Chapitre 5

La récursivité

Les fonctions récursives
Les structures de données récursives

1. LA RECURSIVITE 2

1.1. INTRODUCTION 2
1.2. UN TRAITEMENT RECURSIF : CALCUL DE FIBONACCI 2
1.3. UN AUTRE EXEMPLE: LE FACTORIEL 4

2. OPPOSITION RECURSIVITE ET ITERATION 4

3. LES STRUCTURES DE DONNEES RECURSIVES 5

3.1. LES LISTES CHAINEES 5
3.2. LES ARBRES BINAIRES 6
3.3. LES ARBRES N-AIRES 8
3.4. UTILISATION DES ARBRES 8

4. LA PROBLEMATIQUE EN JAVA 9

5. CONCLUSION 10

1. La récursivité

1.1. Introduction

Définition :

La récursivité est la propriété d'un langage informatique d'appeler par un traitement, le traitement lui-même, et d'empiler les paramètres d'appel.

Le principe de récursivité "s'oppose" au principe itératif qui utilise des boucles de variables.

Principe :

Lors de l'appel d'un sous-programme, le programme **empile** au-dessus des données précédentes (données globales et locales du sous-programme appelant), les valeurs des paramètres d'appel puis réalise le sous-programme.

Si ce même sous-programme s'appelle lui-même, il empile à nouveau par-dessus les nouvelles valeurs des paramètres d'appel et des variables locales, et ainsi de suite pour chaque appel.

Puis, au retour de chaque appel, le programme dépile les paramètres concernés et les variables locales.

Le sous-programme doit contenir un test d'arrêt de récursivité.

1.2. Un traitement récursif : calcul de fibonacci

La série dite "de fibonacci" est définie de la manière suivante :
fibo de 0 = 0
fibo de 1 = 1
fibo de n = fibo de n-1 + fibo de n-2

On obtient la série suivante :
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55

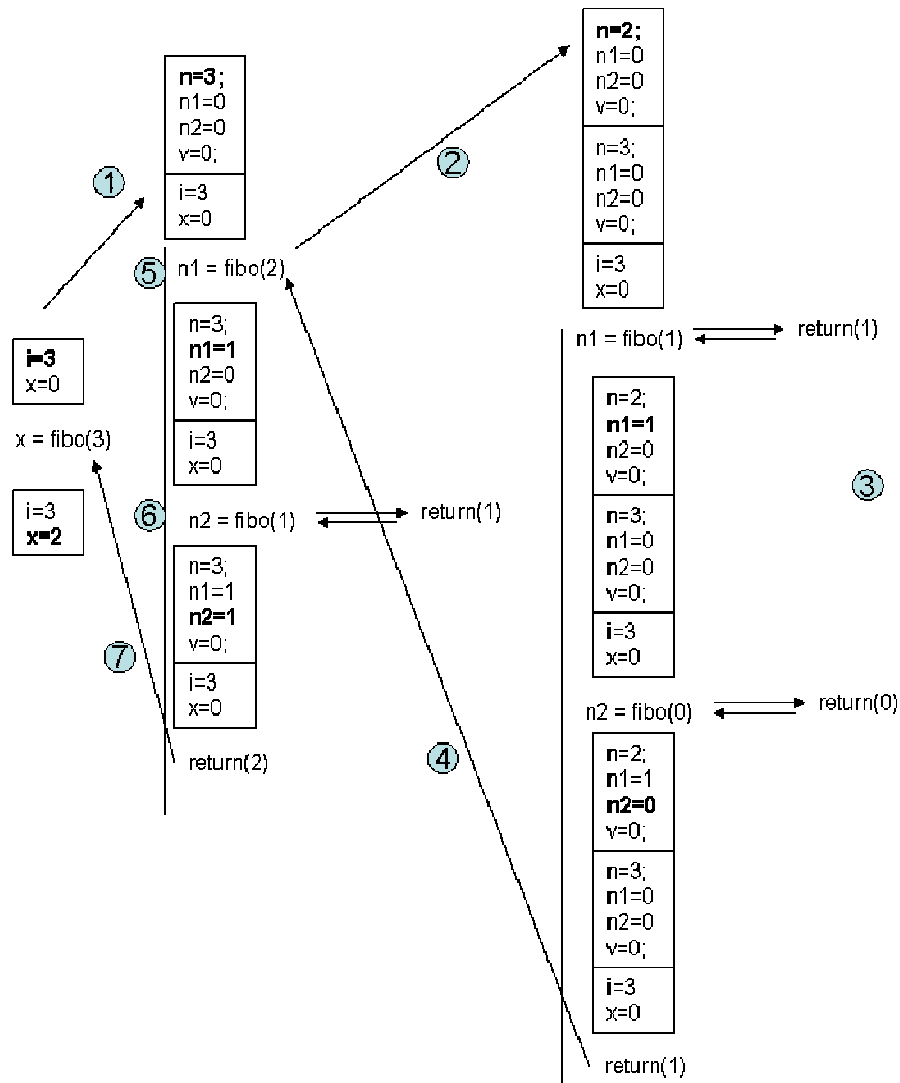
Voici le code de manière récursive :

```
static int fibo(int n)
{
    int n1,n2,v;
    if (n<=1) return n; // si n=1 alors 1, si n=0 alors 0

    n1 = fibo(n-1);
    n2 = fibo(n-2);
    v = n1 + n2;

    return (v);
}
```

Prenons l'exemple de l'appel :

```
int i,x;
i = 3;
x = fibo ( i );
```

Comment se comporte le programme:



1/ Appel de 1er niveau à la méthode fibo. Les paramètres et les variables locales sont empilés. L'exécution de la méthode se fait avec ces valeurs.

2/ Appel du 2ème niveau à la méthode fibo. La aussi, les paramètres et les variables locales sont empilées. L'exécution de la méthode se fait avec ces nouvelles valeurs.

3/ et 4/ Dans ces deux cas la récursivité est "arrêtée" dans les sens où on n'appelle plus la méthode mais on retourne une valeur particulière.

On obtient alors un résultat 1 + 0 = 1

5/ Le résultat de 1 est retourné à l'appelant.

La pile est dépilée du contexte d'appel.

6/ La méthode de 1er niveau continue donc à s'exécuter avec toujours le même contexte d'appel initial. On comprend donc bien que la valeur de **n** n'a pas changée (le contexte est le même).

7/ On fait donc appel à fibo(n-2) qui est fibo(3-2)=fibo(1) qui retourne 1.

8/ Le résultat 1+1 est retourné.

La valeur de x est donc 2.

ATTENTION, aux paramètres qui sont des objets (pointeurs).

Au retour de l'appel, le paramètre est bien restauré (c'est le même) <u>mais pas le contenu pointé</u> qui a pu évoluer au fur et à mesure des appels récursifs.

1.3. Un autre exemple: le factoriel

```
static int fact(int n)
{
    if (n==0) return 1;
    else return n * fact(n-1);
}
```

On voit bien la puissance synthétique et descriptive d'une telle méthode pour décrire un traitement.

2. Opposition Récursivité et Itération

On oppose souvent les deux principes.

On peut s'apercevoir que la première englobe le deuxième. C'est-à-dire que tout traitement itératif peut s'écrire en récursivité mais pas l'inverse (sans simuler le fonctionnement d'une pile).

Il existe des langages comme LISP qui, historiquement; ne contient pas de boucle et tout se fait avec de la récursivité. Mais aussi, la structure des données manipulée est également récursive.

De plus, il existe des problèmes informatiques dont les solutions sont simples en récursivité et deviennent très compliqués en itératif.

L'exemple le plus célèbre est celui des tous de Hanoï.

Le problème des tours de Hanoï :

Le problème des tours de Hanoï est un jeu de réflexion imaginé par le mathématicien français Édouard Lucas, et consistant à déplacer des disques de diamètres différents d'une tour de « départ » à une tour d'« arrivée » en passant par une tour « intermédiaire » et ceci en un minimum de coups, tout en respectant les règles suivantes :

- ***on ne peut déplacer plus d'un disque à la fois,***
- ***on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.***

On suppose que cette dernière règle est également respectée dans la configuration de départ.

Solution :

```
static void hanoi(int n,String a,String b,String c)
{
    if (n!=0)
        {
            hanoi(n-1,a,c,b);
            System.out.println("deplacer de "+a+" vers "+ c);
            hanoi(n-1,b,a,c);
        }
}
```

Et c'est tout !!

```
Exécution pour 3 :
  hanoi(3,"A","B","C");

  deplacer de A vers C
  deplacer de A vers B
  deplacer de C vers B
  deplacer de A vers C
  deplacer de B vers A
  deplacer de B vers C
  deplacer de A vers C

Exécution pour 4 :
  hanoi(4,"A","B","C");

  deplacer de A vers B
  deplacer de A vers C
  deplacer de B vers C
  deplacer de A vers B
  deplacer de C vers A
  deplacer de C vers B
  deplacer de A vers B
  deplacer de A vers C
  deplacer de B vers C
  deplacer de B vers A
  deplacer de C vers A
  deplacer de B vers C
  deplacer de A vers B
  deplacer de A vers C
  deplacer de B vers C
```

Face à la complexité apparente du problème ont peut être "bluffé" par la simplicité d'écriture du programme.

Mais, attention à l'explosion combinatoire des appels récursifs. Il peut y avoir une forte consommation de la mémoire : la pile augmente.
Le maximum mémoire d'une pile est le chemin le plus long dans l'arborescence des appels.
C'est pourquoi, pour des appels nombreux on préfère utiliser des boucles qui par définition ne consomment pas de mémoire.

3. Les structures de données récursives

Un type T d'une structure de données est récursive si un de ses attributs est défini sur T.
Cela marche aussi s'il y a un type intermédiaire.

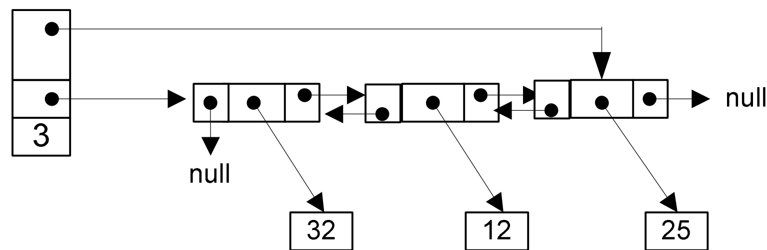
En programmation objet, on parle de classe récursive.

Dans cette définition sont englobées les structures de données définies par le système d'exploitation comme l'arborescence des répertoires qui sous-entend une structuration arborescence
et donc une définition récursive en interne des données du système d'exploitation.

3.1. Les listes chainées

Les listes chaînées sont des collections constituées de "cellules" chaînées les unes aux autres par un pointeur. Chaque cellule contient la valeur de la collection.

Un exemple de représentation avec double chainage :




```
public class ListeChainee
{
    private int nb;         // Nombre d'element de la liste
    // Cet attribut permet de ne pas parcourir tous les elements
    //  afin de les compter
    private Cellule prem;   // Pointe sur le 1er element de la liste
    private Cellule dern;   // Pointe sur le deuxieme element de la liste
}

class Cellule
{
    Cellule prec;  // Cellule precedente
    Object  value; // Valeur de la cellule
    Cellule suiv;  // Cellule suivante
}
```

Voir sur le site http://jacques.laforgue.free.fr l'exemple **Exemple44_ListeChainee**

Dans cet exemple :

- tous les traitements itératifs sont écrits en récursivité bien qu'il soit possible d'écrire ces traitements avec une boucle while.
- le type d'élément est Object cela signifie que grâce à l'auto-boxing, cette liste permet de gérer n'importe quel élément de type primitif ou d'une classe (qui implémente la méthode toString).
- le double chainage permet d'optimiser l'insertion et permet de parcourir les éléments dans les deux sens.
- implémentation de méthodes permettant de se déplacer dans le chainage

3.2. Les arbres binaires

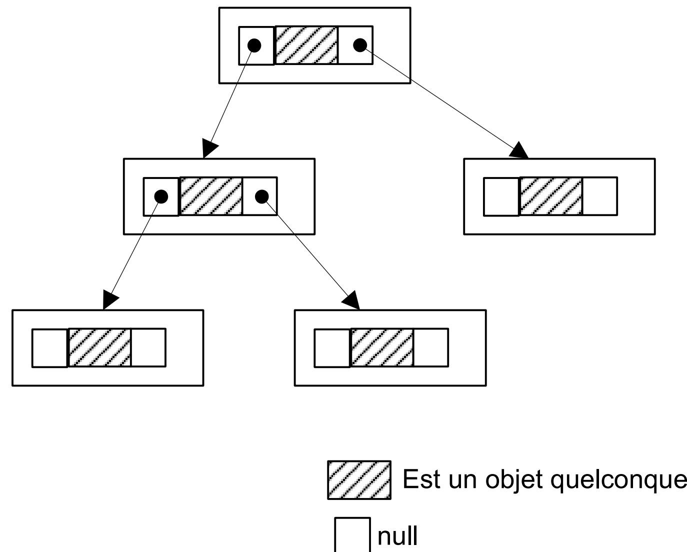
L'arbre est un concept mathématique qui est un cas particulier de la théorie des graphes.
Il ne s'agit pas ici de faire un cours sur les graphes mais de voir comment il est possible d'implémenter un arbre pour ensuite l'utiliser.

Un arbre est un nœud qui est caractérisé par 3 informations :

- la valeur du nœud qui est un objet quelconque. Cela permet d'y mettre n'importe quelle information qui peut être complexe ;
- la référence vers le sous-arbre gauche qui est donc un arbre ;

- la référence vers le sous-arbre droit qui est donc aussi un arbre ;

Les arbres sont donc des structures de données par définition récursives : un arbre est constitué de deux sous-arbres.



Est un objet quelconque

null

```
public class Arbre
{
   private Object value;
   private Arbre gauche;
   private Arbre droit;
}
```

On voit ici que la classe Arbre a deux attributs de type Arbre : la donnée est récursive.

Deuxième implémentation possible :

```
public class Arbre
{
    private ArrayList<Object> noeud;
}
```

Dans cette implémentation, il n'est pas visible que la classe Arbre contient d'autres Arbre.

Il faut comprendre que :

- nœud.get(0) est de type Object et contient la valeur du nœud
- nœud.get(1) est de type Object et contient un Arbre (un Arbre est un Object) : l'arbre gauche.
- nœud.get(2) est de type Object et contient un Arbre : l'arbre Droit.

Le choix de cette implémentation est donc d'utiliser la classe **ArrayList** qui est une donnée récursive et polymorphe car n'importe quel élément peut être à son tour un ArrayList.

La donnée ArrayList est une donnée récursive.

On peut donc créer des arbres (binaires ou n-aires) sans la nécessité de créer une classe.
Mais il est plus "objet" de créer une classe comme on va le voir dans l'exemple.

On peut donc créer des arbres sans la nécessité de créer une classe

Implémentation de la 1ère solution :

Voir sur le site http://jacques.laforgue.free.fr l'exemple **Exemple43_ArbreBinaire**

3.3. Les arbres n-aires

Un arbre n-aire est un nœud qui est caractérisé par 2 informations :

- la valeur du nœud qui est un objet quelconque. Cela permet d'y mettre n'importe quelle information qui peut être complexe ;
- une collection (tableau, ArrayList, ...) qui contient la liste des références vers les sous-arbres qui sont donc des arbres ;

```
public class Arbre
{
    private Object value;              // Valeur du noeud
    private ArrayList<Arbre> fils;     // Les sous-arbres
}
```

Voir sur le site http://jacques.laforgue.free.fr l'exemple **Exemple42_Arbre**

3.4. Utilisation des Arbres

Les arbres sont utilisés dans de très nombreux domaines : base de données, intelligence artificielle, réseau, système, scientifiques, ...

Par exemple :

- gestion des arborescences de répertoires et de fichiers;
- optimisation d'accès aux données en séquentielles indexées
- programme d'aide à la décision (jeux, simulation du raisonnement)
- programmation des réseaux sémantiques
- recherche opérationnel (graphe)

Au-delà des arbres, il y a aussi les graphes dont les arbres est un cas particulier.

4. La problématique en JAVA

Dans les exemples précédents, nous avons utilisé la récursivité dans 2 cas :

- le calcul par récurrence en utilisant des données Java primitives (int, double, …) qui sont empilées sur la pile d'exécution
- le parcours dans une donnée récursive en utilisant des "pointeurs" (pointeur sur le nœud d'un arbre) sur ces données en paramètres des appels récursives

Le principe fort de la récursivité est le suivant :

- on est dans une méthode qui "analyse" un certain **état** passé en paramètre d'entrée
- et dans cette méthode on veut appeler (récursivement) la même méthode en mettant en entrée de ce nouvel appel **un nouvel état**
- et on veut que en retour de cet appel, l'état courant ne soit pas altéré par l'appel récursif.

  Faire un schéma

Nous sommes en programmation objet. Cet état est donc un objet.

Dans certains langages comme le C++, un objet n'est pas nécessairement un pointeur mais une données structurée qui quand on la passe en paramètre (passage par valeur), cet objet est copié dans son intégralité sur la pile d'exécution du nouvel appel. Ce qui entraine que lors du retour de l'appel, l'objet initial n'est pas altéré, dans le cas où la méthode modifie l'objet..

Hors en JAVA, un objet est un pointeur. En JAVA, on passe donc toujours un objet en passant son pointeur.

Il faut donc passer en paramètre une copie de l'objet.

pour cela, en JAVA, on peut utiliser la méthode clone() (qu'il faut coder).

Exemple : le problème des huit reines en récursivité.

Voir sur le site http://jacques.laforgue.free.fr l'exemple **Exemple45_PbDes8Reines**

Dans le cadre de NFA031, un exercice avait été proposé (Exercice07_PbDes8Reines) dans lequel une programmation non récursive avait été proposée.

On peut donc en les comparants, voir la simplicité de l'algorithme récursif :

```
public Echiquier resoudre()
{
  if (valide())
    {
      if (nbReines()==8)
        return this;
      else
        {
          for(Point p:getCasesVides())
            {
              Echiquier e = this.clone();
              e.setCase(p.x,p.y,1);
              Echiquier sol = e.resoudre();
              if (sol!=null) return sol;
            }
          return null;
        }
    }
  else return null;
}
```

5. Conclusion

La récursivité ne doit pas remplacer les boucles pour deux raisons :

- la consommation mémoire éventuellement excessive
- la complexité de l'écriture de certains programmes car pour chaque boucle, il faut créer une méthode récursive

On réserve donc la récursivité à l'écriture de programme qui induit une démarche récursive, soit les domaines de l'informatique suivante :

- gestion d'arbre de données (binaire ou n-aire)
- analyse syntaxique, sémantique
- compilateur et interpréteur
- transformation de format (XML)
- implémentation des bases de données
- calcul scientifique (équations récursives)
- programmation de jeu (stratégie)
- … etc …