Les structures de contrôle

Les structures de contrôle permettent de modifier l'ordre d'exécution des instructions dans votre code. Deux types de structures sont disponibles :

- Les structures de décision : elles aiguilleront l'exécution du code en fonction des valeurs que pourra prendre une expression de test.
- Les structures de boucle : elles feront exécuter une portion de code un certain nombre de fois, jusqu'à ce qu'une condition soit remplie ou tant qu'une condition est remplie.

1. Structures de décision

Deux solutions sont possibles.

a. Structure if

Quatre syntaxes sont utilisables pour l'instruction if.

if (condition)instruction;

Si la condition est vraie alors l'instruction est exécutée. La condition doit être une expression qui, une fois évaluée, doit fournir un boolean true ou false. Avec cette syntaxe, seule l'instruction située après le if sera exécutée si la condition est vraie. Pour pouvoir faire exécuter plusieurs instructions en fonction d'une condition il faut utiliser la syntaxe ci-après.

```
if (condition)
{
    Instruction 1;
    ...
    Instruction n;
}
```

Dans ce cas le groupe d'instructions situé entre les accolades sera exécuté si la condition est vraie.

Vous pouvez également spécifier une ou plusieurs instructions qui elles seront exécutées si la condition est fausse.

```
if (condition)
{
    Instruction 1;
    ...
    Instruction n;
}
else
{
    Instruction 1;
```

```
    ...
    Instruction n;
}
```

Vous pouvez également imbriquer les conditions avec la syntaxe.

```
if (condition1)
{
    Instruction 1
    ...
    Instruction n
}
else if (Condition 2)
{
    Instruction 1
    ...
    Instruction n
}
else if (Condition 3)
{
    Instruction 1
    ...
    Instruction n
}
else
{
    Instruction 1
    ...
    Instruction n
}
```

Dans ce cas, on teste la première condition. Si elle est vraie alors le bloc de code correspondant est exécuté sinon on teste la suivante et ainsi de suite. Si aucune condition n'est vérifiée, le bloc de code spécifié après le else est exécuté. L'instruction else n'est pas obligatoire dans cette structure. Dans ce cas, il se peut qu'aucune instruction ne soit exécutée si aucune des conditions n'est vraie.

Il existe également un opérateur conditionnel permettant d'effectuer un if ... else en une seule instruction.

```
condition ? expression1 : expression2;
```

Cette syntaxe est équivalente à celle-ci :

```
If (condition)
expression1;
else
expression2;
```

b. Structure switch

La structure switch permet un fonctionnement équivalent mais offre une meilleure lisibilité du code. La syntaxe est la suivante :

```
Switch (expression)
{
    Case valeur1:
            Instruction 1
            ...
            Instruction n
            Break;
    Case valeur2:
            Instruction 1
            ...
            Instruction n
            Break;
    Default:
            Instruction 1
            ...
            Instruction n
}
```

La valeur de l'expression est évaluée au début de la structure (par le switch) puis la valeur obtenue est comparée avec la valeur spécifiée dans le premier case.

Si les deux valeurs sont égales, alors le bloc de code 1 est exécuté.

Sinon, la valeur obtenue est comparée avec la valeur du case suivant, s'il y a correspondance, le bloc de code est exécuté et ainsi de suite jusqu'au dernier case.

Si aucune valeur concordante n'est trouvée dans les différents case alors le bloc de code spécifié dans le default est exécuté. Chacun des blocs de code doit se terminer par l'instruction break.

Si ce n'est pas le cas l'exécution se poursuivra par le bloc de code suivant jusqu'à ce qu'une instruction break soit rencontrée ou jusqu'à la fin de la structure switch. Cette solution peut être utilisée pour pouvoir exécuter un même bloc de code pour différentes valeurs testées.

La valeur à tester peut être contenue dans une variable mais elle peut également être le résultat d'un calcul. Dans ce cas, le calcul n'est effectué qu'une seule fois au début du switch. Le type de la valeur testée peut être numérique entière, caractère, chaîne de caractères ou énumération. Il faut bien sûr que le type de la variable testée corresponde au type des valeurs dans les différents case.

Si l'expression est de type chaîne de caractères, la méthode equals est utilisée pour vérifier l'égalité avec les valeurs des différents case. La comparaison fait donc une distinction entre minuscules et majuscules.

```
BufferedReader br;
      br=new BufferedReader(new InputStreamReader(System.in));
      String reponse="";
      reponse=br.readLine();
      switch (reponse)
      {
         case "oui":
```

```
            case "OUI":
                System.out.println("réponse positive");
                break;
            case "non":
            case "NON":
                System.out.println("réponse négative");
                break;
            default:
                System.out.println("mauvaise réponse");
        }
```

2. Les structures de boucle

Trois structures sont à notre disposition :

```
while (condition)
do ... while (condition)
for
```

Elles ont toutes pour but de faire exécuter un bloc de code un certain nombre de fois en fonction d'une condition.

a. Structure while

Cette structure exécute un bloc de façon répétitive tant que la condition est true.

```
while (condition)
{
    Instruction 1
    ...
    Instruction n
}
```

La condition est évaluée avant le premier passage dans la boucle. Si elle est false à cet instant alors le bloc de code n'est pas exécuté. Après chaque exécution du bloc de code la condition est à nouveau évaluée pour vérifier si une nouvelle exécution du bloc de code est nécessaire. Il est recommandé que l'exécution du bloc de code contienne une ou plusieurs instructions susceptibles de faire évoluer la condition. Si ce n'est pas le cas la boucle s'exécutera sans fin. Il ne faut surtout pas placer de caractère ; après le while car dans ce cas, le bloc de code n'est plus associé à la boucle.

```
int i=0;
while (i<10)
{
    System.out.println(i);
    i++;
}
```

b. Structure do ... while

```
do
```

```
{
    Instruction 1
    ...
    Instruction n
}
while (condition);
```

Cette structure a un fonctionnement identique à la précédente sauf que la condition est examinée après l'exécution du bloc de code. Elle nous permet de garantir que le bloc de code sera exécuté au moins une fois puisque la condition sera testée pour la première fois après la première exécution du bloc de code. Si la condition est true alors le bloc est exécuté une nouvelle fois jusqu'à ce que la condition soit false. Vous devez faire attention à ne pas oublier le point-virgule après le while sinon le compilateur détecte une erreur de syntaxe.

```
do
{
    System.out.println(i);
    i++;
}
while(i<10);
```

c. Structure for

Lorsque vous connaissez le nombre d'itérations à réaliser dans une boucle il est préférable d'utiliser la structure for. Pour pouvoir utiliser cette instruction, une variable de compteur doit être déclarée. Cette variable peut être déclarée dans la structure for ou à l'extérieur, elle doit dans ce cas être déclarée avant la structure for.

La syntaxe générale est la suivante :

```
for(initialisation;condition;instruction d'itération)
{
    Instruction 1
    ...
    Instruction n


}
```

La partie initialisation est exécutée une seule fois lors de l'entrée dans la boucle. La partie condition est évaluée lors de l'entrée dans la boucle puis à chaque itération. Le résulat de l'évaluation de la condition détermine si le bloc de code est exécuté, il faut pour cela que la condition soit évaluée comme true. Après l'exécution du bloc de code l'instruction d'itération est à son tour exécutée. Puis la condition est à nouveau testée et ainsi de suite tant que la condition est évaluée comme true.

Voici ci-dessous deux boucles for en action pour afficher une table de multiplication.

```
int k;
for(k=1;k<10;k++)
```

```
{
   for (int l = 1; l < 10; l++)
   {
      System.out.print(k * l + "\t");
   }
   System.out.println();
}
```

Nous obtenons le résultat suivant :

```
1   2   3   4   5   6   7   8   9
2   4   6   8   10   12   14   16   18
3   6   9   12   15   18   21   24   27
4   8   12   16   20   24   28   32   36
5   10   15   20   25   30   35   40   45
6   12   18   24   30   36   42   48   54
7   14   21   28   35   42   49   56   63
8   16   24   32   40   48   56   64   72
9   18   27   36   45   54   63   72   81
```

Une autre syntaxe de la boucle for permet de faire exécuter un bloc de code pour chaque élément contenu dans un tableau ou dans une instance de classe implémentant l'interface Iterable. La syntaxe générale de cette instruction est la suivante :

```
for (type variable : tablo)
{
    Instruction 1
    ...
    Instruction n
}
```

Il n'y a pas de notion de compteur dans cette structure puisqu'elle effectue elle-même les itérations sur tous les éléments présents dans le tableau ou la collection.

La variable déclarée dans la structure sert à extraire un à un les éléments du tableau ou de la collection pour que le bloc de code puisse les manipuler. Il faut bien sûr que le type de la variable soit compatible avec le type des éléments stockés dans le tableau ou la collection. La variable doit obligatoirement être déclarée dans la structure for et non à l'extérieur. Elle ne sera utilisable qu'à l'intérieur de la structure. Par contre vous n'avez pas à vous soucier du nombre d'éléments car la structure est capable de gérer elle-même le déplacement dans le tableau ou la collection. Voici un petit exemple pour clarifier la situation !

Avec une boucle classique :

```
   String[] tablo={"rouge","vert","bleu","blanc"};
   int cpt;
   for (cpt = 0; cpt < tablo.length; cpt++)
   {
         System.out.println(tablo[cpt]);
   }
```

Avec la boucle for d'itération :

```
String[] tablo={"rouge","vert","bleu","blanc"};
for (String s : tablo)
{
    System.out.println(s);
}
```

Le code placé à l'intérieur de cette structure for ne doit pas modifier le contenu de la collection.

Il est donc interdit d'ajouter ou de supprimer des éléments pendant le parcours de la collection. Le problème ne se pose pas avec un tableau. La taille d'un tableau étant fixe, il est bien impossible d'y ajouter ou d'y supprimer un élément. Le code suivant met en évidence cette limitation lors du parcours d'une ArrayList. L'ajout d'un élément à l'Arraylist en cours d'itération déclenche une exception de type ConcurrentModificationException.

```
ArrayList<String> lst;
st=new ArrayList<String>();
lst.add("client 1");
lst.add("client 2");
lst.add("client 3");
lst.add("client 5");


for(String st:lst)
{
    System.out.println(st);
    f(st.endsWith("3"))
    {
        lst.add("client 4");
    }
}
```

d. Interruption d'une structure de boucle

Trois instructions peuvent modifier le fonctionnement normal des structures de boucle.

break

Si cette instruction est placée à l'intérieur du bloc de code d'une structure de boucle elle provoque la sortie immédiate de ce bloc de code. L'exécution se poursuit par l'instruction placée après le bloc de code. Cette instruction doit en général être exécutée de manière conditionnelle, sinon les instructions situées après à l'intérieur de la boucle ne seront jamais exécutées.

Dans le cas de boucles imbriquées, il est possible d'utiliser l'instruction break associée avec une étiquette. L'exemple de code ci-dessous effectue le parcours d'un tableau à deux dimensions et s'arrête dès qu'une case contenant la valeur 0 est rencontrée.

```
  int[][] points = {
                { 10,10,},
                { 0,10 },
                { 45,24 }};
    int x=0,y=0;
    boolean trouve=false;
      recherche:
           for (x = 0; x <points.length; x++)
           {
                for (y = 0; y < points[x].length;y++)
                {
                     if (points[x][y] == 0)
                        {
                            trouve = true;
                            break recherche;
                        }
                }
           }
       if (trouve)
       {
            System.out.println("resultat trouvé dans la case "
+ x + "-" + y);
       }
       else
       {
            System.out.println("recherche infructueuse");
       }
```

Continue

Cette instruction permet d'interrompre l'exécution de l'itération courante d'une boucle et de continuer l'exécution à l'itération suivante après vérification de la condition de sortie de boucle. Comme pour l'instruction break elle doit être exécutée de manière conditionnelle et accepte également l'utilisation d'une étiquette.

Voici un exemple de code utilisant une boucle sans fin et ses deux instructions pour afficher les nombres impairs jusqu'à ce que l'utilisateur saisisse un retour chariot.

```
import java.io.IOException;
public class TestStructures {
    static boolean stop;
    public static void main(String[] args)
    {
        new Thread()
        {
            public void run()
            {
                int c;
                try
                {
                    c=System.in.read();
                    stop=true;
                }
                catch (IOException e)
```

```
                {
                    e.printStackTrace();
                }
            }
        }.start();
        long compteur=0;
        while(true)
        {
            compteur++;
            if (compteur%2==0)
                continue;
            if (stop)
                break;
            System.out.println(compteur);
        }
    }
}
```

Return

L'instruction return est utilisée pour sortir immédiatement de la méthode en cours d'exécution et poursuivre l'exécution par l'instruction suivant celle qui a appelé cette méthode. Si elle est placée dans une structure de boucle, elle provoque bien sûr la sortie immédiate de la boucle puis de la méthode dans laquelle se trouve la boucle. L'utilisation de cette instruction dans une fonction dont le type de retour est autre que void oblige à fournir à l'instruction return une valeur compatible avec le type de retour de la fonction.