Séquence 17

Méthode récursive

La récursivité est une technique de programmation qu'il est pratiquement indispensable de maîtriser pour résoudre certains problèmes. Une méthode est dite récursive si le corps de la méthode contient une ou plusieurs invocations de la méthode elle même. Nous allons d'abord présenter un cas de récursivité, puis voir comment la récursivité peut s'utiliser comme alternative à une boucle. Puis nous verrons un exemple où l'utilisation de la récursivité est vraiment utile pour résoudre simplement un problème réel.

1 Une méthode qui s'appelle elle-même

Nous allons voir une méthode statique toute simple qui s'appelle elle-même. Cette méthode n'a pas d'intérêt réel si ce n'est de montrer qu'elle peut être compilée et exécutée, qu'elle ne comporte pas d'erreur en Java. Cette méthode affiche une étoile sur une ligne, puis s'invoque elle-même.

```
1 public class Rec{
2     public static void methodeRec(){
3         System.out.println("*");
4         methodeRec();
5     }
6     public static void main(String[] args){
7         methodeRec();
8     }
9 }
```

On voit ligne 4 une invocation de la méthode `methodeRec` et celle-ci est à l'intérieur de la définition de cette méthode (lignes 2 à 5).

Ce programme compile, mais à l'exécution, il y a une exception `StackOverflowError` qui est levée. En effet, ce programme ne s'arrêt jamais, car à chaque fois qu'on exécute la méthode `methodeRec`, il y a un nouvel appel à la méthode `methodeRec`. Ces appels consomment de la mémoire dans la pile jusqu'à ce qu'il n'y ait plus de place du tout dans la pile. Lorsque cela arrive, l'exception `StackOverflowError` est levée. Sur mon ordinateur, cela se produit après environ 10 000 appels de la méthode.

Nous allons modifier légèrement le programme pour que l'appel à la méthode ne soit plus systématique : nous allons le rendre aléatoire avec deux chances sur trois qu'il ait lieu. Nous utilisons pour cela la méthode statique `Math.random` qui renvoit un nombre compris entre 0 et 1, choisi aléatoirement.

```
1  public class Rec2{
2      public static void methodeRec(){
3          System.out.println("*");
4          if (Math.random() > 1.0/3.0){
5              methodeRec();
6          }
7      }
8      public static void main(String[] args){
9          methodeRec();
10     }
11 }
```

On ne sait pas combien de lignes vont être affichées et ce nombre va varier d'une exécution à l'autre. Voici des exemples d'affichages d'exécutions successives du programme.

```
> java Rec2
*
> java Rec2
*
*
> java Rec2
*
*
*
> java Rec2
*
> java Rec2
*
```

On voit sur ces exécutions que le programme s'exécute sans erreur. Il est donc possible de mettre une invocation d'une méthode dans le corps de cette méthode même.

2 Invocation récursive et boucle

Il est possible de remplacer une boucle quelle qu'elle soit par l'invocation d'une méthode récursive qui réalise l'équivalent d'un tour de boucle, y compris le calcul de la condition de boucle.

Prenons l'exemple d'une boucle for qui affiche à chaque tour de boucle la valeur du compteur de boucle. Un tour de boucle commence par le calcul de la condition. Si la condition est vraie, il y a exécution du corps, puis incrémentation. Si la condition est fausse, la boucle s'arrête.

```
1  public class Rec3{
2      public static void laBoucle(){
3          for (int i=0; i<4; i=i+1){
4              System.out.println(i);
5          }
6      }
7      public static void recursive(int i){
8          if (i<4){
9              System.out.println(i);
```

```
10              recursive(i+1);
11          }
12      }
13      public static void main(String[] args){
14          laBoucle();
15          recursive(0);
16      }
17  }
```

Permi les choses que l'on constate, plusieurs sont à noter :

— Chaque variable est locale à une exécution d'une méthode. La variable utilisée à l'intérieur de la boucle devient un paramètre de la méthode récursive. Ici, cela concerne la variable de boucle, mais cela sera le cas également de toute autre variable utilisée dans le corps de la boucle et dont la valeur est préservé d'un tour de boucle à l'autre.

— L'incrémentation se fait dans l'invocation récursive (ligne 10).

— L'initialisation à 0 de la variable de boucle est réalisée par le premier appel de la méthode récursive (dans le main, ligne 15).

3 Calcul de moyenne

Voyons un autre exemple avec cette fois une boucle qui calcule un résultat utilisé hors de la boucle. Il s'agit du calcul d'une moyenne de `n` entiers entrés au clavier. La boucle sert à saisir au clavier les nombres dont on veut faire la moyenne et à calculer leur somme. Ensuite, la somme est utilisée hors de la boucle pour calculer la moyenne.

Dans la version récursive, le résultat calculé, la somme, va être renvoyé par la méthode. Il y a trois variables passées en paramètres parce qu'elles servent à tous les tours de boucle successifs. En revanche, la variable `valeur` qui prend une valeur différente et indépendante à chaque tour de boucle n'a pas besoin d'être transmise : elle est une variable locale de la méthode récursive. Il aurait d'ailleurs été possible de la déclarer dans le corps de la boucle.

```
import java.util.Scanner;
public class Rec4{
    public static int moyenneBoucle(int n, Scanner scan){
        int valeur, res, somme = 0;
        for (int i=0; i<n; i=i+1){
            System.out.print("Entrez␣un␣nombre:␣");
            valeur = scan.nextInt();
            scan.nextLine();
            somme = somme + valeur;
        }
        res = somme/n;
        return res;
    }
    public static int calculeSommeRec(int i, int n, int somme, Scanner scan){
        int valeur, res;
        if (i<n){
            System.out.print("Entrez␣un␣nombre:␣");
            valeur = scan.nextInt();
            scan.nextLine();
            res = calculeSommeRec(i+1,n,somme+valeur,scan);
```

```
        }else{
            res = somme;
        }
        return res;
    }
    public static int moyenneAvecSommeRec(int n, Scanner scan){
        int somme, res;
        somme=calculeSommeRec(0,n,0,scan);
        res = somme/n;
        return res;
    }
    public static void main(String[] args){
        Scanner scan = new Scanner(System.in);
        System.out.println(moyenneBoucle(3,scan));
        System.out.println(moyenneAvecSommeRec(3,scan));
    }
}
```

Dans cette version, nous avons écrit la méthode `moyenneAvecSommeRec` comme un équivalent strict de la boucle de la méthode `moyenneBoucle`. En fait, on peu aller plus loin en faisant directement le calcul de moyenne à l'intérieur de la méthode récursive. Le calcul de moyenne doit être fait après la boucle, c'est-à-dire au moment où la condition est fausse. Cela correspond au cas `else` de la méthode récursive. Cela donne la méthode suivante.

```
    public static int moyenneRec(int i, int n, int somme, Scanner scan){
        int valeur, res;
        if (i<n){
            System.out.print("Entrez␣un␣nombre:␣");
            valeur = scan.nextInt();
            scan.nextLine();
            res = moyenneRec(i+1,n,somme+valeur,scan);
        }else{
            res = somme/n;
        }
        return res;
    }

    public static void main(String[] args){
        Scanner scan = new Scanner(System.in);
        System.out.println(moyenneRec(0,3,0,scan));
    }
```

4 Récursivité et pile

Il y a une étroite parenté entre une méthode récursive comportant une seule invocation récursive et une boucle. On peut quasiment mécaniquement traduire une boucle en une telle méthode et réciproquement.

D'ailleurs, la récursivité comme les boucles comporte un risque d'un programme qui ne se termine jamais. Mais alors qu'un programme qui boucle s'exécute sans fin, la récursivité en Java va prendre fin assez vite à cause de la saturation de la pile. C'est ce qui est arrivé avec le premier programme de cette s'équence de cours qui provoque une exception `StackOverflowError`.

Lors de l'exécution d'une méthode récursive, il y a allocation d'un espace mémoire nouveau à l'exécution de chaque invocation de la méthode, et cet espace reste alloué jusqu'à la fin de l'exécution des invocations suivantes. Voyons une trace d'exécution qui permet de suivre l'imbrication des différentes excéutions de la méthode. Pour traduire cette imbrication, nous allons passer en paramètre un niveau d'indentation des affichage et afficher le début et la fin d'exécution de la méthode.

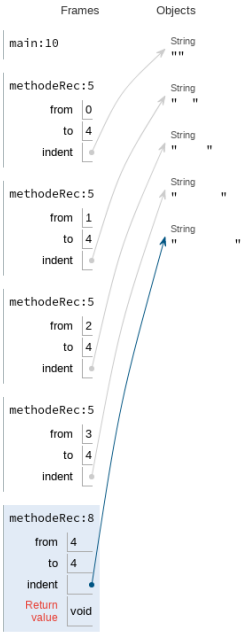
```
public class Rec5{
    public static void methodeRec(int from, int to, String indent){
        System.out.println(indent + "début␣d'exécution␣de␣methodeRec("+from+")");
        if (from<to){
            methodeRec(from+1,to, indent+"␣␣");
        }
        System.out.println(indent + "fin␣d'exécution␣de␣methodeRec("+from+")");
    }
    public static void main(String[] args){
        methodeRec(0,4,"");
    }
}
```

L'exécution du programme donne l'affichage suivant.

```
> java Rec5
début d'exécution de methodeRec(0)
  début d'exécution de methodeRec(1)
    début d'exécution de methodeRec(2)
      début d'exécution de methodeRec(3)
        début d'exécution de methodeRec(4)
        fin d'exécution de methodeRec(4)
      fin d'exécution de methodeRec(3)
    fin d'exécution de methodeRec(2)
  fin d'exécution de methodeRec(1)
fin d'exécution de methodeRec(0)
```

On voit qu'il y a un moment où quatre invocations différentes sont en cours d'exécution, une invocation correspondant à l'équivalent de chacun des quatre tours de boucle avec une valeur de la variable de boucle comprise entre 1 et 3, plus une cinquième invocation qui correspond au cas où la condition est fausse.

À ce moment-là, il y aura dans la pile 5 espaces différents avec à chque fois les paramètres `from`, `to` et `indent`. Faisons une exécution avec Pythontutor pour obtenir une image de la mémoire à ce moment-là.



On voit effectivement qu'il y a cinq espaces mémoires différents pour les cinq invocations à methodeRec. À chaque fois, il y a les trois paramètres, un dont la valeur est identique dans les cinq espaces et deux dont les valeurs changent (`from` a une valeur qui croît et `indent` une longueur qui augmente.

5 Intérêt de la récursivité

Nous avons vu que l'on peut remplacer une boucle par une méthode avec un appel récursif. Mais le véritable intérêt de la récursivité ne réside pas là. Lorsqu'on a le choix entre une simple boucle et une méthode récursive qui tend à occupper beaucoup de place dans la pile, on peut raisonnablement préférer la boucle.

La récursivité est en revanche bien plus intéressante lorsqu'il y a plusieurs appels récursifs dans la même méthode. Nous allons en donner un exemple ici avec un labyrinthe. On utilise un damier pour représenter un labyrinthe. Chaque case peut être occupée par un mur ou non. Un pion va être posé sur une case du damier. Il peut se déplacer vers les 4 points cardinaux. Il ne peut pas se déplacer en diagonale. Pour simplifier le codage, on suppose qu'il y a des murs sur tout le pourtour du tableau. Cela fait que chaque case où peut se trouver le pion est entourée de 4 voisines. Il n'y a pas de cas particulier à considérer sur les bords et dans les coins du damier.

La question que l'on se pose est : quelles cases le pion peut atteindre après un nombre illimité de déplacements ? Nous allons représenter notre damier par un tableau de caractères dans lequel les murs sont représentés par un X majuscule et les couloirs par un espace.

La méthode de calcul récursive que l'on va utiliser implémente le principe suivant. Le pion peut atteindre la case sur laquelle il est placé. Cette case est accessible en 0 déplacements. Ensuite, il peut essayer d'aller sur chacune des cases voisines, dans les 4 directions. Si cette case voisine est un mur, elle n'est pas accessible. Si elle est un couloir, elle est accessible et de là, le pion peut essayer d'aller sur chacune des voisines. Le même raisonnement est à appliquer sur chacune des voisines, puis des voisines des voisines, etc.

Voici le code du programme, avec une méthode main qui applique la méthode récursive sur un exemple de labyrinthe.

```
1  public class AccessibiliteCase{
2      public static void marqueAccessibleCaseEtVoisines(int x, int y,
3                                                        char[][] tab){
4          if (tab[x][y]==' '){
5              tab[x][y] = 'ř';
6              marqueAccessibleCaseEtVoisines(x-1,y,tab);
7              marqueAccessibleCaseEtVoisines(x+1,y,tab);
8              marqueAccessibleCaseEtVoisines(x,y-1,tab);
9              marqueAccessibleCaseEtVoisines(x,y+1,tab);
10         }
11     }
12     public static void afficheTableau(char[][] tab){
13         for (int i=0;i<tab[0].length;i++){
14             for (int j=0; j<tab.length; j++){
15                 System.out.print(tab[j][i]);
16             }
17             System.out.println();
18         }
19     }
20     public static void main(String[] args){
21         char[][] tab = {{'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'},
22                         {'X', ' ', ' ', ' ', ' ', ' ', 'X', 'X'},
23                         {'X', ' ', ' ', ' ', ' ', ' ', 'X', 'X'},
24                         {'X', ' ', ' ', ' ', ' ', ' ', 'X', 'X'},
25                         {'X', 'X', 'X', ' ', ' ', 'X', ' ', 'X'},
26                         {'X', ' ', 'X', 'X', ' ', 'X', ' ', 'X'},
27                         {'X', ' ', ' ', 'X', 'X', 'X', ' ', 'X'},
28                         {'X', ' ', ' ', ' ', ' ', ' ', ' ', 'X'},
29                         {'X', ' ', ' ', ' ', ' ', ' ', ' ', 'X'},
30                         {'X', ' ', ' ', ' ', ' ', ' ', ' ', 'X'},
31                         {'X', 'X', 'X', 'X', ' ', 'X', 'X', 'X'},
32                         {'X', ' ', ' ', ' ', ' ', 'X', ' ', 'X'},
33                         {'X', ' ', ' ', ' ', ' ', 'X', ' ', 'X'},
34                         {'X', ' ', 'X', 'X', 'X', 'X', ' ', 'X'},
35                         {'X', ' ', ' ', ' ', ' ', 'X', ' ', 'X'},
36                         {'X', 'X', 'X', 'X', ' ', 'X', 'X', 'X'},
37                         {'X', ' ', ' ', ' ', ' ', 'X', ' ', 'X'},
38                         {'X', ' ', 'X', 'X', 'X', 'X', ' ', 'X'},
39                         {'X', ' ', ' ', ' ', ' ', ' ', ' ', 'X'},
40                         {'X', 'X', 'X', 'X', 'X', 'X', ' ', 'X'},
41                         {'X', ' ', 'X', ' ', ' ', ' ', ' ', 'X'},
42                         {'X', ' ', 'X', ' ', 'X', 'X', 'X', 'X'},
43                         {'X', ' ', 'X', ' ', ' ', ' ', ' ', 'X'},
44                         {'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'}};
45         System.out.println("Voici le labyrinthe:\n");
46         afficheTableau(tab);
47         marqueAccessibleCaseEtVoisines(7,3,tab);
48         System.out.println("\n\nLes cases accessibles depuis la case (7,3)" +
49                            " sont marquées par un rond: \n");
50         afficheTableau(tab);
```

```
51      }
52  }
```

On voit lignes 6 à 9 qu'il y a quatre invocations de méthodes récursives, des invocations de la méthode `marqueAccessibleCaseEtVoisines` : il s'agit de la visite des quatre cases voisines, une dans chacune des directions.

Un autre élément clé dans la méthode récursive est l'instruction `if` qui fait que lorsque la méthode est appelée sur une case qui contient autre chose qu'un espace, c'est-à-dire si la case n'est pas un couloir vide, la méthode ne fait rien : elle ne marque pas la case comme accessible et elle n'appelle pas récursivement la méthode sur les 4 voisines. Cela correpond à deux cas : la case est un mur. Il est normal de ne pas la noter comme accessible. Le second cas est celui d'une case déjà notée accessible avec un petit rond. Cela signifie que cette case a déjà été visitée et l'appel sur les 4 voisines a déjà été fait.

Dans ce programme, il y a un vrai risque de tourner en rond, car la notion de voisine est une relation symétrique : une case est la voisine de sa voisine. Dans l'exploration de toutes les voisines, il y aurait un risque de tourner en rond en revenant toujours sur les mêmes cases. Un risque de faire des calculs en boucle. Ce risque est éliminé par le fait de marquer les cases déjà visitées avec le rond.