Java 101

Loïc Ledoyen

Table of Contents

Introduction

Java est un langage sorti en 1998 et développé par Sun Microsystems.

On fait la distinction entre la JVM (Java Virtual Machine) qui permet l'exécution d'un programme Java et le JDK (Java Development Kit), qui comprend la JVM, ainsi que les outils de développement permettant de créer un programme Java, notamment le compilateur.

En 2006, le compilateur et la machine virtuelle (JVM pour Java Virtual Machine) sont open-sourcés sous licence GNU.
En 2009, Oracle acquiers Sun Microsystems et Java devient un produit Oracle.
L'utilisation des binaires du JDK (Java Development Kit) produits par Oracle sont alors soumis à une license complexe pour les usages professionnels.

Il existe cependant d'autres distributions (la différence pour les entreprises étant le niveau de support possible) :

- Eclipse Adoptium (anciennement AdoptOpenJDK)
- Azul Zulu Builds of OpenJDK
- Red Hat build of OpenJDK
- Amazon Corretto
- etc.

Objectifs du Langage

Initialement, le langage Java est créé pour palier aux défauts de C++, et notamment la gestion manuelle de la mémoire, tout en gardant le côté ***orienté objet***.

NOTE	À cette fin le ramasse-miettes (Garbage Collector ou GC) est créé. Il s'agit d'un programme indépendant s'exécutant dans la JVM et dont le rôle est de scruter les objets qui ne sont plus référencés afin de libérer la mémoire correspondante.

Les créateurs du langage souhaitaient également que les programmes écrits en Java soient indépendants de la plateforme d'exécution (que ce soit Linux, Mac ou Windows).

NOTE	Pour cela, un programme Java est d'abord compilé, ce qui produit du *byte code*. Ce *byte code* est ensuite interprété par la JVM qui elle dépend de la plateforme. Un programme Java n'a donc pas besoin du bit d'exécution pour pouvoir être lancé sous Linux, car la JVM va lire (interpréter) les binaires précédemment compilés.

Par la suite, s'ajoutent à ces objectifs initiaux

- une forte rétro-compatibilité, ce qui permettra une large adoption du langage
- une efficacité à l'exécution grâce au compilateur JIT (Just In Time) qui optimise le *byte code* en analysant son utilisation.
  Cette technique est associée à un temps de chauffe, temps nécessaire au JIT pour "observer" l'utilisation qui est faite du *byte code*.
- le support dans la bibliothèque standard d'un certain nombre de connecteurs (TCP, UDP, HTTP, etc.), de structures de données (`List`, `Set`, `Map`, etc.) et d'algorithmes (b-search, etc.)
- une évolution constante permettant de s'adapter aux paradigmes modernes, comme la programmation fonctionnelle ou réactive

Globalement, Java se veut un langage de haut niveau, facile d'accès et performant (ce point peut être débattu, car la JVM utilise une mémoire non négligeable).

Adoption

Java compte aujourd'hui parmi les languages les plus populaires en entreprise, pour les raisons précédentes, mais également grâce à un écosystème riche.

Dans cet écosystème on trouve notamment :

- Maven et Gradle (et d'autres systèmes de construction moins répandus) qui permettent de gérer *simplement* un projet et ses dépendances
- d'autres languages qui compilent vers du *byte code* (Groovy, Scala, Kotlin, etc.)
- une communauté sans cesse croissante de programmeurs venant enrichir une grande collection de bibliothèques.

- des IDE de très bonne qualité (IntelliJ, Eclipse, etc.)

1. Le langage Java, première partie

1.1. Point d'entrée d'un programme Java

Un programme java a un unique point d'entrée qui sera utilisé pour le lancer.

Ce point d'entrée, la fonction `main` est une méthode publique, statique, sans type de retour et prenant en paramètre un tableau de chaîne de caractères :

Listing 1. Fichier Launcher.java

```
public class Launcher { ①
    public static void main(String[] args) { ②
        System.out.println("Hello " + (args.length > 0 ? args[0] : "World")); ③
    }
}
```

① La classe contenant la fonction `main` doit être publique

② la *signature* de la méthode doit être exacte, si elle n'est pas statique ou publique ou que les paramètres ne sont pas un unique tableau de chaîne de caractères, la fonction ne sera pas reconnue

③ du code *procédural* peut être écrit à l'intérieur de cette méthode, en Java, les instructions doivent se terminer par un point-virgule `;`

Le paramètre de la fonction `main` correspond aux arguments passés au programme.

Par exemple :

```
javac Launcher.java ①
jar -cf helloWorld.jar Launcher.class ②
java -jar helloWorld.jar Launcher Lernejo # Will display Hello Lernejo ③
```

① compilation, cela va produire le fichier Launcher.class contenant du *byte code*

② création d'une archive Java contenant l'unique classe compilée

③ execution du programme, Java va chercher une fonction `main` dans la classe `Launcher` qui lui est indiquée comme point d'entrée, le tableau `args` aura une unique entrée, le seul paramètre qui est passé au programme : `"Lernejo"`

1.2. Les types

En Java il existe deux formes de type, les types ***primitifs*** et les types objets.

Les premiers commencent par une minuscule et représentent directement la donnée en mémoire :

Nom	Nombre de bits	Valeurs possibles	Exemple
boolean	32 (un `int` avec 0 ou 1 comme valeur)	true ou false	true
byte	8	entier positif ou négatif	
short	16	entier positif ou négatif	32
int	32	entier positif ou négatif	1452
long	64	entier positif ou négatif	164478945
float	32	IEEE 754	124.54
double	64	IEEE 754	124587451.1254878
char	16	Unicode	'h'

Les seconds sont des objets et leurs noms commencent par une majuscule.

IMPORTANT	C'est pourquoi il est important, quand on crée un nouveau type (classe, interface, enum ou record) que son nom commence par une **Majuscule**.

1.3. Les variables

Au sein d'une méthode, il est souvent nécessaire de stocker un résultat intermédiaire dans une variable.

En Java, une variable est en fait un référant, un "pointeur" vers une adresse mémoire.

Ainsi en écrivant `var b = a;`, on déclare un second référant `b` qui pointe vers la même adresse mémoire que `a`.

En Java, il est possible de déclarer une variable de plusieurs façons.

- `String a;` : la variable n'est pas assignée, elle ne pointe vers rien, il est nécessaire de lui assigner une valeur afin de pouvoir l'utiliser par la suite.
  Par exemple : `a = "my-name";`
- `int a = 43;` : la variable est déclarée et assignée, elle peut être utilisée, sa valeur peut également être changée.
  Par exemple : `a++;`
- `final MyType a;` : la variable n'est pas assignée, mais grâce au mot-clé `final`, le compilateur va garantir qu'elle ne le sera qu'une unique fois.
  Par exemple :

Listing 2. Fichier NameGenerator.java

```
public class NameGenerator {
    public String generateName(FacialHairStyle hairStyle) {
        final String name;
        if(Gender.BEARDED == hairStyle) {
            name = "Barbarossa";
        } else if(Gender.MUSTACHE == hairStyle) {
            name = "Jenkins";
        } else {
            name = "Saitama";
        }
        return name;
    }
}
```

- Enfin, il est également possible d'utiliser le mot-clé `var` si le type de la variable peut être *inféré* au moment de sa déclaration.
  Cet usage est à recommander aux endroits où la longueur d'un type diminue la lisibilité.
  Par exemple :

Listing 3. Fichier OccurrenceUtils.java

```
public class OccurrenceUtils {
    public Optional<String> mostOccurring(List<String> strings) {
        Map<String, Long> freqMap = strings.stream()
            .collect(Collectors.groupingBy(s -> s, Collectors.counting()));
        Optional<Map.Entry<String, Long>> maxEntryOpt = freqMap.entrySet()
            .stream()
            .max(Map.Entry.comparingByValue());
        return maxEntryOpt.map(Map.Entry::getKey);
    }
}
```

Peut être simplifié :

Listing 4. Fichier OccurrenceUtils.java

```
public class OccurrenceUtils {
    public Optional<String> mostOccurring(List<String> strings) {
        var freqMap = strings.stream()
            .collect(Collectors.groupingBy(s -> s, Collectors.counting()));
        var maxEntryOpt = freqMap.entrySet()
            .stream()
            .max(Map.Entry.comparingByValue());
        return maxEntryOpt.map(Map.Entry::getKey);
    }
}
```

1.4. Opérateurs

Il existe plusieurs opérateurs en Java.

Opérateurs de calculs :

Opérateur	Description	Exemple
+	Additionne deux nombres ou concatène deux chaînes de caractères	1 + a
-	Soustrait deux nombres	8 - a
*	Multiplie deux nombres	b * 4
/	Divise deux nombres	a / 2
%	Modulo (reste de la division entière)	a % 3

Les opérateurs d'assignations stockent le résultat du calcul dans l'opérande de gauche :

Opérateur	Description	Exemple
+=	Additionne deux nombres ou concatène deux chaînes de caractères	a += "toto"
-=	Soustrait deux nombres	a -= 3.2
/=	Multiplie deux nombres	b /= 2
*=	Divise deux nombres	a *= 2

Les opérateurs d'assignations peuvent être écrits avec les opérateurs de calculs.
Par exemple `b /= 2;` est équivalent à `b = b / 2;`.

Les opérateurs d'incrémentation peuvent être placés à gauche ou à droite d'une variable de façon à ce que l'opération soit réalisée avant ou après l'exploitation du résultat.

Par exemple, dans `array[++i] = 0 ;`, c'est la valeur de `i` *après* l'incrémentation qui est utilisée comme index du tableau.
A contrario, dans `array[i--] = 0 ;`, c'est la valeur de `i` *avant* la décrémentation qui est utilisée comme index du tableau.

Les opérateurs de comparaison, renvoient vrai si…

Opérateur	Description	Exemple
==	… les deux valeurs ont la même adresse mémoire	a == 3
!=	… les deux valeurs n'ont pas la même adresse mémoire	a != 3

<	… le nombre de gauche est plus petit (strictement) que celui de droite	a < 3
⇐	… le nombre de gauche est plus petit ou égal à celui de droite	a ⇐ 3
>	… le nombre de gauche est plus grand (strictement) que celui de droite	a > 3
>=	… le nombre de gauche est plus grand ou égal à celui de droite	a >= 3

Certains opérateurs logiques peuvent s'appliquer sur les entiers, auxquels cas ils fonctionnent bit à bit.

Opérateur	Description	cible	Exemple
&&	AND	boolean	a && b
||	OR	boolean	a || b
&	AND	boolean et entiers	a & b
|	OR	boolean et entiers	a | b
^	XOR	boolean et entiers	a ^ b

Les opérateurs de décalage de bit :

Opérateur	Description	Propagation du signe	Exemple
<<	Décale les bits vers la gauche (multiplie par 2 à chaque décalage). Les bits qui sortent à gauche sont perdus, et des zéros sont insérés à droite	oui	6 << 2
<<	Décale les bits vers la droite (divise par 2 à chaque décalage). Les bits qui sortent à droite sont perdus, et le bit non-nul de poids plus fort est recopié à gauche	oui	6 >> 2

>>>	Décale les bits vers la droite (divise par 2 à chaque décalage). Les bits qui sortent à droite sont perdus, et des zéros sont insérés à gauche	non	6 >>> 2

L'opérateur `instanceof` renvoit vrai si le type de l'objet testé, est égal à, ou égal à un sous-type de, l'opérande de droite.
Par exemple :

```
if (a instanceof ArrayList) {
    // ... ①
}
```

① l'execution entrera dans le bloc si l'objet pointé par la variable `a` est de type `ArrayList` ou d'un sous-type d' `ArrayList`

1.5. Nommage

Le nommage a un intérêt prépondérant dans le paradigme objet où le développeur essaie d'exprimer des concepts réels.
Les classes, les champs, les méthodes, les variables, tous doivent avoir un nom clair et représentatif du rôle que joue le composant.
Les noms peuvent être relativement longs sans que ce soit un problème.
La convention en Java est le `camelCase` de manière générale, l' `UpperCamelCase` pour les types (nom de classe, d'interface, d'enum ou de record).
On peut également trouver/utiliser le `lower_snake_case` pour les noms des méthodes de test.

1.6. Annotations

Les annotations sont des *marqueurs* qu'il est possible de placer à différents endroits afin

- de marquer un morceau de code visuellement sans que cela ait un impact sur le comportement du code
- déclencher un comportement à la compilation / construction
- déclencher un comportement en *runtime* (durant l'exécution)

Java fournit entre autre l'annotation `@Override` qui permet de déclarer une méthode comme étant une surcharge d'une méthode parente.
Si jamais il n'existe pas (ou plus) une telle méthode parente, cela provoquera une erreur de compilation.

Listing 5. Fichier Watchable.java

```
public interface Watchable {

    String name();
}
```

Listing 6. Fichier Movie.java

```
public class Movie implements Watchable { ①
    public final String name;

    public Movie(String name) {
        this.name = name;
    }

    @Override ②
    public String name() {
        return name;
    }
}
```

① La classe `Movie` déclare qu'elle *implémente* l'interface `Watchable`

② l'annotation ici déclare la méthode `name` comme étant la surcharge d'une définition dans la hiérarchie de la classe.
Supprimer la méthode de l'interface, ou enlever la référence à l'interface provoquera une erreur de compilation.

Ce mécanisme est utile lorsqu'on implémente ou surcharge une méthode définie dans la bibliothèque standard ou dans une bibliothèque tierce.
Faire une mise à jour de la bibliothèque en question peut changer les définitions connues, et dans ce cas la compilation permet d'identifier qu'il y a quelque-chose à adapter.

1.7. Les Objets

Un objet est constitué de données (son état) **et** de comportements.

L'état est représenté par des champs, et le comportement par des méthodes.

Un objet est une instance de classe.

1.7.1. Anatomie d'une classe

Listing 7. Fichier Cat.java

```
package com.lernejo.animals; ①

import java.util.Random; ②

public class Cat { ③
    private boolean sleeping; ④

    public boolean tryToWakeUp() { ⑤
        if (!sleeping) {
            throw new IllegalStateException("The cat is already awake");
        }
        sleeping = new Random().nextBoolean();
        return sleeping;
    }
}
```

① package

② imports

③ définition de la classe `Cat`, son contenu commence après l'accolade ouvrante et se termine avant la dernière accolade fermante

④ champs

⑤ méthodes

Le package (équivalent du namespace en C++ ou C#) dans lequel se trouve la classe est une façon d'organiser son code afin :

- de ne pas avoir des milliers de fichiers dans le même répertoire
- de faire cohabiter des objets de même nom dans des contextes différents, par exemple
  - `org.junit.jupiter.api.Assertions` classe utilitaire fournie par la bibliothèque JUnit
  - `org.assertj.core.api.Assertions` classe utilitaire fournie par la bibliothèque AssertJ

NOTE	La concaténation du package et du nom de la classe est appelé chemin qualifié. Une classe doit être dans une hiérarchie de répertoires correspondante au package déclaré en entête. C'est-à-dire que la classe ci-dessus doit être compilée comme ceci : `javac com/lernejo/animals/Cat.java`

Les imports, permettent d'utiliser des types qui ne sont pas dans le même package ou dans le package `java.lang`.
Accompagné du mot clé `static` (`import static …`), un import permet d'utiliser une méthode statique sans avoir à la préfixer par la classe la contenant.

Les champs contiennent l'état de l'objet.
Ils sont la plupart du temps `private` afin de pas être accessibles à l'extérieur de la classe qui les

déclare.

Ils peuvent être également `final` si leur état ne doit pas changer après la construction de l'objet.
Un objet dont tous les champs sont `final` est dit *immutable*.

Les méthodes d'un objet représentent son comportement.
Leur visibilité peut être changée, afin de structurer le code.
Une méthode a un unique type de retour, qui peut être `void` dans le cas où la méthode ne retourne pas de donnée à la suite de son exécution.
Une méthode peut également prendre zéro, un ou plusieurs paramètres.
Le nombre de lignes d'une méthode doit être raisonnable afin que sa compréhension puisse se faire rapidement.

1.7.2. Constructeurs

Listing 8. Fichier Cat.java

```
public class Cat {
    public final String name; ①

    public Cat(String name) { ②
        this.name = name; ③
    }
}
```

① Ici le champ est `public`, mais `final`, donc il n'est pas modifiable une fois l'objet créé

② un constructeur prenant un paramètre de type String

③ Assignation de la valeur du paramètre `name` au champs `name` de la classe `Cat`

Un constructeur est une méthode particulière qui n'a pas de type de retour et dont le nom doit scrupuleusement être le même que celui de la classe dans laquelle il est déclaré.

Le constructeur est, comme son nom l'indique appelé à la construction de l'objet.

Pour construire un objet on utilise le mot clé `new`.
Par exemple :

Listing 9. Fichier Launcher.java

```
public class Launcher {
    public static void main(String[] args) {
        Cat myCat = new Cat("Georges");

        System.out.println(myCat.name);
    }
}
```

Une classe peut avoir autant de constructeurs qu'on le souhaite.

Une classe qui ne déclare aucun constructeur explicitement possède un *constructeur par défaut*.
Le constructeur par défaut ne prend aucun paramètre et ne fait rien.
À partir du moment où un constructeur est déclaré explicitement, le constructeur par défaut n'est plus disponible.

1.7.3. Visibilité

La visibilité est un mécanisme qui permet à une classe, un champ, ou une méthode d'être accessible ou non à d'autres entités.

Il existe 4 visibilités en Java

- `public` : accessible à tous
- `private` : accessible uniquement au sein de la classe qui déclare le composant
- `protected` : accessible aux classes qui étendent la classe qui contient le composant ou aux classes qui se trouvent dans le même `package`.
- la visibilité par défaut, dite aussi ***package protected***, quand aucun modificateur de visibilité n'est précisé.
  Le composant est question est alors accessible aux classes se trouvant dans le même package.

Quand on conçoit un programme orienté objet, on va regrouper dans un même package les objets du même domaine, et leurs interactions spécifiques à ce domaine seront ***package protected***.
Les comportements intrinsèques aux objets de ce domaine seront `private`, alors que l'API (Application Programming Interface) accessible au reste du programme sera `public`.

1.8. Concevoir un objet

Un objet doit (dans la majorité des cas) être construit de telle sorte qu'il n'expose pas *à l'extérieur* la façon dont il représente son état.

Tout l'enjeu de la programmation orientée objet est de réduire le couplage entre les concepts pour simplifier la maintenance, l'évolution et la testabilité du code.

Un mauvais exemple :

Listing 10. Fichier TrafficLight.java

```
class TrafficLight {

    private int color; ①

    public void setColor(int newColor) { ②
        this.color = newColor;
    }

    public int getColor() {
        return color;
    }
}
```

① donnée privée, propre à l'objet

② méthode publique permettant de changer la "couleur" du feu

Ici la classe représentant le feu tricolore expose la façon dont elle stocke ses données, et elle ne contient aucune logique.
Un tel objet est dit *anémique*, car il n'a aucun comportement propre et est considéré dans la majorité des cas comme une mauvaise pratique (code smell).
Un autre objet qui utilise cette classe devra lui aussi changer si le type du champ `color` (<1>) change.

Un meilleur design pourrait être :

Listing 11. Fichier TrafficLight.java

```
class TrafficLight {

    private int color;

    public Color nextState() {
        color = (color + 1) % 3;
        return Color.values()[color];
    }

    public enum Color {
        GREEN,
        ORANGE,
        RED,
    }
}
```

Ainsi le "contrat", c'est-à-dire la partie publique de la classe, ne dépend pas de la façon dont l'état est stocké en mémoire, ici avec un `int`.
Par ailleurs, la logique du feu est codée dans l'objet, rendant impossible les cas qui l'étaient avec l'implémentation précédente :

- `trafficLight.setColor(4)`, mais que veut dire la valeur 4 ?

- passage du vert au rouge ou du rouge à l'orange

1.9. Comparer des objets

En Java, l'opérateur `==` permet de comparer que deux objets ont bien la même adresse mémoire.

Cependant, dans la majorité de cas, il est nécessaire de comparer si deux objets ont la même valeur.
Dans ce cas, on utilisera la méthode `equals`.
Cette méthode est déclarée sur la classe `java.lang.Object` dont tous les objets héritent implicitement.
Par défaut le comportement de cette méthode est d'utiliser l'opérateur `==`, mais elle est surchargeable !

Listing 12. Fichier Cat.java

```
public class Cat {

    private final String name;
    private final int color;

    public Cat(String name, int color) {
        this.name = name;
        this.color = color;
    }

    @Override
    public boolean equals(Object o) { ①
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Cat that = (Cat) o;
        return color == that.color && Objects.equals(name, that.name);
    }

    @Override
    public int hashCode() { ②
        return Objects.hash(name, color);
    }
}
```

① La surcharge de la méthode `equals` pour un objet de type `Cat` va retourner `true` pour tout paramètre qui est un objet de type `Cat` également et dont la couleur (`color`) et le nom (`name`) sont les mêmes.

② La méthode `equals` est toujours définie avec la méthode `hashCode`.
La méthode hashCode est utilisée dans plusieurs algorithmes liés à l'identité, notamment dans les *collections* et on considère que son comportement doit être cohérent avec celui de la méthode `equals`.
C'est-à-dire que deux objets qui sont égaux au sens de la méthode `equals` doivent avoir le même `hashCode`, la réciproque n'est pas vrai, deux objets ayant le même `hashCode` ne sont pas forcément égaux (dans ce cas on parle de collision).

1.10. Enum

Les types énumérés sont des classes dont les instances possibles sont limitées et uniques.
Il n'est pas possible de créer de nouvelles instances d'un type énuméré avec le mot-clé `new`.
Les valeurs d'un type énuméré peuvent être assimilées à des constantes et accédées de la même façon.

Listing 13. Fichier FacialHairStyle.java

```
enum FacialHairStyle { ①

    BEARDED, ②
    MUSTACHE,
    BOLD,
    ;

③

    public static boolean isBold(FacialHairStyle hairStyle) {
        return FacialHairStyle.BOLD == hairStyle; ③
    }
}
```

① la structure d'un type énuméré est proche de celle d'une classe, mais on remplace le mot-clé `class` par `enum`

② le contenu d'un enum commence toujours par la liste des différentes valeurs possibles, séparées par des virgules `,` et se terminant par un point-virgule `;`

③ la suite du contenu est la même que pour les classes, champs, constructeurs et méthodes peuvent être ajoutés

Un enum peut avoir un constructeur, quand les valeurs de l'enum sont associées à de la donnée.
Cependant le constructeur d'un enum est implicitement `protected` et ne peut pas être préfixé par le mot-clé `public`.

Listing 14. Fichier FacialHairStyle.java

```
public enum Environment {

    DEV("http://localhost:9876/my-app", ZoneOffset.systemDefault()), ①
    TEST("https://beta.mydomain.com/", ZoneOffset.UTC),
    PROD("https://app.mydomain.com/", ZoneOffset.UTC),
    ;

    public final String baseUrl;
    public final ZoneId zoneId;

    Environment(String baseUrl, ZoneId zoneId) { ②
        this.baseUrl = baseUrl;
        this.zoneId = zoneId;
    }
}
```

① l'utilisation du constructeur se fait par l'ajout de paramètres entre parenthèses après chaque valeur

② le constructeur s'écrit comme celui d'une classe

Un enum peut implémenter une interface, mais ne peut pas étendre une classe abstraite.

Par ailleurs un enum est implicitement `final` et ne peut pas être étendu.

1.11. Record

Un record permet de décrire de manière concise une classe *anémique* (sans comportement) et immutable.
Ainsi les méthodes `equals`, `hashCode`, `toString` ainsi que les accesseurs sont générés de manière à refléter les paramètres du record.

Listing 15. Fichier LocalTemperature.java

```
record LocalTemperature(
    double temperature,
    double latitude,
    double longitude){}
```

Listing 16. Fichier Launcher.java

```
public class Launcher {
    public static void main(String[] args) {
        var t1 = new LocalTemperature(12.3D, 48.8320315D, 2.2277601D);

        System.out.println(t1.temperature()); ①

        var temperatureList = Set.of(t1);
        System.out.println(temperatureList.contains(new LocalTemperature(12.3D,
48.8320315D, 2.2277601D))); // Displays true ②

        System.out.println(t1); // Displays LocalTemperature[temperature=12.3,
latitude=48.8320315, longitude=2.2277601] ③
    }
}
```

① utilisation d'un des accesseurs générés

② utilisation des méthodes générées `equals` et `hashCode` par l'algorithme du `HashSet` (complexité en O(1))

③ utilisation de la méthode générée `toString`

2. Le langage Java, deuxième partie

2.1. Formes conditionnelles

2.1.1. if / else if / else

Le mot clé `if` permet d'exécuter de manière conditionnelle un bloc de code.
Celui-ci prend en paramètre une expression dont le résultat est de type `boolean`.

On peut chainer les exécutions conditionnelles avec le mot clé `else`.

Par exemple :

```
if (a == 1) {
    // ... ①
} else if (b == 2) {
    // ... ②
} else {
    // ... ③
}
```

① ce bloc sera exécuté si `a` vaut 1

② ce bloc sera exécuté si `a` ne vaut pas 1 **et** que `b` vaut 2

③ ce bloc sera exécuté si `a` ne vaut pas 1 **et** que `b` ne vaut pas 2

2.1.2. switch

Dans le cas d'une condition determinée sur la base d'une seule variable, on peut utiliser les mots-clés `switch` et `case` :

```
switch(a) {
    case 1:
        // ... ①
        break; ②
    case 45:
        // ... ③
    default:
       // ... ④
}
```

① ce bloc sera exécuté si `a` vaut 1

② mot-clé permettant de sortir du `switch`

③ ce bloc sera exécuté si `a` vaut 45

④ ce bloc sera exécuté si `a` ne vaut pas 1

Le problème de cette écriture est qu'il est facile d'oublier le mot-clé `break` et d'introduire un bug dans lequel plusieurs branches seront exécutées.
Dans notre exemple, si la valeur de `a` est 45, le `case` correspondant sera exécuté, ainsi que le bloc `default`.

Pour remédier à cela il existe une seconde écriture, dite *expression switch* :

```
String label = switch (experience) {
    case 1, 2 -> {
        System.out.println("case 1 and 2");
        yield "Beginner and junior";
    }
    case 3, 4 -> "Experienced and senior";
    case 5    -> "Expert";
    default   -> throw new IllegalArgumentException("invalid value");
};
```

Avec cette syntaxe, pas de `break`, c'est la valeur à droite de la flèche `→` qui est retournée, il n'y a donc plus d'ambiguïté.
Dans le cas où la logique prend plus d'une expression, on indique la valeur à retourner avec le mot-clé `yield`.

2.1.3. Opérateur ternaire

L'opérateur ternaire permet d'écrire un bloc `if` à deux branches de manière simple :
`int a = "toto".equals(b) ? 3 : 0;`
est équivalent à

```
int a;
if ("toto".equals(b)) {
    a = 3;
} else {
    a = 0;
}
```

2.2. Boucles

2.2.1. for

Une boucle `for` est une boucle dans laquelle on accède à l'index de l'itération courante.

```
for (int index = 1; index < 4; index++) { ①
    System.out.println(index); ②
}
```

① déclaration d'une variable `index` qui prendra les valeurs de 1 à 4 exclus avec un pas de 1

② cette ligne sera exécutée 3 fois avec les valeurs d'`index` 1, 2 et 3

2.2.2. for each

Les boucles `for` peuvent être utilisée pour parcourir les élèments d'un tableau ou d'une *collection*.
Ainsi on peut écrire :

```
String[] names = new String[] {"Jake", "Rosa", "Raymond", "Gina"};
for (int index = 1; index < names.length; index++) {
    String name = names[index];
    System.out.println(name.toLowerCase());
}
```

Ici l'index n'est pas utilisé directement, c'est un moyen d'accéder à chacune des valeurs du tableau.
Il est plus idiomatique dans ce cas d'utiliser un boucle *for each* :

```
String[] names = new String[] {"Jake", "Rosa", "Raymond", "Gina"};
for (String name : names) { ①
    System.out.println(name.toLowerCase());
}
```

① la variable `name` va pointer successivement sur tous les élèments du tableau

On peut mettre dans l'opérande de droite d'un *for each* un tableau ou tout objet implémentant l'interface `java.lang.Iterable`.
Cette interface est l'expression minimale dont a besoin un ensemble d'objets (telle qu'une liste) pour être parcouru.

2.2.3. while

Les boucles `while` permettent d'exécuter un bloc de code tant qu'une expression est *vraie*.

Listing 17. Fichier TcpServer.java

```
public class TcpServer {

    private boolean acceptingConnection = false;

    public void syncStart() throws IOException {
        ServerSocket serverSocket = new ServerSocket(9876);
        acceptingConnection = true;
        while (acceptingConnection) { ①
            Socket socket = serverSocket.accept();
            // ... handle the socket
        }
    }

    public void stop() {
        acceptingConnection = false;
    }
}
```

① la boucle va recommencer tant que la variable `acceptingConnection` sera *vraie*.

2.2.4. do while

Dans le cas où l'évaluation de la condition nécessite d'être faite à l'issue de l'exécution du bloc, il est possible d'utiliser une boucle `do while`.

Listing 18. Fichier TcpServer.java

```
public class MeteoWebServiceCaller {

    private final MeteoHttpClient client;

    public MeteoWebServiceCaller(MeteoHttpClient client) {
        this.client = client;
    }

    public double getTemperatureWithRetry() throws IOException {
        NetworkFailureException error = null; ①
        int tryCount = 0;
        do {
            try {
                return client.getTemperature(); ②
            } catch (IOException e) { ③
                error = e;
                tryCount++;
            }
        } while (error != null && tryCount < 4); ④
        throw error; ⑤
    }
}
```

① la variable `error` est initialisée avec `null`, c'est-à-dire qu'elle ne pointe vers aucune adresse mémoire

② si la méthode `getTemperature` ne renvoie pas d'erreur, on sort de la méthode `callWebService` en retournant le résultat

③ si la méthode `getTemperature` renvoie une erreur de type `IOException`, alors on affecte l'erreur à la variable `error` et on incrémente la valeur de la variable `tryCount` de 1

④ le bloc `do` est recommencé s'il y une erreur et que le compteur `tryCount` est inférieur à 4

⑤ si l'exécution arrive ici, c'est qu'il y a eu une erreur, et on lance la dernière stockée dans la variable `error`

2.2.5. break et continue

Dans toutes les boucles il est possible d'utiliser les mots-clés `break` et `continue`.

L'instruction `break` permet de sortir immédiatement de la boucle.

L'instruction `continue` permet de stopper l'exécution de l'itération courante et de passer à la prochaine, s'il y en a une.

2.3. Exceptions

Les exceptions en Java sont une des deux formes de retour d'une méthode.
Celle-ci peut se terminer en succès et renvoyer une donnée (rien si sont type de retour est `void`) ou

lancer une erreur.

Une erreur remonte la pile d'appel jusqu'à être interceptée.
Si elle n'est jamais interceptée, elle provoque l'arrêt du thread.
Dans le cas du thread principal (découlant de la fonction `main`), c'est l'application qui s'arrête.

Pour lancer une erreur, on utilise le mot-clé `throw`.
Le mot-clé `throws` (avec un `s`) lui permet de déclarer qu'une méthode est susceptible de lancer un certain nombre d'erreurs.
Par exemple :

Listing 19. Fichier MathUtils.java

```
public class MathUtils {
    public int divide(int dividend, int divisor) throws IllegalArgumentException { ①
        if (divisor == 0) {
            throw new IllegalArgumentException("Cannot divide by 0"); ②
        }
        return dividend / divisor;
    }
}
```

① la méthode `divide` déclare qu'elle peut lancer une erreur de type `IllegalArgumentException`

② si le second paramètre de la méthode est 0, une erreur est lancée

Pour intercepter des erreurs, on utilise un bloc `try catch`.
Par exemple :

Listing 20. Fichier Launcher.java

```
public class Launcher {
    public static void main(String[] args) {
        int dividend = Integer.parseInt(args[0]);
        int divisor = Integer.parseInt(args[1]);
        try { ①
            int result = new MathUtils().divide(dividend, divisor);
            System.out.println(result); ②
        } catch (IllegalArgumentException e) { ③
            System.out.println("An error has occurred: " + e.getMessage()); ④
        }
    }
}
```

① déclaration d'un bloc `try`

② cette ligne n'est pas exécutée si une erreur est lancée par la ligne précédente

③ intercepte les erreurs de type `IllegalArgumentException` lancées dans le bloc `try` associé

④ accès au message de l'erreur, on l'occurence : `Cannot divide by 0`;

Toutes les erreurs pouvant être *lancées* implémentent l'interface `java.lang.Throwable`.

Cette interface est implementée par 3 classes majeures, qui spécialisent leurs classes enfants :

- `java.lang.Error` : les erreurs graves qui sont du ressort de la JVM et non de l'application. Il est conseillé de ne pas intercepter ces erreurs
- `java.lang.Exception` : exceptions dites *checked*. Il s'agit d'erreurs applicatives dont la possibilité doit être déclarée par la méthode. Cette déclaration se fait au niveau de la signature de la méthode au moyen du mot-clé `throws`. La compilation échouera si une méthode ne déclare pas une exception mais que le code à l'intérieur est susceptible de la produire. Exemple d'une de ces erreurs : `java.io.IOException` témoignant d'un problème IO (entrée / sortie), lecture d'un fichier impossible, erreur réseau, etc.
- `java.lang.RuntimeException` : exceptions dites *unchecked*. Il s'agit d'erreurs applicatives dont la possibilité peut ne pas être déclarée par la méthode. Il est cependant recommandé de documenter cette possibilité en rajoutant l'exception dans la signature de la méthode. Exemple d'une de ces erreurs : `java.lang.IllegalArgumentException` témoignant du mauvais usage d'une méthode.

2.4. Interfaces

Les interfaces sont des contrats d'objet composables.
Ce contrat contient des signatures de méthodes qu'un objet concret doit définir s'il l'implémente.
Ainsi une interface n'a ni état, ni méthodes concrètes.
Les champs d'une interface sont implicitement `public`, `static` et `final`, c'est-à-dire qu'il s'agit de constantes.
Toutes les méthodes abstraites sont quant à elles implicitement `public`.
On ne peut pas instancier une interface, mais on peut utiliser l'utiliser comme type de champ, de paramètre ou de variable.

Les interfaces permettent d'abstraire l'utilisation de l'implémentation.

L'intérêt de cette abstraction est de pouvoir substituer une implémentation par une autre sans avoir à changer le code qui l'utilise.
Par exemple :

Listing 21. Fichier Animal.java

```
public interface Animal {
    String name();

    String makeACry();

    FeedingStatus feed(String foodType);

    enum FeedingStatus { ①
        ACCEPTED,
        REFUSED,
    }
}
```

① un type déclaré dans une interface sera implicitement `public` et `static`

Listing 22. Fichier Lion.java

```
public class Lion implements Animal {
    @Override ①
    public String name() { ②
        return "Lion";
    }

    @Override
    public String makeACry() {
        return "Groarrrr";
    }

    public FeedingStatus feed(String foodType) {
        return "meat".equals(foodType) ? FeedingStatus.ACCEPTED : FeedingStatus
.REFUSED;
    }
}
```

① annotation précisant qu'il s'agit d'une surcharge, optionnelle

② le mot-clé `public` est nécessaire ici, il n'est pas implicite dans une classe

Listing 23. Fichier Cow.java

```
public class Cow implements Animal {
    @Override
    public String name() {
        return "Cow";
    }

    public String makeACry() {
        return "Meuuuuh";
    }

    public FeedingStatus feed(String foodType) {
        return "grass".equals(foodType) ? FeedingStatus.ACCEPTED : FeedingStatus
.REFUSED;
    }
}
```

Listing 24. Fichier Launcher.java

```
public class Launcher {
    public static void main(String[] args){
        List<Animal> animals = List.of(
            new Lion(),
            new Cow()
        );

        String foodType = "meat";
        for (Animal animal : animals) { ①
            System.out.println("The" + animal.name() + " makes " + animal.cry());
            final String eatSentence;
            if (FeedingStatus.ACCEPTED == animal.feed(foodType)) {
                eatSentence = "eats";
            } else {
                eatSentence = "refuses to eat";
            }
            System.out.println("It " + eatSentence + " " + foodType);
        }
    }
}
```

① seul le concept d'`Animal` est manipulé ici, nous garantissant que tous les objets implémentant cette interface ont les méthodes `name`, `cry` et `feed`

2.4.1. Méthodes concrètes

Une interface peut posséder des méthodes *concrètes* statiques, souvent utilisées comme méthodes utilitaires.

Une interface peut également posséder des méthodes *concrètes* par défaut.
Il s'agit la plupart du temps de comportements reposant sur d'autres méthodes abstraites permettant d'apporter une fonctionnalité de manière transverse à toutes les classes implémentant cette interface.

L'intérêt de cette approche est que cet ajout de fonctionnalité, contrairement à l'utilisation d'une classe abstraite, garde une caractéristique principale des interfaces : la composition.

En effet, un objet peut implémenter plusieurs interfaces.

Listing 25. Fichier Animal.java

```
public interface Animal {
    String name();

    default String formattedName() { ①
        return name().substring(0, 1).toUpperCase() + name().substring(1).
toLowerCase();
    }
}
```

① méthode disponible sur tous les objets implémentant cette interface, peu importe la manière dont la méthode abstraite `name` est implémentée

2.5. Classes abstraites

Si une classe peut implémenter plusieurs interfaces, elle ne peut hériter que d'une seule classe parente.
On parle alors d'héritage, et l'héritage multiple n'existe pas en Java.
On oppose le concept de composition, plus souple, au concept d'héritage, souvent décrié car peu évolutif.

Une classe concrète peut hériter d'une classe abstraite en implémentant toutes ses méthodes abstraites.
Une classe abstraite peut donc avoir des méthodes concrètes, mais aussi des méthodes abstraites, à l'instar d'une interface.

Par exemple :

Listing 26. Fichier Animal.java

```
public abstract class Animal {
    protected final String name;

    protected Animal(String name) {
        this.name = name;
    }

    public String formattedName() {
        return name.substring(0, 1).toUpperCase() + name.substring(1).toLowerCase();
    }

    public abstract FeedingStatus feed(String foodType);
}
```

3. Le langage Java, troisième partie

3.1. Généricité

Les types génériques sont un moyen d'exprimer une relation entre un type et les types utilisés par son état, les sous-types.
Les collections sont des types génériques qui permettent d'exprimer le type des éléments qu'elles peuvent contenir, indépendamment de leur fonctionnement.
On peut donc avoir une liste de `String`, d'`Integer`, ou de n'importe quel type d'objet.

L'intérêt d'une telle écriture est de lier un (ou plusieurs) sous-type à une instance.
Ainsi en créant une liste de `String`, on ne pourra y ajouter que des `String`.

Déclarer un type générique se fait grâce aux chevrons : `List<String>`.
Quand le ou les sous-types sont inférables par le compilateur, ils peuvent être omis en ne gardant que les chevrons : `List<String> names = new ArrayList<>()`.
Cette écriture est appelée *diamond operator* en référence à la forme des deux chevrons `<>`.

Il est tout à fait possible de déclarer ses propres types génériques :

Listing 27. Fichier Item.java

```
public interface Item {
    String name();
    double price();
}
```

Listing 28. Fichier Basket.java

```
public interface Basket<I extends Item> { ①

    void add(I item); ②

    double totalPrice();
}
```

① l'interface `Basket` définit un sous-type `I` qui doit implémenter l'interface `Item`
`I` est donc un prête-nom pour n'importe quel type implémentant `Item`

② la méthode `add` prend en paramètre un objet de type I, soit n'importe quel objet qui implémente `Item`

Listing 29. Fichier MarketItem.java

```
public interface MarketItem extends Item { ①

    double weight();
}
```

① L'interface `MarketItem` étend l'interface `Item`

Listing 30. Fichier MarketBasket.java

```
public class MarketBasket implements Basket<MarketItem> { ①

    private static final double MAX_WEIGHT_IN_KG = 3.0;

    private final List<MarketItem> items = new ArrayList<>();

    public void add(MarketItem item) { ②
        if (totalWeight() + item.weight() > MAX_WEIGHT_IN_KG) { ③
            throw new MarketBasketFullException("Carry these water bottles yourself
!");
        }
        items.add(item);
    }

    private double totalWeight() {
        return items.stream().mapToDouble(MarketItem::weight).sum();
    }

    public double totalPrice() {
        return items.stream().mapToDouble(MarketItem::price).sum();
    }
}
```

① la classe `MarketBasket` implémente l'interface `Basket` avec comme sous-type `I` de celle-ci l'interface `MarketItem`

② la méthode `add` ne pourra prendre en paramètre que des objets qui implémentent `MarketItem`

③ si le poids du panier avec le nouvel élément dépasse le max, une erreur est lancée

3.2. Collections

À la différence des tableaux, les collections sont des objets, ont des méthodes et la plupart du temps, leur taille peut varier.

Ces structures de données se spécialisent au travers de plusieurs interfaces et classes fournies par la bibliothèque standard, mais toutes implémentent l'interface `java.lang.Iterable`.

Cette interface est tout ce dont a besoin la machine virtuelle pour itérer.
En effet, cette interface comporte principalement une méthode `iterator` renvoyant un objet de type `java.util.Iterator`.
A son tour cette interface a deux méthodes :

- `hasNext` renvoyant un `boolean`
- `next` renvoyant un objet ou lançant une `NoSuchElementException` s'il n'y a plus d'éléments à parcourir

Ainsi en appelant la méthode `Iterable#iterator()`, on récupère un curseur permettant d'itérer sur les éléments d'un ensemble.
Mais rien ne précise si cet ensemble est fini.

Listing 31. Fichier NameUtils.java

```
public class NameUtils {
    public void displayNames(Iterable<String> names) {
        Iterator<String> namesIterator = names.iterator(); ①
        while (namesIterator.hasNext()) { ②
            String name = namesIterator.next(); ③
            System.out.println(name);
        }
    }
}
```

① création d'un nouvel `Iterator`

② tant qu'il y aura un élément *après*, la boucle continuera

③ on fait avancer d'un cran le curseur en lui demandant de renvoyer le prochain élément

C'est l'interface `java.util.Collection` qui apporte, entre autres, la méthode `size` et ainsi spécialise ses sous-types en ensembles d'éléments finis.
Elle apporte également des méthodes pour ajouter, supprimer et chercher des éléments dans l'ensemble.

Les opérations d'ajout, de recherche ou de suppression d'éléments reposent sur les deux fonctions d'identité, `equals` et `hashCode`.
Ainsi tout élément placé dans une collection doit ré-implémenter ces deux méthodes.
Dans le cas contraire, les méthodes des collections ne se comporteront pas comme attendu.

3.2.1. List

La caractéristique principale de l'interface `java.util.List`, est qu'elle garantit que les éléments de l'ensemble sont ordonnés.
Cette interface apporte donc des méthodes pour ajouter ou supprimer des éléments par leur index.

L'implémentation la plus simple est `java.util.ArrayList` et repose sur un stockage des éléments dans un tableau.
Par défaut la taille de ce tableau est de 10.
Si on essaie d'ajouter plus d'éléments que la capacité courante du tableau, la liste créera un nouveau tableau de capacité supérieure et y copiera les éléments du précédent tableau ainsi que les nouveaux.

```
List<Person> people = new ArrayList<>(); ①
people.add(new Person("Martin", 59)); ②
people.add(new Person("Alice", 32));

boolean removed = people.remove(new Person("Jane", 61)); ③

int size = people.size(); // size = 2
```

① La variable `people` de type `List` pointe sur un nouvel objet de type `ArrayList`

② ajout d'un élément dans la liste

③ tentative de suppression d'un élément de la liste, qui retournera `false` car l'élément en est absent

3.2.2. Set

La caractéristique principale de l'interface `java.util.Set`, est qu'elle garantit qu'il n'y a pas de doublons dans l'ensemble.
Ainsi insérer plus d'une fois un même objet dans l'ensemble ne fera rien.

L'implémentation la plus simple est `java.util.HashSet` et repose sur une `java.util.HashMap`, en stockant les éléments en tant que clés de la `Map`.

```
Set<Person> people = new HashSet<>(); ①
people.add(new Person("Martin", 59));
people.add(new Person("Alice", 32));
people.add(new Person("Martin", 59)); ②

boolean removed = people.remove(new Person("Martin", 59)); ③

int size = people.size(); // size = 1
```

① La variable `people` de type `Set` pointe sur un nouvel objet de type `HashSet`

② cet ajout n'aura aucun effet, car l'ensemble contient déjà cet élément

③ suppression d'un élément, qui retournera `true` car l'élément était bien présent

3.2.3. Map

L'interface `java.util.Map` n'implémente pas `java.util.Collection`, ni même `java.lang.Iterable` en Java, mais on la considère néanmoins comme "une collection" du fait de son usage similaire.
Il s'agit d'un ensemble d'associations clé → valeur.

L'implémentation la plus simple est `java.util.HashMap` et repose sur le principe de clé de hachage, calculée grâce à la méthode `hashCode`.
L'état d'une `HashMap` est stocké dans un tableau, de taille 16 par défaut.

Quand on ajoute une paire (clé → valeur), le `hashCode` de la clé est calculé.
Ensuite on applique à cette valeur l'opérateur modulo `%` avec la taille du tableau.

Ainsi on obtient un index compris entre 0 et la taille du tableau (exclue).
À cet index, si la cellule est vide, on insère une liste chaînée (`java.util.LinkedList`) avec comme seul élément la valeur.
Si la cellule n'est pas vide, c'est qu'il y a collision, deux éléments ont un `hashCode` dont le modulo avec la taille du tableau est le même.
Dans ce cas, on ajoute la valeur à la liste chaînée déjà présente.
Quand le nombre d'entrées dans le tableau est supérieur à un certain seuil (le *load factor*, par défaut à 75%), un nouveau tableau du double de la taille précédente est créé et les valeurs y sont redistribuées.

L'intérêt de cette technique est qu'accéder à une valeur par sa clé prendra toujours le même temps, quelle que soit la taille de la `HashMap`.
Pour savoir si un élément est dans une liste, il faut la parcourir jusqu'à tomber sur l'élément en question ou la fin de la liste.
Avec une `HashMap` on calculera le `hashCode` de l'élément à trouver, et en considérant qu'il n'y a pas de collision (en pratique il y en a peu), il suffit d'accéder à l'index du tableau correspondant.

```
Map<String, Person> peopleByName = new HashMap<>();
people.put("Martin", new Person("Martin", 59)); ①
people.put("Alice", new Person("Alice", 32));
people.put("Martin", new Person("Martin", 23)); ②

Person martin = people.get("Martin"); ③

int size = people.size(); // size = 2
```

① ajout d'une valeur de type `Person` associée à une clé de type `String`

② remplacement de la valeur à la clé `"Martin"`

③ accès à une valeur par sa clé

3.3. Streams

Les streams, ou flux, sont inspirés de la programmation fonctionnelle qui tend à transformer des ensembles d'objets en d'autres ensembles d'objets grâce à des *fonctions* simples et composables.

Les streams en Java sont des pipelines de transformation qui implémentent l'interface générique `java.util.stream.Stream`.

Un stream peut être obtenu de plusieurs façons, les plus fréquentes étant :

- à partir de valeurs :

```
Stream<Integer> ages = Stream.of(1, 4, 5, 4568);
```

- à partir d'une `Collection` :

```
Set<String> names = Set.of("Donald", "Daisy");
Stream<String> nameStream = names.stream();
```

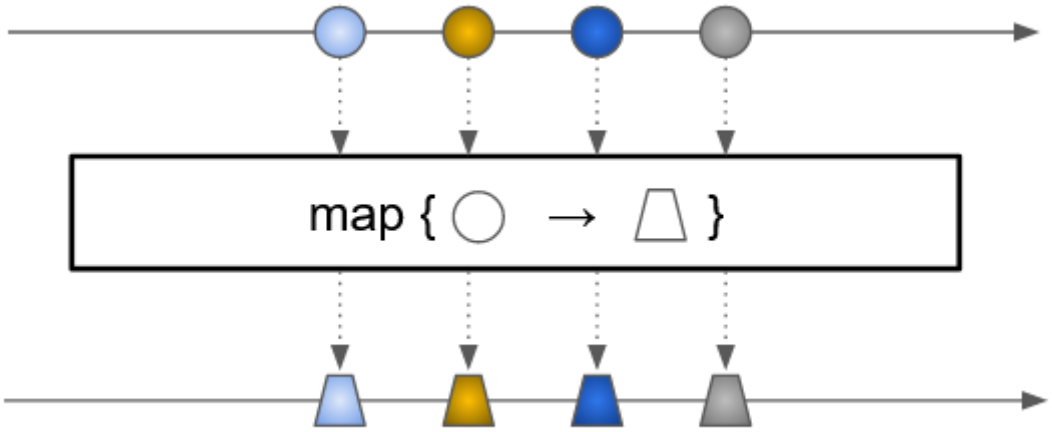
- à partir d'un itérateur

```
List<String> languages = List.of(
        "Java",
        "Kotlin",
        "Scala",
        "Go",
        "Rust");
Iterator<String> languageIterator = languages.iterator();
Stream<String> languageStream = StreamSupport.stream(
        Spliterators.spliteratorUnknownSize(
                languageIterator,
                Spliterator.ORDERED)
        , false);
```

Un `Stream<T>` supporte principalement 3 façons d'appliquer des transformations :

- `map`
  - transforme un ensemble d'objets en un nouvel ensemble de même taille
  - prend en paramètre une `Function<T, U>` transformant un objet de type `T` en un objet de type `U`
  - renvoie un nouveau `Stream<U>` en appliquant la fonction sur chacun des éléments
  - exemple :

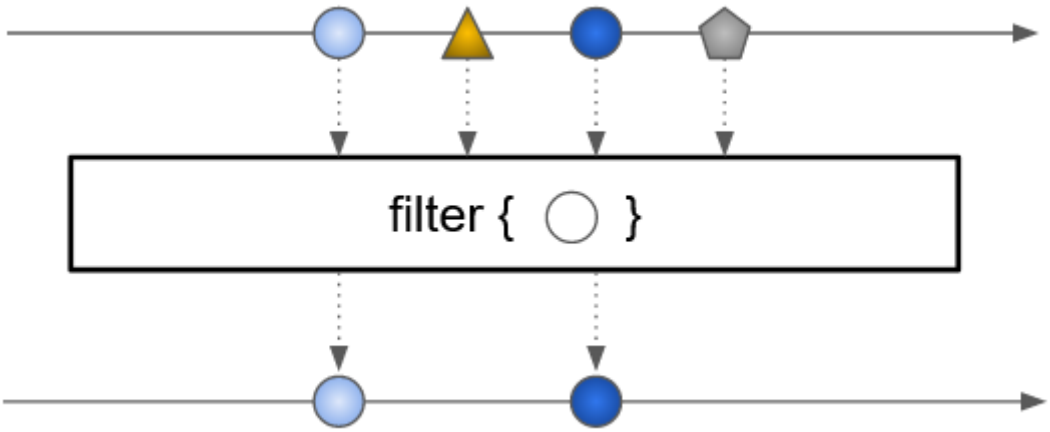
```
Set<Person> people = Set.of(
        new Person("Donald", 32),
        new Person("Daisy", 33)
);
Stream<Integer> ageStream = people.stream().map(p -> p.age);
```



- `filter`
  - transforme un ensemble d'objets en un nouvel ensemble de même taille ou de taille inférieure
  - prend en paramètre un `Predicate<T>` transformant un objet de type `T` en un `boolean`
  - renvoie un nouveau `Stream<T>` en ne gardant que les éléments pour lesquels le prédicat a renvoyé `true`

```
Set<Person> people = Set.of(
        new Person("Donald", 32),
        new Person("Daisy", 33),
        new Person("Riri", 10),
        new Person("Fifi", 11),
        new Person("Loulou", 12)

);
Stream<Person> adultStream = people.stream().filter(p -> p.age > 18);
```

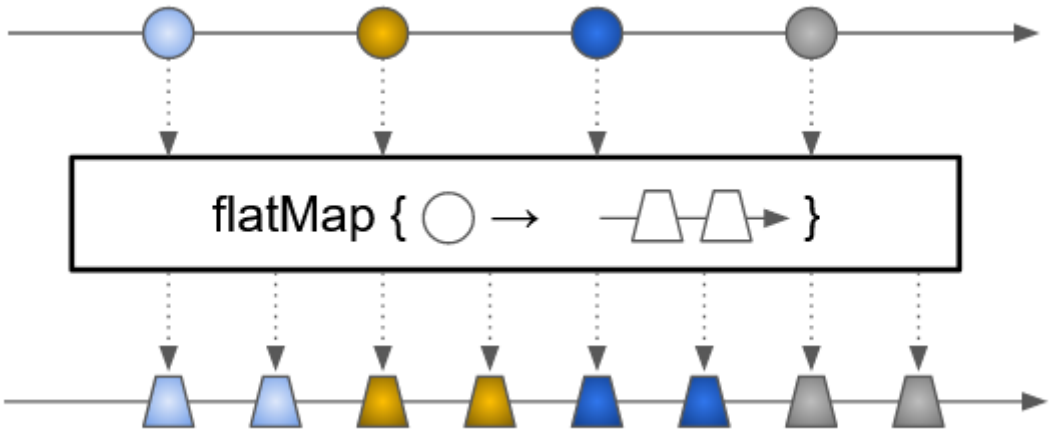


- `flatMap`
  - transforme un ensemble d'objets en un nouvel ensemble, la plupart du temps de taille supérieure

- prend en paramètre une fonction `Function<T, Stream<U>>` transformant un objet de type `T` en un stream d'objets de type `U`
- renvoie un nouveau `Stream<U>` étant la concaténation des streams résultant de l'application de la fonction à chacun des éléments.

```
Set<Student> students = Set.of(
        new Student("Alix", 12.2, 4.0, 14.0),
        new Student("Ilian", 11.3, 18.5, 14.0),
        new Student("Robin", 15.0, 0.0, 16.0)

);
Stream<Double> gradeStream = students.stream().flatMap(s -> s.grades.stream());
```



Afin de transformer un stream en collection, on utilise l'*opération terminale* `collect`.
Cette méthode prend en paramètre un `java.util.stream.Collector` dont les principales implémentations peuvent être construites grâce aux méthodes utilitaires de la classe `java.util.stream.Collectors`.

À noter que les multiples transformations ajoutées sur un stream ne sont exécutées que si nécessaire et uniquement au moment de l'appel d'une opération terminale.

Une *opération terminale*, telle que `count`, `collect`, `reduce`, etc. est une opération dont le retour nécessite l'application des transformations.
Une fois une *opération terminale* appelée sur un stream, celui-ci devient inutilisable.

Par exemple :

```
Set<Person> people = Set.of(
        new Person("Donald", 32),
        new Person("Daisy", 33),
        new Person("Riri", 10),
        new Person("Fifi", 11),
        new Person("Loulou", 12)

);
Set<Person> adultStream = people.stream()
                            .filter(p -> p.age > 18)
                            .collect(Collectors.toSet()); ①
```

① construit un nouveau `HashSet` avec les deux éléments retenus par le prédicat

Ou encore

```
Set<Student> students = Set.of(
        new Student("Alix", "3A", 12.2, 4.0, 14.0),
        new Student("Ilian", "3A", 11.3, 18.5, 14.0),
        new Student("Robin", "4A", 15.0, 0.0, 16.0)

);
double averageGrade = students.stream()
        .filter(s -> "3A".equals(s.group))
        .flatMap(s -> s.grades.stream())
        .collect(Collectors.averagingDouble(Double::doubleValue)); ①
```

① Calcule la moyenne des notes des étudiants du groupe 3A

3.4. Expressions Lambda

En Java tout est objet, y compris les *fonctions*.

Ainsi écrire

```
Predicate<Student> groupPredicate = s -> "4A".equals(s.group);
```

revient à créer une instance d'une implémentation à la volée de l'interface `Predicate`.

C'est fonctionnellement équivalent à :

```
Predicate<Student> groupPredicate = new Predicate<Student>() { ①
    @Override
    public boolean test(Student s) {
        return "4A".equals(s.group);
    }
};
```

① construction d'une classe anonyme, construction à la volée d'une instance d'une classe non nommée et dont l'usage est localisé au bloc où elle est définie

Ou encore à :

Listing 32. Fichier GroupPredicate

```
public class GroupPredicate implements Predicate<Student> {
    @Override
    public boolean test(Student s) {
        return "4A".equals(s.group);
    }
}
```

```
Predicate<Student> groupPredicate = new GroupPredicate();
```

Cette écriture *raccourcie* avec une flèche `->` est appelée *expression lambda* ou *lambda function*.
Le type d'une expression lambda doit être inféré par le compilateur et doit être spécifié au moment de sa création, soit par le type du paramètre d'une méthode, soit par le type d'une variable (comme dans notre précédent exemple).

Le type d'une expression lambda ne peut être qu'une interface à une seule méthode *abstraite*.

Afin de garantir cette spécificité, il est possible d'annoter une interface avec `@FunctionalInterface`.
Annotée de la sorte une interface qui ne possède pas de méthode abstraite ou plus d'une méthode abstraite ne compilera pas.

Il est également possible d'utiliser une référence de méthode comme fonction.
Pour cela, le type de retour, le nombre et le type des paramètres doit correspondre, comme pour une expression lambda.
On utilise l'opérateur `::` pour différencier cette écriture d'un appel de méthode classique.
Par exemple :

```java
public double computeAverageGrade(Collection<Student> students, String group) {
    return students.stream()
        .filter(Objects::nonNull) ①
        .filter(s -> group.equals(s.group))
        .flatMap(s -> s.grades.stream())
        .collect(Collectors.averagingDouble(Double::doubleValue));
}
```

① on ne garde que les éléments non `null` de la collection passée en paramètre de la méthode

Une référence de méthode peut également s'écrire avec une lambda, `.filter(Objects::nonNull)` est équivalent à `.filter(s → Objects.nonNull(s))`.

4. Le langage Java, quatrième partie

4.1. Thread

Un thread ou, fil d'exécution, est le cadre dans lequel s'exécute séquentiellement un programme.
Pour exécuter plusieurs tâches en parallèle, il faut donc créer plusieurs threads.
Un thread (non virtuel) consomme des ressources :

- de la mémoire
- du CPU, ou plus exactement, de la capacité de l'OS à faire plusieurs tâches en parallèle

En effet, il peut (et la plupart du temps il y a) plus de threads que de cœurs de calcul (CPUs).
C'est donc à l'OS d'alterner entre la pause et l'execution des threads.
Plus il y a de threads, et plus l'OS passe de temps (CPU) à coordonner, laissant de moins en moins de place aux threads eux-mêmes.

En Java un thread est modélisé par la classe `java.lang.Thread` et possède un nom.
Quand la JVM démarre, elle va créer (entre autres) le thread **main** qui exécutera la méthode `main`.

Quand un thread n'a plus de travail à exécuter, il s'arrête, et peut être *garbage collecté* si aucune variable ne pointe dessus.

La JVM s'arrête quand il n'y a plus de threads actifs (hors `daemon`).
Dans le cas d'un programme simple qui ne démarre pas de nouveaux threads, une fois la méthode `main` terminée, le thread main s'arrête, il n'y a plus de thread actif, par conséquence la JVM s'arrête.

Dans le cas où des threads (hors `daemon`) ont été créés depuis le thread `main`, la JVM attendra qu'ils se terminent, même si le thread `main` est arrêté.

4.1.1. Créer un nouveau thread

Pour démarrer un nouveau thread, il est possible de créer un nouvel objet de ce type et le démarrer.

```
Runnable action = () -> System.out.println("hello"); ①
Thread t = new Thread(action, "my-super-thread-name");
t.start();
```

① Une tâche à exécuter, implémentation de l'interface fonctionnelle `java.lang.Runnable`

4.1.2. Maitriser sa consommation

Comme on l'a vu, l'utilisation de thread n'est pas anodine pour une application.
C'est pourquoi on choisit la plupart du temps de gérer ses threads au travers d'un mécanisme de recyclage : l'*object pool*.
Ce patron de conception (design pattern) permet de définir une limite ou un comportement à un ensemble de ressources.

Dans notre cas les ressources sont des threads, mais cela fonctionne également avec des connexions à une base de donnée (par exemple).

```
ExecutorService threadPool = Executors.newFixedThreadPool(2); ①
threadPool.submit(() -> System.out.print("hello "));
threadPool.submit(() -> System.out.print("my "));
threadPool.submit(() -> System.out.print("name "));
threadPool.submit(() -> System.out.print("is "));
threadPool.submit(() -> System.out.print("John ")); ②
threadPool.awaitTermination(300L, TimeUnit.MILLISECONDS); ③
threadPool.shutdown(); ④
```

① Création d'un *pool* de 2 threads

② On peut soumettre plus de tâches que de threads, elles seront stockées en mémoire et traitées dès que le *pool* le pourra.
Il n'y a pas de garantie d'ordre, le résultat d'un tel code n'est pas déterministe

③ Attente de la complétion de toutes les tâches envoyées au *pool*

④ Libération des ressources, les 2 threads sont *relâchés* pour être *garbage collectés*

4.1.3. Future

Afin de suivre l'évolution d'une tâche soumise au *pool*, celui-ci renvoie un objet de type `java.util.concurrent.Future`.
Dans le cas d'une tâche avec un type de retour, il est également possible de faire une attente bloquante sur le résultat d'une tâche en particulier.

```
ExecutorService threadPool = Executors.newFixedThreadPool(2);
Future<Double> temperatureFuture = threadPool.submit(() ->
callTemperatureWebSevice());
Future<SunIntensity> sunFuture = threadPool.submit(() -> callSunshineWebSevice()); ①

try {
    Double temperature = temperatureFuture.get(200L, TimeUnit.MILLISECONDS); ②
} catch (InterruptedException e) { ③
    logger.error("Call to temp WS interrupted");
} catch (ExecutionException e) { ④
    logger.warn("Call to temp WS failed", e);
} catch (TimeoutException e) { ⑤
    logger.warn("Call to temp WS timed-out");
}
```

① Les deux appels sont exécutés en parallèle

② Attente bloquante de la complétion de la tâche **temperature**, on cas de succès on récupérera directement la valeur retournée par la méthode s'étant exécutée dans un autre thread

③ Exception survenant si le thread exécutant la tâche est interrompu

④ Exception survenant si une exception est levée par le code de la tâche

⑤ Exception survenant si la tâche n'est pas terminée après le délai spécifié (ici 200 ms)

5. SOLID

Les principes SOLID ont été énoncés par Robert Cecil Martin (Uncle Bob) et leur respect permet un design modulaire dont le principe fondamental est le **découplage** entre les composants.

Ce qu'on entend par découplage est la possibilité de faire un changement (ajout d'une fonctionnalité, correction d'un bug) localisé, sans impacter le reste du code.

- S - Single responsibility : Une responsabilité par classe
- O - Open / Close : ouvert à la composition, fermé à la modification
- L - Liskov substitution : substitution par un sous-type sans modification de la cohérence
- I - Interface segregation : une interface (contrat) différente par client
- D - Dependency inversion : travailler avec la forme la plus abstraite d'un objet

5.1. Exemple pour S et L

Listing 33. Fichier Logger.java

```
public interface Logger {
    void log(Level level, String message);
}
```

Listing 34. Fichier ConsoleLogger.java

```
public class ConsoleLogger implements Logger {
    @Override
    public void log(Level level, String message) {
        System.out.println("[" + level + "] " + message);
    }
}
```

Listing 35. Fichier FileLogger.java

```
public class FileLogger implements Logger {

    private final Path path;

    public FileLogger(Path path) {
        this.path = path;
    }

    @Override
    public void log(Level level, String message) {
        try {
            Files.writeString(path, "[" + level + "] " + message + "\n",
StandardCharsets.UTF_8, CREATE, APPEND);
        } catch (IOException e) {
            throw new UncheckedIOException("Cannot write log message to file: " +
path, e);
        }
    }
}
```

Les implémentations font une chose bien précise, plutôt que d'avoir une unique classe qui gère l'écriture dans la console et dans un fichier avec un `if`.

Par ailleurs, remplacer une implémentation par une autre ne change rien pour le code qui utilise l'interface `Logger`.

5.2. Exemple pour O et D

Listing 36. Fichier CompositeLogger.java

```
public class CompositeLogger implements Logger {

    private final Iterable<Logger> delegates;

    public CompositeLogger(Logger... loggers) {
        this.delegates = Arrays.asList(loggers);
    }

    @Override
    public void log(Level level, String message) {
        delegates.forEach(l -> l.log(level, message));
    }
}
```

Ici pour profiter de fonctionnalités de plusieurs composants, on ne va pas modifier les composants eux-mêmes, mais plutôt les composer.
On pourra par la suite ajouter de nouveaux comportements (logger vers un broker ou une base de

données par exemple) sans modifier, ni les composants existants, ni la logique de composition (ouvert à la composition, fermé à la modification).

Par ailleurs, on utilise la forme la plus abstraite nécessaire, ici l'interface `Logger` plutôt que des implémentations précises.
Ainsi le comportement générique peut s'appliquer à tous les sous-types.

5.3. Exemple pour I

Listing 37. Fichier Vehicule.java

```
public interface Vehicule {

    void startMoving();
}
```

Listing 38. Fichier Container.java

```
public interface Container {

    void addItem(Item item);
}
```

Listing 39. Fichier Car.java

```
public class Car implements Vehicule, Container {

    private final List<Item> trunk;

    @Override
    public void startMoving() {
        startEngine();
        pressAccelerator();
    }

    @Override
    public void addItem(Item item) {
        trunk.add(item);
    }
}
```

La classe `Car` se comporte à la fois comme un véhicule, mais également comme un conteneur, de part sa capacité à stocker des objets dans son coffre.

Cependant les classes intéressées par la capacité d'une voiture à se déplacer ne sont pas forcément les mêmes que celles intéressées par le fait de pouvoir stocker des objets dedans.

On peut donc choisir d'implémenter plusieurs interfaces, chacune en lien avec un domaine

différent, laissant la possibilité au code appelant de travailler avec une version plus abstraite et de décrire des comportements plus génériques.

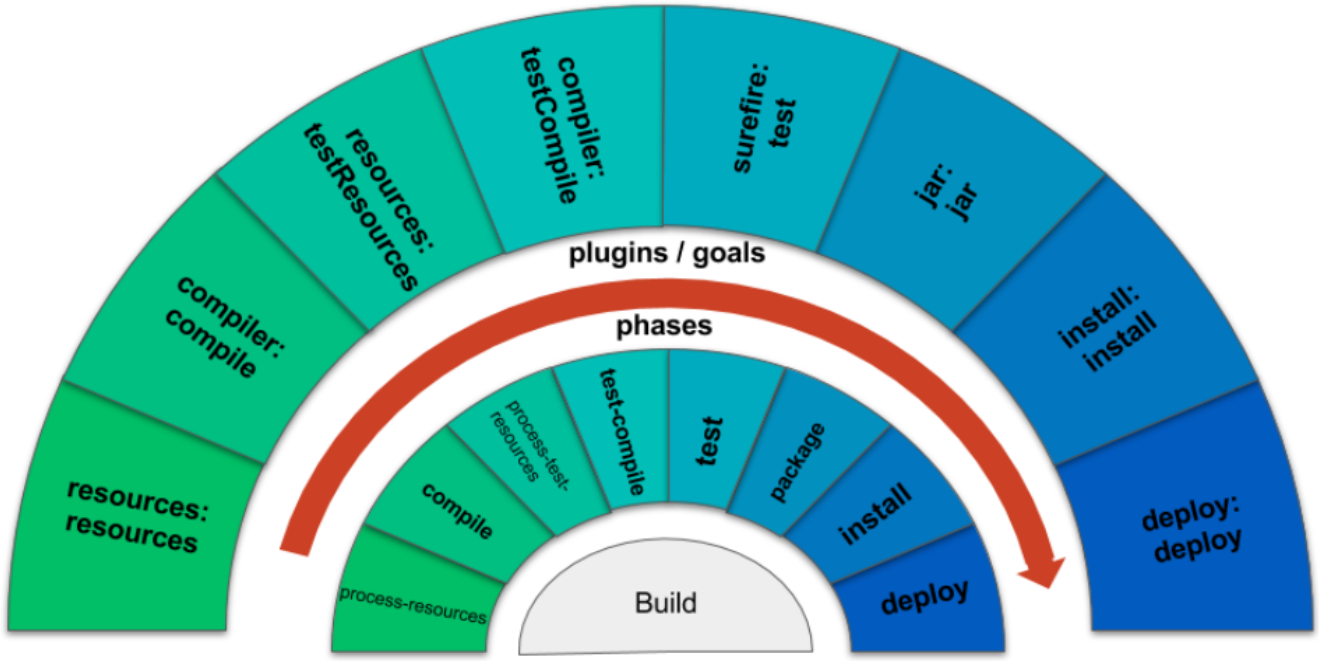
6. Écosystème Java

6.1. Maven

Maven est un outil de construction de projet créé initialement pour Java.
Il permet entre autres de déclarer ses dépendances, compiler le code, construire les binaires et lancer les tests.
C'est aujourd'hui le plus utilisé dans l'écosystème Java.
Il en existe d'autres qui sont moins répandus ou dédiés à un autre langage comme Gradle, SBT, Ivy, Bazel, Make, etc.

Comme beaucoup d'outils de développement (frameworks, intégration continue, IDE, etc.), Maven repose sur une architecture modulaire.
Dans cette architecture, le coeur d'exécution n'apporte que peu de fonctionnalités, mais propose une API pour venir ajouter des fonctionnalités par composition.

Par défaut Maven suit un enchaînement de phases (lifecycle), auxquelles sont associées des plugins par défaut :



Par exemple, associée à la phase **test**, c'est le *goal* **test** du plugin **maven-surefire-plugin** qui est exécuté.
Pour lancer cette phase, on écrira :

```
mvn test
```

Cette commande lancera toutes les phases précédentes, charge aux plugins de ne rien faire si le travail est déjà fait (compilation par exemple).

Il s'agit d'un comportement par défaut.
En effet, Maven fonctionne par *convention* plutôt que par *configuration explicite*.

Même s'il reste possible de configurer Maven pour sortir du comportement par défaut, la plupart des projets préfèrent la simplicité et profitent du même coup d'une structure similaire, ce qui facilite la lecture, et l'utilisation d'outils tiers comme les serveurs d'intégration continue, les solutions SAAS d'analyse statique, etc.

Voici la structure d'un projet Maven :

```
pom.xml ①
src/ ②
|-- main/ ③
|   |-- java/ ④
|       |-- com/
|           |-- mycompany/
|               |-- App.java
|-- test/ ⑤
    |-- java/
        |-- com/
            |-- mycompany/
                |-- AppTests.java
target/ ⑥
```

① Le fichier **pom.xml** décrit toutes les spécificités du projet (coordonnées, scm, dépendances, plugins supplémentaires, etc.)

② Le répertoire **src** contient le code *écrit*

③ le répertoire **main** contient le code de production, le code qui sera embarqué dans les binaires

④ le répertoire **java** contient le code *Java*, il est possible de faire cohabiter plusieurs languages dans des répertoires dédiés.
Par exemple des fichiers *.kt dans un répertoire **kotlin** à côté du répertoire **java**

⑤ le répertoire **test** contient le code de test, ce code ne sera pas embarqué dans les binaires

⑥ le répertoire **target** contient tous les fichiers que Maven va générer, les classes compilées, le résultat des tests, etc.
Ce répertoire est généralement exclu du gestionnaire de code source (**.gitignore** pour Git)

6.1.1. Anatomie d'un fichier POM simple

```xml
<?xml version="1.0"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd"
         xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.mycompany</groupId>
    <artifactId>myproject</artifactId>
    <version>0.0.1-SNAPSHOT</version> ①

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>16</maven.compiler.source>
        <maven.compiler.target>16</maven.compiler.target> ②

        <retrofit.version>1.2.3.4</retrofit.version> ③
        <maven-source-plugin.version>1.2.3.4</maven-source-plugin.version>
    </properties>

    <dependencies>  ④
        <dependency>
            <groupId>com.squareup.retrofit2</groupId>
            <artifactId>retrofit</artifactId>
            <version>${retrofit.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>  ⑤
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-source-plugin</artifactId>
                <version>${maven-source-plugin.version}</version>
                <executions>
                    <execution>
                        <id>attach-sources</id>
                        <goals>
                            <goal>jar-no-fork</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

① groupId, artifactId et version forment les coordonnées uniques d'un projet quand celui-ci est publié dans un dépôt (Maven Central ou autre)

② encodage et version du langage Java permettent de garantir que le code est modifié et compris de la même façon par les différentes parties prenantes (les développeurs et le serveur d'intégration continue)

③ versions des dépendances et plugins utilisés plus bas

④ bloc dans lequel on peut ajouter autant de **dépendances** que l'on souhaite en utilisant leurs coordonnées.
Ici on ajoute une dépendance permettant de modéliser rapidement un client HTTP

⑤ bloc dans lequel on peut ajouter autant de **plugins** que l'on souhaite en utilisant leurs coordonnées.
Ici on ajoute un plugin qui va générer un binaire contenant les *sources* du projet

Les dépendances sont les librairies tierces que l'on souhaite utiliser dans un projet, que ce soit dans le code de production ou le code de test.
Les plugins sont quant à eux des mécaniques supplémentaires que l'on souhaite ajouter au cycle de vie du projet (génération de la documentation, création d'une image docker, analyse statique du code, etc.)

6.2. JUnit

Les tests sont une composante importante de la programmation.
Ils permettent entre autres de :

- vérifier le fonctionnement d'un bloc de code, maintenant et dans le futur
- documenter, en montrant comment le code peut ou doit être utilisé
- rassurer les autres membres d'une équipe de développement sur la qualité du code proposé

Cependant, la librairie standard Java ne fournit pas d'API pour écrire des tests, ni de mécanisme pour les lancer indépendamment du programme.

L'écriture de tests repose donc sur :

- une API fournie par un framework tiers, JUnit est le plus populaire
- un plugin pour le gestionnaire de projet capable d'exécuter le framework, **Surefire** dans le cas de Maven

Ces spécificités sont traduites comme suit dans le fichier **pom.xml** :

```xml
<?xml version="1.0"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd"
         xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <!-- omitted for concision -->

    <properties>
        <!-- omitted for concision -->

        <junit.version>5.7.1</junit.version>
        <assertj.version>3.19.0</assertj.version>
        <maven-surefire-plugin.version>2.22.2</maven-surefire-plugin.version>
    </properties>

    <dependencies>
        <!-- other dependencies can be added here -->

        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter</artifactId>
            <version>${junit.version}</version>
            <scope>test</scope> ①
        </dependency>
        <dependency>
            <groupId>org.assertj</groupId>
            <artifactId>assertj-core</artifactId> ②
            <version>${assertj.version}</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <pluginManagement>
            <plugins>
                <plugin>
                    <artifactId>maven-surefire-plugin</artifactId>
                    <version>${maven-surefire-plugin.version}</version> ③
                </plugin>
            </plugins>
        </pluginManagement>
    </build>
</project>
```

① déclaration de la dépendance JUnit en *scope* **test**, elle ne sera pas disponible pour le code de production (dans **src/main/java**), uniquement pour le code de test (dans **src/test/java**)

② déclaration d'une dépendance permettant d'écrire des vérifications (la plus populaire, mais d'autres existent)

③ surcharge de la version du plugin **Surefire** avec la dernière version, Maven 3 ne prenant pas la dernière version par défaut, et seules les dernières versions sont compatibles avec les dernières versions de JUnit

En Java, les tests sont principalement représentés par des méthodes.
Par défaut, le plugin **Surefire** va rechercher les méthodes de test dans les classes dont le nom fini par `Test`, `Tests` ou `TestCase`.

Pour tester le code suivant :

Listing 40. Fichier src/main/java/com/lernejo/math/MathUtils.java

```
package com.lernejo.math;

public class MathUtils {

    public int fact(int n) {
        if (n < 0) {
            throw new IllegalArgumentException("N cannot be negative");
        }
        return n == 0 ? 1 : n * fact(n - 1);
    }
}
```

On peut écrire cette classe de test :

Listing 41. Fichier src/test/java/com/lernejo/math/MathUtilsTest.java

```
package com.lernejo.math;

import org.assertj.core.api.Assertions; ①
import org.junit.jupiter.api.Test;

class MathUtilsTest {

    private final MathUtils mathUtils = new MathUtils();

    @Test ②
    void fact_of_negative_number_throws() {
        Assertions.assertThatExceptionOfType(IllegalArgumentException.class)
                .isThrownBy(() -> mathUtils.fact(-1))
                .withMessage("N cannot be negative"); ③
    }

    @Test
    void fact_of_3_is_6() {
        int result = mathUtils.fact(3);
        Assertions.assertThat(result).isEqualTo(6); ④
    }
}
```

① *Import* des classes publiques des dépendances de test

② Une méthode de test est *marquée* par une annotation afin de la différencier d'une méthode utilitaire ou interne au test.
Le framework ne lancera que les méthodes identifiées comme des méthodes de test

③ Utilisation de la librairie de vérification pour s'assurer qu'une exception est levée quand on appelle la méthode avec un mauvais paramètre.
On vérifie également le contenu du message d'erreur.

④ Utilisation de la librairie de vérification pour s'assurer que le résultat de `3!` est bien `6`.

Une méthode de test a une structure bien précise :

- **zero**, **une** ou **plusieurs** mises en condition initiale.
  Il s'agit généralement de constituer un jeu de données ou d'amener le système dans un certain état
- **un unique** élément déclencheur.
  Il s'agit de l'appel au bloc de code que l'on souhaite tester.
- **une** ou **plusieurs** vérifications sur l'état de sortie, que ce soit le retour de la méthode testée ou des données accessibles autrement (persistées en base de donnée par exemple)

Dans le cas ou l'on souhaite écrire plusieurs tests similaires à l'exception du jeu de données, il est possible d'écrire des tests paramétrés :

```
@ParameterizedTest ①
@CsvSource({ ②
    "0, 1",
    "1, 1",
    "2, 2",
    "3, 6",
    "4, 24",
    "13, 1932053504"
})
void fact_test_cases(int n, int expectedResult) { ③
    int result = mathUtils.fact(n);
    Assertions.assertThat(result).isEqualTo(expectedResult);
}
```

① Marque la méthode comme test paramétré

② Déclare les jeux de données à utiliser, la méthode sera appelée autant de fois que de jeux de donnée, ici 6 fois

③ Le framework se charge d'appeler la méthode avec les paramètres dans l'ordre où ils ont été déclarés

Conclusion

Ce cours vous a permis de voir un panel des fonctionnalités du langage Java ainsi qu'un aperçu de l'écosystème qui l'entoure.

Il reste bien d'autres choses à explorer, que ce soit dans le langage lui-même, avec l'introspection, les proxys dynamiques, les plugins de compilateur (APT), etc. ou dans l'écosystème qui l'entoure, avec les tests par mutation, la génération de la documentation vivante, l'analyse de code statique et dynamique…

Mais cette histoire sera pour une prochaine fois.

Pour aller plus loin

- La bible de JM Doudoux : https://www.jmdoudoux.fr/accueil_java.htm
- Les vidéos de José Paumard : https://www.youtube.com/c/coursenlignejava/videos
- Une description des types de couplage : https://connascence.io/