

Les structures de contrôle

Comme la plupart des langages impératifs, Java propose un ensemble de structures de contrôle.

if-else

L'expression **if** permet d'exécuter un bloc d'instructions uniquement si l'expression booléenne est évaluée à vrai :

```
if (i % 2 == 0) {  
    // instructions à exécuter si i est pair  
}
```

L'expression **if** peut être optionnellement suivie d'une expression **else** pour les cas où l'expression est évaluée à faux :

```
if (i % 2 == 0) {  
    // instructions à exécuter si i est pair  
} else {  
    // instructions à exécuter si i est impair  
}
```

L'expression **else** peut être suivie d'une nouvelle instruction **if** afin de réaliser des choix multiples :

```
if (i % 2 == 0) {  
    // instructions à exécuter si i pair  
} else if (i > 10) {  
    // instructions à exécuter si i est impair et supérieur à 10  
} else {  
    // instructions à exécuter dans tous les autres cas  
}
```

Si le bloc d'instruction d'un **if** ne comporte qu'une seule instruction, alors les accolades peuvent être omises :

```
if (i % 2 == 0)
    i++;
```

Cependant, beaucoup de développeurs Java préfèrent utiliser systématiquement les accolades.

return

return est un mot clé permettant d'arrêter immédiatement le traitement d'une méthode et de retourner la valeur de l'expression spécifiée après ce mot-clé. Si la méthode ne retourne pas de valeur (**void**), alors on utilise le mot-clé **return** seul. L'exécution d'un **return** entraîne la fin d'une structure de contrôle.

```
if (i % 2 == 0) {
    return 0;
}
```

Écrire des instructions immédiatement après une instruction **return** n'a pas de sens puisqu'elles ne seront jamais exécutées. Le compilateur Java le signalera par une erreur *unreachable code*.

```
if (i % 2 == 0) {
    return 0;
    i++; // Erreur de compilation : unreachable code
}
```

while

L'expression *while* permet de définir un bloc d'instructions à répéter tant que l'expression booléenne est évaluée à vrai.

```
while (i % 2 == 0) {
    // instructions à exécuter tant que i est pair
}
```

L'expression booléenne est évaluée au départ et après chaque exécution du bloc d'instructions.

Si le bloc d'instruction d'un **while** ne comporte qu'une seule instruction, alors les accolades peuvent être omises :

```
while (i % 2 == 0)
    // instruction à exécuter tant que i est pair
```

Cependant, beaucoup de développeurs Java préfèrent utiliser systématiquement les accolades.

do-while

Il existe une variante de la structure précédente, nommée **do-while** :

```
do {
    // instructions à exécuter
} while (i % 2 == 0);
```

Dans ce cas, le bloc d'instruction est exécuté une fois puis l'expression booléenne est évaluée. Cela signifie qu'avec un **do-while**, le bloc d'instruction est exécuté au moins une fois.

for

Une expression **for** permet de réaliser une itération. Elle commence par réaliser une initialisation puis évalue une expression booléenne. Tant que cette expression booléenne est évaluée à vrai, le bloc d'instructions est exécuté et un incrément est appelé.

```
for (initialisation; expression booléenne; incrément) {
    bloc d'instructions
}
```

```
for (int i = 0; i < 10; ++i) {
    // instructions
}
```

il n'est pas possible d'omettre l'initialisation, l'expression booléenne ou l'incrément dans la déclaration d'une expression *for*. Par contre, il est possible de les laisser vides.

```
int i = 0;
for (; i < 10; ++i) {
    // instructions
}
```

Il est ainsi possible d'écrire une expression *for* sans condition de sortie, la fameuse boucle infinie :

```
for (;;) {  
    // instructions à exécuter à l'infini  
}
```

Si le bloc d'instruction d'un **for** ne comporte qu'une seule instruction, alors les accolades peuvent être omises :

```
for (int i = 0; i < 10; ++i)  
    // instruction à exécuter
```

Cependant, beaucoup de développeurs Java préfèrent utiliser systématiquement les accolades.

for amélioré

Il existe une forme améliorée de l'expression *for* (souvent appelée *for-each*) qui permet d'exprimer plus succinctement un parcours d'une collection d'éléments.

```
for (int i : maCollection) {  
    // instructions à exécuter  
}
```

Pour que cette expression compile, il faut que la variable désignant la collection à droite de : implémente le type Iterable ou qu'il s'agisse d'un tableau. Il faut également que la variable à gauche de : soit compatible pour l'assignation d'un élément de la collection.

```
short arrayOfShort[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
  
for (int k : arrayOfShort) {  
    System.out.println(k);  
}
```

break-continue

Pour les expressions **while**, **do-while**, **for** permettant de réaliser des itérations, il est possible de contrôler le comportement à l'intérieur de la boucle grâce aux mots-clés **break** et **continue**.

break quitte la boucle sans exécuter le reste des instructions.

```
int k = 10;
for (int i = 1 ; i < 10; ++i) {
    k *= i
    if (k > 200) {
        break;
    }
}
```

continue arrête l'exécution de l'itération actuelle et commence l'exécution de l'itération suivante.

```
for (int i = 1 ; i < 10; ++i) {
    if (i % 2 == 0) {
        continue;
    }
    System.out.println(i);
}
```

libellé

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

```
int m = 0;

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

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

Dans l'exemple ci-dessus, *boucleDeCalcul* est un libellé qui permet de signifier que l'instruction **break** porte sur la boucle de plus haut niveau. Son exécution stoppera donc l'itération des deux boucles et passera directement à l'affichage du résultat sur la sortie standard.

switch

Un expression **switch** permet d'effectuer une sélection parmi plusieurs valeurs.

```
switch (s) {  
  case "valeur 1":  
    // instructions  
    break;  
  case "valeur 2":  
    // instructions  
    break;  
  case "valeur 3":  
    // instructions  
    break;  
  default:  
    // instructions  
}
```

switch évalue l'expression entre parenthèses et la compare dans l'ordre avec les valeurs des lignes **case**. Si une est identique alors il commence à exécuter la ligne d'instruction qui suit. Attention, un **case** représente un point à partir duquel l'exécution du code commencera. Si on veut isoler chaque cas, il faut utiliser une instruction **break**. Au contraire, l'omission de l'instruction **break** peut être pratique si on veut effectuer le même traitement pour un ensemble de cas :

```
switch (c) {  
  case 'a':  
  case 'e':  
  case 'i':  
  case 'o':  
  case 'u':  
  case 'y':  
    // instruction pour un voyelle  
    break;  
  default:  
    // instructions pour une consonne
```

```
}
```

On peut ajouter une cas **default** qui servira de point d'exécution si aucun **case** ne correspond.

Par convention, on place souvent le cas **default** à la fin. Cependant, il agit plus comme un libellé indiquant la ligne à laquelle doit commencer l'exécution du code. Il peut donc être placé n'importe où :

```
switch (c) {  
    default:  
        // instructions pour une consonne  
    case 'a':  
    case 'e':  
    case 'i':  
    case 'o':  
    case 'u':  
    case 'y':  
        // instructions pour les consonnes et les voyelles  
}
```

En Java, le type d'expression accepté par un **switch** est limité. Un **switch** ne compile que pour un type primitif, une énumération ou une chaîne de caractères.

Revision #1

Created 10 February 2025 10:57:34 by Nicolas

Updated 10 February 2025 11:13:14 by Nicolas