Algorithmique Programmation
– 2ème partie –

DUT en alternance CNAM

2007 - 2008

Table des matières

Chapitre 1

Premiers Pas en Programmation Objet : les Classes et les Objets

Dans la première partie de ce cours, nous avons appris à manipuler des objets de type simple : entiers, doubles, caractères, booléens. Nous avons aussi appris comment utiliser les tableaux pour stocker des collections d'objets de même type : tableaux d'entiers, tableaux de booléens...Cependant, la majorité des programmes manipulent des données plus complexes. Pour ecrire un logiciel bancaire, il faudra représenter dans notre langage de programmation l'ensemble des informations caractérisant un compte et coder les actions qui s'effectuent sur les comptes (retrait, dépot) ; un logiciel de bibliothèque devra représenter l'ensemble des informations caractéristiques des livres et coder les opérations d'ajout ou de retrait d'un livre ....
L'approche Orientée Objet, que nous allons aborder dans ce chapitre, consiste à rendre possible dans le langage de programmation la définition d'objets (des livres, des comptes ...) qui ressemblent à ceux du monde réel, c'est à dire caractérisés par un état et un comportement. L'état d'un compte, pourra être défini par son numéro, le nom de son titulaire, son solde ; son comportement est caractérisé par les opérations de dépot, de retrait et d'affichage du solde.

Dans nos programmes nous aurons plusieurs objets comptes. Chacuns ont un état qui leur est propre, mais ils ont les mêmes caractéristiques : ce sont tous des comptes. En programmation Orientée Objet, nous dirons que ces différents objets comptes sont des objets instances de la classe Compte. Une classe est un prototype qui définit les variables et les méthodes communes à tous les objets d'un même genre. Une classe est un *patron d'objets*. Chaque *classe* définit la façon de créer et de manipuler des *Objets* de ce type.
A l'inverse, un objet est toujours un exemplaire, une instance d'une classe (son patron).
Ainsi, pour faire de la programmation Objet, il faut savoir concevoir des classes, c'est à dire définir des modèles d'objets, et créer des objets à partir de ces classes.
Concevoir une classe, c'est définir :

1. **Les données** caractéristiques des objets de la classe. On appelle ces caractéristiques les *variables d'instance*.
2. **Les actions** que l'on peut effectuer sur les objets de la classe . Ce sont les *méthodes* qui peuvent s'invoquer sur chacuns des objets de la classe.

Ainsi, chaque objet crée possèdera :

1. **Un état**, c'est à dire des valeurs particulières pour les variables d'instances de la classe auquel il appartient.
2. **Des méthodes** qui vont agir sur son état.

1.1 Définir une Classe

Une classe qui définit un type d'objet a la structure suivante :

- Son nom est celui du type que l'on veut créer.
- Elle contient les noms et le type des caractéristiques (les variables d'instances) définissant les objets de ce type.
- Elle contient les méthodes applicables sur les objets de la classe.

Pour définir une classe pour les comptes bancaires on aurait par exemple :

```
class Compte {
   int solde;
   String titulaire;
   int numero;

   void afficher(){
     Terminal.ecrireString(``solde''+ this.solde);
   }
   void depot(int montant){
     this.solde = this.solde+ montant;
   }
   void retrait(int montant){
     this.solde=this.solde-montant;
   }
}
```

1.1.1 Les Variables d'instances

La déclaration d'une variable d'instance se fait comme une déclaration de variable locale au main ou à un sous programme : on donne un nom, précédé d'un type. La différence est que cette déclaration se fait **au niveau de la classe** et non a l'intérieur d'un sous programme.

Nous avons dans la classe `Compte` 3 variables d'instance : `solde` destinée à recevoir des entiers, `titulaire` destinée à recevoir une chaine de caractères et `numero` destinée à recevoir des entiers.

Ainsi l'état de chaque objet instance de la classe `compte` que nous créerons par la suite sera constitué d'une valeur pour chacune de ces trois variables d'instances. Pour chaque objet instance de la classe `Compte` nous pourrons connaitre la valeur de son solde, le nom de son titulaire et le numéro du compte. Ces valeurs seront **propres à chaque objet**.

1.1.2 Les méthodes : premier aperçu

Nous n'allons pas dans ce paragraphe décrire dans le détail la définition des méthodes d'objets, mais nous nous contentons pour l'instant des remarques suivantes : une classe définissant un type d'objets comportera autant de méthodes qu'il y a d'opérations utiles sur les objets de la classe.

La définition d'une méthode d'objet (ou d'instance) ressemble à la défintion d'un sous programme : un type de retour, un nom, une liste d'arguments précédés de leur type. Ce qui fait d'elle une méthode d'objet est qu'elle ne comporte pas le mot clé `static`. Ceci (plus le fait que les méthodes sont dans la classe `Compte`) indique que la méthode va pouvoir être invoquée (appelée) sur n'importe quel objet de type `Compte` et modifier son état (le contenu de ses variables d'instances).

Ceci a aussi des conséquences sur le code de la méthode comme par exemple l'apparition du mot clé `this`, sur lequel nous reviendrons lorsque nous saurons comment invoquer une méthode sur un objet.

1.2 Utiliser une Classe

Une fois définie une classe d'objets, on peut utiliser le nom de la classe comme un nouveau type : déclaration de variables, d'arguments de sous programmes ..... On pourra de plus appliquer sur les objets de ce type toutes les méthodes de la classe.

1.2.1 Déclarer des objets instances de la classe

Si la classe `Compte` est dans votre répertoire de travail, vous pouvez maintenant écrire une autre classe, par exemple `test` qui, dans son main, déclare une variable de type `Compte` :

```
public class test  {
  public static void main (String [] arguments){
    Compte c1 = new Compte();
  }
}
```

Comme pour les tableaux, une variable référencant un objet de la classe `Compte`, doit recevoir une valeur, soit par une affectation d'une valeur déjà existante, soit en créant une nouvelle valeur avec `new` avant d'être utilisée. On peut séparer la déclaration et l'initialisation en deux instructions :

```
public class test  {
  public static void main (String [] arguments){
    Compte  c1;
    c1 = new Compte();
  }
}
```

Après l'exécution de `c1 = new Compte();` chaque variable d'instance de `c1` a une valeur par défaut. Cette valeur est 0 pour `solde` et `numero`, et `null` pour `titulaire`.

1.2.2 Accéder et modifier les valeurs des variables d'instances d'un objet

La classe `Compte` définit la forme commune à tous les comptes. Toutes les variable de type `Compte` auront donc en commun cette forme : un solde, un titulaire et un numéro. En revanche, elles pourront représenter des comptes différents.

Accéder aux valeurs des variables d'instance

Comment connaître le solde du compte `c1` ? Ceci se fait par l'**opérateur noté par un point** :

```
public class test  {
  public static void main (String [] arguments){
    Compte c1 = new Compte();
    Terminal.ecrireInt(c1.solde);
  }
}
```

La dernière instruction à pour effet d'afficher à l'écran la valeur de la variable d'instance `solde` de `c1`, c'est à dire l'entier 0. Comme le champ `solde` est de type `int`, l'expression `c1.solde` peut s'utiliser partout où un entier est utilisable :

```
public class test {
  public static void main (String [] arguments){
    Compte c1 = new Compte();
    int x;
    int []tab = {2,4,6};
    tab[c1.solde]= 3;
    tab[1]= c1.numero;
    x = d1.solde +34 / (d1.numero +4);
  }
}
```

Modifier les valeurs des variables d'instance

Chaque variable d'instance se comporte comme une variable. On peut donc lui affecter une nouvelle valeur :

```
public class test {
  public static void main (String [] arguments){
    Compte c1 = new Compte();
    Compte c2 = new Compte();
    c1.solde =100;
    c1.numero=218;
    c1.titulaire=''Dupont'';
    c2.solde =200;
    c2.numero=111;
    c2.titulaire=''Durand'';

    Terminal.ecrireStringln
    ("valeur de c1:" + c1.solde + " ,"+ c1.titulaire + " ," + c1.numero);
    Terminal.ecrireStringln
    ("valeur de c2:" + c2.solde + " ,"+ c2.titulaire + " ," + c2.numero);
  }
}
```

c1 représente maintenant le compte numero 218 appartenant à Dupont et ayant un solde de 100 euros. et c2 le compte numero 111 appartenant à Durand et ayant un solde de 200 euros.

Affectation entre variables référençant des objets

l'affectation entre variables de types `Compte` est possible, puisqu'elles sont du même type, mais le même phenomène qu'avec les tableaux se produit : les 2 variables référencent le même objet et toute modification de l'une modifie aussi l'autre :

```
public class testBis {
  public static void main (String [] arguments){
    Compte c1 = new Compte();
    Compte c2 = new Compte();
```

```
    c1.solde =100;
    c1.numero=218;
    c1.titulaire=''Dupont'';
    c2 = d1;
    c2.solde = 60;
    Terminal.ecrireStringln
    ("valeur de c1:" + c1.solde + " ,"+ c1.titulaire + " ," + c1.numero);
    Terminal.ecrireStringln
    ("valeur de c2:" + c2.solde + " ,"+ c2.titulaire + " ," + c2.numero);
  }
}
```

Trace d'exécution :

```
%> java testBis
%valeur de c1:3 , 6 ,2004
%valeur de d2:3 ,6 , 2004
```

1.2.3 Invoquer les méthodes sur les objets.

Une classe contient des variables d'instances et des méthodes. Chaque objet instance de cette classe aura son propre état, c'est à dire ses propres valeurs pour les variables d'instances. On pourra aussi invoquer sur lui chaque méthode non statique de la classe. Comment invoque-t-on une méthode sur un objet ?

Pour invoquer la méthode `afficher()` sur un objet `c1` de la classe `Compte` il faut écrire : `c1.afficher();` .

Comme l'illustre l'exemple suivant :

```
public class testAfficher  {
   public static void main (String [] arguments){
    Compte c1 = new Compte();
    Compte c2 = new Compte();
c1.solde =100;
c1.numero=218;
c1.titulaire=''Dupont'';
     c1.afficher();
     c2.afficher();
   }
}
```

L'expression `c1.afficher();` invoque la méthode `afficher()` sur l'objet `c1`. Cela a pour effet d'afficher à l'écran le solde de `c1` c'est à dire 200. L'expression `c2.afficherDate();` invoque la méthode `afficher()` sur l'objet `c2`. Cela affiche le solde de `c2` c'est à dire 0.

```
> java testafficher
%0 , 0 , 0
%4 , 12 , 2000
```

Ainsi, les méthodes d'objets (ou méthodes non statiques) s'utilisent par invocation sur les objets de la classe dans lesquelles elles sont définies. l'objet sur lequel on l'invoque ne fait pas partie de la liste des arguments de la méthode. Nous l'appellerons l'objet courant.

1.3 Retour sur les méthodes non statiques

Dans une classe définissant un type d'objet, on définit l'état caractéristiques des objets de la classe (les variables d'instances) et les méthodes capables d'agir sur l'état des objets (méthodes non statiques). Pour utiliser ces methodes sur un objet `x` donné, on ne met pas x dans la liste des arguments de la méthode. On utilisera la classe en déclarant des objets instances de cette classe. Sur chacun de ces objets, la notation pointée permettra d'accéder à l'état de l'objet (la valeur de ses variables d'instances) ou de lui appliquer une des méthodes de la classe dont il est une instance.
Par exemple, si `c1` est un objet instance de la classe `Compte` `c1.titulaire` permet d'accéder au titulaire de ce compte, et `c1.dépot(800)` permet d'invoquer la méthode dépot sur `c1`.

1.3.1 Les arguments des méthodes non statiques

Contrairement aux sous programmes statique que nous écrivions jusqu'alors, on voit que les méthodes non statiques on un **argument d'entrée implicite**, qui ne figure pas parmi les arguments de la méthode : l'objet sur lequel elle sera appliqué, que nous avons déjà appelé *l'objet courant*.

Par exemple, la méthode `afficher` de la classe `compte` n'a aucun argument : elle n'a besoin d'aucune information supplémentaire à l'objet courant.
Une méthode d'objet peut cependant avoir des arguments. C'est le cas par exemple de `depot` : on dépose sur un compte donné (objet courant) un certain `montant`. Ce montant est une information supplémentaire à l'objet sur lequel s'invoquera la méthode et nécessaire à la réalisation d'un dépot.

Les seuls arguments d'une méthode non statique sont les informations nécessaires à la manipulation de l'objet courant (celui sur lequel on invoquera la méthode), jamais l'objet courant lui même.

1.3.2 Le corps des méthodes non statiques

Les méthodes non statiques peuvent consulter ou modifier l'état de l'objet courant. Celui ci n'est pas nommé dans la liste des arguments. Il faut donc un moyen de désigner l'objet courant dans le corps de la méthode.
C'est le role du mot clé `this`. Il fait référence à l'objet sur lequel on invoquera la méthode. A part cela, le corps des méthodes non statiques est du code Java usuel.

Par exemple dans la définition de `afficher` :

```
void afficher(){
  Terminal.ecrireString(''solde''+ this.solde);
}
```

`this.solde` désigne la valeur de la variable d'instance `solde` de l'objet sur lequel sera invoqué la méthode.

Lors de l'exécution de `c1.afficher()`, `this` désignera `c1`, alors que lors de l'exécution de `c2.afficher()` , `this` désignera `c2`.

En fait, lorsque cela n'est pas ambigu, on peut omettre `this` et écrire simplement le nom de la méthode sans préciser sur quel objet elle est appelée. Pour la méthode `afficher` cela donne :

```
void afficher(){
  Terminal.ecrireString(''solde''+ solde);
}
```

1.3.3 Invocation de méthodes avec arguments

Lorsqu'une méthode d'objet a des arguments, on l'invoque sur un objet en lui passant des valeurs pour chacun des arguments.

Voici un exemple d'invoquation de `depot` :

```
public class testDepot  {
   public static void main (String [] arguments){
    Compte c1 = new Compte();
    c1.solde =100;
    c1.numero=218;
    c1.titulaire=''Dupont'';
    c1.afficher();
    c1.depot(800);
    c1.afficher();
   }
}
```

1.3.4 Lorsque les méthodes modifient l'état de l'objet

La méthode `depot` modifie l'état de l'objet courant. L'invocation de cette méthode sur un objet modifie donc l'état de cet objet. Dans notre exemple d'utilisation, le premier `c1.afficher()` affiche 100, alors que le second `c1.afficher()` affiche 900. Entre ces deux actions, l'exécution de `c1.depot(800)` a modifié l'état de `c1`.

1.3.5 Lorsque les méthodes retournent un résultat

Les méthodes non statiques peuvent evidemment retourner des valeurs. On pourrait par exemple modifier `depot` pour qu'en plus de modifier l'état de l'objet, elle retourne le nouveau solde en résultat :

```
 class Compte {
    int solde;
    String titulaire;
    int numero;

    void afficher(){
      Terminal.ecrireString(``solde''+ this.solde);
    }
    int depot(int montant){
      this.solde = this.solde+ montant;
      return this.solde;
    }
 }
```

Maintenant, dépot fait 2 choses : tout d'abord elle modifie l'état de l'objet courant, puis elle retourne l'entier correspondant au nouveau solde. On peut donc utiliser son invocation sur un objet comme n'importe quelle expression de type `int`.

Par exemple

```
public class testDepot  {
   public static void main (String [] arguments){
```

```
    Compte c1 = new Compte();
    c1.solde =100;
    c1.numero=218;
    c1.titulaire=''Dupont'';
    Terminal.ecrireIntln(c1.depot(800));
   }
}
```

1.4 Les types des variables d'instances peuvent être des classes

Lorsqu'on définit un classe, on peut choisir comme type pour les variables d'instances n'importe quel type existant.

On peut par exemple définir le classe `Personne` par la donnée des deux variables d'instances, l'une contenant la date de naissance de la personne et l'autre son nom. La date de naissance n'est pas un type prédéfini. Il faut donc aussi définir une classe `Date`

```
public class Date {
    int jour;
    int mois;
    int annee;
    public  void afficherDate(){
        Terminal.ecrireStringln(
          this.jour + "␣,␣" + this.mois + "␣,␣" + this.annee);
    }
    // On ne montre pas les autres methodes ...
}
```

```
 public class Personne{
    Date naissance;
    String nom;
// on ne montre pas les methodes...
}
```

Lorsqu'on déclare et initialise une variable `p2` de type `Personne`, en faisant :
`Personne p2 = new Personne();` l'opérateur `new` donne des valeurs par défaut à `nom` et `naissance`. Mais la valeur par défaut pour les objets est `null`, ce qui signifie que la variable `p2.naissance` n'est pas initialisée.
Dès lors, si vous faites `p2.naissance.jour = 18;` l'éxecution provoquera la levée d'une exception :

```
> java DateTest2
Exception in thread "main" java.lang.NullPointerException
        at DateTest2.main(DateTest2.java:99)
```

Il faut donc aussi initialiser p2.naissance :
`p2.naissance= new Date();` avant d'accéder aux champs jour mois et annee.

On peut alors descendre progressivement à partir d'une valeur de type `Personne` vers les valeurs des champs définissant le champ `Date`. Si par exemple `p1` est de type `Personne` alors :

1. p1.nom est de type `String` : c'est son nom.
2. p1.naissance est de type `Date` : c'est sa date de naissance.
3. p1.naissance.jour est de type `int` : c'est son jour de naissance.
4. p1.naissance.mois est de type `int` : c'est son mois de naissance.
5. p1.naissance.annee est de type `int` : c'est son annee de naissance.

Le champ `naissance` d'une personne n'est manipulable que via l'opérateur **.**, l'affectation et les méthodes définies pour les dates.

```
public class PersonneTest  {
  public static void main (String [] arguments){
    Personne p2 = new Personne();
    p2.nom=''toto'';
    p2.naissance= new Date();
    p2.naissance.jour = 18;
    Terminal.ecrireIntln(p2.naissance.jour);
    Terminal.ecrireString ( p2.nom + "␣␣date␣naissance:");
    p2.naissance.AfficherDate();
  }
}
```

1.5 Les classes d'objets peuvent aussi avoir des méthodes statiques

Lorsqu'on définit une classe caractérisant un ensemble d'objets, on définit des variables d'instance et des méthodes non statiques. Ces méthodes définissent un comportement de l'objet courant.

Mais les classe peuvent aussi avoir des méthodes statiques, identiques à celles que nous avons définies dans les chapitres précédents.
Imaginons par exemple que nous voulions définir dans la classe `Date` les méthodes `bissextile` et `longueurMois`. Rien ne nous empêche de les définir comme des méthodes statiques.

```
public class Date  {
    int jour;
    int mois;
    int annee;
    public  void afficherDate(){
        Terminal.ecrireStringln(
          this.jour + "␣,␣" + this.mois + "␣,␣" + this.annee);
    }
    public static boolean bissextile(int a ){
      return ((a%4==0) && (!(a%100 ==0) || a%400==0));}

    public static int longueur(int m , int a ){
      if (m == 1 || m== 3 || m==5 ||m== 7  || m==8|| m==10 || m== 12)
        {return 31;}
      else if (m==2) {if ( bissextile(a)){return 29;} else {return 28;}}
      else {return 30;}
    }
}
```

Déclarer une méthode statique signifie que cette méthode n'agit pas, ni ne connaît en aucune manière, l'objet courant. Autrement dit, c'est une méthode qui n'agit que sur ses arguments et sur sa valeur de retour. Techniquement, cela signifie qu'une telle méthode ne peut jamais, dans son corps, faire référence à l'objet courant, ni aux valeurs de ses variables d'instance.

De la même façon, on ne les invoque pas avec des noms d'objets, mais avec des noms de classe. Pour `longueur`, comme elle est dans la classe `Date`, il faudra écrire :

```
Date.longueur(12,2000);
```

Dans la classe `Date`, on pourra utiliser ces méthodes dans la définition d'autres méthodes, sans faire référence à la classe `Date`.

Dans la pratique, on écrit peu de méthodes statiques et beaucoup de méthodes non statiques.

Méthodes statiques et non statiques :

– Les méthodes non statiques définissent un comportement de l'objet courant auquel elles font, dans leur corps, référence au moyen de `this` :

```
        public  void afficherDate(){
         Terminal.ecrireStringln(
   this.jour + " , " + this.mois + " , " + this.annee);
        }
```

On les appelle avec des noms d'objets : `d2.afficherDate();`

– Les méthodes statiques ne connaissent pas l'objet courant. Elles ne peuvent faire référence à `this`, ni aux variables d'instance.

```
    public static boolean bissextile(int a ){
      return ((a%4==0) && (!(a%100 ==0) || a%400==0));}
```

On les appelle avec un nom de classe : `Date.longueur(12,2000);`

Maintenant que nous avons `bissextile` et `longueur`, nous pouvons définir dans `Date` la méthode `lendemain`. C'est une méthode d'objet, sans argument qui a pour effet de modifier l'état de l'objet courant en le faisant passer à la date du lendemain :

```
public class Date  {
    int jour;
    int mois;
    int annee;

    public static boolean bissextile(int a ){
      return ((a%4==0) && (!(a%100 ==0) || a%400==0));}

    public static int longueur(int m , int a ){
      if (m == 1 || m== 3 || m==5 ||m== 7  || m==8|| m==10 || m== 12)
        {return 31;}
      else if (m==2) {if ( bissextile(a)){return 29;} else {return 28;}}
      else {return 30;}
    }

    public  void lendemain(){
      if (this.jour < longueur(this.mois,this.annee)){
        this.jour = this.jour +1;
      }
      else if (this.mois == 12) {
```

```
            this.jour = 1; this.annee=this.annee +1 ;this.mois =1;
        }
        else {
            this.jour = 1 ;this.mois =this.mois+1;
        }
    }
}
```

```
public class Datetest3  {
  public static void main (String [] arguments){
    Date d2  = new Date();
    d2.annee=2000;
    d2.jour =28;
    d2.mois =2;
    d2.afficherDate();
    d2.lendemain();
    d2.afficherDate();
  }
}
```

Et voici ce que cela donne lors de l'exécution :

```
> java Datetest3
28 , 2 , 2000
29 , 2 , 2000
```

1.6 Les constructeurs.

Revenons un instant sur la création d'objets.
Que se passe-t-il lorsque nous écrivons : `Date d1= new Date()` ?
L'opérateur `new` réserve l'espace mémoire nécessaire pour l'objet `d1` et initialise les données avec des valeurs par défaut. Une variable d'instance de type `int` recevra 0, une de type `boolean` recevra `false`, une de type `char` recevra `'\0'` et les variables d'instances d'un type objet recevra `null`. Nous expliquerons ce qu'est `null` dans le prochain chapitre. l'opérateur `new` réalise cela en s'aidant d'un *constructeur* de la class `Date`. En effet, lorsqu'on écrit `Date()`, on appelle une sorte de méthode qui porte le même nom que la classe. Dans notre exemple, on voit que le constructeur `Date` qui est appelé n'a pas d'arguments.

Un constructeur ressemble à une méthode qui porterai le même nom que la classe, mais ce n'est pas une méthode : la seule façon de l'invoquer consiste à employer le mot clé `new` suivi de son nom, c'est à dire du nom de la classe[1]. Ceci signifie qu'il s'exécute avant toute autre action sur l'objet, lors de la création de l'objet.

1.6.1 Le constructeur par défaut

La classe `Date` ne contient pas explicitement de définition de constructeur. Et pourtant, nous pouvons, lors de la création d'un objet, y faire référence après l'opérateur `new`. Cela signifie que Java fournit pour chaque classe définie, un constructeur par défaut. Ce constructeur par défaut :

– a le même nom que la classe et

[1] En fait, on peut aussi l'invoquer dans des constructeurs d'autres classes (cf chapitre sur l'héritage)

– n'a pas d'argument.

Le constructeur par défaut ne fait pratiquement rien. Voilà a quoi il pourrait ressembler :

```
public Date(){

}
```

1.6.2 Définir ses propres constructeurs

Le point intéressant avec les constructeurs est que nous pouvons les définir nous mêmes. Nous avons ainsi un moyen d'intervenir au milieu de `new`, d'intervenir lors de la création des objets, donc avant toute autre action sur l'objet.

Autrement dit, en écrivons nos propres constructeurs, nous avons le moyen, en tant que concepteur d'une classe, d'intervenir pour préparer l'objet à être utilisé, avant toute autre personne utilisatrice des objets.

Sans définir nos propres constructeurs de Date, les objets de types `Date` commencent mal leur vie : il naissent avec 0,0,0 qui ne représente pas une date correcte. En effet, 0 ne correspond pas à un jour, ni à un mois. C'est pourquoi une bonne conception de la classe `Date` comportera des définitions de constructeurs.

attention : Dès que nous définissons un constructeur pour une classe, le constructeur par défaut n'existe plus.

Un constructeur se définit comme une méthode sauf que :

1. Le nom d'un constructeur est toujours celui de la classe.
2. Un constructeur n'a jamais de type de retour.

Dans une classe, on peut définir autant de constructeurs que l'on veut, du moment qu'ils se différencient par leur nombre (ou le type) d'arguments . Autrement dit, on peut surcharger les constructeurs.

Nous pourrions par exemple, pour notre classe `Date`, définir un constructeur sans arguments qui initialise les dates à 1,1,1 (qui est une date correcte) :

```
public class Date {
  int jour;
  int mois;
  int annee;

  public Date(){
    this.jour =1;
    this.mois=1;
    this.annee =1;
    }
  // ....
}
```

Maintenant, toute invocation de `new Date()` exécutera ce constructeur.

Il peut aussi être utile de définir un constructeur qui initialise une date avec des valeurs données. Pour cela, il suffit d'écrire un constructeur avec 3 arguments qui seront les valeurs respectives des champs. Si les valeurs d'entrée ne représentent pas une date correcte, nous levons une erreur :

```
  public class Date {
    // --- Les variables d'instances ---
    int jour;
    int mois;
```

```
int annee;

// --- Les constructeurs ---
public Date(){
  this.jour =1;
  this.mois=1;
  this.annee =1;
}

public  Date (int j, int m, int a){
  if (m >0 && m<13 && j <=longueur(m,a)){
    this.jour=j;
    this.mois = m;
    this.annee = a;
  }
  else {
    throw new ErreurDate();
  }
}
// --- Les methodes ---
public  void afficherDate(){
  Terminal.ecrireStringln
  (this.jour + " , " + this.mois + " , " + this.annee);
}

public int getAnnee(){
  return this.annee;
}

public void setAnnee(int aa){
  this.annee=aa;
}

public  void lendemain(){
  if (this.jour < longueur(this.mois,this.annee)){
    this.jour = this.jour +1;
  }
  else if (this.mois == 12) {
    this.jour = 1; this.annee=this.annee +1 ;this.mois =1;
  }
  else {
    this.jour = 1 ;this.mois =this.mois+1;
  }
}

public  static boolean bissextile(int annee ){
  return ((annee%4==0) && (!(annee%100 ==0) || annee%400==0));
}

public static int longueur(int m , int a ){
  if (m == 1 || m== 3 || m==5 ||m== 7  || m==8|| m==10 || m== 12){
```

```
        return 31;
      }
      else if (m==2) {if ( bissextile(a)){return 29;} else {return 28;}}
      else {return 30;}
    }
  }
  class ErreurDate extends Error{}
```

```
  public class datetest4  {
    public static void main (String [] arguments){
      Date  d1  = new Date ();
      Date  d2  = new Date (2,12,2000);
      d1.afficherDate();
      d2.afficherDate();
      d2.lendemain();
      d2.afficherDate();
```

l'éxécution de `datetest4` donne :

```
> java datetest4
  1 , 1 , 1
  2 , 12 , 2000
  3 , 12 , 2000
```

Répetons encore une fois que lorsqu'on définit ses propres constructeurs, le constructeur par défaut n'existe plus. En conséquence, si vous écrivez des constructeurs qui ont des arguments, et que vous voulez un constructeur sans argument, vous devez l'ecrire vous même.

- Un constructeur :
  - est un code qui s'éxécute au moment de la création de l'objet (avec `new`),
  - porte le même nom que la classe,
  - ne peut pas avoir de type de retour,
  - peut avoir ou ne pas avoir d'arguments,
  - sert à initialiser l'état de chaque objet crée.
- Si on ne définit aucun constructeur, il en existe un par défaut, qui n'a pas d'argument.
- Le constructeur par défaut n'existe plus dès que l'on définit un constructeur même si on ne définit pas de constructeur sans argument.
- 2 constructeurs doivent avoir des listes d'arguments différentes.

1.7 Résumé

- Une classe est un patron d'objet qui définit :
  1. Les données caractéristiques des objets : les variables d'instances.
  2. Les constructeurs permettant d'initialiser les objets lors de leur création. (facultatif)
  3. le comportement des objets : les méthodes que l'on pourra invoquer sur les objets.
- Un objet est une instance d'une classe.

1. On les créé en faisant `new` suivi d'un appel de constructeur : `Date d1 = new Date();` `Date d2 = new Date(12, 6,2003);`
2. Ils ont leur propres valeurs pour chacune des variables d'instances.
3. On peut leur appliquer les méthodes de la classe dont ils sont l'instance : `d1.afficherDate();` .

– Les méthodes non statiques définissent un comportement de l'objet courant auquel elles font, dans leur corps, référence au moyen de `this` :

```
public  void afficherDate(){
 Terminal.ecrireStringln(
   this.jour + " , " + this.mois + " , " + this.annee);
}
```

On les appelle avec des noms d'objets : `d2.afficherDate();`

– Les méthodes statiques ne connaissent pas l'objet courant. Elles ne peuvent faire référence à `this`, ni aux variables d'instances.

```
public static boolean bissextile(int a ){
  return ((a%4==0) && (!(a%100 ==0) || a%400==0));}
```

On les appelle avec un nom de classe : `Date.longueur(12,2000);`

– Un constructeur :
  – est un code qui s'éxécute au moment de la création de l'objet (avec `new`)
  – porte le même nom que la classe,
  – ne peut pas avoir de type de retour,
  – peut avoir ou ne pas avoir d'argument,
  – sert à initialiser l'état de chaque objet crée.
– Si on ne définit aucun constructeur, il en existe un par défaut, qui n'a pas d'arguments.
– Le constructeur par défaut n'existe plus dès que l'on définit un constructeur.
– 2 constructeurs doivent avoir des listes d'arguments différentes (par le nombre et/ou par le type des arguments).

Chapitre 2

Types primitifs et références

2.1 Introduction

En Java, pour déclarer une variable, il faut donner son nom, précédé du *type* qu'on souhaite lui attribuer. Ces types peuvent ˆêtre des types primitifs (`int n;`), ou bien correspondre à des classes (`Date d1;`). Les variables `n` et `d1` en Java, ne sont pas de la même sorte. La première contient une valeur élémentaire, tandis que la seconde contient une *référence* à un objet de type `Date`. Voilà ce que nous allons étudier en détail dans ce chapitre, en commençant par les variables de type primitif et en expliquant ensuite ce qu'est une variable référence.

2.2 Variables de type primitif

Les types primitifs sont : `boolean, char, byte, short, int, long, float, double`. Pour représenter les valeurs de ces types, il faut un nombre fixe de bits : par exemple 8 bits pour les `byte` et 32 pour les `int`.

Lorsqu'on déclare une variable, on donne son type : Une variable déclarée de type primitif contient une valeur de type primitif. Ainsi, lors des déclarations suivantes :

```
int n;
char c = 'a';
n=5;
double d = 2.0;
```

Le nom `n` désigne un espace mémoire de 32 bits, qui contient l'entier 5 (après exécution de la ligne 3) ; `c` désigne un espace mémoire de 16 bits, qui contient `'a'` et `d` désigne un espace mémoire de 64 bits, qui contient `2.0`. Dans la suite du programme, on peut accéder au contenu de ces espaces mémoire par leur nom : `n, d` ou `c` et modifier ce contenu grâce à l'affectation : `n = 10`.

En résumé, en déclarant une variable d'un type primitif, on donne un nom à un espace mémoire. La taille de cet espace dépend du type de la variable. Lorsque on affecte une valeur à la variable, on met cette valeur dans l'espace mémoire correspondant. C'est en ce sens que nous disons que la variable *contient* une valeur de son type.

2.3 Variables références et objets

2.3.1 Introduction

Que se passe t il lorsque nous créons une variable dont le type est une classe ?

```
Date d = new Date();
```

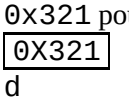
(On suppose que `Date` est la classe définie dans le cours sur les classes et les objets). Une nouvelle variable, `d` est crée. Par ailleurs, `new Date()` crée une instance de la classe `Date`, un objet. Le point clé est le suivant : contrairement aux variables primitives du paragraphe précédent, `d` ne *contient pas un objet* , mais une *référence à un objet*. Si une variable primitive contient des bits qui représentent la valeur qu'elle contient, une variable de type complexe contient des bits qui représentent *une façon d'accéder à l'objet*. Elle ne contient pas l'objet lui même, mais quelque chose qui permet de retrouver l'adresse où se situe l'objet en mémoire. C'est pourquoi nous disons que ces variables contiennent l'adresse de l'objet ou encore un pointeur sur l'objet. Nous appellerons ces variables des variables *références*.

Nous n'avons pas besoin de savoir comment la JVM implémente les références aux objets. C'est sans importance parce que nous ne pouvons les utiliser que pour accéder à l'objet. Nous savons en revanche que pour une JVM donnée, toutes les références sont de même taille quelque soit l'objet qui est référencé.

2.3.2 Détail de la création d'un objet

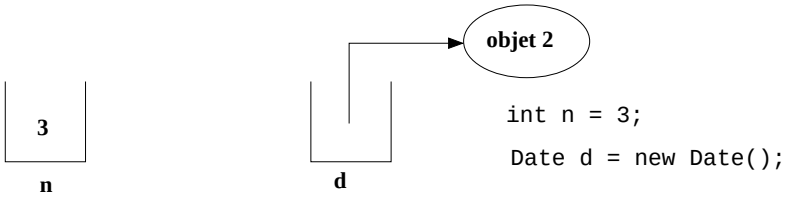
Nous pouvons maintenant détailler le processus qui est à l'oeuvre lors de l'exécution d'une déclaration de variable référence. Prenons l'exemple de `Date d = new Date()`. Ce processus se décompose en trois étapes :

1. `Date d` `= new Date()` : déclaration d'une variable référence.
   De l'espace mémoire est alloué pour une variable référence de nom `d`. Cet espace mémoire est de taille fixe, suffisant pour contenir une adresse. La taille de l'espace mémoire réservé pour `d` ne dépend en rien de la taille qu'il faut pour stocker un objet `Date`, mais dépend de la taille qu'il faut pour stocker une adresse mémoire. A la suite de `Personne p = new Personne();` par exemple, l'espace alloué à `p` serait strictement de même taille que celui alloué à `d`. A cette étape, `p` et `d` ne contiennent encore aucune adresse.
   
   `d`
2. `Date d =` `new Date()` : création d'un objet `Date`.
   Appelons cet objet *objet1*. C'est l'opérateur `new` qui s'occupe de réserver de l'espace mémoire pour stocker un objet. Il détermine combien d'espace est nécessaire pour stocker cet objet et détermine l'adresse où sera stocké cet objet. Imaginons, pour fixer les esprits, que cette adresse soit `0x321` :
   

3. `Date d` `=` `new Date()` : Liaison de l'objet à la référence. L'adresse ou réside l'objet en mémoire, `0x321` pour notre exemple, est donnée comme valeur à la variable référence `d` :
   

Comme nous ne connaissons pas réellement l'adresse de l'objet, nous préférons la représenter graphiquement par une flèche vers l'espace mémoire contenant l'objet. Nous représenterons les 2 sortes de variables

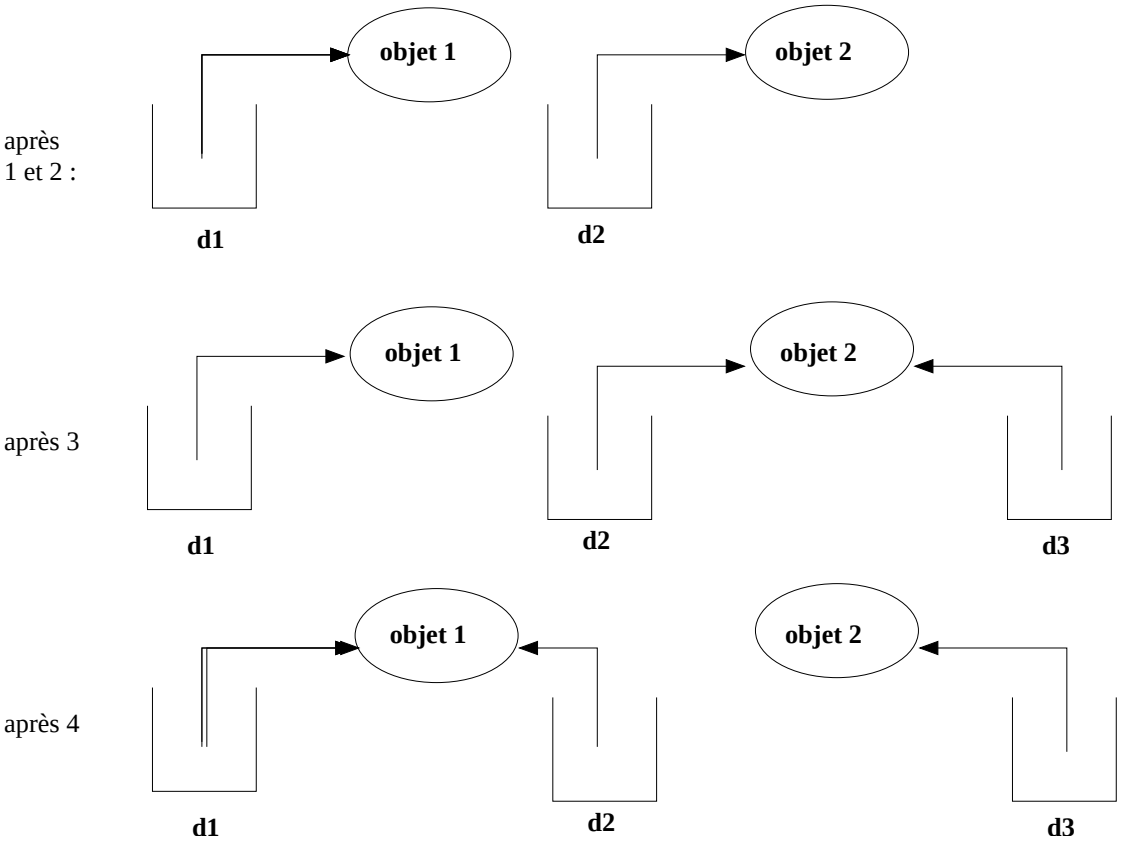
de la façon suivante :



2.3.3 Manipuler les références

Une variable référence contient une adresse vers un objet d'un certain type. Comme toute autre variable, on peut lui donner une valeur par affectation. Comme Java est typé, on ne peut lui donner qu'une valeur de même type, c'est à dire une adresse vers un objet de même type. Mais nous ne connaissons pas explicitement les adresses, donc les deux seules façons de donner une valeur à une variable référence sont d'utiliser `new`, comme nous venons de le voir, ou de lui affecter la valeur d'une autre variable référence de même type.

```
Date d1 = new Date();
Date d2 = new Date();
Date d3 = d2;
d2=d1;
```



A la suite des 2 premières instructions, nous avons 2 références et 2 objets. En ligne 3, nous avons 3 références mais seulement 2 objets : `d3`, prend comme valeur celle de `d2`, donc l'adresse de l'objet2. `d2` et `d3` référencent le même objet. Ce sont deux façons différentes d'accéder au même objet. En ligne 4, nous avons toujours 3 références et 2 objets : `d2` prends maintenant l'adresse de l'objet1. `d1` et `d2` référencent le même objet.

2.3.4 Manipuler les objets référencés

Une variable référence contient une adresse, un moyen d'accéder à un objet, pas l'objet lui même. Mais comment accéder via la variable à cet objet et comment le modifier ? Nous le savons déjà : par la notation pointée.
Si `d1` est une variable qui référence une date, sa valeur est une adresse, mais `d1.` désigne l'objet date qu'elle référence. Nous pouvons ainsi lui appliquer toutes les méthodes des dates et accéder au variables d'instances `jour, mois, annee` :

```
public class Chap12a {
    public static void main(String [] args){
      Date d1 = new Date();
      d1.afficherDate();
      d1.lendemain();
      d1.jour = d1.jour +1;
      Terminal.ecrireStringln("Annee " + d1.annee);
    }
}
```

Son Exécution produit :

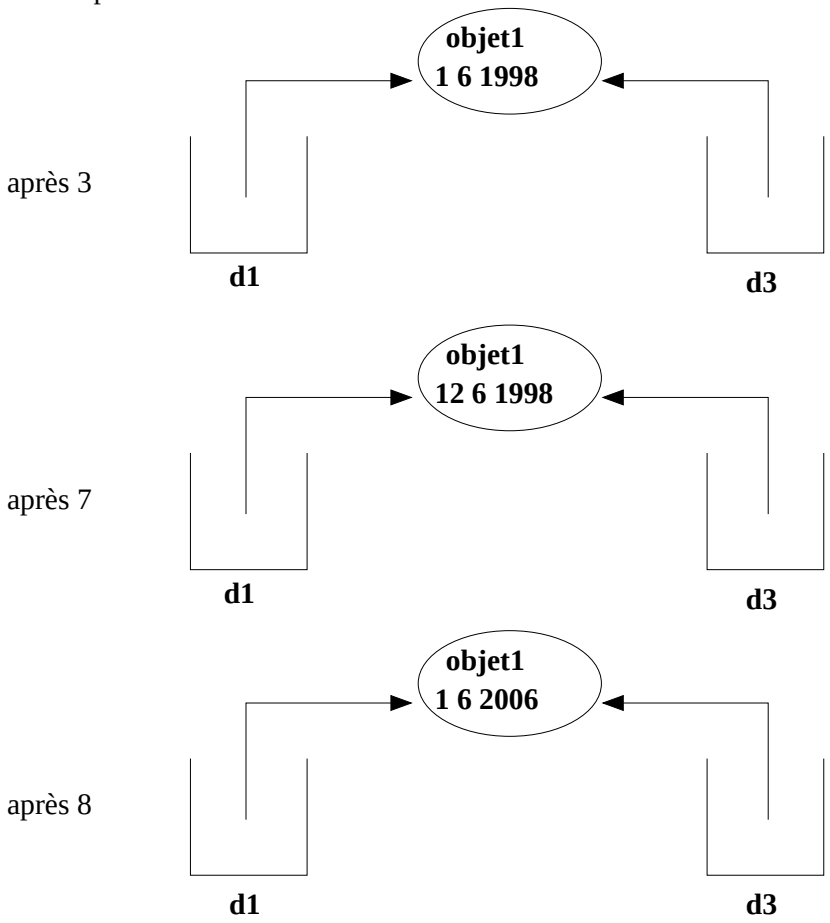
```
simonot@saturne:> java Chap12a
1 , 1 , 1
1
```

2.3.5 Attention au partage des données

Nous avons vu que plusieurs variables peuvent référencer le même objet. Lorsque c'est le cas, le même objet est accessible (et donc modifiable) par l'intermédiaire de chacune des variables qui le référencent :

```
public class Chap12b {
    public static void main(String [] args){
      Date d1 = new Date(1,6,1998);
      Date d3 = d1;
      d1.afficherDate();
      d3.afficherDate();
      d3.jour = 12;
      d1.annee= 2006;
      d1.afficherDate();
      d3.afficherDate();
    }
  }
```

Dans ce programme, `d1` et `d3` référencent le même objet, que l'on modifie par l'intermédiaire de l'une et de l'autre. A chaque instant, la date pointée par `d1` et `d2` et la même. C'est le même objet qu'on modifie que ce soit par l'intermédiaire de `d1` ou de `d2`.



L' exécution de ce programme produit donc évidemment :

```
simonot@saturne:> java Chap12b
1 , 6 , 1998
1 , 6 , 1998
12 , 6 , 2006
12 , 6 , 2006
```

Le plus souvent, on ne veut pas que plusieurs variables référencent le même objet, mais on veut qu'elles aient à un moment donné, les mêmes valeurs. C'est souvent le cas lors de l'initialisation d'une variable locale par exemple. Dans ce cas, il faut procéder par recopie de la valeur des variables d'instances, comme nous l'avons fait pour `d2` dans l'exemple qui suit.

```
public class Chap12c {
  public static void main(String [] args){
    Date d1 = new Date(1,6,1998);
    Date d2 = new Date();
    Date d3 = d1;    // d3 et d1 referencent le meme objet

    d2.jour = d1.jour;
```

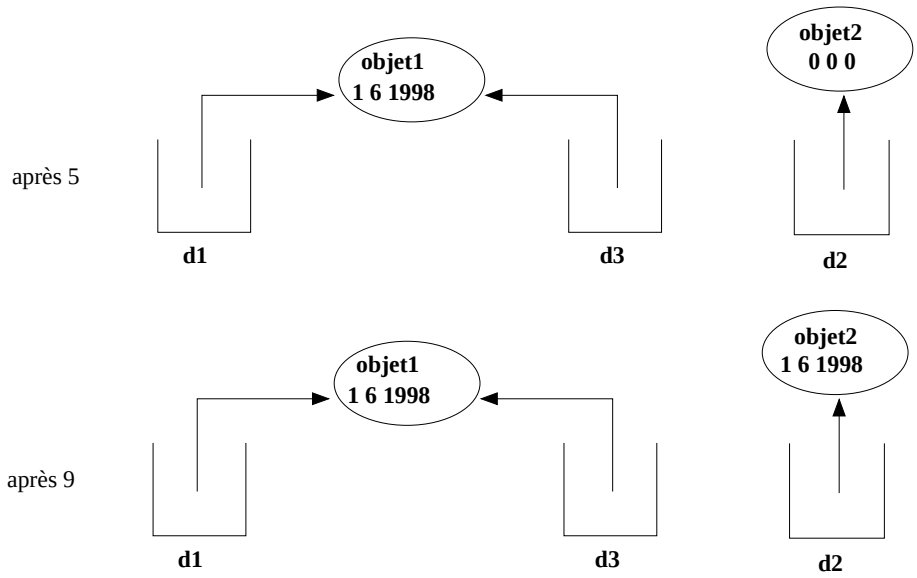
```
      d2.annee = d1.annee;
      d2.mois = d1.mois;
// d2 recopie dans sa date, les valeurs de la date referencee par d1

      d1.afficherDate();
      d2.afficherDate();
      d3.afficherDate();

      d2.jour = 18;
      d1.afficherDate();
      d2.afficherDate();

      d3.jour = 12;
      d1.annee= 2006 ;

      d1.afficherDate();
      d2.afficherDate();
      d3.afficherDate();
    }
  }
```



L' exécution du programme produit donc :

```
simonot@saturne:> java Chap12c
1 , 6 , 1998
1 , 6 , 1998
1 , 6 , 1998
1 , 6 , 1998
18 , 6 , 1998
12 , 6 , 2006
18 , 6 , 1998
```

12 , 6 , 2006

2.4 Retour sur l'égalité entre références

Nous pouvons maintenant comprendre le comportement de l'opérateur d'égalité ==.

L'opérateur == compare les bits contenus dans les variables. Lorsque les variables sont de types primitifs (comme n et m), cela teste si les entiers qu'elles contiennent sont les mêmes. Mais, comme les bits contenus dans les variables références représentent des adresses, cela teste si les adresses sont égales, c'est à dire si les variables référencent le même objet.

```
public class Chap12d  {
  public static void main (String [] arguments)
  {
    int n =3;
    int m = 2+1;
    if (m==n){
      Terminal.ecrireStringln("n==m");
    }
    else {
      Terminal.ecrireString("n!=m");
    }
    Date d1 = new Date(1,1,2000);
    Date d2 = d1;
    Date d3 = new Date(1,1,2000);
    if (d1==d2){
      Terminal.ecrireStringln("d1==d2");
    }
    else {
      Terminal.ecrireStringln("d1!=d2");
    }
    if (d1==d3){
      Terminal.ecrireStringln("d1==d3");
    }
    else {
      Terminal.ecrireStringln("d1!=d3");
    }
  }
}
```

L'exécution de ce programme produit :

```
simonot@saturne:> java Chap12d
n==m
d1==d2
d1!=d3
```

`n == m` vaut `true` car `n` et `m` contiennent la même valeur : `3`.
`d1 == d2` vaut `true` car `d1` et `d2` contiennent la même valeur : l'adresse d'un même objet.
`d1 == d3` vaut `false` car `d1` et `d3` contiennent 2 adresses différentes.

2.5 Retour sur le passage des paramètres

En Java, le passage des paramètres, lors de l'appel d'une méthode, se fait par valeur. Nous avons déjà étudié cela lors du chapitre sur les sous programmes. Ceci signifie que ce sont les valeurs des arguments d'appel qui sont transmises lors de l'exécution d'un appel de méthode. Ce mode de passage des paramètres uniforme induit des comportements différents suivant que les arguments d'appel sont des variables primitives ou des références. C'est ce que nous allons détailler maintenant.

2.5.1 Passage par valeur sur des arguments de type primitif

Prenons un exemple simple et détaillons les étapes de l'exécution d'un appel de méthode.

```
class Prim1 {
  static int m1(int a){
    return a*a;
  }
  public static void main(String [] args){
    int b = 3;
    Terminal.ecrireIntln(m1(b));
  }
}
```

Exécution de `m1(b)` :

1. La valeur de `b` est calculée : comme c'est une variable primitive, sa valeur est l'entier qu'elle contient , c'est à dire 3.
2. De l'espace mémoire est alloué pour l'argument `a` de la méthode. `a` est initialisée avec la valeur de `b` : 3
3. Le corps de `m1`, ici limité à `return a*a` est exécuté : On calcule `a*a` ce qui donne 9, la valeur de l'appel `m1(b)` est donc 9. La variable a n'existe plus.

Ainsi, on voit que `m1(b)` est strictement équivalent à `m1(3)`. On comprends d'autres part que le mode de passage des paramètres par valeur interdit de modifier la valeur des arguments d'appel, lorsqu'ils sont de type primitifs. Prenons un autre exemple :

```
class Prim2 {
  static void m2(int a){
    a= a*a;
  }
  public static void main(String [] args){
    int b = 3;
    (m2(b));
    Terminal.ecrireIntln(b);
  }
}
```

Cet exemple affiche 3 et non 9, ce qui n'est pas surprenant : le temps de l'exécution de l'appel `m2(b)`, une variable locale `a`, initialisée avec la valeur de `b` 3, est crée. Le corps de `m2` modifie `a`. en lui donnant 9. A la fin de l'exécution de cet appel, `a` n'existe plus. L'initialisation de `a` avec la valeur de `b` est l'unique lien qui existe entre `a` et `b`. `b` n' est donc pas modifié.

2.5.2 Passage par valeur sur des variables références

En Java, le passage des paramètres se fait toujours par valeur. Mais, lorsque les arguments des fonctions sont des références, ce mode permet la modification des arguments. On passe par valeur, donc on transmet la valeur d'une variable référence, c'est à dire l'adresse d'un objet, à une autre variable référence : elles partagent donc le même objet. Afin de détailler cela, adaptons notre exemple :

```
class Refe {
  static void m2(Date a){
    a.jour=4;
  }
  public static void main(String [] args){
    Date b = new Date(1,1,2000);
    m2(b);
    b.afficherDate();
  }
}
```

Exécution de `m2(b)`

1. La valeur de `b` est calculée : comme c'est une variable référence, sa valeur est l'adresse où réside l'objet `Date` dont les variables d'instances valent `1,1,2000`.
2. De l'espace mémoire est alloué pour l'argument `a` de la méthode et on l'initialise avec la valeur de `b` : l'adresse de l'objet `Date 1,1,2000`. Cet objet est partagé par `a` et `b`.
3. Le corps de `m2`, ici limité à `a.jour=4` est exécuté : la variable d'instance `jour` de l'objet référencé par `a` (et donc aussi celui de `b`) prends la valeur 4. L'exécution du corps de `m2` est terminée, la variable a n'existe plus. Mais l'objet référencé par `b` a été modifié.

l'éxécution de `b.afficherDate()` produit donc `4 , 1 , 2000`

2.6 Retour sur les Tableaux et les strings

2.6.1 Les tableaux

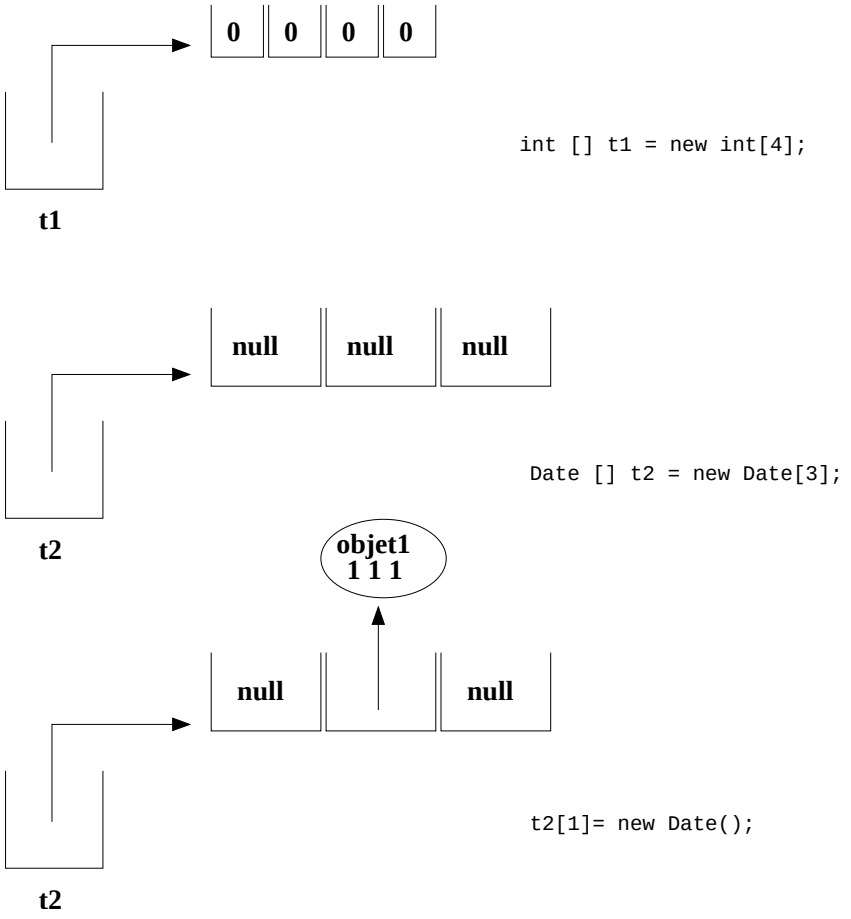
Les tableaux sont des objets. La déclaration d'une variable tableau est donc une référence à un objet. Il faut l'initialiser avec `new`. Tout ce qui a été dit sur les variables références et en particulier sur le partage des données, l'égalité et le passage des paramètres s'applique donc.

Une variable tableau contient l' adresse d'une suite consécutive de variables qui représentent les cases du tableau. Le nombre de ces variables et le type de ces variables sont fixés lors de la déclaration. Ces variables n'ont pas de nom propre, on y accède par le biais de la variable tableau. Prenons un exemple :
`int [] t1 = new int[4]`
déclare une variable `t1` qui référence un objet tableau de 4 cases destinées à contenir des entiers. `t1` référence donc une suite de 4 variables de type `int`. La première de ces variables se nomme `t1[0]`, la seconde `t1[1]`...la quatrième `t1[3]`.

Dans le cas de `t1`, les variables représentant les cases du tableau sont initialisées à 0. Mais dans le cas d'un tableau destiné à contenir des objets, les variables représentant les cases du tableau sont elles mêmes des variables références. Prenons un exemple :
`Date [] t2 = new Date[3]`
déclare une variable `t2` qui référence un objet tableau de 3 cases destinées à contenir des objets Dates.

`t2` référence une suite de 3 variables de type `Date`. Ces variables sont donc des variables références et contiendront donc des adresses. Elles sont initialisées avec `null`. `null` est de même type que les adresses données en valeur aux variables références mais n'est pas une adresse réelle. `null` ne correspond à aucune adresse mémoire existante. Toute tentative d'accéder au contenu de l'"'adresse" `null` provoquera la levée de l'exception `NullPointerException`. Il faudra donc aussi utiliser l'opérateur new sur chacune des cases du tableau, avant d'accéder aux cases du tableau avec l'opérateur `..` On peut le faire par exemple au moyen de l'instruction suivante :

```
for (int i=0; i< t2.length; i++){
    t2[i] = new Date();
}
```



2.6.2 Les Chaines de caractères

Comme les tableaux, les éléments du type String sont des objets et par conséquent les variables de type String des variables références. En général, on les declare et les initialise non pas avec `new`, mais en donnant directement une chaîne de caractère : `String s = ''coucou'';`
`s` est une variable référence, elle contient donc l'adresse de l'emplacement mémoire où est stocké "coucou". Mais un objet de type `String` ne se comporte pas comme un tableau de caractère. En particulier, il n'est pas possible de modifier l'un de ses caractères.

En revanche la classe `String` contient de nombreuses méthodes. Nous en détaillons quelques une dans ce qui suit :

- `public char charAt(int index)` recherche le caractère placé à la position `index`
- `public int indexOf(String str)` localise `str` dans un mot, à partir du début du mot. Retourne -1 si `str` n'est pas dans le mot.
- `public int lastIndexOf(String str)` localise `str` dans un mot, à partir de la fin du mot. Retourne -1 si `str` n'est pas dans le mot.
- `public String substring(int debut, int fin) throws StringIndexOutOfBoundsExc` extrait la sous chaine compris entre les indices `debut` et `fin`.
- `public int length()` renvoie la longueur du mot.

Voici quelques exemples d'utilisation de ces méthodes :

```
class TestString{
  public static void main(String [] args){
    String s = "Il rencontre un chien et un chat";
    int k;
    String t;

    for (int i = 0; i <s.length(); i++){
      Terminal.ecrireStringln("en " + i + " il y a :" + s.charAt(i)  );
    }
    Terminal.ecrireString("la sous chaine entre 7 et 11 est :" );
    Terminal.ecrireStringln(s.substring(7,11) );

    Terminal.ecrireString("entrer un mot :" );
    t = Terminal.lireString();
    k=s.indexOf(t);
    if (k==-1){
      Terminal.ecrireStringln(t + " n'est pas dans " + s );
    }
    else{
      Terminal.ecrireStringln("la premiere position de " + t +
      " est : " + k );
    }
    k=s.lastIndexOf(t);
    if (k==-1){
      Terminal.ecrireStringln(t + " n'est pas dans " + s );
    }
    else{
      Terminal.ecrireStringln("la derniere position de " + t +
      " est : " + k );
    }
  }
}
```

```
simonot@saturne:> java TestString
en 0 il y a :I
en 1 il y a :l
en 2 il y a :
en 3 il y a :r
```

```
en 4 il y a :e
en 5 il y a :n
en 6 il y a :c
en 7 il y a :o
en 8 il y a :n
en 9 il y a :t
en 10 il y a :r
en 11 il y a :e
en 12 il y a :
en 13 il y a :u
en 14 il y a :n
en 15 il y a :
en 16 il y a :c
en 17 il y a :h
en 18 il y a :i
en 19 il y a :e
en 20 il y a :n
en 21 il y a :
en 22 il y a :e
en 23 il y a :t
en 24 il y a :
en 25 il y a :u
en 26 il y a :n
en 27 il y a :
en 28 il y a :c
en 29 il y a :h
en 30 il y a :a
en 31 il y a :t
la sous chaine entre 7 et 11 est :ontr
entrer un mot :un
la premiere position de un est : 13
la derniere position de un est : 25
```

2.7 Addendum au chapitre 1 :Les variables statiques (ou de classe)

Le chapitre 9 a offert une première approche des notions de Classes et d'Objets. Ce paragraphe a pour but d'ajouter une notion caractéristiques des classes : les variables statiques. Nous savons déjà que les classes peuvent contenir

1. des variables d'instances
2. des constructeurs
3. des méthodes statiques ou non statiques.

Les classes peuvent contenir une quatrième sorte d'éléments : des variable statiques.

2.7.1 Déclaration de variables statiques

On les declarent en faisant précéder la déclaration usuelle du mot clé `static`. Par exemple :

```
static int a;
```

A titre d'exemple, ajoutons à la classe `Date` une variable statique `nb` :

```
public class Date {
  // --- Les variables d'instances ---
  int jour;
  int mois;
  int annee;
  // --- La variable statique ---
  static int nb ;
  // --- le reste est inchange ___
    ...
```

2.7.2 Rôle et comportement des variables statiques

Chaque objet à sa propre copie des variables d'instances. Elles représentent l'état personnel de l'objet. En revanche, il y a une seule copie des variables d'instances par classe. Tous les objets instances de la classe partagent la même copie. On peut accéder au contenu des variables d'instances par la classe ou par les objets instances de la classe. Et voici des exemples d'utilisation de la variable statique `nb` ajoutée à la classe `Date` :

```
class public class TestStatic  {
  public static void main (String [] arguments){
    Date d1= new Date(1,1,2000);
    Date d2 = new Date(2,2, 2004);
    Terminal.ecrireIntln(Date.nb); // acces a nb par la classe Date.
    Terminal.ecrireIntln(d1.nb); // acces a nb par une variable d'instance
    Terminal.ecrireIntln(d2.nb); // acces a nb par une autre variable d'instance
    Date.nb = 2;
    Terminal.ecrireStringln (Date.nb + ", " + d1.nb + " , " + d2.nb);

                              d1.nb = 1;
    d1.jour = 23;
    Terminal.ecrireStringln (Date.nb + ", " + d1.nb + " , " + d2.nb);
    Terminal.ecrireStringln (  d1.jour + " , " + d2.jour);
  }
}
```

produit :

```
simonot@jupiterd:> java TestStatic
0
0
0
2, 2 , 2
1, 1 , 1
23 , 2
```

Cet exemple nous montre bien que les variables statiques sont globales à tous les objets instances d'une classe : à chaque instant `d1.nb` et `d2.nb` et `Date.nb` ont la même valeur, contrairement aux variables d'instances dont les valeurs sont personnelles à chaque objet. Ainsi, dans une classe, nous avons deux sortes d'éléments :

- les variables d'instances et les méthodes non statiques qui agissent sur les variables d'instances propres à chaque objet. C'est pourquoi ces méthodes sont aussi appelées méthodes *d'instances*. Chaque objet crée a sa propre copie des variables et méthodes d'instances.
- Les variables et méthodes statiques, dites aussi variables et méthodes *de classe*. Une seule et même copie de ces éléments est partagée par tous les objets de la classe.

Les variables de classe, comme `nb` sont bien sur accessibles dans la classe ou elles sont définies et en particulier dans les méthodes de la classe. Ce sont des variables globales à la classe.

Chapitre 3

Héritage

Ce chapitre du cours traite de concepts relatifs à la programmation objet (hiérarchie de classe, héritage, extension, masquage) et sera illustré par un exemple de représentation de comptes bancaires et d'opérations liées à la manipulation de ceux-ci (retrait, dépôt, consultation).

3.1 Une classe simple pour représenter des comptes bancaires

Décrivons maintenant la classe permettant de représenter un compte bancaire. Il faut, au minimum, connaître le propriétaire (que l'on représentera simplement par son nom), le numéro du compte (que l'on représentera par tableau de caractères) et la somme disponible sur le compte (représentée par un double). Pour les opérations sur un compte, on se limite aux opération de retrait (on décrémente si possible le compte d'une somme donnée), le dépôt (on augmente le compte d'une somme donnée) et la consultation (on retourne la somme disponible sur un compte). Dans le cas où un retrait est impossible on lèvera une exception de type `provisionInsuffisanteErreur` définie séparément. En plus du constructeur par défaut, nous fournirons également un constructeur plus complet permettant de construire un compte en fixant le numéro, le propriétaire et le solde initial. Enfin nous proposons également une méthode permettant d'effectuer un virement d'un compte vers un autre.

Ceci conduit à la construction de la classe suivante :

```
public class CompteBancaire{
    String nomProprietaire ;
    char[] numero ;

    double solde ;

    public CompteBancaire(){
    }

    public CompteBancaire(String proprio, char[] num, double montant){
        this.nomProprietaire = proprio ;
        this.numero = num ;
        this.solde = montant ;
    }

    public double soldeCourant (){
        return this.solde ;
```

```
    }

    public void depot(double montant){
        this.solde += montant ;
    }

    public void retrait(double montant)
                                throws provisionInsuffisanteErreur {
        System.out.println("Appel de retrait sur compte simple");
        if (this.solde < montant){
            throw new provisionInsuffisanteErreur() ;
        }else{
            solde -= montant ;
        }
    }

    public void virement(CompteBancaire c, double montant)
                                throws provisionInsuffisanteErreur{
        c.retrait(montant);
        this.depot(montant);
    }
}

class provisionInsuffisanteErreur extends Exception{
}
```

On peut alors utiliser cette classe dans le programme suivant :

```
class Test1ComptesBancaires {
    public static void main(String[] args)
                                    throws provisionInsuffisanteErreur{
        CompteBancaire c1 ;
        CompteBancaire c2 ;
        CompteBancaire c3 ;
        String num1 = "123456789";
        String num2 = "145775544";
        String num3 = "A4545AA54";
        c1 = new CompteBancaire("Paul", num1.toCharArray(), 1000.00);
        c2 = new CompteBancaire("Paul", num2.toCharArray(), 2300.00);
        c3 = new CompteBancaire("Henri", num3.toCharArray(), 5000.00);
        c1.depot(100.00);
        c2.virement(c1, 1000.00);
        Terminal.ecrireDouble(c2.soldeCourant());
    }
}
```

3.2 Extension des données, notion d'héritage et syntaxe

Supposons maintenant que l'on souhaite prendre en considération la possibilité d'avoir un découvert sur un compte (un retrait conduit à un solde négatif) ou la possibilité d'associer une rémunération des

dépôts sur un compte (compte rémunéré) ; ces cas ne sont pas prévus dans la classe que nous avons définie. Une première possibilité est de modifier le code source de la classe précédente. Ceci peut impacter tout programme (ou toute classe) qui utilise cette classe et conduit à une difficulté : comment définir un compte rémunéré sans découvert ou un compte avec découvert possible mais non rémunéré. Il faudrait dans ce cas dupliquer à la main ces classes pour les particulariser ; il n'est pas difficile d'imaginer l'ensemble des modifications qu'entraînerait cette solution sur les programmes utilisant la classe `CompteBancaire`.

Les langages de programmation modernes offrent différentes solutions à ce problème ; le but est chaque fois de permettre une extension ou une spécialisation des types de base. Avec la programmation par objets ceci se réalise naturellement avec la notion **d'héritage**. Le principe consiste à définir une nouvelle classe à partir d'une classe existante en "ajoutant" des données à cette classe et en réutilisant de façon implicite toutes les données de la classe de base. Si A est la classe de base et si B est la classe construite à partir de A on dit que B "hérite" de A ou encore que B "spécialise" A ou encore que B "dérive" de A. On dira également que A est la classe "mère" de B et que B est une sous-classe, ou une classe "fille" de A. Dans le cas où une classe hérite d'une seule classe (c'est le cas en Java) on parle d'héritage simple (au lieu d'héritage multiple) et la relation mère-fille définit un arbre comme représenté par le dessin suivant où $B1$ et $B2$ héritent de A et où C hérite de $B1$ (et donc également par transitivité de A). On notera qu'en Java toute classe hérite de la classe `Object`.

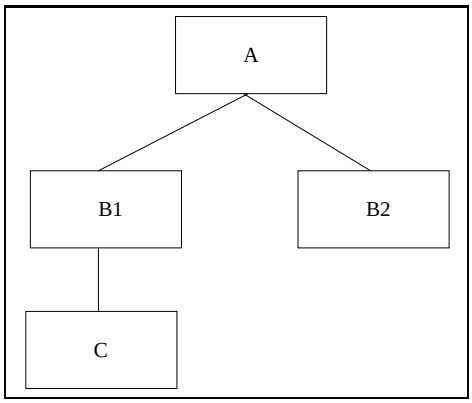


FIG. 3.1 – Hiérarchie de classes

Pour être plus précis, lors de la construction d'une nouvelle classe par extension d'une classe existante, on distingue trois sortes de méthodes (ou de variables) :

- les méthodes (et les variables) qui sont propres à la nouvelle classe ; il s'agit là d'extension. Ces méthodes (données) n'existent que pour les objets de la nouvelle classe et portent un nom qui n'est pas utilisé dans la classe mère ou, un nom déjà utilisé mais avec des paramètres en nombre ou de type différents (il y a alors une surcharge du nom de la méthode).
- les méthodes (et les variables) qui sont issues de la classe mère ; il s'agit là d'héritage. Ces méthodes (données) sont les même pour les objets de la classe mère et pour les objets de la classe héritée. Par défaut, toute méthode (variable) est héritée. On notera que lors de l'utilisation d'une méthode héritée sur un objet de la classe fille il y a une conversion implicite de l'objet de la classe fille en un objet de la classe mère. Le code d'une méthode (ou la donnée) héritée est soit partagé lorsque la méthode est statique soit propre à chaque instance dans le cas contraire (cas par défaut).
- les méthodes (ou les variables, mais cela est plus rare) qui redéfinissent des méthodes (variables) existantes dans la classe mère ; il s'agit là de masquage. Ces méthodes ont le même nom et le même profil (nombre et type des paramètres, valeur de retour) que les méthodes qu'elles redéfinissent. Dans

ce cas, la nouvelle méthode se substitue à la méthode de la classe mère. Ceci permet de prendre en compte l'enrichissement (ou la spécialisation) que la classe fille apporte par rapport à la classe mère en spécialisant le code de la méthode. Lorsque dans une classe dérivée on souhaite désigner une méthode (ou une donnée) de la classe mère (et en particulier lorsque celle-ci est masquée) on désigne celle-ci en préfixant le nom de la méthode (ou de la variable) par `super`. Ainsi, si B dérive de A et si x est un nom utilisé dans A et B, la dénomination `super.x` dans une méthode de B désignera x relatif à A et non x relatif à B (qui serait désigné par `this.x`). Dans le cas particulier du constructeur, un appel à `super` désigne un appel au constructeur approprié (celui dont le nombre et le type des paramètres conviennent) de la classe mère.

La syntaxe utilisée en Java pour définir qu'une classe B étend une classe A consiste à préciser lors de la déclaration de la nouvelle classe cette extension avec la notation `class B extends A`. Tout ce qui sera défini au niveau de cette nouvelle classe B sera propre à B ; tout ce qui n'est pas redéfini sera hérité de A, c'est-à-dire qu'un objet instance de B pourra accéder à ces données ou méthodes comme un objet instance de A.

Appliquons ces principes à la construction de deux classes dérivées de la classe `CompteBancaire`. La première permettra de représenter des comptes bancaires avec découvert autorisé ; la seconde prendra en compte une possible rémunération du compte. Pour le premier type de compte, il faut préciser quel est le découvert maximum autorisé. Pour le second type de compte, il faudra définir quel est la rémunération (taux) et, par exemple, quel est le seuil minimal à partir duquel la rémunération s'applique.

3.3 Extension de la classe `CompteBancaire` en `CompteAvecDecouvert`

Pour cette nouvelle classe nous introduisons une nouvelle donnée : le montant maximum du découvert. Cela nous conduit à modifier la méthode `retrait` de telle sorte qu'un retrait puisse être effectuer même si le solde n'est pas suffisant (il faudra néanmoins que le solde reste plus grand que l'opposé du découvert maximal autorisé). Cette méthode `retrait` masquera la méthode de même nom de la classe mère. Par ailleurs, on introduira une nouvelle méthode `fixeDecouvertMaximal` qui permet de modifier le montant du découvert maximum autorisé. Les autres données et méthodes seront reprises telles quelles (par le mécanisme d'héritage). Pour la définition du constructeur on utilise le constructeur de la classe mère (par un appel à `super`). Notons que `super` est toujours appelé dans un constructeur soit de manière implicite en début d'exécution du constructeur soit de manière explicite par un appel à `super`.

```
public class CompteAvecDecouvert extends CompteBancaire{
    double decouvertMax ;

    public void fixeDecouvertMaximal(double montant){
        this.decouvertMax = montant ;
    }

    public CompteAvecDecouvert (String proprio,
                                char[] num,
                                double montant,
                                double decouvertMax){
        super(proprio, num, montant) ;
        this.decouvertMax = decouvertMax;
    }

    public void retrait(double montant)
```

```
                                    throws provisionInsuffisanteErreur {
        System.out.println("Appel de retrait sur compte avec decouvert");
        if (this.solde - montant < -decouvertMax){
            throw new provisionInsuffisanteErreur() ;
        }else{
            solde -= montant ;
        }
    }
}
```

On peut alors completer le programme de test par les instructions suivantes :

```
        CompteAvecDecouvert d1;
        String num4 = "DD545AA54";

        d1 = new CompteAvecDecouvert("Jacques", num4.toCharArray(),
                                    6000.00, 0.00) ;

        try {
            d1.retrait(7000.00);
            System.out.println("retrait bien passee ;" +
                               "nouveau solde = " + d1.soldeCourant() );
        } catch (Exception e)
            {System.out.println(
               "Un probleme est survenu lors du premier retrait");};

        d1.fixeDecouvertMaximal(2000.00);
        try {
            d1.retrait(7000.00);
            System.out.println("retrait bien passee ;"+
                               "nouveau solde = " + d1.soldeCourant() );
        } catch (Exception e)
            {System.out.println(
               "Un probleme est survenu lors du second retrait");};

        d1.depot(5000.00);
```

3.4 Extension de la classe `CompteBancaire` en `CompteAvecRemuneration`

Pour cette extension, il faut au minimum connaitre le taux de rémunération du compte à partir duquel la rémunération s'applique. On suppose que les intérêts sont versés à part et intégrés sur le compte une fois l'an. Afin de simplifier l'écriture de cette classe nous supposerons qu'il existe une méthode qui calcule les intérêts (le code est un peu complexe et nécessite la connaissance des dates auquelles sont faites les opérations afin de calculer les durées pendant lesquels l'inétrêt s'applique).

```
public class CompteRemunere extends CompteBancaire{
    double taux ;
    double interets;

    public void fixeTaux(double montant){
```

```
        this.taux = montant ;
    }

    public CompteRemunere (String proprio, char[] num, double montant,
                           double taux){
        super(proprio, num, montant) ;
        fixeTaux(montant);
        interets = 0.0;
    }

    public void retrait(double montant)
                                    throws provisionInsuffisanteErreur {
        System.out.println("Appel de retrait sur compte remunere");
        if (this.solde < montant){
            throw new provisionInsuffisanteErreur() ;
        }else{
            solde -= montant ;
        }
    }

    public void calculInteret(){
        interets += 1.0; // bien sur, le code reel est plus complexe
    }

}
```

On peut alors définir un compte rémunéré avec découvert autorisé. Il suffit pour cela de dériver de la classe `CompteRemunere` la classe `CompteRemunereAvecDecouvert`. Cela se fait simplement de la façon suivante (il suffit d'introduire une nouvelle variable et un nouveau constructteur et de redéfinir la méthode de retrait) :

```
public class CompteRemunereAvecDecouvert extends CompteRemunere{
    double decouvertMax ;

    public void fixeDecouvertMaximal(double montant){
        this.decouvertMax = montant ;
    }

    public CompteRemunereAvecDecouvert (String proprio,
                                        char[] num,
                                        double montant,
                                        double taux,
                                        double decouvertMax){
        super(proprio, num, montant, taux) ;
        // ici super designe la classe CompteRemunere
        fixeDecouvertMaximal(decouvertMax);
    }

    public void retrait(double montant)
                    throws provisionInsuffisanteErreur {
        System.out.println(
          "Appel de retrait sur compte remunere avec decouvert");
```

```
        if (this.solde - montant < -decouvertMax){
            throw new provisionInsuffisanteErreur() ;
        }else{
            solde -= montant ;
        }
    }
}
```

On obtient alors l'arbre de classes suivant :

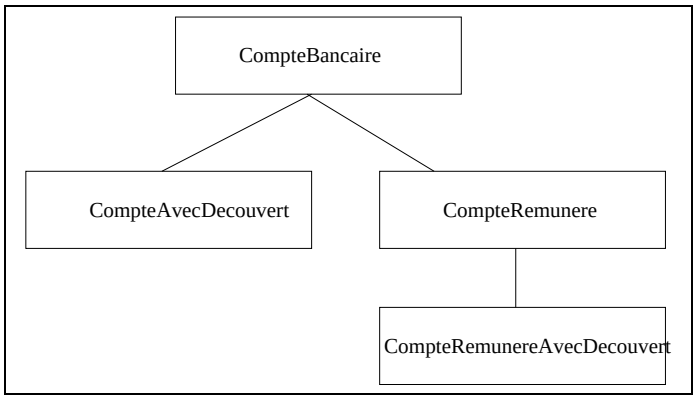


FIG. 3.2 – Hiérarchie des classes définies dans ce cours

Cette nouvelle classe peut être utilisée de la façon suivante :

```
        dd = new CompteRemunereAvecDecouvert("Pierre", num5.toCharArray(),
                                             6000.00, 0.00, 0.05) ;

        dd.depot(1000.00);
        try {
            dd.retrait(8000.00);
            System.out.println(
                 "retrait 1 bien passee sur dd; nouveau solde = "
                 + dd.soldeCourant() );
        } catch (Exception e)
            {System.out.println(
                 "Un probleme est survenu lors d'un retrait");};

        dd.fixeDecouvertMaximal(2000.00);

        try {
            dd.retrait(8000.00);
            System.out.println(
               "retrait 2 bien passee sur dd; nouveau solde = "
               + dd.soldeCourant() );
        } catch (Exception e)
            {System.out.println(
               "Un probleme est survenu lors d'un retrait");};

        dd.depot(5000.00);
```

```
dd.fixeTaux(0.06);
```

On notera que, la méthode `retrait` utilisée lors de ces instructions est celle qui est redéfinie dans la nouvelle classe. La méthode `fixeTaux` est par contre la méthode hérité de la classe `CompteRemunere` et les méthodes `depot` et `soldeCourant` sont celles héritées de la classe mère `CompteBancaire`. On le voit, l'héritage est transitif.

3.5 Transtypage, classe déclarée et classe d'exécution

En programmation fortement typée le mélange des types n'est permis qu'avec le respect de règles strictes. Il est par exemple interdit d'affecter une valeur réelle à une variable booléenne. Il existe néanmoins des possibilités de convertir une valeur d'un type donnée en une valeur d'un autre type ; on parle de transtypage ou de conversion de type (cast). Cette conversion peut être implicite lorsqu'elle est fait automatiquement par le compilateur (conversion directe ou insertion de code de conversion) ou explicite par l'emploi d'une notation appropriée. Ce mécanisme a déjà été étudié en début d'année pour les types élémentaires de Java.

La règle liée à la programmation objet est que toute classe est compatible avec ses sous-classes c'est à dire que si B est une sous-classe de A alors toute variable de type A peut être affectée par une valeur de type B (par une instruction d'affectation ou lors d'un passage de paramètre). Par exemple, si $c1$ est un compte bancaire et si $d1$ est un compte bancaire avec découvert autorisé alors l'affectation $c1 = d1$ est correcte. Par contre, l'affectation $d1 = c1$ sera incorrecte. De même une instruction `c1.virement(d1, 100.0)` sera correcte et il y aura une conversion implicite de $d1$ en une valeur de type `compteBancaire` lors de l'appel de la méthode `virement`. On peut bien entendu également exécuter l'instruction `d1.virement(c1, 100.0)`.

Lors de l'appel `c1.virement(d1, 100.0)`, le paramètre formel de la méthode virement désignant le compte sur lequel on va opérer un retrait (nommons le c) est associé au paramètre effectif $d1$. Une conversion a lieu. Cependant, c qui est de classe déclarée `CompteBancaire` sera de classe réelle `CompteAvecDecouvert` lors de l'exécution de cet appel (il restera associé au paramètre effectif $d1$ qui est une instance de la classe `CompteAvecDecouvert`. Ainsi, l'instruction `c1.retrait(montant)` de cette méthode fera appel à la méthode `retrait` de la classe `CompteAvecDecouvert` qui est la classe réelle de c lors de cet appel. Par contre, l'appel de `d1.virement(c1, 100.0)`, conduit à un appel de la méthode méthode `retrait` de la classe `CompteBancaire` car cette fois-ci, le paramètre formel c est associé à la variable $c1$ qui est un objet de la classe `CompteBancaire` et donc, la classe réelle de c lors de cette exécution est `CompteBancaire`. Les instructions suivants permettent de visualiser ces principes (les méthodes `retrait` affichent des messages différents).

```
d1.depot(5000.00);
try {
    c1.virement(d1, 1000.00);
    d1.virement(c1, 1000.00);
} catch (Exception e)
    {System.out.println(
        "Un probleme est survenu lors d'un virement");};
```

3.6 Liaison dynamique

Dans les exemples précédents, les méthodes appelées à l'exécution du programme sont connues à la compilation.

Supposons maintenant que l'on déclare un compte bancaire c sans l'initialiser :

```
CompteBancaire c;
String num = "AAA4AA54";
```

Les règles de typages de Java font qu'il est possible d'associer à c par création un compte ordinaire ou un compte avec découvert. En effet, toute classe est compatible avec ses sous-classes, et donc, c qui est de la classe `CompteBancaire` est compatible avec la classe `CompteAvecDecouvert`.

Les deux instructions

```
c = new CompteBancaire("Marie", num.toCharArray(), 10000.00) ;
```

```
c = new CompteAvecDecouvert ("Marie", num.toCharArray(), 10000.00, 0.0) ;
```

sont donc correctes ; dans le premier cas, c sera de classe réelle `CompteBancaire` et dans le second cas de classe réelle `CompteAvecDecouvert`. Donc, l'instruction `c.retrait(100.00)` appelera soit la méthode `retrait` des comptes avec découvert (second cas) soit la méthode `retrait` des comptes ordinaires. Cette liaison entre appel et méthode peut se faire lors de la compilation (dans le cas où le compilateur a suffisamment d'informations pour calculer cette liaison) ou simplement lors de l'exécution du programme ; on parle alors de liaison tardive. La suite d'instructions suivantes illustre ce mécanisme : selon la réponse faite par l'utilisateur, le programme associe à c un compte ordinaire ou un compte avec découvert.

```
public class TestLiaisonTardive{

    public static void main(String[] args){
        CompteBancaire c;
        String num = "AAA4AA54";

        Terminal.ecrireString("Voulez vous creer un compte decouvert O/N : ");

        char reponse;
        reponse = Terminal.lireChar();
        if ((reponse != 'O') && (reponse != 'o')){
            Terminal.ecrireString("Creation d'un compte ordinaire");

            c = new CompteBancaire("Marie", num.toCharArray(), 10000.00) ;
        }
        else{
            Terminal.ecrireString("Creation d'un compte avec decouvert");

            Terminal.ecrireString("Quel est le couvert maximal autorisé : ");
            double max;
            max = Terminal.lireDouble();
            c = new CompteAvecDecouvert ("Marie", num.toCharArray(), 10000.00, max) ;
        }

        System.out.println("solde avant retrait " + c.soldeCourant() );
```

```
        try {
            System.out.println("Tentative de retrait de 11000.00");
            c.retrait(11000.00);
            System.out.println("le retrait c'est bien passee; nouveau solde = "
                               + c.soldeCourant() );
        }
        catch (Exception e)
            {System.out.println("Un probleme est survenu lors du retrait");}
    }
}
```

3.7 Droits d'accès

Pour terminer ce chapitre détaillons les façons de permettre ou d'interdire l'accès aux méthodes ou aux variables d'une classe grace aux mots cléss `public`, `private`, `protected` utlisés lors de leur déclaration.

Les classes défines jusqu'ici ont toujours donné accès à toutes leurs variables d'instance et à leurs méthodes, en ne précisant pas de protection (on verra que ne pas préciser de protection ne donne pas toujours tous les droits d'accès).

D'une manière générale :

- toute classe doit donner accès à une partie de ses variables d'instance ou méthodes, sinon elle est inutilisable
- donner accès à tout n'est pas une bonne stratégie

La façon de protéger les éléments d'une classe ne dépend que du programmeur, mais il existe quelques principes généralement respectés :

- une classe décrit une catégorie d'objets, qui offrent un certain nombre de *fonctionnalités*. Exemple : une liste, qui permet de rajouter un élément, de supprimer un élément, de rechercher un élément, de calculer sa longueur, etc.
- *on donne accès aux fonctionnalités* de la classe, mais *on cache la manière dont ces fonctionnalités sont implémentées*.
- ainsi, un programme qui utilise les fonctionnalités d'une classe ne devra pas changer si seule l'implémentation des fonctionnalités change.
- typiquement, les fonctionnalités sont offertes à travers un ensemble de méthodes de la classe, tandis que leur implémentation concerne les variables d'instance, d'autres méthodes auxiliaires, parfois des classes auxiliaires.

Donc très souvent *l'accès aux variables d'instance est protégé*, tandis que *les méthodes principales* (qui expriment les fonctionnalités des objets de la classe) *sont accessibles*.

1. Lorsqu'une méthode, une variable (ou une classe) est marquée `public` (comme par exemple `public int x;`) elle est accessible (et donc utlisable) à partir de n'importe quelle autre méthode de n'importe quelle classe.
2. Lorsqu'une méthode ou une variable est marquée `private` elle n'est accessible qu'à l'intérieur de la classe où elle définie.

3. Lorsqu'une ou une variable est marquée `protected` elle est utilisable à l'intérieur de la classe où elle définie et aussi dans toutes les méthodes des classes dérivées de cette classe.
4. Enfin, si rien n'est précisé, la classe, la méthode ou la variable est accessible par toute méthode de toute classe du même **paquetage** (manière de regrouper des classes en Java).

Remarque : le paquetage auquel appartient une classe est déclaré en tant que première instruction dans le fichier de la classe, à l'aide du mot clé `package`. Par exemple, toutes les classes représentant des comptes pourraient faire partie d'un paquetage *comptes*, ce qui est déclaré au début de chacune des classes par
`package comptes;`

En absence de déclaration de paquetage, on considère que la classe fait partie du *paquetage par défaut*, qui regroupe toutes les classes défines dans un même répertoire.

Remarque : les classes `CompteBancaire` et dérivées fonctionnent correctement (telles qu'elles sont définies) seulement si elles se trouvent dans le même répertoire (même paquetage). Cela permet à toute classe d'avoir accès à toutes les variables d'instance et méthodes des autres classes.

Remarque : si pour les classes `CompteBancaire` et dérivées on déclare toutes les variables d'instance `private` et toutes les méthodes (et constructeurs) `public`, le code n'est pas correct, car les classes dérivées ont besoin d'accéder aux variables d'instance des classes de base. Exemple : la méthode *retrait* des classes dérivées de `CompteBancaire` utilise la variable s'instance *solde* de `CompteBancaire`!

Dans ce cas, il y a deux solutions typiques :

1. déclarer la variable d'instance en question comme `protected`.
2. rajouter des méthodes publiques *valeurSolde* et *modifierSolde* à `CompteBancaire` pour avoir accès aux opérations sur le solde dans toutes les classes.

Le mot-clé `final` exprime lui aussi une certaine forme de "protection" des données. Il exprime *le caractère constant de l'élément en question* :

– pour une variable, il s'agit d'une *définition de constante*, qui doit avoir une valeur initiale et qui ne pourra plus jamais en changer. Les constantes sont définies comme des variables finales et statiques (de classe).
Ex. `static final int LIMITE_DECOUVERT=500;`
– pour une classe, il s'agit d'une classe à partir de laquelle on ne peut pas dériver de sous-classes.
– pour une méthode, il s'agit d'une méthode qui ne peut pas être masquée (remplacée par une autre définition) dans les sous-classes.

3.8 Classes abstraites et interfaces

3.8.1 Classes abstraites

Une classe abstraite est une classe dont le rôle dans la hiérarchie d'héritage est de faire le lien entre plusieurs autres classes, *sans permettre la création d'objets*.

Généralement, une classe abstraite sert de parent (ancêtre) commun aux classes "concrètes" qu'elle relie. En d'autres mots, *la classe abstraite sert à définir des caractéristiques communes à plusieurs sous-classes*, sans pour autant y avoir d'objet comme instance cette classe.

Supposons qu'il existe en plus des comptes bancaires *des comptes postaux*, qui ont beaucoup de caractéristiques en commun avec les comptes bancaires. On peut alors créer une classe abstraite `CompteAbstrait`, super-classe pour `CompteBancaire` et `ComptePostal`.

```
abstract public class CompteAbstrait{
    String nomProprietaire ;
    char[] numero ;
    double solde ;

    public CompteAbstrait(){}
    public CompteAbstrait(String proprio, char[] num, double montant){
            this.nomProprietaire = proprio ;
            this.numero = num ;
            this.solde = montant ;
    }
    public double soldeCourant (){
            return this.solde ;
    }
    abstract public void depot(double montant);
    abstract public void retrait(double montant)
                                throws provisionInsuffisanteErreur;
}

public class CompteBancaire extends CompteAbstrait{
    public CompteBancaire(String proprio, char[] num, double montant){
            super(proprio, num, montant);
    }
    public void depot(double montant){...}
    public void retrait(double montant)
                                throws provisionInsuffisanteErreur{...}
    public void virement(CompteBancaire c, double montant)
                                throws provisionInsuffisanteErreur{...}
}

public class ComptePostal extends CompteAbstrait{
    Livret livretA;

    public ComptePostal(String proprio, char[] num, double montant){
            super(proprio, num, montant);
            this.livretA = ...;
    }
    public void depot(double montant){...}
    public void retrait(double montant)
                                throws provisionInsuffisanteErreur{...}
    public void depotLivret(double montant){...}
    public double soldeLivret(){...}
}
```

Il n'existe pas de compte de type `CompteAbstrait`, mais seulement des comptes bancaires ou des comptes postaux. Le rôle de la classe `CompteAbstrait` est de réunir les caractéristiques communes des deux types de comptes. Tous les types de compte ont un nom de propriétaire, un numéro et un solde, ainsi que des méthodes pour consulter le solde et pour faire des dépôts et des retraits d'argent.

Parmi les méthodes de `CompteAbstrait`, $depot$ et $retrait$ sont des *méthodes abstraites*, ce qui signifie qu'elles ne sont définies que dans les sous-classes "concrètes". Seule l'entête de la méthode abstraite

est définie dans la classe abstraite, ce qui oblige toute sous-classe concrète de définir l'implémentation qui lui est spécifique.

Par contre, la méthode *soldeCourant* est une méthode non abstraite, commune à tous les types de compte, définie de la même façon que dans les hiérarchies d'héritage des classes non abstraites. L'implémentation de cette méthode est donc commune à toutes les sous-classes (à moins qu'une sous-classe souhaite la rédéfinir).

De même, le constructeur de `CompteAbstrait` est utilisé dans les sous-classes, de la même façon que dans les hiérarchies d'héritage des classes non abstraites. Par contre, ce constructeur ne sera jamais appelé pour créer un objet `CompteAbstrait`.

En plus de `CompteAbstrait`, `CompteBancaire` définit une méthode *virement*, tandis que `ComptePostal` relie le compte à un livret A, avec deux méthodes supplémentaires, *depotLivret*, pour transférer un montant donné du compte postal vers le livret et *soldeLivret*, pour consulter le solde du livret.

Dans les programmes utilisant ces classes, `CompteAbstrait` est utilisée comme une super-classe de `CompteBancaire` et `ComptePostal`, avec l'interdiction de créer d'objet `CompteAbstrait`.

Le transtypage est possible pour les classes abstraites, de la même façon que pour les classes non-abstraites. Par exemple, un objet `CompteBancaire` peut être représenté par une variable de type `CompteAbstrait`.

3.8.2 Interfaces

Les interfaces offrent un mécanisme complémentaire d'héritage de fonctionnalités entre classes en Java.

Une interface est similaire à *une classe abstraite qui ne définit que des méthodes abstraites*. Une interface ne définit donc pas de constructeur, ni de variables d'instance.

Une classe qui "hérite" d'une interface *doit* implémenter *toutes* les méthodes de l'interface. Bien sûr, la classe en question peut définir d'autres méthodes, des variables d'instance et des constructeurs.

On dit qu'une classe *implémente* une interface et non pas qu'elle étend l'interface (ou qu'elle hérite de l'interface). L'implémentation d'interfaces est le seul moyen en Java pour réaliser de l'héritage multiple : *une classe peut implémenter plusieurs interfaces.*

Dans l'exemple précédent, le compte abstrait peut être représenté par une interface `InterfaceCompte`. On peut également imaginer une interface `InterfaceLivret` qui définit les opérations de dépôt et de consultation du livret. Dans ce cas, l'exemple pourra prendre la forme suivante :

```
public interface InterfaceCompte{
    public double soldeCourant ();
    public void depot(double montant);
    public void retrait(double montant)
                        throws provisionInsuffisanteErreur;
}

public class CompteBancaire implements InterfaceCompte{
    String nomProprietaire ;
    char[] numero ;
    double solde ;

    public CompteBancaire(){}
    public CompteBancaire(String proprio, char[] num, double montant){
            this.nomProprietaire = proprio ;
            this.numero = num ;
            this.solde = montant ;
    }
    public double soldeCourant (){...}
```

```
    public void depot(double montant){...}
    public void retrait(double montant)
                        throws provisionInsuffisanteErreur{...}
    public void virement(CompteBancaire c, double montant)
                         throws provisionInsuffisanteErreur{...}
}

public interface InterfaceLivret{
    public double soldeLivret();
    public void depotLivret(double montant);
}

public class ComptePostal implements InterfaceCompte, InterfaceLivret{
    String nomProprietaire ;
    char[] numero ;
    double solde ;
    Livret livretA;

    public ComptePostal(String proprio, char[] num, double montant){
            this.nomProprietaire = proprio ;
            this.numero = num ;
            this.solde = montant ;
            this.livretA = ...;
    }
    public void depot(double montant){...}
    public void retrait(double montant)
                        throws provisionInsuffisanteErreur{...}
    public void depotLivret(double montant){...}
    public double soldeLivret(){...}
}
```

La classe `CompteBancaire` implémente l'interface `InterfaceCompte`, tandis que la classe `ComptePostal` implémente deux interfaces : `InterfaceCompte` et `InterfaceLivret`. Les interfaces ne précisent que les entêtes des méthodes à implémenter - leur corps, ainsi que toutes les variables d'instance sur lesquelles se font les actions, sont décrits dans les classes.

Remarque : pour respecter la définition des comptes bancaires et postaux introduits dans la section sur les classes abstraites, on retrouve les mêmes variables d'instance (nom propriétaire, numéro et solde) avec les mêmes types dans les classes `CompteBancaire` et `ComptePostal`. En pratique, deux classes qui implémentent la même interface peuvent avoir des variables d'instance totalement différentes, la seule restriction est d'implémenter les méthodes de l'interface, chaque classe à sa façon.

Comme pour les classes abstraites, *le transtypage* est également possible pour les interfaces. On peut déclarer des variables, des paramètres et des résultats de méthodes de type interface. La différence est que l'"héritage" multiple pour les interfaces permet à un même objet de jouer des rôles différents, un pour chacune des interfaces qu'il implémente. Par exemple, un objet `ComptePostal` peut être représenté à la fois par une variable de type `InterfaceCompte` et par une de type `InterfaceLivret`.

Autres caractéristiques des interfaces :

- Les interfaces définissent toujours *des méthodes publiques*.
- Les seuls autres éléments qui peuvent être définis dans une interface, à part les méthodes, sont *les constantes*. Les constantes en question seront définies dans toutes les classes qui implémentent l'in-

terface.
Ex. `static final int LIMITE_DECOUVERT=500;`

– Une interface ne peut pas implémenter d'autres interfaces, par contre elle peut *étendre* une interface plus générale. Il est donc possible de construire des *hiérarchies d'héritage pour les interfaces*. Par exemple, dans le paquetage `java.util`, l'interface `Collection<E>` a plusieurs sous-interfaces, parmi lesquelles `List<E>` et `Set<E>`.

Chapitre 4

Les exceptions

4.1 Introduction : qu'est-ce qu'une exception ?

De nombreux langages de programmation de haut niveau possède un mécanisme permettant de gérer les erreurs qui peuvent intervenir lors de l'exécution d'un programme. Le mécanisme de gestion d'erreur le plus répandu est celui des exceptions. Nous avons déjà abordé le concept d'exception dans le cours sur les fonctions : lorsqu'une fonction n'est pas définie pour certaines valeur de ses arguments on lève une exception en utilisant le mot clef `throw`. Par exemple, la fonction factorielle n'est pas définie pour les nombres négatifs, et pour ces cas, on lève une exception :

```
class Factorielle {
    static int factorielle(int n){
        int res = 1;
        if (n<0){
            throw new PasDefini();
        }
        for(int i = 1; i <= n; i++) {
            res = res * i;
        }
        return res;
    }
}

class PasDefini extends Error {}
```

Une exception signale une erreur comme lorsqu'un nombre négatif est passé en argument à la fonction factorielle. Jusqu'ici, lever une exception signifiait interrompre définitivement le programme avec l'affichage d'un message d'erreur décrivant l'exception à l'écran. Cependant, il est de nombreuses situations où le programmeur aimerait gérer les erreurs sans que le programme ne s'arrête définitivement. Il est alors important de pouvoir intervenir dans le cas où une exception a été levée. Les langages qui utilisent les exceptions possèdent toujours une construction syntaxique permettant de "rattraper" (ou "récupérer") une exception, et d'exécuter un morceau de code spécifique à ce traitement d'erreur.

Examinons maintenant comment définir, lever et récupérer une exception en Java.

4.2 Définir des exceptions

Afin de définir une nouvelle sorte d'exception, on crée une nouvelle classe en utilisant une déclaration de la forme suivante :

```
class NouvelleException extends ExceptionDejaDefinie {}
```

On peut remarquer ici la présence du mot clé **extends**, dont nous verrons la signification dans un chapitre ultérieur qui traitera de l'héritage entre classes. Dans cette construction, `NouvelleException` est le nom de la classe d'exception que l'on désire définir en "étendant" `ExceptionDejaDefinie` qui est une classe d'exception déjà définie. Sachant que `Error` est prédéfinie en Java, la déclaration suivante définit la nouvelle classe d'exception `PasDefini` :

```
class PasDefini extends Error {}
```

Il existe de nombreuses classes d'exceptions prédéfinies en Java, que l'on peut classer en trois catégories :

- Celles définies par extension de la classe `Error` : elles représentent des erreurs critiques qui ne sont pas censées être gérées en temps normal. Par exemple, une exception de type `OutOfMemoryError` est levée lorsqu'il n'y a plus de mémoire disponible dans le système. Comme elles correspondent à des erreurs critiques elles ne sont pas normalement censées être récupérées et nous verrons plus tard que cela permet certaines simplifications dans l'érciture de méthodes pouvant lever cette exception.
- Celles définies par extension de la classe `Exception` : elles représentent les erreurs qui doivent normalement être gérées par le programme. Par exemple, une exception de type `IOException` est levée en cas d'erreur lors d'une entrée sortie.
- Celles définies par extension de la classe `RuntimeException` : elles représentent des erreurs pouvant éventuellement être gérées par le programme. L'exemple typique de ce genre d'exception est `NullPointerException`, qui est levée si l'on tente d'accéder au contenu d'un tableau ou d'un objet qui vaut `null`.

Chaque nouvelle exception entre ainsi dans l'une de ces trois catégories. Si on suit rigoureusement ce classement (et que l'on fait fi des simplifications évoquées plus haut), l'exception `PasDefini` aurait due être déclarée par :

```
class PasDefini extends Exception {}
```

ou bien par :

```
class PasDefini extends RuntimeException {}
```

car elle ne constitue pas une erreur critique.

4.3 Lever une exception

Lorsque l'on veut lever une exception, on utilise le mot clé `throw` suivi de l'exception à lever, qu'il faut avoir créée auparavant avec la construction **`new`** `NomException()` (comme tout objet en Java). Ainsi lancer une exception de la classe `PasDefini` s'écrit :

```
throw new PasDefini();
```

Lorsqu'une exception est levée, l'exécution normale du programme s'arrête et on saute toutes les instructions jusqu'à ce que l'exception soit rattrapée ou jusqu'à ce que l'on sorte du programme. Par exemple, si on considère le code suivant :

```
public class Arret {
    public static void main(String [] args) {
        int x = Terminal.lireInt();

        Terminal.ecrireStringln("Coucou 1");        // 1
        if (x >0){
            throw new Stop();
        }
        Terminal.ecrireStringln("Coucou 2");        // 2
        Terminal.ecrireStringln("Coucou 3");        // 3
        Terminal.ecrireStringln("Coucou 4");        // 4
    }
}
class Stop extends RuntimeException {}
```

l'exécution de la commande `java Arret` puis la saisie de la valeur 5 (pour la variable `x`) produira l'affichage suivant :

```
Coucou 1
Exception in thread "main" Stop
        at Arret.main(Arret.java:7)
```

C'est-à-dire que les instructions 2, 3 et 4 n'ont pas été exécutées. Le programme se termine en indiquant que l'exception `Stop` lancée dans la méthode `main` à la ligne 7 du fichier `Arret.java` n'a pas été rattrapée.

4.4 Rattraper une exception

4.4.1 La construction `try catch`

Le rattrapage d'une exception en Java se fait en utilisant la construction :

```
try {
    ... // 1
} catch (UneException e) {
    ... // 2
}
.. // 3
```

Le code 1 est normalement exécuté. Si une exception est levée lors de cette exécution, les instructions restantes dans le code 1 sont abandonnées. Si la classe de l'exception levée dans le bloc 1 est `UneException` alors le code 2 est exécuté (car l'exception est récupérée). Dans le code 2, on peut faire référence à l'exception en utilisant le nom donné à celle-ci lorsque l'on nomme sa classe. Ici le nom est `e`. Dans le cas où la classe de l'exception n'est pas `UneException`, le code 2 **et le code 3** sont sautés. Ainsi, le programme suivant :

```
public class Arret2 {
    public static void P () {
        int x = Terminal.lireInt();

        if (x >0){
            throw new Stop();
```

```
        }
    }
    public static void main(String [] args) {
        Terminal.ecrireStringln("Coucou 1");        // 1

        try {
            P ();
            Terminal.ecrireStringln("Coucou 2");    // 2
        } catch (Stop e){
            Terminal.ecrireStringln("Coucou 3");    // 3
        }
        Terminal.ecrireStringln("Coucou 4");        // 4
    }
}
class Stop extends RuntimeException {}
```

produit l'affichage suivant lorsqu'il est exécuté et que l'on saisit une valeur positive :

```
Coucou 1
Coucou 3
Coucou 4
```

On remarquera que l'instruction 2 n'est pas exécuté (du fait de la levée de l'exception dans `P`.

Si on exécute ce même programme mais en saisissant une valeur négative on obtient :

```
Coucou 1
Coucou 2
Coucou 4
```

car l'exception n'est pas levée.

En revanche le programme suivant, dans lequel on lève une exception `Stop2`, qui n'est pas récupérée.

```
public class Arret3 {
    public static void P () {
        int x = Terminal.lireInt();

        if (x >0){
            throw new Stop2();
        }
    }
    public static void main(String [] args) {
        Terminal.ecrireStringln("Coucou 1");        // 1
        try {
            P ();
            Terminal.ecrireStringln("Coucou 2");    // 2
        } catch (Stop e){
            Terminal.ecrireStringln("Coucou 3");    // 3
        }
        Terminal.ecrireStringln("Coucou 4");        // 4
    }
}
class Stop extends RuntimeException {}
class Stop2 extends RuntimeException {}
```

produit l'affichage suivant lorsqu'il est exécuté et que l'on saisit une valeur positive :

```
Coucou 1
Exception in thread "main" Stop2
        at Arret3.P(Arret3.java:7)
        at Arret3.main(Arret3.java:15)
```

4.4.2 Rattraper plusieurs exceptions

Il est possible de rattraper plusieurs types d'exceptions en enchaînant les constructions `catch` :

```
public class Arret3 {

    public static void P () {
        int x = Terminal.lireInt();

        if (x >0){
            throw new Stop2();
        }
    }

    public static void main(String [] args) {
        Terminal.ecrireStringln("Coucou 1");        // 1

        try {
            P ();
            Terminal.ecrireStringln("Coucou 2");    // 2
        } catch (Stop e){
            Terminal.ecrireStringln("Coucou 3");    // 3
        }
        } catch (Stop2 e){
            Terminal.ecrireStringln("Coucou 3 bis");// 3bis
        }
        Terminal.ecrireStringln("Coucou 4");        // 4
    }
}
class Stop extends RuntimeException {}
class Stop2 extends RuntimeException {}
```

A l'exécution, on obtient (en saisissant une valeur positive) :

```
Coucou 1
Coucou 3 bis
Coucou 4
```

4.5 Exceptions et méthodes

4.5.1 Exception non rattrapée dans le corps d'une méthode

Comme on l'a vu dans les exemples précédents, lorsqu'une exception est levée lors de l'exécution d'une méthode et qu'elle n'est pas rattrapée dans cette méthode, elle "continue son trajet" à partir de l'appel de la méthode. Même si la méthode est sensée renvoyer une valeur, elle ne le fait pas :

```
public class Arret {
    static int lance(int x) {
        if (x < 0) {
            throw new Stop();
        }
        return x;
    }

    public static void main(String [] args) {
        int y = 0;
        try {
            Terminal.ecrireStringln("Coucou 1");
            y = lance(-2);
            Terminal.ecrireStringln("Coucou 2");
        } catch (Stop e) {
            Terminal.ecrireStringln("Coucou 3");
        }
        Terminal.ecrireStringln("y vaut "+y);
    }
}

class Stop extends RuntimeException {}
```

A l'exécution on obtient :

```
Coucou 1
Coucou 3
y vaut 0
```

4.5.2 Déclaration `throws`

Lorsqu'une méthode lève une exception définie par extension de la classe `Exception` il est nécessaire de préciser au niveau de la déclaration de la méthode qu'elle peut potentiellement lever une exception de cette classe. Cette déclaration prend la forme **`throws`** `Exception1, Exception2, ...` et se place entre les arguments de la méthode et l'accolade ouvrant marquant le début du corps de la méthode. On notera que cette déclaration n'est pas obligatoire pour les exceptions de la catégorie `Error` ni pour celles de la catégorie `RuntimeException`.

4.6 Exemple résumé

On reprend l'exemple de la fonction factorielle :

```
class Factorielle {
    static int factorielle(int n) throws PasDefini { // (1)
        int res = 1;
        if (n<0){
            throw new PasDefini();                    // (2)
        }
        for(int i = 1; i <= n; i++) {
            res = res * i;
```

```
        }
        return res;
    }

    public static void main (String [] args) {
        int x;
        Terminal.ecrireString("Entrez un nombre (petit):");
        x = Terminal.lireInt();
        try {                                          // (3)
            Terminal.ecrireIntln(factorielle(x));
        } catch (PasDefini e) {                        // (3 bis)
            Terminal.ecrireStringln("La factorielle de "
                                    +x+" n'est pas définie !");
        }
    }
}

class PasDefini extends Exception {}                   // (4)
```

Dans ce programme, on définit une nouvelle classe d'exception `PasDefini` au point (4). Cette exception est levée par l'instruction `throw` au point (2) lorsque l'argument de la méthode est négatif. Dans ce cas l'exception n'est pas rattrapée dans le corps et comme elle n'est ni dans la catégorie `Error` ni dans la catégorie `RuntimeException`, on la déclare comme pouvant être levée par factorielle, en utilisant la déclaration `throws` au point (1). Si l'exception `PasDefini` est levée lors de l'appel à factorielle, elle est rattrapée au niveau de la construction `try catch` des points (3) (3 bis) et un message indiquant que la factorielle du nombre entré n'est pas définie est alors affiché. Voici deux exécutions du programme avec des valeurs différentes pour `x` (l'exception est levée puis rattrapée lors de la deuxième exécution) :

```
> java Factorielle
Entrez un nombre (petit):4
24
> java Factorielle
Entrez un nombre (petit):-3
La factorielle de -3 n'est pas définie !
```

Annexe : quelques exceptions prédéfinies

Voici quelques exceptions prédéfinies dans Java :

- `NullPointerException` : accès à un champ ou appel de méthode non statique sur un objet valant `null`. Utilisation de `length` ou accès à un case d'un tableau valant `null`.
- `ArrayIndexOutOfBoundsException` : accès à une case inexistante dans un tableau, création d'un tableau de taille négative.
- `StringIndexOutOfBoundsException` : accès au i^{eme} caractère d'un chaîne de caractères de taille inférieure à i.
- `NumberFormatException` : erreur lors de la conversion d'un chaîne de caractères en nombre.

La classe `Terminal` utilise également l'exception `TerminalException` pour signaler des erreurs.

Chapitre 5

Récursivité

5.1 La notion de récursivité

5.1.1 La décomposition en sous-problèmes

Le processus d'analyse 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.

Exemple : afficher un rapport avec les élèves d'une classe, groupés par la note obtenue à un examen, en ordre décroissant des notes et en ordre alphabétique pour chaque note.

Une décomposition possible de ce problème en sous-problèmes :
- introduction des données (noms des élèves et notes)
- tri des élèves en ordre décroissant des notes
- pour chaque note obtenue, en ordre décroissant, répéter
  - extraction des noms d'élèves qui ont obtenu cette note
  - calcul du nombre d'élèves qui ont obtenu cette note
  - tri de ces élèves en ordre alphabétique
  - affichage de la note, du nombre d'élèves qui ont cette note et des noms de ces élèves

Chacun de ces sous-problèmes pourra être décomposé à son tour en sous-sous-problèmes, etc. En programmation, la décomposition en sous-problèmes correspond au découpage d'un programme en sous-programmes ; a chaque sous-problème correspond un sous-programme.

La décomposition ci-dessus décrit *l'algorithme* de résolution du problème en utilisant l'appel aux sous-programmes qui traitent les sous-problèmes.

5.1.2 Décomposition récursive

Dans certains cas, le sous-problème est une illustration du problème initial, mais pour un cas "plus simple". Par conséquent, *la solution du problème s'exprime par rapport à elle-même* ! On appelle ce phénomène **récursivité**.

Exemple : calcul de la factorielle d'une valeur entière positive n (n ! = 1 * 2 * .... * n)

Mais, n ! = (1 * 2 * .... * (n-1)) * n, donc **n ! = (n-1) ! * n**.

Pour calculer la valeur de n !, il suffit donc de savoir calculer (n-1) ! et ensuite de multiplier cette valeur par n. Le sous-problème du calcul de (n-1) ! est le même que le problème initial, mais pour un cas "plus simple", car $n - 1 < n$.

Conclusion : le problème a été réduit à lui-même, mais pour un cas plus simple.

En programmation, le sous-programme qui traite le problème fait un appel à lui-même (!) pour traiter le cas plus simple, ce qui revient à un appel avec des paramètres différents (“plus simples”). On appelle cela *un appel récursif*.

La fonction `factorielle` aura alors la forme suivante :

```
int factorielle (int n){
  int sous_resultat = factorielle (n-1); //appel récursif
  int resultat = sous_resultat * n;
  return resultat;
}
```

Remarque très importante : un appel récursif peut produire lui-même un autre appel récursif, etc, ce qui peut mener à une suite infinie d'appels. En l'occurrence, la fonction ci-dessus *est incorrecte*. Il faut arrêter la suite d'appels au moment où le sous-problème peut être résolu directement. Dans la fonction précédente, il faut s'arrêter (ne pas faire d'appel récursif) si $n = 1$, car dans ce cas on connaît le résultat ($1 ! = 1$).

Conclusion : ***dans tout sous-programme récursif il faut une condition d'arrêt***.

La version correcte de la fonction `factorielle` est la suivante :

```
int factorielle (int n){
  int resultat;
  if(n==1) resultat = 1;
  else {
    int sous_resultat = factorielle (n-1); //appel récursif
    resultat = sous_resultat * n;
  }
  return resultat;
}
```

Remarque : la fonction factorielle peut être écrite d'une manière plus compacte, mais nous avons préféré cette version pour mieux illustrer les sections suivantes. Normalement, on n'a pas besoin des variables locales et on peut écrire directement :

```
int factorielle (int n){
  if(n==1) return 1;
  else return factorielle(n-1) * n;
}
```

Remarque : l'existence d'une condition d'arrêt ne signifie pas que l'appel récursif s'arrête grâce à celle-ci. Prenons l'exemple de l'appel `factorielle(-1)` : celà produit un appel à `factorielle(-2)`, qui produit un appel à `factorielle(-3)`, etc. La condition d'arrêt (n=1) n'est jamais atteinte et on obtient une suite infinie d'appels.

Conclusion : dans un sous-programme récursif, *il faut s'assurer que la condition d'arrêt est atteinte après un nombre fini d'appels*.

Remarque : la condition d'arrêt doit être choisie avec soin. Elle doit correspondre en principe au cas "le plus simple" qu'on veut traiter, sinon certains cas ne seront pas couverts par le sous-programme. Dans notre cas, la fonction `factorielle` ne sait pas traiter le cas n=0, qui est pourtant bien défini (0 ! = 1) et qui respecte la relation de décomposition récursive (1 ! = 0 ! * 1).

Le programme complet qui utilise la fonction `factorielle`, modifiée pour tenir compte des remarques ci-dessus, est le suivant :

```
public class TestFactorielle{
  static int factorielle (int n){
    int resultat;
    if(n<0) throw new MauvaisParametre();
    else if(n==0) resultat = 1;
    else {
      int sous_resultat = factorielle (n-1); //appel récursif
      resultat = sous_resultat * n;
    }
    return resultat;
  }

  public static void main(String[] args){
    Terminal.ecrireString("Entrez un entier positif : ");
    int x = Terminal.lireInt();
    Terminal.ecrireStringln(x + "! = " + factorielle(x));
  }
}

class MauvaisParametre extends Error{}
```

5.1.3 Récursivité directe et indirecte

Quand un sous-programme fait appel à lui même, comme dans le cas de la factorielle, on appelle cela *récursivité directe*. Parfois, il est possible qu'un sous-programme A fasse appel à un sous-programme B, qui lui même fait appel au sous-programme A. Donc l'appel qui part de A atteint de nouveau A après être passé par B. On appelle cela *récursivité indirecte* (en fait, la suite d'appels qui part de A peut passer par plusieurs autres sous-programmes avant d'arriver de nouveau à A !).

Exemple : un exemple simple de récursivité indirecte est la définition récursive des nombres pairs et impairs. Un nombre n positif est pair si n-1 est impair ; un nombre n positif est impair si n-1 est pair. Les conditions d'arrêt sont données par les valeurs n=0, qui est paire et n=1, qui est impaire.

Voici le programme complet qui traite ce problème :

```
public class TestPairImpair{
  static boolean pair(int n){
    if(n<0) throw new MauvaisParametre();
    else if(n==0) return true;
    else if(n==1) return false;
    else return impair(n-1);
  }
```

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

  public static void main(String[] args){
    Terminal.ecrireString("Entrez un entier positif : ");
    int x = Terminal.lireInt();
    if(pair(x)) Terminal.ecrireStringln("nombre pair");
    else Terminal.ecrireStringln("nombre impair");
  }
}

class MauvaisParametre extends Error{}
```

La récursivité indirecte est produite par les fonctions `pair` et `impair` : `pair` appelle `impair` et `impair` appelle à son tour `pair`.

Un autre exemple, qui combine les deux types de récursivité, est présenté ci-dessous :

Exemple : calculer les suites de valeurs données par les relations suivantes :

$x_0 = 1$; $x_n = 2*y_{n-1} + x_{n-1}$

$y_0 = -1$; $y_n = 2*x_{n-1} + y_{n-1}$

Les fonctions récursives qui calculent les valeurs des suites x et y sont présentées ci-dessous. On retrouve dans chaque fonction à la fois de la récursivité directe (`x` fait appel à `x` et `y` à `y`) et de la récursivité indirecte (`x` fait appel à `y`, qui fait appel à `x`, etc). L'exemple ne traite pas les situations de mauvaises valeurs de paramètres (çàd $n < 0$).

```
int x (int n){
  if(n==0) return 1;
  else return 2*y(n-1) + x(n-1);
}

int y (int n){
  if(n==0) return -1;
  else return 2*x(n-1) + y(n-1);
}
```

5.2 Evaluation d'un appel récursif

Comment est-il possible d'appeler un sous-programme pendant que celui-ci est en train de s'exécuter ? Comment se fait-il que les données gérées par ces différents appels du même sous-programme ne se "mélangent" pas ? Pour répondre à ces questions il faut d'abord comprendre le modèle de mémoire dans l'exécution des sous-programmes en Java.

5.2.1 Modèle de mémoire

Chaque sous-programme Java (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. Prenons pour exemple la fonction suivante :

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

La zone de mémoire occupée par cette fonction est illustrée dans la figure 5.1

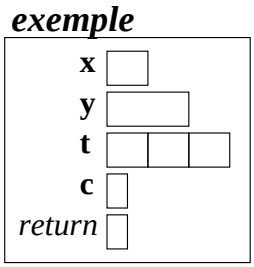


FIG. 5.1 – Zone de mémoire occupée par la fonction `exemple`

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**. Les zones de mémoire des sous-programmes sont empilées suivant l’ordre des appels et dépilées dès que le sous-programme se termine. Par conséquent, la zone de mémoire d’un sous-programme n’existe physiquement *que pendant que le sous-programme est en cours d’exécution.*

Supposons que le programme principal qui appelle la fonction exemple a la forme suivante :

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

Le contenu de la pile pendant l’appel de la fonction `exemple` est présenté dans la figure 5.2.

Avant l’appel, tout comme après la fin de l’exécution de la fonction `exemple`, la pile ne contient que la zone de `main`. La place libérée par `exemple` sera occupée par un éventuel appel ultérieur d’un autre sous-programme (peut-être même `exemple`, s’il est appelé plusieurs fois par `main`!).

Remarque : Il y a une séparation nette entre les variables locales des différents sous-programmes, car elles occupent des zones de mémoire distinctes (*ex*. les variables x, y et c).

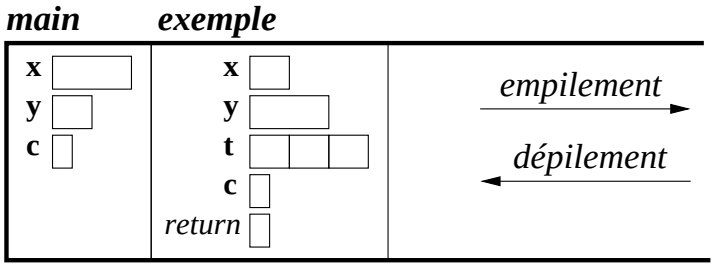


FIG. 5.2 – La pile pendant l'appel de fonction `exemple`

Conclusion : la pile contient à un moment donné les zones de mémoire de l'enchaînement de sous-programmes en cours d'exécution, qui part du programme principal.

5.2.2 Déroulement des appels récursifs

Dans le cas des programmes récursifs, la pile est remplie par l'appel d'un même sous-programme (qui s'appelle soi-même). Prenons le cas du calcul de `factorielle(2)` : le programme principal appelle `factorielle(2)`, qui appelle `factorielle(1)`, qui appelle `factorielle(0)`, qui peut calculer sa valeur de retour sans autre appel récursif. La valeur retournée par `factorielle(0)` permet à `factorielle(1)` de calculer son résultat, ce qui permet à `factorielle(2)` d'en calculer le sien et de le retourner au programme principal.

L'enchaînement des appels et des calculs est illustré dans la figure 5.3.

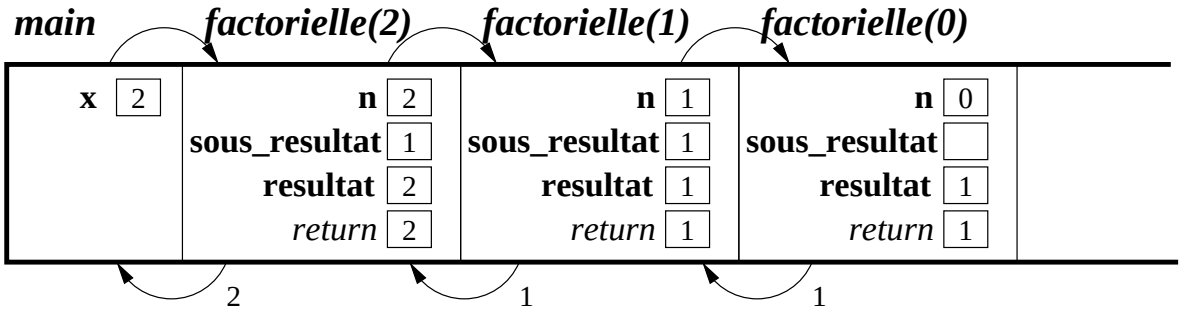


FIG. 5.3 – Enchaînement des appels récursifs pour `factorielle(2)`

Chaque instance de `factorielle` récupère le résultat de l'appel récursif suivant dans la variable *`sous_resultat`*, ce qui lui permet de calculer resultat et de le retourner à son appelant.

5.3 Comment concevoir un sous-programme récursif ?

Dans l'écriture des programmes récursifs on retrouve généralement les étapes suivantes :

1. *Trouver une décomposition récursive du problème*
   (a) Trouver l'élément de récursivité qui permet de définir les cas plus simples (*ex*. une valeur numérique qui décroît, une taille de données qui diminue).
   (b) Exprimer la solution dans le cas général en fonction de la solution pour le cas plus simple.

2. *Trouver la condition d'arrêt de récursivité et la solution dans ce cas*
   – Vérifier que la condition d'arrêt est atteinte après un nombre fini d'appels récursifs dans tous les cas
3. *Réunir les deux étapes précédentes dans un seul programme*

Exemple : calcul du nombre d'occurrences n d'un caractère donné c dans une chaîne de caractères donnée s.

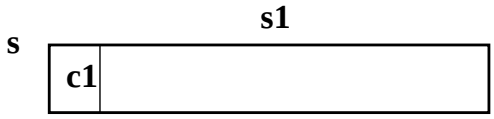


FIG. 5.4 – Décomposition récursive d'une chaîne `s` en un premier caractère `c1` et le reste de la chaîne `s1`

1. Décomposition récursive
   (a) Elément de récursivité : la chaîne, dont la taille diminue. Le cas "plus simple" est la chaîne sans son premier caractère (notons-la $s1$) - voir la figure 5.4.
   (b) Soit $n1$ le nombre d'occurrences de c dans $s1$ (la solution du problème plus simple).
   Soit $c1$ le premier caractère de la chaîne initiale s.
   Si $c1 = c$, alors le résultat est $n = n1 + 1$, sinon $n = n1$.
2. Condition d'arrêt : si s= "" (la chaîne vide) alors n = 0.
   La condition d'arrêt est toujours atteinte, car toute chaîne a une longueur positive et en diminuant de 1 la taille à chaque appel récursif on arrive forcément à la taille 0.
3. La fonction sera donc la suivante :

```
int nbOccurrences (char c, String s){
  if(s.length() == 0) return 0;  //condition d'arrêt: chaîne vide
  else{
    int sous_resultat = nbOccurrences(c, s.substring(1, s.length()-1));
    if(s.charAt(0) == c) return 1 + sous_resultat;
    else return sous_resultat;
  }
}
```

5.3.1 Sous-programmes récursifs qui ne retournent pas de résultat

Les exemples précédents de récursivité sont des *calculs récursifs*, représentés par des fonctions qui retournent le résultat de ce calcul. La décomposition récursive montre comment le résultat du calcul dans le cas "plus simple" sert à obtenir le résultat final.

La relation entre la solution du problème et la solution du cas "plus simple" est donc *une relation de calcul* (entre valeurs).

Toutefois, dans certains cas le problème ne consiste pas en un calcul, mais en une *action récursive* sur les données (affichage, modification de la valeur). Dans ce cas, l'action dans le cas "plus simple" représente une partie de l'action à réaliser dans le cas général. Il s'agit donc d'une *relation de composition entre actions*.

Exemple : affichage des éléments d'un tableau t en ordre inverse à celui du tableau.

Cet exemple permet d'illustrer aussi la méthode typique de décomposer récursivement un tableau Java. La différence entre un tableau et une chaîne de caractères en Java est qu'on ne dispose pas de fonctions d'extraction de sous-tableaux (comme la méthode `substring` pour les chaînes de caractères). On peut programmer soi-même une telle fonction, mais souvent on peut éviter cela, comme expliqué ci-dessous.

La méthode la plus simple est de considérer comme élément de récursivité non pas le tableau, mais *l'indice du dernier élément du tableau*. Ainsi, le cas "plus simple" sera non pas un sous-tableau, mais un indice de dernier élément plus petit - voir la figure 5.5.

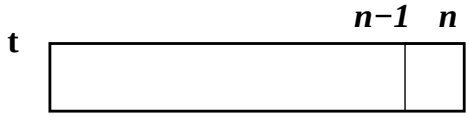


FIG. 5.5 – Décomposition récursive d'un tableau `t` en utlisant l'indice `n` du dernier élément

1. Décomposition récursive
   (a) Elément de récursivité : l'indice n du dernier élément du tableau t.
   Le cas "plus simple" est l'indice du dernier élément n-1, ce qui correspond au tableau t sans son dernier élément.
   (b) L'action pour le tableau t est décomposée de la manière suivante :
   1. Affichage de $t(n)$
   2. Affichage récursif pour t sans le dernier élément.
2. Condition d'arrêt : si n= 0 (un seul élément dans t) alors l'action est d'afficher cet élément ($t(0)$).
   La condition d'arrêt est toujours atteinte, car tout tableau a au moins un élément, donc $n >= 0$. En diminuant n de 1 à chaque appel récursif, on arrive forcément à la valeur 0.
3. La fonction sera donc la suivante :

```
void affichageInverse(int[] t, int n){
  if(n==0) Terminal.ecrireIntln(t(0)); //condition d'arrêt: un seul élément
  else{
    Terminal.ecrireIntln(t(n));
    affichageInverse(t, n-1);
  }
}
```

5.4 Récursivité et itération

Par l'appel répété d'un même sous-programme, la récursivité permet de réaliser des traitements répétitifs. Jusqu'ici, pour réaliser des répétitions nous avons utilisé *les boucles (itération)*. Suivant le type de problème, la solution s'exprime plus naturellement par récursivité ou par itération. Souvent il est aussi simple d'exprimer les deux types de solution.

Exemple : la solution itérative pour le calcul du nombre d'occurrences d'un caractère dans une chaîne est comparable en termes de simplicité avec la solution récursive.

```
int nbOccurrences (char c, String s){
  int accu = 0;
  for(int i=0; i<s.length(); i++)
    if(s.charAt(i)==c) accu++;
  return accu;
}
```

En principe tout programme récursif peut être écrit à l'aide de boucles (par itération), sans récursivité. Inversement, chaque type de boucle (while, do...while, for) peut être simulé par récursivité. Il s'agit en fait de deux manières de programmer différentes. Utiliser l'une ou l'autre est souvent une question de goût et de style de programmation - cependant chacune peut s'avérer mieux adaptée que l'autre à certaines classes de problèmes.

5.4.1 Utilisation de l'itération

Un avantage important de l'itération est *l'efficacité*. Un programme qui utilise des boucles décrit précisément chaque action à réaliser - ce style de programmation est appelé *impératif* ou *procédural*. Le code compilé d'un tel programme est une image assez fidèle des actions décrites par le programme, traduites en code machine. Les choses sont différentes pour un programme récursif.

Conclusion : si on recherche l'efficacité (une exécution rapide) et le programme peut être écrit sans trop de difficultés en style itératif, on préférera l'itération.

5.4.2 Utilisation de la récursivité

L'avantage de la récursivité est qu'elle se situe à un *niveau d'abstraction supérieur* par rapport à l'itération. Une solution récursive décrit comment calculer la solution à partir d'un cas plus simple - ce style de programmation est appelé *déclaratif*. Au lieu de préciser chaque action à réaliser, on décrit ce qu'on veut obtenir - c'est ensuite au système de réaliser les actions nécessaires pour obtenir le résultat demandé.

Souvent la solution récursive d'un problème est *plus intuitive* que celle itérative et le programme à écrire est *plus court* et *plus lisible*. Néanmoins, quelqu'un habitué aux boucles et au style impératif peut avoir des difficultés à utiliser la récursivité, car elle correspond à un type de raisonnement particulier.

Le style déclaratif produit souvent du code moins efficace, car entre le programme descriptif et le code compilé il y a un espace d'interprétation rempli par le système, difficile à optimiser dans tous les cas. Cependant, les langages déclaratifs offrent un confort nettement supérieur au programmeur (comparez SQL avec un langage de bases de données avec accès enregistrement par enregistrement).

Dans le cas précis de la récursivité, celle-ci est moins efficace que l'itération, car la répétition est réalisée par des appels successifs de fonctions, ce qui est plus lent que l'incrémentation d'un compteur de boucle.

Dans les langages (comme Java) qui offrent à la fois l'itération et la récursivité, on va préférer la récursivité surtout dans les situations où la solution itérative est difficile à obtenir, par exemple :

- si les structures de données manipulées sont récursives (*ex*. les arbres).
- si le raisonnement lui même est récursif.

5.4.3 Exemple de raisonnement récursif : les tours de Hanoi

Un exemple très connu de raisonnement récursif apparaît dans le problème des tours de Hanoi. Il s'agit de n disques de tailles différentes, trouées au centre, qui peuvent être empilés sur trois piliers. Au début,

tous les disques sont empilés sur le pilier de gauche, en ordre croissant de la taille, comme dans la figure suivante. Le but du jeu est de déplacer les disques un par un pour reconstituer la tour initiale sur le pilier de droite. Il y a deux règles :

1. on peut seulement déplacer un disque qui se trouve au sommet d'une pile (non couvert par un autre disque) ;
2. un disque ne doit jamais être placé sur un autre plus petit.

Le jeu est illustré dans la figure 5.6.

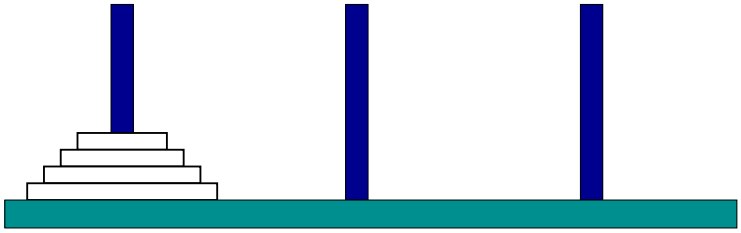


FIG. 5.6 – Les tours de Hanoi

La solution s'exprime très facilement par récursivité. On veut déplacer une tour de n disques du pilier de gauche vers le pilier de droite, en utilisant le pilier du milieu comme position intermédiaire.

1. Il faut d'abord déplacer vers le pilier du milieu la tour de n-1 disques du dessus du pilier de gauche (en utilisant le pilier de droite comme intermédiaire).
2. Il reste sur le pilier de gauche seul le grand disque à la base. On déplace ce disque sur le pilier de droite.
3. Enfin on déplace la tour de n-1 disques du pilier du milieu vers le pilier de droite, au-dessus du grand disque déjà placé (en utilisant le pilier de gauche comme intermédiaire).

La procédure récursive `deplaceTour` réalise cet algorithme et utilise la procédure `deplaceUnDisque` pour afficher le déplacement de chaque disque individuel.

Remarque : Il est difficile d'écrire un algorithme itératif pour ce problème. Essayez comme exercice d'écrire le même programme avec des boucles !

```
public class Hanoi{
  static void deplaceUnDisque (String source, String dest){
    Terminal.ecrireStringln(source + " -> " + dest);
  }

  static void deplaceTour(int taille, String source, String dest, String interm){
    if(taille==1) deplaceUnDisque(source, dest);
    else{
      deplaceTour(taille-1, source, interm, dest);
      deplaceUnDisque(source, dest);
      deplaceTour(taille-1, interm, dest, source);
    }
  }

  public static void main(String[] args){
    Terminal.ecrireString("Combien de disques ? ");
```

```
    int n = Terminal.lireInt();
    deplaceTour(n, "gauche", "droite", "milieu");
  }
}
```

L'exécution de ce programme produit le résultat suivant :

```
% java Hanoi
Combien de disques ? 3
gauche - droite
gauche - milieu
droite - milieu
gauche - droite
milieu - gauche
milieu - droite
gauche - droite
```

Chapitre 6

Listes chaînées

6.1 La notion de liste

Une liste est une structure de données qui permet de stocker une séquence d'objets d'un même type. En cela, les listes ressemblent aux *tableaux*. La séquence d'entiers 3, 7, 2, 4 peut être représentée à la fois sous forme de tableau ou de liste. La notation [3, 7, 2, 4] représentera la liste qui contient cette séquence.

Il y a cependant des différences fondamentales entre listes et tableaux :

- Dans un tableau on a accès immédiat à n'importe quel élément par son indice (accès dit *aléatoire*), tandis que dans une liste chaînée on a accès aux éléments un après l'autre, à partir du premier élément (accès dit *séquentiel*).
- Un tableau a une taille fixe, tandis qu'une liste peut augmenter en taille indéfiniment (on peut toujours rajouter un élément à une liste).

Définition (récursive) :

En considérant que la liste la plus simple est *la liste vide* (notée [ ]), qui ne contient aucun élément, on peut donner une définition récursive aux listes chaînées d'éléments de type T :

- la liste vide [ ] est une liste ;
- si e est un élément de type T et l est une liste d'éléments de type T, alors le couple (e, l) est aussi une liste, qui a comme premier élément e et dont le reste des éléments (à partir du second) forment la liste l.

Cette définition est *récursive*, car une liste est définie en fonction d'une autre liste. Une telle définition récursive est correcte, car une liste est définie en fonction d'une liste plus courte, qui contient un élément de moins. Cette définition permet de construire n'importe quelle liste en partant de la liste vide.

Conclusion : la liste chaînée est une structure de données récursive.

La validité de cette définition récursive peut être discutée d'un point de vue différent. Elle peut être exprimée sous la forme suivante : quelque soit la liste l considérée,

- soit l est la liste vide [ ],
- soit l peut être décomposée en un premier élément e et un reste r de la liste, l=(e, r).

Cette définition donne *une décomposition récursive* des listes. La condition générale d'arrêt de récursivité est pour la liste vide. Cette décomposition est valide, car la liste est réduite à chaque pas à une liste plus courte d'une unité. Cela garantit que la décomposition mène toujours à une liste de longueur 0, la liste vide (condition d'arrêt).

6.2 Représentation des listes chaînées en Java

Il y a plusieurs façons de représenter les listes chaînées en Java. L'idée de base est d'utiliser *un enchaînement de cellules* :

- chaque cellule contient un élément de la liste ;
- chaque cellule contient une référence vers la cellule suivante, sauf la dernière cellule de la liste (qui contient une référence nulle) ;
- la liste donne accès à la première cellule, le reste de la liste est accessible en passant de cellule en cellule, suivant leur enchaînement.

La figure 6.1 illustre cette représentation pour la liste exemple ci-dessus [3, 7, 2, 4].

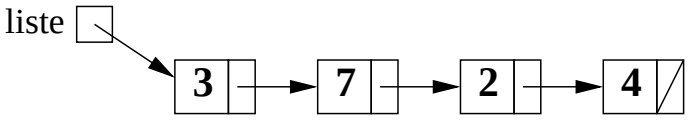


FIG. 6.1 – Représentation par enchaînement de cellules de la liste [3, 7, 2, 4]

En Java, les références ne sont pas définies explicitement, mais implicitement à travers les objets : un objet est représenté par une référence vers la zone de mémoire qui contient ses variables d'instance.

Comme une liste est définie par une référence vers une cellule (la première de la liste), *la méthode la plus simple est de représenter une liste par sa première cellule*. La classe suivante définit une liste d'entiers.

```
class Liste{
  int element;
  Liste suivant;

  Liste(int premier, Liste reste){
    element = premier;
    suivant = reste;
  }
}
```

Le constructeur de la classe `Liste` crée une liste comme une première cellule qui contient le premier élément de la liste et une référence vers le reste de la liste.

Remarque : Cette solution a l'inconvénient de ne représenter que *des listes avec au moins une cellule*. La liste vide, qui selon la définition est également une liste, ne peut pas être un objet de la classe `Liste`. Il est néanmoins possible de représenter la liste vide *par une référence nulle*. Cette représentation imparfaite des listes a l'avantage de la simplicité et nous l'utiliserons dans le reste de ce cours.

6.3 Opérations sur les listes chaînées

Nous étudierons les opérations les plus importantes sur les listes chaînées, qui seront représentées comme des méthodes de la classe `Liste`.

Remarque : Puisque la classe `Liste` ne peut pas représenter la liste vide, des opérations de base comme le test de liste vide ou l'action de vider une liste ne peuvent pas être des méthodes de la classe `Liste`. Ces opérations doivent être effectuées à l'extérieur de la classe `Liste`, par le programme qui utilise les listes.

Conformément à leur définition, les listes sont des structures *récursives*. Par conséquent, les opérations sur les listes s'expriment naturellement par des algorithmes récursifs. En même temps, les listes sont des structures linéaires, parfaitement adaptées à un parcours *itératif*. Nous donnerons aussi la variante itérative des opérations sur listes, à titre comparatif.

La représentation de la classe `Liste` ci-dessous montre sous forme de méthodes les opérations sur listes que nous discuterons.

```
class Liste{
  int element;
  Liste suivant;

  public Liste(int premier, Liste reste){
    element = premier;
    suivant = reste;
  }

  public int premier(){...}
  public Liste reste(){...}
  public void modifiePremier(int elem){...}
  public void modifieReste(Liste reste){...}
  public int longueur(){...}
  public boolean contient(int elem){...}
  public Liste insertionDebut(int elem){...}
  public void concatenation(Liste liste){...}
  public Liste suppressionPremier(int elem){...}
}
```

6.3.1 Opérations sans parcours de liste

Pour ces opérations il n'y a pas de variantes itératives et récursives, l'action est réalisée directement sur la tête de la liste.

Obtenir le premier élément et le reste de la liste

Ces opérations sont réalisées par les méthodes `premier` (qui retourne le premier élément de la liste), respectivement `reste` (qui retourne le reste de la liste).

En fait, ces méthodes ne sont pas nécessaires, au sens strict du terme, puisque dans la classe `Liste` on a accès direct aux variables d'instance *element* et *suivant*.

Nous verrons plus tard que l'accès direct aux variables d'instance est déconseillé dans la programmation orientée-objet. A la place, on préferre "cacher" les variables d'instance et donner accès à leur valeurs à travers des méthodes. Ceci fera l'objet d'un prochain cours.

Aussi, ces méthodes correspondent aux éléments de la décomposition récursive d'une liste et sont bien adaptées à l'écriture d'algorithmes récursifs sur les listes.

```
public int premier(){
  return element;
}

public Liste reste(){
```

```
  return suivant;
}
```

Remarque : Il ne faut pas oublier que `premier` et `reste` sont des méthodes de la classe `Liste`, donc une instruction comme `return element` signifie `return this.element`. Une liste est représentée par sa première cellule, donc `return this.element` retourne l'élément de cette première cellule.

Modifier le premier élément et le reste de la liste

Ces opérations sont réalisées par les méthodes `modifiePremier` (qui modifie le premier élément de la liste), respectivement `modifieReste` (qui modifie le reste de la liste).

Ces méthodes permettent donc de *modifier* les composants `element` et `suivant` de la première cellule de la liste. Elles sont complémentaires aux méthodes `premier` et `reste`, qui permettent de *consulter* ces mêmes composants.

Leur présence dans la classe `Liste` correspond au même principe que celui évoqué pour `premier` et `reste` : remplacer l'accès direct aux variables d'instance par des appels de méthodes.

```
public void modifiePremier(int elem){
  element = elem;
}

public void modifieReste(Liste reste){
  suivant = reste;
}
```

Insérer un élément en tête de liste

L'insertion en tête de liste est réalisée par la méthode `insertionDebut`. C'est une opération simple, car elle nécessite juste un accès à la tête de la liste initiale. On crée une nouvelle cellule, à laquelle on enchaîne l'ancienne liste.

La méthode `insertionDebut` retourne une nouvelle liste, car la liste initiale est modifiée par l'insertion (la première cellule de la nouvelle liste est différente).

```
public Liste insertionDebut(int elem){
  return new Liste(elem, this);
}
```

6.3.2 Opérations avec parcours de liste - variante itérative

Ces opérations nécessitent un parcours complet ou partiel de la liste. Dans la variante itérative, le parcours est réalisé à l'aide d'une référence qui part de la première cellule de la liste et suit l'enchaînement des cellules.

La figure 6.2 illustre le principe du parcours itératif d'une liste à l'aide d'une référence *ref*.

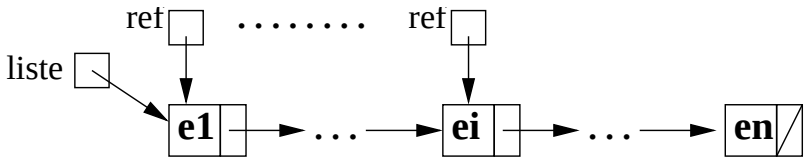


FIG. 6.2 – Parcours itératif d'une liste à l'aide d'une référence

Calculer la longueur d'une liste

Cette opération, réalisée par la méthode `longueur`, nécessite un parcours complet de la liste pour compter le nombre de cellules trouvées.

```
public int longueur(){
  int compteur = 0;
  Liste ref=this;
  while(ref != null){
    compteur++;
    ref=ref.reste();   //ou ref=ref.suivant
  }
  return compteur;
}
```

Remarque : La référence *ref* est positionnée au début sur la première cellule de la liste et avance jusqu'à ce qu'elle devient nulle à la fin de la liste. Chaque fois qu'elle pointe vers une nouvelle cellule, le compteur est incrémenté. Remarquez que l'avancement dans la liste se fait en utilisant la méthode `reste` de la cellule courante, ou sinon l'accès direct à la variable d'instance `suivant`.

Vérifier l'appartenance d'un élément à une liste

Cette opération, réalisée par la méthode `contient`, nécessite un parcours partiel de la liste pour chercher l'élément en question. Si l'élément est retrouvé, le parcours s'arrête à ce moment-là, sinon il continue jusqu'à la fin de la liste.

```
public boolean contient(int elem){
  Liste ref=this;
  while(ref != null){
    if(ref.premier() == elem)    //ou ref.element==elem
      return true;               //l'élément a été retrouvé
    else ref=ref.reste();        //ou ref=ref.suivant
  }
  return false;                  //l'élément n'a pas été retrouvé
}
```

Remarque : L'arrêt du parcours en cas de succès se fait en retournant directement `true` pendant l'exécution de la boucle, ce qui termine à la fois la boucle et la fonction. Remarquez que le test de l'élément courant pointé par la référence utilise la méthode `premier` de la cellule courante, ou sinon l'accès direct à la variable d'instance `element`.

Concaténation de deux listes

Cette opération, réalisée par la méthode `concatenation`, rajoute à la fin de la liste courante toutes les cellules de la liste passée en paramètre. Elle nécessite un parcours complet de la liste courante, afin de trouver sa dernière cellule.

La liste obtenue par concaténation a toujours la même première cellule que la liste initiale. On peut dire donc que la liste initiale a été modifiée, en lui rajoutant par concaténation une autre liste.

```
public void concatenation(Liste liste){
  Liste ref=this;
  while(ref.reste() != null){     //ou ref.suivant!=null
    ref=ref.reste();              //ou ref=ref.suivant
  }
  //ref pointe maintenant sur la dernière cellule de la liste
  ref.modifieReste(liste);       //ou ref.suivant=liste
}
```

Remarque : La condition de la boucle *while* change ici, car on veut s'arrêter sur la dernière cellule et non pas la dépasser comme dans les méthodes précédentes. Remarquez que l'enchaînement des deux listes se fait en modifiant la variable d'instance `suivant` de la dernière cellule à l'aide de la méthode `modifieReste`, ou sinon par modification directe de celle-ci.

Suppression de la première occurrence d'un élément

Cette opération, réalisée par la méthode `suppressionPremier`, élimine de la liste la première apparition de l'élément donné en paramètre (s'il existe). La méthode retourne une liste, car la liste obtenue par suppression peut avoir une autre première cellule que la liste initiale. Ceci arrive quand la cellule éliminée est exactement la première de la liste.

La suppression d'une cellule de la liste se fait de la manière suivante (voir la figure 6.3) :

- si la cellule est la première de la liste, la nouvelle liste sera celle qui commence avec la seconde cellule.
- sinon la cellule a un prédécesseur ; pour éliminer la cellule il faut la "court-circuiter", en modifiant la variable `suivant` du prédécesseur pour qu'elle pointe vers la même chose que la variable `suivant` de la cellule.

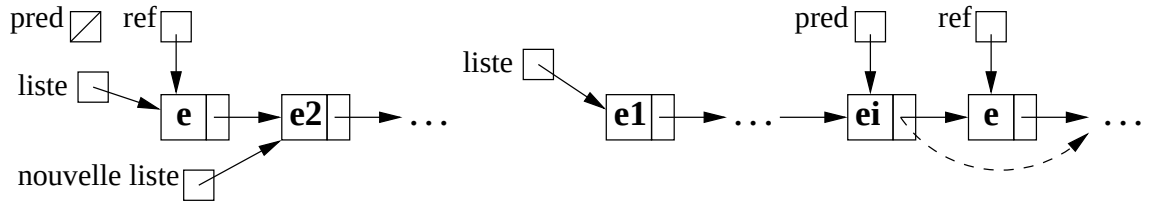


FIG. 6.3 – Suppression d'une cellule de la liste

Le parcours de la liste à la recherche de l'élément donné a une particularité : si la référence s'arrête sur la cellule qui contient l'élément, la suppression n'est plus possible, car on n'a plus accès au prédécesseur (dans les listes chaînées on ne peut plus revenir en arrière).

La méthode utilisée est alors de parcourir la liste avec *deux références* :

- une (*ref*) qui cherche la cellule à éliminer ;

– une autre (*pred*) qui se trouve toujours sur le prédécesseur de *ref*.

```
public Liste suppressionPremier(int elem){
  Liste ref=this;    //référence qui cherche elem
  Liste pred=null;   //prédécesseur, initialisé à null
  while(ref != null && ref.premier() != elem){  //recherche elem
    pred = ref;          //avance pred
    ref = ref.reste();   //avance ref
  }
  if(ref != null){           //elem trouvé dans la cellule ref
    if(pred == null)         //première cellule de la liste
      return ref.reste();    //retourne le reste de la liste
    else{                    //milieu de la liste
      pred.modifieReste(ref.reste()); //modifie le suivant du prédécesseur
      return this;           //première cellule non modifiée
    }
  }
  else return this;          //élément non trouvé
}
```

6.3.3 Opérations avec parcours de liste - variante récursive

Ces opérations sont basées sur une décomposition récursive de la liste en *un premier élément* et *le reste de la liste* (voir la figure 6.4). Le cas le plus simple (condition d'arrêt) est la liste avec une seule cellule.

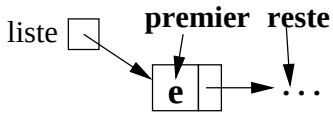


FIG. 6.4 – Décomposition récursive d'une liste

Calcul récursif de la longueur d'une liste

La formule récursive est :

```
liste.longueur() = 1                    si reste==null
                   1 + reste.longueur() si reste!=null
```

La fonction récursive `longueur` s'écrit alors très facilement, comme suit :

```
public int longueur(){
  Liste resteListe = reste();  //ou resteListe=suivant;
  if(resteListe == null)       //condition d'arrêt
    return 1;
  else return 1 + resteListe.longueur();
}
```

Vérification récursive de l'appartenance à une liste

La formule récursive est :

```
liste.contient(e) = true              si premier==e
                    false             si premier!=e et reste==null
                    reste.contient(e) si premier!=e et reste!=null
```

Remarquez que dans ce cas il y a deux conditions d'arrêt : quand on trouve l'élément recherché ou quand la liste n'a plus de suite (reste est vide). La fonction récursive `contient` s'écrit alors comme suit :

```
public boolean contient(int elem){
  if(elem == premier()) return true; //ou elem==element
  Liste resteListe = reste();        //ou resteListe=suivant;
  if(resteListe == null)             //condition d'arrêt
    return false;
  else return resteListe.contient(elem);
}
```

Concaténation récursive de deux listes

L'action récursive est :

```
liste.concatenation(l): reste=l                si reste==null
                        reste.concatenation(l) si reste!=null
```

Si la liste a une seule cellule (pas de cellule suivante, reste est vide), alors il faut modifier la référence vers la cellule suivante pour enchaîner la liste paramètre. Sinon, il suffit de concaténer le reste avec la liste paramètre.

```
public void concatenation(Liste liste){
  Liste resteListe = reste();  //ou resteListe=suivant;
  if(resteListe == null)       //condition d'arrêt
    modifieReste(liste);       //ou suivant=liste
  else resteListe.concatenation(liste);
}
```

Suppression récursive de la première occurrence d'un élément

La formule récursive est :

```
liste.suppressionPremier(e) =
       reste                  si premier==e
       Liste(premier, null)   si premier!=e et reste==null
       Liste(premier, reste.suppressionPremier(e)) si premier!=e et
                                                   reste != null
```

Si le premier élément de la liste est celui recherché, le résultat sera le reste de la liste. Sinon, il faut supprimer récursivement l'élément recherché du reste. Mais le résultat de cette suppression est une liste qui provient de *reste*, donc elle n'a plus le premier élément de la liste initiale. Il faut donc le rajouter en tête de liste.

```
public Liste suppressionPremier(int elem){
  Liste resteListe = reste();  //ou resteListe=suivant;
  if(elem == premier())        //ou elem==element
    return resteListe;
  else if(resteListe == null) return new Liste(premier(), null);
  else return new Liste(premier(), resteListe.suppressionPremier(elem));
}
```

Remarque : L'inconvénient de cette méthode est qu'on crée des copies de toutes les cellules qui ne contiennent pas l'élément recherché. Pour éviter cela, la définition doit mélanger calcul et effets de bord, comme suit :

```
liste.suppressionPremier(e) =
             reste   si premier==e
             liste   si premier!=e et reste==null
             liste après l'action reste=reste.suppressionPremier(e)
                     si premier!=e et reste!=null
```

```
public Liste suppressionPremier(int elem){
  Liste resteListe = reste();  //ou resteListe=suivant;
  if(elem == premier())        //ou elem==element
    return resteListe;
  else if(resteListe == null) return this;
  else{
    modifieReste(resteListe.suppressionPremier(elem));
    return this;
  }
}
```

6.4 Listes triées

Nous nous intéressons maintenant aux listes chaînées dans lesquelles les éléments respectent un ordre croissant. Dans ce cas, les méthodes de recherche, d'insertion et de suppression sont différentes. L'ordre des éléments permet d'arrêter plus tôt la recherche d'un élément qui n'existe pas dans la liste. Aussi, l'insertion ne se fait plus en début de liste, mais à la place qui préserve l'ordre des éléments.

Nous utiliserons une classe `ListeTriee` qui est très similaire à la classe `Liste`, mais dans laquelle on s'intéresse uniquement aux méthodes de recherche (`contientTriee`), d'insertion (`insertionTriee`) et de suppression (`suppressionTriee`). Les autres méthodes sont identiques à celles de la classe `Liste`.

```
class ListeTriee{
  int element;
  ListeTriee suivant;

  public ListeTriee(int premier, ListeTriee reste){
    element = premier;
    suivant = reste;
  }
```

```
  public int premier(){...}
  public ListeTriee reste(){...}
  public void modifiePremier(int elem){...}
  public void modifieReste(ListeTriee reste){...}

  public boolean contientTriee(int elem){...}
  public ListeTriee insertionTriee(int elem){...}
  public ListeTriee suppressionTriee(int elem){...}
}
```

6.4.1 Recherche dans une liste triée

La différence avec la méthode `contient` de `Liste` est que dans le parcours de la liste on peut s'arrêter dès que la cellule courante contient un élément plus grand que celui recherché. Comme tous les éléments suivants vont en ordre croissant, ils seront également plus grands que l'élément recherché.

Variante iterative :

```
public boolean contientTriee(int elem){
  ListeTriee ref=this;
  while(ref != null && ref.premier() <= elem){
    if(ref.premier() == elem)
      return true;
    else ref=ref.reste();
  }
  return false;
}
```

Variante récursive :

```
public boolean contientTriee(int elem){
  if(elem == premier()) return true;
  if(elem < premier()) return false;
  ListeTriee resteListe = reste();
  if(resteListe == null)
    return false;
  else return resteListe.contientTriee(elem);
}
```

6.4.2 Insertion dans une liste triée

L'insertion d'un élément doit se faire *à la bonne place* dans la liste. La bonne place est *juste avant la première cellule qui contient un élément plus grand que celui à insérer*.

Variante iterative :

La figure 7.3 montre les deux cas d'insertion :
- en début de liste.
  Elle est similaire alors à l'opération `insertionDebut` de la classe `Liste`

– en milieu de liste.
  La recheche de la position d'insertion se fait alors avec une méthode similaire à celle utilisée pour la méthode `suppressionPremier` de la classe `Liste`, avec deux références. Le prédécesseur doit être modifié pour pointer vers la nouvelle cellule, tandis que celle-ci doit pointer vers la cellule courante (*ref*).

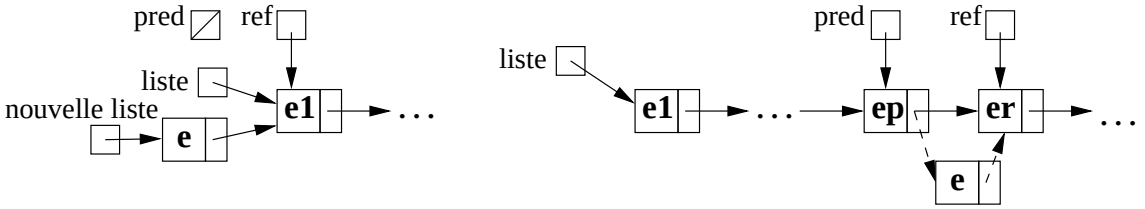


FIG. 6.5 – Insertion dans une liste triée

Dans la figure 7.3, à droite, la référence *ref* peut ne pas pointer vers une cellule. Ceci arrive quand l'élément à insérer est plus grand que tous les éléments de la liste. Dans ce cas, l'insertion se fait à la fin de la liste, donc au moment de l'arrêt de la recherche, *ref* est nulle et *pred* est sur la dernière cellule de la liste. Ce cas est identique à celui présenté dans la figure 7.3, à la variable `suivant` de la nouvelle cellule on donne tout simplement la valeur de *ref*.

```
public ListeTriee insertionTriee(int elem){
  ListeTriee ref=this;    //référence qui cherche elem
  ListeTriee pred=null;   //prédécesseur, initialisé à null
  while(ref != null && ref.premier() < elem){  //recherche position
    pred = ref;           //avance pred
    ref = ref.reste();    //avance ref
  }
  if(pred == null)        //insertion au début
    return new ListeTriee(elem, this);
  else{                   //insertion au milieu de la liste
    ListeTriee nouveau = new ListeTriee(elem, ref);
    pred.modifieReste(nouveau);    //modifie le suivant du prédécesseur
    return this;          //première cellule non modifiée
  }
}
```

Variante récursive :

Nous utilisons le même schéma de décomposition récursive de la liste, en un premier élément (*premier*) et un reste de la liste (*reste*). L'insertion récursive suit le schéma suivant :

```
liste.insertionTriee(e) =
    ListeTriee(e, liste)                        si e<=premier
    liste après reste=ListeTriee(e, null)       si e>premier et reste==null
    liste après reste=reste.insertionTriee(e)   si e>premier et reste!=null
```

Si l'élément à insérer est plus petit ou égal à *premier*, l'insertion se fera en tête de liste. Sinon, l'insertion se fera récursivement dans le reste. Si *reste* est vide, alors *reste* sera remplacé par la nouvelle cellule à insérer. Si *reste* n'est pas vide, l'insertion se fait récursivement dans *reste* et le résultat remplace l'ancien *reste*.

```
public ListeTriee insertionTriee(int elem){
  if(elem <= premier())
    return new ListeTriee(elem, this);
  else{
    ListeTriee resteListe = reste();
    if(resteListe == null)
      modifieReste(new ListeTriee(elem, null));
    else
      modifieReste(resteListe.insertionTriee(elem));
    return this;
  }
}
```

6.4.3 Suppression dans une liste triée

La suppression de la première occurrence d'un élément dans une liste triée est assez similaire à la suppression dans une liste non triée. La seule chose qui change est la recherche de la cellule à supprimer, qui bénéficie du tri des valeurs. Aussi, si plusieurs occurrences de l'élément existent, elles sont forcément l'une à la suite de l'autre à cause du tri. Il serait donc plus simple d'éliminer *toutes* les occurrences de l'élément que dans le cas des listes non triées. Nous nous limiterons cependant ici à la suppression de la première occurrence seulement.

Variante iterative :

Par rapport à la méthode `suppressionPremier` de la classe `Liste`, ici le parcours de la liste à la recherche de l'élément ne se fait que tant que *ref* pointe une valeur plus petite que celui-ci. Une fois le parcours arrêté, la seule différence est qu'il faut vérifier que *ref* pointe vraiment la valeur recherchée.

```
public Liste suppressionTriee(int elem){
  Liste ref=this;    //référence qui cherche elem
  Liste pred=null;   //prédécesseur, initialisé à null
  while(ref != null && ref.premier() < elem){   //recherche elem
    pred = ref;          //avance pred
    ref = ref.reste();   //avance ref
  }
  if(ref != null && ref.premier() == elem){ //elem trouvé dans la cellule ref
    if(pred == null)        //première cellule de la liste
      return ref.reste();   //retourne le reste de la liste
    else{                   //milieu de la liste
      pred.modifieReste(ref.reste()); //modifie le suivant du prédécesseur
      return this;          //première cellule non modifiée
    }
  }
  else return this;         //élément non trouvé
}
```

Variante récursive :

Par rapport à la variante récursive de la méthode `suppressionPremier` de la classe `Liste`, une nouvelle condition d'arrêt est rajoutée :

quand l'élément à éliminer est plus petit que *premier*, il n'existe sûrement pas dans la liste et celle-ci est retournée non modifiée.

```
public Liste suppressionTriee(int elem){
  Liste resteListe = reste();
  if(elem == premier())
    return resteListe;
  else if(elem < premier()) return this;  //nouvelle condition d'arrêt
  else if(resteListe == null) return this;
  else{
    modifieReste(resteListe.suppressionTriee(elem));
    return this;
  }
}
```

6.5 Un exemple de programme qui utilise les listes

Considérons un exemple simple de programme qui manipule des listes chaînées en utilisant la classe `Liste`.

Le programme lit une suite de nombres entiers terminée par la valeur 0 et construit une liste avec ces entiers (sauf le 0 final). La liste doit garder les éléments dans l'ordre dans lequel ils sont introduits. Egalement, une valeur ne doit être stockée qu'une seule fois dans la liste.

Le programme lit ensuite une autre suite de valeurs, également terminée par un 0, avec lesquelles il construit une autre liste, sans se soucier de l'ordre des éléments. Les éléments de cette seconde liste devront être éliminés de la liste initiale.

A la fin, le programme affiche ce qui reste de la première liste.

Remarque : Pour garder les éléments dans l'ordre d'introduction, l'insertion d'un nouvel élément doit se faire *en fin de liste*. La méthode `insertionDebut` ne convient donc pas. La solution est d'utiliser la méthode `concatenation`.

Remarque : Pour qu'un élément soit stocké une seule fois dans la liste, il faut d'abord vérifier s'il n'y est pas déjà, à l'aide de la méthode `contient`.

Voici le programme qui réalise ces actions :

```
public class ExempleListes{
  public static void main(String[] args){
    //1. Création première liste
    Terminal.ecrireStringln("Entrez les valeurs terminées par un 0 :");
    Liste liste = null;  //liste à construire
    do{
      int val = Terminal.lireInt();
      if(val==0) break;
      if(liste==null)
        liste = new Liste(val, null);
      else if(! liste.contient(val))
        liste.concatenation(new Liste(val, null));
    } while(true);
```

```
    //2. Création seconde liste
    Terminal.ecrireStringln("Valeurs à éliminer (terminées par un 0) :");
    Liste liste2 = null;
    do{
      int val = Terminal.lireInt();
      if(val==0) break;
      if(liste2==null)
        liste2 = new Liste(val, null);
      else liste2 = liste2.insertionDebut(val);
    } while(true);

    //3. Elimination de la première liste
    for(Liste ref = liste2; ref != null; ref = ref.reste()){
      if(liste==null) break;  //plus rien à éliminer
      liste = liste.suppressionPremier(ref.premier());
    }

    //4. Affichage liste restante
    Terminal.ecrireStringln("Valeurs restantes :");
    for(Liste ref = liste; ref != null; ref = ref.reste())
      Terminal.ecrireString(" " + ref.premier());
    Terminal.sautDeLigne();
  }
}
```

Chapitre 7

Algorithmique

7.1 Problématique

7.1.1 Algorithmes et structures de données

Les programmes manipulent des données (généralement des collections de données) et visent en principe un traitement particulier ou un calcul sur ces données. Dans l'écriture du programme, deux éléments sont essentiels :

- *les structures de données*, utilisées pour représenter les collections de valeurs à traiter ;
- *les algorithmes* de traitement.

Niklaus Wirth, le créateur du langage Pascal a énoncé cette fameuse équation :

Algorithmes + Structures de données = Programmes

Les structures de données sont très variées, nous avons déjà introduit dans ce cours deux des plus utilisées : les tableaux et les listes chaînées. D'autres exemples de structures de base sont les arbres de différents types, les tables de hachage, etc. Il n'y a pas de structure meilleure que d'autres dans l'absolu, chacune a ses avantages et ses inconvénients pour des types différents de problèmes. Le choix d'une structure de données appropriée pour un problème donné est un élément très important dans la conception du programme.

Les algorithmes décrivent l'enchaînement d'opérations à réaliser sur les structures de données afin de réaliser le calcul ou le traitement souhaité. *Un algorithme est donc intimement lié aux structures de données qu'il manipule*.

L'algorithmique étudie essentiellement les algorithmes de base sur les principales structures de données, ceux qui sont les plus utilisés en pratique. Comme les structures de données représentent généralement des collections de valeurs, les opérations les plus courantes sont :

- la recherche d'une valeur dans la collection,
- l'insertion d'une valeur dans la collection,
- la suppression d'une valeur dans la collection,
- le tri des valeurs de la collection,
- etc.

La description d'un algorithme n'est pas forcément liée à un langage de programmation. Pour écrire un algorithme on a seulement besoin d'utiliser les opérations de base sur les structures de données. On emploie souvent un langage général (pseudo-code) qui utilise des affectations, des conditionnelles, des boucles, etc. Il est ensuite assez facile à adapter un tel algorithme à un langage de programmation donné.

7.1.2 Un exemple d'algorithme

Considérons un algorithme simple : la recherche d'une valeur dans un tableau. L'algorithme peut être décrit comme suit :

Algorithme *recherche*
Entrées : tableau T, valeur v
Sorties : boolean qui indique si v se trouve dans T ou non
début
```
pour chaque indice dans T répéter
   si T[indice] == v alors retourner vrai
fin répéter
retourner faux
```
fin *recherche*

L'algorithme ci-dessus a l'avantage d'être valable pour des tableaux d'éléments de n'importe quel type pour lequel on sait tester l'égalité de deux valeurs. On peut facilement transcrire cet algorithme en Java, ADA, C, etc. Voici par exemple la recherche dans un tableau d'entiers, en Java :

```
int recherche(int[] T, int v){
  for(int i=0; i<T.length; i++){
    if(T[i]==v) return true;
  }
  return false;
}
```

7.1.3 Complexité des algorithmes

Pour un problème donné il existe normalement plusieurs algorithmes. Il est donc important de pouvoir *comparer* de tels algorithmes. Il y a deux critères principaux utilisés pour la comparaison :

- la quantité de mémoire utilisée par l'algorithme ;
- le temps d'exécution.

Le second critère est de loin le plus utilisé, les situations où la différence des quantités de mémoire est importante sont plus rares. Mais le temps d'exécution d'un algorithme dépend de beaucoup de paramètres : les données, l'environnement d'exécution, etc. Alors, la méthode d'évaluation du temps d'exécution est basée sur les éléments suivants :

- on identifie les *opérations importantes* pour le temps d'exécution dans l'algorithme
- on calcule le nombre d'opérations importantes réalisées par l'algorithme, *en fonction de la quantité de données* traitée par celui-ci.

La complexité de l'algorithme est donnée donc par le nombre d'opérations importantes réalisées par l'algorithme en fonction de la taille des données. Prenons l'exemple de l'algorithme précédent. L'opération importante est clairement *la comparaison* de deux valeurs.

En notant n la taille du tableau, combien de comparaisons réalise l'algorithme ? La réponse est : ... ça dépend. Généralement, on évalue la complexité dans trois cas :

- *dans le cas le plus favorable* : quand la valeur recherchée est la première dans le tableau, une seule comparaison est nécessaire.
- *dans le cas le plus défavorable* : quand la valeur recherchée est la dernière dans le tableau ou ne se trouve pas dans le tableau, n comparaisons sont nécessaires.

– *en moyenne* : la complexité ne peut être évaluée qu'en faisant des hypothèses statistiques sur les cas d'exécution. Pour notre algorithme, prenons les hypothèses suivantes :
  – quand la valeur recherchée se trouve dans le tableau, on la trouve après avoir parcouru en moyenne la moitié du tableau ;
  – pour simplifier, on considérera qu'en moyenne une fois sur deux la valeur recherchée ne se trouve pas dans le tableau.

  Donc, si la valeur se trouve dans le tableau on fait en moyenne $\frac{1}{2}n$ comparaisons. Si elle ne se trouve pas dans le tableau on fait `n` comparaisons. Selon la s econde hypothèse ci-dessus, le nombre moyen de comparaisons sera la moyenne des deux, donc $\frac{3}{4}n$.

En conclusion, la complexité de l'algorithme de recherche dans un tableau est :
– $C_{min}(n) = 1$
– $C_{max}(n) = n$
– $C_{moy}(n) = \frac{3}{4}n$

Remarque : Si l'intervalle possible des valeurs dans le tableau est très large (par exemple tous les entiers), la complexité moyenne est proche de `n`, car la probabilité qu'une valeur au hasard soit dans le tableau est très mince. Donc la grande majorité des recherches se solderont par des échecs et coûteront `n` comparaisons.

7.1.4 Ordre de grandeur de la complexité

En réalité, quand on évalue la complexité d'un algorithme, ce qui nous intéresse le plus souvent c'est la rapidité de celui-ci dans le traitement *de grandes quantités d'information*. En d'autres termes, on s'intéresse à la manière dont le temps d'exécution augmente avec la taille des données. Pour de grandes tailles de données, ce qui compte c'est *l'ordre de grandeur* de la complexité.

Définition 1 *Soit* `C(n)` *la complexité d'un algorithme* `A`. *On dit que la complexité de* `A` *est d'ordre de grandeur* `f(n)`, *noté* `C(n) = O(f(n))`, *s'il existe une constante* `c`, *telle que pour tout* `n` *suffisamment grand on a* `C(n) <= c f(n)`.

Remarque 1 : Cette définition dit qu'à partir d'un certain `n`, la complexité ne dépasse jamais `f(n)` (éventuellement multipliée par une constante).

Remarque 2 : Les constantes de multiplication ne comptent pas dans l'ordre de grandeur de la complexité.

Remarque 3 : Cette définition de l'ordre de grandeur n'exprime *qu'une limite supérieure* de la complexité. Il existe des définitions plus précises, mais nous nous limiterons ici à celle-ci, pour des raisons de simplicité.

Remarque 4 : Puisque `O(f(n))` ne donne qu'une limite supérieure de la complexité, rien ne nous empêche de prendre un `f(n)` "trop complexe", pour être sûrs de dépasser `C(n)`. Par exemple si `C(n)=n`, il est vrai que `C(n) = O(n)`, mais aussi que $C(n) = O(n^2)$ ou que $C(n) = O(n^3)$, etc. Nous avons l'intérêt d'avoir l'estimation la plus précise de l'ordre de grandeur, alors dans notre exemple l'ordre de grandeur sera `C(n) = O(n)`. La règle ci-dessous sera toujours appliquée dans l'estimation de l'ordre de grandeur.

Règle : Dans l'estimation de l'ordre de grandeur on gardra seulement *le terme le plus significatif* de `C(n)`, c'est à dire celui qui provoque l'augmentation la plus forte quand `n` augmente. Comme les constantes de multiplication ne comptent pas, on utilisera toujours la constante 1.

Suivant cette règle, si `f(n)` est un polynôme d'ordre `k` en `n`, l'ordre de grandeur sera $O(n^k)$. Alors, pour notre exemple d'algorithme de recherche on a :

- $C_{min}(n) = O(1)$
- $C_{max}(n) = O(n)$
- $C_{moy}(n) = O(n)$

L'importance de l'ordre de grandeur de la complexité

L'ordre de grandeur de la complexité nous permet de savoir si l'algorithme peut traiter dans un temps raisonnable des données de grande taille. Pour un algorithme `O(n)` (dit linéaire), le temps d'exécution est proportionnel à la taille des données, donc si on double `n`, le temps double. Par contre pour un algorithme $O(n^2)$ (dit quadratique), si `n` double, le temps est multiplié par 4, pour $O(n^3)$ le temps est multiplié par 8, etc. Pour un algorithme exponentiel $O(a^n)$, doubler `n` signifie obtenir le carré du temps d'exécution ! Et il existe des algorithmes connus qui ont une complexité encore plus grande.

Parmi les problèmes complexes les plus connus, celui du commis voyageur demande à trouver le chemin le plus court pour un commis voyageur qui doit visiter `n` villes. Si on calcule tous les ordres de parcours possibles pour décider du chemin le plus court, il y a `n!` ordres possibles de parcours des `n` villes. Ceci donne une complexité `O(n!)`, supérieure à une complexité exponentielle. Dans un tel algorithme, augmenter `n` de seulement une unité multiplie le temps d'exécution par `n`. Déjà pour 20 villes, à une opération par nanoseconde (10^{-9}s), le temps d'exécution dépasse mille années !

L'ordre de grandeur détermine la classe de complexité de l'algorithme : temps constant, linéaire, quadratique, polynômial, exponentiel, etc. La classe de complexité est le critère le plus utilisé de classification des algorithmes selon la complexité. A l'intérieur d'une même classe, pour comparer deux algorithmes il faut regarder d'autres critères : le coefficient du terme significatif dans la complexité, les autres termes, la mémoire occupée, etc.

7.2 Algorithmes de recherche dans un tableau trié

Nous nous intéressons ici à la recherche d'une valeur dans un tableau déjà trié en ordre croissant. Nous étudions deux algorithmes, la recherche séquentielle et la recherche dichotomique.

7.2.1 Recherche séquentielle dans un tableau trié

La recherche séquentielle d'une valeur dans un tableau trié ressemble à la recherche dans un tableau non-trié, présentée ci-dessus. Le tableau est parcouru de gauche à droite à la recherche de la valeur, mais l'avantage par rapport au tableau non-trié apparaît quand la valeur ne se trouve pas dans le tableau. Dans ce cas, le parcours peut s'arrêter dès que l'élément courant du tableau dépasse la valeur recherchée. On est sûrs dans ce cas, a cause du tri, que toutes les valeurs suivantes dans le tableau sont plus grandes que la valeur recherchée.

Algorithme *recherche_triée*
Entrées : tableau `T` trié, valeur `v`
Sorties : boolean qui indique si `v` se trouve dans `T` ou non
début

```
pour chaque indice dans T répéter
   si T[indice] == v alors retourner vrai
```

```
    si T[indice] > v alors retourner faux
  fin répéter
  retourner faux
```
fin *recherche_triée*

Cet algorithme est valable pour tout tableau pour lequel on sait comparer les éléments par égalité et par une relation d'ordre. La fonction Java qui réalise cet algorithme pour un tableau d'entiers est la suivante :

```
boolean recherche_triee(int[] T, int v){
  for(int i=0; i<T.length; i++){
    if(T[i] == v) return true;
    if(T[i] > v) return false;
  }
  return false;
}
```

Complexité

A la différence de la recherche non-triée, chaque position nécessite deux comparaisons au lieu d'une. Par contre, la recherche triée ne parcourt en moyenne que la moitié du tableau, que la valeur se trouve ou non dans le tableau. On obtient les complexités suivantes :

- dans le cas le plus favorable, une seule comparaison est nécessaire, $C_{min}(n) = 1 = O(1)$.
- dans le cas le plus défavorable (valeur recherchée plus grande que la dernière valeur du tableau), on fait $C_{max}(n) = 2n = O(n)$.
- en moyenne, on parcourt la moitié du tableau, donc $C_{moy}(n) = 2 * \frac{1}{2}n = n = O(n)$.

En conclusion, la recherche triée a le même ordre de complexité que la recherche non-triée et, paradoxellement, fais plus de comparaisons en moyenne et surtout dans le cas le plus défavorable !

Heureusement, on peut améliorer l'algorithme de recherche triée de la manière suivante. Au lieu de faire les deux comparaisons pour chaque position, on en fait une seule, en cherchant la position *après* l'élément recherché, c'est à dire en avançant tant que l'élément du tableau est $\leq$ v. Voici l'algorithme amélioré :

Algorithme *recherche_triée2*
Entrées : tableau T trié, valeur v
Sorties : boolean qui indique si v se trouve dans T ou non
début
```
  indice = premier indice de T
  tant que indice dans T et T[indice] ≤ v répéter
    indice = indice + 1
  fin répéter
  si indice-1 dans T et T[indice-1] == v alors retourner vrai
  sinon retourner faux
  fin si
```
fin *recherche_triée2*

La fonction Java qui réalise cet algorithme pour un tableau d'entiers est la suivante :

```
boolean recherche_triee2(int[] T, int v){
  int indice = 0;
```

```
  while(indice < T.length && T[indice] <= v)
    indice++;
  return indice>0 && T[indice-1] == v;
}
```

Cet algorithme va une position plus loin dans le tableau que l'algorithme précédent, si l'élément recherché se trouve dans le tableau (ou plusieurs positions si l'élément recherché apparaît plusieurs fois, sur des positions successives dans le tableau). Néanmoins, il fait une seule comparaison par position, plus une comparaison finale pour vérifier si la valeur recherchée se trouve sur la position précédente (attention, on ne compte que les comparaisons de valeurs dans le tableau, pas les comparaisons d'indices).

Complexité de l'algorithme amélioré

- dans le cas le plus favorable, une seule comparaison est nécessaire, quand `v` est plus petite que le premier élément de `T`. $C_{min}(n) = 1 = O(1)$.
- dans le cas le plus défavorable, on va jusqu'à la fin du tableau et on fait la comparaison finale, donc $C_{max}(n) = n + 1 = O(n)$.
- en moyenne, on parcourt la moitié du tableau, plus la comparaison finale, donc $C_{moy}(n) = \frac{1}{2}n + 1 = O(n)$.

En conclusion, l'algorithme amélioré est plus efficace que la recherche non-triée en moyenne, aussi efficace (quasiment) dans le cas le plus défavorable et pareil pour le cas le plus favorable.

7.2.2 Recherche dichotomique dans un tableau trié

L'algorithme de recherche séquentielle triée n'utilise pas vraiment le fait que les valeurs sont stockées dans un tableau. Le même algorithme peut par exemple être transposé facilement aux listes chaînées. En exploitant la possiblité d'accès direct à n'importe quel indice du tableau, on peut améliorer fortement les performances de la recherche triée.

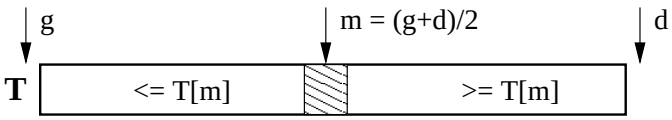


FIG. 7.1 – Recherche dichotomique

L'idée de la recherche dichotomique est montrée dans la figure 7.1. Si on regarde la valeur de l'élément du milieu (indice `m`) de `T`, il y a 3 possibilités :

- `v==T[m]`, alors la recherche s'arrête, on a trouvé la valeur.
- `v<T[m]`, alors il faut continuer la recherche dans la moitié de gauche du tableau, délimitée par les indices `g` et `m`.
- `v>T[m]`, alors il faut continuer la recherche dans la moitié de droite du tableau, délimitée par les indices `m` et `d`.

De cette façon, le tableau à explorer est découpé en deux à chaque pas (dichotomie) et donc sa taille est réduite de moitié. Soit la valeur est retrouvée au milieu d'un de ces tableaux, soit le tableau devient vide et la recherche échoue.

Remarque : Les limites du tableau, les indices `g` et `d` se trouvent *en dehors du tableau à explorer*. Ceci simplifie la condition d'arrêt sur échec : le tableau est vide quand `d-g==1`.

Algorithme *recherche_dichotomique*

Entrées : tableau T trié, valeur v

Sorties : boolean qui indique si v se trouve dans T ou non

début

```
g = indice avant le premier indice de T
d = indice après le dernier indice de T
tant que d - g > 1 répéter
   m = (g + d)/2
   si v == T[m] alors retourner vrai
   si v < T[m] alors d = m //recherche dans la moitié de gauche
   sinon g = m //recherche dans la moitié de droite
   fin si
fin répéter
retourner faux
```

fin *recherche_dichotomique*

La fonction Java qui réalise cet algorithme pour un tableau d'entiers est la suivante :

```
boolean recherche_dichotomique(int[] T, int v){
  int g = -1;
  int d = T.length;
  while(d-g > 1){
    int m = (g+d)/2;
    if(v == T[m]) return true;
    if(v < T[m]) d = m;
    else g = m;
  }
  return false;
}
```

Complexité de la recherche dichotomique

La recherche dichotomique réduit à moitié la taille du tableau exploré, à chaque pas. Donc après k pas, la taille du tableau devient $\frac{n}{2^k}$. Il y a donc $log_2(n)$ pas avant que le tableau ne devienne vide. Pour chaque tableau, on fait une ou deux comparaisons avec l'élément médian (une en cas de succès, deux si la recherche doit continuer, pour déterminer dans quelle moitié on continue).

- dans le cas le plus favorable, une seule comparaison est nécessaire, quand on trouve v au milieu du tableau initial, donc $C_{min}(n) = 1 = O(1)$.
- le cas le plus défavorable est quand on ne trouve pas v dans T, donc pour chacun des $log_2(n)$ pas on fait 2 comparaisons. $C_{max}(n) = 2log_2(n) = O(log(n))$.
- en moyenne, on ne réduit pas beaucoup la complexité, car il y a généralement plus de chances que v ne soit pas dans T. Même si v s'y trouve, on peut démontrer qu'en moyenne on fait $log_2(n) - 1$ pas, ce qui ne change pratiquement pas la complexité. Donc $C_{moy}(n) = O(log(n))$.

Conclusion : La recherche dichotomique est beaucoup plus efficace que la recherche séquentielle. Les deux algorithmes appartiennent à des classes de complexité différentes : la recherche dichotomique est logarithmique, tandis que la recherche séquentielle est linéaire.

Pour voir la différence d'efficacité entre les deux algorithmes, considérons les cas suivants :

– n = 10 : la recherche séquentielle fait en moyenne 5 comparaisons, la recherche dichotomique environ 4
– n = 100 : la recherche séquentielle fait en moyenne 50 comparaisons, la recherche dichotomique environ 7
– n = 1000 : la recherche séquentielle fait en moyenne 500 comparaisons, la recherche dichotomique environ 10
– n = 1.000.000 : la recherche séquentielle fait en moyenne 500.000 comparaisons, la recherche dichotomique environ 20

7.3 Algorithmes de tri d'un tableau

Les algorithmes de tri sont parmi les plus étudiés en algorithmique, car ils sont très souvent utilisés en pratique et sont plus complexes que la recherche, l'insertion, la suppression, etc. Nous présentons deux types d'algorithmes de tri : des algorithmes dits *simples*, qui ont une complexité quadratique $O(n^2)$ et des algorithmes *évolués*, de complexité quasi-linéaire $O(n\,log(n))$.

7.3.1 Algorithmes simples de tri : le tri par sélection

L'idée de l'algorithme est de chercher le plus petit élément et de le placer au début du tableau. Ensuite on cherche le plus petit de ce qui reste et on le place sur la deuxième position, etc.

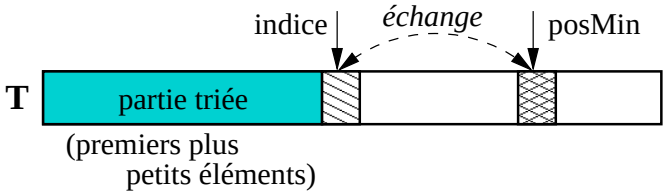


FIG. 7.2 – Tri par sélection

La figure 7.2 illustre l'algorithme de tri par sélection :
– on parcourt le tableau avec un indice, à partir du premier, jusqu'à l'avant-dernier élément.
– à un moment donné, les éléments à gauche de l'indice sont les plus petits éléments du tableau, dans l'ordre.
– on cherche le plus petit élément dans le tableau, à partir de l'indice jusqu'à la fin du tableau.
– on échange l'élément de l'indice avec l'élément le plus petit trouvé.
– on avance l'indice sur l'élément suivant ; quand il arrive sur le dernier élément du tableau, le tri est terminé, car le dernier élément qui reste est forcément le plus grand du tableau.

Algorithme *tri_sélection*
Entrées : tableau `T`
Effet : tableau `T` trié
début
`pour chaque indice` dans `T`, sauf le dernier `répéter`
`posMin` = ***position_min***(`T`, `indice`)
échanger les éléments d'indices `posMin` et `indice` dans `T`
`fin répéter`

fin *tri_sélection*

Algorithme *position_min*
Entrées : tableau `T`, indice de début `debut`
Sorties : position de l'élément minimal dans `T` entre `debut` et la fin de `T`
début
`posMin` = `debut`
`pour chaque indice` dans `T`, à partir de `debut+1` `répéter`
`si T[indice] < T[posMin] alors posMin = indice`
`fin répéter`
`retourner posMin`
fin *position_min*

Voici maintenant une réalisation de cet algorithme en Java, sous la forme d'un programme complet, qui demande à l'utilisateur un tableau d'entiers et qui affiche le tableau trié.

```
public class TriSelection{
  public static void main(String[] args){
    Terminal.ecrireStringln("Introduire le tableau d'entiers:");
    int[] T = lireTableau();
    triSelection(T);
    Terminal.ecrireStringln("Voici le tableau trie:");
    ecrireTableau(T);
  }
  static int[] lireTableau(){
    Terminal.ecrireString("Nombre d'elements:");
    int n = Terminal.lireInt();
    int[] tab = new int[n];
    for(int i=0; i<n; i++){
      Terminal.ecrireString("Element " + i + "=");
      tab[i] = Terminal.lireInt();
    }
    return tab;
  }
  static void ecrireTableau(int[] tab){
    for(int i=0; i<tab.length; i++)
      Terminal.ecrireString(" " + tab[i]);
    Terminal.sautDeLigne();
  }
  static void triSelection(int[] T){
    for(int i=0; i<T.length-1; i++){
      int posMin = positionMin(T, i);
      if(posMin != i){ //échanger T[posMin] et T[i]
        int temp = T[i]; T[i] = T[posMin]; T[posMin] = temp;
      }
    }
  }
  static int positionMin(int[] T, int debut){
    int posMin = debut;
    for(int i=debut+1; i<T.length; i++){
      if(T[i] < T[posMin]) posMin = i;
```

```
    }
    return posMin;
  }
}
```

Complexité du tri par sélection

Deux types d'opérations sont importantes dans les algorithmes de tri en général : *les comparaisons d'éléments* et *les copies d'éléments*. Comptons le nombre d'opérations réalisées dans le tri par sélection.

Comparaisons

Les comparaisons d'éléments sont réalisées dans ***position_min***, à la recherche de l'élément minimal de la portion de tableau. Le nombre de comparaisons dépend de la taille du tableau dans lequel on recherche le minimum.

Pour un tableau de taille `k`, on doit faire `k-1` comparaisons pour trouver le minimum. La première fois le tableau a la taille `n`, ensuite `n-1`, puis `n-2`, etc. Donc le nombre de comparaisons est :

$$Comp(n) = (n-1) + (n-2) + ... + 1 = \frac{n*(n-1)}{2} = O(n^2)$$

Ce nombre est le même dans tous les cas (le plus favorable, le moins défavorable ou moyen).

Copies

Les copies d'éléments sont réalisées lors de l'échange, à chaque pas, de l'élément courant de l'indice et du minimum trouvé à droite. Pour un échange, il y a 3 copies d'éléments.

Néanmoins, la copie ne se fait pas si l'élément courant est le même que le minimum trouvé à ce pas-là. Cela signifie que dans le cas le plus favorable, quand le tableau est déjà trié, l'algorithme ne fait aucune copie. Dans le cas le plus défavorable, on fait un échange à chacun des `n-1` pas, donc il y a en tout `3(n-1)` copies. En moyenne, on peut considérer que l'échange se fait seulement dans une partie des pas (probablément plus de la moitié), ce qui ne modifie pas l'ordre de grandeur de la complexité moyenne, qui reste linéaire. En conclusion :

$$Copies_{min}(n) = 0 = O(1)$$

$$Copies_{max}(n) = 3(n-1) = O(n)$$

$$Copies_{moy}(n) = O(n)$$

Conclusion : Le tri par sélection est assez inefficace en termes de comparaisons (complexité quadratique) et ne profite pas de l'éventuel tri partiel ou total du tableau initial. Par contre il est très efficace en termes de copies d'éléments (complexité linéaire) et sans aucune copie à faire si le tableau est déjà trié.

7.3.2 Algorithmes simples de tri : le tri par insertion

L'idée du tri par insertion est de construire pas à pas le tableau trié *en insérant* dans le tableau résultat (trié) un nouvel élément à chaque pas.

Le tableau résultat pourrait être différent du tableau initial - au début il serait vide et ensuite à chaque pas on insère dans ce tableau un nouvel élément du tableau initial, jusqu'à ce que tous les éléments soient insérés. A chaque pas, le tableau résultat reste trié et chaque nouvelle insertion garde le tableau trié.

Il n'est cependant pas nécessaire d'utiliser un autre tableau pour le résultat. Les éléments du tableau initial sont parcourus de gauche à droite, donc à un moment donné la partie gauche du tableau (à gauche de l'élément courant) a déjà été parcourue et peut être utilisée pour stocker le résultat (voir la figure 7.3).

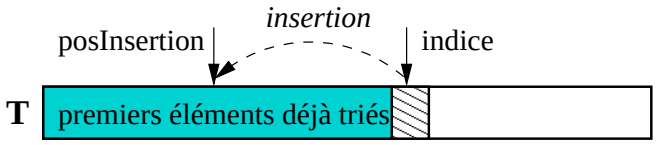


FIG. 7.3 – Tri par insertion

- on parcourt le tableau avec un indice, à partir du second, jusqu'au dernier élément.
- à un moment donné, les éléments à gauche de l'indice ont déjà été traités et le résultat est que la partie du tableau à gauche de l'index est déjà triée. Au début, l'indice est sur le second élément et la partie du tableau à sa gauche est forcément triée, car constituée d'un seul élément (le premier).
- on insère l'élément de l'indice dans la partie de tableau triée, située à gauche de l'indice. Ainsi la taille du tableau résultat a augmenté de 1.
- on avance l'indice sur l'élément suivant ; quand il dépasse le dernier élément du tableau, le tri est terminé.

Algorithme *tri_insertion*
Entrées : tableau `T`
Effet : tableau `T` trié
début
`pour chaque indice` dans `T`, sauf le premier `répéter`
insertion(`T`, `indice`)
`fin répéter`
fin *tri_insertion*

Algorithme *insertion*
Entrées : tableau `T`, indice de fin `fin`
Condition : `T` est trié du début jusqu'à `fin-1`
Effet : tableau `T` trié du début jusqu'à `fin`
début
`posInsertion` = ***position_insertion***(`T`, `fin-1`, `T[fin]`)
`si posInsertion == fin alors retourner` *//rien à faire*
`aInserer` = `T[fin]`
`pour chaque indice` descendant de `fin-1` à `posInsertion répéter`
`T[indice+1] = T[indice]` *//déplacement d'une position à droite*
`fin répéter`
`T[posInsertion] = aInserer` *//l'insertion*
fin *insertion*

Algorithme *position_insertion*

Entrées : tableau `T`, indice de fin `fin`, `valeur` à insérer
Condition : `T` est trié du début jusqu'à `fin`
Sorties : position d'insertion triée

début

```
indice = fin
tant que indice \geq premier indice de T et T[indice] > v répéter
   indice = indice - 1
fin répéter
retourner indice + 1
```

fin *position_insertion*

Remarque : La recherche de la position d'insertion avec ***position_insertion*** utilise une méthode de recherche séquentielle dans un tableau trié, similaire à celle optimisée présentée ci-dessus. La différence est que la recherche se fait de la fin vers le début du tableau.

La raison est que l'élément est recherché *pour être inséré* ensuite dans le tableau à sa place. Il se trouve déjà à la suite (après la fin) du tableau exploré. Or, dans le cas le plus favorable, l'élément doit être inséré justement à cette place, ce qui arrive quand l'élément à insérer est plus grand que tous les éléments du tableau. Dans ce cas, il n'y a aucune opération à faire pour l'insertion.

Faire la recherche de la fin vers le début permet de limiter le nombre de comparaisons à une seule dans le cas le plus favorable pour l'insertion. Dans la même situation, une recherche du début vers la fin doit parcourir tout le tableau.

Remarque : Il est tout à fait possible d'utiliser la recherche dichotomique pour trouver la position d'insertion. On obtient ainsi un algorithme plus efficace.

Voici maintenant une réalisation de cet algorithme en Java, sous la forme d'un programme complet, comme pour le tri par sélection.

```
public class TriInsertion{
  public static void main(String[] args){
    Terminal.ecrireStringln("Introduire le tableau d'entiers:");
    int[] T = lireTableau();
    triInsertion(T);
    Terminal.ecrireStringln("Voici le tableau trie:");
    ecrireTableau(T);
  }
  static int[] lireTableau(){
    Terminal.ecrireString("Nombre d'elements:");
    int n = Terminal.lireInt();
    int[] tab = new int[n];
    for(int i=0; i<n; i++){
      Terminal.ecrireString("Element " + i + "=");
      tab[i] = Terminal.lireInt();
    }
    return tab;
  }
  static void ecrireTableau(int[] tab){
    for(int i=0; i<tab.length; i++)
      Terminal.ecrireString(" " + tab[i]);
    Terminal.sautDeLigne();
```

```
  }
  static void triInsertion(int[] T){
    for(int i=1; i<T.length; i++)
      insertion(T, i);
  }
  static void insertion(int[] T, int fin){
    int posInsertion = positionInsertion(T, fin-1, T[fin]);
    if(posInsertion == fin) return; //déjà en place
    int aInserer = T[fin];
    for(int i=fin-1; i>=posInsertion; i--)
      T[i+1] = T[i]; //décalage d'une position à droite
    T[posInsertion] = aInserer;
  }
  static int positionInsertion(int[] T, int fin, int elem){
    int pos = fin;
    while(pos >= 0 && T[pos] > elem)
      pos--;
    return pos+1;
  }
}
```

Complexité du tri par insertion

Comptons le nombre d'opérations réalisées dans le tri par insertion.

Comparaisons

Les comparaisons d'éléments sont réalisées dans ***position_insertion***, à la recherche de la position d'insertion triée. Le nombre de comparaisons dépend de la taille du tableau trié et de la méthode utilisée (séquentielle ou dichotomique).

Pour un tableau de taille k :

- pour *la recherche séquentielle* on fait :
  - 1 comparaison, dans le cas le plus favorable, pour une insertion à la fin.
  - k comparaisons, dans le cas le plus défavorable, pour une insertion au début.
  - $\frac{k}{2}$ comparaisons en moyenne.
- pour *la recherche dichotomique* on fait :
  - 1 comparaison, dans le cas le plus favorable, si l'élément se trouve déjà au milieu du tableau (on inserera juste après lui).
  - $O(log_2(k))$ comparaisons, dans le cas le plus défavorable et en moyenne.

La recherche se fait successivement dans des tableaux de tailles 1, 2, ..., n-1. Donc le nombre de comparaisons est :

- pour *la recherche séquentielle* :
  - $Comp_{min}(n) = n - 1 = O(n)$, dans le cas le plus favorable (tableau déjà trié).
  - $Comp_{max}(n) = 1 + 2 + ... + (n-1) = \frac{n*(n-1)}{2} = O(n^2)$, dans le cas le plus défavorable (tableau trié à l'envers).
  - $Comp_{moy}(n) = \frac{1}{2}Comp_{max}(n) = \frac{n*(n-1)}{4} = O(n^2)$, en moyenne.
- pour *la recherche dichotomique* :
  - $Comp_{min}(n) = n - 1 = O(n)$, dans le cas le plus favorable, qui n'est pas pour le tableau déjà trié (mais pour un cas beaucoup moins fréquent). On pourrait néanmoins modifier légèrement la

recherche dichotomique pour vérifier d'abord la fin du tableau, ce qui fera correspondre la cas favorable avec le tableau trié, sans modifier la complexité.
- $Comp_{max}(n) = Comp_{moy}(n) = O(log(1) + log(2) + ... + log(n-1)) \simeq O(n\,log(n))$, dans le cas le plus défavorable et en moyenne.

Copies

Les copies d'éléments sont réalisées lors de l'insertion, à chaque pas, de l'élément courant dans le tableau résultat de gauche. Pour insérer un élément dans le tableau résultat, les éléments à droite de la position d'insertion doivent être déplacés d'une position à droite. Ensuite, l'élément à insérer est copié à cette position.

Pour insérer dans un tableau de taille `k`, on fait :
- aucune copie, dans le cas le plus favorable, pour une insertion à la fin.
- `k+1` copies, dans le cas le plus défavorable, pour une insertion au début.
- $\frac{1}{2}k + 1$ copies, en moyenne.

L'insertion se fait successivement dans des tableaux de tailles 1, 2, ..., `n-1`. Donc le nombre de copies est :
- $Copies_{min}(n) = 0 = O(1)$, dans le cas le plus favorable (tableau déjà trié).
- $Copies_{max}(n) = 2 + 3 + ... + n = \frac{n*(n+1)}{2} - 1 = O(n^2)$, dans le cas le plus défavorable (tableau trié à l'envers).
- $Copies_{moy}(n) = n - 1 + \frac{1}{2}(1 + 2 + ... + (n-1)) = n - 1 + \frac{n*(n-1)}{4} = O(n^2)$, en moyenne.

Conclusion : Le tri par insertion est assez inefficace en termes de copies (complexité quadratique), mais avec la recherche dichotomique il est efficace en termes de comparaisons (complexité quasi-linéaire). Il profite bien de l'éventuel tri préalable du tableau initial, avec une complexité linéaire en comparaisons et aucune copie.

7.3.3 Algorithmes évolués de tri : le tri rapide

Les algorithmes précédents sont des algorithmes simples de tri. Ils ont une complexité globale quadratique, même si le tri par sélection est linéaire en nombre de copies et le tri par insertion avec dichotomie est quasi-linéaire en nombre de comparaisons.

Le tri rapide est probablement l'algorithme évolué de tri le plus utilisé en pratique. Bien que sa complexité dans le cas le plus défavorable est quadratique, en moyenne il a une complexité quasi-linéaire. En pratique, il donne de meilleures performances en moyenne que d'autres algorithmes évolués, qui ont une complexité quasi-linéaire dans tous les cas.

L'idée du tri rapide est la suivante. On choisit (au hasard) une valeur appelée *valeur pivot*. Ensuite on sépare les éléments du tableau, de manière à ce qu'on retrouve *à gauche des éléments inférieurs à la valeur pivot* et *à droite des éléments supérieurs à la valeur pivot* (voir la figure 7.4, en haut). Si maintenant les deux parties du tableau sont triées séparément, alors le tableau entier sera trié. L'astuce est d'utiliser récursivement la même méthode de tri (le tri rapide) à chacune de ces deux parties.

Idéalement, la valeur pivot sépare toujours le tableau en deux parties de même taille. Ainsi, à chaque pas la taille des tableaux à traiter est divisée par 2, ce qui minimise le nombre de pas, qui est dans ce cas de l'ordre de $log_2(n)$.

Par contre, si la valeur pivot est mal choisie, le découpage peut être complétement déséquilibré, avec une partie vide et une autre comprenant tout le tableau. Dans ce cas, l'algorithme peut ne pas se terminer. Il est donc important que la taille des parties de tableau diminue. Pour cela, *on choisit comme valeur pivot la valeur d'un élément du tableau,* qu'on appelera *pivot*.

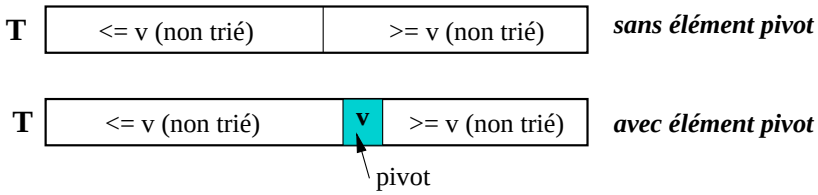


FIG. 7.4 – Séparation des valeurs par rapport à une valeur pivot `v`

L'objectif est qu'après la phase de séparation par rapport à la valeur pivot, on retrouve le pivot entre les deux parties (figure 7.4, en bas). Donc, après la phase de séparation on retrouve :

- une partie gauche avec des éléments $\leq$ valeur pivot,
- l'élément pivot,
- une partie droite avec des éléments $\geq$ valeur pivot.

De cette façon, on est sûrs que la taille des parties décroit à chaque pas.

La partie la plus importante de l'algorithme est donc *la phase de séparation*. L'idée de son algorithme est la suivante :

- on choisit comme pivot l'élément moyen parmi le premier élément du tableau, le dernier et celui placé au milieu du tableau. On échange les valeurs des 3 éléments tel que `T[debut]` $\leq$ `T[milieu]` $\leq$ `T[fin]`. Ainsi le pivot se retrouve au milieu du tableau.
- on parcourt le tableau avec un indice qui part de gauche et avance pour retrouver un élément plus grand que le pivot, qui n'est pas à sa place à gauche du tableau.
- on parcourt aussi le tableau avec un indice qui part de droite et recule pour retrouver un élément plus petit que le pivot, qui n'est pas à sa place à droite du tableau.
- on échange les éléments trouvés par les indices de gauche et de droite, qui se retrouvent maintenant dans la bonne partie du tableau (figure 7.5, en haut).
- on continue le parcours des deux indices jusqu'à ce qu'ils se rejoignent.
- pour que le pivot reste entre les deux parties séparés, dès qu'un des deux indices arrive sur le pivot, mais pas l'autre, on échange le pivot avec l'élément pointé par l'autre indice (figure 7.5, en bas).

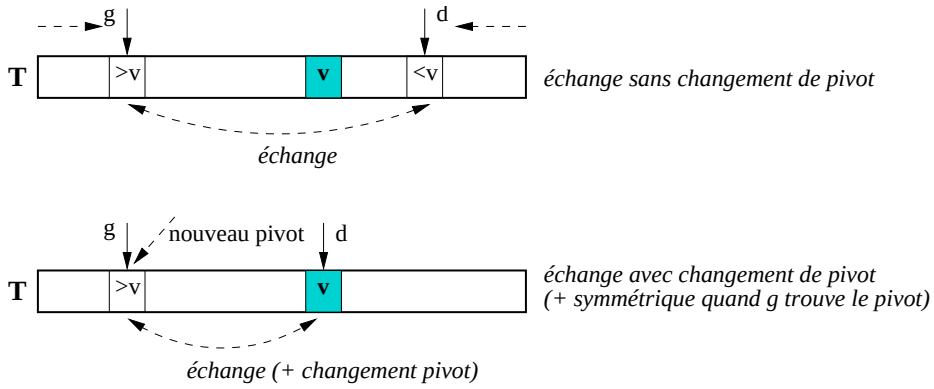


FIG. 7.5 – Echange de valeurs pour la séparation

Algorithme *tri_rapide*

Entrées : tableau `T`, `debut` et `fin` indices de début et de fin

Effet : tableau T trié entre debut et fin
début
si fin - debut < 2 alors *//tri "manuel"*
trier directement T entre debut et fin
retourner
fin si
fixer_pivot(T, debut, fin)
posPivot = ***séparation***(T, debut, fin)
tri_rapide(T, debut, posPivot-1)
tri_rapide(T, posPivot+1, fin)
fin *tri_rapide*

Algorithme *fixer_pivot*
Entrées : tableau T, debut et fin indices de début et de fin
Effet : choix pivot parmi T[debut], T[fin], T[(debut+fin)/2]
échange des 3 valeurs tel que T[debut] ≤ T[(debut+fin)/2] ≤ T[fin]
début
milieu = (debut+fin)/2
si T[debut] > T[fin] alors échange T[debut], T[fin]
si T[milieu] < T[debut] alors échange T[debut], T[milieu]
sinon si T[milieu] > T[fin] alors échange T[fin], T[milieu]
fin *fixer_pivot*

Algorithme *séparation*
Entrées : tableau T, debut et fin indices de début et de fin
Condition : T[(debut+fin)/2] contient la valeur pivot
Sorties : pos position finale du pivot après séparation
Effet : valeurs ≤ pivot avant pos et ≥ pivot après pos
début

```
posPivot = (debut+fin)/2
gauche = debut
droite = fin
tant que gauche < droite répéter
   tant que gauche < posPivot et T[gauche] ≤ T[posPivot] répéter
      gauche = gauche + 1
   fin répéter
   tant que posPivot < droite et T[droite] \geq T[posPivot] répéter
      droite = droite - 1
   fin répéter
   si gauche < droite alors
      échanger T[gauche] et T[droite]
      si gauche == posPivot alors posPivot = droite
      sinon si droite == posPivot alors posPivot = gauche
      fin si
   fin si
```

```
  fin répéter
  retourner posPivot
fin séparation
```

Voici maintenant une réalisation récursive de cet algorithme en Java, sous la forme d'un programme complet.

```
public class TriRapide{
  public static void main(String[] args){
    Terminal.ecrireStringln("Introduire le tableau d'entiers:");
    int[] T = lireTableau();
    triRapide(T, 0, T.length-1);
    Terminal.ecrireStringln("Voici le tableau trie:");
    ecrireTableau(T);
  }
  static int[] lireTableau(){
    Terminal.ecrireString("Nombre d'elements:");
    int n = Terminal.lireInt();
    int[] tab = new int[n];
    for(int i=0; i<n; i++){
      Terminal.ecrireString("Element " + i + "=");
      tab[i] = Terminal.lireInt();
    }
    return tab;
  }
  static void ecrireTableau(int[] tab){
    for(int i=0; i<tab.length; i++)
      Terminal.ecrireString(" " + tab[i]);
    Terminal.sautDeLigne();
  }
  static void triRapide(int[] T, int debut, int fin){
    if(fin - debut < 2){ //tri manuel
      if(fin - debut < 1) return; //déjà trié
      if(T[fin] < T[debut]){ //échanger
        int temp=T[fin]; T[fin]=T[debut]; T[debut]=temp;
      }
      return;
    }
    fixerPivot(T, debut, fin);
    int posPivot = separation(T, debut, fin);
    triRapide(T, debut, posPivot-1);
    triRapide(T, posPivot+1, fin);
  }
  static void fixerPivot(int[] T, int debut, int fin){
    int milieu = (debut + fin)/2;
    //rendre T[debut] <= T[fin]
    if(T[debut] > T[fin]){
      int temp=T[fin]; T[fin]=T[debut]; T[debut]=temp;
    }
    if(T[milieu] < T[debut]){ //éviter que T[milieu] soit le plus petit
      int temp=T[milieu]; T[milieu]=T[debut]; T[debut]=temp;
    }
```

```
    else if(T[milieu] > T[fin]){ //éviter que T[milieu] soit le plus grand
      int temp=T[milieu]; T[milieu]=T[fin]; T[fin]=temp;
    }
  }
  static int separation(int[] T, int debut, int fin){
    int posPivot = (debut + fin)/2;
    int gauche = debut;
    int droite = fin;
    while(gauche < droite){
      while(gauche < posPivot && T[gauche] <= T[posPivot]) gauche++;
      while(droite > posPivot && T[droite] >= T[posPivot]) droite--;
      if(gauche < droite){
        int temp=T[gauche]; T[gauche]=T[droite]; T[droite]=temp;
        if(gauche == posPivot) posPivot=droite;
        else if(droite == posPivot) posPivot=gauche;
      }
    }
    return posPivot;
  }
}
```

Complexité du tri rapide

La phase de séparation réalise un nombre de comparaisons proportionnel à la taille du tableau, car `gauche` et `droite` balayent ensemble la totalité du tableau. On a donc pour un tableau de taille `k` un nombre de comparaisons `O(k)`, quelque soit le cas.

Pendant la séparation, le nombre de copies dépend du nombre de couples d'éléments qui ne sont pas dans la bonne moitié de tableau. Donc pour un tableau de taille `k`, on a :

- aucune copie, dans le cas le plus favorable (tableau trié)
- `O(k)` copies, dans le cas le plus défavorable (tableau trié à l'envers), de même qu'en moyenne.

La phase de fixation du pivot ne modifie en rien ces complexités, elle rajoute un nombre constant de comparaisons et de copies. Dans le cas d'un tableau déjà trié, elle ne rajoute aucune copie.

Pour évaluer la complexité totale, il faut évaluer combien de fois et sur des tableaux de quelle taille se fait la séparation.

On peut démontrer que *le cas le plus favorable* se produit quand chaque séparation divise le tableau en deux parties de taille (presque) égale. On peut alors voir le processus de tri comme une succession de pas : au pas 1 on obtient 2 parts de taille $\frac{n}{2}$, au pas 2, 4 parts de taille $\frac{n}{4}$, ..., au pas `i`, 2^i parts de taille $\frac{n}{2^i}$, etc.

Il y a donc au plus $log_2(n)$ pas avant que les tableaux n'arrivent à la taille du tri manuel. A chaque pas, la somme des complexités pour les parts de tableau qui composent le tableau complet est `O(n)`, car :

- la complexité de la séparation est linéaire,
- la somme des tailles des parts est `n`.

Donc, dans le cas de la séparation la plus favorable, la complexité du tri rapide est quasi-linéaire $O(n\,log(n))$. On peut montrer que même si la séparation ne se fait pas en deux parts de taille égale, la complexité reste en moyenne de l'ordre de $O(n\,log(n))$.

Conclusion : Le tri rapide a une complexité dans le cas le plus favorable et en moyenne de l'ordre de $O(n\,log(n))$.

Le cas le plus défavorable est quand la séparation d'un tableau de taille `k` produit toujours une part vide et l'autre de taille `k-1`. Dans ce cas, le nombre de pas est `n-2` au lieu de $log_2(n)$ avant d'arriver à des parts de taille au plus 2, pour le tri manuel. Donc, dans ce cas, la complexité globale est de $O(n^2)$.

Conclusion : Le tri rapide a une complexité dans le cas le plus défavorable de l'ordre de $O(n^2)$. Le bon choix du pivot est très important pour éviter au maximum ce cas.

Remarque : Si le tableau est déjà trié, l'algorithme ne fait aucune copie.

7.3.4 Autres algorithmes de tri

Le tri direct

On a démontré mathématiquement que tout algorithme de tri basé sur des comparaisons de valeurs ne peut avoir une complexité meilleure que $O(n\,log(n))$. Il semblerait donc qu'il soit impossible de faire mieux que les algorithmes de tri évolués.

En réalité *il existe des algorithmes de tri qui font mieux !* Comment cela est-il possible ? Tout simplement parce qu'ils n'utilisent pas de comparaisons entre éléments ! L'exemple le plus connu est *le tri direct*.

Le tri direct réalise cet exploit au prix de *quelques limitations importantes*. L'algorithme ne fonctionne que pour des tableaux d'entiers ayant des valeurs dans un intervalle de taille limitée. Il peut être étendu à d'autres types de valeurs, mais il faut pour cela qu'on puisse extraire de chaque valeur un entier et que l'ordre des valeurs corresponde à l'ordre des entiers extraits.

Le tri direct utilise un espace de mémoire supplémentaire, proportionnel à la taille de l'intervalle de valeurs entières à trier (d'où la condition que cet intervalle soit de taille limitée).

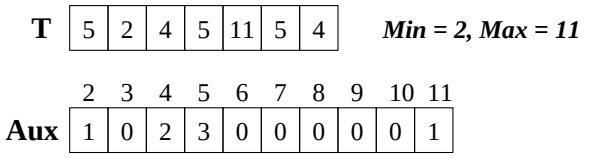


FIG. 7.6 – Exemple de tri direct

Considérons un tableau `T` contenant des entiers dans l'intervalle `[Min, Max]` donné. On a alors besoin d'un tableau auxiliaire `Aux` (voir la figure 7.6), d'indices entre `Min` et `Max`, initialisé à 0 pour tous les éléments. L'idée du tri direct est de parcourir le tableau `T` et pour chaque élément `T[i]` incrémenter la valeur de `Aux[T[i]]`. Ainsi, le tableau `Aux` compte combien de fois apparaissent dans `T` les valeurs de l'intervalle `[Min, Max]`. A la fin, on parcourt `Aux` et on écrit dans `T`, dans l'ordre, chaque *indice* de `Aux`, autant de fois que la valeur de l'élément de `Aux` pour cet indice (donc aucune fois si l'élément est 0).

Remarque : Si l'intervalle `[Min, Max]` n'est pas connu d'avance, il peut être calculé à partir des éléments du tableau, au prix d'un parcours de `T` à la recherche des éléments le plus petit et le plus grand, qui coûte `O(n)` comparaisons.

Algorithme *tri_direct*
Entrées : tableau `T`, intervalle de valeurs `[Min, Max]`
Sorties : tableau `T` trié
début
`Aux` = tableau d'indices entre `Min` et `Max`
`pour chaque indiceAux` dans `Aux répéter`

```
    Aux[indiceAux] = 0
  fin répéter
  pour chaque indiceT dans T répéter
    Aux[T[indiceT]] = Aux[T[indiceT]] + 1
  fin répéter
  indiceT = premier indice de T
  pour chaque indiceAux dans Aux répéter
    pour chaque fois entre 1 et Aux[indiceAux] répéter
      T[indiceT] = indiceAux
      indiceT = indiceT + 1
    fin répéter
  fin répéter
  retourner T
```
fin *tri_direct*

Le programme Java suivant donne une implémentation complète du tri direct, avec calcul de l'intervalle `[Min, Max]`. Une différence apparaît pour l'accès au tableau `Aux`, à cause des tableaux Java, qui ont toujours les indices à partir de 0. Voici le programme complet :

```
public class TriDirect{
  public static void main(String[] args){
    Terminal.ecrireStringln("Introduire le tableau d'entiers:");
    int[] T = lireTableau();
    int[] MinMax = calculMinMax(T);
    triDirect(T, MinMax[0], MinMax[1]);
    Terminal.ecrireStringln("Voici le tableau trie:");
    ecrireTableau(T);
  }
  static int[] lireTableau(){
    Terminal.ecrireString("Nombre d'elements:");
    int n = Terminal.lireInt();
    int[] tab = new int[n];
    for(int i=0; i<n; i++){
      Terminal.ecrireString("Element " + i + "=");
      tab[i] = Terminal.lireInt();
    }
    return tab;
  }
  static void ecrireTableau(int[] tab){
    for(int i=0; i<tab.length; i++)
      Terminal.ecrireString(" " + tab[i]);
    Terminal.sautDeLigne();
  }
  static int[] calculMinMax(int[] tab){
    int[] minmax = new int[2];
    minmax[0] = tab[0]; //minmax[0] contient le minimum
    minmax[1] = tab[0]; //minmax[1] contient le maximum
    for(int i=1; i<tab.length; i++){
      if(tab[i]<minmax[0]) minmax[0]=tab[i];
      else if(tab[i]>minmax[1]) minmax[1]=tab[i];
```

```
    }
    return minmax;
  }
  static void triDirect(int[] T, int Min, int Max){
    int[] Aux = new int[Max–Min+1]; //initialisé à 0 par défaut
    for(int it=0; it<T.length; it++)
      Aux[T[it] – Min]++;
    int indice = 0; //indice de l'endroit où mettre une valeur triée dans T
    for(int ia=Min; ia<=Max; ia++){
      for(int fois=1; fois<=Aux[ia–Min]; fois++){ //écrit ia Aux[ia–Min] fois
        T[indice] = ia;
        indice++;
      }
    }
  }
}
```

Complexité du tri direct

Si `n` est la taille du tableau `T` et `N` la taille de l'intervalle de valeurs `[Min, Max]`, alors le tri rapide réalise les opérations suivantes :

- un parcours de `T` pour remplir `Aux` et un second parcours de `T` pour y placer les résultats du tri.
- un parcours de `Aux` pour l'initialisation et un autre pour mettre dans `T` le résultat final.

Comme ici on ne manipule que des entiers, toutes les opérations ont leur importance : les modifications et comparaisons sur les compteurs de boucle, les modifications des compteurs stockés dans `Aux`, etc. Les parcours des deux tableaux, évoqués ci-dessus correspondent donc à une complexité `O(n+N)`.

Conclusion : Le tri rapide a une complexité `O(n+N)`. Il n'y a pas de cas le plus favorable ou le moins favorable, tous les cas sont traités pareil (il ne profite pas de l'éventuel tri préalable du tableau).

Remarque : Le tri rapide a besoin d'un espace supplémentaire de mémoire de l'ordre de `O(N)`.

Remarque : Rajouter le calcul initial des valeurs `Min` et `Max` ne change pas l'ordre de grandeur de la complexité de l'algorithme. Ce calcul implique un parcours du tableau `T`, avec `2(n-1)` comparaisons et au maximum `2n` copies de valeurs.