Java (les bases et la POO)

Mes notes pour le cours de programmation de base et de programmation orientée objet

    - Comment concevoir en orienté objet

- Programmation orientée objet (B2)
    - Introduction
    - Exceptions
    - Moteur de production (gradle)
    - Dernière fonctionalités utiles du JDK 17
    - Doublures de test
    - Logging

Les bases du language

Une "cheatsheet" sur la syntaxe du language, et toutes les bases. Ce chapitre ne traite pas l'orienté objet.

Introduction

Ceci sont mes notes de Java. J'ai essayé de les écrire sous le même format que Rust by example.

Vous pouvez trouver mes solutions et celle d'autres personnes sur mon Git :

- codeberg.org/SnowCode/prb

Pour sélectionner les personnes ou différentes versions du code, changer la "branche" Git.

Bonne lecture, amusez vous bien :)

Hello World

Pour le cours de Java on va utiliser OpenJDK 18. On peut l'installer depuis le site de java OpenJDK. JDK veut dire "Java Development Kit" et openJDK est une implémentation de cela en open source. On peut aussi installer un éditeur de code (exemple: Notepad++, Atom, Visual Codium).

Une fois cela fait, on peut créer un dossier pour notre projet "premier-projet" dans lequel on va mettre un autre dossier "hello". Dans ce dossier on va créer un fichier `Hello.java`. Attention que les noms, doivent être exacts à l'exception du premier dossier "premier-projet".

Dans le fichier Hello.java, on peut ajouter le code suivant

```
// Le package contient les classes, cette ligne indique que le fichier fait partie du package "prb"
// Le package doit être du même nom que le dossier dans lequel le fichier est
package hello;

// La classe doit être du même nom que le fichier, dans ce cas ci la classe "Bonjour" doit être dans un dossier appellé "Bonjour.java"
// Tout le code en java est écrit dans des classes. La mention "public" signifie que cette classe peut être réutilisée par d'autres classes ailleurs
public class Hello {
    // static veut dire que ce n'est pas un "objet" de la classe mais est associée. Il ne faut pas créer une instance de la classe pour appeller cette fonction/méthode
    // void signifie qu'il n'y a pas de valeur "return" à la méthode.
    // la méthode main est la fonction recherchée par java quand la classe est lançée. C'est ici qu'est le programme principal
    public static void main(String[] args) {
        // System est une classe par défault de Java
        // println est une méthode (ou fonction) du bloc "out" de la classe "System"
        System.out.println("Bonjour !");
    }
}
```

On peut enfin lançer ce programme en ouvrant l'invite de commande dans le dossier `premier-java` et en lançant

```
javac hello\Hello.java
java hello.Hello
```

La commande javac va compiler un fichier .class. Java est un language hybride, il compile le code dans un language intermédiaire (le bytecode) prévu pour être exécuté par la JVM (Java Virtual Machine) pour pouvoir fonctionner sur toutes les plateformes car seul la JVM est différente sur chaque plateforme. Le slogan de Java "Write once, run everywhere" viens de là.

Variables (types, constantes, cast et String)

Les types primitifs

```
/* Code à intégrer dans la fonction main d'une classe */

// On peut déclarer une variable avant de l'initier
int entier;
entier = 42;

// Ou on peut faire les deux en même temps
boolean test = true;
char lettre = 'A'; // Attention, le ' est nécessaire et ne peut pas être remplacé par "
byte octet = 127
double nombreAVirgule = 5.2;

// On peut aussi créer une constante, sa valeur ne pourra pas changer
final int reponseALaVie = 42;
```

Il y a 8 types primitifs en Java:

Type	Taille en bits	Domaine de valeurs	Information supplémentaire
`boolean`	?	0 (false), 1 (true)	La taille dépends de la JVM
`char`	16	N'importe quel caractère unicode	N/A
`byte`	8	$ -128 $ → $ +127 $	N/A
`short`	16	$ -2^{15} $ → $ 2^15 - 1 $	N/A
`int`	32	$ -2^{31} $ → $ 2^{31} - 1 $	N/A
`long`	64	$ -2^{63} $ → $ 2^{63} -1 $	N/A

Type	Taille en bits	Domaine de valeurs	Information supplémentaire
`float`	16	$ -3.4 * 10^{38} $ → $ 3.4 * 10^{38} $	Nombre à virgule (virgule flotante). La précision est de $ 1.4 * 10^{-45} $
`double`	64	$ -1.7 * 10^{308} $ → $ 1.7 * 10^{308} $	Comme `float` mais avec une précision de $ 4.9 * 10^{-324} $

Convertir entre les types avec cast

```
// Conversion de float en int avec cast
double x = 42.5;
int y = (int)x; // 42
int z = (int)42.5; // 42
int a = (int)(42.0 + 0.5); // 42
```

On peut aussi transformer une valeur en une autre en utilisant un *cast*. Comme vu ci dessus en précisant le type cible en ().

A savoir qu'ici, dans le cas d'une conversion d'un nombre en virgule flotante en entier, on perds les informations des décimales. Cast va toujours arrondir vers le bas dans une conversion comme celle ci. Pour avoir plus de controle sur l'arrondis, on peut utiliser la classe java Math (voir plus tard).

Introduction aux Strings (chaine de caractères)

```
String maChaine = "Hello, World!";
System.out.println(maChaine);


// Convertir un string en un int (cast est impossible car String est une classe)
String nombreStr = "42"; // "42"
int nombreInt = Integer.parseInt(nombreStr); // 42
```

String n'est pas un type primitif, mais bien une classe (ce qui est pourquoi la première lettre est en majuscule) qui corresponds à une chaine de caractère, dans un chapitre suivant nous allons voir quelques méthodes qui peuvent être appliquer aux objets de la classe "String".

N'étant pas un type primitif nous ne pouvons pas utiliser cast dessus mais on peut utiliser d'autres méthodes d'autres classes comme `Integer.parseInt()`, il existe aussi d'autres méthodes du même

genre comme `Double.parseDouble()`.

En savoir plus

- Wikiversity FR - Java: Variables et types
- Oracle Docs - String
- Oracle Docs - Integer.parseInt

Opérations sur les variables (calculs, logiques, binaire)

```
int a = 12;
int b = 20;
int c = 42;

// Opérateurs de calcul
int x = (a + b + 2) / c; // Le résultat sera tronqué car x est un int et non pas un double, donc les décimales ne seront pas prise en compte
int y = c % 3; // Effectue de reste d'une division euclidienne

// Opérateurs lors de l'assignation d'une variable
a = a + 5 // Ajoute 5 à la variable a
a += 5; // Ajoute 5 à la variable a
b++; // Ajoute 1 à la variable b

// Opérateurs logiques / binaires
boolean a = true && false; // false
boolean b = true || false; // true
boolean c = !false; // true

// Ou exclusif, 1 et 1 donne 0 mais 1 et 0 donne 1
boolean d = true ^ true; // false
boolean e = true ^ false; // true
```

Il existe 5 opérateurs de base en Java:

- `+` : addition
- `-` : soustraction
- `*` : multiplication
- `/` : division
- `%` : modulo (reste d'une division euclidienne, exemple $ 13 = 5 * 2 + 3 $ donc $ 13 \mod 5 = 3 $

Il existe ensuite plusieurs opérateurs logiques et binaires, mais on va se concentrer sur les principaux uniquement

- `&&` : ET (true + true = true)
- `||` : OU inclusif (true + false = true) et (true + true = true)
- `^` : OU exclusif (true + false = true) et (true + true = false)
- `!` : NEGATION (true = false) et (false = true)

On peut ensuite effectuer des modifications lors de l'affectation de la variable directement pour gagner du temps en ajoutant l'opérateur devant le signe `=`

On peut aussi ajouter 1 ou diminuer un rapidement en écrivant `x++` ou `x--` par exemple.

En savoir plus

- Wikiversity FR - Java: Opérations

Acquisition des données de l'utilisateur (Console)

Créer un dossier `io` dans `premier-java` et y placer le fichier `Console.java` donné dans la page du cours. Il est possible qu'il soit nécessaire de changer le package du fichier vers `io` si ce n'est pas déjà le cas. Ensuite dans notre Hello.java (ou autre).

```
// io (package) .Console (classe) .lireInt() (méthode)
int variableInput = io.Console.lireInt();
System.out.println(variableInput);
```

La classe Console contient plusieurs méthodes: `lireString`, `lireChar`, `lireInt`, `lireLong`, `lireFloat`, `lireDouble` pour récupérer des informations depuis une saisie de l'utilisateur dans le terminal.

Cette classe Console est une simplification du code Java de base pour aller plus vite.

Pour savoir comment la saisie fonctionne dans le java de base, il suffit de lire le code de Console.java.

En savoir plus

- Voir le fichier Console.java

Affichage formatté (print, println, printf)

```
System.out.print("Quel est votre nom ? ");
String nom = io.Console.lireString();
System.out.println("Hello World");
System.out.printf("Hello %s", nom);

// Intégrer un nombre décimal
int age = 17;
System.out.printf("Hello %s, you are %d", name, age);

// Intégrer un nombre à virgule flotante
double temperature = 16.5;
System.out.printf("The temperature today is %.2f\n", temperature); // The temperature today is 16,50
```

Il existe 3 fontions principales de PrintStream:

- `print` qui va afficher une valeur sans retour à la ligne
- `println` qui va afficher une valeur avec un retour à la ligne automatique (ce qui est comme ajouter \n à la fin de la valeur dans print)
- `printf` qui va permettre de faire un "template"` pour afficher des valeurs.

Printf prends une grande variété de conversions, en voici quelques basiques :

- `%c` pour afficher un `char`
- `%s` pour afficher un `String`
- `%d` pour afficher un nombre décimal (`byte`, `short`, `int`, `long`)
- `%f` pour afficher un nombre à virgule flotante (`float`, `double`)
- `%%` pour afficher un '%'
- `%n` pour afficher un retour à la ligne

La syntaxe est `%[longueur][conversion]` par exemple, `%.5s` va afficher un `String` d'une longueur de 5 caractères. Si le `String` est moins long, `printf` va remplacer l'espace manquant par des espaces, si elle est trop courte, la valeur va être coupée.

En savoir plus

- Oracle Docs - Formatter pour plus d'information sur les conversions avec printf
- Oracle Docs - PrintStream pour voir les différentes fonction de System.out.

Tableaux

```
// Création d'un tableau d'entiers vide d'une longueur de 5 éléments
int[] puissancesDeux = new int[5];
puissancesDeux[0] = 1;
puissancesDeux[1] = 2;
puissancesDeux[2] = 4;
puissancesDeux[3] = 8;
puissancesDeux[4] = 16;
System.out.println(puissanceDeux[4]); // 16

// Création d'un tableau de Strings avec des valeurs prédéfinies
String[] autreTableau = { "un", "deux", "trois", "quatre", "cinq" };
System.out.println(autreTableau[0]); // "un"

// Création d'un tableau de tableau
String[][] tableauDeTableau = { { "un", "deux", "trois" }, { "quatre", "cinq", "six" } };
System.out.println(tableauDeTableau[1][0];

// Connaitre la longueur d'un tableau
System.out.println(tableauDeTableau.length);

// Ajouter un nouvel élément dans un tableau après qu'il soit défini
String[] nouveauTableau = new String[autreTableau.length + 1];
for (int i = 0; i < autreTableau.length; i++) {
nouveauTableau[i] = autreTableau[i];
}

nouveauTableau[autreTableau.length] = "six";
```

Les tableaux en Java ont déjà une longueur définie, ce qui fait qu'ajouter des nouveaux éléments est plus compliqué. Il faut créer un nouveau tableau avec une longueur plus longue et créer un for loop pour ajouter chaque élément dans le nouveau tableau + l'élement à ajouter.

Les indices (le nombre qui est dans les crochets) qui permet de référencer un élément dans le tableau, commencent par 0. Donc le premier élément de la liste est l'indice 0, le deuxième 1, le troisième 2, etc.

On peut connaitre la longueur d'un tableau en utilisant ".length" sur un tableau.

En savoir plus

- Wikiversity FR
- Le powerpoint de Mr Comblin sur les Tableaux

Manipulations de Strings

```
String inputString = "ceci est mon string, 32";
String[] tab = inputString.split(", ");
String str = tab[0].toUpperCase();
int number = Integer.parseInt(tab[1]);
System.out.printf("%s =  %d %n", str, number);

// Tester des strings
boolean finiParTest = inputString.endsWith("test"); // false
boolean commenceParCeci = inputString.startsWith("ceci"); // true

// Avoir la longueur d'un string
System.out.println(inputString.length()); // 23

// Utiliser substring pour diviser un string
String hhmm = "10:30";
System.out.println(hhmm.substring(0, hhmm.length() - 3)); // 10
System.out.println(hhmm.substring(hhmm.length() - 2, hhmm.length())); // 30

// Trouver la position d'un caractère dans un String
int position = hhmm.indexOf(':'); // 2

// Retrouver le caractère à un certain indice de position
char caractere = hhmm.charAt(position); // ':'
System.out.printf("Le caractère à %d est '%c'", position, caractere); // "Le caractère en position 2 est ':'"

// Transformer un int en String
int nombre = 42;
String nombreEnString = String.valueOf(nombre); // "42"
```

String étant une classe, elle contient plusieurs méthodes, en voici 2 qui soit très utiles :

- `.split("regex")` pour diviser un String en un tableau en utilisant un autre String comme délimiteur de la colonne.
- `.toUpperCase()` et `.toLowerCase()` pour convertir un String en majuscule ou minuscule

- `.endsWith()` et `.startsWith()` pour tester si un String termine ou commence par un certain substring.
- `.length()` pour obtenir la longeur de la chaine de caractères.
- `.substring()` pour diviser un string avec 2 positions. Cette méthode prends 2 arguments, la première position (qui est inclusive, donc prise en compte dans le résultat), et la deuxième position qui est exclusive (donc non prise en compte dans le résultat). Donc si on a 0 en premier argument et 5 en deuxième argument, on aura les caractères 1, 2, 3 et 4.
- `.indexOf()` permet de savoir quel est l'indice de position d'un caractère ou un String dans un String.
- `.charAt()` est l'inverse de `indexOf` et retourne un caractère à une position donnée
- `.valueOf()` premet de transformer un autre type en String.

> A noter que l'utilisation des méthodes String indiquées ici ne transforme pas la variable String d'origine, elle ne font que retourner une nouvelle valeur.

En savoir plus

- Oracle Docs - String

Un peu de Math

```
// Génération de nombres aléatoires
double nombreAleatoire = Math.random();

// Puissances et racines
double puissance2 = Math.pow(5.0, 2.0); // 5^2
double racineCarree = Math.sqrt(puissance2);

// Arrondir un nombre
double nombre = 5.67;
System.out.println(Math.ceil(nombre));  // 6.0
System.out.println(Math.floor(nombre)); // 5.0
System.out.println(Math.round(nombre)); // 6
System.out.println(Math.rint(nombre));  // 6.0
```

Pour générer une valeur aléatoire entre 0 et 1 on peut utiliser la méthode `Math.random()`.

Pour faire une puissance de 2 on peut utiliser `Math.pow(double a)` et pour faire une racine `Math.sqrt(double a)`. Toutes ces fonctions renvoie des valeurs de type `double`

Pour faire des arrondis on peut utiliser différentes fonctions :

- `Math.ceil(nombre)` pour arrondir un nombre vers le haut (retourne un double)
- `Math.floor(nombre)` pour arrondir un nombre vers le bas (retourne un double)
- `Math.round(nombre)` pour arrondir un nombre en fonction de sa première décimale (retourne un int)
- `Math.rint(nombre)` pour arrondir un nombre en fonction de sa première décimale (retourne un double)

Pour en savoir plus

- Oracle Docs - Math

Conditions (if, else, switch)

```
System.out.print("Sélectionnez un nombre: ");
int first = io.Console.lireInt();

if (first == 42) {
    System.out.println("Félicitations, vous avez trouvé la réponse à la vie");
} else if (first > 40 && first < 50) {
    System.out.println("Vous y êtes presque");
} else {
    System.out.println("Vous n'avez rien trouvé du tout");
}

// Comprarer des Strings ou autres objets ne se fait pas de la même manière
System.out.print("Quel est votre nom ? ");
String nom = io.Console.lireString();
if (nom.equals("Roger")) {
    // faire quelque chose
}

// Donne le jour en texte à partir d'un numéro, cas par cas
int jour = 4;
switch (jour) {
  case 1:
    System.out.println("Lundi");
    break;
  case 2:
    System.out.println("Mardi");
    break;
  case 3:
    System.out.println("Mercredi");
    break;
  case 4:
    System.out.println("Jeudi");
    break;
```

```
  case 5:
    System.out.println("Vendredi");
    break;
  case 6:
    System.out.println("Samedi");
    break;
  case 7:
    System.out.println("Dimanche");
    break;
} // "Jeudi"

switch (jour) {
  case 6:
    System.out.println("Samedi");
    break;
  case 7:
    System.out.println("Dimanche");
    break;
  default:
    System.out.println("En attente du weekend");
} // "En attente du weekend"
```

Il y a deux types de conditions en java, les `if else` et les `switch`.

If, Else if, Else

Les `if` permettent de mettre une condition pour l'exécution d'un code.

Il existe plusieurs mots clés comme vu dans l'exemple du début

- `if (...) {` : si *condition* alors ...
- `else if (...) {` : si la contion précédente du bloc `if` n'a pas réussi alors si *condition*, ...
- `else` : si aucune des conditions précédentes du bloc `if` n'a réussi alors ...

Ces conditions fonctionnent en utilisant un "opérateur conditionnel" entre deux nombres. En voici un exemple :

Opérateur contionnel	Description
`a == b`	A est égal à B
`a != b`	A est différent de B

Opérateur contionnel	Description
`a > b`	A est plus grand que B
`a < b`	A est plus petit que B
`a >= b`	A est plus grand ou égal à B
`a <= b`	A est plus petit ou égal à B
`a`	A est vrai
`!a`	A est faux (en vérité ceci est un opérateur binaire et non un opérateur conditionnel)

On peut aussi mettre plusieurs conditions ensemble avec des opérateurs logiques (ce sont les même qu'au cours de math et d'archi) :

Opérateur logique	Description
`&&`	Les deux conditions doivent être remplies
`||`	Au moins une des deux conditions doit être remplie
`^`	Une seule des deux conditions doit être remplie mais pas plus

Cas spéciaux avec les Strings

On ne peut pas utiliser un opérateur tel que `==` sur un String car un String est un objet de la classe String. Si on utilise l'opérateur logique, on va comparer la référence mémoire de la variable et non la valeur en elle même.

Il faut donc utiliser la méthode `.equals(String autreString)` à la place comme vu dans l'exemple.

switch

Si on veut limiter l'usage des `if else` et que l'on veut tester si une valeur correspond à x, y ou z valeur, on peut utiliser `switch`

On donne à switch la valeur à tester et plusieurs *cas* (`case`) de valeur possible. Pour chacune d'entre elle, quelque chose à faire avec.

On peut utiliser le mot clé `break` pour arrêter le switch sans tester le reste.

Enfin il y a le `default` qui fonctionne comme le `else`, le default, c'est quand aucun autre cas n'est passé.

En savoir plus

- Wikiversity FR - Boucle et structures conditionnelles
- Wikiversity FR - Opérations

Méthodes static

```
public class Bonjour {
    public static void main(String[] args) {
        System.out.print("Quel est ton nom ? ");
        String nom = Console.lireString();
        direBonjour(nom);

        int premierNombre = 40;
        int deuxiemeNombre = 2;
        int troisiemeNombre = somme(premierNombre, deuxiemeNombre); // 42
    }

    public static void direBonjour(String nom) {
        System.out.printf("Bonjour %s !\n", nom);
    }

    private static int somme(int a, int b) {
        int c = a + b;
        return c;
    }
}
```

Une méthode (ou fonction) est un bloc de code qui peut être réutiliser plusieurs fois. En fonction de comment elle est définie, une méthode peut renvoyer une certaine valeur ou rien, et peut prendre des paramètres ou pas.

Pour mieux comprendre voici comment on peut décomposer notre méthode `direBonjour` au dessus :

- `public` indique l'accessibilité. Elle signifie que d'autres classes peuvent aussi utiliser cette méthode. Au contraire, si cela serait `private` alors seul la classe `Bonjour` pourrait y accéder
- `static` indique que ce n'est pas un "objet" de la classe `Bonjour`, on vera plus tard ce que cela signifie
- `void` signifie que la méthode ne retourne aucune valeur
- `direBonjour` est le nom de notre méthode
- `(String nom)` indique que la méthode ne prends qu'un argument de type String, qui va être appellé "nom" dans le bloc de la méthode.

Dans le deuxième exemple :

- `private` indique que la méthode ne peut être utilisée que dans la classe `Bonjour`
- `static` même chose que pour `direBonjour`
- `int` indique que la méthode retourne un type "int"
- `somme` est le nom de la méthode
- `(int a, int b)` indique que la méthode prends 2 arguments, tout deux de type "int" qui seront appellé dans la méthode "a" et "b".
- `return` indique une valeur à retourner de la fonction. Une fois que la fonction a retourné quelque chose, tout code écrit après ce `return` ne sera pas exécuté.

Mais il y a aussi la fonction main ! On a l'a tout le temps utilisé mais c'est aussi une méthode. La seule différence est que quand on exécute une classe, Java va chercher cette fonction "main" dans notre classe pour l'exécuter. Si il n'y a pas de fonction main, la classe n'est pas exécutable à elle seule.

Plusieurs méthodes avec le même nom

```
public static void main(String[] args) {
    System.out.printf("12 + 15 = %d%n", somme(12, 15));
    System.out.printf("12.5 + 13.0 = %f%n", somme(12.5, 13.0));
}

public static int somme(int a, int b) {
    return a + b;
}

public static double somme(double a, double b) {
    return a + b;
}
```

Dans java, contrairement à d'autres language il peut y avoir plusieurs méthodes qui ont le même nom tant qu'elle ne prennent pas les mêmes arguments (paramètres).

Dans ce cas ci, on a deux variables "somme" mais une prends des types "int" et l'autre prends des types "double". Donc elles peuvent toutes les deux exister dans la même classe sans problème.

Apeller une méthode

Comme vu dans les exemples ci dessus, on peut appeller une méthode quand on est dans la même classe qu'elle. Mais on peut aussi appeller une méthode qui est dans une autre classe.

```
public static void main(String[] args) {
    int input = io.Console.lireInt();
}
```

Dans l'exemple ici on appelle la méthode `lireInt` qui est dans la classe `Console` du package `io` par exemple.

En savoir plus

- Wikiversity FR - Méthodes

Boucles (for, for each, while, do...while)

Pour ne pas avoir besoin de repeter un code beaucoup de fois, on peut utiliser des boucles. Il en existe 4 différentes.

- `while` qui exécute un code en boucle tant qu'une certaine condition est remplie
- `do...while` qui exécute une fois le code, puis la répète encore tant que la condition est remplie
- `for` qui permet d'exécuter un code un nombre définis de fois
- `for each` qui permet d'exécuter un code pour chaque élément dans une collection (par exemple un tableau comme vu dans un chapitre précédent)

La boucle while

```
// Compter jusqu'a 42
int i = 0;
while (i <= 42) {
    System.out.printf("Le compteur i est à %d.\n", i);
    i++;
}

// Ne va rien faire car la condition n'est pas remplie
int j = 0;
while (i == 666) {
    System.out.printf("Le compteur j est à %d.\n", j);
    j++;
}
```

Le premier bloc de code va s'exécuter et on verra 43 lignes allant de 0 à 42. Le deuxième bloc de code ne s'exécutera pas car sa condition que i doit être égal à 666 ne sera pas remplie.

La boucle do...while

```
// Compter jusqu'a 42
do {
    System.out.printf("Le compteur i est à %d.\n", i);
    i++;
} while (i <= 42);

// Ne va faire qu'une seule itération car la condition ne sera pas remplie
int j = 0;
do {
    System.out.printf("Le compteur j est à %d.\n", j);
    j++;
} while (j == 666);
```

Dans ce cas ci, comme avant, le premier bloc de code s'exécutera 43 fois de 0 à 42. Tandis que le second bloc de code ne s'éxécutera qu'une fois et indiquera "0".

Contrairement à while, la boucle do...while s'exécutera toujours un minimum de une fois. Puis se réexécutera si/tant que la condition est remplie.

La boucle for

```
// Compter jusqu'a 42
for (int i = 0; i <= 42; i++) {
    System.out.printf("Le compteur est à %d.\n", i);
}
```

Ce code indiquera 43 lignes allant de 0 à 42. C'est comme une version contractée de la boucle while que nous avons déjà vu plus tot.

La différence est que le nombre d'itération est déjà prédéfinis. On définis une boucle for comme ceci :

```
for (initialisation; condition; modification) {
    // code à exécuter dans la boucle
}
```

La boucle for each

```
// Lire chaque élément d'un tableau
String[] tableau = { "un", "deux", "trois", "quatre", "cinq", "six", "sept", "huit", "neuf", "dix" };

for (String element: tableau) {
    System.out.println(element);
}
```

La boucle for each permet de passer en revue tous les élément d'une *collection*, par exemple dans ce cas ci, d'un tableau (voir dans un chapitre précédent). Dans ce code on assigne pour chaque itération la variable "element" qui correspond à l'élément en cours dans la variable "tableau".

En savoir plus

- Wikiversity FR - Boucles et structures conditionnelles

Le RegEx

Voici un petit résumé de la signification des différents caractères :

Element en regex	Signification
`\`	Indique que le caractère qui suit est litéral et qu'il ne faut pas qu'il soit interpreté comme synatxe du regex
`^`	Début de la chaine de caractères
`$`	Fin de la chaine de caractères
`*`	Match le caractère précédent 0 fois ou plus
`+`	Match le caractère précédent 1 fois ou plus
`?`	Match le caractère précédent 0 ou 1 fois
`.`	Match n'importe quel caractère
`[abc]`	Match n'importe lequel des caractère dans les crochets
`[A-Z]`	Match n'importe quel caractère dans une série (ici allant de A à Z majuscule)

Les Regex peuvent être appliqué dans diverses fonctions de String et Pattern

```
class Test {
    public static void main(String[] args) {
        String monString = "Hello World";
        // String.matches() peut être utiliser pour vérifier si un String correspond à une certaine expression
        if (monString.matches("[A-z]+ [A-z]+")) {
            System.out.println(monString);
        }
    }
}
```

D'autres méthodes peuvent aussi utiliser des regex, tel que la méthode `String.replaceAll` vue dans le chapitre sur les Strings.

Tests unitaires avec jUnit 5

Les tests unitaires permettent d'avoir une vue globale de la santé d'un projet en s'assurant que toute ses fonctions se comportent comme elle doivent.

Dans Eclipse il faut reproduire la structure demandée. Le code source est dans le dossier `src`, tandis que les tests sont dans le dossier `tests`. La structure des packages et des classes est conservée. A l'exception des nom de la classe de test qui finissent par `Tests`.

```
.
├── src
│   └── util
│       └── TableauChaines.java
└── tests
    └── util
        └── TableauChainesTests.java
```

Dans cet exemple on va tester une fonction de `TableauChaines.java`

```
// Il faut indiquer le même package pour le test que ce que l'on test
package util;

// Il faut importer Assert et Test de junit
import static org.junit.Assert.*;
import org.junit.jupiter.api.Test;

// Les fichiers de test finissent par "Tests.java" ainsi leur classe fini par "Tests" aussi
class TableauChainesTests {
    // Chaque test est une fonction void sans argument précédée de @Test
    @Test
    public void contientTest() {
        String[] tableau = {"PRENOM", "SEXE", "LONGUEUR DES CHEVEUX", "LUNETTES"};

        // Ce qui vérifie les tests sont les assertions
        // Il y en a de différent types (assertEquals, assertArrayEquals, assertTrue, etc) dépendant de ce que l'on
test
        // Le premier paramètre est le résultat attendu, le deuxième est le résultat obtenu à partir de la méthode
```

```
que l'on test
        assertTrue(TableauChaines.contient(tableau, "Longueur des cheveux"));
    }
}
```

On peut ensuite exécuter le test en lançant le projet dans Eclipse. Et c'est tout !

Génération de la documentation avec javadoc

La Javadoc permet d'écrire la documentation des méthodes du programme directement dans le code. Un programme va ensuite générer un site pour afficher toute cette documentation.

```
package labo6;


class JourDeLaSemaine {
    /**
     * Adapte la longueur d'une chaîne de caractères en la complétant par des
     * caractères espace ou en réduisant son nombre de caractères.
     *
     * @param chaine  la chaîne de caractères à ajuster.
     * @param largeur La largeur que doit avoir la chaîne de caractères spécifiée.
     * @return La chaîne de caractères ajustée.
     */
    public static String ajusterLargeur(String chaine, int largeur) {
        final int LG_CHAINE = chaine.length();
        chaine = (LG_CHAINE > largeur) ? chaine.substring(0, largeur) : chaine;
        return chaine + " ".repeat(largeur - chaine.length());
    }
}
```

Dans l'exemple ci dessus, on a une méthode "ajusterLargeur" que l'on veut documenter. Pour se faire il suffit d'écrire `/**` puis enter dans Eclipse pour que le template soit généré tout seul.

Dans la première partie on écrit la description de la méthode, puis on décrit à quoi correspondent chaque paramètre et ce que la méthode retourne.

Pour générer la Javadoc par la suite on peut aller sur Eclipse dans le menu "Project" puis "Generate javadoc". La javadoc sera ainsi générée dans le dossier `doc` du projet.

Et voici le résultat pour notre méthode :

screenshot de la javadoc pour la méthode

La programmation orientée objet

Programmation orientée objet

> ⚠ **Attention cette page est en cours de construction**

> La programmation orientée objet (POO) est un *paradigme* de programmation, c'est à dire une manière de programmer. Ce que l'on faisait précédemment est appellé la *programmation fonctionnelle*

L'un des pouvoir de la POO est nottament de pouvoir créer ses propres types appellées "classes", on les reconnais en Java car elle commencent avec une lettre majuscule (par exemple `String` est une classe mais `int` n'en est pas une)

Une classe est donc comme un nouveau type et est défini par des propriétés ainsi que des méthodes qui lui sont propre.

Un objet est une instance de cette classe (c'est une occurence de notre nouveau type).

```
// Ici on crée un "objet" de la classe "String"
String hello_string = "Hello World"

// .length() est une méthode de String qui permet de compter le nombre de caractère de la chaine
hello_string.length();

// println est une méthode statique qui n'a pas besoin d'un objet pour être exécutée
System.out.println(hello_string);
```

Voici comment créer une nouvelle classe :

```
// On crée donc la classe "Color".
// Cette définition de lcasse n'est pas précédée par "public"
// Donc seul les classes qui sont dans le même package peuvent l'utiliser
```

```
class Color {
  // On va définir un attribut de classe (qui sont des constantes)
  // public signifie qu'elle peut être utilisée de n'importe où
  // static signifie que c'est un attribut de classe (et pas d'objet) donc qui ne nécessite pas d'objet pour être accédée
  // final  signifie que c'est une constante
  // Par définition toutes les attributs de classe seront static final
  // Et seront des constantes car sinon cela va affecter TOUS les objets qui utiliseraient cette classe
  public static final float TOLERANCE = 0.001f;

  // On défini les attributs d'objet "r", "g" et "b"
  // Cela va définir l'état des objets de cette classe
  // En d'autres termes ça sera l'identité des objets,
  // ce qui les différenties des autre objets de la même classe
  // Le modificateur "private" signifie que l'on peut y accéder directement uniquement au sein de la classe
  private int r;
  private int g;
  private int b;

  // Ceci est le constructeur, qui défini comment on crée et assigne chaque attribut
  // Il sera utilisé avec le mot-clé "new"
  public Color(int r, int g, int b) {
    // "this" est utilisé pour faire référence à l'objet lui même
    // Il est seulement obligatoire quand on a d'autres variables du même nom qui existent en même temps
    // Par exemple ici les paramètres de la fonction ont le même nom, donc this est obligatoire
    this.r = max(min(r, 255), 0);
    this.g = max(min(g, 255), 0);
    this.b = max(min(b, 255), 0);
  }

  // Ici on crée une méthode qui va retourner une nouvelle instance de la classe Color
  // Etant donné que cette méthode est statique, on a pas besoin d'une instance de classe pour l'utiliser
  public static Color fromRGB(int r, int g, int b) {
    // On utilise donc le constructeur défini précédemment
    return new Color(r, g, b);
  }

  // Pour modifier et récupérer les attributs qui sont private et s'assurer que l'état de l'objet est toujours cohérent on va faire de "l'encapsulation d'attributs"
  // Ajout d'un "getter" (ou "accesseur") pour récupérer des attributs de la classe (on ne peut pas les récupérer
```

```
directement car l'attribut est private)
  public int getRed() {
    return this.r;
  }

  // Ajout d'un "setter" (ou "mutateur") pour écrire dans des attributs private de la classe (en passant des tests)
  public void setRed(int r) {
    this.r = max(min(r, 255), 0);
  }

  // Ceci est une méthode d'objet (il n'y a pas le mot "static" dans sa définition)
  public float getLightness() {
    // ici on pourait aussi remplacer r g et b par this.r, this.g et this.b
    // mais ce n'est pas obligatoire car il n'y a pas d'autres variables du même nom
    final int maxComponent = Math.max(r, Math.max(g, b));
    final int minComponent = Math.min(r, Math.min(g, b));

    return (maxComponent + minComponent) / (2.0f * 255.0f);
  }
}
```

Maintenant si on veut utiliser cette classe `Color` :

```
// On a pas besoin d'utiliser "new" car c'est déjà dans le contenu de la méthode
Color blue = Color.fromRGB(0, 0, 255);

// ici en revanche on utilise le constructeur, donc on doit utiliser "new"
Color red = new Color(255, 0, 0);

// Maintenant on peut utiliser l'une de ses méthode d'objet
float blue_lightness = blue.getLightness();
float red_lightness = red.getLightness();
```

Pour résumer

La différence entre une classe et un objet :

- Créer une classe c'est comme créer un nouveau type avec ses propre propriétés et ses propres méthodes

- Un objet est une occurence d'une classe (c'est une instance de cette classe)
- On peut créer un objet à partir du constructeur de la classe en utilisant `new`

Ce qui différentie plusieurs objets d'une même classe :

- Les attributs définissent l'état d'un objet, c'est à dire son "ADN", ce qui va le différentier des autres objets de la même classe

Ce qui défini une méthode statique

- Une méthode statique est reconnaissable par le mot-clé `static` dans sa définition. (A savoir que `static` peut aussi être utilisé sur une variable pour devenir une variable de classe)
- Une méthode statique n'a pas besoin d'une instance de la classe (d'objet) pour fonctionner (exemple: println)

Ce qui défini une méthode d'objet

- Une méthode d'objet a besoin d'une instance de la classe (objet) pour fonctionner
- Une méthode d'objet peut récupérer les attributs de celui-ci

Comment utiliser le mot-clé `this` dans les méthodes d'objet :

- Le mot-clé `this` permet d'éviter la confusion entre les noms de variables et représente l'objet lui-même

Les modificateurs d'accès aux variables et classes :

- Tous les attributs ou méthodes ayant le modificateur `private` alors on ne pourra y accéder uniquement au sein de la classe
- Par défault toutes les classes, attributs et méthodes en Java peuvent être accédée par tous les membre du même package
- Tous les attributs, méthodes ou classes ayant le modificateur `public` peuvent être accédé par n'importe quel classe quelque soit le package

L'encapsulation d'attributs permet de ne pas accéder aux attributs directement pour ne pas avoir un état de l'objet qui soit invalide.

- On met donc le modificateur des attributs en `private`
- On crée des méthodes "getter" ("accesseurs") pour récupérer les attributs. Ces méthodes vont souvent commencer par `get` par convention.
- On crée des méthodes "setter" ("mutateurs") pour écrire dans ces attributs (en faisant passer des tests pour vérifier que la valeur que l'utilsiateur veut écrire est valide), ces méthodes vont souvent commencer par `set` par convention.
- **Note:** Si l'un des attributs de l'objet est un autre objet (tel qu'un tableau), il est obligatoire d'en faire une copie quand on le `set` ou le `get` sinon cela brise l'encapsulation des attributs. → copie défensive

Null, equals et toString

Dans ce chapitre on va parler des références (addresses en mémoire) ainsi que du fonctionnement du `.equals()` et du `.toString()`

Qu'est ce qu'une référence "null"

Plus tot, en particulier avec les tableaux, on pouvait se retrouver avec une valeur `null` symolisant l'absence de valeur.

En réalité une valeur null symbolise plus précisément l'absence d'addresse (l'absence de référence). Et il est très important de toujours prendre en compte les cas dans le programme où une valeur nulle pourrait être retournée et la traiter en fonction.

```
// Ici on crée un tableau de Strings d'une longueur de 5 vide
String[] monTableau = new String[5];


// Ici on demande de retourner l'élement en position 0 sans l'avoir redéfini
System.out.println(monTableau[0]);
// On se retrouve avec "null" car il n'y a aucune valeur dans cette position du tableau
```

== vs .equals()

Comme vu dans le chapitre sur les Strings le `.equals()` n'est pas la même chose que l'opérateur booléen `==` sur les objets.

La raison est que `==` va comparer les références tandis que `.equals()` va comparer les valeurs des propriétés.

```
// Si deux chaines ont la même valeur lors de leur définitons, ils auront la même addresse en mémoire
String helloA = "hello world";
String helloB = helloA;
String helloC = "hello world";
```

```
if (helloA == helloB && helloB == helloC) {
    System.out.println("Les références de A, B et C sont les mêmes.");
}

// En revanche si on fait des transformations sur ce String (même si il a au final la même valeur), alors l'addresse sera différente
String helloD = helloA.toUpperCase().toLowerCase();

if (helloA != helloD && helloA.equals(helloD)) {
    System.out.println("Les références de A et D ne sont pas les mêmes.");
    System.out.println("En revanche les deux ont la même valeur.");
}

// Ainsi le == compare les références
// Tandis que le .equals compare les valeurs
```

Héritage et Object

Quand on crée une classe en Java, notre classe va automatiquement "hériter" tout un tas de méthodes et/ou propriétés d'une autre clase par défault dans Java appellée "Object".

Nous allons donc avoir certaines méthodes par défault tel que .equals ou .toString. Mais il faut en général redéfinir ces derniers.

Redéfinition du .equals()

La raison pour laquelle il faut redéfinir le .equals est que par défault, il va avoir le même effet que `==` (comparer les références plus tot que les valeurs). Donc voici un exemple de redéfinition :

```
// @Override indique que l'on va redéfinir une méthode (celle de Object en l'occurence)
// Java va tester au moment de la compilation pour voir si il y a bien une méthode du même nom existant déjà mais il n'est pas obligatoire
@Override
public boolean equals(Object obj) {
    // Si les deux objets ont la même référence, alors elles ont les même valeurs et sont égales
    if (this == obj)
        return true;
```

```
    // Si l'objet n'est PAS une instance de la classe Color, alors il ne peut pas être égal
    if (!(obj instanceof Color))
        return false;

    // Si l'objet est une instance de la classe Color et que tous ses attributs sont égaux, alors les deux objets sont égaux
    Color other = (Color) obj;
    return b == other.b && g == other.g && r == other.r;
}
```

Ce code (sans les commentaires) a été automatiquement généré par Eclipse (en allant dans `Source` puis `Generate hashCode() and equals()` puis cocher les attributs et la case `Use instanceof to compare types`)

> Petite précision : Dans le code on utilise pas de `else` car quand un `return` arrive, la suite du code n'est pas exécuté donc ce n'est pas nécessaire.

> Deuxième précision : Dans Eclipse, on doit cocher la case pour instanceof sinon Eclipse va utiliser une méthode par défault appellée `.getClass()` et le comportement sera différent. Si une nouvelle classe `MyColor` hérite (prends tous les attributs et méthodes) de `Color`, avec instanceof on peut comparer des objets de `Color` et `MyColor` tandis qu'avec `getClass()` on ne pourra comparer que des `MyColor` ensemble ou des `Color` ensemble.

Redéfinition du .toString()

Le `toString` est utilisé pour avoir une représentation textuelle de l'objet (qui doit être courte et informative).

```
@Override
public String toString() {
    return String.format("Color(%d, %d, %d)", r, g, b);
}
```

Il y a aussi un outil dans Eclipse pour faire cela mais il est merdique et beaucoup plus compliqué que de l'écrire en code directement.

Résumé

Les références :

- Une référence `null` signifie une absence de valeur (une absence d'addresse en mémoire)

Equals et == :

- `==` compare les références et non les valeurs.
- `equals()` sert à comparer les valeurs (propriétés), c'est ce qu'il faut utiliser pour comparer des objets.

L'héritage de Object :

- L'héritage signifie qu'une classe "hérite" de toutes les méthodes et attributs d'une autre
- Toutes les classes par défault dans Java héritent de la classe `Object`
- Il faut donc redéfinir certaines méthodes par défault

Redéfinition de méthodes :

- On peut utiliser le mot clé `@Override` pour signifier que la méthode est une redéfinition d'une autre
- `.equals()` doit être redéfini pour ne pas avoir le même comportement que `==`
- `.toString()` est utilisé pour avoir une courte représentation textuelle d'un objet et doit lui aussi être redéfini

Les enums (classes limitées)

Parfois on connais déjà le domaine d'une classe et il est assez réduit. Par exemple si on a une classe `Suit`, on sait déjà que les seules valeurs possibles sont `Spade`, `Heart`, `Diamond` et `Tremol`.

On peut un peu imaginer les enums comme des collections de constantes. Contrairement à d'autres languages de programmation comme le Rust où les `enum` sont plus flexibles.

Définition d'une enum simple

Dans ce cas on peut donc créer un objet special appellé `enum` en y définissant les valeurs possibles :

```
public enum Suit {
    SPADE, HEART, DIAMOND, TREMOL;

    // On pourrait créer nos méthodes ici si on a en a
    public int getValue() {
        // On peut utiliser un swicth pour tester un enum (this fait référence à l'objet actuel)
        switch (this) {
            case SPADE: return 1;
            case HEART: return 2;
            case DIAMOND: return 3;
            case TREMOL: return 4;
            default: return 0; // Une valeur par défault est obligatoire même si tous les cas ont été traité
        }
    }

    // TODO Faire un exemple avec les positions des objets

    public void print() {
        // Les enums ont une valeur string par défault
        System.out.println(this); // Va écrire SPADE, HEART, DIAMOND ou TREMOL dans la console
    }
```

```
    // On peut aussi récupérer un membre d'un enum depuis un String
    public static Suit getSuitFromString(String name) {
        // Attention ! Le nom est case sensitive et si le nom n'est pas trouvé le programme va crash
        return Suit.valueOf(name);
    }
}
```

Définition d'une enum avec attributs

Mais on peut aussi avoir des attributs dans des enums :

```
public enum Suit {
    SPADE(false), HEART(true), DIAMOND(true), TREMOL(false);

    // On va avoir un seul attribut ici appellé "isRed" pour savoir la couleur de la carte
    // Il est condiéré être une bonne pratique de mettre les attributs en final car ils ne sont pas sensé être modifié
    private final boolean isRed;

    // On crée un constructeur pour pouvoir modifier cette valeur
    // Le constructeur n'est pas public car on ne peut pas créer de nouveaux objets
    Suit(boolean isRed) {
        this.isRed = isRed;
    }

    // On va juste faire une petite méthode pour montrer que l'on récupère cet attribut comme dans une classe
    public String getColor() {
        if (this.isRed) {
            return "Red";
        } else {
            return "Black";
        }
    }
}
```

Comment utiliser une enum

Maintenant pour l'utiliser on a plus besoin de `new` :

```
// On met l'objet SPADE dans une variable "spade" de type "Suit"
Suit spade = Suit.SPADE;

// On peut récupérer la position d'un élément dans un enum avec .ordinal()
System.out.printf("La position de HEART dans l'enum est : %d\n", Suit.HEART.ordinal());

// On peut aussi retrouver un élément par sa position en transformant l'enum en Array et en prenant la position
1
Suit heart = Suit.values()[1];

// Et on peux comparer les positions de deux avec .compareTo()
int difference = heart.compareTo(Suit.HEART);
System.out.printf("Si 0 == %d alors c'est un coeur\n", difference);

// On peut maintenant comme n'importe quel classe, appeller ses méthodes tel que "getColor()"
System.out.println(spade.getColor());
```

Convertir une enum en une autre

Mais on peut aussi transformer une enum en une autre si les noms sont compatibles.

```
// Disons une permiere enum "Foo" qui as un attribut String value
// Les attributs vont être perdu lors de la conversion en revanche
enum Foo {
  ONE("hello"), TWO("world"), THREE("foo"), FOUR("bar");

  private final String value;

  Foo(String value) {
    this.value = value;
  }
```

```
  public String getValue() {
    return this.value;
  }
}

// Voici une seconde enum "Bar", la clé ici est que les membres de Foo et de Bar ont le même nom
enum Bar {
  ONE, TWO, THREE, FOUR;
}

// Disons que l'on veut convertir des membres de Foo en Bar
Foo first = Foo.ONE;

// Pour cela on peut prendre le nom de foo avec toString
String name = first.toString();

// Ensuite on peut récupérer le membre du deuxième sur base de son nom avec valueOf
Bar second = Bar.valueOf(name);

// Et donc maintenant "second" vaut bien Bar.ONE et que first vaut toujours bien Foo.ONE
System.out.println(second.equals(Bar.ONE)); // affiche true
System.out.println(first.equals(Foo.ONE));  // affiche true
```

Résumé

Un enum *en Java* :

- Est un moyen de fixer le domaine d'une classe à quelques éléments
- Est immuable (on ne peut pas ajouter de nouveaux objets, ni modifier les attributs)
- Supporte l'utilisation de Switch et peuvent être automatiquement converti en String.
- Chaque membre possède une position et on peut comparer les différents objets par leurs positions.
- Ne nécésite pas de `new` pour être instancié.
- Ses attributs doivent être `private final` pour le rendre immuable et encapsuler ses attributs.

Les méthodes par défault des enums :

- `.compareTo(AutreEnum)` pour comparer les positions de 2 membres d'un enum
- `.ordinal()` pour avoir la position d'un membre de l'enum
- `.values()` pour convertir l'enum en tableau

- `.toString()` donne le nom du membre de l'enum
- `.valueOf(String)` pour récupérer un enum à partir d'un String (donc avec toString très utile pour convertir 2 enums qui ont les même noms de membres)

Les concepts (encapsulation, composition, heritage, etc)

Initialisation, surcharges de constructeurs et méthodes de fabriques

```
class Matricule {
    // Java va d'abord initialiser les attributs de classe une seule fois dans le programme
    public static final String DEFAULT_DEPARTMENT = "Software Development";
    private static int next = 1; // Cet attribut est static mais pas final. Ce qui veut dire qu'il va changer pour toute la classe et donc pour tous les objets aussi

    // Java va ensuite initialiser les attributs d'objets (autant de fois qu'il n'y a d'objets)
    private String department = DEFAULT_DEPARTMENT;
    private int year = 2022;
    private int sequenceNumber;

    // On va déifnir notre constructeur principal avec "public NomDeLObjet(paramètres)"
    // Et on peut faire référence aux attributs de l'objet sous la forme de "this.attribut"
    public Matricule(String department, int year, int seq) {
        if(department != null && !department.isBlank()) {
            this.department = department;
        }
        this.year = Math.abs(year);
        this.sequenceNumber = Math.abs(seq);
        if(seq == next) {
            next++;
```

```
        }
    }

    // On peut ensuite définir des surcharges de constructeurs qui vont utiliser notre constructeur de base
    // Pour cela on doit utiliser "this(paramètres)"
    public Matricule(String department, int year) {
        // Celui ci fait appel au constructeur 1
        // ⚠ ceci doit être la première instruction du constructeur
        this(department, year, next);
    }

    public Matricule(String department) {
        // Et celui ci fait appel au constructeur 2
        this(department, 2023);
    }

    // TODO faire une méthode de fabrique
}
```

Composition vs héritage

Composition

La composition est une technique qui permet de mettre en relations plusieurs éléments entre eux pour créer des structures plus complexes.

Pour cela il suffit de mettre des objets dans les attributs d'autres objets. Ainsi un objet qui contient d'autres objets est appellé *composite* et les objets qui le compose sont appellé *composants*.

```
// Ici c'est un enum mais cela pourrait très bien être juste une classe
enum Level {
    HEADMASTER(1, "Headmaster"),
    PROFESSOR(3, "Professor"),
    GRADUATED(5, "Graduated"),
    STUDENT(7, "Student");

    // reste du code ici
```

```
}

// Wizard est un composite
// Level et String sont des composants de Wizard
class Wizard {
    private Level level;
    private String name;

    // reste du code ici
}
```

Héritage

> **Attention** ⚠ **:** L'héritage n'est plus considéré comme étant une bonne pratique et ne devrait donc être utilisé que dans le cas de maintenance d'une codebase legacy.

```
// Ici on dit que la classe "PlayingCard" va hériter de "BaseCard"
// C'est à dire qu'elle va hériter de toutes ses méthodes et attributs qui ne sont pas private
// Cela veut dire que la classe "PlayingCard" aura toutes les méthodes disponibles par BaseCard + de nouvelles
class PlayingCard extends BaseCard {

}
```

Ne faites pas du traffic d'organes (encapsulation et bonnes pratiques)

Encapsuler ses attributs

```
public class Group {
    // Ces attributs sont private donc ne peuvent pas être directement modifié par un tiers objet
    private String name;
    private String[] groupMembers;
    private int score;

    // Pour pouvoir quand même y accéder on va donc définir un accesseur (getter) qui commence toujours par "get"
    public int getScore() {
        return this.score;
    }

    // Pour pouvoir quand même le modifier on va donc définir un modificateur (setter) qui commence toujours par "set"
    // Cela permet ainsi d'utiliser nos propre conditions et de garantir que l'objet est toujours dans un état cohérent
    public void setScore(int score) {
        if (score >= 0) {
            this.score = score;
        }
    }

    // On fait de même pour "groupMembers"
    // sauf que groupMembers est un tableau et comme les objets, cela signifie que c'est la *référence* qui sera passée
```

```
    // Par conséquent cela pourrait tout de même permettre à l'utilisateur de modifier le contenu de l'objet
    // Nous allons donc faire une "copie défensive"
    public void getGroupMembers() {
        return Arrays.copyOf(this.groupMembers);
    }

    // Un String est aussi un objet sauf que c'est un objet immuable donc nous n'avons pas besoin de faire de copie défensive
    public String getName() {
        return this.name;
    }

    // Et nous n'allons pas définir de setter pour le nom et le groupe car nous souhaitons qu'il ne puisse plus être changé après la construction de l'objet
    // Une classe ou un attribut peut être rendu immuable avec l'utilisation du mot clé "final"
}
```

Ainsi `private` permet de limiter l'accès à quelque chose (méthode, attribut, etc) à seulement la classe courrante. Mais il y a d'autres niveaux également :

Nom	Effet
`private`	Seul la classe courrante peut y accéder
`protected`	Toutes les classes dans le même package **et** les classes qui héritent de la classe actuelle peuvent y accéder
`public`	Tout le monde peut y accéder
Ne rien mettre (par défault)	Seul les classes qui sont dans le même package peut y accéder

Il est conseillé de surtout utiliser `public` et `private`.

N'exploitez pas vos amis

Un objet a ses responsabilités, elle ne doit pas simplemenet stoquer des données mais doit aussi avoir des fonctionalités.

- ⚠ Ne pas faire ça

```
// Cette classe ne fait que stoquer des objets et ça ne devrait pas être le role des autres classse d'implémenter ses fonctions
```

```
public class Apple {
    private int x;
    private int y;

    public int getX() {
        return x;
    }
    public void setX(int x) {
        this.x = x;
    }
    public int getY() {
        return y;
    }
    public void setY(int y) {
        this.y = y;
    }
```

On peut donc ajouter une nouvelle méthode `locateApple` pour placer une pomme dans une certaine position par exemple:

```
    public void locateApple(int dotSize, int randPos) {
        int r = (int) (Math.random() * randPos);
        this.x = ((r * dotSize));

        r = (int) (Math.random() * randPos);
        this.y = ((r * dotSize));
    }
```

En résumé une classe doit implémenter des fonctionalités et éviter de demander aux autres classes de faire son travail.

Ne kidnappez pas les objets

La "loi de déméter" sert à protéger les pauvres objets que vous maltraitez.

Elle défini que vous ne devez intéragir directement qu'avec vos amis et ne pas parler aux inconnus. Et vos amis sont uniquement :

- Les objets en paramètres
- Les objets en attributs

- Les objets de la même classe que vous
- Les objets que vous créez

En revanche les objets qui sont retournés par des méthodes d'une autre classe ne peuvent pas être utilisées directement.

Donc ça c'est juste non...

```
int rank = game.getActivePlayer().getHand().getCardAt(i).getRank();
//         ↓        ↓           ↓        ↓         ↓
//        Game     Player      CardHand  Card      int
```

Dans cet exemple, nous avons un objet de classe `Game` mais on va récupérer et aussi dépendre aussi sur les classes `Player`, `CardHand` et `Card`. Ce qui n'est vraiment pas une bonne idée et rends l'infrastructure du code beaucoup plus complexe.

On pourrait par exemple créer une méthode `getActivePlayerCard(int i)` dans `Game` pour obtenir un Card et réduire le nombre de dépendences (notre classe est amie avec Game et Game (où notre nouvelle méthode est) est amie avec CardHand).

Comment concevoir en orienté objet

Le RDD signifie "Responsability Driven Developement" et c'est un systeme qui permet de concevoir des applications en orienté objet. C'est une méthodologie assez abstraite qui se focus sur les "responsabilités" de chaque classes (appellées ici "candidats")

Pour mieux comprendre la suite il est donc important de se familiariser avec les concepts de candidats et responsabilités

- Une **responsabilité** est une obligation de faire une tâche ou de connaître quelque chose.
- Un **candidat** est une classe (ou tout composant logiciel) suceptible d'avoir des responsabilités

Identifier les fonctionalités

Use Cases

Image not found or type unknown

Tout d'abord il faut identifier les différentes fonctionalités du système, c'est à dire, du point de vue de l'utilisateur·ice. Qu'est ce que l'on peut faire avec l'application.

Dans l'exemple du laboratoire 6 de programmation orienté objet avec une caisse, lors de l'itération 1 on a les fonctionalités suivantes :

- Scanner un article sur base de la première lettre de son nom (SKU)
- Demander le total des articles scannés en comptabilisant leurs réductions de groupes

Cela ressemble donc un peu au processus d'identification des "Use Cases" en analyse.

Ensuite on peut répèter les étapes suivantes pour chaque fonctionalités du système.

Identifier les candidats

Stéréotypes des roles des candidats

Image not found or type unknown

Pour ce faire on peut commencer par identifier les différents stéréotypes des rôles des candidats et les garder dans un coin de sa tête :

1. Le détenteur d’informations connaît et fournit des informations. (va connaître, savoir des informations)
2. Le structureur maintient des relations entre les objets, éventuellement de même type, et des informations à propos de leurs associations.
3. Le fournisseur de services effectue un travail pour le compte d’un utilisateur. (va lire, afficher, calculer, fournir, ajouter, vérifier, etc)
4. Le contrôleur prend des décisions et pilote les actions de ses collaborateurs. (va contrôler, décider, coordonner, notifier, signaler, etc)
5. Le coordinateur réagit à des événements et délègue leurs traitements à d’autres. (va associer, regrouper, lister des éléments)
6. L’adaptateur convertit les données échangées entre l’application et un acteur externe. (par exemple un superviseur par rapport à une vue)

C'est intéressant de garder ces différents stéréotypes de rôles dans la tête pour identifier plus facilement les candidats. Dans notre projet, au vu de la description on peut **par exemple** (car il peut y avoir pleins d'interpretations différentes) identifier les candidats suivants :

- Une vue (pour l'interface)
- Un superviseur (**tip!** si il y a une vue il y a toujours un superviseur)
- Un panier de produits scannés
- Un catalogue des produits disponibles
- Un produit
- Une règle de prix (qui définis les réductions potentielles)
- Un SKU
- Un prix

A savoir que c'est possible d'en avoir moins, ici pour cette première itérations, les éléments produits, prix et SKU sont très peu voir absolument pas utile. Mais les prévoir quand même peut rendre le projet plus modulaire pour y ajouter de nouvelles fonctoinalités par la suite. Ce qui est l'une des raisons pour laquelle il est important de savoir clairement toutes les fonctionalités à représenter.

Identifier les responsabilités

Classification des responsabilités

Une fois que l'on a les candidats potentiels on peut maintenant essayer de trouver les responsabilités qui vont y être assignées.

Une responsabilité est représentée par un groupe **verbal**, donc on peut identifier des responsabilités potentielles avec les mots (et catégories) suivantes par exemples :

- Une connaissance (connaître, savoir)
- Une connexion entre objets (associer, regrouper, lister)
- Un service (lire, afficher, calculer, fournir, ajouter, vérifier, parcourir, convertir, etc)
- Une prise de décisio (contrôler, décider, coordonner, notifier, signaler)

Dans le cas de ce projet on a identifier les responsabilités suivantes :

- Régis au scan d'un produit
- Demander le montant total
- Afficher le montant total
- Ajouter un produit au panier
- Récupérer le montant total
- Récupérer un produit dans la catalogue
- Calculer le prix total
- Regrouper la liste des produits scannés
- Regrouper la liste des produits disponibles
- Fournit un produit sur base de sa première lettre
- Connais une règle de prix
- Connais un SKU
- Connais le nom du SKU
- Connais la première lettre du SKU
- Calcule le prix pour un certain nombre d'articles (produits)
- Connais son prix unitaire
- Connais sa règle de "combo de prix" (réductions)
- Connais sa valeur
- Connais sa devise

Verifier les candidats et les responsabilités

Ensuite on peut relire les responsabilités et les candidats pour s'assurer qu'il n'y a pas de doublons.

Ensuite on peut essayer d'associer les différentes responsabilités aux différents candidats.

Enfin on peut mettre des stéréotypes de rôles sur les candidats (voir plus haut). Et il faut de préférence que chaque candidat aie 2 stéréotypes (il peut y en avoir 3 mais certainement pas moins de 2 ou plus de 3)

Si un candidat n'est pas valide il faut donc essyer de voir qui peut se charger des responsabilités, fusioner avec un autre candidat. Ou dans le cas contraire où il y aurait trop de responsabilités pour un candidat, de les diviser dans différents candidats ou en créer des nouveaux.

Créer des cartes CRC et les lier entre elles

Exemples de cartes CRC

Une fois que l'on a notre liste de candidats et responsabilités vérifiées, on peut les associer pour de bon sur des cartes **CRC** (Classe Responsabilité Collaborateur). Chaque carte va représenter l'un des candidat et va contenir :

1. Le nom
2. La liste des responsabilités associées
3. La liste des autres classes avec qui elle va intéragir pour respecter ses responsabilités

Ainsi on peut d'abord lier les candidats et les responsabilités puis ensuite lier les cartes CRC entre elles. Pour ce faire on peut essayer de simuler le chemin de l'action d'un·e utilisateur·ice. Imaginons pour scanner un article par exemple :

1. On scanne un produit (Vue → Superviseur)
2. On récupère l'item dans le catalogue (Superviseur → Catalogue)
3. On ajoute l'item au panier (Superviseur → Panier)

Et on fait de même pour le calcul du total :

1. On demande pour avoir le total (Vue → Superviseur)
2. On récupère le montant total (Superviseur → Panier)
3. On calcule le prix pour X nombre d'un même produit (Panier → Produit → Règle de prix → Prix)

Ainsi on obtien la structure suivante :

Cartes CRC

Création d'un schéma de collaboration

On peut ensuite représenter les différentes classes de façon à faire des liens entre elles. Et on peut ensuite décrire les appels de fonctions qui vont être fait entre elles pour chaque fonctionalité.

Par exemple pour scanner un article :

Chemin	Appel de fonction
Utilisateur → Vue	Scan d'un article
Vue → Superviseur	`onScan(n)`
Superviseur → Catalogue	`getProduct(n)`
Superviseur → Panier	`addProduct(Product)`

Et pour obtenir le total :

Chemin	Appel de fonction
Utilisateur → Vue	Demande le total
Vue → Superviseur	`onCheckout()`
Superviseur → Panier	`getTotal()`
Panier → Produit	`getPrice(5)`
Produit → Regle de prix	`getPrice(5)`
Superviseur → Prix	`getDevise()`
Superviseur → Vue	`setTotal(50, "EUR")`

On peut représenter tout ceci avec le diagramme suivant :

Diagramme de collaboration

Ainsi on doit pouvoir couvrir l'intégralité des responsabilités, des fonctionalités et des candidats/classes.

On peut aussi vérifier ici que la règle de Demeter est bien respectée. La règle de Demeter consiste en :

- Chaque classe ne doit connaître des choses que sur les classes les plus "proches"
- Chaque classe ne parle qu'a ses attributs ou ce ses méthodes prennent en paramètre mais pas sur des retours de méthodes
- En somme chaque classe ne parle qu'aux objets de la même classe, des objets en attributs, des objets créés par cette dernière et les objets qu'elle a reçu en paramètre

La loi de Demeter est à nuancer toute fois, c'est à éviter (surtout quand c'est des communicatoins avec des classes très lointaines l'une de l'autre) mais des infractions à la loi de Demeter peuvent toute fois arriver. Par exemple ici la liaison entre Superviseur et Prix enfreint la loi de Demeter

Créer le diagramme de classe

Sur base de toutes les méthodes et classes définies, on peut maintenant les grouper dans des classes et y préciser les attributs dont elle a besoin (les informations qu'elle doit mémoriser)

Ensuite on peut faire les liens entre elles. C'est à dire faire les liens comme ceux précisés dans les cartes CRC, en y ajoutant les attributs.

Ainsi on peut vérifier qu'il n'y a pas de dépendence circulaire. Si il y en a ont peut essayer de faire une *inversion des dépendences* à l'aide d'une interface.

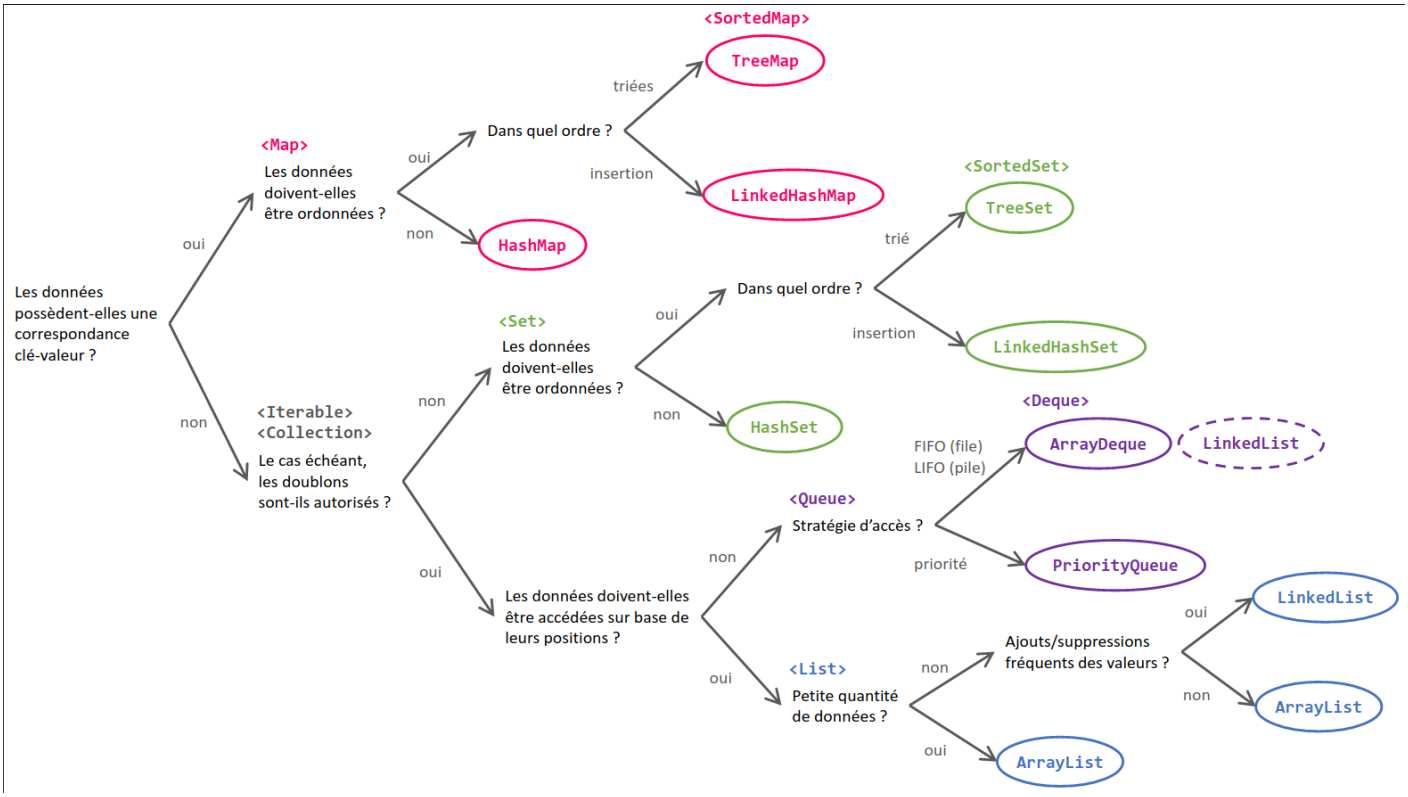
Par exemple ici "La vue compose le superviseur et le superviseur compose la vue" devient "Le Superviseur compose la vue, la vue implémente VueInterface et VueInterface compose Superviseur" ainsi il n'y a plus de dépendences circulaires.

Notre schéma final nous donne donc ceci :

Diagramme de classes

> La flèche en pointillé signifie "A implémente B" Les autres flèches pleines signifient "A compose B"

Donner des types et choisir les bonnes collections



Une fois que l'on connait les attributs, les méthodes, les classes et les responsabilités. On peut commencer à mettre des types sur les différentes donneés. Et quelque chose d'important en est de choisir la bonne collection pour représenter des collections d'éléments. L'image ci dessus devrait pouvoir vous aider à choisir la bonne.

Programmation orientée objet (B2)

Le cours de programmation orientée objet de deuxième année

Introduction

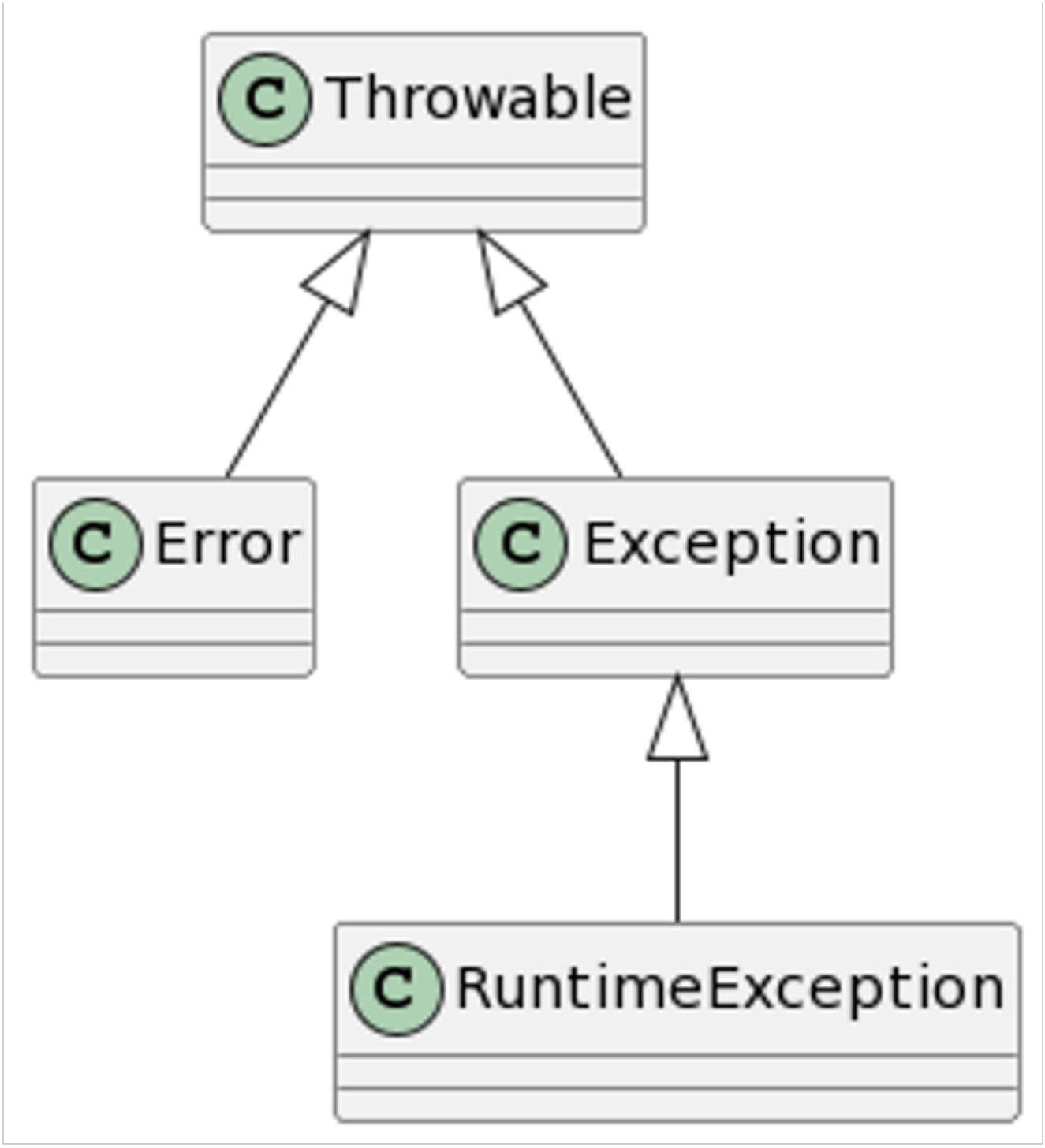
Logiciels

- Gradle (obligatoire)
- Un éditeur de code (au choix), par exemple Eclipse, IntelliJ ou VS Code
- Java JDK 17

Contenu du cours

- Exceptions
- Tests
- Nouveautés Java
- Évènements
- Types génériques
- Réflexion?

Exceptions



On va surtout utiliser Exception et Runtime Exception, mais pas directement (on va plus tot utiliser des exceptions dérivées de ceux-là, tel que NullPointerException ou IllegalArgumentException).

Par défault, le comportement d'une exception est de print l'erreur dans la console, cependant on ne va pas tellement l'utiliser en cours car on va utiliser un système de logging plus élaboré.

Try - catch

```
try {
    // On essaye d'exécuter un certain code ici...
} catch (NullPointerException e) {
    // Le code ici s'exécutera si il y a une NullPointerException et l'erreur sera mise dans la variable e.
} catch (IllegalArgumentException | IndexOutOfBoundException e) {
    // Le code ici s'exécutera dans le cas d'un IllegalArgumentException ou d'une IndexOutOfBoundException et sera mise dans la variable e.
}
// Si aucun problème n'est arrivé ou qu'elle a été bien catchée, le code ici s'exécutera...
```

Les try-catch permettent d'exécuer son propre code dans le cas d'une erreur.

Réexécution de fonction

Si on veut réexécuter une fonction il vaut mieux éviter de rappeller simplement la fonction récursivement dans le `catch` car si une erreur persiste, cela augmente grandement le stack des fonctions appellée ce qui peut mener à faire crash le programme.

Il vaut mieux utiliser un do-while

```
// On initialise les variables en dehors de la boucle
boolean locked = false;
int entier = 0;
do {
    System.out.print("Entrer un entier : ");
    try {
        entier = lireEntier();
        // Si le code a fonctionné, on remet le locked à false pour sortir de la boucle
        // On doit le remettre à false car si elle a raté la première fois, le locked aura été mis à true par le catch
        locked = false;
    } catch {
        // Si un soucis survient, on met la variable locked à true, pour que la boucle réexécute le code
        // Il n'y a ainsi aucune récursion, donc pas de risque de stackoverflow
```

```
        locked = true;
    }
} while (locked);
```

Throw - throws

```
public class PersonalException extends Exception {
    // Ici on met le code de l'exception, par exemple on peut y mettre des messages d'erreurs, des fonctions spéciales, etc.
}

// La méthode suivante retourne une exception
public static void method() {
    // Si quelque chose ne fonctionne pas on peut retourner notre exception custom
    throw new PersonalException();
}
```

On peut créer nos propre exceptions pour des cas particulier de nos programmes en étendant la classe Exception puis en utilisant `throw new` pour l'appeller.

Finally

```
try {
    // On essaye d'exécuter un certain code
} catch (NullPointerException e) {
    // On exécute le code ici si le code dans le try ne fonctionne pas
} finally {
    // Quoi qu'il arrive, le code ici sera exécuté, même si le catch retourne une exception.
}
// Si une erreur arrive dans catch ou finally, le code ici ne sera pas exécuté
```

Lorsque l'on veut qu'un code s'exécute quoi qu'il arrive, on peut utiliser le bloc `finally`, ainsi même si l'un des `catch` retourne une exception, on exécutera quand même le bloc finally.

Tester les exceptions

On peut également tester des exceptions avec `assertThrows`

```
assertThrows(RunTimeException.class, ()=>maMethode());
```

On passe directement la méthode à `assertThrows`. La syntaxe étrange s'appelle une lambda, c'est une fonction anonyme temporaire qui exécute la méthode à tester. Cette *lambda* est nécessaire, sinon on passe le résultat de `maMethode` à la place de passer la méthode elle même. La fonction assertThrows va ensuite tester pour voir si une exeception est émise par la méthode, et si oui, elle va tester que l'exception est bien de la classe `RunTimeException`.

Moteur de production (gradle)

L'objectif est d'automatiser les actions pour produire un logiciel, gérer les dépendances, les détecter et adapter la production du logiciel à la platforme. Gradle est un système qui permet d'automatiser tout ça.

Cela permet donc de rendre un projet indépendant de l'environement de développement de la personne qui écrit le code, car Gradle va automatiquement gérer toutes les dépendences nécessaires.

Historique des moteurs de production

Année	Programme	Langage	Avantage
1977	make	C et autres	
2000	Apache Ant	Java	
2004	Maven	Java	Plus complet que Apache Ant
2008	Gradle	Java (et autres comme Kotlin)	Plus complet que Maven

Que font Gradle et Maven

- La compilation
- Le packaging (jar, war, etc)
- Gestion des dépendences
- Génération de la documentation
- Gestionnaire de sources
- Accès au depot des gestinnaires des dépendences

- Le déploiement en différents environements (test, développement, production, etc)

Qu’est ce que Gradle

Gradle est un moteur de production tournant sur la JVM (Java Virtual Machine). Un moteur de production sert à automatiser les étapes de construction d’un projet. Un moteur de production est nécessaire car le faire à la main serait source d’erreur, très lent et complexe. Gradle au fur et à mesure du temps est devenu très important dans l’écosystème Java.

Pour aller plus loin que les informations du cours, on peut aller consulter la documentation officielle de Gradle.

Gestion simplifiée des dépendences

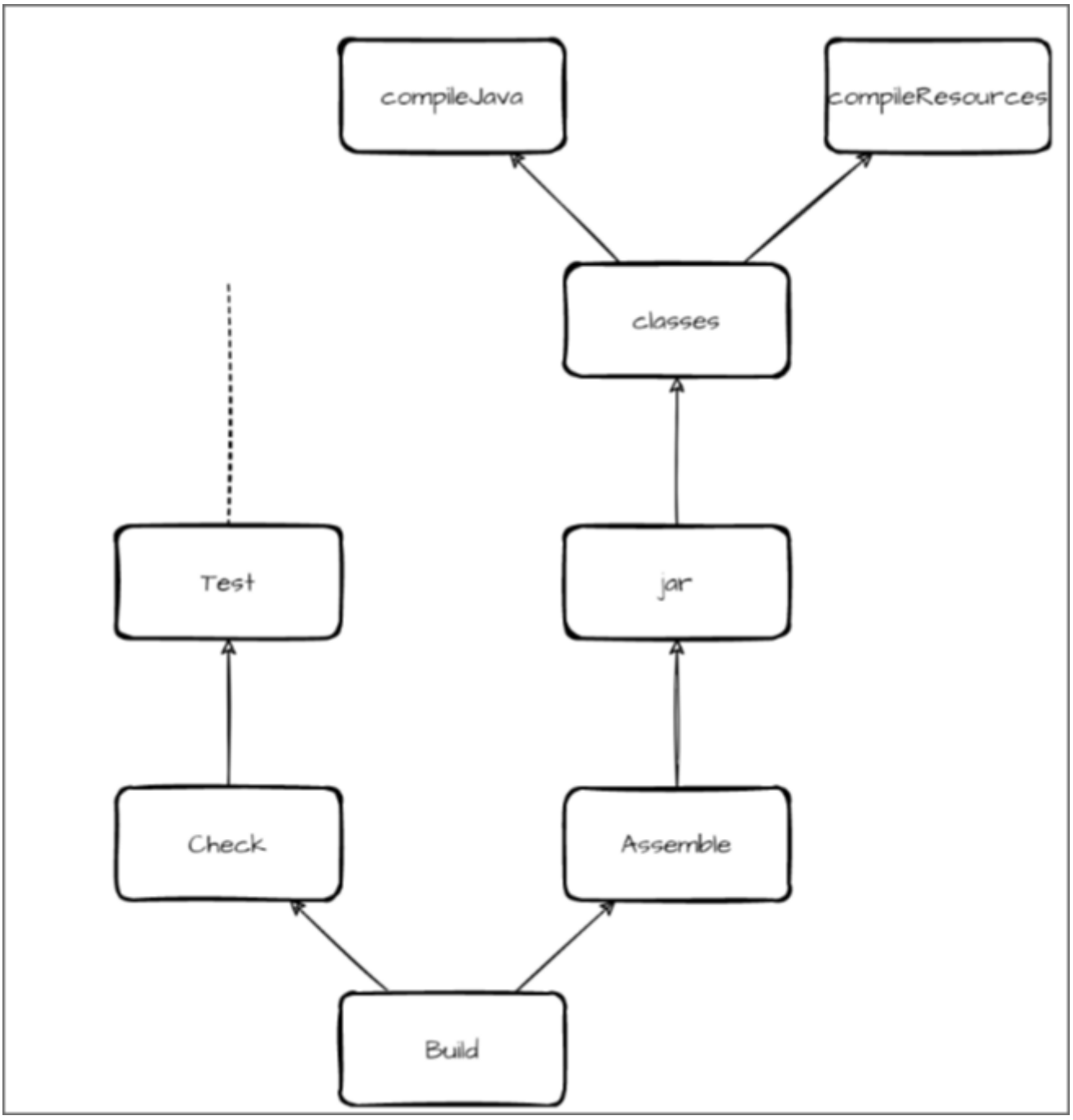
Gradle permet par exemple une gestion simplifiée des dépendances qui permet de simplement installer beaucoup de dépendences depuis les répertoires Maven (un autre moteur de production).

On peut en savoir plus en consultant le site MVN repository.

Les tâches Gradle

Gradle fonctionne sur base de tâches. Chaque tâche permet d’obtenir des sorites (fichiers ou répertoires mis à jour) à partir d’entrées (fichiers, répertoires, configuration, etc).

Les tâches peuvent en inclure d’autres, par exemple la tâche `build` d’un projet Java standard inclus la tache `check` qui va effectuer les tests, et la tâche `assemble` qui va générer l’exécutable.



Système de plugin

De base Gradle est très simple et peu utilisable en pratique. Pour rendre Gradle vraiment utile il faut y intégrer des plugins, comme le plugin `java` par exemple qui permet de compiler des applications Java. Ce système rends donc Gradle très flexible car il peut être beaucoup étendu et n'importe qui peut en créer des plugins

Processus d'une constructoin de projet Gradle

1. L'**initiation** met en place l'environement et détermine les projets qui le compose

2. La **configuration** construit et configure le DAG des tâches (l’arbre qui définit les liens entre les tâches) et définit les tâches à exécuter pour accomplir la tache initiale
3. L’**exécution** exécute les tâches identifiées

La seule phase dans laquelle on intervient est la phase de configuration.

Avantage de Gradle

- Plus rapide
- Plus flexible (pas limité à Java, contrairement à Maven)
- Fichier de config et plus simple
- Gestion des dépendances plus complète
- Indépendant
- Multilangage

Configuration de Gradle

- `settings.gradle` qui est à la racine du projet et contient le nom du projet ainsi que l'ensemble des sous-projets
- `build.gradle` est dans chaque (sous-)projet et contient l'ensemble des éléments utile pour compiler le projet (version de Java, dépendance, , tests, PMD, etc)

Liste des librairies

On peut rechercher les libraries sur le site *MVN Repository*.

Qu'est ce qu'une librairie

On ne s'amuse pas à réinventer la roue dès que l'on veut faire un programme. Du coup on va réutiliser des composants qui ont déjà été fait par d'autres personnes. Une librarie c'est exactement ça, c'est une collection d'outils qui permettent de programmer plus facilement et plus rapidement sans devoir réinveter les même choses en boucle.

Identification d'une dépendence/librarie

Il y a 3 éléments qui identifie une dépendence :

- Le *group id* qui défini le groupe des librairies
- Le *artifact id* qui défini la dépendence dans le groupe
- La *version* de la dépendence

Il est important de vérifier la version des dépendences qui ne sont pas toujours compatibles entre elles et il faut éviter de prendre des versions trop vieilles.

Installation de Gradle

Suivez les instructions sur le site officiel de Gradle.

Création d'un projet et autopsie

Pour créer un projet vous pouvez simplement aller créer un dossier vide, puis entrer dedans avec votre terminal et lancer la commande : `gradle init`

Après quoi Gradle va vous poser certaines questions, pour l'exemple voici ce que vous devez répondre :

- Project type ? → 2 (application)
- Implementation language ? → 3 (java)
- Build script DSL ? 2 (groovy)
- Test framework ? 4 (JUnit Jupiter)

Pour toutes les autres questions, faites juste ENTER pour choisir l'option par défault

Autopsie des tâches

Une fois cela fait nous pouvons ensuite avoir une liste des tâches possibles. Pour cela il suffit d'exécuter `./gradlew tasks`.

Ainsi on voit que si on fait `./gradlew run` ça va lancer le projet (par défault ça va afficher Hello World) ou encore si on fait `./gradlew check` ça va effectuer tous les tests.

Autopsie de la structure des fichiers

```
.
├── app
│   ├── build.gradle
│   └── src
│       ├── main
│       │   ├── java
│       │   │   └── hello
│       │   │       └── App.java
│       │   └── resources
│       └── test
│           ├── java
│           │   └── hello
│           │       └── AppTest.java
│           └── resources
├── .gitattributes
├── .gitignore
├── .gradle
│   └── file-system.probe
├── gradle
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradlew
├── gradlew.bat
└── settings.gradle

13 directories, 11 files
```

- `app/` est le dossier qui contient tout ce qui est relatif à votre projet (en particulier le code source)
  - `app/build.gradle` contient les informations sur le build du projet
  - `app/src/` contient le code source de l'application
    - `app/src/main/` et `app/src/test/` contiennent les deux partie du projet (main pour le code principal et test pour les test unitaires), ces dossiers sont ensuite divisé par langage de programmation (ici Java uniquement), puis par package. Par exemple le App.java se situe dans le package Java test.gradle.
- `.gradle/` est le cache du projet
- `gradle/` contient le *wrapper* de Gradle, c'est grâce à cela que va être automatiquement téléchargée la version Gradle appropriée si on viendrait à essayer de lancer le projet sans avoir Gradle installé ou sans avoir la bonne version de Gradle installée pour le projet.

- `gradlew` et `gradlew.bat` sont les scripts (un pour Linux/macOS et un pour Windows) qui vont exécuter le wrapper gradle pour télécharger la bonne version automatiquement. Il est recommander de les utiliser à la place de la commande `gradle` même si les deux fonctionnent en vérité.
- `settings.gradle` défini quels sont les projets qui font partie du dossier ainsi que d'éventuels autres plugins.
- `.gitignore` défini les fichiers à ignorer dans git
- `.gitattributes` défini les préférences du projet pour git

Autopsie des fichiers de configuration

Vous pouvez maintenant aller ouvrir les fichiers `app/build.gradle` et `settings.gradle` pour découvrir ce qu'il se passe à l'intérieur.

Build.gradle

Le fichier `build.gradle` indique :

- La liste des plugins (ici, uniquement "application")
- La sources des dépendances
- Liste des dépendances
- La définition de la version de java pour paramétrer le plugin "application"
- La définition de la classe principale de l'application (ici test.gradle.App)
- La configuration de la tache de test

Settings.gradle

Le fichier `settings.gradle` indique :

- Une liste de plugins, ici il y a uniquement un plugin servant à télécharger automatiquement des versions du JDK
- Le nom du projet global
- La liste des sous projets (ici il n'y en a qu'un c'est `app`)

Intégration avec Eclipse

Importer un projet Gradle

On peut par exemple importer notre projet Gradle créé plus tot dans Eclipse. Pour cela on peut aller dans Eclipse > File > Import > Gradle > Existing Gradle Project. Ensuite on peut sélectioner le dossier de notre projet Gradle. Enfin dans les “Import Options”, on peut cocher la case “Override workspace settings” et définir le Java home à la localisation du JDK aproprié. Une fois cela fait on peut cliquer sur “Finish” pour importer le projet. Pour être sur que tout a bien charger on peut faire un clic droit sur le projet puis aller dans Gradle puis dans “Refresh Gradle project”

Lancer une tâche Gradle

Pour lancer une tache ou voir la liste des tâches on peut aller dans l’onglet “Gradle tasks”, si l’onglet n’est pas affiché on peut l’afficher en faisant Window > Show view > Other > Gradle Tasks.

Ensuite il suffit de cliquer sur les tâches que l’on veut exécuter, on peut ensuite voir son résultat dans l’onglet Console.

A savoir qu’Eclipse génère tout un tas de fichiers qui ne sont pas intéressants à ajouter dans Git, il vaut donc mieux les ajouter au `.gitignore`

```
# Backupfiles from mergetools #
*.orig

# Java Class Files #
*.class

# Package Files #
*.jar
*.war
*.ear
*.zip

# Eclipse #
.project
.settings
.classpath
bin
```

Ajouts de plugins et dépendences supplémentaires

Il y a certain plugins Gradle que l'on est obligé d'avoir pour l'activité intégrative. Il faut en installer 2 :

- PMD qui fournit des rapports sur les potentielles faiblesses de notre code
- JaCoCo qui fournit un rapport sur le code coverage (la couverture de code couverte par les tests)

PMD (dans Gradle)

Pour installer le plugin PMD dans Gradle, on va tout d'abord l'ajouter dans la liste des plugins du `build.gradle` du dossier `app`.

```
plugins {
    id 'application'
    id 'pmd' // ← Ajout de cette ligne
}
```

Ensuite on peut ajouter le fichier ruleset de pmd présent sur l'espace de cours dans le projet. Une fois cela fait, on peut modifier de nouveau le build.gradle pour y ajouter la configuration de PMD :

```
pmd {
    ruleSets = []
    ruleSetFiles = files("pmd-ruleset.xml") // ← mettre le chemin de fichier relatif vers le fichier PMD ici
    maxFailures = 15 // Défini la limite acceptable d'erreurs PMD avant de stopper le build
}
```

Une fois cela fait PMD va afficher des erreurs dans la console lors du build si l'une de ses règles n'est pas respectée. Et au delà 5 erreurs, PMD va faire stopper le build.

PMD génère aussi un rapport dans le dossier `app/build/reports/pmd/`.

PMD dans Eclipse

Pour avoir les erreurs affichées directement dans l'IDE quand on écrit le code on peut activer le plugin Eclipse comme en B1.

Installation

Pour cela on peut aller dans Help > Eclipse Marketplace > eclipse-pmd > Install.

Ensuite il faut accepter la license et autoriser toutes les sources du plugin quand demandé.

Une fois le plugin installé, il va vous demander de redémarrer Eclipse.

Configuration

Ensuite pour l'activer sur notre projet, on peut aller dans les propriétés du projet (clic droit sur le projet > Properties) puis aller dans l'onglet "PMD" et choisir l'option "Enable PMD for this project".

Une fois cela fait on peut cliquer sur Add > Project > Browse puis sélectionner le fichier ruleset et cliquer sur Finish.

Une fois cela fait on a maintenant les alertes directement dans le code.

Jacoco

Pareil que PMD on doit d'abord ajouter Jacoco dans la liste des plugins :

```
plugins {
    id 'application'
    id 'pmd'
    id 'jacoco' // ← Ajout de cette ligne
}
```

Ensuite on peut le configurer, ici on va le configurer comme dit dans sa documentation en le faisant exécuter à chaque test (un report Jacoco sera ainsi généré à chaque fois que les tests seront effectués) :

```
test {
    finalizedBy jacocoTestReport // report is always generated after tests run
}


jacocoTestReport {
    dependsOn test // tests are required to run before generating the report
}
```

Une fois cela terminé, on peut maintenant lancer la tâche `check` ou `test` et un report jacoco devrait être généré dans le dossier `app/build/reports/jacoco`

Gestion des dépendences

Chaque dépendence dans Gradle a une certaine portée, ainsi certaines dépendencent ont une portée uniquement sur les tests unitaires et d'autres sont nécessaires pour le code principal.

Par exemple si on veut installer la dépdance `Apache Commons Text`. On peut aller rechercher le nom de la dépendance sur le site MVN repository. Ensuite on peut cliquer sur la version qui nous intéresse (ici 1.10.0) puis cliquer sur l'onget "Gradle (short)" pour savoir ce que nous devons ajouter dans la section `dependencies`

Par défault le site MVN repository va proposer de l'installer pour `implementation`, cependant on pourrait très bien choisir `api` ou `testImplementation` :

- `implementation` est la portée requise pour compiler la source de production du projet qui ne font pas partie de l'API exposée par le projet.
- `api` est la portée requise pour compiler la source de production du projet qui font partie de l'API exposée par le projet
- `testImplementation` est la portée requise pour compiler et exécuter la source de test du projet. Par exemple, le projet a décidé d'écrire le code de test avec le cadre de test JUnit.

Vous pouvez trouver plus d'information sur ce sujet sur la page consacrée à la gestion de dépendences Java de Gradle.

Projets modulaires avec Gradle

Gradle permet de décomposer le code en différents modules ce qui permet de maintenir le code plus facilement, ainsi que de le rendre plus robuste et portable.

- Pour l'activer on peut créer un projet avec `gradle init` et activer l'option pour y créer des sous-projets.
- Modifier le projet pour y ajouter des sous-projets

Lorsque l'on crée un projet modulaire avec gradle init, la librairie de test automatiquement choisie est JUnit Jupyter

Structure des projets modulaires

Les projets modulaires ont une structure plus complexe que les projets non-modulaire car ils ont des sous-projets supplémentaires que `app`. Par défault quand on init un projet Gradle avec des sous-projets, on va avoir les dossiers `app`, `buildSrc`, `list` et `utilities` qui vont être créés.

- Le dossier `app` par convention sert toujours pour le code principal.
- Le dossier `buildSrc` sert à créer des plugins ou tâches Gradle spécifiques
- Les dossiers `list` et `utilities` sont juste là à titres d'exemples. Vous pouvez par exemple avoir un sous-projet pour une librarie utilisée par d'autres sous projets, ou encore un sous-projet pour une application CLI et une autre pour une application Android.

Dernière fonctionalités utiles du JDK 17

> Je n’ai ici gardé que les plus importants changements à utiliser.

Depuis JDK 8

Lambdas et interfaces fonctionnelles

Les lambdas qui sont des fonctions anonymes stoquées dans des variables. Et les interfaces fonctionnelles sont des interfaces n’ayant qu’une seule méthode pour une lambda.

Grâce à cela, on peut avoir des types de fonctions plus précis et rendre le code plus sûr.

Voici un exemple d'interface fonctionnelle :

```
// Définition d'une interface foncitonnelle "MyFunction" ayant une fonction prenant en argument 2 int et en retournant 1
@FunctionalInterface
interface MyFunction {
    int apply(int x, int y);
}

// On peut ensuite créer une lambda suivant cette interface
// Le type est donc MyFunction, les arguments sont x et y et le corps de la méthode est après la flèche
MyFunction add = (x, y) -> x + y;

// On peut ensuite utiliser notre fonction
System.out.println(add.apply(1,1)); // Ceci va afficher "2"
```

Default methods in interfaces

Maintenant on peut écrire un code par défault dans les interfaces (rendant donc l'implémentation de ces méthodes par défault optionnelle).

```
public interface Vehicle {
    // Ceci sont des méthodes d'interface normales, il est donc obligatoire de les écrire pour implémenter l'interface
    String getBrand();
    String speedUp();
    String slowDown();

    // Ceci sont des méthodes ayant un code par défault, il n'est donc pas obligatoire de les écrire pour implémenter l'interface
    default String turnAlarmOn() {
        return "Turning the vehicle alarm on.";
    }

    default String turnAlarmOff() {
        return "Turning the vehicle alarm off.";
    }
}
```

Date and time API

Avant Java 8 il n'y avais pas d'API par défault pour gérer le temps et les dates. Maintenant il y en a une. Vous pouvez avoir tous les détails dans la Javadoc de java.time.

A partir de Java 8 on peut simplement récupérer la date et l'heure actuelle en faisant `LocalDateTime.now()`.

Stream API

Pour expliquer plus en détails les avantages de la stream API, voici quelques exemples de code avec et sans stream.

Les streams API sont assez important et sont attendu à l'AI.

Vous pouvez apprendre à les utiliser en regardant la Javadoc de Stream.

Exemple 1

Sans stream api :

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
for (int number: numbers) {
    if (number % 2 == 0) {
        System.out.println(number);
    }
}
```

Avec stream api :

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.stream().filter(n -> n % 2 == 0).forEach(System.out::println);
```

Exemple 2

Sans stream api :

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
for (String name: names) {
    System.out.println(name.toUpperCase());
}
```

Avec stream api :

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.stream() // Génère le Stream
    .map(String::toUpperCase) // Passe tout en uppercase
    .forEach(System.out::println); // Print chaque élément dans la console
```

Exemple 3

Imaginons que l'on a une liste de valeurs en nombre et en String et que l'on veut faire la moyenne tout en passant le premier élément :

Sans la stream API :

```
public static double getAverage(List<String> values) {
    double sum = 0.0;                                   // Commence à 0
    final int counter = values.size();                  // Compte le nombre d'items dans la liste
    if (list.isEmpty()) return 0;                       // Retourne 0 si la liste est vide pour empécher une division par
0
    for (int i = 1; i < values.size(); i++) {           // Fait une boucle passant le premier item
        String stringValue = value.get(i);              // Récupère la valeur en String
        double numericValue = Double.parseDouble(stringValue);  // La converti en nombre
        sum += numericValue;                            // L'ajoute à la somme
    }
    return sum / counter;                               // Fait le calcul de la somme
}
```

Avec la stream API :

```
public static double getAverage(List<String> values) {
    return values.stream()                          // Converti la liste en Stream
        .skip(1)                                    // Passer le premier élément
        .mapToDouble(value -> Double.parseDouble(value)) // Tout convertir en double
        .average()                                  // Caluler la moyenne de tout
        .orElse(0.0);                               // Si la liste est vide, va retourner 0
}
```

Depuis JDK 9

Listes immuables

On peut maintenant créer des listes (ou d'autres collections) immuables en Java avec `of` ou `copyOf` :

```
List<String> list = List.of("Hello", "World");
```

Depuis JDK 10-11

Inférence de type

Depuis Java 10 ou 11 on peut maintenant utiliser l'infférence de type, on est donc plus obligé de préciser le type d'une variable locale. On peut simplement utiliser le mot clé `var`

```
// Avant Java 10
String message = "Hello, World!";

// Possible après Java 10
var message = "Hello, World!";
```

JDK 17

Sealed classes

Les *sealed classes* permettent de limiter les classes qui peuvent hériter de la classe actuelle. Il n'est pas recommandé d'utiliser ceci pour la même raison qu'il n'est pas recommandé d'utiliser l'héritage.

```
// Seule les classes Cricle et Square pourront hériter de Shape
public sealed class Shape permits Circle, Square {
}
```

Switch expressions

Les switch peuvent maintenant fonctionner avec n'importe quel type (et pas uniquement int comme avant), de plus on peut en écrire plusieurs sur la même ligne :

```
int month = 2;
int daysInMonth = switch (month) {
    case 1, 3, 5, 7, 8, 10, 12 -> 31;
    case 4, 6, 9, 11 -> 30;
    case 2 -> 28;
```

```
    default -> throw new IllegalArgumentException("Invalid month: " + month);
};
```

Records

Les records permettent de créer très simplement des data class pour représenter certains concepts et les stoquer.

Voici comment représenter une Personne avec un nom et un age avant les Records :

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return this.name;
    }

    public int getAge() {
        return this.age;
    }
}
```

Et voici comment représenter à peu près la même chose avec un Record

```
record Person(String name, int age) {}
```

Text blocks

Depuis JDK 17 on peut aussi simplement faire des Strings de plusieurs lignes avec `"""`.

Voici comment faire un String de 3 lignes avant JDK 17 :

```
String message = "Welcome\n How are you?\n I hope you have a nice day";
```

Et voici comment faire la même chose depuis JDK 17 :

```
String message = """
                 Welcome
                 How are you?
                 I hope you have a nice day
                 """;
```

Doublures de test

Introduction

Les doublures de tests permettent d'isoler les classes à tester et de briser les intéraction entre elles. Les doublures de tests ne remplace pas JUnit et permet de tester les appels que la classe va faire aux autres classes.

Par exemple, imaginons que l'on a une classe `Messagerie` qui peut lire les messages d'un·e utilisateur·ice sur base de son identifiant et mot de passe. Pour vérifier l'utliisateur·ice, la Messagerie va faire appel à la classe `Identification`. Cependant ici on ne veut tester uniquement Messagerie mais pas la classe d'Identification.

Pour avoir plus d'explications vous pouvez aller voir cet article (le code en exemple n'y est pas forcément de la meilleure des qualités mais c'est pratique pour comprendre le principe des doublures).

Pourquoi utiliser une doublure ?

Il y a plusieurs raisons pour laquelle on voudrait mettre en place une doublure :

- La classe testée fait appel à un composant dificile ou couteux à mettre en place (par exemple une base de donnée, c'est nottament le cas pour notre classe `Identification` dans l'exemple)
- Le test vise à vérifier le comportement d'une classe dans une situation exceptionnelle (imaginons, une déconnexion réseau, on ne vas pas réellement déconnecter la machine juste pour faire un test)
- La classe testée fait appel à un composant qui n'existe pas encore ou qui n'est pas suffisament stable (cela permet par exemple de tester notre classe `Messagerie` alors que la classe `Identification` dont elle dépends n'existe pas encore)
- Le test fait appel à du code lent (par exemple, si notre Identification prends un certain temps, cela ralentirait grandement les tests pour rien)
- Le test fait appel à du code non déterministe (par exemple à l'heure ou à l'aléatoire. Par exemple si une classe ferait appel à une classe générant des nombres entre 1 et 6, nos tests serait faux 5 fois sur 6)
- Séparer le code de test du code de l'application (par exemple si on crée une méthode `fakeGenerateNumber()` dans une classe aléatoire, cela complique les choses pour rien)

Types de doublures

Les stub objects

Un stub est simplement une classe écrite à la main spécialement pour le contexte du test. On sait quelle valeurs sont attendues d'avance et on les hardcode dans la classe.

Par exemple, dans le cas de l'idenficiation de et de la messagerie on peut avoir une interface `Identification` et créer une classe `IdentificationStub` implémentant cette interface de façon à hardcoder les valeurs attendues pour le test (par exemple) :

```
public class IdentificationStub implements Identification {
    boolean identify(String username, String password) {
        // Renvois true si l'identifiant est 'toto' et le mot de passe 'mdp'
        return "toto".equals(username) && "mdp".equals(password)
    }
}
```

Pour le test il suffit alors simplement d'injecter la classe `IdentificationStub` dans le constructeur de la classe `Messagerie`.

```
public class MessagerieTest {
    @Test
    void testLireMessages() {
        // On injecte le stub dans la classe à tester
        Messagerie messagerie = new Messagerie(new IdentificationStub());

        // On fait les tests comme on le souhaite dessus...
    }
}
```

Les fake objects

Un fake est une doublure écrite à la main qui implémente le comportement attendu mais de façon plus simple que la classe réelle. Contrairement au stub qui est écrit spécialement pour un test précis, le fake a vocation à être suffisament générique pour être utilisé dans plusieurs tests. Il est donc plus complexe que le stub mais plus réutilisable.

Pour reprendre l'exemple précédent on peut imaginer une classe `IdentificationFake` qui implémente `Identification` mais qui a une méthode supplémentaire `addAccount(String username, String password)` permettant de personaliser le test.

```
public class IdentificationFake implements Identification {
    Map<String, String> comptes = new HashMap<String, String>();

        @Override
        public boolean identify(String username, String password) {
            // Vérifie que l'identifiant et le mot de passe soit dans la liste des comptes
            return comptes.containsKey(identifiant) && comptes.get(identifiant).equals(password);
        }

        public void addAccount(String username, String password) {
            // Ajoute un nouvel identifiant-mot de passe dans la fausse liste des comptes
            comptes.put(username, password);
        }
}
```

Comme pour le stub on peut donc aller l'injecter dans le constructeur lors du test, à la différence qu'ici on peut l'utiliser pour plusieurs tests différents et le configurer différemment à chaque fois.

```
public class MessagerieTest {
    private Identification identification = new IdentificationFake();

    @Test
    void testLireMessages() {
        // On configure notre fake object
        identification.addAccount("toto", "mdp");

        // On peut ensuite l'injecter dans notre classe à tester
        Messagerie messagerie = new Messagerie(identification);

        // Enfin on peut faire nos tests comme on le souhaite...
    }

    // on peut ensuite faire d'autres tests sur le même principe sans avoir à créer plusieurs classes pour chaque
cas
}
```

Les dummy objects

Les dummy sont le type de doublure le plus simple, ce sont simplement des classes implémentant l'interface attendue mais ne faisant absolument rien car ils ne sont jamais vraiment utilisés.

Par exemple si on teste un cas précis où l'identification n'est jamais utilisée on peut créer une classe dummy implémentant `Identification` et qui renverrai toujours la même valeur (car quelque soit la valeur on s'en fout puis ce qu'elle ne sera pas utilisée) :

```
public class IdentificationDummy implements Identification {
    @Override
    public boolean identify(String username, String password) {
        return true;
    }
}
```

Ici le code du test se fait exactement comme pour le stub

```
public class MessagerieTest {
    @Test
    void testLireMessages() {
        // On injecte le stub dans la classe à tester
        Messagerie messagerie = new Messagerie(new IdentificationDummy());

        // On fait les tests comme on le souhaite dessus...
    }
}
```

Les mock objects

Les mocks objects sont plus complexes mais plus flexibles que les autres et c'est ceux là que l'on va priviléger pour l'activité intégrative en utilisant la librarie Mockito.

Contrairement aux autres, les mocks sont générés par une librarie, on a donc pas besoin de créer la classe nous même, il suffit juste de dire dans notre test que l'on souhaite créer un Mock et quelle valeur on veut que certaines méthodes retournent.

Ainsi les Mocks ont la simplicité des Fake objects mais sans avoir à créer la moindre classe soi-même.

Pour l'exemple précédent à la place de créer tout une classe on a simplement à définir ceci dans le test :

```
public class MessagerieTest {
    // On demande à Mockito de créer un mock pour nous
    @Mock private Identification identification;

    @Test
    void testLireMessages() {
        // On configure le mock pour lui dire les paramètres et réponses attendues
        when(identification.identify("toto", "mdp")).thenReturn(true);

        // On l'injecte dans le constructeur de la messagerie
        Messagerie messagerie = new Messagerie(identification);

        // Enfin on peut faire les tests sur la messagerie comme on le souhaite...
    }

    // On peut ensuite réutiliser notre mock de la même façon pour d'autres tests
}
```

Libraries

Il existe 2 librairies principales pour faire du *mocking* en Java, mais ici c'est Mockito qui a été privilégié.

Mockito

- facile d'utilisation
- configuration via annotation simple
- très grande communauté
- Choisi pour le cours

La documentation de Mockito est assez affreuse mais au moins elle est là, vous pouvez retrouver quelques liens intéressants sur leur site, ainsi que leur documentation officielle.

EasyMock

- Facile d'utilisation
- Configuration simple mais plus chiant que l'autre
- Moins utilisé que mockito

Spy objects

Les spy objects permettent de vérifier qu'une méthode à été appellée, de savoir combien de fois et avec quels arguments. Pour reprendre l'exemple précédent, si on imagine que la méthode `identify` est void, et ne retourne donc rien; on pourra tout de même la tester en vérifiant qu'elle a bien été appellée avec les bons arguments.

Cela peut être fait dans un Fake object mais est beaucoup plus compliqué à mettre en place, cela est en revanche trivial à faire avec Mockito :

```
public class MessagerieTest {
    @Mock private Identification identification;
    @Test
    void testLireMessages() {
        // On injecte la méthode dans le constructeur de la messagerie
        Messagerie messagerie = new Messagerie(identification);

        // On fait nos tests...

        // On peut ensuite par exemple aller vérifier que la méthode ~identify~ a été appellée exactement une fois
avec les paramètres "toto" et "mdp":
        verify(identification, times(1)).identify("toto", "mdp");
    }

}
```

De plus Mockito permet également d'espioner de vrais objets.

```
public class MessagerieTest {
    @Spy private Identification identification = new RealIdentification();

    @Test
    void testLireMessages() {
        // On injecte la classe espion dans la messagerie
        Messagerie messagerie = new Messagerie(identification);

        // On fait nos tests

        // On peut vérifier que l'identification a bien été appellée :
        verify(identification, times(1)).identify("toto", "mdp");

        // Note, si on le souhaite on pourrait même stub les méthodes de la vrai classe en faisant
when().thenReturn() par exemple
```

```
    }
}
```

Logging

Le logging permet de débugger plus simplement les application avec plus de finesse qu'avec `System.out.println`, cela permet nottament de filtrer les logs selon le type (DEBUG, INFO, etc) ainsi que rediriger le flux des logs dans des fichiers.

JUL

JUL est la classe de log par défault dans Java, elle peut être importée depuis `java.util.logging`.

Log4j2

Log4J 2 est le successeur de Logback qui lui même est le succèsseur de log4j. Plus personne n'utilise (ou n'est sensé utiliser) log4j à cause de très très gros soucis de sécurité.

Cette librairie n'est pas inclue de base dans Java mais est disponible dans la dépendence `org.apache.logging.log4j:log4j-code:2.20.0`.

Log4j2 est également configuré avec de l'XML ou avec un fichier properties.