R1.01 : Initiation au développement Cours 5

M.Adam, N.Delomez, JF.Kamp, L.Naert

IUT de Vannes

9 août 2022

Sommaire

Les deux autres formes de boucles

- `do-while`
- `for`

La boucle `do-while`

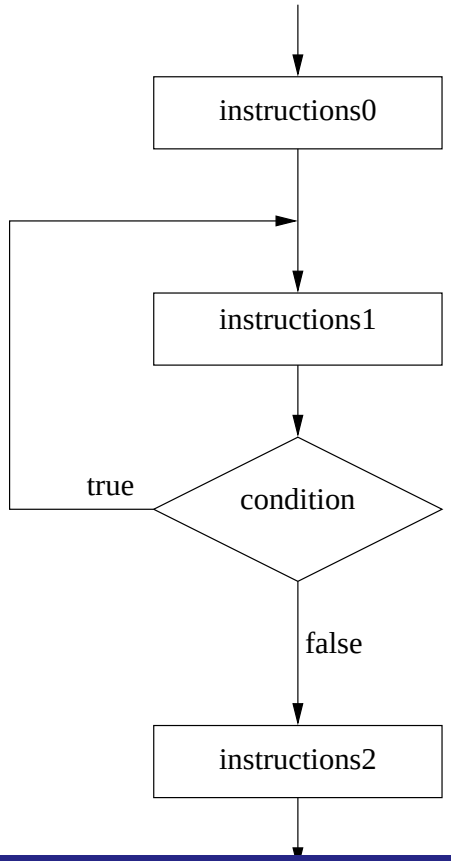
C'est une boucle qui s'exécute au moins une fois.

Syntaxe

```
do {
    instructions ;
} while (booleen condition);
```

Sémantique

```
instructions0;
do {
     instructions1;
} while (condition);
instructions2;
```



Sémantique

```
instructions0;
instructions1;
while (condition) {
    instructions1;
}
instructions2;
```

|

```
instructions0;

do {
    instructions1;
} while (condition);
instructions2;
```

Un exemple avec while

```
/**
 * Saisie d'un entier positif
 * @author M.Adam
 */
class While {
    void principal () {
        int val;

        System.out.println ("Saisie d'un nombre positif");
        val = SimpleInput.getInt ("Entrez un entier positif");
        while (val < 0) {
            val = SimpleInput.getInt ("Entrez un entier positif");
        }
        System.out.println ("La valeur saisie est : " + val);
    }
}
```

Même exemple avec do-while

```
/**
 * Saisie d'un entier positif
 * @author M.Adam
 */
class DoWhile {
    void principal () {
        int val;

        do {
            val = SimpleInput.getInt ("Entrez un entier positif");
        } while (val < 0);
        System.out.println ("La valeur saisie est : " + val);
    }
}
```

Utilisation

Une boucle `do-while` est à utiliser quand il faut exécuter au moins une fois la boucle avant de sortir.

La boucle `for`

C'est une boucle qui :

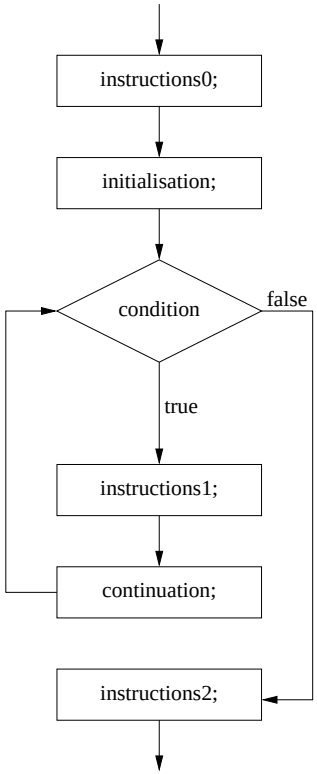
- s'exécute un **nombre connu de tours**.

Syntaxe

```
for (initialisation; condition; continuation) {
    instructions;
}
```

Sémantique

```
instructions0;
for (initalisation; condition; continuation) {
    instructions1;
}
instructions2;
```



```
instructions0;                  | instructions0;
i = 0;                          |
while (i < j) {                 | for (i = 0; i < j;  i = i + 1) {
    instructions1;              |     instructions1;
    i = i + 1;                  |
}                               | }
instructions2;                  | instructions2;
```

Remarque : souvent dans ce cadre `i = i + 1` s'écrit `i++`.

Un exemple avec while

```
/**
 * Affiche 10 nombres aléatoirement
 * @author M.Adam
 */
class WhileF {
    void principal (){
        System.out.println ("Affiche 10 nombres aléatoirement");
        int i = 0;
        while (i < 10) {
            System.out.println (i + " " + Math.random() * 100);
            i = i + 1;
        }
        System.out.println ("fin de l'algorithme");
    }
}
```

Même exemple avec for

```
/**
 * Affiche 10 nombres aléatoirement
 * @author M.Adam
 */
class For {
    void principal (){
        System.out.println ("Affiche 10 nombres aléatoirement");
        for (int i = 0; i < 10; i = i + 1) {
            System.out.println (i + " " + Math.random() * 100);
        }
        System.out.println ("fin de l'algorithme");
    }
}
```

Remarque

Une boucle `for` est à utiliser quand le nombre de tours de boucle est connu.
Dans l'exemple précédent il faut s'interdire de modifier la variable `i` dans la corps de la boucle `for`.

Programmer ces deux formes de boucles

La boucle `while` est suffisante pour programmer tous les algorithmes.
Les autres formes de boucles permettent

- parfois de simplifier l'écriture des boucles,
- souvent de simplifier leur lecture.

La boucle `do-while`

Deux solutions :

- adapter la méthode pour la boucle `while`,
- utiliser la méthode pour la boucle `while` et reconnaitre le schéma de traduction pour la transformer en boucle `do-while`.

La boucle `for`

Deux solutions :

- comme le nombre de tours est connu, programmer dès le départ une boucle pour,
- utiliser la méthode pour la boucle `while` et reconnaitre le schéma de traduction pour la transformer en boucle `for`.

Sommaire

Les tests unitaires

A chaque méthode `methode()` est associée une méthode `testMethode()`.
Cette méthode permet le test unitaire de la méthode :

- tester tous les cas de **bon fonctionnement**,
- tester un maximum de cas de bon fonctionnement si l'exhaustivité n'est pas possible,
- ne pas traiter les cas d'erreurs car, pour la première partie de la ressource R1.01, nous faisons l'hypothèse que les paramètres sont corrects et n'engendre pas un arrêt inopiné de l'exécution de ma méthode,
- n'utiliser que des variables locales,
- être "bavarde".

Exemple de Test Unitaire

Soit la méthode `palindrome()` dont la description est donnée par :

```
/**
 * Teste si une chaîne est un palindrome
 * @param mot chaîne à tester
 * @return vrai si la chaîne est un palindrome, faux sinon
*/
boolean palindrome (String mot)
```

Tests à effectuer

-
-
-
-
-

Code de testPalindrome()

```
void testPalindrome() {
        System.out.println ();
        System.out.println ("*** testPalindrome()");

        // Arrange
        String mot1 = "radar";
        System.out.print ("palindrome (\"" + mot1 + "\") \t= true  : ");
        // Act
        System.out.println (palindrome(mot1));

        // Arrange
        String mot2 = "abba";
        System.out.print ("palindrome (\"" + mot2 + "\")  \t= true  : ");
        // Act
        System.out.println (palindrome(mot2));

        // Arrange
        String mot3 = "rider";
        System.out.print ("palindrome (\"" + mot3 + "\") \t= false : ");
        // Act
        System.out.println (palindrome(mot3));

        //Arrange
        String mot4 = "roar";
        System.out.print ("palindrome (\"" + mot4 + "\") \t= false : ");
        // Act
        System.out.println (palindrome(mot4));

        // Arrange
        String mot5 = "";
        System.out.print ("palindrome (\"" + mot5 + "\")  \t= true  : ");
        // Act
        System.out.println (palindrome(mot5));
```

Exécution

```
$

*** testPalindrome()
palindrome ("radar")      = true  : true
palindrome ("abba")       = true  : true
palindrome ("rider")      = false : false
palindrome ("roar")      = false : false
palindrome ("")           = true  : true
palindrome ("a")          = true  : true
```

Remarque : Le code est très répétitif.

Factoriser les cas de tests

```
/**
 * teste un appel de palindrome
 * @param mot    mot à tester
 * @param result resultat attendu
**/
void testCasPalindrome (String mot, boolean result) {
    // Arrange
    System.out.print ("palindrome (\"" + mot + "\") \t= " + result + "\t : ");

    // Act
    System.out.println (palindrome (mot));
}

/**
 * Teste la fonction palindrome"
 */
void testPalindrome() {
    System.out.println ();
    System.out.println ("*** testPalindrome()");

    testCasPalindrome ("radar", true);
    testCasPalindrome ("abba", true);
    testCasPalindrome ("rider", false);
    testCasPalindrome ("roar", false);
    testCasPalindrome ("", true);
    testCasPalindrome ("a", true);

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

Exécution

```
$

*** testPalindrome()
palindrome ("radar")     = true   : true
palindrome ("abba")      = true   : true
palindrome ("rider")     = false  : false
palindrome ("roar")      = false  : false
palindrome ("")          = true   : true
palindrome ("a")         = true   : true
```

Remarque : Le programmeur est obligé de vérifier visuellement que le test s'est bien déroulé.

Code de testPalindrome()

Il est même possible "d'automatiser" le test :

```
/**
  * teste un appel de palindrome
  * @param mot    mot à tester
  * @param result resultat attendu
**/
void testCasPalindrome (String mot, boolean result) {
    // Arrange
    System.out.print ("palindrome (\"" + mot + "\") \t= " + result + "\t : ");

    // Act
    boolean resExec = palindrome (mot);

    // Assert
    if ( resExec == result ){
        System.out.println ("OK");
    } else {
        System.err.println ("ERREUR");
    }
}
```

Exécution

Avec la méthode `palindrome()` incorrecte :

```
$
*** testPalindrome()
palindrome ("radar")     = true      : ERREUR
palindrome ("abba")      = true      : ERREUR
palindrome ("rider")     = false     : OK
palindrome ("roar")      = false     : OK
palindrome ("")          = true      : OK
palindrome ("a")         = true      : OK
```

Avantages

- Chaque méthode est testée au fur et à mesure.
- Chaque méthode de test doit être indépendante, avec ses propres variables.
- Les méthodes de test peuvent être codées par une autre personne.
- Les méthodes de test peuvent être codées avant la méthode à tester.
- La recherche d'erreurs est facilitée.
- La construction des autres méthodes qui utilisent du code déjà écrit et testé est plus solide.
- Il est possible ultérieurement de d'utiliser à nouveau une méthode de test.

Inconvénients

- L'écriture initiale du code est plus longue.
- Il faut avoir bien réfléchir aux tests et être le plus exhaustif possible.
- Il faut savoir remettre en cause une méthode déjà testée.

Problèmes complexes

Parfois un problème à résoudre est très complexe à programmer.

Divide and conquer

L'idée est de décomposer un gros problème en de plus petits problèmes plus simples à résoudre.

L'exemple du tri dit par sélection

Voir la vidéo :
`https://www.youtube.com/watch?v=Ns4TPTC8whw`!
Ou sous AlgoTouch : `https://tinyurl.com/C05AGT01`

La méthode afficheTab()

```
/**
 * affiche le contenu d'un tableau d'entiers
 * @param t tableau d'entiers
 **/
void afficheTab (int[] t)
```

Le test de afficheTab()

```
$

*** testAfficheTab()()
afficheTab ({1, 3, 4, 10, 5 , 7}) : {1, 3, 4, 10, 5, 7}

afficheTab ({})                   : {}

afficheTab ({1})                  : {1}

$
```

La méthode echange()

```
/**
 * echange deux valeurs dans un tableau d'entiers
 * @param tab tableaux d'entiers
 * @param i   premier indice pour l'échange
 * @param j   deuxième indice pour l'échange
*/
void echange(int[] tab, int i, int j)
```

```
$

*** testEchange()
echange ( {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}, 4, 5) :
Avant : {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}
Après : {10, 20, 30, 40, 60, 50, 70, 80, 90, 100}

echange ( {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}, 9, 9) :
Avant : {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}
Après : {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}

echange ( {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}, 0, 0) :
Avant : {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}
Après : {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}

echange ( {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}, 0, 9) :
Avant : {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}
Après : {100, 20, 30, 40, 50, 60, 70, 80, 90, 10}

------------------
(program exited with code: 0)
```

Procédure placeMin()

L'idée est de placer au rang i la plus petite des valeurs se trouvant dans la suite du tableau.

```
/**
 * place au rang i la plus petite valeur de la suite du  tableau
 * @param tab tableau d'entiers
 * @param i   rang de départ où sera placée la valeur la plus petite
 */
void placeMin(int[] tab, int i)
```

Toutes les valeurs des paramètres sont supposées être correctes.

Principe

Parcourir le tableau à partir du rang i. Si la valeur courante est plus petite que celle à l'indice i, les deux cases sont permutées.

Corps de boucle

L'indice `j` est utilisé pour le parcours du tableau.

```
if (tab[i] > tab[j]) {
    echange (tab, i, j);
}
j = j + 1;
```

Conditions de sortie

- `j >= tab.length`, l'indice est sortie du tableau.

Condition de continuation

- `!(j >= tab.length)`
- qui se réécrit `j < tab.length`

Initialisation

```
j = i + 1;
```

Il est inutile de comparer `tab[i]` et `tab[i]`.

Terminaison

rien

Code complet

```
/**
 * place au rang i la plus petite valeur de la suite du  tableau
 * @param tab tableau d'entiers
 * @param i   rang de départ où sera placée la valeur la plus petite
 */
void placeMin(int[] tab, int i) {
    int j;

    j = i + 1;
    while (j < tab.length) {
        if (tab[i] > tab[j]) {
            echange (tab, i, j);
        }
        j = j + 1;
    }
}
```

Évidemment, certains auront reconnu une boucle `for`.

Procédure testPlaceMin()

```
$

*** testPlaceMin()
placeMin ({100, 30, 20, 40, 60, 70, 90, 80, 50, 40}, 0) :
Après    : {20, 100, 30, 40, 60, 70, 90, 80, 50, 40}

placeMin ({20, 100, 30, 40, 60, 70, 90, 80, 50, 40}, 0) :
Après    : {20, 100, 30, 40, 60, 70, 90, 80, 50, 40}

placeMin ({20, 100, 30, 40, 60, 70, 90, 80, 50, 40}, 5) :
Après    : {20, 100, 30, 40, 60, 40, 90, 80, 70, 50}

placeMin ({20, 100, 30, 40, 60, 70, 90, 80, 50, 40}, 9) :
Après    : {20, 100, 30, 40, 60, 70, 90, 80, 50, 40}

-----------------
```

Procédure triSelection()

```
/**
 * tri d'un tableau d'entiers selon la méthode dite de sélection
 * @param tab tableau d'entiers à trier
 */
void triSelection(int[] tab)
```

Principe

Le tableau est parcouru entièrement par un indice, `i`. À chaque tour, la plus petite valeur se trouvant dans le tableau à partir du rang `i` est placée en `i`.

Corps de boucle

```
placeMin(tab, i);
i = i + 1;
```

Conditions de sortie

- `i >= tab.length` en dehors du tableau

Condition de continuation

- `!(i >= tab.length)`
- qui se réécrit `i < tab.length`

Initialisation

```
i = 0;
```

Terminaison

rien

Code complet

```
/**
 * tri d'un tableau d'entiers selon la méthode dite de sélection
 * @param tab tableau d'entiers à trier
 */
void triSelection(int[] tab) {
    int i = 0;
    while(i < tab.length) {
        placeMin(tab, i);
        i = i + 1;
    }
}
```

Évidemment, là encore, il s'agit d'une boucle `for`.

Procédure testTriSelection()

```
$
*** testTriSelection()
triSelection ({100, 30, 20, 40, 60, 70, 90, 80, 50, 40}) :
Après       : {20, 30, 40, 40, 50, 60, 70, 80, 90, 100}

triSelection ({20, 30, 40, 40, 50, 60, 70, 80, 90, 100}) :
Après       : {20, 30, 40, 40, 50, 60, 70, 80, 90, 100}

triSelection ({10, 9, 8, 7, 6, 5, 4, 3, 2, 1}) :
Après       : {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

triSelection ({}) :
Après       : {}

triSelection ({100}) :
Après       : {100}
------------------
(program exited with code: 0)
```

Sommaire

A retenir

- Il existe d'autres formes de boucles : `for` et `do-while`.
- La boucle `for` sert quand le nombre de tours est connu.
- La boucle `do-while` sert quand le corps de boucle doit toujours être exécuté au moins une fois.
- Chaque méthode `met` doit être testée individuellement par une méthode `testMet()`.
- Quand un problème est complexe, il faut le décomposer en problèmes moins complexes à programmer.
- La décomposition d'un problème en sous-problème rend la mise au point plus simple.