Chapitre 3 : Les structures de contrôle

▼ La programmation impérative

Expressions et déclarations

Une **expression** est une ou plusieurs lignes de code qui s'évaluent en une valeur, qui sont assignables. Par exemple :

```
31 + 6
"Assignable"
condition1 == condition2
```

Une **déclaration** est une **instruction** du programme qui va effectuer une **action** sur celui-ci, qui va **altérer son état**. Par exemple :

```
var calcul = 1 + 3;
```

Cette **déclaration** a pour effet **d'assigner** la valeur **4** à la variable **calcul**.

Flot de contrôle et programmation impérative

Le **flot de contrôle** (ou `control flow`) est responsable d'exécuter les **instructions** d'un programme dans un certain ordre.

Pour le faire, on utilise des structures de contrôle pour jouer sur les comportements d'un programme, et donc sur l'ordre des instructions à exécuter.

Pour le faire, on utilise des **structures de contrôle** pour jouer sur les comportements d'un programme, et donc sur l'ordre des **instructions** à exécuter.

Ces principes sont les caractéristiques principales de la `programmation impérative`.

En informatique, la `programmation impérative` est un **paradigme de programmation** qui décrit les opérations en **séquences d'instructions exécutées** par l'ordinateur pour modifier **l'état** du programme.

Voici les **trois** types d'exécution possibles lors du développement d'applications `Java` :

- **l'exécution séquentielle** : les instructions sont exécutées **ligne après ligne**. C'est l'exécution par **défaut**, lorsqu'il n'y a pas de **structures de contrôle**. A noter que les structures de contrôles sont également exécutées **séquentiellement** les unes par rapport aux autres.
- **l'exécution sélective** : des instructions différentes sont exécutées en fonction de conditions. Ce type d'exécution a lieu en présence de structures de contrôle alternatives, appelées également instructions conditionnelles.
  - Concerne les **instructions** `if`, `switch` et `l'opérateur ternaire`.
- **l'exécution itérative** : une ou plusieurs instructions sont **répétées** en utilisant des **boucles**, également appelées **instructions itératives**.

- Concerne les **boucles** `for`, `while`.

Les **structures conditionnelles** et les **boucles** seront décrites dans les leçons suivantes

▼ Les instructions if, else if et else

Les **structures de contrôle conditionnelles** permettent d'exécuter des **bouts de code** en fonction de certaines **conditions**. Pour bien les utiliser, on se base sur les mêmes opérateurs de **comparaison** et **d'égalité** que ceux-vu dans la leçon sur "Les opérateurs" au chapitre 2.

Instruction if

Cette **instruction** ne génère pas de valeur et n'est donc pas assignable à une variable. C'est bien une instruction qui permet **d'influer** sur le **flot de contrôle**.

Sur une ligne

L'instruction `if` sur une seule ligne se définit de la manière suivante :

```
if (age >= 18) System.out.println("Majeur");
```

Si la **condition** est **vraie**, permet l'exécution **d'une instruction** de manière concise.

Avec bloc de code

L'instruction if en bloc de code se définit de la manière suivante :

```
if (age >= 18) {
    System.out.println("Majeur");
    System.out.println("plusieurs instructions autorisées");
}
```

Si la **condition est vrai**, permet l'exécution de **plusieurs instructions**.

Instruction else if

L'instruction `else if` permet d'ajouter des tests similaires au `if` si les conditions des précédents `if` ne sont pas vraies. Il s'utilise de la manière suivante :

```
if (age >= 18) {
    System.out.println("Majeur");
} else if (age < 0) {
    System.out.println("Age non valide");
} else if ... //autant de else if que nécessaire
```

Il est possible **d'imbriquer** autant de `else if` qu'on en a besoin.

> **Attention :** Il faut bien mettre les conditions des `else if` dans un ordre qui ne provoque pas de **bugs** dans l'application. Ca peut arriver **si deux conditions de deux if différents sont vraies en même temps** (on ne rentrera que dans le premier bloc de code). Si on souhaite tester deux conditions qui peuvent être vraies en même temps, il faut alors les **séparer** en deux if différents.

Instruction else

L'instruction `else` permet d'ajouter un c**omportement par défaut** si les conditions des `if` / `else if` ne sont pas vraies. Il s'utilise de la manière suivante :

```
if (age >= 18) {
    System.out.println("Majeur");
  } else {
    System.out.println("Mineur");
}
```

Ici, si aucune **condition** n'est vraie, alors on affiche la chaîne "**Mineur**" dans la console.

Aller plus loin : la saisie utilisateur dans la console

Il est possible de demander à un utilisateur de saisir des données dans la console lors de l'exécution de programmes Java. Voici comment faire :

```
var saisieUtilisateur = new Scanner(System.in);

System.out.println("Saisissez un age :");
var age = saisieUtilisateur.nextInt();

// Il est également possible de demander la saisie d'autres types que le int :
saisieUtilisateur.nextBoolean();
saisieUtilisateur.nextLine();
saisieUtilisateur.nextFloat();
// ...
```

Les objets Scanner et System

`Scanner` est un o**bjet natif Java** qui va nous permettre de **lire** du contenu depuis une **source** : ici `System.in`.

`System.in` fait référence à l'entrée standard utilisateur qui correspond par défaut au terminal.

Les fonctions de saisies de Scanner

Il est ensuite possible d'appeler des fonctions de Scanner pour lire des données provenant de la source (donc ici du terminal) : **nextInt()**, **nextBoolean()**, **nextLine()**, ...

Dans l'exemple précédent, **saisieUtilisateur.nextInt()** essaie de **convertir** l'entrée de l'utilisateur depuis la console en **entier**, et **d'assigner** la valeur de l'entier à la **variable age**.

Attention, si la saisie de l'utilisateur ne correspond pas au type attendu, le programme lancera une **exception**. Nous verrons dans le chapitre sur la **gestion d'erreurs** comment gérer les exceptions dans nos programmes.

Ici, il est simplement important de garder en tête que toute saisie d'un utilisateur peut être source de bugs et qu'il sera important d'apprendre à gérer les potentiels problèmes dans le code.

Import

Pour utiliser `Scanner`, il faut **importer** l'objet de la manière suivante en première ligne du fichier **Main.java** :

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

Contrairement aux objets utilisés jusqu'ici, il est nécessaire de l'importer car il n'est chargé par défaut au lancement de **Java** (`System` est bien chargé par défaut par exemple).

> **Note : Intellij** importe Scanner pour vous si vous cliquer sur `Alt + Entrée` quand une erreur apparait dans l'éditeur

▼ L'instruction switch

Tout comme l'instruction `if`, l'instruction `switch` n'est pas une **expression**, elle n'est pas **assignable**.

Il est possible de l'implémenter de la manière suivante :

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
            System.out.print("Donner un nombre:");
                Scanner clavier = new Scanner(System.in);
                int nb = clavier.nextInt();
                switch(nb){
                case 1 :
                        System.out.println("Lundi");
                        break;
                case 2 :
                        System.out.println("Mardi");
                        break;
                case 3:
                        System.out.println("Mercredi");
                        break;
                default:
                        System.out.println("Invalide");
    }
}
```

switch est appliqué à une variable pour tester le contenu de celle-ci.

L'instruction case

Permet de définir les différentes valeurs possible de la variable testée. Ce sont des **tests d'égalité** uniquement.

L'instruction default

Permet de définir le comportement par défaut du switch si aucune des **conditions d'égalité n'est vraie.**

▼ L'opérateur ternaire

Contrairement à `if` et `switch`, `l'opérateur ternaire` est une **expression** car c'est un **opérateur**. Comme tous les opérateurs, son résultat est assignable dans une variable. Donc, ce n'est pas exactement une structure de contrôle, mais son utilisation est très similaire à l'utilisation d'un `if / else`.

Comparaison avec if / else

Le code `if / else` suivant :

```
var resultat = "";
if (age >= 18) {
    resultat = "Majeur";
} else {
    resultat = "Mineur";
}
System.out.println(resultat);
```

peut être réécrit avec `l'opérateur ternaire` pour obtenir exactement le même résultat :

```
var resultat = age >= 18 ? "Majeur" : "Mineur";
System.out.println(resultat);
```

On voit que le code est ici bien plus concis, `l'opérateur ternaire` est très pertinent quand le teste à réaliser correspond à un `if / else` uniquement.

▼ Introduction aux itérables

Le but de cette leçon est de donner un premier aperçu de l'utilisation des itérables car ceux-ci sont utilisables dans les boucles, que nous allons voir dans les prochaines leçons.

Note : les itérables seront vu plus en détails dans le chapitre : "**Tableaux et collections**".

Les itérables

Une définition très simple d'un `itérable` est : "**qui peut s'itérer**". En programmation, cela correspond à des collections de données que l'on peut **parcourir élément par élément**.

Les **collections** sont incontournables car elles permettent de **regrouper différentes données dans une seule variable.**

> **Note :** Java étant fortement typé, les collections ont également un type. Une collection d'entiers ne peut donc contenir que des entiers.

Dans cette leçon, nous nous intéressons uniquement à ces deux types de collections : les `tableaux` et les `listes`.

Les tableaux

Les **tableaux** permettent de définir des collections de **types natifs** ou **d'objets**.

Leur **taille est fixe** et ne peut pas être changée une fois qu'ils sont déclarés.

Création d'un tableau

Il est possible de définir des tableaux de **deux** manières :

```
int[] tableau = new int[5];
tableau[0] = 2; // 2 dans la position 0 du tableau
tableau[1] = 5; // 5 dans la position 1 du tableau
tableau[5] = 2; // Interdit car dépasse la taille maximum => exception
//Exception ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5

int[] tableau2 = {1, 4, 2, 9}; // Tableau de 4 éléments, plus concis
```

Lecture des valeurs

Il est possible d'accéder aux valeurs d'un tableau :

```
int[] tableau = {1, 4};
System.out.println(tableau[0]);
System.out.println(tableau[1]);
System.out.println(tableau[2]); // Exception car dépasse la taille du tableau
```

Inférence

Pour que l'**inférence** fonctionne à la déclaration d'un tableau, il faut le faire ainsi :

```
var tableau = new int[]{1, 4};
```

Sans préciser le type dans l'**expression** à droite de l'**assignation**, le compilateur ne peut pas le deviner par lui-même.

Les listes

Les `listes` permettent de définir des **collections d'objets uniquement**.

Leur **taille n'est pas fixe**, il est possible d'**ajouter** ou de **supprimer** des éléments à la volée, quand elles sont **mutables** (c'est le cas de la liste de type `ArrayList` que nous allons utiliser).

> **Attention tout de même, les listes immuables existent également.**

Création d'une liste

Exemple de liste mutable :

```
var fruits = new ArrayList<String>();
ou
List<String> fruits = new ArrayList<String>();

fruits.add("Pommes");
fruits.add("Poires");
fruits.add("Fraises")
```

Lecture des valeurs

```
System.out.println(fruits.get(0));
System.out.println(fruits.get(1));
System.out.println(fruits.get(2));
System.out.println(fruits.get(3)); // Exception car dépasse la taille de la liste
```

Import

Pour utiliser `ArrayList`, il faut importer l'objet de la manière suivante en première ligne du fichier Main.java :

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

Contrairement aux objets utilisés jusqu'ici, il est nécessaire de l'importer car il n'est chargé par défaut au lancement de **Java** (**String** est bien chargé par défaut par exemple).

> **Note : Intellij** importe **ArrayList** pour vous si vous cliquer sur `Alt + Entrée` quand une erreur apparait dans l'éditeur.

Aller plus loin : la généricité

Lors de la création des listes, une nouvelle notation a été utilisée : `<>`.

Celle-ci est utilisée pour faire de la `programmation générique` en **Java**. C'est une syntaxe **particulière** qui vous nous permettre de préciser le **type de donnée** qui sera stockée dans la **liste** :

- `new ArrayList<Integer>();`
  - Création une liste d'entiers
- `new ArrayList<Boolean>();`
  - Création une liste de booléens
- `new ArrayList<String>();`
  - Création une liste de chaînes de caractères
- etc

Nous verrons plus en détails le fonctionnement de la **généricité** dans **le chapitre sur la programmation générique**. D'ici là, il est important de comprendre que le type `ArrayList` peut stocker des données de **n'importe quel type**, il faut pour cela préciser le type voulu lors de la création de la liste entre les `<>`.

La boucle for

La boucle `for` permet de **répéter** des **instructions** un **certains nombre de fois** ou même de parcourir des **collections**.

for classique

Pour répéter 10 fois les même instructions :

```
for (int compteur = 0; compteur < 10; compteur ++) {
    System.out.println("Compteur : " + (compteur+1));
}
```

- `int compteur = 0`
  - Une variable est déclarée pour servir de **compteur**
- `compteur < 10`
  - **Condition de sortie** de la boucle for
- `compteur++`
  - **Incrémentation** de la variable compteur à **chaque fin de répétition** de la boucle

De la même manière que les **structures de contrôle conditionnelles**, on définit un bloc de code avec les caractères `{ et }`. Toutes les **instructions** présentes dans le bloc sont répétées par la boucle.

Boucle infinie

Il est possible de définir une boucle **infinie** :

```
for ( ; ; ) {
    System.out.println("Message infini");
}
```

Dans ce cas, la boucle répète indéfiniment les instructions présentes dans le bloc de code.

Boucles imbriquées

Il est possible d'imbriquer autant de `for` qu'on le souhaite :

```
for (int compteur = 0; compteur < 10; compteur++) {
    for (int compteur2 = 0; compteur2 < 10; compteur2++) {
        for (int compteur3 = 0; compteur3 < 10; compteur3++) {
            //...
        }
    }
}
```

Attention, l'imbrication de boucles peut amener très vite à un très grand nombre d'itérations. Ici, le nombre de répétition est `10*10*10=1000`. Mal utilisées, cela peut amener **des problèmes de performances** dans l'application.

for sur itérables

`Java` offre une boucle for améliorée pour rendre le parcours d'itérables le plus simple possible. Peu importe la collection parcourue, l'utilisation du for sera toujours la même.

Parcours de tableaux

Pour parcourir un tableau :

```
String[] villes = {"Dakar", "Thiès", "Saint Louis"};
for (String ville : villes) {
    System.out.println(ville);
}
```

- `String ville`
  - Déclaration d'une variable qui **contiendra** toutes les **valeurs** de l'itérable à parcourir
- `: villes`
  - La variable qui **définit** l'itérable à parcourir

Parcours de listes

Exactement la même boucle que pour le tableau :

```
var villes = new ArrayList<String>();
villes.add("Paris"); villes.add("Lyon"); villes.add("Marseilles");
for (String ville : villes) {
    System.out.println(ville);
}
```

Aller plus loin: La Méthode forEach() de ArrayList

La méthode `forEach()` de **ArrayList** permet d'effectuer certaines opérations pour chaque élément de **ArrayList**.

Cette méthode **parcourt chaque élément de l'Iterable de ArrayList** jusqu'à ce que tous les éléments aient été traités par la méthode ou qu'une exception soit soulevée. L'opération est effectuée dans **l'ordre d'itération** si cet ordre est spécifié par la méthode. Les exceptions soulevées par l'opération sont transmises à l'appelant.

Syntaxe:

```
public void forEach(Consumer<? super E> action)
```

- `Paramètre` : Cette méthode prend un paramètre action qui représente l'action à effectuer pour chaque élément.
- `Retourne` : Cette méthode ne renvoie rien.
- **Exception** : Cette méthode lève l'exception **NullPointerException** si l'action spécifiée est **nulle**.

Exemple d'utilisation:

```
var villes = new ArrayList<String>();
villes.add("Dakar");
villes.add("Thiès");
villes.add("Saint Louis");
villes.forEach((ville) -> System.out.println(ville));
```

▼ La boucle while

La boucle `while` permet de répéter des instructions **tant qu'une condition est vraie**.

Instruction while

Pour répéter 10 fois la même instruction :

```
var compteur = 0;
while (compteur < 10) {
    System.out.println("Compteur : " + compteur);
    compteur++;
}
```

- `compteur < 10`
  - La **condition** de la boucle while
- Le contenu du bloc while est **répété tant que la condition précédente est vraie**. Il est important d'avoir une **action en fin de boucle** qui permet lors d'une itération de rendre l**a condition du while fausse**. Ici, l'instruction **compteur++ permet de rendre la condition fausse après 10 répétitions**.

Boucle infinie

Si **aucune action n'amène à rendre la condition du while fausse**, une boucle infinie comme la suivante est alors exécutée :

```
while (true) {
    System.out.println("Infini");
}
```

Dans ce cas, la boucle répète indéfiniment les instructions présentes dans le bloc de code.

Instruction do while

Cette instruction fonctionne de la même manière que **while** :

```
var compteur = 0;
do {
    System.out.println("Compteur : " + compteur);
    compteur++;
} while (compteur < 10);
```

Les différences :

- La boucle commence par `do` et **n'a pas de condition à ce niveau**. On exécute donc le contenu du bloc **au moins une première fois**.
- le `while` à la fin **vérifie la condition et sort de la boucle si elle est fausse**. Ne pas oublier le `;` à la fin de la ligne qui est nécessaire ici.

Pour résumer, un `do while` exécute **forcément au moins une itération** car la condition de sortie est **vérifiée** à la **fin de la boucle**.

▼ Les instructions continue et break

Dans cette leçon, les **structures de contrôle itératives et conditionnelle**s sont utilisées en même temps pour comprendre l'utilisation de deux nouvelles instructions : `continue` et `break`.

L'instruction continue

L'instruction peut s'utiliser dans les **boucles** comme dans l'exemple suivant :

```
for (int compteur = 0; compteur < 3; compteur++) {
    if (compteur == 2) continue;
```

```
    System.out.println("Compteur : " + compteur);
}
```

Quand continue s'exécute, la boucle passe à l'itération suivante directement sans exécuter le reste du bloc. Voici le résultat de l'exécution :

- La boucle `for` est définie pour itérer 3 fois
  - La première itération affiche `Compteur : 0`
  - La deuxième itération affiche `Compteur : 1`
  - À la troisième itération, `compteur` est égal à `2` : la condition du `if` est **vérifiée**, `continue` est donc **exécuté**. **On passe donc à l'itération suivante** et **aucun affichage n'a lieu** cette fois-ci.

L'instruction break

L'instruction peut s'utiliser également dans le boucles :

```
for (int compteur = 0; compteur < 10; compteur++) {
    if (compteur == 2) break;
    System.out.println("Compteur : " + compteur);
}
```

Quand `break` s'exécute, **on sort de la boucle même si la condition de sortie n'est pas encore vraie.**

Dans l'exemple précédent, même si la **boucle** est définie pour **itérer 10 fois**, on obtient finalement exactement la même exécution que dans l'exemple de l'instruction `continue`.

La raison : `break` est exécuté quand `compteur` est égal à `2`, on **sort** donc de la **boucle** à la **troisième itération**. Le programme n'affiche alors que `Compteur : 0` et `Compteur : 1` et n'exécute pas les 10 itérations malgré la condition de sortie.

Attention aux conditions

Dans les exemples précédents, nous avons vu que les `continue` et `break` étaient utilisés à **l'intérieur** d'un `if`. C'est plutôt logique car sinon, les instructions seraient exécutée dès la **première itération**. On voit donc que ces instructions ont du sens quand elle sont liées à une **structure de contrôle conditionnelle**.

Rappel : switch et break

Dans une leçon précédente, `break` a été utilisé avec l'instruction `switch` :

```
var chiffre = 2;
switch (chiffre) {
    case 1 :
        System.out.println("I");
        break;
    case 2 :
        System.out.println("II");
        break;
    default:
        System.out.println("Invalide");
}
```

Le **but** du `break` dans ce cas-ci est **d'éviter** l'exécution du code de **plusieurs cas en même temps**.