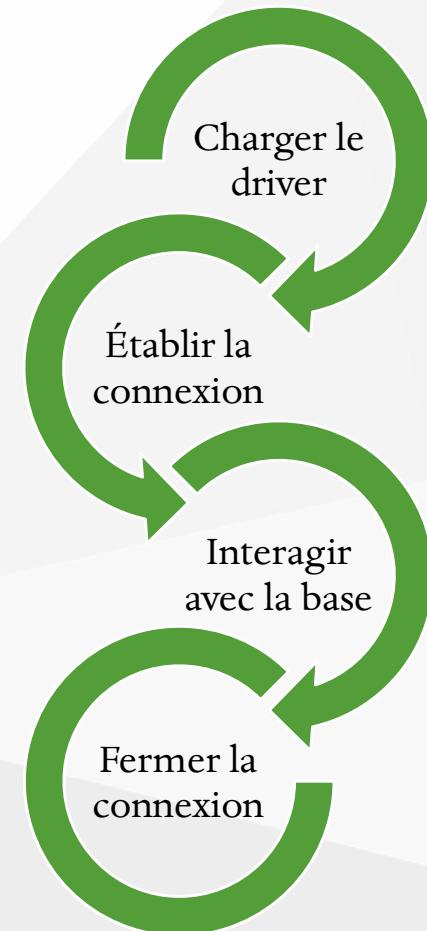
CEPENDANT, C'EST UNE BONNE PRATIQUE DE TOUJOURS UTILISER LES OBJETS WRAPPER DANS LES BEANS DONT LES PROPRIÉTÉS CORRESPONDENT À DES CHAMPS D'UNE BASE DE DONNÉES, AFIN D'ÉVITER CE GENRE D'ERREURS !

ETAPE POUR CRÉER UNE CONNEXION

Pour créer une connexion à une base de données avec le langage JAVA, il y a plusieurs étapes successives à réaliser.

1. Charger le driver de la BDD
2. Établir la connexion à la BDD
3. Faire les différentes interactions (CRUD)
4. Fermer la connexion.

Comme nous faisons appel à une ou des ressource(s), il faut également penser à capturer toutes les exceptions provenant de la BDD.



INSTALLATION

Télécharger l'extension

- Pour pouvoir connecter votre application Java à une base de données MySQL, il vous faudra télécharger l'extension (driver) "MySQL Connector.jar"

<https://dev.mysql.com/downloads/connector/j/>

Chaque SGBD a son driver donc par conséquent, il faut penser à correctement choisir le bon driver.

- Pour MySQL, vous pouvez trouver le nom du driver à utiliser sur <https://dev.mysql.com/doc/connector-j/en/connector-j-reference-driver-name.html>

Installer de l'extension

Dans notre JDK, nous avons l'ensemble des classes pour interagir avec la base de données.

L'extension "MySQL Connector" ne fait pas partie de la JDK.

Il faut donc ajouter cette librairie à notre projet.

Pour cela, il existe 2 manières de faire :

1. Soit l'ajouter à la main.
2. Soit passer par un gestionnaire de projet (ex: Maven).

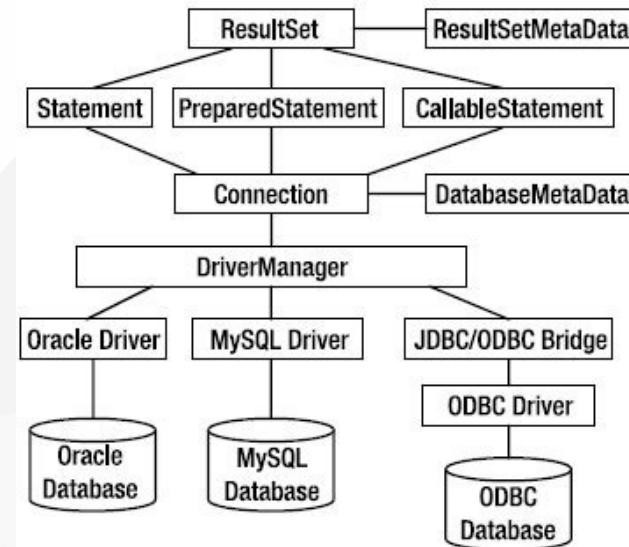
JAVA DATABASE CONNECTIVITY (JDK)

Il s'agit de classes Java contenu dans le package [Java.sql](#) permettant de se connecter et d'interagir avec des bases de données.

- La base de données est prête, les tables sont créées, remplies et nous avons le [driver](#) nécessaire à notre base de données.

On crée donc une instance de l'objet [Driver](#) permettant d'ouvrir une [Connection](#), permettant d'obtenir un [Statement](#) qui nous retournera un [ResultSet](#)

Enfin, pour terminer proprement une connexion, il faut fermer les différents espaces ouverts. Chaque objet possède une méthode [close\(\)](#)





CRÉATION DE LA CONNEXION

Se connecter à la BDD

1ÈRE CONNEXION

Pour effectuer une connexion à la base de données, nous avons besoin d'une **url**, d'un **login** et d'un **mot de passe**.

- **L'url :** protocol//hosts//database]?properties]
- **Login :** compte de connexion
- **Mot de passe :** mot de passe de connexion

La plupart des méthodes lèvent l'exception **java.sql.SQLException** qui étend la classe **Exception**.

```
private void firstConnection() { 1 usage new *  
    String BDD = "airdejava";  
    String url = "jdbc:mysql://localhost:3306/airdejava";  
    String user = "root";  
    String password = "root";  
  
    try {  
        // chargement du driver  
        Class.forName(className: "com.mysql.cj.jdbc.Driver");  
  
        // Cr ation de la connection  
        Connection connection = DriverManager.getConnection(url, user, password);  
  
        // petit message indiquant que tout s'est bien pass   
        System.out.println("Connected to database");  
  
        // fermeture de la connection  
        connection.close();  
  
    } catch (ClassNotFoundException e) {  
        // exception pour le driver  
        throw new RuntimeException(e);  
    } catch (SQLException e) {  
        // exception pour SQL  
        throw new RuntimeException(e);  
    }  
}
```

FICHIER DE CONFIGURATION

Pour l'exemple, l'utilisation de variables placées dans le code n'est pas conseillée.

Il est préférable d'utiliser des fichiers de configurations qu'on place dans notre architecture et qu'on va ensuite lire pour en déduire les données de configuration.

Puisque ce fichier contient des informations confidentielles, comme le mot de passe de connexion à la base de données, il est hors de question de le placer dans un répertoire public de l'application.

On va alors créer un répertoire dans notre application en dehors du dossier src contenant notre code.

Dans une vraie application, les fichiers de configuration ne sont pas stockés au sein de l'application, mais en dehors. Il devient ainsi très facile d'y modifier une valeur sans avoir à naviguer dans le contenu de l'application.

```
1 #conf DB  
2 jdbc.driver.class=com.mysql.cj.jdbc.Driver  
3 jdbc.url=jdbc:mysql://localhost:3306/airdejava  
4 jdbc.login=root  
5 jdbc.password=root
```

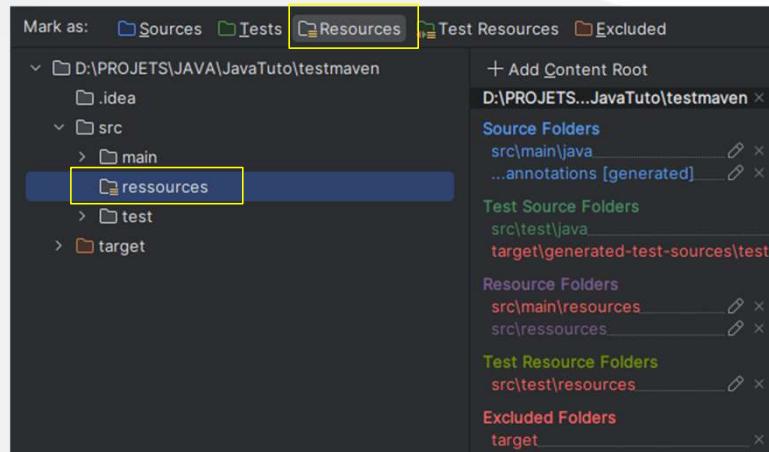
Exemple de fichier conf.properties

Il fonctionne selon le principe clé=valeur

UTILISATION DU FICHIER DE CONFIGURATION

Votre fichier de configuration va être placé dans un répertoire qu'on va déclarer dans notre application comme ressources.

Pour cela, avec IntelliJ, il suffit d'aller dans [open module settings](#) et de déclarer votre dossier comme ressources (un petit symbole apparaît sur le dossier).



```

private void propertiesConnection() { // usage new *

    // chemin vers le fichier de propriétés
    final String PATHCONF = "conf.properties";
    // attribut properties
    Properties prop = new Properties();

    // chargement du fichier de propriétés
    try (InputStream is = getClass().getClassLoader().getResourceAsStream(PATHCONF)) {

        prop.load(is);

    } catch (IOException e) {
        throw new RuntimeException(e);
    }

    // traitement du fichier de propriétés et de la connexion à la BDD
    try {

        // chargement du driver
        Class.forName(prop.getProperty("jdbc.driver.class"));

        // récupération des paramètres pour la connexion
        String url = prop.getProperty("jdbc.url");
        String user = prop.getProperty("jdbc.login");
        String password = prop.getProperty("jdbc.password");

        Connection connection = DriverManager.getConnection(url, user, password);

        // petit message indiquant que tout s'est bien passé
        System.out.println("Connected to database from properties");

        // fermeture de la connection
        connection.close();

    } catch (ClassNotFoundException e) {
        throw new RuntimeException(e);
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}

```



INTERACTION

Utiliser la BDD

MANIPULATION DE COMMANDE SQL

- La requête est créée avec l'interface **Statement** qui possède les méthodes nécessaires pour réaliser les requêtes sur la base associé à la connexion.
- 3 types (interfaces) d'objets statement :
 - **Statement** : requêtes simples (SDSL Statique)
 - **PreparedStatement** : requêtes paramétrables (ou dynamiques) précompilées (requêtes avec paramètres d'entrée/sortie)
 - **CallableStatement** : encapsule procédures SQL stockées dans le SGBD
- Création d'un statement :
 - **Statement stm = connexion.createStatement();**
 - **PreparedStatement stm = connexion.prepareStatement(str);**
 - **CallableStatement stm = connexion.prepareCall(str);**

```
// un simple select
private void selectFromPersonne(Connection con) { 1 usage new *

    String selectSQL = "select * from personne";

    try {
        // établissement d'un Statement
        Statement stmt = con.createStatement();

        // utilisation du statement pour envoyer la requête et récupération du résultat
        ResultSet resultSet = stmt.executeQuery(selectSQL);

        while(resultSet.next()) {
            System.out.println(resultSet.getString(columnLabel: "per_nom"));
            System.out.println(resultSet.getString(columnLabel: "per_prenom"));
            System.out.println(resultSet.getString(columnLabel: "per_adr"));
            System.out.println(resultSet.getString(columnLabel: "per_cp"));
            System.out.println(resultSet.getString(columnLabel: "per_ville"));
            System.out.println(resultSet.getString(columnLabel: "per_tel"));
            System.out.println(resultSet.getString(columnLabel: "per_email"));
            System.out.println("-----");
        }
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
```

```
private List<Personne> findAll(Connection con) { //usage new *

    List<Personne> personnes = new ArrayList<>();
    String selectSQL = "select * from personne";

    try {
        // établissement d'un Statement
        Statement stmt = con.createStatement();

        // utilisation du statement pour envoyer la requête et récupération du résultat
        ResultSet resultSet = stmt.executeQuery(selectSQL);

        while(resultSet.next()) {

            Personne personne = new Personne();

            personne.setId(resultSet.getInt( columnLabel: "per_id"));
            personne.setCiviliteId(resultSet.getInt( columnLabel: "civ_id"));
            personne.setNom(resultSet.getString( columnLabel: "per_nom"));
            personne.setPrenom(resultSet.getString( columnLabel: "per_prenom"));
            personne.setAdresse(resultSet.getString( columnLabel: "per_adr"));
            personne.setEmail(resultSet.getString( columnLabel: "per_cp"));
            personne.setVille(resultSet.getString( columnLabel: "per_ville"));
            personne.setTelephone(resultSet.getInt( columnLabel: "per_tel"));
            personne.setFax(resultSet.getInt( columnLabel: "per_fax"));
            personne.setEmail(resultSet.getString( columnLabel: "per_email"));
            personne.setDateNaissance(resultSet.getDate( columnLabel: "per_datenaissance"));

            personnes.add(personne);
        }
    }

    return personnes;
} catch (SQLException e) {
    throw new RuntimeException(e);
}
}
```

UTILISATION DE L'OBJET

Comme désormais, on maîtrise l'objet, la bonne manière de faire est de créer un objet métier.

UTILISATION DE STRINGBUIL DER

Lorsqu'on vient à créer nos requêtes, il est préférable d'utiliser la classe StringBuilder à la place de la classe String.

Pourquoi ? Nos requêtes vont être une association de chaîne de caractères par concaténation le plus souvent.

Pour rappel, String est immuable c'est-à-dire que toutes modifications apportées sur String produisent un autre objet String.

StringBuilder ou StringBuffer sont eux mutable.

```
StringBuilder sqlFindAclient = new StringBuilder();
sqlFindAclient.append("select ");
sqlFindAclient.append("c.idsociety,");
sqlFindAclient.append("socialreason,");
sqlFindAclient.append("domainsoc,");
sqlFindAclient.append("numstreet,");
sqlFindAclient.append("namestreet,");
sqlFindAclient.append("postcode,");
sqlFindAclient.append("city,");
sqlFindAclient.append("phone,");
sqlFindAclient.append("email,");
sqlFindAclient.append("commentsoc,");
sqlFindAclient.append("revenue,");
sqlFindAclient.append("workforce ");
sqlFindAclient.append("from \"client\" c ");
sqlFindAclient.append("join \"society\" s ");
sqlFindAclient.append("on c.idsociety = s.idsociety ");
sqlFindAclient.append("where c.idsociety = ?");
```

L'INTERFACE RESULTSET

Ensuite ce **Statement** offre des méthodes permettant de passer des ordre SQL à exécuter.

3 méthodes d'exécutions d'un Statement :

- **ResultSet executeQuery(String SQL)** : pour les requêtes qui retournent un résultat (SELECT) accessible au travers d'un objet ResultSet
- **Int executeUpdate(String SQL)** : pour les requêtes qui ne retournent pas de résultats (INSERT, DELETE, UPDATE...)
- **boolean execute(String SQL)** : quand on ne sait pas si la requête retourne ou non un résultat, procédures stockées.

```
// utilisation du statement pour envoyer la requête et récupération du
ResultSet resultSet = stmt.executeQuery(selectSQL);

while(resultSet.next()) {

    Personne personne = new Personne();

    personne.setId(resultSet.getInt( columnLabel: "per_id"));
    personne.setCiviliteId(resultSet.getInt( columnLabel: "civ_id"));
    personne.setNom(resultSet.getString( columnLabel: "per_nom"));
    personne.setPrenom(resultSet.getString( columnLabel: "per_prenom"));
    personne.setAdresse(resultSet.getString( columnLabel: "per_adr"));
    personne.setEmail(resultSet.getString( columnLabel: "per_cp"));
    personne.setVille(resultSet.getString( columnLabel: "per_ville"));
    personne.setTelephone(resultSet.getInt( columnLabel: "per_tel"));
    personne.setFax(resultSet.getInt( columnLabel: "per_fax"));
    personne.setEmail(resultSet.getString( columnLabel: "per_email"));
    personne.setDateNaissance(resultSet.getDate( columnLabel: "per_datenaiss"));

    personnes.add(personne);
}
```

L'INTERFACE RESULTSET

`executeQuery()` renvoie une instance de `ResultSet` qui permet d'accéder aux champs des n-uplets sélectionnés.

Les rangées du `ResultSet` se parcourront itérativement ligne par ligne.

- `Boolean next()` permet d'avancer à la ligne suivante, retourne false si pas de ligne suivante placé avant la première ligne à la création du `ResultSet`
- Les colonnes sont référencées par leur numéro ou par leur nom
- L'accès aux valeurs des colonnes se fait par des méthodes `getXXX(String nomCol)` ou `getXXX(int numCol)` où XXX dépend du type de la colonne de la table SQL

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP	CLOB	BLOB	ARRAY	REF	STRUCT	JAVA OBJECT	
getByte	X	x	x	x	x	x	x	x	x	x	x	x	x													
getShort	x	X	x	x	x	x	x	x	x	x	x	x														
getInt	x	x	X	x	x	x	x	x	x	x	x	x														
getLong	x	x	x	X	x	x	x	x	x	x	x	x														
getFloat	x	x	x	x	X	x	x	x	x	x	x	x														
getDouble	x	x	x	x	x	X	x	x	x	x	x	x														
getBigDecimal	x	x	x	x	x	x	X	x	x	x	x	x														
getBoolean	x	x	x	x	x	x	x	X	x	x	x	x														
getString	x	x	x	x	x	x	x	x	X	x	x	x	x	x	x	x	x	x	x	x	x	x	x			
getBytes														X	X	x										
getDate											x	x	x			x	x									
getTime											x	x	x					x	x	x						
getTimestamp											x	x	x					x	x	x						
getAsciiStream											x	x	X	x	x											
getUnicodeStream											x	x	X	x	x											
getBinaryStream														x	x	X										
getBlob																				x						
getClob																					x					
getArray																					x					
getRef																						x				
getCharacterStream																	x	x	X	x	x	x	x	x	x	
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	X	X	

Table 5.1: Use of `ResultSet.getXXX` Methods to Retrieve JDBC Types

LES REQUÊTES PRÉPARÉES

Si on doit répéter plusieurs fois la même requête (par exemple `select * from login where log_user = num_id`), alors la requête préparée permet d'écrire une seule requête dans laquelle on prévoit un ou plusieurs paramètres :

- `Select * from login where log_user = ?`

Le symbole `?` désigne un paramètre dont la valeur sera fixée par la suite.

Une requête préparée est transmise au SGBR à l'aide de la méthode `prepareStatement` de la classe `Connection` qui nous renvoie un objet de type `PreparedStatement` contenant toutes les informations nécessaires à l'exploitation de cette requête préparée par le SGBDR.



```

private Personne findById(Connection con, int pId) { 1 usage new*
    Personne personne = new Personne();
    String selectById = "select * from personne where per_id=?";
    try {
        // utilisation d'un prepareStatement
        PreparedStatement pstmt = con.prepareStatement(selectById);
        pstmt.setInt( parameterIndex: 1, pId);

        ResultSet resultSet = pstmt.executeQuery();

        while(resultSet.next()) {

            personne.setId(resultSet.getInt( columnLabel: "per_id"));
            personne.setCiviliteId(resultSet.getInt( columnLabel: "civ_id"));
            personne.setNom(resultSet.getString( columnLabel: "per_nom"));
            personne.setPrenom(resultSet.getString( columnLabel: "per_prenom"));
            personne.setAdresse(resultSet.getString( columnLabel: "per_adr"));
            personne.setEmail(resultSet.getString( columnLabel: "per_cp"));
            personne.setVille(resultSet.getString( columnLabel: "per_ville"));
            personne.setTelephone(resultSet.getInt( columnLabel: "per_tel"));
            personne.setFax(resultSet.getInt( columnLabel: "per_fax"));
            personne.setEmail(resultSet.getString( columnLabel: "per_email"));
            personne.setDateNaissance(resultSet.getDate( columnLabel: "per_datenaisance"));

        }
        return personne;
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}

```

LES REQUÊTES PRÉPARÉES

Ensuite, on peut transmettre au SGBDR, les valeurs voulues des paramètres, à l'aide des méthodes de la forme `setXXX(rang, valeur)` de la classe `PreparedStatement`, auxquelles on précise :

- Le rang du paramètre concerné
- La valeur prévue pour ce paramètre.

Enfin, on exécute la requête par l'une de des méthodes vu ultérieurement.



```
System.out.println("Affichage d'une personne");
System.out.println(findById(connection, pld: 41));
```

```
Affichage d'une personne
41 / 2 - BRETANI MARIE - 56 RUE DE LA LORRAINE null MARSEILLE
```

RÉCUPÉRER UN ID AUTO- GÉNÉRÉ

Avec le `prepareStatement`, on peut récupérer une information sur la requête réalisée.

Par exemple dans le cas d'un insert, si on souhaite récupérer l'ID auto-généré, il faut indiquer lors que la requête préparée, à notre `prepareStatement` le besoin de retourner l'information

<https://docs.oracle.com/javase/8/docs/api/java/sql/Connection.html#prepareStatement-java.lang.String-int->

```
private int insertPersonne(Connection con, Personne personne) { 1usage new

    SimpleDateFormat simpleDateFormat = new SimpleDateFormat( pattern: "yyyy-MM-dd"
    int newId = 0;

    StringBuilder insertSQL = new StringBuilder();
    insertSQL.append("insert into Personne (civ_id,per_nom,per_prenom,per_adr,")
    insertSQL.append("per_cp, per_ville, per_denaissance) ");
    insertSQL.append("values (?, ?, ?, ?, ?, ?)");

    try {
        // preparation de la requete
        PreparedStatement pstmtt = con.prepareStatement(insertSQL.toString(),
            PreparedStatement.RETURN_GENERATED_KEYS);

        // mise en place des parametres
        pstmtt.setInt( parameterIndex: 1, personne.getCiviliteId());
        pstmtt.setString( parameterIndex: 2, personne.getNom());
        pstmtt.setString( parameterIndex: 3, personne.getPrenom());
        pstmtt.setString( parameterIndex: 4, personne.getAdresse());
        pstmtt.setString( parameterIndex: 5, personne.getCodePostal());
        pstmtt.setString( parameterIndex: 6, personne.getVille());

        // conversion en SQL DATE
        String formattedDate = simpleDateFormat.format(personne.getDateNaissance());
        pstmtt.setDate( parameterIndex: 7, Date.valueOf(formattedDate));

        // execution
        pstmtt.executeUpdate();

        // recuperation de l'ID
        ResultSet rs = pstmtt.getGeneratedKeys();
        if (rs.next()) {
            newId = rs.getInt( columnIndex: 1);
        }

        return newId;
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
```

LES TRANSACTIONS UTILISATION EN JAVA

TRANSACTION

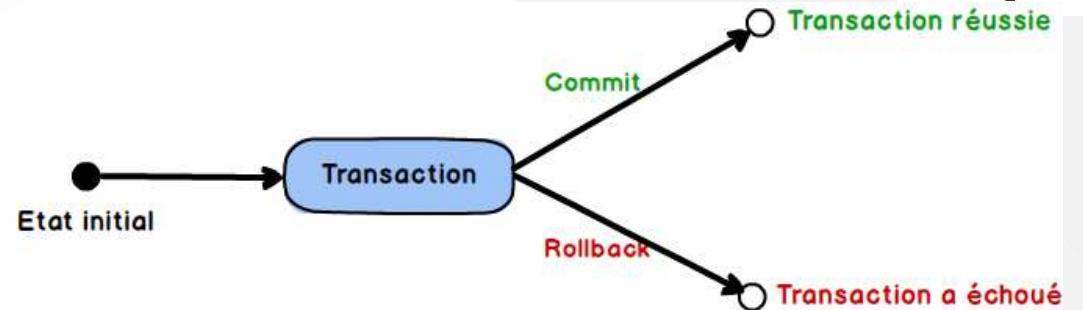
- Par défaut, la connexion est en "auto-commit". Un commit est automatiquement lancé après chaque ordre SQL qui modifie la base.
- Pour définir une transaction composée de plusieurs requêtes SQL, il faut désactiver l'auto-commit :

```
connect.setAutoCommit(false);
```

- Il faut alors explicitement valider ou annuler la transaction par :

`connect.commit()` : valide la transaction

`connect.rollback()` : annule la transaction



```

try{
    //définir la gestion des transactions manuelles
    conn.setAutoCommit(false);
    Statement stmt = conn.createStatement();

    String query = "INSERT INTO login VALUES (1, 'John', 'tx514')";
    stmt.executeUpdate(query);

    //Soumettre une instruction SQL mal formée qui brise le code
    String query = "INSERT IN login VALUES (2, 'invit', 'rx897')";
    stmt.executeUpdate(query);

    // S'il n'y a pas d'erreur.
    conn.commit();
}
catch(SQLException se){
    // S'il y a une erreur.
    conn.rollback();
}
  
```

LES TRANSACTIONS

Il faut noter que :

- Pas besoin d'effectuer START TRANSACTION lors de votre transaction puisque les requêtes sont explicitement encadrées par un START TRANSACTION par le SGBD.
- Il est possible de protéger nos transactions pour éviter "des lectures sales" avec un niveau d'isolation qui appliquent des règles d'accès très strictes.
- <https://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html>

Isolation Level	Transactions	Dirty Reads	Non-Repeatable Reads	Phantom Reads
TRANSACTION_NONE	Not supported	<i>Not applicable</i>	<i>Not applicable</i>	<i>Not applicable</i>
TRANSACTION_READ_COMMITTED	Supported	Prevented	Allowed	Allowed
TRANSACTION_READ_UNCOMMITTED	Supported	Allowed	Allowed	Allowed
TRANSACTION_REPEATABLE_READ	Supported	Prevented	Prevented	Allowed
TRANSACTION_SERIALIZABLE	Supported	Prevented	Prevented	Prevented

- Habituellement, vous n'avez pas besoin de faire quoi que ce soit concernant le niveau d'isolation de la transaction.
- Vous pouvez simplement utiliser le niveau par défaut pour votre SGBD. Par exemple, pour Java, il est TRANSACTION_READ_COMMITTED.
- JDBC vous permet de savoir à quel niveau d'isolation de transaction votre SGBD est réglé (getTransactionIsolation) et vous permet également de le mettre à un autre niveau (setTransactionIsolation).

LES TRANSACTIONS : JALON

- `setSavePoint()` : Cette méthode vous permet de positionner un jalon dans votre transaction. Ce jalon peut alors être utilisé avec la méthode `rollback()`
- `releaseSavePoint()` : Cette méthode permet de supprimer un jalon de votre transaction.

```
1. try{
2.     //définir la gestion des transactions manuelles
3.     conn.setAutoCommit(false);
4.     Statement stmt = conn.createStatement();
5.
6.     //définir un point de sauvegarde
7.     Savepoint savepoint = conn.setSavepoint("Savepoint1");
8.
9.     String query = "INSERT INTO Emp VALUES (101, 'Alex')";
10.    stmt.executeUpdate(query);
11.
12.    //Soumettre une instruction SQL mal formée qui brise le code
13.    String query = "INSERT IN Emp VAL (102, 'Bob')";
14.    stmt.executeUpdate(query);
15.
16.    // S'il n'y a pas d'erreur.
17.    conn.commit();
18. }
19. catch(SQLException se){
20.     // S'il y a une erreur.
21.     conn.rollback(savepoint);
22. }
```

DESIGN PATTERN SINGLETON

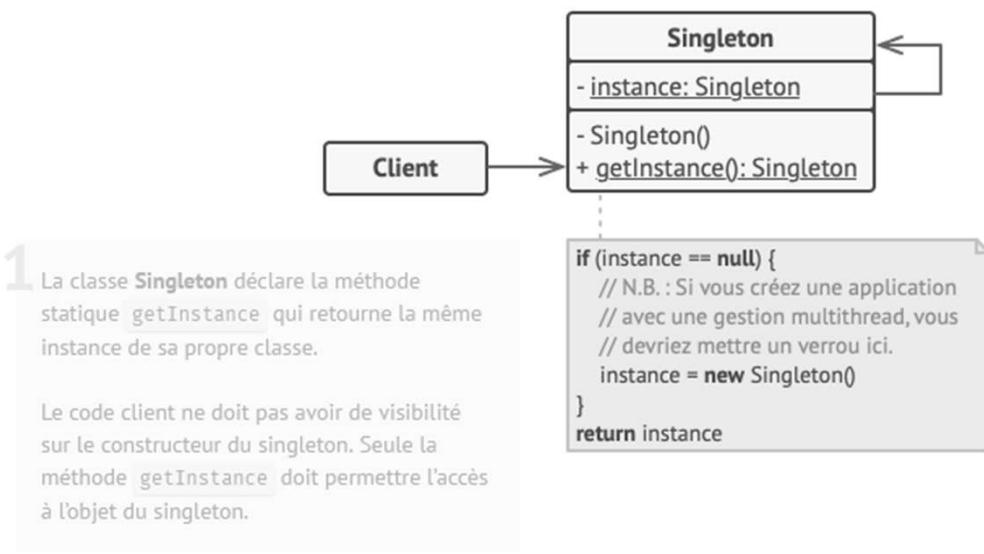
PATTERN SINGLETON

Singleton est un patron de conception de création qui garantit que **l'instance d'une classe n'existe qu'en un seul exemplaire**, tout en fournissant un point d'accès global à cette instance.

Le singleton règle deux problèmes à la fois, mais ne respecte pas le principe de responsabilité unique.

- Il garantit l'unicité d'une instance pour une classe.
- Il fournit un point d'accès global à cette instance.

Structure



1 La classe **Singleton** déclare la méthode statique `getInstance` qui retourne la même instance de sa propre classe.

Le code client ne doit pas avoir de visibilité sur le constructeur du singleton. Seule la méthode `getInstance` doit permettre l'accès à l'objet du singleton.

SINGLETON

Le constructeur de notre singleton

```
public class Singleton { 5 usages ▲ neojero *

//public static final Logger LOGGER = Logger.getLogger(Singleton.class.getName());

private static final Properties props = new Properties(); 5 usages
private static Connection connection; 2 usages
final String PATHCONF = "conf.properties"; 1 usage

private Singleton() { 1 usage ▲ neojero
try (InputStream is = getClass().getClassLoader().getResourceAsStream(PATHCONF)) {

    props.load(is);

} catch (IOException e) {
    throw new RuntimeException(e);
}

// traitement du fichier de propriétés et de la connexion à la BDD
try {

    // chargement du driver
    Class.forName( props.getProperty("jdbc.driver.class"));

    // récupération des paramètres pour la connexion
    String url = props.getProperty("jdbc.url");
    String user = props.getProperty("jdbc.login");
    String password = props.getProperty("jdbc.password");

    connection = DriverManager.getConnection(url, user, password);

} catch (ClassNotFoundException e) {
    throw new RuntimeException(e);
} catch (SQLException e) {
    throw new RuntimeException(e);
}
}
```

SINGLET ON

Les méthodes de gestion de l'instance :

`getInstanceDB()` : retourne l'instance en cours sinon la crée

`closeInstanceDB()` : ferme l'instance en cours

`getConnection()` : retourne l'instance.
Cette méthode est en private.

```
public static Connection getInstanceDB(){ 2 usages  ± neojero
    if (getConnection() == null){
        new Singleton();
        //LOGGER.info("RelationWithDB infos : Connection established");
        System.out.println("RelationWithDB infos : Connection established");
    }
    else {
        //LOGGER.info("RelationWithDB infos : Connection already existing");
        System.out.println("RelationWithDB infos : Connection already existing");
    }
    return getConnection();
}

public static void closeInstanceDB() { 1 usage  ± neojero *
    try{
        getConnection().close();
        //LOGGER.info("RelationWithDB infos : Connection terminated");
        System.out.println("RelationWithDB infos : Connection terminated");
    }
    catch(SQLException sqle){
        //LOGGER.severe("RelationWithDB erreur : " + sqle.getMessage() +
        // [SQL error code : " + sqle.getSQLState() + "]");
        throw new RuntimeException(sqle);
    }
}

/**
 * @return the connection
 */
private static Connection getConnection() { 3 usages  ± neojero
    return connection;
}
```

TEST AVEC SINGLETON

Désormais , je fais appel à mon Singleton pour initialiser la connexion à la base de données.

```

/* ----- METHODE SINGLETON ----- */
private List<Personne> findAll() {...}

private Personne findById(int pId) { 1 usage à nejero *

    Personne personne = new Personne();
    StringBuilder selectById = new StringBuilder("select * from personne where per_id=?");

    try {
        Connection con = Singleton.getInstanceDB();
        // contrôle
        System.out.println(con);

        // utilisation d'un prepareStatement
        PreparedStatement pstmt = con.prepareStatement(selectById.toString());

        pstmt.setInt( parameterIndex: 1, pId);

        ResultSet resultSet = pstmt.executeQuery();

        while(resultSet.next()) {

            personne.setId(resultSet.getInt( columnLabel: "per_id"));
            personne.setCivileId(resultSet.getInt( columnLabel: "civ_id"));
            personne.setNom(resultSet.getString( columnLabel: "per_nom"));
            personne.setPrenom(resultSet.getString( columnLabel: "per_prenom"));
            personne.setAdresse(resultSet.getString( columnLabel: "per_adr"));
            personne.setEmail(resultSet.getString( columnLabel: "per_cp"));
            personne.setVille(resultSet.getString( columnLabel: "per_ville"));
            personne.setTelephone(resultSet.getInt( columnLabel: "per_tel"));
            personne.setFax(resultSet.getInt( columnLabel: "per_fax"));
            personne.setEmail(resultSet.getString( columnLabel: "per_email"));
            personne.setDateNaissance(resultSet.getDate( columnLabel: "per_datenaisance"));

        }
        return personne;
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}

```

CLASSES ET MÉTHODES GÉNÉRIQUES

GÉNÉRALISATION

QU'EST-CE QUE LA GÉNÉRICITÉ ?

En Programmation Orientée Object (POO), la généricité est un concept permettant de définir des algorithmes (types de données et méthodes) identiques qui peuvent être utilisés sur de multiples types de données.

Cela permet donc de réduire les quantités de codes à produire.

La généricité permettra d'avoir un code plus fortement typé et donc plus sûr.

- Déjà utilisé avec les collections comme ArrayList
 - `ArrayList<String> collection = new ArrayList<String>();`
- L'interface Collection, Iterable, Map
- L'opérateur diamant <>
 - `ArrayList<String> collection = new ArrayList<>();`
 - Ici, on utilise l'inférence de type qui déduit le type lors de la déclaration de la variable.
- Les conventions de dénomination de types paramétrés sont importantes pour apprendre la généricité en Java. Les types paramétrés courants sont les suivants:
 - T – Type
 - E – Element
 - K – Key
 - N – Number
 - V – Value

CLASSES GÉNÉRIQUES

Les classes

La déclaration d'une classe générique ressemble à une déclaration de classe non générique, sauf que le nom de la classe est suivi du type paramétré.

Pour indiquer qu'un type est générique, il faut ajouter la liste des types génériques supportés entre caractères < et > à la suite du nom de la classe et avant l'accolade ouvrante.

```
public class Maclasse<T> {  
}
```

Au sein de la classe

Les attributs et les méthodes seront définis en fonction du type générique.

```
public class Maclasse<T> {  
  
    private T element;  
  
    public MaClasse( T pelement ) {  
        this.setElement( pelement );  
    }  
}
```

CLASSES GÉNÉRIQUES

Utilisation

```
public class Start {  
  
    public static void main(String[] args) {  
  
        MaClasse<String> box = new MaClasse<>("toto");  
        System.out.println( box.getElement() );  
        box.setElement("tata");  
        System.out.println( box.getElement() );  
  
    }  
}
```

Restriction du typage

Il est possible de limiter les types utilisables pour une classe générique.

Par exemple, si l'on souhaite que notre classe soit limitée à des types dérivant d'une classe.

```
public class MaClasse< E extends SecondClasse > {  
  
    // TODO: implémentation de la classe générique  
}
```

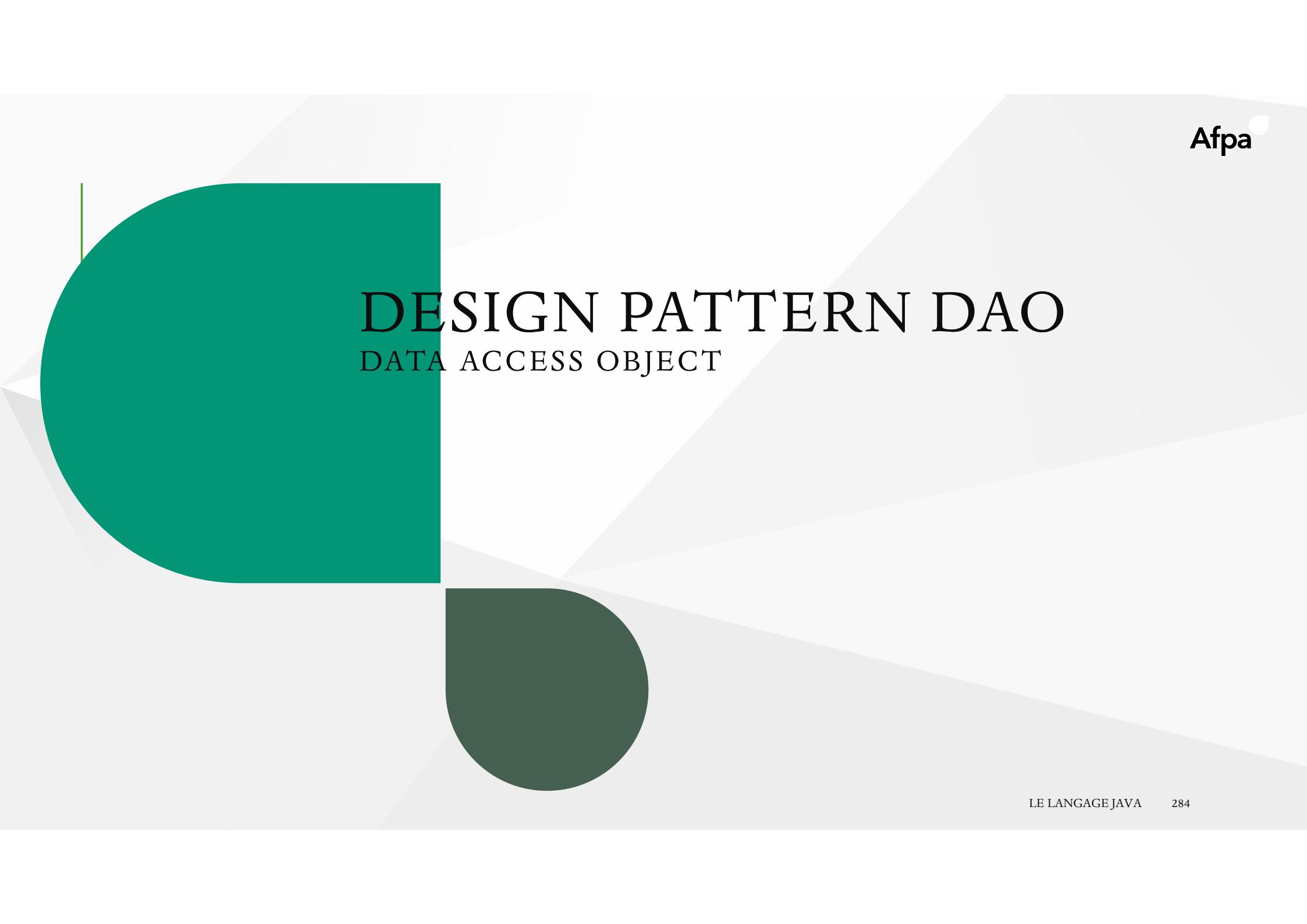
MÉTHODES GÉNÉRIQUES

- Les méthodes génériques ont un type paramétrisé (qui représente un paramètre de type générique) avant le type de retour de la déclaration de méthode
- Les méthodes génériques peuvent avoir différents types paramétrés séparés par des virgules dans la signature de méthode.
- Le corps d'une méthode générique est comme une méthode normale

```
public class Generic {  
  
    // méthode générique  
    public static <E> void display(E[] tab) {  
        for(E e : tab) {  
            System.out.printf("%s ", e);  
        }  
        System.out.println();  
    }  
  
    public static void main(String args[]) {  
        Integer[] i = {1, 2, 3};  
        System.out.println("Tableau des entiers:");  
        display(i); // passer un tableau entier  
  
        Character[] c = {'A', 'B', 'C'};  
        System.out.println("\nTableau des caractères:");  
        display(c); // passer un tableau de caractères  
    }  
}
```

```
Tableau des entiers:  
1 2 3
```

```
Tableau des caractères:  
A B C
```



DESIGN PATTERN DAO

DATA ACCESS OBJECT

POURQUOI UNE DAO ?

Inconvénients

Depuis la mise en place de JDBC, on a appliqué le pattern MVC mais ce n'est pas suffisant.

Dans la pratique, si on code de cette manière, nous lions alors très fortement, le code responsable des traitements métier au code responsable du stockage des données.

Problèmes

il devient impossible d'exécuter séparément l'un ou l'autre. Et ceci est fâcheux pour plusieurs raisons :

- il est impossible de mettre en place des tests unitaires
- impossible de tester le code métier de l'application sans faire intervenir le stockage (BDD, etc.)
- impossible de ne tester que le code relatif au stockage des données, obligation de lancer le code métier.
- il est impossible de changer de mode de stockage. (SGBD ou autre sans devoir réécrire tout le code métier).

ISOLER LE STOCKAGE DE DONNÉES

L'idée est qu'au lieu de faire communiquer directement nos objets métiers avec la base de données, ceux-ci vont parler avec la couche DAO.

Ensuite cette couche DAO va communiquer avec le système de stockage.

Au final, on isole purement et simplement le code responsable du stockage des données.

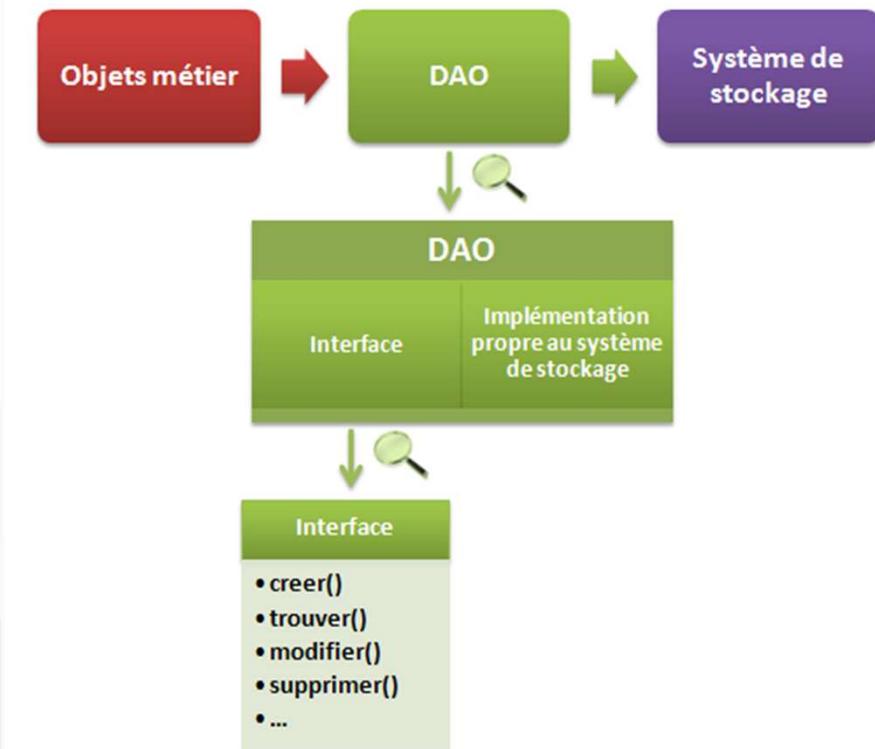
Le pattern DAO (Data Access Object) permet de faire le lien entre la couche métier et la couche persistante, ceci afin de centraliser les mécanismes de mapping entre notre système de stockage et nos objets Java. Il permet aussi de prévenir un changement éventuel de système de stockage de données.



PRINCIPE

Le principe du pattern DAO est de séparer la couche modèle d'une application en deux sous-couches distinctes

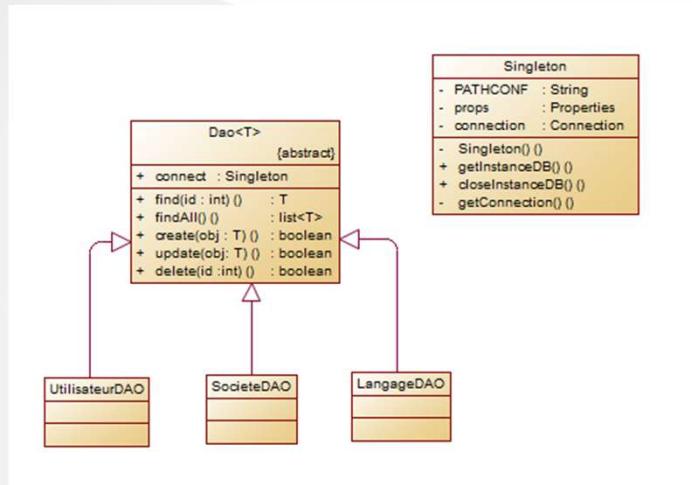
- une couche gérant les traitements métier appliqués aux données, souvent **nommée couche métier**. Typiquement, tout le travail de validation réalisé dans nos objets
- une couche gérant le stockage des données, logiquement **nommée couche de données**. Il s'agit là des opérations classiques de stockage : le CRUD.



MISE EN PLACE D'UNE DAO STANDARD

Selon le diagramme de classe, nous allons mettre en place cette DAO.

Tout d'abord, nous allons commencer par la classe abstraite DAO dont toutes nos DAO vont hériter.



```

public abstract class DAO<T> { 1 usage 1 inheritor new *

    protected Connection connect = Singleton.getInstanceDB(); 3 usages

    /**
     * Méthode de création d'un objet T
     * @param obj
     * @return validation de la création
     */
    public abstract int create(T obj); no usages 1 implementation new *

    /**
     * Méthode de suppression d'un objet T
     * @param obj
     * @return validation de la suppression
     */
    public abstract boolean delete(T obj); no usages 1 implementation new *

    /**
     * Méthode de Mise à jour d'un objet T
     * @param obj
     * @return validation de la mise à jour
     */
    public abstract boolean update(T obj); no usages 1 implementation new *

    /**
     * Méthode de recherche spécifique d'un objet T
     * @param pId
     * @return l'objet T recherché
     */
    public abstract T find(Integer pId); 1 usage 1 implementation new *

    /**
     * Méthode de recherche de tous les objets T
     * @return liste de tous les objets T
     */
    public abstract List<T> findAll(); 1 usage 1 implementation new *
}

```

COMPLÉTER LA DAO

Pour l'exemple, voici le début de l'implémentation de la classe PersonneDAO

Cette classe hérite de DAO<T> à qui on fournit le type concerné par la classe.

Pour chaque action du CRUD, il faut donc coder le traitement de la DAO vers la couche de données et s'assurer du traitement correct de la requête.

Evidemment, c'est un exemple et il manque quelques traitements dont les exceptions, les loggers pour parfaire le travail...

```
public class PersonneDAO extends DAO<Personne> { 3 usages new *
@Override no usages new *
public int create(Personne obj) {

    SimpleDateFormat simpleDateFormat = new SimpleDateFormat( pattern: "yyyy-MM-dd");
    int newId = 0;

    Stringbuilder insertSQL = new Stringbuilder();
    insertSQL.append("insert into Personne (civ_id,per_nom,per_prenom,per_adr,");
    insertSQL.append("per_cp, per_ville, per_datenissance)");
    insertSQL.append("values (?, ?, ?, ?, ?, ?, ?)");

    try {
        // preparation de la requete
        PreparedStatement pstat = connect.prepareStatement(insertSQL.toString(),
            PreparedStatement.RETURN_GENERATED_KEYS);

        // mise en place des parametres
        pstmt.setInt( parameterIndex: 1, obj.getCiviliteId());
        pstmt.setString( parameterIndex: 2, obj.getNom());
        pstmt.setString( parameterIndex: 3, obj.getPrenom());
        pstmt.setString( parameterIndex: 4, obj.getAdresse());
        pstmt.setString( parameterIndex: 5, obj.getCodePostal());
        pstmt.setString( parameterIndex: 6, obj.getVille());

        // conversion en SQL DATE
        String formattedDate = simpleDateFormat.format(obj.getDateNaissance());
        pstmt.setDate( parameterIndex: 7, Date.valueOf(formattedDate));

        // execution
        pstmt.executeUpdate();

        // recuperation de l'ID
        ResultSet rs = pstmt.getGeneratedKeys();
        if (rs.next()) {
            newId = rs.getInt( columnIndex: 1);
        }

        return newId;
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
```

COMPLÉTER LA DAO

La méthode find de ma classe PersonneDAO

```
@Override 1usage new*
public Personne find(Integer pId) {

    Personne personne = new Personne();
    StringBuilder selectById = new StringBuilder("select * from personne where per_id=?");

    try {
        // utilisation d'un prepareStatement
        PreparedStatement pstmt = connect.prepareStatement(selectById.toString());
        pstmt.setInt( parameterIndex: 1, pId);
        ResultSet resultSet = pstmt.executeQuery();

        while(resultSet.next()) {
            personne.setId(resultSet.getInt( columnLabel: "per_id"));
            personne.setCivileId(resultSet.getInt( columnLabel: "civ_id"));
            personne.setNom(resultSet.getString( columnLabel: "per_nom"));
            personne.setPrenom(resultSet.getString( columnLabel: "per_prenom"));
            personne.setAdresse(resultSet.getString( columnLabel: "per_adr"));
            personne.setEmail(resultSet.getString( columnLabel: "per_cp"));
            personne.setVille(resultSet.getString( columnLabel: "per_ville"));
            personne.setTelephone(resultSet.getInt( columnLabel: "per_tel"));
            personne.setFax(resultSet.getInt( columnLabel: "per_fax"));
            personne.setEmail(resultSet.getString( columnLabel: "per_email"));
            personne.setDateNaissance(resultSet.getDate( columnLabel: "per_datenaisance"));
        }
        return personne;
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
```

UTILISATION DE LA DAO

Pour utiliser notre DAO, il suffit de faire le mapping entre nos objets JAVA et notre DAO.

Pour cela, je crée un objet `PersonneDAO` où sont implémentés mes méthodes du CRUD

Puis selon mes besoins, je fais appel à la méthode du CRUD :

- `personneDAO.findAll()` pour afficher toutes les personnes
- `personneDAO.find(45)` pour afficher la personne ayant l'id égal à 45

```
// utilisation de mon implementation DAO de Personne
PersonneDAO personneDAO = new PersonneDAO();

for (Personne p : personneDAO.findAll()) {
    System.out.println(p.toString());
}

System.out.println("----");

System.out.println("Affichage d'une personne");
System.out.println(personneDAO.find( pid: 45));
```

GESTION DES EXCEPTIONS

CRÉATION D'UNE CLASSE DÉDIÉE DE GESTION
DES EXCEPTIONS

CRÉATION D'UNE CLASSE EXCEPTION POUR LA DAO

Afin de cacher la nature de stockage des données au reste de l'application et de mieux traiter les éventuelles erreurs.

Une bonne pratique est de masquer les exceptions spécifiques derrières des exceptions propres à la DAO.

- Ainsi, vous allez pouvoir gérer vos messages, évitant de remonter des informations propres à la bases de données.

```
public class DAOException extends RuntimeException {  
    /*  
     * Constructeurs  
     */  
    public DAOException( String message ) {  
        super( message );  
    }  
  
    public DAOException( String message, Throwable cause ) {  
        super( message, cause );  
    }  
  
    public DAOException( Throwable cause ) {  
        super( cause );  
    }  
}
```

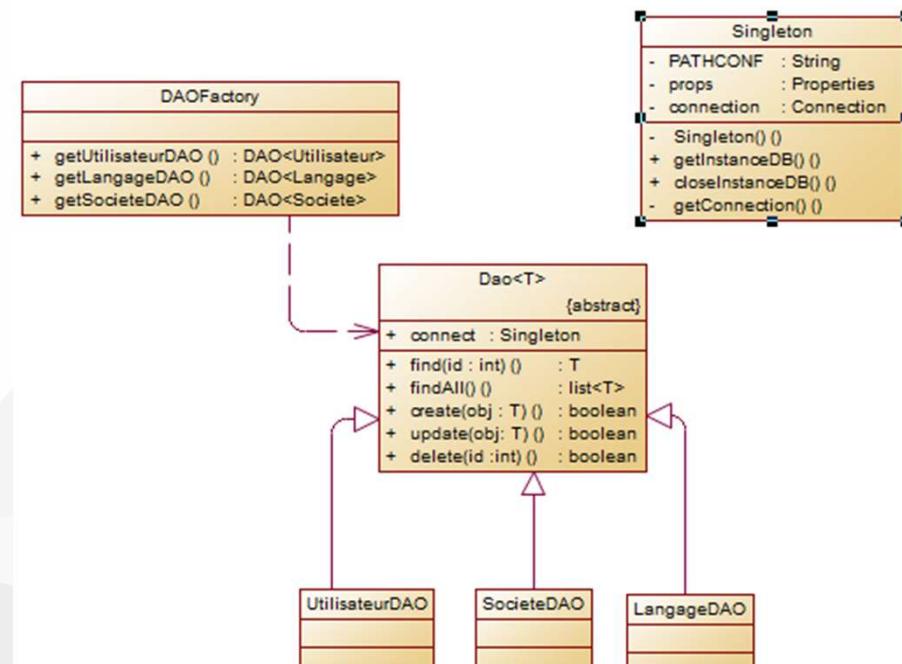
FACTORY PATTERN

CRÉATION D'UNE FABRIQUE

DAO + FACTORY

Pour plus de souplesse, la DAO sera couplée avec le Factory

On représente le pattern Factory selon le diagramme de classe ci-contre :



CRÉATION DE LA FACTORY

La Factory a pour rôle de créer nos DAO.

Cela ne va pas impacter le fonctionnement de notre application où il faudra simplement passer par la factory pour créer un objet DAO.

```
public class DAOFactory {  
  
    public static DAO<Societe> getSocieteDAO(){  
        return new SocieteDAO();  
    }  
  
    public static DAO<Utilisateur> getUtilisateurDAO(){  
        return new UtilisateurDAO();  
    }  
  
    public static DAO<Langage> getLangageDAO(){  
        return new LangageDAO();  
    }  
}
```

```
public class Main {  
  
    /**  
     * @param args  
     */  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
  
        DAO<Utilisateur> userDAO = new UtilisateurDAO();  
        DAO<Societe> userDAO = DAOFactory.getUtilisateurDAO();  
        Utilisateur user = new Utilisateur("test@test.fr", "a@frg", "Doe",  
        LocalDateTime.now());  
  
        if ( userDAO.create(user) ) then {  
            System.out.println("Création ok");  
        } else {  
            System.out.println("erreur pendant la Crédation");  
        }  
    }  
}
```

AJOUT DE LA GESTION DE NOTRE CONNEXION DANS LA FACTORY

On pourra dès lors demander à notre factory de faire en sorte de gérer la connection et ainsi de retier ce rôle à notre classe DAO<T>

Cette **factory** aura donc comme nouvelles responsabilités de :

- lire les informations de configuration depuis le fichier *properties* ;
- charger le driver JDBC du SGBD utilisé ;
- fournir une connexion à la base de données.

```
public class DAOFactory {  
  
    private static final String PATHCONF = "src/main/ressources/confDB.properties";  
    private static final Properties props = new Properties();  
    private static Connection connection;  
  
    // chargement du fichier de config  
    private DAOFactory() throws DAOException {  
        FileInputStream file = new FileInputStream(PATHCONF);  
        props.load(file);  
        props.setProperty("user", props.getProperty("jdbc.login"));  
        props.setProperty("password", props.getProperty("jdbc.password"));  
  
        connection = DriverManager.getConnection(  
            props.getProperty("jdbc.url"),  
            props  
        );  
    }  
  
    // singleton  
    public static Connection getInstanceDB() throws DAOException {  
        if (getConnection() == null){  
            new DAOFactory();  
        }  
        return getConnection();  
    }  
  
    // fermeture de l'instance  
    public static void closeInstanceDB() throws DAOException {  
        DAOFactory.getConnection().close();  
    }  
  
    // obtention de la connection  
    private static Connection getConnection() {  
        return connection;  
    }  
}
```

MODIFICATION DE NOTRE CLASSE DAO<T>

La modification consiste à modifier l'obtention de la connexion par la DAOFactory

Par contre, pas besoin de modifier nos DAO.

Il faudra dans l'application initialisé un objet DAOFactory qui servira à mettre en service votre connexion vers la couche de données.

```
public abstract class DAO<T> {

    public Connection connect = DAOFactory.getInstanceDB();

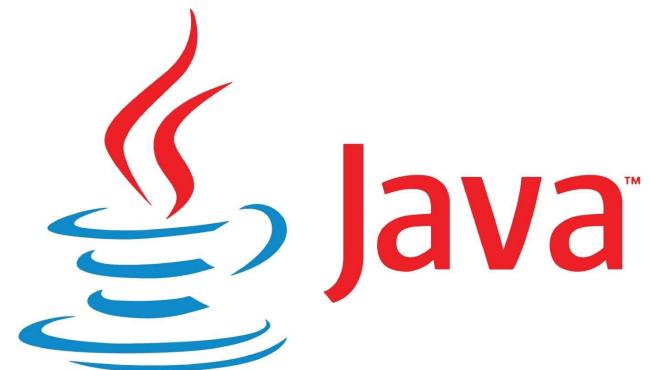
    /**
     * Méthode de création d'un objet T
     * @param obj
     * @return validation de la création
     */
    public abstract boolean create(T obj);

    /**
     * Méthode de suppression d'un objet T
     * @param obj
     * @return validation de la suppression
     */
    public abstract boolean delete(T obj);

    /**
     * Méthode de Mise à jour d'un objet T
     * @param obj
     * @return validation de la mise à jour
     */
    public abstract boolean update(T obj);

    /**
     * Méthode de recherche spécifique d'un objet T
     * @param pId
     * @return l'objet T recherché
     */
    public abstract T find(Integer pId);

    /**
     * Méthode de recherche de tous les objets T
     * @return liste de tous les objets T
     */
    public abstract List<T> findALL();
}
```



Journalisation par des loggers

POURQUOI AVEZ-VOUS BESOIN D'UNE JOURNALISATION ?

Il y a des points où une application s'intègre à d'autres services.

- Si l'intégration ne fonctionne pas, il devient facile de déterminer de quel côté se trouve le problème.
- Il est également souhaitable de consigner les informations importantes stockées dans une base de données. Par exemple, la création d'un utilisateur admin.

Les loggers sont des outils essentiels pour le développement et le débogage en Java. Ils permettent de suivre l'exécution de votre programme, de diagnostiquer les problèmes et de surveiller les performances.

Outils de journalisation en Java

- Parmi les solutions de journalisation bien connues en Java, nous pouvons souligner les suivantes :
 - [JUL](#) — `java.util.logging`
 - [JCL](#) — Jakarta Commons Logging
 - [Log4j](#)
 - Retour de session ([Logback](#))
 - [SLF4J](#) - Façade de journalisation simple pour Java

LE LOGGING

Définition

Le logging va permettre au sein d'une application :

- D'ajouter des traitements pour l'émission et le stockage de messages pour donner suite à des événements.
- De garder trace des exceptions avec des différents niveaux d'alertes.
 - En plus des exceptions, on va générer un message.
- De conserver des messages pour une exploitation immédiate ou à posteriori.
 - L'exception étant éphémère, le logging sera lui conservé.

Description

Une API de logging comporte 3 composants principaux :

- **Logger** : invoqué pour émettre un message avec un niveau de gravité associé
- **Formatter** : utilisé pour formater le contenu du message
- **Appender** : utilisé pour envoyer le message sur un support de stockage.

RECOMMANDATIONS ET CONVENTIONS

Voici quelques règles pour une bonne mise en œuvre du logging :

- Chaque message doit contenir la date/heure d'émission et la classe émettrice.
- Ne plus utiliser de System.out pour afficher des messages mais utiliser une API de Logging.
- Ne plus utiliser la méthode printStackTrace() de la classe Exception pour afficher des messages mais utiliser une API de Logging.
- Eviter les messages émis trop fréquemment (par exemple dans une boucle avec un nombre important d'itérations ou dans une méthode fréquemment invoquée, ...)
- Utiliser le niveau de gravité en adéquation avec le message.

Il est fortement recommandé d'utiliser une API de logging plutôt que d'utiliser la méthode System.out.println() ou System.err.println() pour plusieurs raisons :

- Une API de logging permet un contrôle sur le format des messages en proposant un format standard pouvant inclure des données telles que la date/heure, la classe, le thread, ...
- Une API de logging permet de gérer différentes cibles de stockage des messages.
- Une API de logging permet de modifier à l'exécution le niveau de gravité des messages pris en compte.

Sur des applications utilisées par plusieurs utilisateurs, par exemple une application web, il peut être très utile de faire figurer dans le message une identité sur le responsable de l'action (par exemple, l'adresse IP d'une requête http).

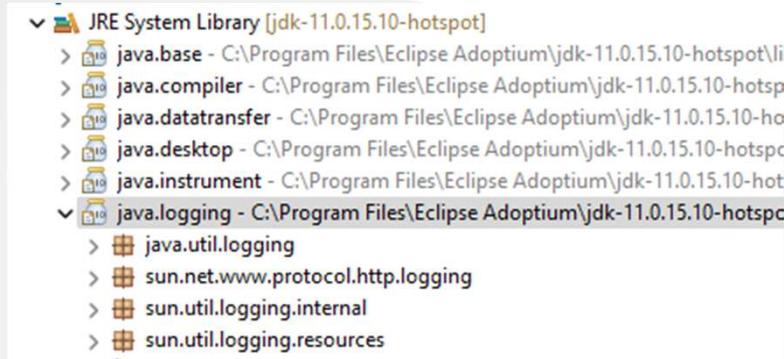
JUL

JAVA UTIL LOGGING

L'API LOGGING DE JAVA

L'API utilisée : `java.util.logging` est fournie par défaut dans le JDK.

- Introduit à la JDK 1.4



Pour utiliser un journal dans une classe, il faut donc :

1. Créer un attribut **statique** et **protégé** faisant référence au journal.
2. Appeler la méthode `getLogger()` de la classe `Logger` de l'API `Logging` qui prend en paramètre le nom du journal.
3. L'affecter à un flux de sortie, généralement un fichier. **Par défaut, il s'agit de la console.**

Bonne pratique : Comme pour les exceptions, afin de centraliser le logger en 1 seul endroit, on va se créer sa propre classe Logger.

LES DIFFÉRENTS NIVEAUX DE GRAVITÉ

Niveau	Description
ALL	Tous les niveaux
SEVERE	Niveau le plus élevé
WARNING	Avertissement
INFO	Information
CONFIG	Configuration
FINE	Niveau faible
FINER	Niveau encore plus faible
FINEST	Niveau le plus faible
OFF	Aucun niveau

- Pour poster un message dans le journal, il faut utiliser la méthode `log(level, msg)` de l'objet `Logger`
 - En argument, `level` indique le niveau de gravité et `msg`, le message.
 - Si le `level` du message est l'un de ceux géré alors celui-ci sera envoyé dans le journal
- Il existe d'autres méthodes pour log des messages
 - `Severe()`, `warming()`, `info()`, `config()`, `fine()`, `finer()`, `finest()`
 - `Logp()`, `logrb()`, `entering()`, `exiting()`, `throwing()`
- Nous avons 7 + 2 niveaux de gravité
 - `SetLevel(Level.xx)` permet de choisir le niveau de gravité notre `Logger`
 - Par Défaut que les messages de niveaux supérieurs ou égal à `INFO`

L'API LOGGING DE JAVA

Java Logger

`Java.util.logging.logger` est la classe qui permet de créer un logger de message en Java

```
Logger logger = Logger.getLogger(MyClass.class.getName());
```

Java logging levels

`Java.util.logging.level` définit les différents niveaux de logging.

Les logs seront générés pour tous les niveaux égaux ou supérieurs au niveau défini.

1. SEVERE (highest)
2. WARNING
3. INFO
4. CONFIG
5. FINE
6. FINER
7. FINEST

Par exemple, si le niveau d'enregistrement est réglé sur INFO, des journaux seront générés pour les messages INFO, WARNING et SEVERE.

```
logger.setLevel(Level.FINE);
```

EXEMPLE

Création d'une classe logger qui va contenir :

- Un objet static de type Logger
- Une méthode qui va me retourner mon logger

Ensuite, à l'endroit où je souhaite logger un message, j'utilise la méthode appropriée en faisant appel à ma classe statique.

- On peut optimiser l'appel en important directement la méthode statique dans la classe

```
import static fr.pompey.cda23016.logger.MyLogger.getLogger;
```



```
import java.util.logging.Logger;  
  
public class MyLogger { 6 usages new *  
  
    private static Logger logger = Logger.getLogger(MyLogger.class.getName());  
  
    public static Logger getLogger() { 5 usages new *  
        return logger;  
    }  
  
    public MyLogger() {} no usages new nov nov. 22, 2024 10:03:07 AM fr.pompey.cda23016.App start  
}  
|  
| nov. 22, 2024 10:03:07 AM fr.pompey.cda23016.App start  
INFO: fr.pompey.cda23016.App : message infos  
nov. 22, 2024 10:03:07 AM fr.pompey.cda23016.App start  
SEVERE: fr.pompey.cda23016.App : message severe  
nov. 22, 2024 10:03:07 AM fr.pompey.cda23016.App start  
WARNING: fr.pompey.cda23016.App : message warming  
nov. 22, 2024 10:03:07 AM fr.pompey.cda23016.App start  
INFO: fr.pompey.cda23016.App : message infos
```

```
/* ----- METHODE LOGGER ----- */  
MyLogger.getLogger().log(Level.INFO, msg: getClass().getName()+" : message infos");  
MyLogger.getLogger().log(Level.SEVERE, msg: getClass().getName()+" : message severe");  
MyLogger.getLogger().log(Level.WARNING, msg: getClass().getName()+" : message warming");  
  
// optimisation en utilisant un import static de la méthode  
// et faisant appel à la méthode de niveau correspondant  
getLogger().info( msg: getClass().getName()+" : message infos");
```

LE LOGGING

Environnement et configuration

Le niveau de gravité des messages sera différent en fonction de l'environnement (développement, Production, test etc...) mais le code de l'application doit rester le même.

On peut aussi agir sur le formatage du message et de même, choisir différentes sorties (fichier, console, BDD etc...)

- On agira alors sur la configuration du logging.

Inconvénients

Les API de logging ont plusieurs inconvénients :

- Il faut définir avec précision les messages à ajouter dans les journaux et la pertinence des informations
- Il faut définir avec précision le niveau de gravité des messages.
- L'utilisation d'une API peut dégrader les performances d'une application.

LE LOGGING

Formats de sortie

Handler	Description
ConsoleHandler	correspond au flux d'erreur standard <code>System.err</code>
FileHandler	simple fichier texte du système de fichiers
SocketHandler	flux réseau vers une machine et un port à déterminer
MemoryHandler	buffer cyclique en mémoire vive
StreamHandler	flux quelconque de sortie

Formatage

Il existe deux classes de formatage de base dans l'API Logging :

Formatter	Description
SimpleFormatter	Formatage en texte simple
XMLFormatter	Format XML

L'API LOGGING DE JAVA

Java Handler

Nous pouvons ajouter plusieurs Handler à notre logger :

Il y a **2 handler** par défaut en Java.

- ConsoleHandler (console).
- FileHandler (fichier).

Nous pouvons également faire appel à des Handler spécifiques.

<https://docs.oracle.com/en/java/javase/14/docs/api/java.logging/java/util/logging/FileHandler.html>

Java Formatter

Les formatter sont utilisée pour formater les messages dans les journaux.

Il existe deux formats dans l'API :

1. **SimpleFormatter** : Ce formateur génère des messages texte avec des informations de base. ConsoleHandler utilise cette classe de formater pour imprimer des messages de journalisation pour la console.
2. **XMLFormatter** : Ce formateur génère un message XML pour le journal, FileHandler utilise XMLFormatter comme formateur par défaut.

[https://docs.oracle.com/en/java/javase/14/docs/api/java.logging/java/util/logging/SimpleFormatter.html#format\(java.util.logging.LogRecord\)](https://docs.oracle.com/en/java/javase/14/docs/api/java.logging/java/util/logging/SimpleFormatter.html#format(java.util.logging.LogRecord))

L'API LOGGING DE JAVA

`java.util.logging.LogManager` est la classe qui lit la configuration de journalisation, crée et maintient les instances d'enregistrement.

Nous pouvons utiliser cette classe pour définir notre propre configuration spécifique à l'application.

Voici un exemple ci-contre de fichier de configuration API Logging Java.

Si nous ne précisons aucune configuration, c'est lu dans JRE Home lib/logging.properties

```
handlers= java.util.logging.ConsoleHandler

.level= FINE

# default file output is in user's home directory.
java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.limit = 50000
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter

# Limit the message that are printed on the console to INFO and above.
java.util.logging.ConsoleHandler.level = INFO
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter

com.journaldev.files = SEVERE
```

```
LogManager.getLogManager().readConfiguration(new FileInputStream("mylogging.properties"));
```

EXEMPLE DE FICHIER DE CONFIGURATION

Ci contre, un exemple de fichier de configurations d'un logger.

- Deux handler : console et fichier
- Niveau de capture : INFO
- Définition du pattern pour le nom du fichier avec interdiction de réécriture
- Définition d'un pattern de message

```
handlers= java.util.logging.ConsoleHandler, java.util.logging.FileHandler
.level= INFO
java.util.logging.ConsoleHandler.level = INFO
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter

#
# "/" the local pathname separator
# "%t" the system temporary directory
# "%h" the value of the "user.home" system property
# "%g" the generation number to distinguish rotated logs
# "%u" a unique number to resolve conflicts
# "%" translates to a single percent sign "%"
#
java.util.logging.FileHandler.pattern = %h/testMavenLog.%u.xml
java.util.logging.FileHandler.append = false

# On active les logs du package sur INFO (et donc WARNING et SEVERE).
fr.pompey.cda23016 = INFO

# On change le format des logs pour notre SimpleFormatter.
# 1 : la date de production du log.
# 2 : la source (nom du type et nom de la méthode) si elle est connue ou bien le nom du logger.
# 3 : le nom du logger (celui passé en paramètre de la méthode Logger.getLog()).
# 4 : le niveau du log (INFO, SEVERE...).
# Dans l'exemple ci-dessus, je l'affiche sur 10 caractères et aligné par la gauche (-) : %4$-10s.
# 5 : le message du log.
# 6 : l'exception associée si elle est spécifiée.

java.util.logging.SimpleFormatter.format=[%1$s] %4$-10s | (%3$s) %2$-50s | %5$s\n
```

UTILISATION PAR L'EXEMPLE

Pour faire appel à notre configuration, on peut avant le lancement de notre main, faire appel à un bloc static pour charger la configuration.

- Dans ce bloc, j'utilise une instance du LogManager pour charger mon fichier de configuration.

```
import static fr.pompey.cda23016.logger.MyLogger.getLogger;
import static fr.pompey.cda23016.logger.MyLogger.getLogManager;

/**
 * App
 */
public class App {  ↳ nejero.+

    /**
     * BLOC CHARGE avant le main en mémoire
     */
    static {
        try {
            getLogManager().readConfiguration(
                App.class.getClassLoader().getResourceAsStream(name: "logging.properties"));
        } catch (IOException e) {
            getLogger().log(Level.SEVERE, msg: "Error loading properties", e);
        }
    }
}
```

SORTIES

En sortie de programme, j'obtiens un affichage console de mes logs et un fichier XML dans mon répertoire utilisateur.

Sortie console :

```
C:\Users\jboeb\.jdks\openjdk-21.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.3.1\lib\idea_rt.jar" -Dfile.encoding=UTF-8 fr.pompey.cda23016.App
[2024-11-25T13:40:31.209993300+01:00[Europe/Paris]] INFO    | (fr.pompey.cda23016.logger.MyLogger) - Hello World!
[2024-11-25T13:40:31.253546200+01:00[Europe/Paris]] SEVERE   | (fr.pompey.cda23016.logger.MyLogger) - 
[2024-11-25T13:40:31.254968400+01:00[Europe/Paris]] WARNING | (fr.pompey.cda23016.logger.MyLogger) - 
[2024-11-25T13:40:31.254968400+01:00[Europe/Paris]] INFO    | (fr.pompey.cda23016.logger.MyLogger) -
```

Sortie fichier :



```
<message>fr.pompey.cda23016.App : message infos</message>
</record>
<record>
<date>2024-11-25T12:40:31.253546200Z</date>
<millis>1732538431253</millis>
<nanos>546200</nanos>
<sequence>1</sequence>
<logger>fr.pompey.cda23016.logger.MyLogger</logger>
<level>SEVERE</level>
<class>fr.pompey.cda23016.App</class>
<method>start</method>
<thread>1</thread>
<message>fr.pompey.cda23016.App : message severe</message>
</record>
<record>
<date>2024-11-25T12:40:31.254968400Z</date>
<millis>1732538431254</millis>
<nanos>968400</nanos>
<sequence>2</sequence>
<logger>fr.pompey.cda23016.logger.MyLogger</logger>
<level>WARNING</level>
<class>fr.pompey.cda23016.App</class>
<method>start</method>
<thread>1</thread>
<message>fr.pompey.cda23016.App : message warming</message>
</record>
<record>
<date>2024-11-25T12:40:31.254968400Z</date>
<millis>1732538431254</millis>
```

LOG4J2

API [HTTPS://LOGGING.APACHE.ORG/LOG4J/2.X/](https://logging.apache.org/log4j/2.x/)



LOG4J2

LOGGING FOR JAVA VERSION 2.X

Projet Open Source distribué sous licence Apache

- API qui permet d'utiliser et de paramétriser un système de gestion de journaux (logs)
- Gère plusieurs niveaux de gravités
- Plusieurs flux (fichier sur disque, journal d'événements Windows, connexion TCP/IP, BDD etc...)

[Log4j2](#) utilise trois composants principaux pour assurer l'envoi de messages selon un certain niveau de gravité et contrôler à l'exécution le format et la ou les cibles de destination des messages :

- [Category/Logger](#) : ces classes permettent de gérer les messages associés à un niveau de gravité
- [Appenders](#) : ils représentent les flux qui vont recevoir les messages de log
- [Layouts](#) : ils permettent de formater le contenu des messages de log

La popularité de [Log4J2](#) est largement liée à sa facilité d'utilisation, ses nombreuses fonctionnalités extensibles et sa fiabilité.

Comme le logging n'est jamais une fonctionnalité principale d'une application, Log4j se veut facile à mettre en œuvre.

PRINCIPALES CARACTÉRISTIQUES

Les principales caractéristiques de Log4j sont :

- Utilisation d'une hiérarchie de loggers basée sur leurs noms
- Support en standard de plusieurs niveaux de gravité
- Configuration externalisable dans un fichier au format .properties ou XML
- Thread-safe
- Optimisé pour réduire les temps de traitements
- Prise en charge des exceptions associables aux messages
- Support de nombreuses cibles de destination des messages
- Extensible

- Un autre avantage de log4J est de pouvoir être utilisé avec toutes les versions du JDK depuis la 1.1.
- L'externalisation de la configuration de Log4j dans un fichier externe permet de modifier la configuration des traitements de logging sans avoir à modifier le code source de l'application.
- La hiérarchie des loggers permet un contrôle très fin de la granularité des messages ce qui réduit le volume de données des logs.
- Log4j propose en standard plusieurs destinations de stockage des messages : fichiers, gestion d'événements Windows, Syslog Unix, base de données, email, message JMS, ...

INSTALLATION PAR MAVEN

Log4j2 peut s'utiliser de deux manières (manuellement ou avec maven).

Manuellement, il faut inclure les librairies dans son **classpath** `log4j-api-x.x.x.jar` et `log4j-core-x.x.x.jar`

Par Maven, l'installation est plus simple puisque il suffit de rajouter les dépendances nécessaires à votre fichier POM (voir ci-contre)

Attention : à prendre les dernières versions stables de l'API

```
<!-- https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-core -->
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.24.1</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-api -->
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.24.1</version>
</dependency>
```

ACQUISITION D'UN LOGGER

Pour acquérir un logger, il faut utiliser la méthode statique [LogManager.getLogger](#).

- Bien entendu, la classe [LogManager](#) doit être celle localisée dans le package [org.apache.logging.log4j](#).
- votre méthode d'acquisition peut accepter en paramètre plusieurs types et notamment une chaîne de caractères.
 - Il s'agit du nom du logger : il est très important, car il servira ultérieurement à configurer ce logger.
 - [Une bonne pratique est de nommer le logger avec le nom de la classe MyClass.class.getName\(\)](#)

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class MyLog4j2 { 4 usages new *

    // etape 1 : acquisition d'un logger
    private static final Logger log4J = LogManager.getLogger(MyLog4j2.class);

    public static Logger getLog4J() { 2 usages new *
        return log4J;
    }

    public MyLog4j2() {} no usages new *

}
```

NIVEAUX DE LOGS

Il nous faut maintenant bien comprendre les différents niveaux de logs avec l'API Log4J2.

Chaque niveau de log est associé à une valeur numérique lors de l'instanciation de la constante.

- **OFF** = Integer.MIN_VALUE
- **FATAL** = 100
- **ERROR** = 200
- **WARN** = 300
- **INFO** = 400
- **DEBUG** = 500
- **TRACE** = 600
- **ALL** = Integer.MAX_VALUE

Event Level	LoggerConfig Level						
	TRACE	DEBUG	INFO	WARN	ERROR	FATAL	OFF
ALL	YES	YES	YES	YES	YES	YES	NO
TRACE	YES	NO	NO	NO	NO	NO	NO
DEBUG	YES	YES	NO	NO	NO	NO	NO
INFO	YES	YES	YES	NO	NO	NO	NO
WARN	YES	YES	YES	YES	NO	NO	NO
ERROR	YES	YES	YES	YES	YES	NO	NO
FATAL	YES	YES	YES	YES	YES	YES	NO
OFF	NO	NO	NO	NO	NO	NO	NO

Trace < Debug < Info < Avertissement < Erreur < Fatal

PRODUIRE UN LOG

Comme vous pourrez le constater, le log informatif n'est pas affiché : c'est normal, car par défaut les logs sont activés qu'à partir du niveau Level.ERROR.

- Pour changer cela, il va falloir fournir un fichier de configuration pour Log4J2 afin de modifier le niveau de capture.

```
/* ----- METHODE LOG4J2 ----- */
MyLog4j2.getLog4J().info("LOG4J2");

try {
    int value = (int) (Math.random() * 2);
    int result = 3 / value;
    MyLog4j2.getLog4J().info( message: "VALUE = {} - RESULT = {}", value, result);
} catch (Exception e) {
    MyLog4j2.getLog4J().error( message: "Houston, we have a problem : {}", e);
}
```

```
C:\Users\jboeb\.jdks\openjdk-21.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2021.3.1\lib\jbagent.jar" -Dfile.encoding=UTF-8 fr.pompey.cda23016.logger.MyLog4j2
Hello World!
16:17:34.087 [main] ERROR fr.pompey.cda23016.logger.MyLog4j2 - Houston, we have a problem : {}
java.lang.ArithmetricException Create breakpoint : / by zero
    at fr.pompey.cda23016.App.start(App.java:49)
    at fr.pompey.cda23016.App.main(App.java:38)

Process finished with exit code 0
```

CONFIGURER NOS LOGS

Vous pouvez configurer Log4J grâce à des fichiers de configurations selon plusieurs formats :

- [Format XML](#)
- [Format properties](#)
- [Format JSON](#)
- [Format YAML](#)

Tous ces différents formats sont basés sur le même principe, il est alors facile de passer d'un format à un autre.

1. Si Log4J2 ne trouve pas trace d'un fichier de configuration, alors il est capable il est configuré avec une sortie en console et un niveau d'erreur sur ERROR
2. Cependant, s'il trouve trace d'un fichier de configuration nommé par défaut *log4j2.xml* (*ou selon le format*) alors celui-ci prend le dessus sur la configuration par défaut.

CONFIGURER NOS LOGS

<https://logging.apache.org/log4j/2.x/manual/layouts.html>

APPENDERS

Log4J définit la notion **d'Appender** pour identifier la destination de nos logs.

- Il existe plusieurs Appender et notamment :
 - L'Appender [console](#)
 - L'Appender [JDBC](#)
 - L'Appender [JPA](#)
 - L'Appender [File](#)
 - L'Appender [NoSQL](#)
 - L'Appender [RollingFile](#)
 - L'Appender [Socket](#)

Formaters

Vous pouvez également formater vos logs :

- %t : permet d'injecter le nom du thread ayant produit le log.
- %-5p : permet d'injecter le niveau (level) du log. Dans notre cas, il occupera cinq caractères et sera aligné par la gauche.
- %-60c : permet d'injecter le nom du logger ayant généré ce log (dans notre cas, le nom pleinement qualifié de la classe).
- %m : permet d'injecter le message du log.
- %F : permet d'injecter le nom du fichier Java contenant la ligne de code ayant produit le log.
- %L : permet d'injecter le numéro de ligne de code, dans le fichier considéré, ayant produit le log.
- %n : tout simplement pour injecter un retour à la ligne en fin de log.

EXEMPLE DE FICHIER DE CONFIGURATION

Voici, un exemple de fichier log4j2.xml contenant une configuration sur la sortie console où on indique un pattern pour le message et un niveau de capture correspondant à INFO

Ce fichier est placé dans le répertoire de ressources du projet.

- Résultat :

Cette fois-ci avec la configuration, on observe bien la capture des messages de niveau INFO et un message définit avec un pattern spécifique.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://logging.apache.org/Log4j/2.0/config"
    status="WARN">
    <Appenders>
        <Console name="stdout" target="SYSTEM_OUT">
            <PatternLayout pattern="[%t] %-5p | %-30c | %m (%F:%L)%n"/>
        </Console>
    </Appenders>

    <Loggers>
        <Root level="info">
            <AppenderRef ref="stdout"/>
        </Root>
    </Loggers>
</Configuration>
```

```
Hello World!
[main] INFO  | fr.pompey.cda23016.logger.MyLog4j2 | LOG4J2 (App.java:492)
[main] ERROR | fr.pompey.cda23016.logger.MyLog4j2 | Houston, we have a problem : {} (App.java:499)
java.lang.ArithmetricException Create breakpoint : / by zero
    at fr.pompey.cda23016.App.start(App.java:496) [classes/:?]
    at fr.pompey.cda23016.App.main(App.java:38) [classes/:?]

Process finished with exit code 0
```

```
C:\Users\jboeb\.jdks\openjdk-21.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2021.3.1\lib\idea_rt.jar" -Dfile.encoding=UTF-8
Hello World!
[main] INFO  | fr.pompey.cda23016.logger.MyLog4j2 | LOG4J2 (App.java:492)
[main] INFO  | fr.pompey.cda23016.logger.MyLog4j2 | VALUE = 1 - RESULT = 3 (App.java:497)
```

EXEMPLE DE FICHIER DE CONFIGURATION

Voici, un exemple de fichier log4j2.xml contenant une configuration sur la sortie console et une sortie sur fichier où on indique un pattern pour le message et un niveau de capture correspondant à INFO avec chaque sortie, un pattern différent.



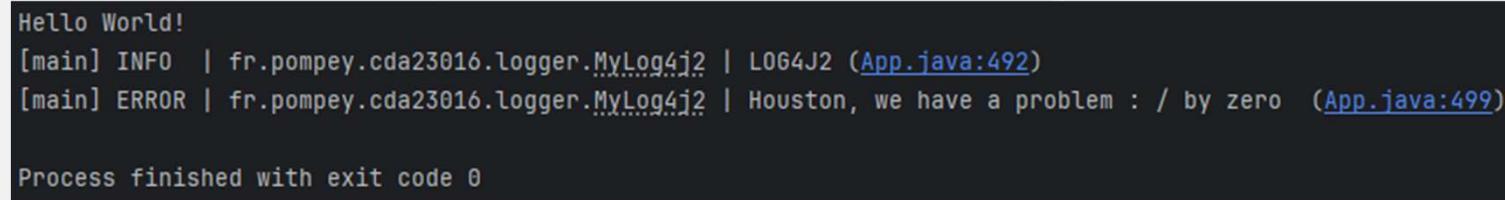
The screenshot shows a terminal window titled "all.log". The window has standard OS X-style controls at the top. Below the title, there are three menu items: "Fichier", "Modifier", and "Affichage". The main area of the window displays two lines of log output:

```
2024-11-26 16:48:37.676 [main] INFO fr.pompey.cda23016.logger.MyLog4j2 - LOG4J2
2024-11-26 16:48:37.682 [main] ERROR fr.pompey.cda23016.logger.MyLog4j2 - Houston, we have a problem : / by zero
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<Configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://logging.apache.org/log4j/2.0/config"
  status="WARN">
  <Properties>
    <Property name="basePath">C:/temp/logs</Property>
  </Properties>
  <Appenders>
    <Console name="stdout" target="SYSTEM_OUT">
      <PatternLayout pattern="[%t] %-5p | %-30c | %m (%F:%L)%n"/>
    </Console>

    <File name      ="MyFile"
          fileName   ="${basePath}/all.log"
          immediateFlush = "false"
          append      = "false">
      <PatternLayout pattern="%d{yyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
    </File>
  </Appenders>

  <Loggers>
    <Root level="info">
      <AppenderRef ref="stdout"/>
      <AppenderRef ref="MyFile"/>
    </Root>
  </Loggers>
</Configuration>
```



The screenshot shows a terminal window displaying the output of a Java application named "Hello World!". The window has standard OS X-style controls at the top. The main area of the window displays the following log output:

```
Hello World!
[main] INFO  | fr.pompey.cda23016.logger.MyLog4j2 | LOG4J2 (App.java:492)
[main] ERROR | fr.pompey.cda23016.logger.MyLog4j2 | Houston, we have a problem : / by zero (App.java:499)

Process finished with exit code 0
```

SLF4J

SIMPLE LOGGING FAÇADE FOR JAVA

[HTTPS://WWW.SLF4J.ORG/](https://www.slf4j.org/)



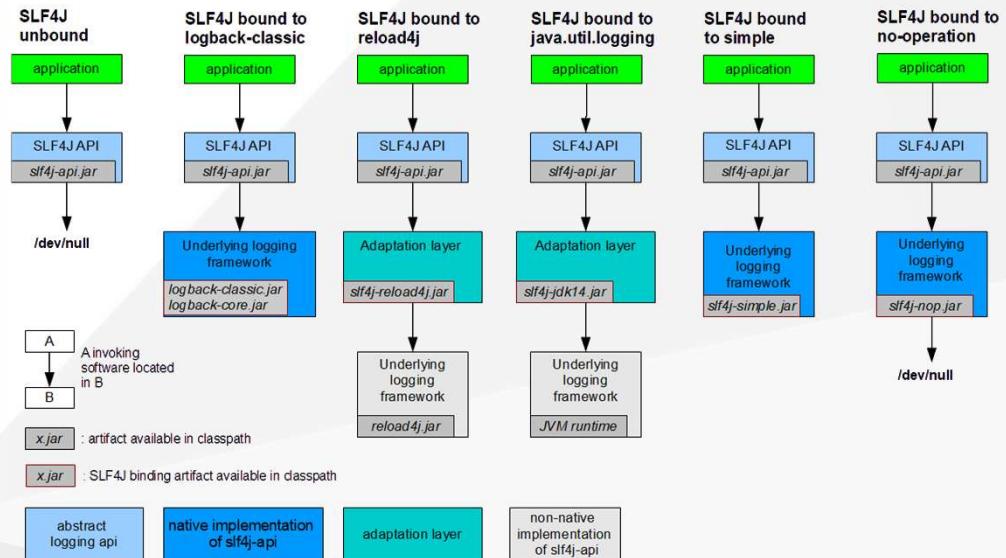
INTRODUCTION

SLF4J (Simple Logging Facade For Java) agit comme une façade pour différents Framework de journalisation.

Il offre une API générique, rendant la journalisation indépendante de l'implémentation réelle.

- Cela permet à différents Framework de journalisation de coexister. Et cela aide à migrer d'un Framework à un autre.

Il peut donc prendre en compte des Framework comme Jakarta Commons Logging (API Standard), Logback et Log4j



INSTALLATION

2 solutions pour installer SLF4J : soit par le biais d'une installation manuelle, soit en utilisant l'outil Apache Maven.

1. A minima, il vous faut la dépendance SLF4J.
2. Ensuite, il vous faut télécharger les dépendances de votre logger comprenant le driver pour que SLF4J puisse utiliser l'API Log4J2

```
<!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-api -->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>2.0.16</version>
</dependency>
```

NIVEAUX DE LOGS

Les niveaux de logs supportés par SLF4J sont les suivants :

- **error**
 - permet d'autoriser les messages de niveau « erreur / error » dans les logs. Dans ce cas, le message d'erreur et l'exception associée, s'ils sont passés au logger utilisé, pourront apparaître dans les logs.
- **warn**
 - permet d'autoriser les messages de niveau « avertissement / warning » dans les logs. Si la configuration de votre système de log autorise la production des avertissements, alors les erreurs seront aussi ajoutées aux logs.
- **info**
 - d'autoriser les messages de niveau « informatif » dans les logs. Si la configuration de votre système de log autorise la production des messages informatifs, alors les avertissements et les erreurs seront aussi générés.
- **debug**
 - permet les messages de niveau « déboggage » dans les logs. Si la configuration de votre système de log autorise la production des messages de debug, alors les messages informatifs, les avertissements et les erreurs seront aussi générés.
- **trace**
 - permet les messages de niveau « trace » dans les logs. Si la configuration de votre système de log autorise la production des traces, alors les messages de debug, les messages informatifs, les avertissements et les erreurs seront aussi générés.

PRODUIRE UN LOGGER

Pour acquérir un logger, il faut utiliser la méthode statique `org.slf4j.LoggerFactory.getLogger`

ATTENTION : Vous ne pouvez pas implémenter plusieurs logger en même temps.

L'intérêt de SLF4J est de permettre de passer d'une API à une autre sans devoir retoucher le code.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class MySlf4j { 3 usages new *

    private static Logger logSlf4j = LoggerFactory.getLogger(MySlf4j.class); 1 usage

    public static Logger getlogSlf4j() { 1 usage new *
        return logSlf4j;
    }
}
```

```
MySlf4j.getlogSlf4j().info("test info");
MySlf4j.getlogSlf4j().error("test erreur");
```

INTERFAÇADE DU LOGGER

Logback

Pour utiliser Logback avec SLF4J, il faut utiliser cette dépendance ci-dessous.

- Par défaut, sans configuration, il utilise un [appender Console](#) et un [niveau DEBUG](#)
- Sinon, il faut créer un fichier [logback.xml](#) pour y contenir la configuration souhaitée.

```
<!-- https://mvnrepository.com/artifact/ch.qos.logback/logback-classic -->
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.5.12</version>
</dependency>
```

JUL

Pour utiliser JUL avec SLF4J, il faut utiliser ces deux dépendances ci-dessous.

- Par défaut, sans configuration, il utilise un [appender Console](#) et un [niveau ERROR](#)
- Sinon, il faut créer un fichier [logging.properties](#) pour y contenir la configuration souhaitée.

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-jdk14</artifactId>
    <version>2.0.16</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.slf4j/jul-to-slf4j -->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>jul-to-slf4j</artifactId>
    <version>2.0.16</version>
</dependency>
```

INTERFAÇADE DU LOGGER

LOG4J

Pour utiliser Log4J avec SLF4J, il faut utiliser ces trois dépendances ci-contre.

- Par défaut, sans configuration, il utilise un appender **Console** et un niveau **ERROR**
- Sinon, il faut créer un fichier **log4j2.xml** pour y contenir la configuration souhaitée.
- **Attention :** Pour interfaçer Log4J2 avec SLF4J, il faut prendre la version 1.7.36 de SLF4J

```
<!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-api -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>1.7.36</version>
</dependency>
```

```
<!-- https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-slf4j-impl -->
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-slf4j-impl</artifactId>
  <version>2.24.2</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-api -->
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-api</artifactId>
  <version>2.24.2</version>
</dependency>

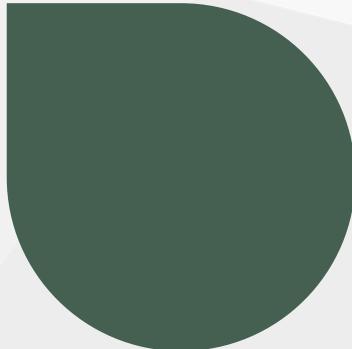
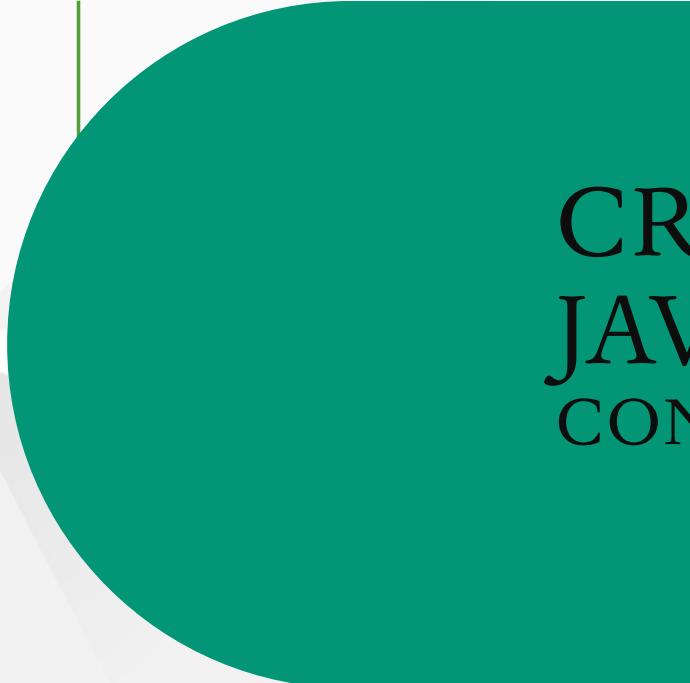
<!-- https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-core -->
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-core</artifactId>
  <version>2.24.2</version>
</dependency>
```

MERCI !

Jérôme BOEBION
Concepteur Développeur d'Applications
Version 2.0 - révision 2024

Afpa
se former, avancer



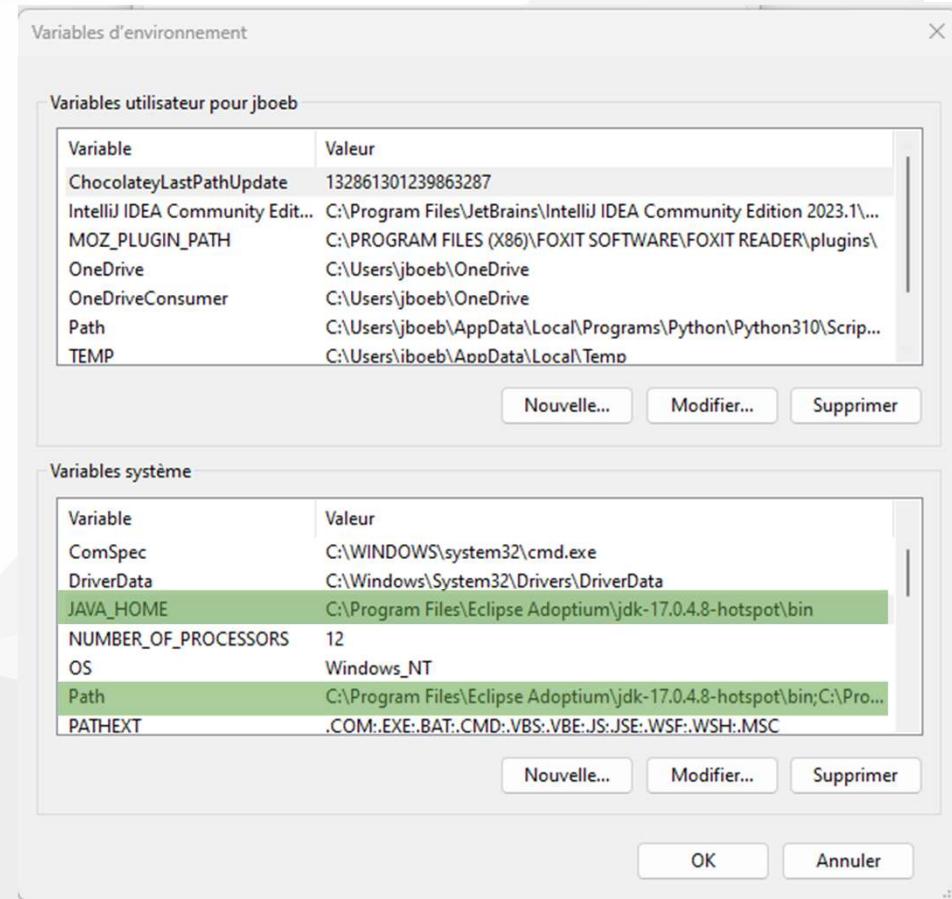


CRÉATION D'UN EXE EN JAVA

CONSTRUIRE SON APPLICATION

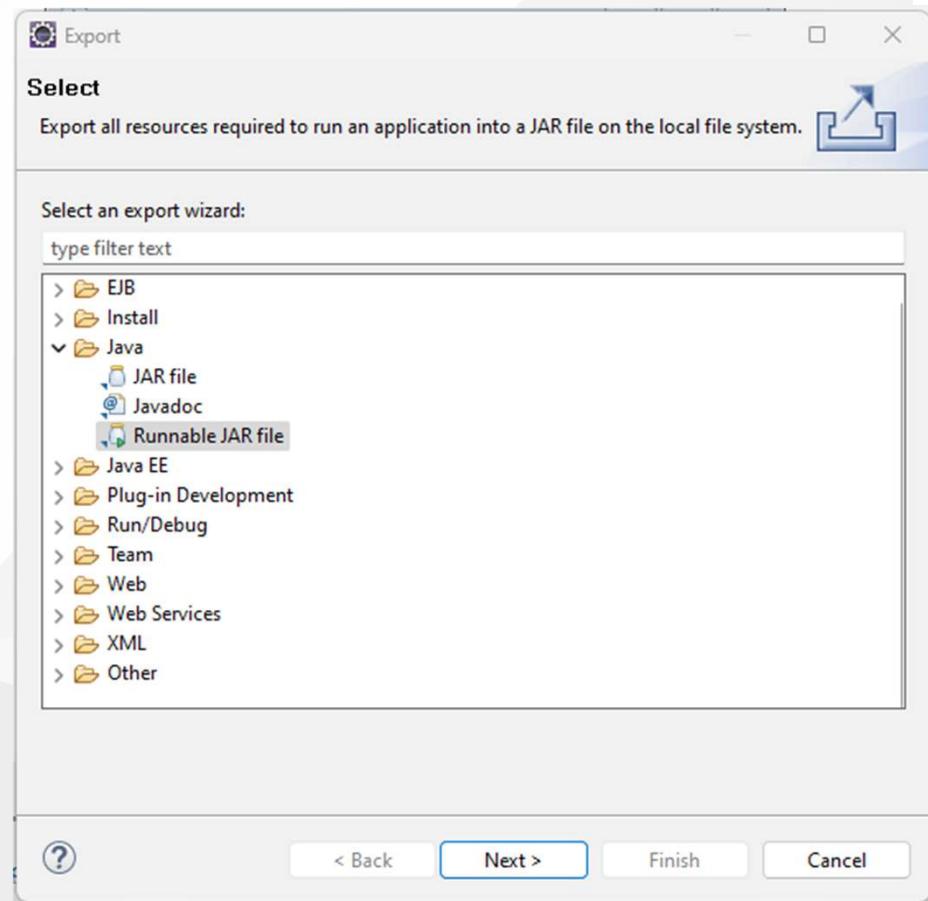
CONTRÔLER.

Vérifier la présence de votre JDK dans vos variables d'environnements.



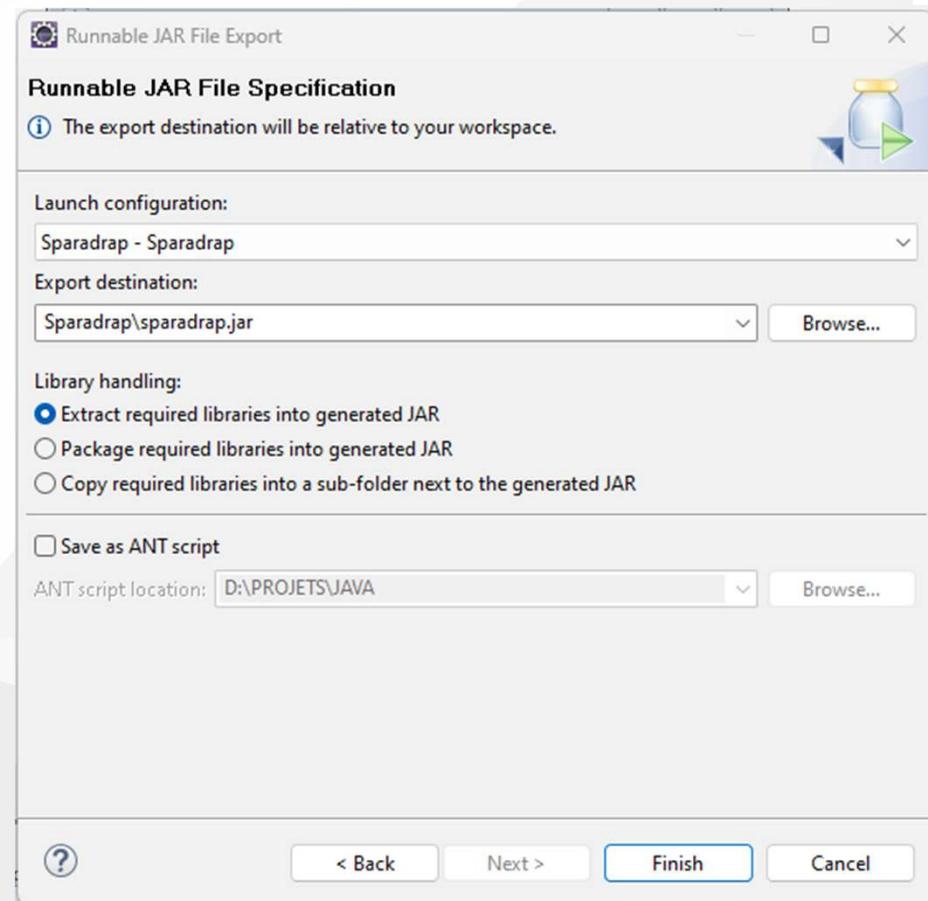
DEPUIS ECLIPSE

1. Clic droit sur le projet
2. Refresh puis Export
 1. Sélectionner [Runnable JAR File](#)



CRÉATION DU JAR

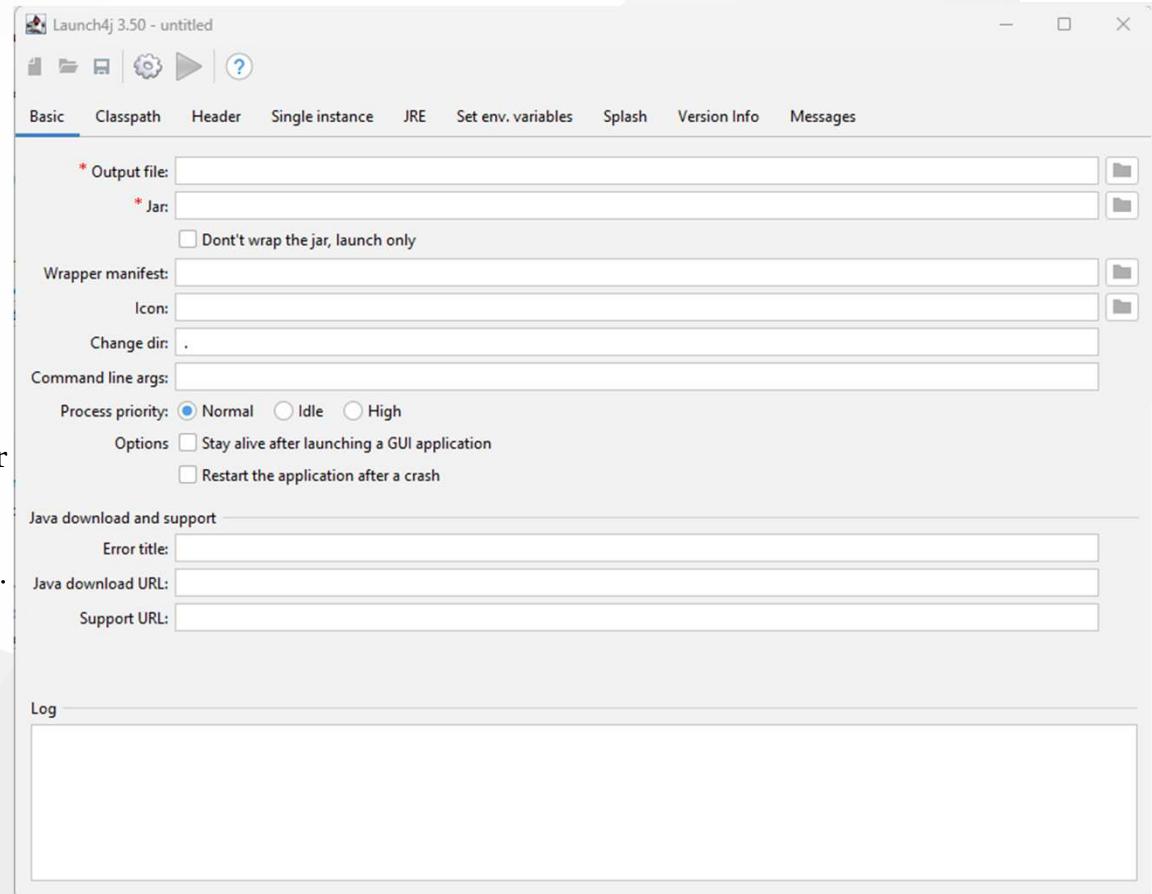
1. Choisir la classe Java de votre point d'entrée de votre application
2. Sélectionner le répertoire de destination du futur fichier .jar
3. Cocher la case **Extract required libraries into generated JAR**
4. Finish



INSTALLATION DU WRAPPER

<https://launch4j.sourceforge.net/>

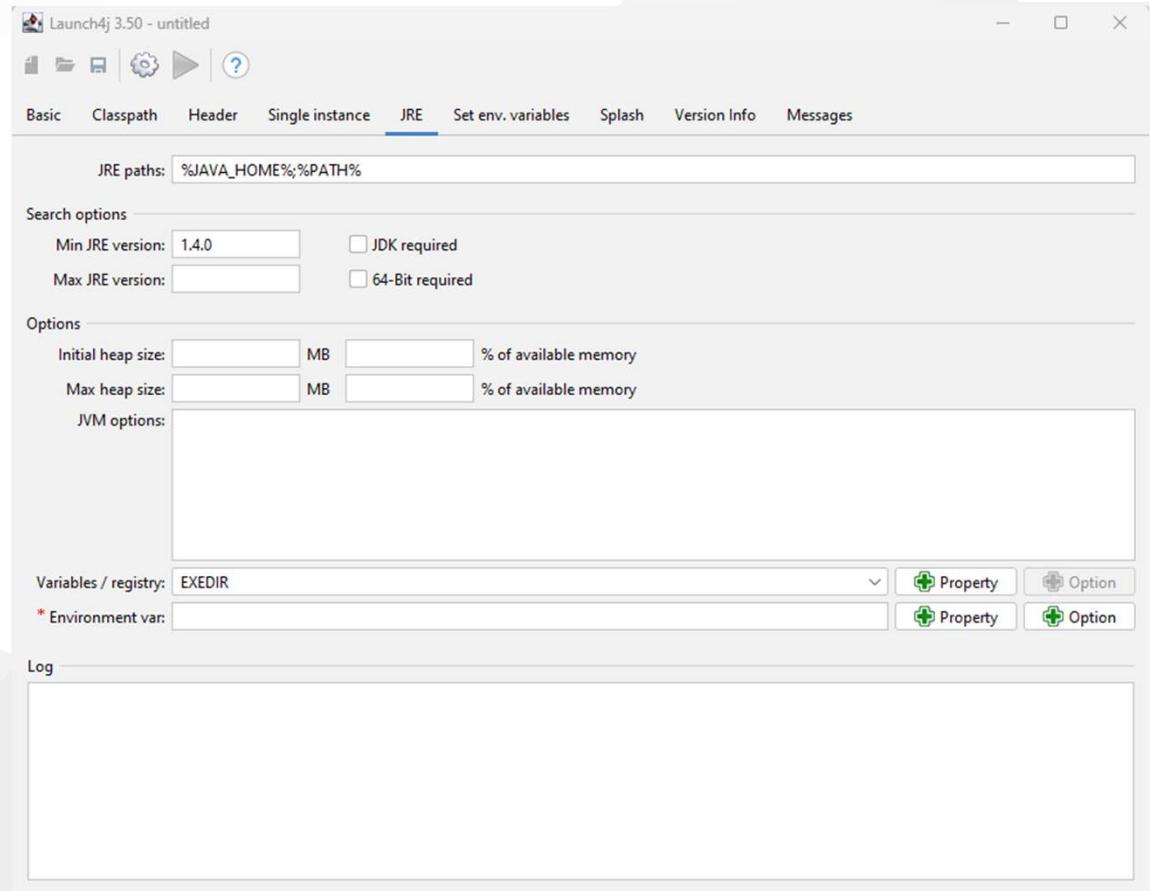
- Télécharger l'application pour permettre la création d'un .exe
1. Indiquer le nom de l'exécutable (ne pas oublier le .exe)
 2. Sélectionner le fichier .jar précédemment créé.
 3. Aller dans l'onglet JRE



ONGLET JRE

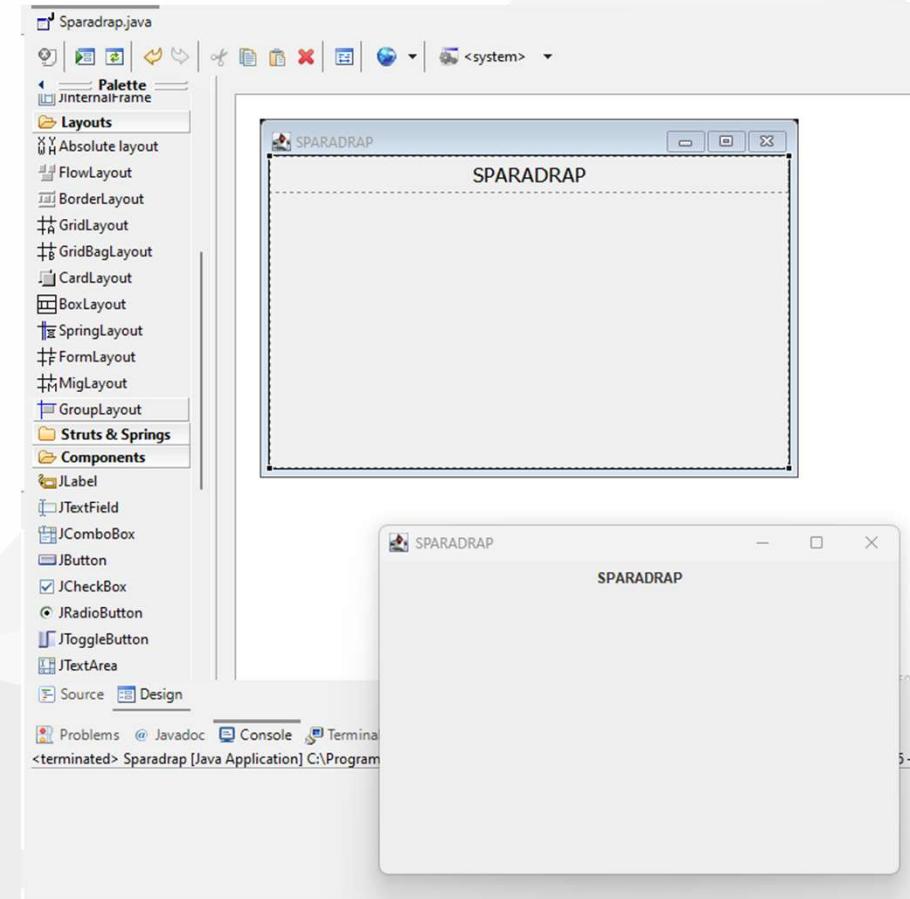
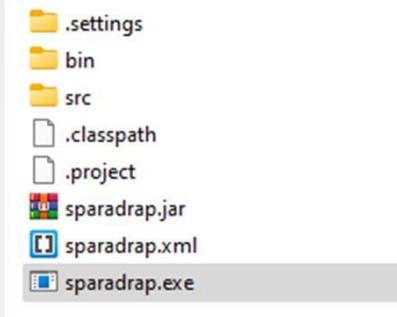
1. **JRE paths** se complète automatiquement
2. Indiquer **Min JRE version** : 1.4.0 
3. Cliquer sur l'icone Build Wrapper
4. Indiquer l'emplacement du fichier .xml

Dans le log, il indique la création du .exe



EXÉCUTION

Désormais, vous pouvez exécuter votre application



ANCIENS EXEMPLES

EXEMPLE

Création d'un formatter, d'un filter et d'un Handler

```
public class MyHandler extends StreamHandler {  
  
    @Override  
    public void publish(LogRecord record) {  
        //add own logic to publish  
        super.publish(record);  
    }  
  
    @Override  
    public void flush() {  
        super.flush();  
    }  
  
    @Override  
    public void close() throws SecurityException {  
        super.close();  
    }  
}
```

```
public class MyFormatter extends Formatter {}  
  
    @Override  
    public String format(LogRecord record) {  
        return record.getLongThreadID()+"::"+record.getSourceClassName()+"::"  
            +record.getSourceMethodName()+"::"  
            +new Date(record.getMillis())+"::"  
            +record.getMessage()+"\n";  
    }  
}
```

```
public class MyFilter implements Filter {  
  
    @Override  
    public boolean isLoggable(LogRecord log) {  
        //don't log CONFIG logs in file  
        if(log.getLevel() == Level.CONFIG) return false;  
        return true;  
    }  
}
```

EXEMPLE

Le programme principal

```
public class LoggingExample {  
    static Logger logger = Logger.getLogger(LoggingExample.class.getName());  
  
    public static void main(String[] args) {  
        try {  
            LogManager.getLogManager().readConfiguration(new FileInputStream("mylogging.properties"));  
        } catch (SecurityException | IOException e1) {  
            e1.printStackTrace();  
        }  
        logger.setLevel(Level.FINE);  
        logger.addHandler(new ConsoleHandler());  
        //adding custom handler  
        logger.addHandler(new MyHandler());  
        try {  
            //FileHandler file name with max size and number of log files limit  
            Handler fileHandler = new FileHandler("./logs/logger.log", 2000, 5);  
            fileHandler.setFormatter(new MyFormatter());  
            //setting custom filter for FileHandler  
            fileHandler.setFilter(new MyFilter());  
            logger.addHandler(fileHandler);  
  
            for(int i=0; i<1000; i++){  
                //logging messages  
                logger.log(Level.INFO, "Msg"+i);  
            }  
            logger.log(Level.CONFIG, "Config data");  
        } catch (SecurityException | IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

EXEMPLE

Création d'un fichier properties qui va contenir la configuration de log4J.

Il faut créer un fichier `log4j2.properties` stocké dans le classpath de l'application.

Vous pouvez dès lors créer plusieurs fichiers de configuration suivant vos différents environnements de développement / production par exemple.

```
2023-11-03 13:30:57 [ERROR] (Log4JExemple.java:main:18) [] Houston, we have a problem  
2023-11-03 13:30:57 [ERROR] (Log4JExemple.java:main:21) [] Hello World with Log4J 2  
2023-11-03 13:30:57 [ERROR] (Log4JExemple.java:main:28) [] Houston, we have a problem.  
java.lang.ArithmetricException: / by zero  
at fr.afpa.pompey.cda22045.app.Log4JExemple.main(Log4JExemple.java:25) ~[bin/:?]
```

```
# Set to debug or trace if log4j initialization is failing  
status = warn  
  
# Name of the configuration  
name = ConfigDemo  
  
# appenders = console  
appender.console.type = Console  
appender.console.name = LogToConsole  
appender.console.layout.type = PatternLayout  
appender.console.layout.pattern = %d{yyyy-MM-dd HH:mm:ss} [%-5p] (%F:%M:%L) %x %m%n%n  
  
# appenders = File  
appender.file.type = File  
appender.file.name = LogToFile  
appender.file.fileName= Logs/myLog.log  
appender.MaxFileSize= 1MB  
appender.file.layout.type=PatternLayout  
appender.file.layout.pattern=%d{yyyy-MM-dd HH:mm:ss} [%-5p] (%F:%M:%L) %x %m%n%n  
  
# Root logger level  
rootLogger.level = error  
  
# rootLogger.appendRef.stdout.ref = LogToConsole  
#rootLogger.appendRef.stdout.ref = LogToConsole  
rootLogger.appendRef.stdout.ref = LogToFile
```