

# *Controlling Program Flow in Java*

Controlling Program Flow in Java (Java SE 11 Developer Certification 1Z0-819)

## Table des matières

Controlling Program Flow in Java .....	1
I. Conditional Logic :.....	3
• Using if-else :.....	3
• Block statement : .....	3
• Chaining if-else : .....	4
• Nested if and more about conditional logic :.....	5
II. The Switch Statement :.....	6
• Introduction : .....	6
• Using switch : .....	6
Branch Control Flow :.....	7
• Branch Ordering : .....	7
• Supported Types : .....	8
• Supported Values : .....	9
• Choosing between Switch and If-else : .....	9
• When to AVOID switch : .....	10
III. Looping.....	11
• Introduction : .....	11
• While loop : .....	11
• Do while loop : .....	11
• For Loop : .....	12
• For Loop Control Variable : .....	12
• For Each Loop : .....	13
• For Each Loop LIMITATIONS : .....	13
IV. Complex Looping and Branching : .....	15
• Nesting Loops and if-else : .....	15
• Nested loops : .....	15

- Branching : ..... 16
- The Return statement :..... 17
- Infinite loops :..... 17

## I. Conditional Logic :

- *Using if-else :*

On voit ici une utilisation classique de la déclaration if-else :

```
if (value1 > value2)
    System.out.println("value 1 is bigger");
else
    System.out.println("value 1 is not bigger");
```

On voit qu'il n'y a **pas de crochet** car cela n'est pas nécessaire si le code qui suit le *if* ou le *else* ne fait qu'une seule ligne (une seule instruction).

Ce à quoi il faut faire attention pour la certification :

1. Présence **obligatoire** de parenthèses entourant la condition.
2. La **condition** doit être un **RESULTAT booléen**. Attention, on peut tout de même avoir une affectation comme condition si elle est booléenne (voir cours précédent).
3. Le else est **optionnel**.

- *Block statement :*

Le « **block statement** » est ce qui permet de regrouper des instructions. Pour cela, en JAVA on utilise les deux crochets (les accolades) qui délimitent un « block ». A l'inverse, Python par exemple est un langage positionnel dont les blocks sont délimités par l'indentation.

Dans cet exemple on voit qu'il y'a un block d'une seule ligne après le « else ». En Python ce serait deux :

```
else
    diff = value2 - value1;
    System.out.println("value 1 is NOT bigger than value 2, diff = " +
diff);
```

La même instruction avec un bloc de deux lignes grâce aux accolades :

```
else {
    diff = value2 - value1;
    System.out.println("value 1 is NOT bigger than value 2, diff = " +
diff);
}
```

Dans le cas où je crée un block if-else, je ne peux pas introduire une instruction indépendante (qui n'est pas compris entre les accolades) après le « if » :

```
if (value1 > value2)
    diff = value1 - value2;
```

```
System.out.println("value 1 is bigger than value 2, diff = " +  
diff); //cette ligne posera problème.  
  
else {  
    diff = value2 - value1;  
}
```

Ces instructions donneront une erreur de compilation. La raison est simple : le « else » n'est pas une instruction **indépendante**, il est une partie **optionnelle** d'un type d'instruction. Pour savoir à quel if il est associé, il examine la ligne qui le précède. Si celle-ci n'est pas un block d'instruction if, il est dissocié et ne fonctionnera pas.

Donc, pour la certification il faut être attentif à :

- **L'indentation** qui ne doit pas vous tromper.
- **Les crochets** qui délimitent des blocks d'instruction.
- 

- *Chaining if-else :*

Cette suite d'instruction permet simplement d'enchaîner plusieurs conditions pour l'exécution d'instructions.

La seule chose à laquelle il faut être attentif dans cet exemple est **l'ordre des instructions**. Dans cet exemple bien qu'on souhaite afficher « Senior adult » car l'âge est celui d'une personne assez âgée, on aura comme résultat « Adult » car la première condition est remplie, donc les autres parties du block if/else ne sont pas exécutées :

```
int age = 70;  
  
if (age > 17) {  
    System.out.println("Adult");  
}  
else if (age > 64) {  
    System.out.println("Senior adult");  
}  
else {  
    System.out.println("Minor");  
}
```

- *Nested if and more about conditional logic :*

Il n'y a pas énormément de chose à retenir dans cette section. Le principal point étant qu'il faut être attentif aux blocks de code. Dans l'exemple suivant on fait un **NESTED IF** (un if niché dans un autre) car l'on veut éviter l'erreur de compilation d'une division par zéro :

```
int students = 0;
int rooms = 4;

if(students > 0)
    if(rooms > 0)
        System.out.println("Students per room: " + students / rooms);
else
    System.out.println("NO students");
```

Ce à quoi il faut être attentif est le bloc else. Dans l'exemple suivant je n'ajoute qu'une paire de crochet, ce qui change l'association du else :

```
if(students > 0) {
    if(rooms > 0)
        System.out.println("Students per room: " + students / rooms);
}
else
    System.out.println("NO students");
```

Dans le premier cas, l'association se fait avec le 2<sup>ème</sup> if, et dans le second cas il se fait avec le 1<sup>er</sup>.

## II. The Switch Statement :

- *Introduction :*

Principe : une valeur est introduite dans une instruction switch qui examine si elle correspond à d'autres valeurs prévues. Il faut une ligne avec le mot-clef **BREAK** à la fin de chaque block case pour signifier qu'il faut quitter la déclaration (« the statement ») switch quand les deux valeurs correspondent.

On peut optionnellement ajouter une ligne « **DEFAULT** » si on veut quand même une instruction dans le cas où la variable testée ne correspond à aucun cas.

```
switch (test-value) {  
    case match-1:  
        statements  
        break;  
    .  
    .  
    .  
    case match-N:  
        statements  
        break;  
    default:  
        statements  
}
```

- *Using switch :*

Pas grand-chose à retenir si ce n'est :

- La valeur testée doit être entre parenthèse.
- Le corps de la déclaration switch doit être entre accolade.
- Le mot clef case indique la valeur à comparer qui doit être **UNIQUE** au sein de la déclaration.

```
char sign = '-';  
switch(sign) {  
    case '+':  
        System.out.println("Positive");  
        break;  
    case '-':  
        System.out.println("Negative");  
        break;  
    default:  
        System.out.println("Sign not recognized");  
}
```

## *Branch Control Flow :*

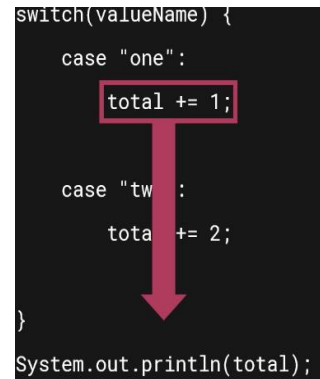
On peut examiner le fonctionnement du Switch par un exemple :

```
int total = 10;
switch(valueName) {
    case "one":
        total += 1;
    case "two":
        total += 2;
} //13
```

Le résultat sera ici **13** car une fois qu'on a une correspondance, s'il n'y pas de break, toutes les lignes suivantes seront exécutées jusqu'à ce qu'on atteigne l'exit du If.

C'est pour cela qu'on utilise le mot clef **BREAK** qui indique qu'on sort de la déclaration IF.

```
int total = 10;
switch(valueName) {
    case "one":
        total += 1;
        break;
    case "two":
        total += 2;
        break;
}
```



```
switch(valueName) {
    case "one":
        total += 1;
    case "two":
        total += 2;
}
System.out.println(total);
```

Ici le résultat sera de 11 grâce au BREAK.

- *Branch Ordering :*

Ce qu'il faut savoir est que l'ordre des instructions CASE n'as pas d'importance. On peut placer les valeurs à comparer dans n'importe quel CASE, ça ne change rien tant qu'elles sont uniques.

On peut aussi déplacer le DEFAULT et le mettre dans n'importe quelle position. Cela ne se fait pas car on perd en lisibilité mais le résultat sera tout aussi **VALIDE** :

```
int sum = 0;
switch(operation) {
    case '+':
        sum++;
        break;
    default:
        System.out.println("Current sum: " + sum);
        break;
    case '-':
        System.out.println("Subtraction no allowed");
        break;
}
```

L'ordre des branches n'a PAS d'impact tant qu'elles se terminent par un BREAK.

Par exemple dans le cas suivant l'ordre à de l'importance car il y'a l'absence d'un *break* :

```
int sum = 0;
switch(operation) {
    case '+':
        sum++; //pas de break
    default:
        System.out.println("Current sum: " + sum);
        break;
    case '-':
        System.out.println("Subtraction no allowed");
        break;
}
```

Donc le *print* qui est dans *default* sera aussi exécuté ! Toujours dans l'exemple précédent si je change le char et que je lui attribue espace comme valeur : *char* = ' ' ; j'aurai comme résultat l'exécution de la ligne qui suit *default* uniquement (la 6<sup>ème</sup> ligne donc).

Evidemment on utilisera par l'instruction comme tel, néanmoins la certification peut nous demander de prédire le résultat d'un Switch selon les situations qu'on vient de voir.

### • *Supported Types* :

Il faut être attentif car on ne peut pas utiliser tous les types avec Switch :

- Tous les « *integral data types* » (char, bytes, short, int) sont compatibles **SAUF LONG**.
- Les « *wrappers* »<sup>1</sup> sont autorisés (sauf celui de long).
- Les strings sont reconnus (ndr cela dépend des versions de Java).
- Les valeurs **ENUM** sont autorisées :

```
Day today = Day.SUN;
switch(today) {
    case SAT:
    case SUN:
        System.out.println("Weekend");
        break;
    default:
        System.out.println("Weekday");
        break;
}
```

```
enum Day {
    SUN, MON, TUE, WED, THU, FRI, SAT
}
```

Figure 1 Déclaration d'une classe enum

Dans cet exemple on voit l'utilisation combinée d'une classe **ENUM** avec un switch. De plus on peut voir ici un cas utile **d'ABSENCE** de break, au cas ou je veux une instruction identique si mon cas est un samedi ou un dimanche, et qui diffère de l'instruction des autres jours de la semaine.

<sup>1</sup> Pour rappel, ce sont les classes des types primitifs.



- *Supported Values :*

Ce qui est mis entre parenthèse dans le *Switch* peut être toute forme d'expression, tant que cela renvoi une valeur.

Dans l'exemple qui suit je mets le modulo d'une variable, ce qui est une forme d'expression valide :

```
final int evenValue = 0; //on notera qu'il y a mis FINAL
final int oddValue = 1;
switch(iVal % 2) { //valide
    case evenValue:
        System.out.println("even");
        break;
    case oddValue:
        System.out.println("odd");
        break;
} //si iVal = 10 alors le résultat sera oddValue (2ème case)
```

Dans le cas des valeurs du *case*, on ne peut avoir que des expressions de valeurs **CONSTANTES**. Dans l'exemple précédent on remarque que les case peuvent prendre des noms de variables tant que celle-ci sont déclarées FINAL car cela les rend constante.

Dans l'exemple suivant on voit qu'un des *case* prend en compte une **EQUATION**, ce qui est valide car le résultat sera constant :

```
final int evenValue = 0;
switch (iVal % 2) {
    case evenValue:
        System.out.println("even");
        break;
    case evenValue + 1:
        System.out.println("odd");
        break;
}
```

- *Choosing between Switch and If-else :*

De manière générale, le switch est plus **restrictif** mais il permet dans certain cas qu'on s'exprime plus clairement qu'avec un « if ».

#### if-else

- Extremely flexible
- Can handle most-any condition

#### Switch

- Test must be based on exact value match
- Type and value requirements

*Pour résumer, la différence entre les deux.*

```
if(grade == 'A')
    System.out.println("Great work!");
else if(grade == 'B' || grade == 'C' || grade == 'D')
    System.out.println("You passed!");
else
    System.out.println("Try again next time");
```

```
switch(grade) {
    case 'A':
        System.out.println("Great work!");
        break;
    case 'B':
    case 'C':
    case 'D':
        System.out.println("You passed!");
        break;
    default:
        System.out.println("Try again next time");
}
```

Ici un exemple de différence de lisibilité quand j'utilise un *if* ou un *switch* avec les mêmes données pour avoir un résultat identique.

- *When to AVOID switch :*

Si on reprend l'exemple du *if* avec la comparaison d'âge :

```
int age = 70;

if (age > 17) {
    System.out.println("Adult");
}
else if (age > 64) {
    System.out.println("Senior adult");
}
else {
    System.out.println("Minor");
}
```

Et qu'on essaye de le transposer en *switch* cela ne fonctionnera pas car les deux conditions qui sont ici, ne sont pas basées sur des correspondances **EXACTES** mais sur un test relationnel de la variable.

Autre exemple de transposition impossible :

On voit qu'on ne pourrait pas convertir ces instructions en *switch* car bien que la condition soit transposable, elle dépend d'une variable qui **MEME** si elle est en **FINAL**, est le résultat d'un appel de **METHODE**. Ce qui fait que la valeur de *maxItems* ne peut pas être déterminée de manière fiable au moment de la compilation si elle est transposée dans un *switch*.

```
int items = 100;
final int maxItems = readMaxItems();
if (items == maxItems)
    System.out.println("Max reached");
else
    System.out.println("Still room");
```

Method call makes maxItems value unknowable at compile time

### III. Looping

- *Introduction :*


Toutes les « loop » ont des caractéristiques en commun :

- Le « **Loop body** » : qui comprend la partie qu'on répète. Qui comprend par défaut une seule déclaration.
- Le « **Loop iteration** » : une lecture du code contenu dans la boucle.

- *While loop :*

On déclare un booléen entre parenthèse. Si la condition est vraie il y'a au moins une itération de la boucle (c'est-à-dire un passage).

```
int someValue = 4;
int factorial = 1;
while(someValue > 1) {
    factorial *= someValue;
    someValue--;
}
System.out.println(factorial);
```



- *Do while loop :*

Tout comme la *while loop*, il y'a une condition booléenne qui est entre parenthèse sauf qu'il y'a au moins une itération de la boucle avant que la condition soit examinée.

La chose à laquelle il faut être attentif dans le cas des *loop* est l'absence ou non d'accolades :

Un exemple **valide** :

```
do {
    System.out.print(iVal + " * 2 = ");
    iVal *= 2;
    System.out.println(iVal);
} while(iVal < 25);
```

Un exemple **non-valide** :

```
do
    System.out.print(iVal + " * 2 = ");
    iVal *= 2;
while(iVal < 25);
```

- *For Loop :*

La boucle prend trois paramètres entre parenthèse et séparés par un point-virgule :

- L'initialisation.
- La condition.
- La mise à jour à la fin de l'itération.

```
for (initialize; condition; update)
    statement ;
```

Tout comme *while*, la condition est examinée au début et la boucle peut contenir une seule déclaration en l'absence d'accolade.

A titre de comparaison, les deux boucles comprenant les mêmes paramètres mises côte à côte.

On voit que  
le `for` nous  
fait gagner  
en espace.

```
int i = 1;
while(i < 100) {
    System.out.println(i);
    i *= 2;
}

for(int i = 1; i < 100; i *= 2)
    System.out.println(i);
```

- *For Loop Control Variable :*

Il faut être **attentif** à la portée de la variable qui est déclarée entre les parenthèses d'une boucle `for` :

```
int factorial = 1;
for (int num = 3; num > 1; num--){
    factorial *= num;
    System.out.println(num + " | " + factorial);
}
System.out.println("Value of num is" + num); // this line won't compile
```

Il y'aura ici une erreur de compilation car on essaye d'utiliser une variable locale (*num*) en dehors de sa portée d'opération.

Autre aspect important, l'utilisation à l'intérieur de la boucle de la variable locale :

```
int factorial = 1;
for (int num = 3; num > 1; num--){
    factorial *= num;
    System.out.println(num + " | " + factorial);
    num = num + 5;
}
```

On voit ici qu'on augmente de 5 le *num*, ce qui n'entraîne pas d'erreur de compilation mais il faut tout de même l'éviter si l'on ne veut pas générer des erreurs de calcul ou de rendre le code confus.

- ***For Each Loop :***

C'est une boucle qui simplifie l'itération sur une collection. Elle exécute le corps de la boucle (*loop body*) une seule fois pour chaque item qui appartient à la collection.

Elle gère automatiquement :

- La répétition (le nombre d'itération).
- L'accès à chaque item.

Elle supporte deux types :

- Les *Array [ ]*.
- Tous les types qui implémentent l'interface « *Iterable* ».

Les paramètres comprennent :

- Le type de données du tableau.
- Le nom de la variable qui les récupère à chaque itération.
- Le nom du tableau dans lequel on itère, précédé de deux points.

```
float[] vals = { 10.0f, 20.0f, 15.0f };
float sum = 0.0f;
for(float currentVal : vals) {
    System.out.println("currentVal = " + currentVal);
    sum += currentVal;
}
```

- ***For Each Loop LIMITATIONS :***

Si elle est bien pratique, cette boucle n'en n'est pas moins limitée dans son utilisation. Si l'on veut itérer dans deux tableaux en même temps, il faudra d'office passer par une autre forme de boucle (il y'a une cas d'utilisation pratique d'une *forEach* dans un autre, à voir plus loin).

Cet exemple, dans lequel on additionne chaque donnée qui se situe au même index pour les deux tableaux n'est pas faisable avec un *forEach* :

```
int[] left = {5, 3, 7};
int[] right = {12, 9, 8};
for(int i = 0; i < left.length; i++) {
    int result = left[i] + right[i];
    System.out.println("result = " + result);
}
```

Idem si l'on veut parcourir un tableau en démarrant de la fin, et un tableau du début :

```
int[] left = {5, 3, 7};  
int[] right = {12, 9, 8};  
for(int i = 0, j = right.length - 1; i < left.length; i++, j--) {  
    int result = left[i] + right[j];  
    System.out.println("result = " + result);  
}
```

La particularité en **PLUS** ici, est que l'on peut déclarer **plusieurs variables différentes DANS UNE SEULE** boucle. Ici on a I et J.

## IV. Complex Looping and Branching :

- *Nesting Loops and if-else :*

**Nesting** (imbrication) : C'est le fait de placer des déclarations de boucles et de conditions l'une dans l'autre.

Exemple :

- Un if/else à l'intérieur d'un autre if/else.
- Une boucle dans un if/else.
- Une boucle dans une boucle.
- Etc.

```
int iVal = 1;
if(iVal < 5)
    do
        System.out.println("iVal = " + iVal++);
    while(iVal < 5);
else
    System.out.println("iVal is not less than 5");
```

La seule chose à retenir que dans le cas d'une imbrication il faut être attentif : si on a plus d'une ligne de code pour chaque fonction il faudra des accolades {}.

Par exemple ce code, bien qu'il soit sans accolade, est tout à fait correct.

- *Nested loops :*

**Nested loop** : C'est une boucle contenue dans une autre. En terminologie anglaise on dit : *the outer loop contains the inner loop*.

```
int[][] multi = {{100, 105, 110},
                 {200, 205, 210},
                 {300, 305, 310}};
for(int i = 0; i < multi.length; i++)
    for(int j = 0; j < multi[i].length; j++)
        System.out.println(multi[i][j]);
```

Principe : Pour chaque itération de la boucle externe, la boucle interne fait toutes ses itérations.

Dans cet exemple, une « nested loop » pour itérer à travers un tableau à deux dimensions.

Pareil dans cet exemple, sauf que ce sont des *forEach* qui sont imbriqués et qui fonctionnent pour un tableau à deux dimensions.

```
int[][] multi = {{100, 105, 110},
                 {200, 205, 210},
                 {300, 305, 310}};

for(int[] simple : multi)
    for(int value : simple)
        System.out.println(value);
```

- **Branching :**

**Branching :** Cela permet de modifier le « flux » de code. Cela grâce à deux mots clefs :

```
int iVal = 0;
while (iVal < 10) {
    iVal++;
    if(iVal % 2 == 0)
        continue;
    System.out.println(iVal);
}
```

**CONTINUE :** Permet de passer le reste de la portion de code dans l'itération.

On voit ici que le `continue` nous fera passer la dernière ligne **SI la valeur est paire**. Donc si la valeur vaut 1 le *CONTINUE* est exécuté et on passe à l'itération suivante.

**BREAK :** Interrompt la boucle ou le switch.

Dans ce cas on fait une incrémentation qui va à chaque fois modifier les valeurs de *sum*.

Lorsque la condition sera vraie le `break` sera exécuté et interrompra la boucle. Ce qui fait qu'on la quittera.

```
int sum = 0, iVal = 1;
while (iVal < 10) {
    sum += iVal;
    System.out.println("iVal=" + iVal + " sum=" + sum);
    if (sum > 5)
        break;
    iVal++;
}
```

Dans le cas d'une *nested loop*, il n'interrompt que la boucle dans laquelle cette déclaration se trouve.



- *The Return statement :*

**Return** : Permet de sortir d'une méthode, y compris de toutes les boucles.

Contrairement à un Break, quand il est exécuté, dans le cas où on a des boucles dans des boucles (nested loop), on les **interrompt** toutes.

- *Infinite loops :*

Elles sont en générale involontaires mais on peut en avoir besoin dans certaine condition. On peut s'y prendre de deux manières :

Avec un while :

```
while (true) System.out.println("Loop");
```

Avec un for :

```
for (;;) System.out.println("loopFor");
```