Chapitre IV : La récursivité

- Introduction
- La pile d'exécution
- Récursivité imbriquée
- Récursivité croisée
- Récursivité terminale et non terminale

Introduction

• Un programme est dit récursif s'il s'appelle lui-même → Il s'agit forcément d'une fonction.

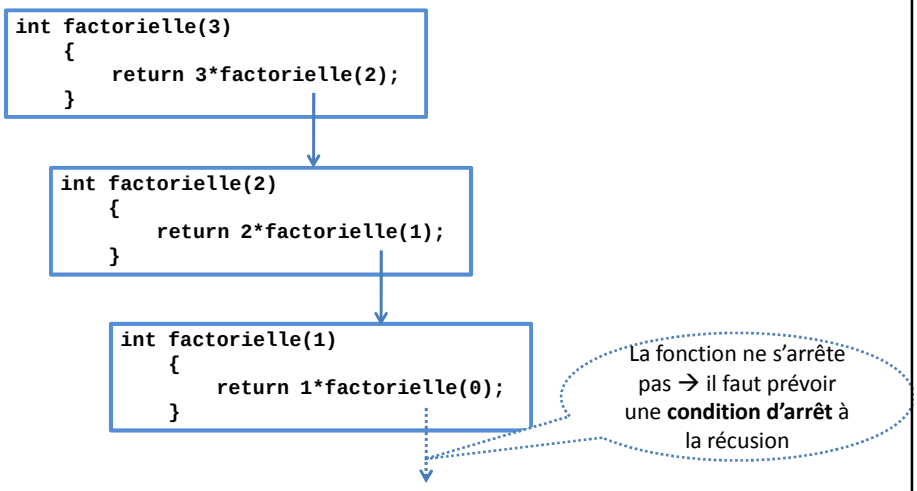
Exemple : la factorielle, **n! = 1 x 2 x … x n** donc **n! = n x (n-1)!**

```
public static int factorielle(int n)
    {
        return n*factorielle(n-1);
    }
```

• **Objectif :** La récursivité permet de décomposer un problème en sous-problèmes "plus simples". A leur tour, ces sous-problèmes seront décomposés jusqu'à un niveau d'opérations "élémentaires", faciles à réaliser.

Introduction

• L'appel récursif est traité comme n'importe quel appel de fonction.



Introduction

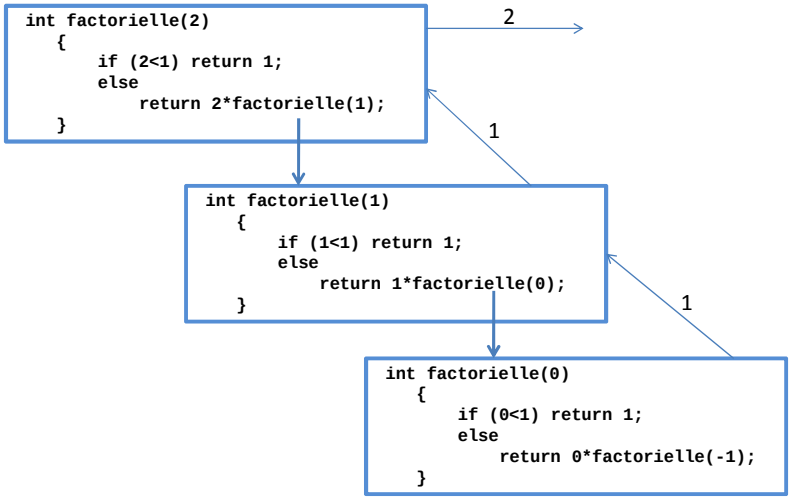
• La factorielle avec condition d'arrêt :

si n <1 n! = 1 sinon n! = n x (n-1)!.

```
public static int factorielle(int n)
    {
        if (n<1) return 1;
        else
            return n*factorielle(n-1);
    }
```

Introduction

• La fonction factorielle s'arrêt au bout d'un certaines récursions :



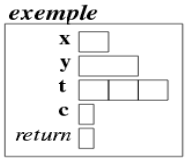
La pile d'exécution

• Chaque sous-programme (le programme principal "main" aussi) utilise une zone de mémoire pour stocker ses paramètres et ses variables locales, de plus, une fonction réserve aussi dans sa zone de mémoire une place pour le résultat retourné.

Exemple :

```
boolean exemple ( int x, double y)
{ int [] t = new int[3];
char c;
...
}
```

La zone mémoire occupée par cette fonction :



• L'allocation de cette zone de mémoire se fait au moment de l'appel du sous-programme , dans une zone de mémoire spéciale du programme, appelé **la pile d'exécution**.

La pile d'exécution

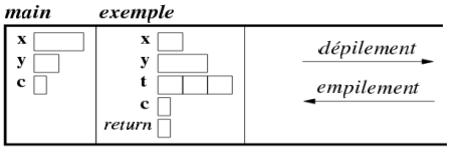
• Les zones de mémoire des sous-programmes sont empilées suivant l'ordre des appels et dépilées dès que le sous-programme se termine.

• La zone de mémoire d'un sous-programme n'existe physiquement que pendant que le sous-programme est en cours d'exécution.

Exemple :

```
public class Principal{
      public static boolean exemple (int x, double y){ ... }
      public static void main(String[] args){
               double x;int y; boolean c;
               .........
               c = exemple(y, x);
               .........
             }
        }
```

• Le contenu de la pile pendant l'appel de la méthode exemple :

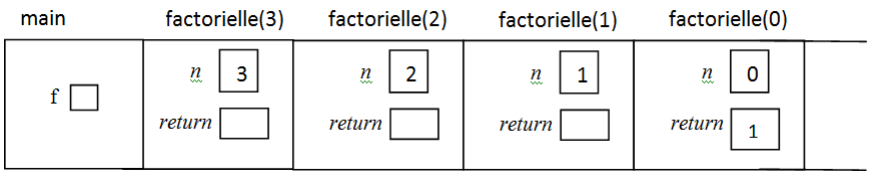


La pile d'exécution

Exemple :

```
public class LaFacto{
    public static void main(String[] args) {
        int f;
        f=factorielle(3);
        System.out.println("La facotrielle : "+f);
    }
    public static int factorielle(int n)
    {
        if (n<1) return 1;
        else return n*factorielle(n-1);
    } }
```

•L'état de la pile d'exécution pendant l'appel de factorielle(0)



La pile d'exécution

Exemple : Débordement de la pile d'exécution

```
public class TestPileExec{
    public static void main(String[] args) {
        testPile(1);
    }
    public static void testPile(int n)
    {
        System.out.println("Numéro d'appel : "+n);
        testPile(n+1);
    }
}
```

• Exécuter trop d'appels de fonction fera déborder la pile d'exécution!

```
Numéro  d'appel  : 1
Numéro  d'appel  : 2
...
Numéro  d'appel  : 5613
```

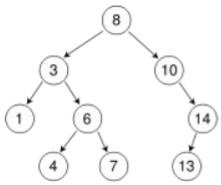
Exception in thread "main" java.lang.StackOverflowError

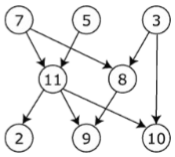
Itératif versus récursif

• Il est souvent possible d'écrire un même programme en itératif et en récursif.

• L'exécution d'une version récursive d'un programme est généralement un peu moins rapide que celle de la version itérative, même si le nombre d'instructions est le même (à cause de la gestion des appels de fonction).

• Sur des structures de données naturellement récursives, il est bien plus facile d'écrire des programme récursifs qu'itératifs.





• Certains programme sont extrêmement difficiles à écrire en itératif

Récursivité imbriquée

• La récursivité imbriquée consiste à faire un appel récursif à l'intérieur d'un autre appel récursif .

Exemple : la fonction d'ackerman

A(m,n) = n+1 si m = 0,

A(m,n) = A(m-1,1) si n=0 et m > 0

A(m,n) = A(m-1, A(m,n-1)) sinon

•La méthode Java correspondante

```
public static int ack(int m,int n)
  {
   if (m==0) return (n+1);
   else
       if(n==0 && m>0) return ack(m-1,1);
       else return ack(m-1,ack(m,n-1));
  }
```


Récursivité croisée

• La récursivité croisée consiste à écrire des fonctions qui s'appellent l'une l'autre.

Exemple :

```
public static boolean pair(int n)
    {
        if (n==0) return true;
        else
             if(n==1) return false;
             else return impair(n-1);
    }

public static boolean impair(int n)
    {
        if (n==0) return false;
        else
             if(n==1) return true;
             else return pair(n-1);
    }
```

Récursivité terminale et non terminale

• Une fonction récursive est dite terminale si aucun traitement n'est effectué à la remontée d'un appel récursif (sauf le retour d'une valeur).

• Une fonction récursive est dite non terminale si le résultat de l'appel récursif est utilisé pour réaliser un traitement (en plus du retour d'une valeur).

Exemple 1 : Forme Non terminale de la fonction factorielle

```
public static int factorielle(int n)
    {
        if (n<1) return 1;
        else
            return n*factorielle(n-1);
    }
```

Exemple 1 : Forme terminale de la fonction factorielle

```
public static int factorielle(int n,int res)
    {
         if(n<1)
             return res;
         else
             return factorielle(n-1,n*res);
    }
```

la fonction doit être appelée en mettant toujours res à 1

Récursivité terminale et non terminale

• Une fonction récursive terminale est en théorie plus efficace (mais souvent moins facile à écrire) que son équivalent non terminale : il n'y a qu'une phase de descente et pas de phase de remontée.

• En récursivité terminale , les appels récursifs n'ont pas besoin d'êtres empilés dans la pile d'exécution car l'appel suivant remplace simplement l'appel précédent dans le contexte d'exécution.