2002-2003

UFR Mathématique de la Décision

Deuxième Année de DEUG Science mention MASS

Programmation Objet II

Fabrice Rossi et Fabien Napolitano

© Fabrice Rossi et Fabien Napolitano, 2002-2003

Le code de la propriété intellectuelle interdit les copies ou reproductions destinées à une utilisation collective. Toute représentation ou reproduction intégrale ou partielle faite par quelque procédé que ce soit, sans le consentement de l'auteur ou de ses ayants cause, est illicite et constitue une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

Chapitre 1

La notion d'interface

Dans la première partie du cours nous avons vu à travers la classe `Object` et ses méthodes les prémisses d'un mécanisme très complexe appelé *héritage*. Grace à ce mécanisme nous avons pu définir des structures dynamiques `ListeChainee` et `TableauDynamique` (qui sont implantés en Java standard par les classes `LinkedList` et `ArrayList`) capables de contenir des objets de n'importe quels types. Par ailleurs nous avons vu comment, grace à la redéfinition des méthodes de la classe `Object`, il est possible de redéfinir la façon dont sont affichés et comparés les objets d'une classe : pour cela il sufffit de programmer dans la nouvelle classe des méthodes `toString` et `equals` redéfinissant les méthodes de la classe `Object`. Malheureusement la classe `Object` à elle seule ne permet pas de résoudre l'ensemble des problèmes posés par un programme complexe. En effet dans ce cadre les seules méthodes que nous puissions redéfinir sont les méthodes de `Object` et celles ci correspondent uniquement aux propriétées les plus génériques des objets. Ainsi dans de nombreux cas il est nécessaire d'utiliser des transtypages afin de pouvoir appeler des méthodes plus spécifiques, et ce mécanisme se révèle souvent assez lourd.

Les interfaces Java sont un moyen très simple permettant de résoudre ce problème de façon élégante. Grace à la notion d'interface, nous allons voir comment il est possible d'indiquer que plusieurs types d'objets différents ont des propriétés communes. Nous pourrons ainsi écrire des programmes plus abstraits et plus généraux.

1 Exemples de problèmes nécessitant des interfaces

Considérons les deux exemples indiqués dans le tableau suivant :

Objets Dessinable	Objets Comparable
Point	Integer
Cercle	String
Rectange	Double
...	...

A gauche, les classes `Point`, `Cercle` et `Rectangle` représentent des figures géométriques que nous voulons dessiner à l'écran. En particulier tous les objets de gauche sont "dessinables". A droite, les classes `Integer`, `String` et `Double` représentent des objets "comparables" : deux entiers sont comparables suivant l'ordre usuel des entiers, deux chaines de caractères suivant l'ordre lexicographique (l'ordre du dictionnaire !), etc. Bien que les objets dessinables d'une part et les objets comparables d'autre part aient des propriétés communes, il nous ait pour le moment impossible d'utiliser directement ce fait.

Supposons par exemple que les classes `Point`, `Cercle` et `Rectangle` possèdent toutes trois une méthode `void dessine()` permettant de dessiner l'objet appelant sur l'écran. Par exemple la classe `Point` se présente comme suit :

```
public class Point  {
        private double x,y;

        public Point(double a,double b) {
                x=a;
                y=b;
        }
        public void dessine() {
                // code permettant de dessiner le point
        }
        public double getX() {
                return x;
        }
        public double getY() {
                return y;
        }
}
```

Si nous voulons écrire une méthode `void dessineTab(Object[] tab)` permettant de dessiner sur l'écran tous les objets dessinables du tableau `tab`, il va nous falloir pour chaque élément de `tab` faire un test pour savoir si cet élément est de type `Point` ou de type `Cercle` ou de type `Rectangle`, puis faire un transtypage dans la classe correspondante avant d'appeler la méthode `dessine`. Par exemple nous aurons la programmation suivante :

```
public void dessineTab(Object[] tab) {
        for (int i=0;i<tab.length;i++)  {
                if (tab[i] instanceof Point) {
                        Point p=(Point)tab[i] ;
                        p.dessine ;
                }
                if (tab[i] instanceof Cercle) {
                        Cercle c=(Cercle)tab[i] ;
                        c.dessine ;
                }
                if (tab[i] instanceof Rectangle) {
                        Rectangle r=(Rectangle)tab[i] ;
                        r.dessine ;
                }
        }
}
```

Comme vous pouvez le constater : cette programmation est extrèmement lourde et les répétitions sont source d'erreurs. Par ailleurs, si nous programmons une nouvelle classe d'objets dessinables, par exemple une classe représentant les ellipses, il faudra modifier la méthode `dessineTab` en conséquence (ainsi que toutes les autres méthodes étant censées agir sur des objets

dessinables).

Dans le cas des objets comparables, nous savons que les classes `Integer`, `String`, `Double` possèdent toutes les trois une méthode `public int compareTo(Object obj)` permettant de comparer l'objet appelant avec le paramètre. Si nous voulons écrire une méthode permettant de trouver le plus petit élément d'un tableau, il nous faudra écrire trois méthodes différentes : une pour les `Integer`, une pour les `String` et une pour les `Double`. Par exemple pour les `Integer` la méthode pourrait s'écrire ainsi :

```
public static Integer minimum(Integer[] tab) {
        Integer min=tab[0] ;
        for (int i=0;i<tab.lengh;i++) {
                if (min.compareTo(tab[i])>0)
                        min=(Integer)tab[i] ;
        }
        return min ;
}
```

Dans ce cas pour programmer les méthodes permettant de trouver le minimum d'un tableau de `String` (respectivement de `Double`) il suffit de remplacer `Integer` par `String` (respectivement `Double`) dans le code précédent. Une fois de plus la recopie sera source d'erreurs. De plus si nous trouvions une méthode plus efficace pour trouver le minimum il nous faudrait reprogrammer toutes les méthodes. Une fois de plus la solution actuelle n'est pas satisfaisante.

Idéalement nous voudrions pouvoir indiquer à l'ordinateur que les classes `Point`, `Cercle` et `Rectangle` font partie d'une "classe" plus grande : la classe des objets `Dessinable`. Ainsi nous pourrions écrire directement des méthodes prenant en paramètres des objets `Dessinable`. De la même façon nous voudrions indiquer que les classes `String`, `Integer` et `Double` font partie de la "classe" plus grande des objets `Comparables`. En pratique nous allons pouvoir faire exactement cela grace à la notion d'interface. En `Java`, une interface décrit un contrat, c'est-à-dire un ensemble de méthodes qui doivent être programmées par une classe pour que celle-ci puisse affirmer remplir le contrat. Grâce à cette technique, nous allons pouvoir fournir au compilateur des informations supplémentaires sur les classes qui vont permettre d'écrire plus simplement certaines méthodes.

2 Solution du problème posé par les objets dessinable

Le contrat que remplissent les classes `Point`, `Cercle` et `Rectangle` est simple : elles implantent toutes trois une méthode `dessine`, sans paramètre ni résultat. On peut définir l'interface correspondante, appelée `Dessinable` :

Dessinable

```
1 public interface Dessinable {
2   public void dessine();
3 }
```

On remarque que la syntaxe est très proche de celle de la déclaration d'une classe. En fait, pour définir une interface, il suffit de donner le squelette d'une classe, en remplaçant le mot `class` par `interface` et le corps de chaque méthode par un point virgule.

Remarque 1. Comme pour les classes, une interface doit être déclarée dans un fichier qui porte le nom de l'interface auquel on a ajouté le suffixe `.java`. Contrairement aux classes, les interfaces ne

peuvent comporter ni variable d'instance, ni constructeur. Enfin, il est interdit de placer une paire d'accolades au contenu vide comme corps d'une méthode dans une interface : on doit impérativement remplacer celui-ci par un point virgule.

Pour des raisons d'efficacité, le compilateur ne détermine pas par lui-même si une classe vérifie le contrat précisé par une interface. Quand on souhaite qu'une classe remplisse un contrat, il faut l'indiquer explicitement. Pour ce faire, on utilise la construction `implements`, comme le montre l'exemple de la nouvelle version de la classe `Point` :

```
public class Point implements Dessinable {
        private double x,y;

        public Point(double a,double b) {
                x=a;
                y=b;
        }
        public void dessine() {
                // code permettant de dessiner le point
        }
        public double getX() {
                return x;
        }
        public double getY() {
                return y;
        }
}
```

La seule différence avec la version précédente est la première ligne, qui contient `Point implements Dessinable`. En anglais, *implements* est la conjugaison à la troisième personne du singulier du verbe *to implement*. Ce verbe se traduit par **implanter**[1]. Il signifie à peu près "programmer effectivement un ensemble de méthodes". De ce fait, `Point implements Dessinable` signifie simplement que la classe `Point` accepte de remplir le contrat décrit par `Dessinable`, c'est-à-dire qu'elle programme toutes les méthodes contenues dans cette interface (ici, la méthode `dessine`). On écrit de la même façon les classes `Cercle` et `Rectangle` afin qu'elles *implantent* l'interface `Dessinable`.

Remarque 2. Il s'agit bien ici de programmer (d'implanter) **toutes** les méthodes décrites dans l'interface, **exactement** comme elles sont décrites dans l'interface (c'est-à-dire avec exactement les mêmes types de paramètres et le même type de retour). Si ce n'est pas le cas, le contrat est rompu et le compilateur affiche un message d'erreur. Notons néammoins qu'implanter une méthode ne signifie pas utiliser dans l'implantation les mêmes noms pour les paramètres formels que ceux utilisés dans l'interface.

Remarque 3. Rien n'empêche une classe remplissant un contrat de définir des méthodes qui n'apparaissent pas dans le contrat. Ici la classe `Point` implante, en plus de la méthode `dessine`, un constructeur ainsi que des méthodes `getX` et `getY`.

Une fois programmées l'interface `Dessinable` et les classes `Point`, `Cercle` et `Rectangle` implantant `Dessinable`, nous pouvons écrire comme prévu une nouvelle méthode `dessineTab` générique :

```
public void dessineTab(Object[] tab) {
```

[1] Le verbe *implémenter* n'existe pas en français, même s'il est fréquemment utilisé dans les ouvrages d'informatique.

```
        for (int i=0;i<tab.length;i++)
                if (tab[i] instanceof Dessinable) {
                        Dessinable d=(Dessinable)tab[i] ;
                        d.dessine() ;
                }
}
```

Cette solution est clairement plus simple que la première solution. De plus, elle est *extensible*. En effet, il est possible maintenant de programmer d'autres classes implantant `Dessinable` (par exemple une classe `Ellipse`), sans avoir à modifier la méthode (contrairement à la première solution). En fait il est encore possible de simplifier cette méthode en supposant par avance que tous les éléments de tableau `tab` sont de type `Dessinable`. Dans ce cas on obtient :

```
public void dessineTab(Dessinable[] tab) {
        for (int i=0;i<tab.length;i++)
                tab[i].dessine() ;
}
```

Comme nous l'avons fait dans les chapitres précédents, il est possible de représenter les hiérarchies entre objets par un arbre. Dans le cas présent l'arbre des objets est représenté figure 1.

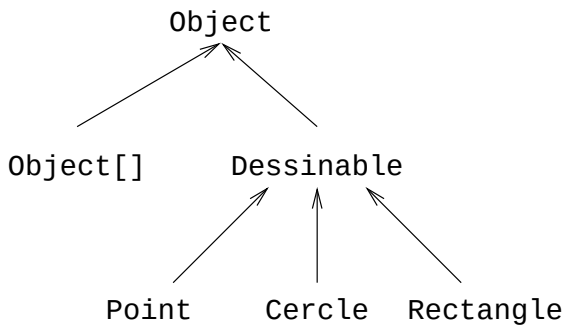


FIG. 1 – Les types dans l'application de dessin

3 Solution du problème des objets comparables

De la même façon que les classes `Cercle`, `Point` et `Rectangle` remplissent un même contrat (le contrat consistant ici à posséder une méthode `void dessine()`), les classes `String`, `Integer` et `Double` remplissent elles aussi un même contrat : elles possèdent toutes une méthode `int compareTo(Object obj)`. En fait il existe en Java une interface appelée `Comparable` définissant précisemment ce contrat. La programmation de cette interface est la suivante (attention : cette interface est déja programmée et vous n'avez pas à la réécrire) :

```
public interface Comparable {
        public int compareTo(Object obj) ;
}
```

Comme vous l'avez deviné les classes `String`, `Integer` et `Double` implantent toutes trois l'interface `Comparable` (de même que de nombreuses autres classes). Ainsi les trois méthodes `minimum` peuvent être remplacées par une seule méthode générique :

```
public static Comparable minimum(Comparable[] tab) {
        Comparable min=tab[0] ;
        for (int i=0;i<tab.lengh;i++) {
                if (min.compareTo(tab[i])>0)
                        min=(Comparable)tab[i] ;
        }
        return min ;
}
```

Dans la suite nous reviendrons sur l'interface `Comparable` et nous expliquerons en particulier comment programmer de nouvelles classes implantant cette interface.

4 Utilisation des interfaces

Voici en résumé les points importants à retenir sur l'utilisation des interfaces (la plupart de ces points sont apparus implicitement dans les sections précédentes).

Construction

Une interface n'est pas une classe : elle ne comporte pas de constructeur. On ne peut donc pas créer d'objet instance d'une interface. De ce fait, une interface s'utilise seulement comme type pour une variable, le paramètre d'une méthode ou encore le type du résultat d'une méthode.

Affectation

Les règles d'affectation sont exactement les mêmes que pour les objets usuels. Si une classe `B` implante une interface `A`, il est possible de placer une référence vers un objet de type `B` dans une variable de type `A` (et donc de renvoyer une référence vers un objet de type `B` dans une méthode qui doit renvoyer une référence de type `A`, ou encore passer un paramètre de type `B` alors qu'on attend un paramètre de type `A`). Par exemple le code suivant est correct :

```
Dessinable d1=new Point(1.5,2.5);
Dessinable d2=new Cercle(-1.2,7.4,2.5);
```

Typage (statique)

Les règles de typage statique (à la compilation) sont les mêmes que pour les objets usuels. Pour reprendre l'exemple précédent, un appel de méthode à partir de la variable `d1` considère (pour le typage) l'objet appelant comme étant de type `Dessinable`. Sont alors utilisables toutes les méthodes qui apparaissent dans `Dessinable` (ainsi que dans `Object`, comme toujours). Si la classe de l'objet définit d'autres méthodes, celle-ci ne sont pas utilisables. Le programme suivant à donc une deuxième ligne correcte et une troisième ligne refusée à la compilation :

```
Dessinable d=new Point(1.5,2.5);
d.dessine();
double u=d.getX();
```

Comme pour `Object`, il est toujours possible de revenir en arrière par transtypage (éventuellement en utilisant `instanceof` pour tester au préalable la faisabilité de la conversion). On peut donc obtenir une version correcte de la troisième ligne du programme précédent en écrivant à la place :

```
double u=((Point)d).getX();
```

Exécution (dynamique)

Comme pour `Object`, l'exécution d'une méthode fonctionne de manière dynamique. Mais le problème est ici beaucoup plus simple. Un appel de méthode par l'intermédiaire d'une variable dont le type correspond à une interface se traduit toujours par l'appel de la méthode correspondante dans la classe de l'objet appelant. Considérons l'exemple suivant :

```
Dessinable d1=new Point(1.5,2.5);
Dessinable d2=new Cercle(-1.2,7.4,2.5);
d1.dessine();
d2.dessine();
```

Les lignes 3 et 4 sont acceptées à la compilation car `d1` et `d2` sont de type `Dessinable` qui contient une méthode `dessine`. A l'exécution de la ligne 3, l'objet auquel `d1` fait référence est de type `Point` donc c'est la méthode de la classe `Point` qui est appelée. De la même façon, l'exécution de la ligne 4 se traduit par l'appel de la méthode `dessine` de la classe `Cercle`.

Transtypage et `instanceof`

Le principe est le même que pour les objets : `a instanceof type` est vrai si et seulement si une variable de type `type` peut contenir la référence `a`. Donc, si `a` fait référence par exemple à un `Point`, `a instanceof Dessinable` vaut `true`. De façon générale, si `type` correspond à une interface, `a instanceof type` vaut `true` si et seulement si `a` fait référence à un objet dont la classe implante l'interface `type`.

De plus le transtypage d'une référence ne se fait pas nécessairement vers un type définit par une classe. Ainsi peut-on écrire `(Dessinable)a`, ce qui signifie qu'on veut considérer `a` comme une référence de type `Dessinable`. Considérons par exemple le programme suivant :

```
Object o=new Point(1.0,2.0);
Dessinable d=(Dessinable)o;
d.dessine();
```

Ce programme est parfaitement correct. La deuxième ligne profite du fait que `o` fait référence à un objet dont la classe implante l'interface `Dessinable`, sans pour autant avoir besoin de connaître précisément la classe de cet objet.

5 Combinaison d'interfaces

Imaginons que certains objets `Dessinable` possèdent aussi la propriété d'être "déplaçable" : c'est à dire qu'ils possèdent une méthode de la forme `void deplace(double a,double b)`. Dans cette section nous allons voir comment indiquer au mieux à l'ordinateur cette nouvelle propriété. A priori trois possibilités se présentent :

1. Créer une nouvelle interface `Deplacable` et indiquer que certains objets sont à la fois `Deplacable` et `Dessinable`.

2. Créer une nouvelle interface `Deplacable` et une interface `DeplacableDessinable` "regroupant" les interfaces `Deplacable` et `Dessinable`.
3. Créer une nouvelle interface `DeplacableDessinable` étendant l'interface `Dessinable`.

Nous allons étudier tour à tour ces trois possibilités en essayant de montrer comment choisir une des trois dans une situation donnée (différente de l'exemple des objets déplacables et dessinables).

5.1 Implanter deux interfaces

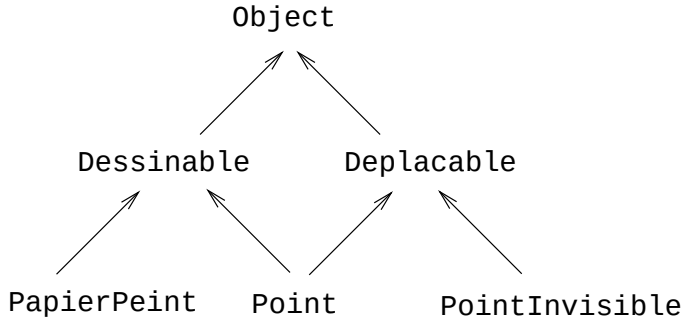
D'après ce qui précède, l'interface `Deplacable` peut s'écrire comme suit :

```
public interface Deplacable {
        public void deplace(double a,double b) ;
}
```

Pour indiquer que la classe `Point` (par exemple) correspond à un type d'objets déplaçables et dessinables il faut : l'indiquer dans l'en-tête de la classe et programmer l'ensemble des méthodes correspondant aux deux interfaces. Par exemple on aura la programmation suivante (les points indiquent les méthodes déja programmées plus haut) :

```
public class Point implements Dessinable,Deplacable {
        ...
        public void dessine() {
                // code permettant de dessiner le point
        }
        public void deplace(double a,double b) {
                x+=a ;
                y+=b ;
        }
        ....
}
```

Si l'on regarde l'arbre des objets dans ce cas on aura par exemple la figure suivante (ou nous avons ajouté les classes `PointInvisibles` et `PapierPeint` comme exemples d'objet uniquement déplacable ou uniquement dessinable) :



En particulier : l'arbre des objets n'est plus un arbre puisque `Point` a deux antécédents distincs. En pratique cela ne pose aucun problème car les interfaces ne programmant aucune méthode il n'y aura jamais d'ambiguité (même si par hazard la même signature de méthode était définie dans les interfaces `Deplacable` et `Dessinable`). En pratique un objet de type `Point` peut être considéré soit comme un objet `Deplacable` soit comme un objet `Dessinable` : les lignes suivantes sont correctes :

```
Dessinable d=new Point(1,2) ;
Deplacable e=new Point(3,4) ;
```

Nous pouvons maintenant programmer des méthodes tenant compte de cette dualité. Par exemple la méthode suivante déplace puis dessine tous les objets du tableau paramètre qui sont à la fois déplacables et dessinables :

```
public static void deplaceEtDessine(Object[] tab,double a,double b) {
        for (int i=0;i<tab.length;i++)
                if (tab[i] instanceof Deplacable &&
                    tab[i] instanceof Dessinable) {
                        Deplacable d=(Deplacable)tab[i] ;
                        d.deplace(a,b) ;
                        Dessinable e=(Dessinable)tab[i] ;
                        e.dessine() ;
                }
}
```

Cette programmation est assez lourde. En particulier il faut utiliser deux `instanceof` (il est impossible d'écrire `instanceof Deplacable,Dessinable`), et deux transtypages. Ainsi nous nous retrouvons plus ou moins au point de départ de ce chapitre et la solution n'est pas très satisfaisante. En pratique cette solution est à réserver au cas ou les deux interfaces sont conceptuellement très éloignées.

5.2 Regrouper deux interfaces dans une nouvelle interface

La deuxième solution possible prolonge la première et permet de rendre le programme plus simple. Il s'agit de créer une nouvelle interface regroupant les propriétés de `Deplacable` et `Dessinable`. Pour cela il suffit de la programmer de la façon suivante :

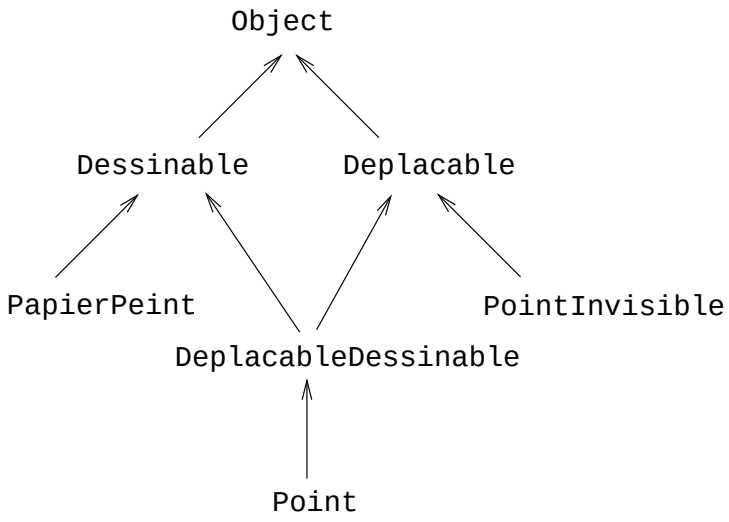
```
public interface DeplacableDessinable extends Deplacable,Dessinable {
}
```

Le mot clef `extends` indique à l'ordinateur que l'interface `DeplacableDessinable` possède toutes les propriétées des deux interfaces `Deplacable` et `Dessinable`. En particulier une classe implantant cette interface devra programmer toutes les méthodes définies dans `Deplacable` et `Dessinable`. En fait nous verrons dans la troisième solution que l'on peut également ajouter de nouvelles méthodes. Dans la programmation de la classe `Point` il suffit maintenant d'indiquer que la type `Point` implante l'interface combinée `DeplacableDessinable` :

```
public class Point implements DeplacableDessinable {
        ...
        public void dessine() {
                // code permettant de dessiner le point
        }
```

```
        public void deplace(double a,double b) {
                x+=a ;
                y+=b ;
        }
        ....
}
```

L'arbre des objets est représenté sur la figure suivante :



En utilisant la nouvelle interface combinée, il est maintenant possible de donner une programmation plus simple de `deplaceEtDessine` :

```
public static void deplaceEtDessine(Object[] tab,double a,double b) {
        for (int i=0;i<tab.length;i++)
                if (tab[i] instanceof DeplacableDessinable) {
                        DeplacableDessinable d=(DeplacableDessinable)tab[i] ;
                        d.deplace(a,b) ;
                        d.dessine() ;
                }
}
```

On peut même encore simplifier le programme en supposant dés le début que tous les objets de `tab` sont déplaçables et dessinables. Dans ce cas la méthode devient simplement :

```
public static void deplaceEtDessine(DeplacableDessinable[] tab,double a,double b) {
        for (int i=0;i<tab.length;i++) {
                tab[i].deplace(a,b) ;
                tab[i].dessine() ;
        }
}
```

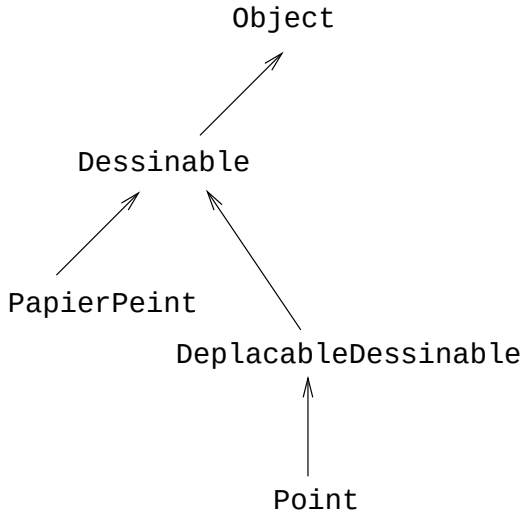
En général cette deuxième solution est toujours meilleure que la première quand les interfaces combinées sont conceptuellement proches car dans ce cas de nombreuses méthodes seront amenées à utiliser l'interface combinée.

5.3 Etendre une interface

Imaginons que tous les objets déplaçables soient nécessairement dessinables. Alors il n'existe que deux types d'objets possibles (en relation avec le problème initial !) : d'une part les objets déplaçables et dessinables, d'autre part les objets dessinables mais pas déplacables (en particulier la classe `PointInvisible` n'existe plus dans ce cas). Dans ces conditions il est absurde de créer une interface correspondant uniquement aux objets déplaçable puisque ceux ci n'existent jamais indépendamment des objets dessinables. La solution consiste à créer l'interface `DeplacableDessinable` en étendant les propriétées de `Dessinable` de la façon suivante :

```
public interface DeplacableDessinable extends Dessinable {
        public void deplace(double a,double b) ;
}
```

Les objets `DeplacableDessinable` devront tous posséder à la fois une méthode `dessine` et une méthode `deplace`. Dans ce cas les programmations de la classe `Point` et de la méthode `deplaceEtDessine` restent inchangées par rapport à la deuxième solution. Par contre l'arbre des objets est à nouveau un arbre comme vous pouvez le voir sur la figure suivante :



Cette solution doit être privilégiée dans tous les cas où une propriété en implique une autre (ici déplacable implique dessinable).

Chapitre 2

L'exemple des fonctions

Dans les sections précédentes, nous avons vu comment définir et implanter de nouvelles interfaces. Les interfaces que nous avons définies permettaient de regrouper plusieurs classes différentes en considérant l'ensemble de leur propriété communes. Dans cette section nous allons voir comment utiliser les interfaces afin de représenter des algorithmes en utilisant des objets. Pour cela nous utiliserons l'exemple des fonctions mathématiques.

Considérons une fonction arbitraire $x \mapsto f(x)$. On se propose de trouver une racine de l'équation $f(x) = 0$ au voisinnage d'un point x_0 (estimation iniitiale de la racine). L'un des algorithmes les plus efficaces pour effectuer cette recherche est l'algorithme de Newton (figure 1). On commence par calculer $f(x_0)$ puis on suit la tangente à la courbe $(x, f(x))$ au point $(x_0, f(x_0))$ vers l'axe des abscisses. L'intersection de la tangente et de l'axe des abscisses est un nouveau point x_1 dont on constate facilement qu'il est plus proche de la racine de l'équation que x_0. A partir de x_1 on calcul un nouveau point x_2 par le même procédé, et ainsi de suite jusqu'à obtenir une approximation suffisamment bonne de la racine.

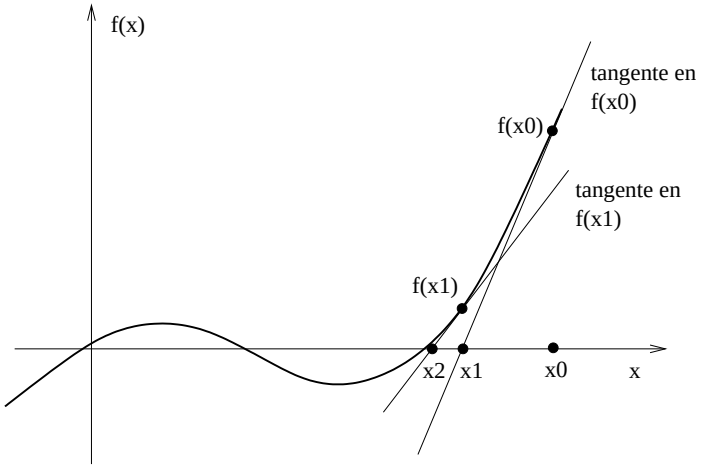


FIG. 1 – Algorithme de Newton

Il suffit donc à chaque étape de remplacer x_0 par $x_0 - \frac{f(x_0)}{f'(x_0)}$ jusqu'à ce qu'une condition d'arrêt soit satisfaite ; par exemple $|f(x_0)| < \epsilon$.

Considérons pour commencer le cas de la fonction cosinus. Dans ce cas l'algorithme de Newton se programme de la façon suivante :

```
public static double newtonCosinus(double x0,double epsilon) {
        while (Math.abs(Math.cos(x0))>=epsilon) {
                x0=x0-Math.cos(x0)/(-Math.sin(x0)) ;
        }
        return x0 ;
}
```

Bien que ce programme soit correct il souffre d'un défaut majeur : nous avons été obligé de coder directement dans le programme la fonction cosinus dont on veut trouver la racine. Pour trouver la racine d'une autre fonction, il faut reprogrammer l'algorithme de Newton.

Dans cette partie nous allons définir une nouvelle interface `Fonction` permettant de représenter la notion de fonction de variable réelle et d'utiliser les algorithmes classiques :

- calcul d'intégrales numériques
- recherche de zeros (par dichotomie et par Newton)
- calcul symbolique

1 L'interface Fonction

Une fonction est un algorithme associant à un nombre réel un autre nombre réel : la valeur de la fonction au premier point. L'interface `Fonction` peut donc s'écrire :

```
public interface Fonction {
        public double eval(double x) ;
}
```

Etant donné cette interface nous pouvons maintenant programmer des classes représentant des fonctions. Par exemple pour le cosinus on a :

```
public class Cosinus implements Fonction {
        public double eval(double x) {
                return Math.cos(x) ;
        }
}
```

Pour utiliser notre classe `Cosinus` il suffit de construire une nouvelle instance de `Cosinus` grace au consttructeur sans paramètre puis d'appeler sa méthode `eval`. Par exemple le programme suivant affiche `1.0` :

```
                Fonction cos=new Cosinus() ;
                System.out.println(cos.eval(0)) ;
```

Bien sur nous pouvons programmer d'autres classes implentant l'interface `Fonction` ayant un comportement plus complexe et en particulier ajoutant de nouvelles méthodes ou variables d'instance. Par exemple pour les polynomes on peut écrire :

```
public class Polynome implements Fonction {
        private double[] coefficients ;

        public Polynome(double[] coefficients) {
                this.coefficients=coefficients ;
        }
        public double eval(double x) {
```

```
                double resultat=0 ;
                for (int i=coefficients.length-1;i>0;i--) {
                        resultat+=coefficients[i] ;
                        resultat*=x ;
                }
                resultat+=coefficients[0] ;
                return resultat ;
        }
        /* renvoie la somme du polynome appelant et du parametre */
        public Polynome somme(Polynome p) {
                int n;
                if (coefficients.length<p.coefficients.length)
                        n=p.coefficients.length ;
                else
                        n=coefficients.length ;
                double[] tab=new double[n] ;
                for (int i=0;i<n;i++) {
                        if (i<coefficients.length)
                                tab[i]+=coefficients[i] ;
                        if (i<p.coefficients.length)
                                tab[i]+=p.coefficients[i] ;
                }
                return new Polynome(tab) ;
        }
}
```

Pour calculer la valeur du polynome en un point nous utilisons la méthode de Hörner permettant à la fois d'améliorer la précision du résultat et de réduire le nombre de multiplications nécessaires. Cette méthode est basé sur l'identité suivante :

$$a_k x^k + a_{k-1}x^{k-1} + +a_{k-2}x^{k-2} + \cdots + a_0 = (\dots((a_k x + a_{k-1})x + a_{k-2})\dots) + a_0$$

Evidemment on peut s'inspirer de ces exemples afin de programmer des classes représentant l'ensemble des fonctions usuelles : sinus, exponentielle, valeur absolue, fonction affine, ... Il faut toutefois faire attention car il est facile de programmer des classes implentant l'interface `Fonction` mais ne représentant pas réellement des fonctions de variables réelles. Par exemple la fonction suivante représente un bruit :

```
import java.util.* ;
public class Bruit implements Fonction {
        public static Random alea=new Random() ;

        public double eval(double x) {
                return alea.nextDouble() ;
        }
}
```

Bien qu'une telle classes puisse être utile, elle ne correspond pas à la représentation que nous nous faisons de l'interface `Fonction`. Il faut donc toujours bien distinguer la représentation mentale d'une interface et sa définition informatique.

2 Calcul d'intégrales numériques

La méthode la plus simple permettant de calculer numériquement l'intégrale d'une fonction sur un intervale $[a, b]$ est l'intégration par les rectangles : on divise l'intervalle $[a, b]$ en n intervales égaux et on remplace la fonction sur chacun de ses intervalles par un rectangle, enfin on calcul la somme des aires des rectangles (figure 2).

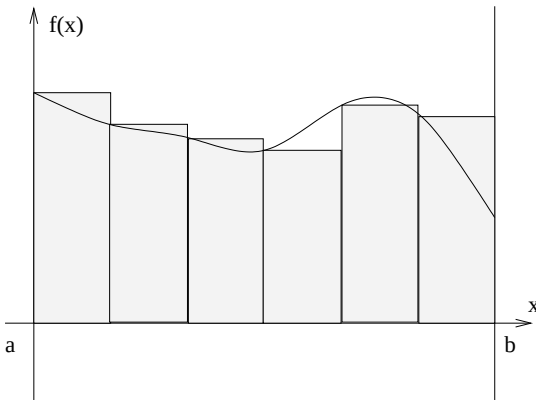


FIG. 2 – Intégrale par la méthode des rectangles ($n = 6$)

Cet algorithme se programme comme suit :

```
public static double integraleRectangle(Fonction f,double a,double b,int n) {
        double resultat=0. ;
        for (int i=0;i<n;i++)
                resultat+=(b-a)/n*f.eval(a+i*(b-a)/n) ;
        return resultat ;
}
```

Comme vous pouvez le constater la méthode `integraleRectangle` prend en paramètre une `Fonction`. En particulier elle est applicable à n'importe quelle classe implentant l'interface `Fonction`. Par exemple le programme suivant affiche `1.0078334198735825` (environ égal à $1 = \int_0^{\frac{\pi}{2}} \cos(t)\,\mathrm{dt}$) :

```
        Cosinus f=new Cosinus() ;
        System.out.println(integraleRectangle(f,0,Math.PI/2,100)) ;
```

Bien sur il existe d'autres méthodes permettant de calculer l'intégrale d'une fonction. Par exemple la méthode des trapèzes consiste à remplacer la fonction sur chaque petit interval par un trapèze et non plus un rectangle :

```
public static double integraleTrapeze(Fonction f,double a,double b,int n) {
        double resultat=0. ;
        for (int i=0;i<n;i++)
                resultat+=(b-a)/n*(f.eval(a+i*(b-a)/n)+f.eval(a+(i+1)*(b-a)/n))/2 ;
        return resultat ;
}
```

Cette méthode est légèrement plus précise. Par exemple le programme suivant affiche `0.9999794382396078` :

```
        Fonction f=new Cosinus() ;
        System.out.println(integraleTrapeze(f,0,Math.PI/2,100)) ;
```

En pratique il est parfois difficile de choisir entre les différentes méthodes. Pour contourner ce problème on peut définir une interface correspondant à l'algorithme de calcul d'une intégrale numérique, puis écrire des classes implentant cette interface et correspondant aux différents algorithmes d'intégration possible. On est ainsi amener à programmer les classes suivantes :

```
public interface  Integrateur {
/* interface représentant un schéma d'intégration numérique */
        public double  integre(Fonction f,double a,double b,int n) ;
}

public class IntRectangle implements Integrateur {

        public double integre(Fonction f,double a,double b,int n) {
                double resultat=0. ;
                for (int i=0;i<n;i++)
                        resultat+=(b-a)/n*f.eval(a+i*(b-a)/n) ;
                return resultat ;
        }
}

public class IntTrapeze implements Integrateur {
/* integrale par la méthode des trapèzes */
        public double integre(Fonction f,double a,double b,int n) {
                double resultat=0. ;
                for (int i=0;i<n;i++)
                        resultat+=
                                (b-a)/n
                                *(f.eval(a+i*(b-a)/n)+f.eval(a+(i+1)*(b-a)/n))/2 ;
                return resultat ;
        }
}
```

Pour intégrer une fonction avec la méthode des trapèzes on pourra maintenant écrire :

```
        Integrateur trapeze=new IntTrapeze() ;
        Fonction f=new Cosinus() ;
        System.out.println(trapeze.integre(f,0,Math.PI/2,100) ;
```

On peut donc avoir des interfaces dont les méthodes prennent en paramètres d'autres interfaces et ainsi avoir des algorithmes agissant sur d'autres algorithmes.

3 Recherche de racines

Etant donné une fonction f sur un intervalle $[a, b]$ telle que $f(a)f(b) < 0$ il existe d'après le théorème des valeurs intermédiaires une racine de l'équation $f(x) = 0$ sur $[a, b]$. Pour trouver cette racine on peut utiliser la dichotomie : soit c le milieu de $[a, b]$, si $f(c)f(a) < 0$ remplacer b par c sinon remplacer a par c. L'algorithme d'arrête quand l'intervale $[a, b]$ est plus petit qu'une valeur ϵ choisie au départ (la précision avec laquelle on cherche la racine). Cette algorithme peut se programmer comme suit :

```
public static double dichotomie(Fonction f,double a,double b,double epsilon) {
        if (f.eval(a)*f.eval(b)>0) return Double.NaN ;
        while ((b-a)>epsilon) {
                double c=(a+b)/2 ;
                if (f.eval(a)*f.eval(c)<0)
                        b=c ;
                else
                        a=c ;
        }
        return (a+b)/2 ;
}
```

Bien sur cet méthode fonctionne avec n'importe quelle fonction vérifiant l'hypothèse. Par exemple le programme suivant trouve la racine de cosinus sur $[0, \pi]$ (et affiche : `1.571179821991868`) :

```
Fonction f=new Cosinus() ;
System.out.println(dichotomie(f,0,Math.PI,0.001)) ;
```

Malheureusement nous sommes toujours incapable de résoudre un des problèmes posés au départ : la recherche de racines par la méthode de Newton. En effet pour appliquer la méthode de Newton nous devons absolument pouvoir calculer la dérivée de la fonction. Une solution possible consisterait à calculer la dérivée $f'(x)$ numériquement en l'approximant par $\frac{f(x-\epsilon)-f(x+\epsilon)}{2\epsilon}$ où ϵ est un petit paramètre. Néammoins il n'est pas toujous évident de choisir une bonne valeur pour ϵ. Nous allons voir dans la prochaine section qu'il existe une solution beaucoup plus élégante basée sur les interfaces.

4 Calcul formel : l'interface FonctionDerivable

Les dérivées de la plupart des fonctions sont connues ou peuvent être trouvée facilement en utilisant les identitées :

$$(f+g)' = f' + g'$$
$$(f \times g)' = f' \times g + f \times g'$$
$$(f \circ g)' = g' \times f' \circ g$$

Dans cette partie nous allons voir comment utiliser ce type d'identitées pour calculer symboliquement la dérivée d'une fonction. Pour cela nous allons utiliser l'interface `FonctionDerivable` :

```
public interface FonctionDerivable extends Fonction {
        public FonctionDerivable derive() ;
}
```

Le but de la méthode `derive()` est évidemment de renvoyer la dérivée de la fonction appelante. En particulier l'interface `FonctionDerivable` représente les fonctions infiniment dérivables. Etant donné que toute fonction dérivable est une fonction, nous avons choisi d'étendre l'interface `Fonction` (et non pas de passer par une interface intermédiaire `Derivable`). En particulier toute classe implantant `FonctionDerivable` doit proposer une méthode `eval`.

Nous pouvons maintenant programmer des classes implantant cette interface. Imaginons par exemple que nous souhaitions représenter sinus et cosinus comme des fonction dérivables. Dans ce cas nous devons utiliser les formules :

$$sin' = cos$$
$$cos' = -sin$$

La deuxième formule pose problème : en effet nous devons être capable de représenter informatiquement l'opposé d'une fonction dérivable afin de pouvoir programmer cette formule. Pour cela nous commençons par écrire une classe `Oppose` représentant l'opposé d'une fonction dérivable :

```
public class Oppose implements FonctionDerivable {
        private FonctionDerivable f ;

        public Oppose(FonctionDerivable f) {
                this.f=f ;
        }
        public double eval(double x) {
                return -f.eval(x) ;
        }
        public FonctionDerivable derive() {
                return new Oppose(f.derive()) ;
        }
        public String toString() {
                return "-"+f ;
        }
}
```

Grâce à cette classe nous pouvons réécrire les classes `Cosinus` et `Sinus` afin qu'elles implantent `FonctionDerivable` :

```
public class Cosinus implements FonctionDerivable {
        private static FonctionDerivable derivee=new Oppose(new Sinus()) ;

        public double eval(double x) {
                return Math.cos(x) ;
        }
        public FonctionDerivable derive() {
                return derivee ;
        }
        public String toString() {
                return "cos" ;
        }
}

public class Sinus implements FonctionDerivable {
        private static FonctionDerivable derivee=new Cosinus() ;

        public double eval(double x) {
                return Math.sin(x) ;
        }
        public FonctionDerivable derive() {
                return derivee ;
        }
        public String toString() {
                return "sin" ;
        }
}
```

Pour essayer notre programme nous écrivons :

```
FonctionDerivable c=new Cosinus() ;
System.out.println(c.derive()) ;
```

On vérifie que ce programme affiche bien `-sin` (gràce aux méthodes `toString` que nous avons reprogrammées). Vous remarquerez que dans les classes `Cosinus` et `Sinus` on utilise une variable de classe pour la derivée afin d'éviter de la recalculer à chaque fois. Essayer de comprendre ce qui se passe dans la mémoire de l'ordinateur.

De la même façon que nous avons programmée une classe `Oppose`, nous pouvons programmer des classes `Somme` et `Produit` représentant la somme et le produit de fonction dérivable :

```
public class Somme implements FonctionDerivable {
        private FonctionDerivable f ;
        private FonctionDerivable g ;

        public Somme(FonctionDerivable f,FonctionDerivable g) {
                this.f=f ;
                this.g=g ;
        }
        public double eval(double x) {
                return f.eval(x)+g.eval(x) ;
        }
        public FonctionDerivable derive() {
                return new Somme(f.derive(),g.derive()) ;
        }
        public String toString() {
                return f+"+"+g ;
        }
}

public class Produit implements FonctionDerivable {
        private FonctionDerivable f ;
        private FonctionDerivable g ;

        public Produit(FonctionDerivable f,FonctionDerivable g) {
                this.f=f ;
                this.g=g ;
        }
        public double eval(double x) {
                return f.eval(x)*g.eval(x) ;
        }
        public FonctionDerivable derive() {
                return new Somme(
                        new Produit(f.derive(),g),
                        new Produit(f,g.derive())) ;
        }
        public String toString() {
                return f+"*"+g ;
        }
}
```

Le programme suivant :

```
FonctionDerivable c=new Cosinus() ;
FonctionDerivable s=new Sinus() ;
FonctionDerivable p=new Produit(c,s) ;
System.out.println("la derivee de "+p+" est "+p.derive()) ;
```

affiche bien :

```
la derivee de cos*sin est -sin*sin+cos*cos
```

Afin de calculer formellement la dérivée de n'importe quelle fonction définie à partir des fonctions élémentaires (cosinus,sinus,exponentielle,...), il nous suffit de programmer chacune des fonctions élémentaires et d'ajouter une classe représentant la composée de deux fonctions.

5 La méthode de Newton

Grace à l'interface `FonctionDerivable` nous pouvons programmer une méthode de Newton générique (applicable à n'importe quelle fonction dérivable) :

```
public static double newton(FonctionDerivable f,double x0,double epsilon) {
        FonctionDerivable derivee=f.derive() ;
        while (Math.abs(f.eval(x0))>=epsilon) {
                x0=x0-f.eval(x0)/derivee.eval(x0) ;
        }
        return x0 ;
}
```

PROJET I

Automates cellulaires

Le but de ce projet est de programmer un ensemble de classes permettant de représenter de très nombreux *automates cellulaires*. Les automates cellulaires sont une version informatique des bactéries. Chaque automate se dévellope en fonction de son environnement immédiat en suivant quelques règles très simples. De nombreux automates étant placés dans une même boite, on observe d'étranges comportement collectifs. Le premier automate cellulaire a été inventé au début des années 70 par John Conway ; il est aujourd'hui célèbre sous le nom de "jeu de la vie". Quelques exemples de figures obtenues avec des automates cellulaires sont inclues dans le texte pour vous donner une idée des résultats obtenues. Bien sur il est beaucoup plus intéressant d'observer l'évolution des automates plutot que le résultat à un instant donné et il vous reste donc beaucoup de choses à voir. Toutes les règles de transformations données en exemple sont issues du site Web de Mirek Wojtowicz dont l'adresse est **www.mirxoj.opus.chelm.pl/ca/index.html**. Si les pages suivantes vous intéressent il vous est très vivement conseillé de vous connecter sur ce site pour obtenir des informations supplémentaires.

1 La classe Automate

La classe `Automate` dont le squelette est donné table 1 permet de représenter informatiquement un automate cellulaire. Chaque automate est caractèrisé par deux entiers : son état actuel et l'état dans lequel il va se transformer dans le futur, et par la liste de ses voisins. La méthode `actualise` remplace l'état actuel par l'état futur. Programmez la classe `Automate`.

```
import java.util.* ;
public class Automate {
        public int etatActuel ;
        public int etatFutur ;
        public LinkedList voisins ;

        public Automate() ;

        public Automate(int etatActuel) ;

        public void actualise() ;
}
```

Tab. 1 – Squelette de la classe Automate

2 L'interface Regle

Afin de faire évoluer l'ensemble des automates cellulaires contenus dans une boite il faut disposer d'une règle de transformation des automates. Chaque règle de transformation sera dans la suite une classe implantant l'interface `Regle` donnée table 2.

```
public interface Regle {
        public void calculEtatFutur(Automate cellule) ;
        public int rayon() ;
        public int couleurs() ;
}
```

TAB. 2 – L'interface Regle

```
import java.util.* ;
public class Boite {
/* classe représentant une boite remplie d'automates cellulaires */

        private Automate[][] contenu ;
        /* regle de transformation des automates cellulaires */
        private Regle regle ;
        /* generateur de nombres aléatoires */
        public static Random alea=new Random() ;

        /* construit une boite de largeur et longueur donnée */
        public Boite(int largeur,int longueur,Regle regle) ;

        public int largeur() ;
        public int longueur() ;

        /* renvoie l'automate contenu dans la case (i,j) */
        public Automate get(int i,int j) ;
}
```

TAB. 3 – Squelette de la classe Boite

Dans une classe implantant `Regle`, la méthode `calculEtatFutur` permet de calculer l'état de l'automate à la prochaine étape, en fonction de son environnement et de son état actuel. La méthode `int rayon()` renvoie la taille du voisinnage d'un automate cellulaire (c'est à dire grossièrement la distance maximale à laquelle il peut être influencé par un voisin). La méthode `int couleurs()` renvoie le nombre d'états possibles pour un automate.

3 La classe Boite

La classe `Boite` dont le squelette est donné table 3 représente une boite rectangulaire contenant des automates. Les méthodes de cette classe permettent de créer une nouvelle boite remplie d'automates dont les états sont choisis au hazard, puis de faire évoluer l'ensemble des automates suivants une règle donnée.

1. Programmer le constructeur, les méthodes `largeur()` et `longueur()` et la méthode `get(int i,int j)`. Le constructeur initialisera le tableau contenant les automates. Pour le moment on supposera que l'état initial de chaque automate est 0.
2. Soit r un entier plus grand que 0. Le *voisinnage de taille* r d'un automate de coordonnees (a, b) dans le tableau `contenu` est l'ensemble des automates dont les coordonnees (i, j) satisfont

$|i-a| \leq r$, $|j-a| \leq r$ et $(i,j) \neq (a,b)$ ($|x|$ désigne la valeur absolue du nombre x). La dernière inégalité signifie qu'une automate ne fait pas partie de son voisinnage. Ecrire une méthode :

```
LinkedList getVoisinnage(int x,int y,int rayon) ;
```

renvoyant la liste des automates composant le voisinnage de taille *rayon* de l'automate de coordonnées (x, y). En programmant cette méthode vous ferez attention à ce qu'elle ne provoque jamais d'erreur quelle que soit la valeur de (x, y) (pensez au cas ou $(x, y) = (0, 0)$ par exemple).

3. Au début les automates ne savent pas quels sont leurs voisins car leur variable `voisins` n'est pas encore initialisée. Or un automate ne peut pas évoluer si il ne connait pas ses voisins. Ecrire une méthode :

   ```
   void initialiseVoisins() ;
   ```

   permettant de donner une valeur aux variables `voisins` des automates du tableau `contenu`. Plus précisemment la méthode commencera par obtenir la distance maximale `rayon` à laquelle les automates peuvent agir les uns sur les autres grace à la méthode `rayon()` de l'interface `Regle`. Puis elle remplacera la variable `voisins` de chaque automate par la liste des automates de son voisinage de taille `rayon`.

4. Afin d'obtenir une évolution intéressante il est nécessaire que les états des automates dans la boite soit choisis au hazard avant de commencer. Comme on l'a vu la méthode `couleurs()` d'une instance de `Regle` renvoie le nombre d'états n autorisés pour les automates d'après cette règle. Ecrire une méthode :

   ```
   void repartitionAleatoire() ;
   ```

   permettant de choisir au hazard l'état des automates de la boite. Plus précisemment cette méthode parcourt l'ensemble des automates du tableau `contenu` et remplace l'état actuel de chaque automate par un entier tiré au hazard entre 0 et n compris (n étant le nombre maximal d'états d'après la variable d'instance `regle`).

5. Modifier le constructeur de la classe `Boite` en ajoutant un appel aux méthodes `repartitionAleatoire()` et `initialiseVoisins()`.

6. La méthode `void evolution()` de la classe `Boite` permet de faire évoluer l'ensemble des automates de la boite suivant une règle donnée. Pour cela la méthode `evolution()` commence par parcourir l'ensemble des automates du tableau `contenu` et calcul l'état future de chaque automate grace à la méthode `calculEtatFuture` de `regle`. Une fois l'état future de chaque automate calculé, la méthode parcourt à nouveau le tableau `contenu` et remplace l'état actuel de chaque automate par son état future en utilisant la méthode `actualise()` de la classe `Automate`. Programmer la méthode `evolution`.

4 Règles de transformation de type Life

La classe `Life` représente une famille de règles d'évolution pour les automates. En particulier elle implante l'interface `Regle`.

Le constructeur de la classe `Life` prend en paramètres deux tableaux d'entiers : `int[] survie` et `int[] naissance`. Le nombre d'état maximal pour la règle `Life` est 1. La distance maximale a laquelle les automates peuvent agir les unes sur les autres dans la regle `Life` est également 1.

1. Ecrire le constructeur de la classe `Life` ainsi que les méthodes `rayon()` et `couleurs()`. Les tableaux `survie` et `naissance` doivent être stockés en variables d'instance.

2. L'état futur d'un automate d'après la règle `Life` est calculé comme suit :
   - soit n le nombre de voisins de l'automate dont l'état est 1.
   - si l'état actuel de l'automate est 0
     - si n appartient au tableau `naissance` alors l'état future de l'automate est 1.
     - sinon l'état future de l'automate est 0.
   - si l'état actuel de l'automate est 1
     - si n appartient au tableau `survie` alors l'état future de l'automate est 1.
     - sinon l'état future de l'automate est 0.

   Programmer la méthode `calculEtatFuture(Automate cellule)` de la classe `Life`.

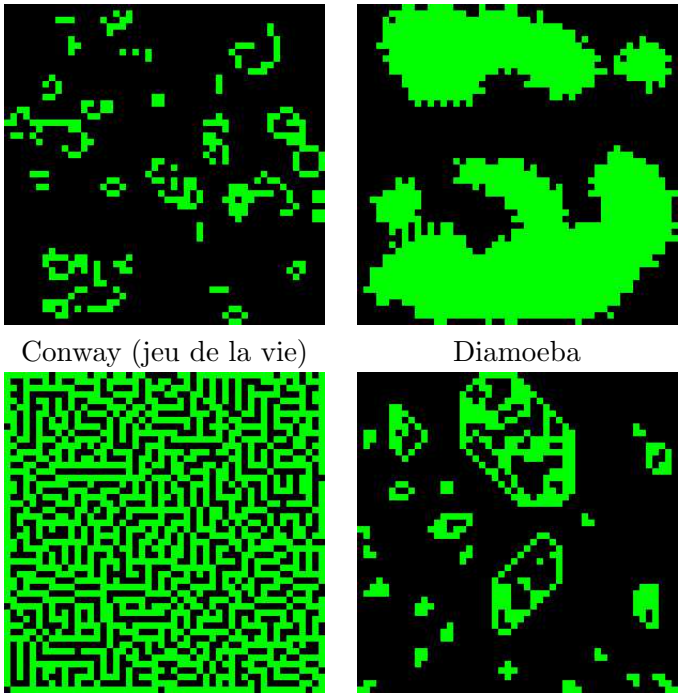
Lorsque vous creer une nouvelle instance de la classe `Life` il faut choisir avec attention les paramètres `survie` et `naissance`. Le tableau 4 vous donnera un certain nombre d'idées intéressantes.

nom de la règle	survie	naissance
2x2	1,2,5	3,6
34Life	3,4	3,4
Amoeba	1,3,5,8	3,5,7
Assimilation	4,5,6,7	3,4,5
Coagulations	2,3,5,6,7,8	3,7,8
Conway	2,3	3
Coral	4,5,6,7,8	3
DayAndNight	3,4,6,7,8	3,6,7,8
Diamoeba	5,6,7,8	3,5,6,7,8
Flakes	0,1,2,3,4,5,6,7,8	3
Gnarl	1	1
HighLife	2,3	3,6
InverseLife	3,4,6,7,8	0,1,2,3,4,7,8
LongLife	5	3,4,5
Maze	1,2,3,4,5	3
Mazectric	1,2,3,4	3
Move	2,4,5	3,6,8
PseudoLife	2,3,8	3,5,7
Replicator	1,3,5,7	1,3,5,7
Seeds		2
Serviettes		2,3,4
Stains	2,3,5,6,7,8	3,6,7,8
WalledCities	2,3,4,5	4,5,6,7,8

TAB. 4 – Quelques formes intéressantes de la règle Life

5 Partie graphique

Programmez une classe `PanelBoite` permettant de dessiner une boite contenant des automates cellulaires. Chaque automate devra être représenté comme un petit rectangle dont la couleur sera déterminé par l'état actuel de l'automate. Vous programmerez également une classe `EcouteurClavierBoite` permettant à l'utilisateur de faire évoluer l'ensemble des automates contenus dans la boite en appuyant sur la touche espace (bien sur si vous connaissez les `Thread` il faut bien mieux en utiliser un pour faire évoluer régulièrement les automates).



TAB. 5 – Exemple d'évolutions d'automates de type Life

Pour essayer votre programme choisissez une des règles de type `Life` donnée dans la table 4 puis créez une boite de taille 50×50 et enfin appelez la méthode `main` de la classe `PanelBoite`. Maintenant appuyez sur espace et regardez.

6 Règles de transformations de type Generations

La classe `Generations` représente une autre famille de règle d'évolution. Cette règle est très similaire à `Life`, en particulier on retrouve les tableaux `survie` et `naissance` mais comporte un paramètre supplémentaire permettant d'obtenir de nombreux effets très intéressants : la durée de vie maximale. Ce paramètre est un entier devant être passé en paramètre au constructeur. A chaque étape on applique les mêmes règles de transformations que pour la classe `Life` avec les différences suivantes : un automate qui vient de naître est dans l'état 0, un automate qui ne vient pas de naitre et ne survit pas, ne disparait pas immédiatement mais vieilli (son `etatActuel` est augmenté de 1). Bien sur tout automate dont l'état actuel dépasse la durée de vie maximale meurt et son état revient à 0.

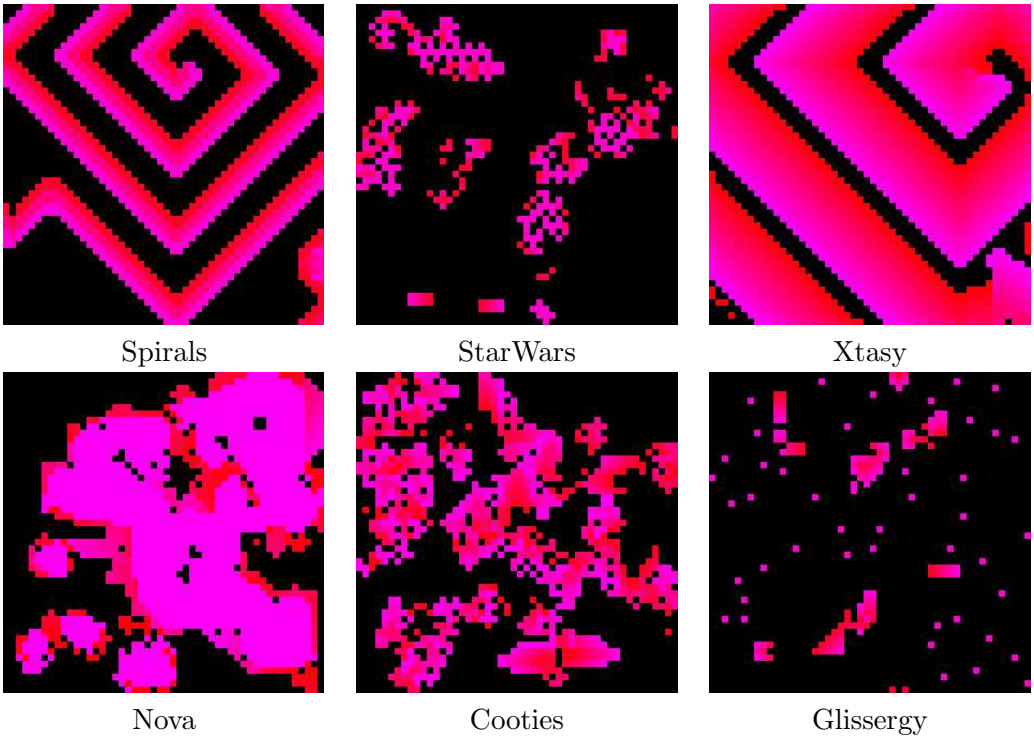
Le tableau 6 présente quelques paramètres intéressant pour les règles de type `Generations`.

7 Règles de type Cyclic

La classe `Cyclic` représente un troisième type de règles d'évolution possible. Chaque règle de type `Cyclic` possède trois paramètres `R`, `S`, `C`. Le paramètre `R`, le rayon, donne la taille des voisinnages que l'on doit considérer pour l'évolution de chaque cellule. Le paramètre `S` est appelé le seuil. Le paramètre `C`, appelé aussi couleur, indique le nombre d'état maximal. Les états sont organisés cycliquement c'est à dire que l'état suivant `C` est 0 (bien sur l'état suivant 0 est 1 etc).

nom de la règle	survie	naissance	durée de vie maximale
Banners	2,3,6,7	3,4,5,7	5
BelZhab	2,3	2,3	8
BelZhabSediment	1,4,5,6,7,8	2,3	8
Bloomerang	2,3,4	3,4,6,7,8	24
Bombers	3,4,5	2,4	25
Brain6	6	2,4,6	3
BriansBrain		2	3
Burst	0,2,3,5,6,7,8	3,4,6,8	9
BurstII	2,3,5,6,7,8	3,4,6,8	9
Caterpillars	1,2,4,5,6,7	3,7,8	4
Chenille	0,5,6,7,8	2,4,5,6,7	6
CircuitGenesis	2,3,4,5	1,2,3,4	8
Cooties	2,3	2	8
EbbAndFlow	0,1,2,4,7,8	3,6	18
EbbAndFlowII	0,1,2,4,6,8	3,7	18
Faders	2	2	25
Fireworks	2	1,3	21
FlamingStarbows	3,4,7	2,3	8
Frogs	1,2	3,4	3
FrozenSpirals	3,5,6	2,3	6
Glisserati	0,3,5,6,7	2,4,5,6,7,8	7
Glissergy	0,3,5,6,7,8	2,4,5,6,7,8	5
Lava	1,2,3,4,5	4,5,6,7,8	8
Lines	0,1,2,3,4,5	4,5,8	3
LivingOnTheEdge	3,4,5	3	6
MeteorGuns	0,1,2,4,5,6,7,8	3	8
Nova	4,5,6,7,8	2,4,7,8	25
OrthoGo	3	2	4
PrairieOnfire	3,4,5	3,4	6
RainZha	2	2,3	8
Rake	3,4,6,7	2,6,7,8	6
SediMental	4,5,6,7,8	2,5,6,7,8	4
Snake	0,3,4,6,7	2,5	6
SoftFreeze	1,3,4,5,8	3,8	6
Spirals	2	2,3,4	5
StarWars	3,4,5	2	4
Sticks	3,4,5,6	2	6
Swirl	2,3	3,4	8
ThrillGrill	1,2,3,4	3,4	48
Transers	3,4,5	2,6	5
TransersII	0,3,4,5	2,6	6
Wanderers	3,4,5	3,4,6,7,8	5
Worms	3,4,6,7	2,5	6
Xtasy	1,4,5,6	2,3,5,6	16

TAB. 6 – Exemples de paramètres pour les règles de type Generations

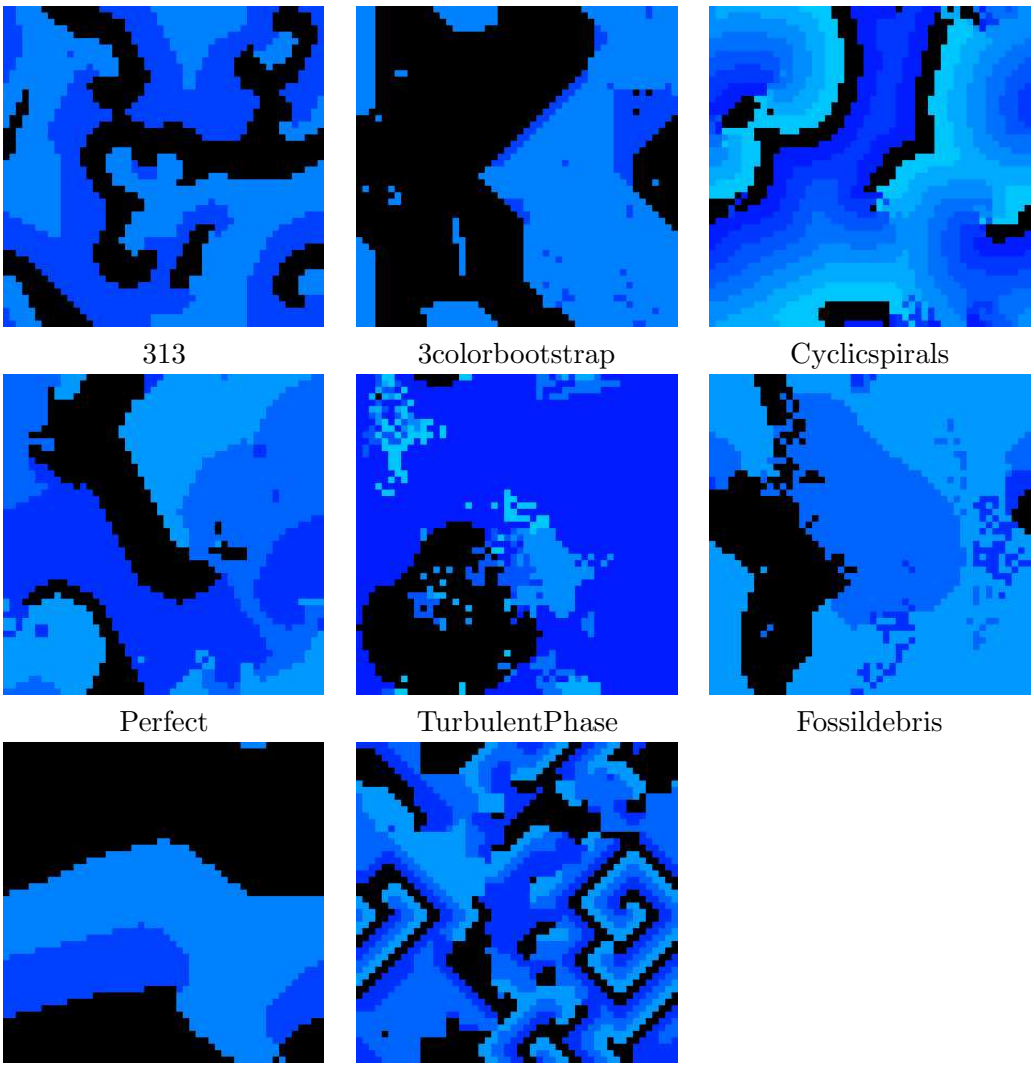


TAB. 7 – Exemple d'évolutions d'automates de type Generation

Au début les états de chaque cellule doivent être initialisés au hazard. A chaque étape une cellule passe dans l'état suivant ssi il y a au moins `S` cellules dans l'état suivant dans son voisinnage de taille `R`. La table 8 donne quelques exemples de paramètres intéressants.

nom de la règle	rayon	seuil	couleur
313	1	3	3
3colorbootstrap	2	11	3
Cyclicspirals	3	5	8
Fossildebris	2	9	4
Imperfect	1	2	4
LavaLamp	2	10	3
Perfect	1	3	4
TurbulentPhase	2	5	8

TAB. 8 – Exemple de règles d'évolution du type Cyclic



TAB. 9 – Exemple d'automates de type Cyclic

Chapitre 3

Abstraction de la notion de liste

Comme nous l'avons vu précédemment, il existe deux classes : `ArrayList` et `LinkedList` permettant de manipuler des listes d'éléments de tailles variables. Chacun de ces deux classes présente des avantages et des inconvénients propres. Dans le cas de `ArrayList`, l'accès à un élément quelconque est très rapide (comme dans un tableau !). Au contraire l'ajout de nouveaux éléments dans un `ArrayList` prend en moyenne un temps proportionnel au complémentaire de la position de l'élément ajouté. Par exemple l'ajout d'un élément en première position prend un temps proportionnel au nombre d'élément stocké, l'ajout d'un élément en dernière position prend (en moyenne !) un temps très court. Dans le cas des `LinkedList` la situation est approximativement inverse : l'ajout d'un élément au début ou à la fin de la liste est très rapide, alors que l'accès direct à un élément est lent. Enfin il existe d'autres méthodes de programmation dont les performances sont intermédiaires entre `ArrayList` et `LinkedList` : le temps d'accès à un élément comme le temps nécessaire à l'ajout d'un élément dépendent du nombre d'éléments, mais l'ajout est plus rapide que dans un `ArrayList`, alors que l'accès est plus rapide que dans un `LinkedList`.

Dans la pratique, le programmeur sait parfois choisir la programmation la plus efficace et aucun problème ne se pose. Ce genre de situations est malheureusement rare et il est parfois difficile de choisir, surtout quand on commence l'écriture d'un programme (quand le programme est écrit, on peut faire des tests pour trouver les parties les plus lentes et donc les améliorer, parfois en changeant de programmation pour les listes). Il est donc souhaitable en général de pouvoir écrire un programme *sans faire d'hypothèse* sur la façon dont seront programmées certaines classes. Pour ce faire, il suffit de mettre en place une interface, c'est-à-dire un contrat pour chaque classe qu'on ne sait pas encore programmer. Au lieu d'utiliser la classe en question, on va utiliser l'interface partout où c'est possible.

1 Solution retenue pour les listes

Pour les listes, `Java` définit une interface `List`. Cette interface comporte à peu près les mêmes méthodes que `ArrayList` et `LinkedList`. Pour être précis, voici une définition simplifiée de l'interface `List`. Certaines méthodes sont absentes pour des raisons techniques, d'autres pour des raisons pédagogiques et seront vus dans la suite.

List

```
1 public interface List {
2   public void add(int index, Object element);
3   public boolean add(Object o);
4   public void clear();
5   public boolean contains(Object o);
6   public boolean equals(Object o);
```

```
7     public Object get(int index);
8     public int indexOf(Object o);
9     public boolean isEmpty();
10    public int lastIndexOf(Object o);
11    public boolean remove(Object o);
12    public Object remove(int index);
13    public Object set(int index, Object element);
14    public int size();
15    public Object[] toArray();
16  }
```

On remarque que les méthodes `ensureCapacity` et `trimToSize` des `ArrayList` ne sont pas présentes. C'est normal car elles sont spécifiques à une programmation basée sur les tableaux.

Chaque programmation particulière des listes est une classe qui implante l'interface `List`, ajoutant au besoin ses propres méthodes. C'est le cas par exemple de `ArrayList` et de `LinkedList`.

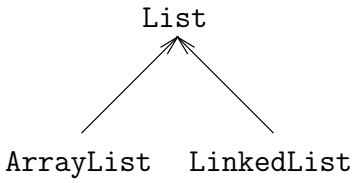


FIG. 1 – L'interface `List` et deux implantations

2 Exemple d'utilisation

Afin de mieux comprendre comment utiliser l'interface `List`, nous allons écrire un programme permettant d'obtenir la liste des mots d'un texte. Puis nous comparerons les performances de ce programme avec les `ArrayList` et `LinkedList` (comme `ArrayList` et `LinkedList` implantent l'interface `List`, il est possible de placer une référence vers un objet de type `ArrayList` ou `LinkedList` dans une variable de type `List`).

On suppose pour simplifier que nous disposons d'une classe `Parseur` permettant d'obtenir successivement les mots d'un texte (séparés par des espaces) :

```
public class Parseur {
        ...
        public Parseur(String texte) {...}

        /* renvoie true si il reste des mots non lus */
        public boolean hasNext() { ...}

        /* renvoie le prochain mot */
        public String next() {...}
}
```

Pour constituer la liste des mots du texte il suffit d'écrire par exemple (en utilisant les ArrayList) :

```
List l=new ArrayList() ;
/* calcul de la liste des mots */
while (p.hasNext())
        l.add(p.next()) ;
/* tri de la liste des mots */
Collections.sort(l) ;
```

De même pour utiliser les `LinkedList` il suffit de remplacer la première ligne par :

```
List l=new LinkedList() ;
```

Avec un texte de 500 pages, sur un ordinateur relativement ancien, on obtient la liste des mots en 90 s avec `ArrayList` et en 70 s avec `LinkedList`. Si on écrit maintenant un programme renvoyant la liste des mots distincs :

```
List l=... ;
/* calcul de la liste des mots */
while (p.hasNext()) {
        Object temp=p.next() ;
        if (!l.contains(temp))
                l.add(temp) ;
}
/* tri de la liste des mots */
Collections.sort(l) ;
```

On obtient cette fois la réponse en 120 s avec les `ArrayList` et en 150 s avec les `LinkedList`. Comme vous le voyez de petites modifications dans le programme peuvent induire des temps d'exécution très différents suivant que l'on utilise des `ArrayList` ou des `LinkedList`. Pour cela il est en général conseillé d'utiliser toujours des variables de type `List` dans un programme et de ne spécifier le type de liste utilisée que au moment de la construction. Ainsi il est très facile de trouver la programmation la plus adaptée pour le cas considéré.

Remarque 1. Comme pour `ArrayList` et `LinkedList` il faut écrire `import java.util.*` au début du programme afin de pouvoir utiliser l'interface `List`.

3 La classe Collections

Dans les deux programmes précédents nous avons utilisé la méthode `sort` de la classe `Collections` afin de trier la liste des mots. La classe `Collections`, à ne pas confondre avec l'interface `Collection` que nous verrons plus tard, propose un ensemble de méthodes de classe permettant de manipuler des listes ou des collections d'objets (dans ce qui suit vous pouvez pour le moment remplacer `Collection` par `List`). Les méthodes les plus utiles sont :

```
/* remplace tous les élements de l par obj */
public static void fill(List l,Object obj) ;

/* inverse l'ordre des éléments de l */
public static void reverse(List l) ;

/* mélange les éléments de l */
public static void shufffle(List l) ;
```

```
/* trie la liste l du plus petit au plus grand élément */
public static void sort(List l) ;

/* renvoie le plus petit élément de la collection c*/
public static Object min(Collection c) ;

/*  renvoie le plus grand élément de la collection c*/
public static Object max(Collection c) ;
```

Il est assez facile de comprendre comment sont programmées les méthodes `fill` et `reverse` (`shuffle` est un peu plus compliqué algorithmiquement). Par contre si vous essayez de programmer `sort`, `min` et `max` vous allez être confronté au problème suivant : comment comparer des `Object`. Nous allons voir dans le chapitre suivant que ce problème se résoud grace à l'interface `Comparable`.

Remarque 2. Dans la pratique, l'utilisation d'une `List` n'est pas toujours souhaitable, en particulier quand on veut travailler sur des types fondamentaux (la création d'une enveloppe pour chaque valeur fondamentale est assez lourde). Heureusement, la classe `Arrays` propose des méthodes `fill`, `sort`, `min` et `max` pour chaque type tableau fondamental[1].

[1] Il s'agit bien ici d'une méthode **différente** pour chaque type fondamental.

PROJET II

Interpreteur Lisp

A la fin des années 1950, John McCarthy créa un nouveau langage de programmation appelé **Lisp** (en anglais : list processing). Ce langage spécialisé dans la manipulation de listes repose sur un jeu d'instruction très simple permettant par combinaisons de réaliser les opérations les plus complexes. Le but de ce problème est de vous permettre de réaliser en Java un interpréteur **Lisp** ; c'est à dire un programme capable de comprendre les instructions du langage **Lisp** . Ces instructions sont au nombre de sept :

`quote` (noté aussi `'`)
`car`
`cdr`
`cons`
`if`
`set`
`lambda`

Afin de comprendre le fonctionnement de **Lisp** nous allons donner quelques exemples. Dans chacun de ces exemples le texte tapé par l'utilisateur est précédé de `>` et le résultat de l'évaluation apparait à la ligne suivante. Par exemple si l'utilisateur tape `quote test` le résultat sera `test` :

```
>quote test
test
```

Une expression **Lisp** consiste en une liste d'atomes séparés par des espaces. Chacun des atomes de l'expression est soit une instruction, soit une chaine de caractère, soit une expression **Lisp** . L'instruction `quote` (en français `citation`), qui sera abrégée dans la suite en `'`, renvoie l'élément suivant de l'expression :

```
>'test
test
>'ceci-est-un-test
ceci-est-un-test
>'(citation d'une liste)
(citation d'une liste)
>('une 'autre 'citation)
(une autre citation)
```

Dans chacun de ces exemples, l'interpréteur provoquera une erreur si vous enlevez les guillemets. En effet dans ce cas vous lui demandez d'évaluer un mot inconnu. Par exemple :

```
>test
erreur: test n'est pas défini
```

Ainsi dans tous les cas les mots utilisés dans une expression doivent soit être des instructions du langage **Lisp** , soit apparaître comme des citations. L'instruction `set` permet de définir de nouvelles instructions. Par exemple pour définir une nouvelle instruction `dix` dont la valeur sera `10` il suffit de taper :

```
 >set 'dix '10
```

On vérifie que la nouvelle instruction `dix` s'évalue bien en `10` :

```
>dix
10
```

En fait ce comportement est beaucoup plus puissant qu'il ne parait. Par exemple on peut maintenant définir `10` indirectement (remarquez l'absence de guillemets devant `dix` dans `set dix '5+5`) :

```
>set dix '5+5
>dix
10
>10
5+5
```

L'instruction `car` permet d'obtenir le premier élément d'une liste. Au contraire l'instruction `cdr` renvoie la liste des éléments moins le premier. Par ailleurs si `car` ou `cdr` sont appliqués à une liste vide ils renvoient une liste vide. Bien entendu on peut combiner plusieurs instructions.

```
>car '((a b c) x y)
(a b c)
>cdr '((a b c) x y)
(x y)
>car (car '((a b c) x y))
a
>cdr (cdr '((a b c) x y))
y
>car (cdr '((a b c) x y))
x
```

L'instruction `cons` permet de construire une nouvel liste en ajoutant un élément au début d'une liste déja existante :

```
>cons 'a ()
(a)
>cons 'a ('b 'c)
(a b c)
>cons ('a 'b) ('x 'y 'z)
((a b) x y z)
>cons '(a b) ('x 'y 'z)
((a b) x y z)
```

Toutes ces instructions permettent donc d'effectuer des manipulations élémentaires sur les listes. Néammoins elles sont très limitées. La puissance de **Lisp** vient des deux instructions `if` et `lambda`. L'instruction `if` s'utilise de manière analogues aux autres langages de programmation : `if condition si-oui si-non` s'évalue en `si-oui` si l'évaluation de `condition` est une liste non vide et en `si-non` si l'évaluation de `condition` est une liste vide.

```
>if ('t) 'vrai 'faux
vrai
>if () 'vrai 'faux
faux
```

Enfin l'instruction `lambda` permet de définir des fonctions. Afin de mieux comprendre cette instruction nous allons voir comment définir l'addition sur les entiers naturels en **Lisp** . Tout d'abord il faut imaginer une représentation des entiers naturels. Le plus simple est de définir l'entier k comme une liste de k `1`. Ainsi 1 sera représenté par `(1)`, 2 par `(1 1)`, 3 par `(1 1 1)`, etc. Afin d'éviter les erreurs lors de l'évaluation nous commencons par définir le symbol `1` (`1` s'évalue en lui même) :

```
>set '1 '1
>1
1
```

La somme de deux entiers est évidemment la concaténation de leur listes. Par exemple `(1 1)` plus `(1 1 1)` est égal à `(1 1 1 1 1)`. Pour commencer nous commençons par définir la fonction `1+` ajoutant 1 à un entier :

```
>set '1+ (lambda (x) (cons 1 x))
>1+ (1 1)
(1 1 1)
```

Il faut interpréter la construction précédente comme suit : le symbole `1+` représente la fonction à un paramètre `x` qui s'écrit formellement `(cons 1 x)`. Ainsi lorsque l'on évalue `1+ (1 1)`, on effectue l'opération `(cons 1 (1 1))` dont le résultat est `(1 1 1)` (ici le paramètre `x` vaut `(1 1)`). De la même façon on défini `1-` :

```
>set '1- (lambda (x) (cdr x))
```

Pour définir la somme de `x` et `y` il suffit alors d'appliquer la formule suivante : si `y` égal zero alors le résultat est `x` sinon le résultat est égal à $1 + (x + (y - 1))$ (définition par récurrence sur `y`).

```
>set '+ (lambda (x y) (if y (1+ (+ x (1- y))) x))
>+ (1 1 1) (1 1)
(1 1 1 1 1)
```

La fonction `+` est par définition une fonction à deux paramètres `x` et `y` dont la formule est `(if y (1+ (+ x (1- y))) x)`. Dans l'exemple précédent `x` vaut `(1 1 1)` et `y` vaut `(1 1)`. Comme vous pouvez l'imaginer on peut définir également la multiplication, la division, etc. En fait le langage **Lisp** présenté ici est aussi puissant que n'importe quel autre langage de programmation évolué.

1 La classe Expression

Afin de représenter les expressions **Lisp** vous allez programmer une classe `Expression` en suivant le squelette indiqué dans la table 1. Le constructeur construit l'expression triviale : la liste des éléments est vide.

1. Programmez la méthode `nil`. Cette méthode renvoie `true` ssi la liste des éléments est vide.
2. Programmez la méthode `car`. Cette méthode renvoie le premier élément de l'expression si celle-ci est non triviale et elle renvoie l'expression appelante sinon.
3. Programmez la méthode `cdr`. Cette méthode renvoie l'expression appelante dont on a supprimé le premier élément (et l'expression appelante diectement si elle est triviale).
4. Programmez la méthode `cons`. Cette méthode ajoute `obj` au début de l'expression appelante et la renvoie.
5. Programmez la méthode `toString`.

```
import java.util.* ;

public class Expression {
        /* liste des éléments */
        private List elements ;

        public Expression() ;

        public boolean nil() ;
        public Object car() ;
        public Expression cdr() ;
        public Expression cons(Object obj) ;

        public String toString() ;
}
```

TAB. 1 – Squelette de la classe Expression

```
public class Parseur {
        /* position dans le texte */
        private int position=0 ;
        /* texte à parcourir */
        private String texte ;

        public  Parseur(String texte) ;
        public int level() ;
        public boolean hasNext() ;
        public String next() ;
}
```

TAB. 2 – Squelette de la classe Parseur

2 Parseur d'expression

Afin de vérifier que votre programme fonctionne il est extrèmement utile de pouvoir construire une expression directement à partir d'une chaine de caractères la représentant. Pour cela nous allons écrire une classe `Parseur` permettant de lire une `String` mot à mot (étant entendu que les mots sont séparés soit par des espaces, soit par des parenthèses, soit par des guillemets simples '). Le squelette de la classe `Parseur` est donné dans la table 2.

1. Programmez le constructeur.
2. Programmez la méthode `hasNext()` renvoyant `true` ssi il reste des caractères à parcourir dans le texte.
3. Programmez la méthode `next()` renvoyant le prochain mot du texte. Si le texte est `"xx (yy (zz) uu)"'` les appels à cette méthode devront renvoyer successivement `"xx"`, `"("` ,`"'"`, `"yy"`, `"("`, `"zz"` `")"` ,`"uu"` et enfin `")"` (remarquez que l'on renvoie les parenthèses et les guillemets simples).
4. Ajoutez à la classe `Expression` un constructeur prenant en paramètre un `Parseur` et construi-

sant l'expression correspondante au texte parcouru par le parseur. L'algorithme de cette méthode est le suivant :
- tant qu'il reste des éléments dans le parseur
  - si l'élément suivant du parseur est une parenthese fermante terminer la boucle.
  - si l'élément suivant du parseur est une parenthèse ouvrante créer une nouvelle expression en utilisant comme paramètre le parseur et l'ajouter à la fin de la liste des éléments de l'expression actuelle.
  - sinon ajouter l'élément suivant à la fin de la liste des éléments de l'expression actuelle.

3 Evaluation d'une expression simple

Afin d'évaluer les expressions de **Lisp** nous allons ajouter deux méthodes à la classe `Expression` :

```
public Object eval() ;
public Object next() ;
```

Ces deux méthodes s'appelant l'une l'autre il est impératif de les programmer en même temps. La méthode `next` renvoie le premier *élément signifiant* de l'expression appelante et le supprime. Son algorithme est le suivant :
- stocker dans une variable `car` le premier élément de l'expression et le supprimer de l'expression appelante
- si `car` est une `Expression` l'évaluer (avec `eval`) puis renvoyer le résultat.
- si `car` est égal à `"quote"` ou `"'"`, supprimer le premier élément de l'expression appelante et le renvoyer.
- renvoyer `car`.

Remarque 1. Attention : ne confondez pas le premier élément de l'expression (donné par la méthode `car()`) avec le premier élément signifiant. Par exemple pour l'expression : `('a 'b)` le premier élément est `'` tandis que le premier élément signifiant est l'évaluation de `'a` c'est a dire `a`.

La méthode `eval` évalue l'expression appelante. Son algorithme est le suivant :
- si l'expression est triviale, la renvoyer
- calculer le premier élément signifiant de l'expression (grace à la méthode `next`)
  - si cet élément est égal à `"car"` : renvoyer le `car()` du prochain élément signifiant (qui est nécessairement une `Expression`).
  - sinon si cet élément est égal à `"cdr"` : renvoyer le `cdr()` du prochain élément signifiant (qui est nécessairement une `Expression`).
  - sinon si cet élément est égal à `"cons"` : calculer les deux prochains éléments signifiants et renvoyer leur `cons(...)` (le deuxième élément signifiant est nécessairement une `Expression`).
- creer une nouvelle expression pour contenir le résultat et lui ajouter la premier élément signifiant calculé plus haut
- tant que l'expression appelante n'est pas triviale : ajouter à la fin de résultat le prochain élément signifiant.
- renvoyer le résultat.

1. Programmez les méthodes `eval` et `next` et vérifier que votre programme fonctionne en utilisant les exemples donnés dans l'introduction.
2. La partie que vous avez programmée ne permet pas à l'interpréteur de comprendre l'instruction `if`. Ajoutez à la méthode `eval` le code nécessaire pour exécuter cette instruction : si l'expression est de la forme `if element1 element2 element3` il faut évaluer `element1`

(grace à `next`) puis suivant le résultat évaluer soit `element2` soit `element3` (attention ça n'est pas aussi simple qu'il y parait et il faut bien vérifier que ça fonctionne dans tous les cas).

4 L'instruction set et l'interface Map

On considère l'interface `Map` suivante (implantée en standard par Java) :

```
public interface Map {
        Object get(Object clef) ;
        Object put(Object clef,Object valeur) ;
}
```

Cette interface représente la notion de dictionnaire. Pour ajouter un nouveau mot au dictionnaire ainsi que sa définition on appel la méthode `put(mot,definition)` (vous n'avez pas à vous préoccuper de la valeur de retour de `put`). Pour obtenir la définition associée à un mot on appel la méthode `Object get(mot)` ; cette méthode renvoie la définition correspondante si le mot est dans le dictionnaire et elle renvoie la référence `null` si le mot n'est pas dans le dictionnaire. On suppose que la classe `Expression` comporte une variable de classe de type `Map` appelée `dictionnaire` :

```
public static Map dictionnaire=new HashMap() ;
```

1. Ajoutez à la méthode `eval` le code permettant d'exécuter l'instruction `set` du langage **Lisp** . Pour ajouter un nouveau mot **Lisp** , il suffit de l'ajouter dans le dictionnaire : la clef est le nouveau mot et la valeur est sa traduction.
2. Modifiez la méthode `next` de telle façon que si `car` est dans le dictionnaire alors la méthode renvoie sa valeur. Verifiez, en utilisant les exemples de l'introduction, que vous pouvez maintenant utiliser les mots définis avec `set`.

5 Représentation et évaluation des fonction

5.1 Clones et transformations

Afin de pouvoir ajouter à notre interpréteur **Lisp** la capacité d'exécuter des fonctions définies par l'utilisateur, il nous faut d'abord apprendre à remplacer dans une expression les variables formelles par leur valeurs concrètes. Imaginons par exemple que nous souhaitions évaluer la fonction `+` sur les listes `(1 1 1)` et `(1 1)`. La définition de `+` donnée dans l'introduction est :

```
set '+ (lambda (x y) (if y (1+ (+ x (1- y))) x))
```

Il nous faut donc remplacer dans l'expression `(if y (1+ (+ x (1- y))) x)`, `x` par `(1 1 1)` et `y` par `(1 1)` afin d'obtenir l'expression :

```
(if (1 1) (1+ (+ (1 1 1) (1- (1 1)))) (1 1 1))
```

Bien que cette opération paraisse très simple il est assez difficile de la programmer correctement[2].

Le premier élément dont nous avons besoin est un moyen d'obtenir le clone d'une expression : c'est à dire une expression dont la représentation est la même mais telle que les modifications de l'expression (par effet de bord par exemple) n'affectent pas son clone. Pour cela vous allez écrire un nouveau constructeur `public Expression(Expression e)` construisant le clone de l'expression paramètre `e`. L'algorithme de cette méthode est le suivant :

[2]En fait la plus grande difficulté vient du fait que presque toutes les méthodes que nous avons définis agissent par effet de bord ; il faut donc remplacer différentes instances des paramètres formelles par des clones de leur définition.

- Parcourir la liste des éléments de `e`
  - si l'élément actuel est une expression ajouter son clone à la liste des éléments de l'expression en construction
  - sinon ajouter l'élément à la liste des éléments de l'expression en construction

Pour effectuer la substitution des variables réelles aux variables formelles dans une expression `e` il faut utiliser l'algorithme suivant :

- parcourir la liste des éléments de `e`
  - si l'élément actuel est une expression : appliquer le même algorithme à cette expression puis ajouter le résultat à la liste des éléments de l'expression résultat
  - si l'élément actuel est une variable formelle : ajouter à la liste des éléments de l'expression résultat un clone de la définition de cette variable
  - sinon ajouter directement l'élément actuel à la liste des éléments de l'expression résultat

Programmez une méthode `Expression transforme(Expression e)` suivant cet algorithme.

5.2 La classe FonctionLisp

Considérons la classe `FonctionLisp` suivante :

```
public class FonctionLisp {
        public Expression symboles ;
        public Expression formule ;

        public FonctionLisp(Expression symboles,Expression formule) {
                this.symboles=symboles ;
                this.formule=formule ;
        }

        public String toString() {
                return "lambda "+symboles+" "+formule ;
        }
}
```

Cette classe permet de représenter une fonction définie par la liste de ses variables formelles (contenues dans l'expression `symboles`) et par sa formule.

1. Ajouter à la méthode `eval` le code permettant de comprendre l'instruction `lambda` correspondant à la définition d'une fonction. Si le premier élément signifiant de l'expression est égal à `"lambda"`, il faut appliquer l'algorithme suivant :
   - calculer l'expression `symboles` égale au premier élément de l'expression appelante
   - supprimer le premier élément de l'expression appelante
   - calculer l'expression `formule` égale au premier élément de l'expression appelante
   - supprimer le premier élément de l'expression appelante
   - renvoyer la fonction correspondante
2. Ajouter à la méthode `eval` le code permettant d'interpréter une fonction définie par l'utilisateur (cas ou le premier élément signifiant de l'expression appelante est une instance de `FonctionLisp`). Utilisez les résultats de la section précédentes et faites très attention aux effets de bord.

6 Question subsidiaire

Définir en **Lisp** la multiplication des entiers naturels et la division euclidienne.

Chapitre 4

Le problème du tri

1 Exemple introductif

Trier des valeurs est un problème central en informatique. On peut même dire que la plupart des applications pratiques de l'informatique utilisent un algorithme de tri : c'est le cas par exemple des tableurs (trier une colonne ou une ligne), des bases de données (le tri permet de rechercher plus rapidement des informations), de la représentation graphique en trois dimensions (l'algorithme le plus utilisé, celui du *Z-Buffer* est une forme d'algorithme de détermination du minimum d'un ensemble), de certains calculs géométriques (par exemple le calcul de l'enveloppe convexe d'un ensemble de points nécessite un tri), etc.

Le principal problème posé par le tri est la difficulté de programmation d'un algorithme de tri efficace. Même un algorithme élémentaire comme celui présenté dans la classe suivante n'est pas complètement trivial. Il s'agit ici de l'algorithme de *tri par sélection*, dans lequel on commence par trouver l'élément le plus petit, puis le second, etc.

```
public class TriSelection {

public static void trier(int[] t) {
        for(int i=0;i<t.length-1;i++) {
        // recherche du plus petit élément à partir de i
        int pos=i;
        for(int j=i+1;j<t.length;j++)
                if(t[j]<t[pos])
                        pos=j;
                // pos est maintenant l'indice du plus petit élément des cases i à
                // t.length-1
                // échange de t[pos] et de t[i] (si nécessaire)
                if(i!=pos) {
                        int tmp=t[pos];
                        t[pos]=t[i];
                        t[i]=tmp;
                }
                // le tableau t est maintenant trié de la case 0 à la case i
        }
}

public static void main(String[] args) {
```

```
        // démonstration
        int[] x={12,3,-2,4,0,10,5};
        trier(x);
        for(int i=0;i<x.length;i++)
        System.out.print(x[i]+" ");
        System.out.println();
        }
}
```

Ce programme d'exemple affiche bien comme prévu :

```
-2 0 3 4 5 10 12
```

Le principal problème posé par cette programmation est qu'elle ne travaille que sur des `int[]`. Comment faire pour trier des `double[]` par exemple ? On pourrait bien entendu trier des tableaux de `Number`, ce qui serait déjà un grand pas en avant. Mais cela n'est pas suffisant. Comment en effet trier un tableau de `String`s par exemple ? De façon plus générale, si on définit une nouvelle classe dont les instances sont comparables, comment faire pour utiliser un programme de tri déjà existant ? La solution réside une fois encore dans l'emploi d'une interface qui va proposer comme contrat l'existence d'une méthode de comparaison d'éléments entre eux.

2 L'interface `Comparable`

2.1 Présentation de l'interface

`Java` définit l'interface suivante :

`Comparable`

```
1 public interface Comparable {
2   public int compareTo(Object o);
3 }
```

Le sens de la méthode `compareTo` qui doit donc être programmée par une classe qui souhaite remplir le contrat `Comparable` est le suivant. L'appel `a.compareTo(b)` renvoie un entier strictement négatif si `a` est "strictement plus petit" que `b`, strictement positif si `a` est "strictement plus grand" que `b` et enfin nul si `a` et `b` sont égaux. L'interface `Comparable` est implantée par les classes suivantes (la liste n'est pas exhaustive) :

- `String`[1]
- les enveloppes numériques (`Double`, `Integer`, etc., mais pas `Number`)
- `Date` (une classe permettant, comme son nom l'indique, de représenter des dates)

2.2 Pourquoi un paramètre de type `Object` ?

On peut se demander pourquoi la méthode `compareTo` prend comme paramètre un `Object`. En effet, dans la pratique, on ne va pas comparer une `String` avec un `Double`. Il serait donc naturel pour les `String`s de définir `compareTo(String s)` et pour les `Double`s, `compareTo(Double d)`. Le seul problème est que ces deux méthodes sont **différentes** car elles ne demandent pas les mêmes paramètres. De ce fait, elles ne programment pas la même interface. Si on veut avoir une interface `Comparable` qui soit utilisable pour toute classe, on est donc obligé d'avoir un paramètre de type `Object`.

[1] mais **pas** `StringBuffer`.

2.3 Exemple d'utilisation

Voici maintenant une nouvelle programmation du tri par sélection, basée sur l'interface `Comparable` :

```
public class TriSelectionC {

public static void trier(Comparable[] t) {
         for(int i=0;i<t.length-1;i++) {
         // recherche du plus petit élément à partir de i
                int pos=i;
                for(int j=i+1;j<t.length;j++)
                if(t[j].compareTo(t[pos])<0)
                        pos=j;
                        // échange de t[pos] et de t[i]
                if(i!=pos) {
                        Comparable tmp=t[pos];
                        t[pos]=t[i];
                        t[i]=tmp;
                }
         }
}

public static void main(String[] args) {
        // démonstration
        Double[] x={new Double(12.5),new Double(3.0),new Double(-2.1),
                new Double(4),new Double(0.25),new Double(10),new Double(5.2)};
        trier(x);
        for (int i=0;i<x.length;i++)
                System.out.print(x[i]+" ") ;
        System.out.println() ;
        String[] s={"ciceron","caton","cesar","lucrece"} ;
        trier(s);
        for (int i=0;i<s.length;i++)
                System.out.print(s[i]+" ") ;
        System.out.println() ;
}
}
```

On voit que le programme utilise le fait que si on peut placer une référence de type `B` dans une variable de type `A`, alors on peut placer une référence sur un tableau de `B` dans une variable de type `A[]`. Ainsi peut-on placer une référence vers un tableau de type `String[]` dans une variable de type `Comparable[]`. On remarque aussi que le programme n'est pas plus complexe que la première version ne fonctionnant que sur les `int`. L'affichage produit par le programme est le suivant (comme prévu) :

```
-2.1 0.25 3.0 4.0 5.2 10.0 12.5
caton cesar ciceron lucrece
```

2.4 Exemple de programmation

Considérons une classe `Entier` représentant les entiers naturels (cette classe est similaire à la classe `Integer`). Les entiers naturels étant ordonnés on veut que cette classe implante l'interface `Comparable`. Une programmation possible est :

```
public class Entier implements Comparable {
        private int val ;

        public Entier(int val) {
                this.val=val ;
        }

         public int getVal() {
                return val ;
        }

        public int compareTo(Object obj) {
                Entier parametre=(Entier)obj ;
                return (this.val-parametre.val) ;
        }

        public String toString() {
                return String.valueOf(val) ;
        }
}
```

Comme vous pouvez le constater, la première ligne de la méthode `compareTo` consiste en un transtypage vers la classe de l'objet appelant. En particulier si l'objet paramètre n'est pas du même type que l'objet appelant la méthode provoquera une erreur à l'exécution. Ceci est tout à fait normal car il est en général impossible de comparer des objets de type différents (par exemple on ne peut pas dire quel est le plus petit élément entre `"abc"` et un `Integer` représentant l'entier `3`). **Lorsque vous programmerez des méthodes `compareTo` il faudra toujours que la première ligne consiste en un transtypage vers la classe de l'objet appelant**. De plus la validité du transtypage ne doit pas être testé préalablement (avec `instanceof` par exemple) car la méthode doit provoquer une erreur à l'exécution si les objets sont de type différents.

Une fois la classe `Entier` programmée, on peut utiliser notre méthode de trie pour trier un tableau d'`Entier`s.

```
        Entier[] u={new Entier(4),new Entier(7),new Entier(3),new Entier(1)} ;
        trier(u);
        for (int i=0;i<u.length;i++)
                System.out.print(u[i]+" ") ;
        System.out.println() ;
```

Le programme affiche bien comme prévu :

```
1 3 4 7
```

3 L'interface Comparator

3.1 Introduction

La solution basée sur l'interface `Comparable` est très satisfaisante pour les applications classiques du tri. Elle pose cependant un problème : pour changer d'ordre, il faut changer de classe. Ceci peut paraître normal : la méthode `compareTo` propose un ordre *naturel* pour les objets de la classe considérée. Le problème est que l'ordre naturel n'existe pas toujours : quel ordre choisir pour les nombres complexes par exemple ? De plus, comme les méthodes de tri de `Java` trient en ordre croissant, il n'est pas possible simplement de trier en ordre décroissant (bien entendu, il suffit de faire le miroir du résultat, mais c'est du temps perdu).

Dans certains cas, on aimerait donc pouvoir préciser la technique de classement à utiliser (c'est-à-dire, plus mathématiquement, la relation d'ordre) sans passer par l'interface `Comparable`. Il s'agit en fait de passer un *algorithme de comparaison* à une méthode de tri.

3.2 L'interface Comparator

`Java` définit donc une interface `Comparator` qui sert à décrire une relation d'ordre. Voici cette interface (définie dans le *package* `java.util`) :

Comparator

```
1 public interface Comparator {
2   public int compare(Object o1,Object o2);
3   public boolean equals(Object obj);
4 }
```

Voici le sens à donner aux deux méthodes :

`int compare(Object o1,Object o2)`

: L'appel `c.compare(o1,o2)` demande au comparateur auquel `c` fait référence de comparer les objets `o1` et `o2`. L'appel renvoie une valeur strictement négative si `o1` est strictement plus petit que `o2`, strictement positive si `o1` est strictement plus grand que `o2` et nulle si les deux objets sont égaux.

`boolean equals(Object obj)`

: Renvoie `true` si et seulement si la *relation d'ordre* représentée par l'objet appelant est le même que celui représenté par l'objet `obj`. Il s'agit de la méthode `equals` classique[2] et quand on programme un `Comparator`, on peut se contenter de la version de `equals` proposée par défaut par la classe `Object`.

Une classe qui implante l'interface `Comparator` est donc une programmation d'une relation d'ordre.

Remarque 1. Il faut bien comprendre qu'il s'agit d'une relation d'ordre au sens général et mathématique du terme. Le fait que `compare` est interprété *informatiquement* comme donnant un résultat positif si le premier objet est plus grand que le second ne fixe pas l'interprétation *mathématique* que doit avoir cette méthode. La section suivante donne un exemple de comparateur qui renverse l'ordre naturel afin de permettre de classer des objets par ordre décroissant.

3.3 Exemples d'implantation

Supposons qu'on veuille maintenant trier notre liste d'`Entier`s du plus grand au plus petit. Pour cela nous allons programmer une classe permettant de comparer les entiers dans l'ordre anti-naturel (0 est plus grand que 1 qui est plus grand que 2 etc).

[2] On peut d'ailleurs s'interroger sur la présence de cette méthode dans l'interface, alors que tout se passerait de la même façon si elle était absente...

```
import java.util.* ;
public class ComparateurAntinaturel implements Comparator {

        public int compare(Object  o1,Object o2) {
                Entier e1=(Entier)o1 ;
                Entier e2=(Entier)o2 ;
                return -e1.compareTo(e2) ;
        }
}
```

Cette classe demande quelques explications :

- la première ligne (avec `import`) permet d'utiliser l'interface `Comparator` ;
- dans la méthode `compare`, on ne teste pas le type des paramètres avant le transtypage. C'est relativement logique : on ne peut pas comparer deux objets qui sont instances de classes différentes. Dans ce cas, il vaut mieux que le programme s'arrête (c'est exactement le même problème que pour la méthode `compareTo` de l'interface `Comparable`, cf section 2)
- pour simplifier nous n'avons pas reprogrammé la méthode `equals`. Par défaut la méthode `equals` de la classe `Object` sera utilisé.

On peut en fait aller beaucoup plus loin en fabriquant un comparateur qui renverse l'ordre naturel en général :

```
import java.util.Comparator;
public class CompareReverse implements Comparator {

        public int compare(Object o1,Object o2) {
                Comparable c2=(Comparable)o2;
                // on inverse l'ordre d'appel usuel
                return c2.compareTo(o1);
        }
}
```

En fait, ce comparateur est déjà défini par `Java` et est obtenu grâce à la méthode `reverseOrder` de la classe `Collections`. .

3.4 Exemple d'utilisation

Voici maintenant une nouvelle version du tri par sélection, travaillant sur un tableau d'`Objects` et utilisant l'interface `Comparator` :

```
import java.util.*;
public class TriSelectionComparator {
        public static void trier(Object[] t,Comparator ordre) {
                for(int i=0;i<t.length-1;i++) {
                        int pos=i;
                        for(int j=i+1;j<t.length;j++)
                                if(ordre.compare(t[j],t[pos])<0)
                                        pos=j;
                        if(i!=pos) {
                                Object tmp=t[pos];
                                t[pos]=t[i];
                                t[i]=tmp;
```

```
                        }
                }
}

public static void main(String[] args) {
        Entier[] u={new Entier(4),new Entier(7),new Entier(3),new Entier(1)} ;
        trier(u,new ComparateurAntinaturel());
        for (int i=0;i<u.length;i++)
                System.out.print(u[i]+" ") ;
        System.out.println() ;
}
}
```

Comme prévu, l'affichage obtenu est :

```
7 4 3 1
```

La ligne

```
trier(u, new ComparateurAntinaturel());
```

peut être remplacée par

```
trier(u, new CompareReverse());
```

ou

```
trier(u, Collections.reverseOrder());
```

Dans les trois cas on obtient le même résultat.

3.5 Mise en œuvre dans Java

Comme `Comparable`, `Comparator` est défini par `Java`. De ce fait, le langage utilise cette interface. La classe `Collections` possède les méthodes suivantes :

`void sort(List list, Comparator c)`
: Trie la liste `list` en utilisant l'ordre défini par `c`.

`Object min(Collection coll, Comparator comp)`
: Renvoie le plus petit élément de la *collection* `coll`, selon l'ordre défini par `comp`.

`Object max(Collection coll, Comparator comp)`
: Renvoie le plus grand élément de la collection `coll`, selon l'ordre défini par `comp`.

De la même façon, la classe `Arrays` définit une méthode `sort` qui prend comme paramètre un `Object[]` et un `Comparator`, et qui réalise le tri du tableau en utilisant l'ordre défini par le comparateur.

Exemple 1 :

Voici un exemple de tri d'un tableau de `Numbers`. On commence par définir un comparateur qui travaille sur les `Numbers` :

```
import java.util.Comparator;
public class CompareNumber implements Comparator {
        public int compare(Object o1,Object o2) {
                Number n1=(Number)o1;
```

```
                Number n2=(Number)o2;
                if(n1.doubleValue()<n2.doubleValue())
                        return -1;
                if(n1.doubleValue()>n2.doubleValue())
                        return 1;
                return 0;
        }
}
```

Voici maintenant un exemple d'utilisation de ce comparateur :

```
import java.util.*;
public class TriArraysComparator {
        public static void main(String[] args) {
                Number[] x={new Double(12.5),new Integer(3),new Float(-2.1f),
                        new Byte((byte)4),new Double(0.25),new Long(10),
                        new Double(5.2)};
                Arrays.sort(x,new CompareNumber());
                System.out.println(Arrays.asList(x));
        }
}
```

Comme prévu, ce programme affiche la ligne suivante :

```
[-2.1, 0.25, 3, 4, 5.2, 10, 12.5]
```

Remarque 2. On peut se demander pourquoi le comparateur `CompareNumber` n'existe pas en `Java`. Ceci est simplement du au fait que la conversion d'un `long` en `double`, même si elle est autorisée, peut provoquer une perte d'information. De ce fait, un tri basé sur le comparateur `CompareNumber` peut produire des résultats faux.

Pour illustrer le problème de la conversion, il suffit de considérer le programme suivant :

```
public class BugLong {
        public static void main(String[] args) {
                long l=Long.MAX_VALUE;
                System.out.println(l);
                // conversion autorisée, mais avec perte :
                float f=l;
                System.out.println(f);
                l=l-1;
                System.out.println(l);
                float g=l;
                System.out.println(g);
                // résultat faux
                System.out.println(f==g);
                // bien entendu, ceci fonctionne :
                long p=l+1;
                System.out.println(p==l);
                // même avec les doubles, cela ne fonctionne pas :
                double u=l;
                l=l-1;
                double v=l;
                System.out.println(u);
```

```
                System.out.println(v);
                System.out.println(u==v);
        }
}
```

L'affichage produit est :

```
9223372036854775807
9.223372E18
9223372036854775806
9.223372E18
true
false
9.223372036854776E18
9.223372036854776E18
true
```

PROJET III

Codage de Huffman

Pour transmettre ou stocker un document électronique il est souvent très pratique de compresser ce document préalablement afin de diminuer le volume des données à transférer. De nombreux programmes commerciaux remplissant cette tache existent et vous avez sans doute déja été amené à utiliser l'un d'entre eux. Dans ce sujet vous allez apprendre à programmer une méthode de compression très efficace, appelée codage de Huffman (d'après le nom de son inventeur). Dans le cas où aucune information n'est disponible sur le texte à compresser (on sait seulement que le texte est une suite de caractères) cette méthode est la plus efficace possible.

Le codage de Huffman s'applique principalement à la compression de données textuelles. Son principe est le suivant : dans l'ordinateur chaque caractère de l'alphabet est représenté par un chiffre variant entre 0 et 65536 appelé le code Unicode du caractère. Ce chiffre exprimé en base 2 correspond à une série de 16 bits (=16 chiffres entre 0 et 1). Par exemple pour le caractère '`a`' on aura le code `0000000001100001`. Ainsi pour enregistrer un texte de n caractères dans l'ordinateur on a généralement besoin de $16 \times n$ bits. Le but du codage de Huffman est de réduire le nombre de bits nécessaire pour chaque caractère. Plus précisemment le nombre de bits nécessaire va devenir une fonction de la fréquence du caractère dans le texte à compresser : plus un caractère est fréquent moins il faudra de bits pour le coder. Le codage se déroule en trois phases : tout d'abord on commence par compter le nombre d'occurences de chaque caractère ; puis en utilisant ces occurences, on construit un arbre binaire dont les feuilles sont les caractères ; cet arbre permet d'associer à chaque caractère un code binaire optimal ; enfin dans la troisième étape on remplace chaque caractère dans le texte par son code optimal.

1 Représentation des caractères

Afin d'effectuer le codage de Huffman d'un texte il nous faut compter le nombre d'occurences de chaque caractère puis associer à chaque caractère un code binaire. L'ensemble de ces informations est représenté par la classe `CharHuffman` dont le squelette est donné dans la table 1. La méthode `toString` renvoie la représentation du `CharHuffman` appelant sous la forme : `c occurence code`. Par exemple le programme suivant :

```
CharHuffman ch=new CharHuffman('a') ;
System.out.println(ch) ;
```

affiche `a 1 null` (au départ le code n'est pas défini et vaut `null`). Par ailleurs le programme :

```
CharHuffman ch=new CharHuffman('u') ;
ch.addOccurence() ;
ch.setCode("0110") ;
System.out.println(ch) ;
```

affiche `u 2 0110`. Programmez la classe `CharHuffman`.

2 Arbre de Huffman

Afin de calculer le code optimal de chaque caractère en fonction de son nombre d'occurences dans le texte à compresser, nous allons construire un arbre appelé arbre de Huffman. Un arbre

```
public class CharHuffman {
/* classe representant un caractère pour le codage de Huffman */

        /* caractère représenté */
        public char c ;
        /* nombre d'occurences */
        private int occurence ;
        /* code de Huffman (en binaire dans une String) */
        private String code ;

        public CharHuffman(char c) ;

        /* ajoute 1 au nombre d'occurences */
        public void addOneOccurence() ;
        /* renvoie le nombre d'occurences */
        public  int getOccurence() ;

        /* modification et obtention du code */
        public void setCode(String code) ;
        public String getCode() ;

        public String toString() ;
}
```

TAB. 1 – Squelette de la classe CharHuffman

```
public class ArbreHuffman {
/* classe représentant un arbre de Huffman */

        /* sommet */
        private CharHuffman sommet ;
        /* successeurs gauches et droit */
        private ArbreHuffman gauche ;
        private ArbreHuffman droite ;

        public ArbreHuffman(CharHuffman sommet) ;
        public ArbreHuffman(ArbreHuffman gauche,ArbreHuffman droite) ;

        public boolean feuille() ;

}
```

TAB. 2 – Squelette de la classe ArbreHuffman

de Huffman est un arbre binaire (chaque noeud a au plus deux descendants) dont les feuilles sont des `CharHuffman`. La classe `ArbreHuffman` dont le squelette est donné table 2 représente informatiquement un arbre de Huffman.

Comme vous le voyez la définition de `ArbreHuffman` est récursive : chaque `ArbreHuffman` possède comme variables d'instance deux `ArbreHuffman`s : le successeur `gauche` et le successeur `droit`. Le constructeur `ArbreHuffman(CharHuffman sommet)` construit un arbre dont le seul noeud est `sommet` (en particulier `gauche` et `droite` sont `null`). Le constructeur `ArbreHuffman(ArbreHuffman gauche,ArbreHuffman droite)` construit un arbre réunion des arbres `gauche` et `droite` : le sommet de l'arbre est `null` et ses descendants gauche et droite sont respectivement `gauche` et `droite`. Par exemple pour construire l'arbre `adf` représenté figure 1, il faut écrire :

```
CharHuffman ca=new CharHuffman('a') ;
CharHuffman cd=new CharHuffman('d') ;
CharHuffman cf=new CharHuffman('f') ;
cf.addOccurence() ;
cf.addOccurence() ;
ArbreHuffman a=new ArbreHuffman(ca) ;
ArbreHuffman d=new ArbreHuffman(cd) ;
ArbreHuffman f=new ArbreHuffman(cf) ;
ArbreHuffman ad=new ArbreHuffman(a,d) ;
ArbreHuffman adf=new ArbreHuffman(ad,f) ;
```

1. Programmez les deux constructeurs et la méthode `feuille()` ; cette méthode renvoie `true` ssi les successeurs de l'arbre appelant sont `null`.
2. La valeur d'un arbre de Huffman est la somme des occurences de ses feuilles. Par exemple pour l'arbre `adf` construit plus haut, la valeur de l'arbre est 5 car les occurences des feuilles sont 1, 1 et 3. Programmez une méthode d'instance `int valeur()` renvoyant la valeur de l'arbre appelant.
3. Modifier la classe `ArbreHuffman` afin d'implanter l'interface `Comparable`. Pour comparer deux

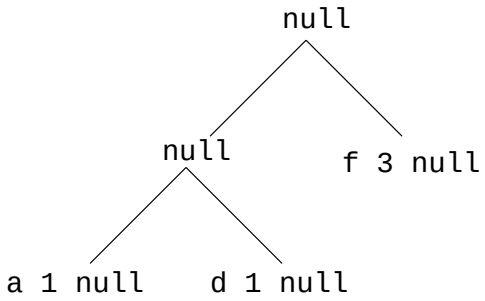


FIG. 1 – Un arbre de Huffman

arbres vous utiliserez leur valeur (le plus petit de deux arbres étant celui qui a la plus petite valeur).

4. Programmez une méthode d'instance `CharHuffman find(char c)` permettant de savoir quelle feuille de l'arbre appelant contient un `CharHuffman` représentant le caractère `c` : cette méthode renvoie la feuille correspondante si elle existe et `null` sinon.

3 Code de Huffman

Etant donné un arbre de Huffman, on peut associer à chaque feuille de l'arbre un code optimal. Pour calculer le code d'une feuille on regarde le chemin reliant la racine de l'arbre à la feuille : se chemin peut se coder par une suite de lettre `g` ou `d` indiquant à chaque noeud si il faut aller vers la gauche ou vers la droite puis on remplace les `g` par `0` et les `d` par `1`. Le résultat est le code de Huffman. Par exemple pour l'arbre `adf` de la figure 1 on obtient les codes suivants : `00` pour la feuille `a`, `10` pour `d` et `11` pour `f`. En effet pour aller de la racine jusqu'a la feuille `a` il faut aller deux fois à gauche, pour aller jusqu'a la feuille `d` il faut aller à gauche puis à droite, et enfin pour aller de la racine à la feuille `f` il faut aller deux fois à droite. Programmez une méthode permettant de calculer les codes des feuilles de l'arbre appelant (cette méthode agit sur les feuilles par effet de bord afin de modifier leur code).

4 Arbre de Huffman d'un texte

Etant donné un texte, nous allons construire l'arbre de Huffman de ce texte.

1. Ecrire une méthode de classe `public static List occurences(String texte)`. Renvoyant la liste des caractères d'un texte avec leurs occurences. Plus précisemment cette méthode devra renvoyer une liste dont les éléments sont des `CharHuffman`, chaque `CharHuffman` correspondant à un des caractères apparaissant dans le texte (avec le bon nombre d'occurences). Par exemple si le texte est `afdff` la méthode renverra la liste :

   ```
   [a 1 null, f 3 null, d 1 null]
   ```

2. Etant donné la liste des occurences des caractères d'un texte (sous la forme de `CharHuffman`) on construit l'arbre de Huffman correspondant grace à l'algorithme suivant :
   - construire une liste dont chaque élément est un arbre de Huffman dont la seule feuille correspond à un des éléments de la liste des occurences
   - Tant que la liste des arbres contient au moins deux éléments :
     - trier la liste
     - remplacer les deux plus petit éléments par leur réunion

– le résultat est l'unique élément restant de la liste des arbres.
Par exemple si la liste des occurences est :

```
[a 5 null, b 12 null, k 1 null, m 6 null, p 3 null]
```

La construction de l'arbre se déroule comme indiqué figure 2. Programmez une méthode de classe effectuant cet algorithme.

5 Codage binaire des caractères

Comme nous l'avons dit dans l'introduction, à chaque caractère correspond un nombre en base 2 comportant 16 chiffres (en particulier la valeur représentée par ce nombre est comprise entre 0 et 65536). Etant donné un caractère `c` il suffit pour obtenir le code Unicode correspondant de transtyper `c` en `int`. Une fois ce transtypage effectué, la représentation binaire de `c` est tout simplement égale à la représentation binaire du code Unicode correspondant à `c`. Soit u un entier entre 0 et 65536 et soit $b_0 b_1 \dots b_{15} b_{16}$ la représentation binaire de u. L'entier u et sa représentation binaire sont liés par l'équation :

$$\begin{aligned} u = \quad & 32768b_0 + 16384b_1 + 8192b_2 + 4096b_3 + 2048b_4 + 1024b_5 + 512b_6 + 256b_7 \\ & +128b_8 + 64b_9 + 32b_{10} + 16b_{11} + 8b_{12} + 4b_{13} + 2b_{14} + b_{15} \end{aligned}$$

1. Programmez une méthode de classe permettant d'obtenir un caractère à partir de son code binaire (représenté comme un tableau de 16 entiers).
2. Programmez une méthode de classe prenant en paramètre un caractère `c` et un entier `n` et renvoyant le n-ème bit de la représentation binaire de `c`.

5.1 Lecture d'une chaine de caractères bit à bit

Etant donnée une chaine de caractère, vous pouvez en Java accéder à n'importe quel caractère de la chaine grace à la méthode d'instance `char charAt(int i)`. Malheureusement pour programmer le codage de Huffman il faut pouvoir lire la chaine de caractère non pas caractère par caractère mais bit à bit : c'est à dire que chaque caractère sera lu sous forme de 8 chiffres entre 0 et 1. Pour pouvoir faire cela simplement vous allez programmer une classe `BitStringReader`.

1. Programmez un constructeur pour la classe `BitStringReader` prenant en paramètre le texte à lire.
2. Programmez une méthode d'instance `int readBit()` permettant d'obtenir un à un les bits composant le texte (chaque appel à `readBit()` renvoie le prochain bit de la chaine de caractère). Pour cela vous utiliserez une variable d'instance de type entier indiquant la position du caractère actuellement lu et une autre variable entière indiquant le numéro du prochain bit à lire. Pour chaque caractère la méthode `readBit()` renvoie un à un les bits composant le code binaire du caractère, avant de passer au caractère suivant.
3. Programmez une méthode `boolean hasNext()` renvoyant `true` ssi il reste des bits à lire dans le texte.

Verifiez que votre classe `BitStringReader` fonctionne correctement. Pour cela vous pouvez utiliser le texte suivant (première phrase d'un livre célèbre) :

```
Longtemps je me suis couché de bonne heure
```

Le code binaire correspondant est :

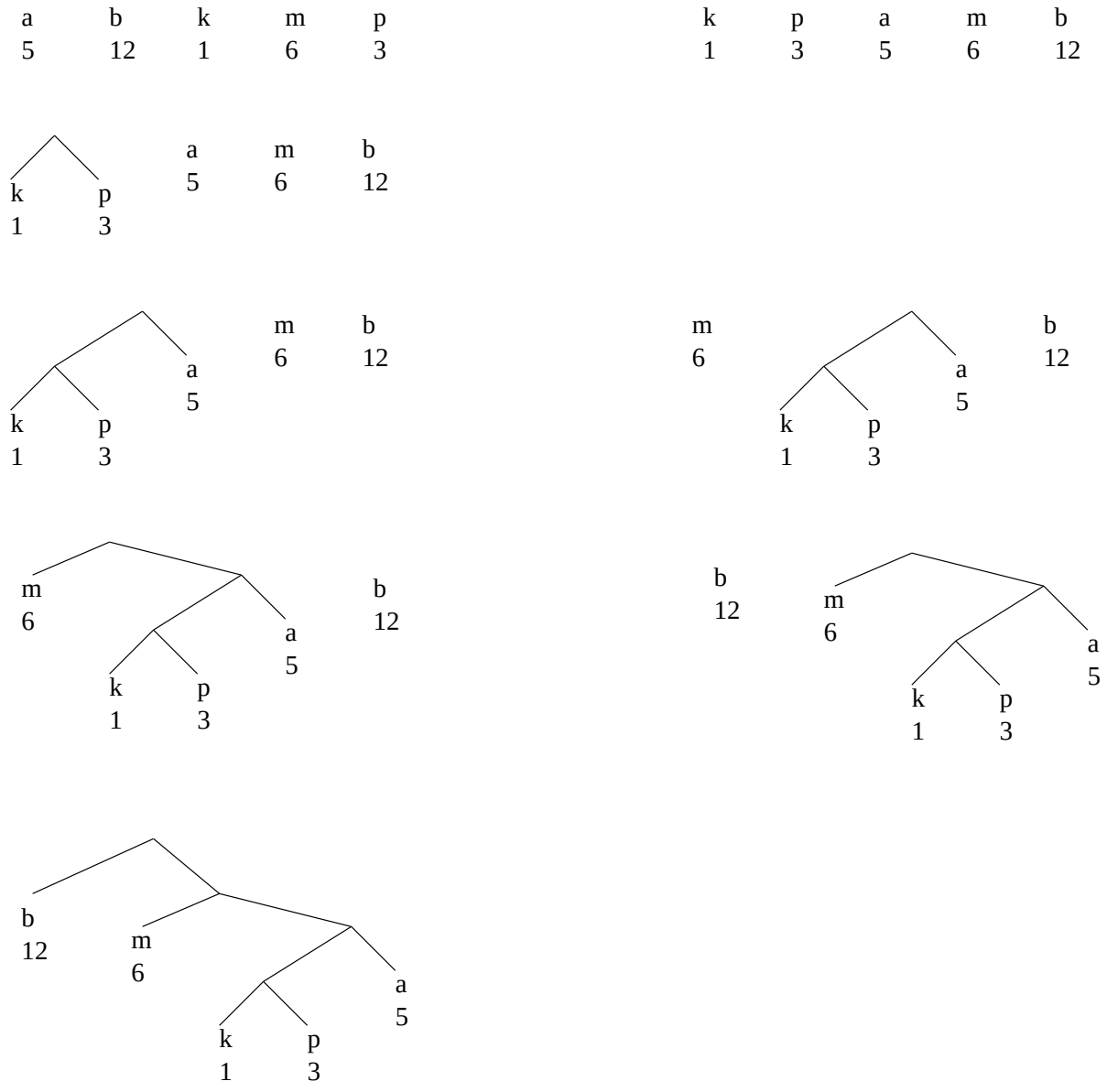


FIG. 2 – Construction d'un arbre de Huffman

```
00000000010011000000000001101111000000000110111000000000011001110000000
00011101000000000001100101000000000110110100000000011100000000000000111
00110000000000100000000000001101010000000001100101000000000010000000
00000001101101000000000110010100000000001000000000000011100110000000
01110101000000000110100100000000111001100000000010000000000000011000
11000000000110111100000000011101010000000001100011000000000110100000000
00001110100100000000001000000000000001100100000000001100101000000000
10000000000000011000100000000011011110000000011011100000000001101110
00000000110010100000000001000000000000001101000000000001100101000000
00011101010000000011100100000000001100101
```

5.2 Ecriture d'une chaine de caractère bit à bit

De la même façon qu'il faut pouvoir lire une chaine de caractère bit à bit avant de la coder, il faut également pouvoir écrire une chaine de caractère bit à bit une fois le codage effectué. Pour cela vous allez programmer une classe `BitStringWriter`. La classe `BitStringWriter` devra possèder une variable d'instance de type `String` ou mieux `StringBuffer` contenant le texte déja codé.

1. Programmez un constructeur sans paramètre
2. Programmez une méthode `void writeBit(int bit)` permettant d'ajouter un bit à la fin de la chaine de caractère.
3. Ajoutez une méthode `toString` permettant d'obtenir le texte déja codé.

6 Compression d'un texte

L'algorithme permettant de compresser un texte est le suivant :
- calculer la liste des occurences des caractères dans le texte
- calculer l'arbre de Huffman correspondant
- calculer le code optimal de chaque feuille
- pour obtenir la représentation binaire du texte compressé, remplacer chaque caractère dans le texte initial par son code optimal

Si l'on part du texte :

```
Longtemps je me suis couché de bonne heure
```

les différentes étapes de l'algorithme sont les suivantes :
- liste des occurences :
  ```
  [L 1 null, o 3 null, n 3 null, g 1 null, t 1 null, e 7 null, m 2 null,
   p 1 null, s 3 null,   7 null, j 1 null, u 3 null, i 1 null, c 2 null,
   h 2 null, é 1 null, d 1 null, b 1 null, r 1 null]
  ```
- arbre de Huffman (table 3).
- résultat de la compression (en binaire)
  ```
  01100000110011110000010011000010010010111101010011111000001111101011101
  11101001011111000000111011000010000110111101100111110111000011001100010
  11111010001111011111001100000000000
  ```

Comme vous pouvez le constater la longueur du texte en binaire est passée de 672 bits à 176 bits soit un gain de 75%.

1. Ecrire une méthode de classe permettant de compresser un texte.

```
@
        @
                @
                        @
                                @
                                        m 2 0000
                                        c 2 1000
                                @
                                        h 2 0100
                                        @
                                                L 1 01100
                                                g 1 11100
                @
                        @
                                @
                                        t 1 00010
                                        p 1 10010
                                @
                                        j 1 01010
                                        i 1 11010
                        @
                                @
                                        é 1 00110
                                        d 1 10110
                                @
                                        b 1 01110
                                        r 1 11110
        @
                @
                        @
                                o 3 0001
                                n 3 1001
                        @
                                s 3 0101
                                u 3 1101
                @
                        e 7 011
                        _ 7 111
```

TAB. 3 – Arbre de Huffman pour "Longtemps je me suis couché de bonne heure" (chaque noeud de l'arbre est indiqué par @ et est suivi de ses descendant gauche et droites)

2. Ecrire une méthode de classe permettant de décompresser un texte (vous pouvez supposer que la méthode prend en paramètre l'arbre de Huffman ayant servi à compresser le texte initial).
3. Ecrire une classe permettant de compresser/decompresser des fichiers.

PROJET IV

Algorithme génétique

Un *algorithme génétique* est un algorithme dont le fonctionnement est basé sur les idées de reproduction et de sélection issues de la théorie de l'évolution. Etant donné un problème à résoudre, on commence par coder l'ensemble des solutions possibles de ce problème. Puis on teste un grand nombre de solutions possibles choisies au hasard. Parmi ces solutions, certaines sont plus proches que les autres de la véritable solution. On classe donc les solutions et on séléctionne les meilleures. En "croisant" et "mutant" les solutions séléctionnées, on génére un nouvel ensemble de solutions. Puis on recommence le processus de séléction, et ainsi de suite, jusqu'à trouver la bonne solution.

Par exemple considérons le problème du mot mystère, consistant à trouver un mot choisi au hazard en disposant pour seule information du nombre de lettres correctes a chaque tentative. Soit "alea" le mot mystère, l'ensemble des solutions possibles est l'ensemble des mots de 4 lettres. On commence par générer au hazard les quatres mots suivants :

```
erty
akil
bler
hyyy
```

Parmi ces quatres mots, `akil` et `bler` sont les mots ayant le plus de lettres en communs avec le mot mystère. On séléctionne ces deux mots, puis on les "croise" et on les "mute" (c'est à dire que l'on mélange les lettres de ces deux mots et on change certaines lettres au hasard) pour obtenir deux nouveaux mots. On dispose maintenant de la liste de mots :

```
akil
bler
aley
bljr
```

Grace a l'opération de croisement/mutation, on a généré le mot `aley` très proche de la solution. En continuant on trouve rapidement le mot mystère.

La première partie de ce problème consiste en une programmation génériques des algorithmes génétiques. Dans cette partie on défini une interface `Genome` regroupant les propriétés communes à tous les organismes génétiques (croisement, mutation, comparaison) et on programme des méthodes de classes implantant les principes de séléction et de reproduction.

La seconde partie consiste à programmer une classe `MotGenetique` implantant l'interface `Genome` et permettant de résoudre le problème du mot mystère par algorithme génétique.

Dans tous le problème vous aurez fréquemment à effectuer des transtypages avant d'appeler une méthode d'instance. En particulier **vous ferez très attention aux types des objets manipulés**. **Vous avez le droit d'utiliser toutes les méthodes des classes `List` et `Collections`.**

1 La classe Population

Dans la suite les classes représentant des organismes génétiques implanteront l'interface `Genome` suivante :

```
public interface Genome extends Comparable {
        public void mutation() ;
        public Genome croisement(Genome parent);
}
```

Le but de cette partie est de programmer dans une classe `Population` des méthodes statiques permettant de manipuler des `List` de `Genome`.

1. Programmez une méthode de classe `List selection(List individus,int n)` permettant de séléctionner les n meilleures solutions parmi la liste `individus`. On suppose que tous les éléments de la liste `individus` sont de type `Comparables`. Pour séléctionner les n meilleurs, vous pouvez utiliser la méthode suivante : trier la liste du plus grand au plus petit, renvoyer une liste contenant seulement les n premiers éléments de la liste triée.
2. Programmez une méthode de classe `List reproduction(List individus,int n)` permettant d'ajouter par croisement n solutions à la liste `individus` (on suppose que tous les éléments de la liste sont des instances de `Génome`). Pour cela vous répeterez n fois l'opération suivante (`taille` désigne la taille de la liste `individus` au départ) : choisir deux entiers x et y distincs au hasard entre 0 et taille-1, ajouter à la liste `individus` le résultat du croisement des éléments numéros x et y de la liste `individus` (pour croiser deux éléments, utiliser la méthode `croisement`).
3. Programmez une méthode de classe `List mutation(List individus)` permettant de muter tous les éléments de la liste paramètre (on suppose que tous les éléments de la liste sont des instances de `Génome`). Pour cela vous utiliserez la méthode `mutation` (on suppose que cette méthode agit par effet de bord).
4. Programmez une méthode de classe `List generation(List individus,int n,int m)`. Cette méthode génère une nouvelle liste de solutions de taille n en séléctionnant les m meilleures solutions dans la liste paramètre (on suppose que tous les éléments de la liste sont des instances de `Génome`). Pour générer une nouvelle génération, on séléctionne les m meilleures solutions, puis on génére par croisement $n - m$ nouvelles solutions, enfin on mute chacune des solutions.

2 La classe MotGenetique

Le but de cette partie est de programmer une classe `MotGenetique` implantant l'interface `Genome` et permettant de résoudre le problème du mot mystère. On suppose que le mot mystère est stocké dans une variable de classe `mystere` de type `String`. Chaque `MotGenetique` possède une variable d'instance `code` de type `String` dont le nombre de caractère est le même que celui du mot mystère. En plus des variables déja mentionnées la classe `MotGenetique` contient une variable de classe `tauxMutation` de type `double` initialisée à 5 (=5% de mutations).

1. Programmez un constructeur sans paramètre pour la classe `MotGenetique`. Ce constructeur est chargé de creer la chaine de caractère `code` en utilisant des caractères choisis au hasard (la taille de code à la fin doit être la même que celle du mot mystère!). Pour générer un caractère au hasard il suffit d'utiliser : `(char)('a'+alea.nextInt('z'-'a'+1))`, où `alea` est une instance de `Random`.
2. Programmez un constructeur prenant en paramètre une chaine de caractère qui est stocké dans la variable `code`.
3. Programmez la méthode `public void mutation()`. Cette méthode transforme la chaine de caractère `code` par mutation. Concrètement pour chaque caractère de `code` on tire un entier

au hazard entre 0 et 99, si cet entier est plus petit que `tauxMutation` on remplace le caractère par un caractère choisi au hazard sinon on garde ce caractère non modifié.

4. Programmez la méthode `public Genome croisement(Genome parent)`. Cette méthode permet de croiser le `MotGenetique` appelant avec le `MotGenetique` paramètre. Pour cela on tire un entier n au hasard entre 0 et `taille-1` (où `taille` est la longueur du mot mystère). On construit une nouvelle chaine de caractère dont les n premiers caractères sont donnés par le `code` du `MotGenetique` appelant et dont les 'taille-n' caractères suivants sont donnés par le `code` du `MotGenetique` paramètre. Puis on renvoie un nouveau `MotGenetique` dont la variable `code` est précisemment cette chaîne.
5. Programmez la méthode `public int compareTo(Object obj)` permettant de comparer deux `MotGenetique`. Soit n le nombre de lettres dans le code du `MotGenetique` appelant coincidant avec des lettres du mot mystère et soit m le nombre de lettres dans le code du `MotGenetique` paramètre coincidant avec des lettres du mot mystère. La méthode `compareTo` renvoie n-m.
6. Programmez une méthode `main` permettant de trouver par algorithme génétique le mot mystère "alea". Pour cela vous utiliserez des listes de 20 solutions.

Chapitre 5

L'itération abstraite

1 Introduction

Dans notre étude de `Java`, nous avons rencontré deux structures de données qui peuvent contenir un nombre arbitraire de références vers des objets : il s'agit des tableaux classiques (en particulier `Object[]`) et des `Lists` (qui se divisent dans la pratique en `ArrayLists` et `LinkedLists`). Ces structures ont en commun le fait que leur contenu est numéroté. Pourtant, nous ne connaissons pas de moyen simple d'écrire une boucle qui passe en revue (qui parcourt) tous les éléments d'une structure *sans tenir compte* du type de celle-ci. On introduit la technique de **l'itération abstraite** pour répondre à cette question.

La notion de conteneur (objet contenant des références vers d'autres objets) n'implique pas la numérotation du contenu. On peut vouloir par exemple représenter informatiquement la notion d'ensemble, sans pour autant vouloir numéroter les éléments. Or, sans numérotation, nous ne savons pas écrire de boucle : comment faire pour passer en revue tous les éléments d'un conteneur ensembliste ?

2 L'interface `Iterator`

2.1 Les méthodes

La solution retenue en `Java` se base sur l'interface `Iterator`. Avant de passer en revue ses méthodes, reprenons un exemple élémentaire de parcours d'un tableau :

```
int[] t={1,2,3} ;
int somme=0 ;
for (int i=0;i<t.length;i++)
        somme+=t[i] ;
```

On voit que pour écrire cette boucle, on utilise les éléments suivants :

1. un test pour savoir s'il reste des éléments (`i<t.length`) ;
2. une opération qui permet de passer à l'élément suivant (`i++`) ;
3. une "méthode" d'accès à l'élément courant (`t[i]`).

Nous utilisons exactement les mêmes "ingrédients" pour réaliser une boucle travaillant sur une `List`, comme par exemple :

```
List l=/* initialisation */
for (int i=0;i<l.size();i++)
        System.out.println(l.get(i)) ;
```

Dans cette version, le test devient `i<l.size()`, le passage au suivant reste identique, alors que l'accès devient `l.get(i)`.

Pour pouvoir réaliser une boucle, ou plus précisément pour pouvoir parcourir un ensemble d'éléments, il faut donc disposer de ces trois opérations. Dans l'interface `Iterator`, l'opération d'accès à un élément et de passage au suivant sont regroupées, alors qu'une méthode de suppression d'un élément est ajoutée. On obtient l'interface suivante :

Iterator

```
1 public interface Iterator {
2   public boolean hasNext();
3   public Object next();
4   public void remove();
5 }
```

2.2 Exemple d'utilisation

Commençons par un exemple simple d'utilisation, basé sur le fait que l'interface `List` (et donc les classes qui l'implantent) possède une méthode `iterator`, sans paramètre, qui renvoie un `Iterator`[1]. Considérons donc le programme suivant :

```
import java.util.*;
public class TestListIterate {
        public static void main(String[] arg) {
                List l=new ArrayList();
                l.add("AB");
                l.add("CD");
                l.add("EF");
                l.add("GH");
                for(Iterator i=l.iterator();i.hasNext();)
                        System.out.println(i.next());
        }
}
```

Ce programme affiche :

```
AB
CD
EF
GH
```

Il est donc clair que l'utilisation de l'objet `Iterator` permet de parcourir tous les éléments de la liste. Voici comment :

1. l'appel `l.iterator()` fabrique un objet dont la classe implante l'interface `Iterator`. On peut considérer qu'il s'agit de l'initialisation d'un index permettant de se repérer dans la liste (on a donc quelque chose de semblable à `int i=0`) ;
2. l'appel `i.hasNext()` permet de savoir s'il reste des éléments à étudier (à parcourir) et joue donc un rôle similaire à `i<l.size()` ;
3. le point complexe est `i.next()`. Cet appel réalise deux choses : il renvoie un nouvel élément à étudier, et prépare l'index `i` pour l'itération suivante. En une instruction, on réalise donc `l.get(i)` et `i++`.

[1] Cette méthode n'a pas été décrite dans la section consacrée à l'interface List pour éviter d'encombrer la présentation.

Remarque 1. En fait, il est possible de calquer complètement une boucle classique sur la boucle basée sur `Iterator`. En effet, l'appel `l.get(i++)` réalise en une seule instruction `l.get(i)` et `i++`. Ceci est une conséquence de la sémantique de `i++`. Cette instruction incrémente la variable `i` mais elle possède aussi une *valeur* qui est celle de `i` *avant* l'incrémentation. On peut donc remplacer la boucle de l'exemple `TestListIterate` par la boucle suivante, très semblable :

```
for(int i=0;i<l.size();)
  System.out.println(l.get(i++));
```

3 Principe général

3.1 Sens à accorder aux méthodes

Un objet dont la classe implante l'interface `Iterator` est appelé un **itérateur**, ou encore un **curseur**. Il peut être considéré comme désignant un **élément courant** dans une collection (c'est-à-dire dans un ensemble d'éléments). En fait, un itérateur est un indice représenté sous forme d'un objet. Voici une documentation des méthodes de l'interface `Iterator` :

`boolean hasNext()`
: Renvoie `true` si et seulement si le curseur désigne encore un élément courant. En d'autres termes, renvoie `true` si et seulement si l'appel à la méthode `next` est possible.

`Object next()`
: Renvoie l'élément courant et passe au suivant (s'il n'existe pas de suivant, cette méthode ne provoque pas d'erreur). S'il n'existe pas d'élément courant, l'appel à cette méthode provoque une erreur.

`void remove()`
: Supprime de la collection parcourue par le curseur le dernier élément qui a été visité (c'est-à-dire le dernier élément renvoyé par `next`). Il faut bien sûr qu'un appel à `next` ait eu lieu avant tout appel à `remove`.

Le principe d'utilisation d'un `Iterator` pour réaliser une boucle est très simple : on commence par obtenir un itérateur par une méthode adaptée de la collection étudiée (il est impossible de créer directement un `Iterator`, car c'est une interface). Ensuite, tant qu'il reste un élément courant (testé par `hasNext`), on traite cet élément en le récupérant par `next`.

3.2 Abstraction de l'itération

Il bien sûr possible d'écrire une méthode prenant comme paramètre un `Iterator`. Une telle méthode permet de réaliser une boucle sur un ensemble d'éléments de façon abstraite : en effet, elle ne connaît absolument pas le type de la collection qui est parcourue par l'itérateur. Voici un exemple élémentaire d'une telle méthode :

```
import java.util.*;
public class IterationAbstraite {
        public static void display(Iterator i) {
                while(i.hasNext())
                System.out.println(i.next());
        }
        public static void main(String[] args) {
                List l=new LinkedList();
                l.add(new Double(2.5));
```

```
                l.add("Test");
                l.add(new Integer(2));
                display(l.iterator());
                System.out.println("----------");
                Iterator i=l.iterator();
                i.next(); // on jette le premier élément
                display(i);
        }
}
```

On voit qu'il est bien sûr possible de manipuler l'itérateur avant de le transmettre à la méthode, ce qui permet d'éliminer des éléments. Réaliser la même chose avec une boucle classique demande de passer un paramètre de plus à la méthode, l'indice de départ de la boucle. Le programme affiche les lignes suivantes :

```
2.5
Test
2
-------
Test
2
```

3.3 Utilisation de `remove`

La méthode `remove` est très pratique car elle permet, en conjonction avec `next`, de supprimer un élément qui vient d'être étudié. Considérons par exemple le problème suivant : étant donné une liste `l`, on veut supprimer de la liste tous les éléments qui ne sont pas des `String`. Ce problème est résolu par le programme suivant :

```
List l=/* creation et initialisation */
Iterator i=l.iterator() ;
while (i.hasNext()) {
        if (!(i.next() instanceof String))
                i.remove() ;
}
```

En pratique cette méthode (avec `Iterator`) sera toujours au moins aussi rapide que la méthode correspondante utilisant les méthodes `get` et `remove`. En particulier le gain de vitesse le plus important sera obtenu dans le cas ou `l` est une `LinkedList`.

3.4 Implantation de l'interface `Iterator`

Les itérateurs peuvent sembler un peu mystérieux pour l'instant. Pour lever en partie ce mystère, nous allons étudier comment produire un `Iterator` pour une classe représentant les tableaux dynamiques. Supposons que nous ayons programmé une classe `TableauDynamique` comme suit (seule les deux méthodes importantes pour la suite sont indiquées) :

```
public class TableauDynamique {
        ...
        public int size() ;
        public Object  get(int i) ;
        ...
}
```

Gardons bien à l'esprit qu'un itérateur est simplement un index représenté sous forme d'objet. Ceci est particulièrement vrai pour les tableaux dynamiques dans lesquels la notion de numéro d'un élément est parfaitement applicable. Pour réaliser un itérateur travaillant pour les `TableauDynamique`s, on va donc créer un objet contenant un index sous forme d'un entier. Considérons la classe suivante :

```
import java.util.*;
public class TableauDynamiqueIterator implements Iterator {
        private TableauDynamique tableau;
        private int position;

        public TableauDynamiqueIterator(TableauDynamique leTableau) {
                tableau=leTableau;
                position=0;
        }
        public boolean hasNext() {
                return position<tableau.size();
        }
        public Object next() {
                // doit produire une erreur si position>=tableau.size()
                return tableau.get(position++);
        }
        public void remove() {
                // doit produire une erreur si position<=0
                position--;
                tableau.remove(position);
        }
}
```

Cette classe demande quelques explications :

- pour pouvoir parcourir un tableau, il faut bien sûr le connaître : c'est le rôle de la variable d'instance `tableau` ;
- la variable `position` contient l'index (le numéro) de l'élément courant. Au départ, l'index est bien entendu positionné en `0` ;
- la méthode `hasNext` teste si l'index courant est valide, c'est-à-dire s'il correspond à une case du tableau dynamique ;
- la méthode `next` utilise la construction indiquée dans une remarque précédente pour réaliser en une seule opération l'accès à l'élément d'index `position` et l'incrémentation de cette variable. On suppose que l'accès en dehors du tableau provoque une erreur ;
- la méthode `remove` est un peu plus délicate. Elle doit supprimer le dernier élément renvoyé par `next`. Comme `next` incrémente `position` de 1, la position du dernier élément renvoyé est donc `position-1`. Mais, quand on supprime cet élément, on décale toute la fin du tableau vers la gauche. S'il existe un élément d'index `position` par exemple, il se retrouve dans la case numéro `position-1` après la suppression. C'est pourquoi la méthode `remove` doit décrémenter la valeur de `position`.

Pour pouvoir utiliser cet itérateur, on ajoute à la classe `TableauDynamique` la méthode suivante :

```
public Iterator iterator() {
        return new TableauDynamiqueIterator(this) ;
}
```

4 Intérêt des itérateurs

Les itérateurs possèdent les avantages suivants :

Itération abstraite

La notion d'itération pour le parcours d'une collection est maintenant totalement abstraite, ce qui permet d'écrire une boucle sur un ensemble d'éléments sans pour autant savoir comment cet ensemble est stocké.

Efficacité

C'est le point le plus important. Comme nous l'avons dit les listes chaînées (les `LinkedList`s) sont efficaces pour l'ajout ou la suppression d'un élément, mais inefficaces pour l'accès à un élément quelconque (le temps d'accès est proportionnel au nombre d'éléments contenus dans la liste). Grâce aux itérateurs, on peut écrire une boucle sur une `LinkedList` qui sera aussi efficace qu'une boucle équivalente sur un `ArrayList`. La technique employée dépasse le cadre de ce chapitre, mais aucune connaissance particulière n'est nécessaire pour utiliser les `Iterator`s des `LinkedList`s !

Suppression de la numérotation

L'utilisation d'un itérateur supprime la notion de numérotation des éléments (naturelle dans un tableau). De ce fait, un itérateur peut être utiliser pour parcourir l'ensemble des éléments d'une collection générale, sans faire d'hypothèse sur la nature de cette collection, comme nous allons le voir dans la section suivante.

PROJET V

CALCUL DE π EN MULTIPRÉCISION

Le but des questions suivantes est d'écrire un programme permettant de calculer le nombre π avec un grand nombre de décimales (cf π avec 3000 à la fin du texte).

Chacun des exercices permet d'écrire une partie du programme final. En particulier les exercices ne sont pas indépendants les uns des autres. Néammoins vous pouvez utiliser dans chaque exercice les méthodes définies dans les exercices précédents mêmes si vous ne les avez pas programmées.

1 Nombres en multiprécision

1.1 La classe Chiffre

La classe `Chiffre` dont le squelette est donné table 1 permet de représenter des chiffres dans une base arbitraire (fixée à 10 dans le squelette) et d'effectuer les opérations arithmétiques élémentaires sur des `Chiffre`s. La variable `valeur` contient la valeur représenté par le `Chiffre` (en particulier `valeur` est compris entre 0 et `BASE-1` au sens large).

```
public class Chiffre {

        /* base pour les calculs */
        public static int BASE=10 ;
        /* valeur representee */
        private int valeur ;

        public Chiffre(int valeur) ;

        /* renvoie la valeur representee */
        public int valeur() ;
        /* renvoie true si le chiffre appelant représente zero */
        public boolean isZero() ;

        public Chiffre plus(Chiffre a,Chiffre retenue) ;
        public Chiffre moins(Chiffre a,Chiffre retenue) ;
        public Chiffre fois(Chiffre a,Chiffre retenue) ;
        public Chiffre divise(Chiffre a,Chiffre retenue) ;

        public String toString() ;

}
```

TAB. 1 – Squelette de la classe Chiffre

1. Programmer le constructeur, les méthodes `valeur` et `isZero`.

2. La méthode `plus` renvoie la somme du `Chiffre` appelant et du paramètre en tenant compte de la retenue (et en la modifiant si necessaire). Soient u, v et r trois entiers. La somme s de u et v avec la retenue r est donnée par l'algorithme suivant :
   – $s = u + v + r$
   – si $s >= BASE$ alors enlever $BASE$ à s et poser $r = 1$
   – sinon poser $r = 0$
   (remarquez que l'algorithme modifie la valeur de la retenue r).
   Programmez la méthode `plus` (la méthode doit modifier la valeur de `retenue` en agissant par effet de bord).
3. La méthode `moins` renvoie la différence du `Chiffre` appelant avec le `Chiffre` paramètre en tenant compte de la retenue. Soient u, v et r trois entiers. La différence d de u et v avec la retenue r est donnée par l'algorithme suivant :
   – $d = u - v - r$
   – si $d < 0$ alors ajouter $BASE$ à d et poser $r = 1$
   – sinon poser $r = 0$
   Programmez la méthode `moins` (la méthode doit modifier la valeur de `retenue` en agissant par effet de bord).
4. La méthode `fois` renvoie le produit du `Chiffre` appelant avec le `Chiffre` paramètre en tenant compte de la retenue. Soient u, v et r trois entiers. Le produit p de u et v avec la retenue r est donné par :

$$p = (u \times v + r)/BASE$$
$$r = (u \times v + r) \,\text{modulo}\, BASE$$

   Programmez la méthode `produit` (la méthode doit modifier la valeur de `retenue` en agissant par effet de bord).
5. La méthode `divise` renvoie le quotient du `Chiffre` appelant par le `Chiffre` paramètre en tenant compte de la retenue. Soient u, v et r trois entiers. Le quotient q de u et v avec la retenue r est donné par :

$$q = (u + r \times BASE)/v$$
$$r = (u + r \times BASE) \,\text{modulo}\, v$$

6. Programmez la méthode `toString`.

1.2 Programmation des entiers naturels

La classe `Naturel` dont le squelette est donné table 2 permet de représenter des entiers naturels (c'est à dire des entiers positifs ou nuls) de taille arbitrairement grande. Par exemple un `Naturel` peut représenter un entier comprenant 200 chiffres. Chaque entier naturel est représenté par la liste de ses `Chiffre`s contenue dans la variable `chiffres`. Par exemple l'entier 12 est représenté par la liste $\{2, 1\}$, l'entier 31415 est représenté par la liste $\{5, 1, 4, 1, 3\}$ (remarquez que l'ordre des chiffres est inversé).

1. Programmez les constructeurs de la classe `Naturel`. Le constructeur sans paramètre se contente d'initialiser la liste `chiffres`. Le constructeur avec le paramètre `int n` construit la représentation de l'entier `n` sous forme de liste de `Chiffre`s (on suppose $n >= 0$). Par exemple si $n = 12$ on doit obtenir la représentation $\{2, 1\}$.
2. Programmez les méthodes `size` et `get` (attention `get` doit toujours renvoyer le bon résultat même si le paramètre `i` est plus grand que la taille de la liste des chiffres).

```
import java.util.* ;

public class Naturel {
        /* liste des chiffres */
        private List chiffres ;

        private Naturel() ;
        public Naturel(int n);

        /* renvoie la taille de la liste des chiffres */
        private int size() ;
        /* renvoie le i-ème chiffre */
        private Chiffre get(int i) ;
        /* renvoie true si le naturel appelant représente zero */
        public boolean isZero() ;

        private Naturel normalise() ;

        public String toString() ;

        public Naturel plus(Naturel nb) ;
        public Naturel moins(Naturel nb) ;
        public void exposant(int puissance) ;
        public Naturel fois(Chiffre c,int puissance) ;
        public Naturel fois(Naturel nb) ;
        public Naturel divise(Chiffre c) ;
        public Naturel[] divise(Naturel nb) ;

}
```

TAB. 2 – Squelette de la classe Naturel

3. En général il existe plusieurs représentation du même entier naturel par des listes de chiffres. Par exemple 12 peut être représenté par la liste $\{2,1\}$ mais aussi par $\{2,1,0\}$, $\{2,1,0,0\}$, etc (en effet $12 = 012 = 0012 = \ldots$). La méthode `normalise` permet d'éliminer les zeros inutiles afin de revenir à la représentation la plus simple. Programmez cette méthode.
4. Programmez la méthode `isZero`. On considère qu'un naturel représente zero si la liste de ses chiffres ne contient que des zeros.
5. Programmez la méthode `toString`.
6. La méthode `plus` renvoie la somme du `Naturel` appelant et du `Naturel` paramètre. Pour calculer cette somme il suffit d'appliquer l'algorithme appris en cours élémentaire (addition chifffre à chiffre avec retenue). Par exemple la somme de 9738 et 513 se calcul somme suit (sur la première ligne apparaissent les valeurs successives de la retenue) :

$$\begin{array}{cccccc} & 1 & 1 & 0 & 1 & \\ & & 9 & 7 & 3 & 8 \\ + & & & 5 & 1 & 3 \\ \hline & 1 & 0 & 2 & 5 & 1 \end{array}$$

Programmez la méthode `plus` en vous inspirant de cet exemple. Pour parcourir les listes de chiffres vous utiliserez impérativement des `Iterator`.

Le même type d'algorithme étant utilisé pour la programmation des méthodes `moins` et `fois`, il est recommandé de passer du temps sur cette question afin de verifier que vous n'avez pas fait d'erreur.

7. La méthode `moins` renvoie la difference du `Naturel` appelant et du `Naturel` paramètre. Pour que cette opération soit possible dans les entiers naturels, on suppose que le paramètre est plus petit que le naturel appelant. Le calcul de la différence se fait alors de façon similaire au cas de la somme (chiffre à chiffre avec retenue). Par exemple la différence de 1234 et 715 se calcul comme suit (sur la première ligne apparaissent les valeurs successives de la retenue) :

$$\begin{array}{cccccc} & {\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 1} & & \\ & 1 & 2 & 3 & 4 & \\ - & & 7 & 1 & 5 & \\ \hline & 0 & 5 & 1 & 9 & \end{array}$$

Programmez la méthode `moins` en vous inspirant de cet exemple. Pour parcourir les listes de chiffres vous utiliserez impérativement des `Iterator`. Attention : comme vous pouvez le constater sur l'exemple : le résultat n'est pas forcément normalisé (il peut y avoir des zeros inutiles) ; n'oubliez pas de le normaliser avant de le renvoyer.

8. La méthode `exposant(int puissance)` renvoie le produit du `Naturel` appelant et de `BASE` à la puissance `puissance` (concrètement il suffit d'ajouter `puissance` zeros au début de la liste des chiffres si `puissance>0` ou de supprimer les `-puissance` premier chiffres si `puissance<0`). Programmez cette méthode.

9. La méthode `fois(Chiffre c)` renvoie le produit du `Naturel` appelant at du `Chiffre c`. Par exemple si le naturel appelant représente 32, et que `c` représente 2 alors le résultat sera 64. Programmez la méthode `fois(Chiffre c,int puissance)`.

10. La méthode `fois(Naturel nb)` renvoie le produit du `Naturel` appelant et du `Naturel` paramètre. Le produit est donné par l'algorithme appris en cours élémentaire ; par exemple le produit de 43 et 227 se calcul comme suit (la dernière ligne contient la somme des produits partiels) :

$$\begin{array}{lcccccc} & & & & 4 & 3 & \\ & \times & & 1 & 2 & 7 & \\ \hline 7 \times 43 = & & & 3 & 0 & 1 & \\ 2 \times 43 = & & & 8 & 6 & & \\ 1 \times 43 = & & 4 & 3 & & & \\ \hline & & 5 & 4 & 6 & 1 & \end{array}$$

Programmez la méthode `fois(Naturel nb)` (utilisez la question précédente).

11. On veut que la classe `Naturel` implante l'interface `Comparable`. Indiquez les modifications nécessaires (y compris la programmation éventuelle de nouvelles méthodes).

12. La méthode `divise(Chiffre c)` renvoie le quotient du naturel appelant par le chiffre paramètre. Programmez cette méthode. Pour cela vous pourrez vous inspirer de l'exemple suivant (attention :les chiffres du résultat sont calculés de gauche a droite et non pas de droite à gauche comme dans les exemples précédents, la retenue la plus à droite est le reste de la

division) :

$$\begin{array}{cccc} & & 1 & 2 & 1 \\ & 1 & 7 & 2 & \\ \text{div} & & & 3 & \\ \hline & 0 & 5 & 7 & \end{array}$$

Programmez la méthode `divise(Chiffre c)`.

13. La division de deux entiers naturels est beaucoup plus difficile que les autres opérations arithmétiques en particulier parceque à chaque étape il faut deviner le quotient partiel. L'algorithme permettant de diviser u par v est le suivant (on suppose que u_i désigne le i-ème chiffre de u : $u = u_n u_{n-1} \dots u_1 u_0$, de même v_i désigne le i-ème chiffre de v : $v = v_m u_{m-1} \dots v_1 v_0$) :
   - si u est plus petit que v alors le quotient est zero et le reste est u (l'algorithme est terminé)
   - multiplier u et v par $facteur = \text{BASE}/(1 + v_m)$
   - pour i allant de n à zero
     - si i est plus grand que la taille de u ajouter un zero comme premier chiffre du quotient puis passer à l'itération suivante
     - soit $q = \min(\text{BASE} - 1, (u_i + \text{BASE}\, u_{i-1})/v_m$
     - tant que $q \times v$ est plus grand que u retirer 1 à q
     - retirer $q \times v$ à u
     - ajouter le chiffre q à la fin des chiffres du quotient
   - le reste de la division est $u/facteur$.

   Verifiez que cet algorithme fonctionne sur des exemples simples. Programmez la méthode `divise(Naturel nb)`.

1.3 Programmation des entiers relatifs

La classe `Entier` permet de représenter les entiers relatifs (positifs ou negatifs). Son squelette est donné dans la table 3. Programmez cette classe (attention aux signes pour les opérations arithmétiques).

1.4 Programmation des nombres réels en précision fixée

Grace à la classe `Entier` il est possible de créer une nouvelle classe représentant des nombres réels en précision arbitraire. Le squelette de la classe `Reel` représentant les nombres réels est représenté table 4. Le principe de cette classe est le suivant : chaque nombre réel est représenté par un entier (sa mantisse). Soit b la base (`Chiffre.BASE`) et p la précision (`precision`), pour obtenir le nombre réel correspondant à une mantisse donnée il suffit de calculer eb^{-p}. Ainsi pour représenter un entier n il faut que la mantisse soit égale à nb^p. Les opérations arithmétiques sur les réels se définissent en utilisant les formules suivantes :

$$\begin{aligned} e_x\,\mathrm{b}^{-p} + e_y\,\mathrm{b}^{-p} &= (e_x + e_y)\,\mathrm{b}^{-p} \\ e_x\,\mathrm{b}^{-p} - e_y\,\mathrm{b}^{-p} &= (e_x - e_y)\,\mathrm{b}^{-p} \\ e_x\,\mathrm{b}^{-p} \times e_y\,\mathrm{b}^{-p} &= (e_x \times e_y) b^{-p}\,\mathrm{b}^{-p} \\ (e_x\,\mathrm{b}^{-p})/(e_y\,\mathrm{b}^{-p}) &= (e_x b^p / e_y)\,\mathrm{b}^{-p} \end{aligned}$$

Programmez la classe `Reel`.

2 Calcul du nombre π

Les formules les plus simples permettant de calculer le nombre π sont basées sur l'identité suivante :

$$\pi = 4\,\text{atan}(1) \tag{1}$$

```
public class Entier {
        private boolean signe ;
        private Naturel n ;

        public Entier(int n) ;

        private Entier(Naturel n,boolean signe) ;

        public Entier plus(Entier e) ;
        public Entier moins(Entier e) ;
        public void exposant(int puissance) ;
        public Entier fois(Chiffre c) ;
        public Entier fois(Entier e) ;
        public Entier divise(Chiffre c) ;
        public Entier[] divise(Entier e) ;
        public String toString() ;

}
```

TAB. 3 – Squelette de la classe Entier

```
public class Reel {
        /* nombre de chiffres après la virgule (en base Chiffre.BASE) */
        public static int precision=100 ;
        /* mantisse */
        private Entier e ;

        /* ce constructeur construit un nombre réel représentant l'entier n */
        public Reel(int n) ;

        /* ce constructeur construit l'entier e * BASE^{-precision} */
        private Reel(Entier e) ;

        public Reel plus(Reel r) ;
        public Reel moins(Reel r) ;
        public Reel fois(Chiffre c) ;
        public Reel fois(Reel r) ;
        public Entier divise(Chiffre c) ;
        /* la méthode divise(Reel r) agit par effet de bord sur le reel appelant */
        public void divise(Reel r) ;


        public String toString() ;
}
```

TAB. 4 – Squelette de la classe Reel

Pour calculer la valeur de l'arctangente d'un nombre il suffit d'utiliser sa série de Taylor (convergente dans le disque du plan complexe de rayon 1) :

$$\text{atan}(x) = \sum_{k=0}^{k=+\infty} (-1)^k x^{2k+1}/(2k+1)$$

Malheureusement avec la formule 1 donnée plus haut pour π la convergence de la série sera très lente (les restes décroissent en $1/k$). Afin d'obtenir une vitesse de convergence beaucoup plus rapide on utilise des formules dérivées de la formule 1. Une formule donnant de très bon résultats est la suivante :

$$\pi = 48\,\text{atan}(1/18) + 32\,\text{atan}(1/57) - 20\,\text{atan}(1/239) \tag{2}$$

Comme vous pouvez vous en convaincre la convergence de la serie de Taylor aux points considérés est très rapide. Pour calculer π il nous suffit de savoir calculer des expressions de la forme $\text{atan}(\frac{1}{x})$ où x est un entier.

Le calcul de $\text{atan}(\frac{1}{x})$ est donné par l'algorithme suivant (on calcul l'arctangente en utilisant un schéma proche de la méthode de Hörner) :

- soit n la partie entière du nombre $1 + (\text{precision} \times \log(\text{BASE})/\log(x) - 3)/2)$ (n est le nombre d'étapes de calcul nécessaires étant données la base et la précision souhaitée)
- soit `resultat` un réel initialisé à 1.
- pour i allant de n à 1
  - effectuer la transformation :

$$\text{resultat} \mapsto 1 - \frac{\text{resultat} \times (2i-1)}{x^2 \times (2i+1)}$$

- diviser le résultat par x

1. Ecrire une méthode de classe `Reel atan_inv(int x)` realisant cet algorithme.
2. Ecrire une méthode main renvoyant π avec 50 décimales

Avec le programme décrit dans les pages précédentes on obtient en quelques secondes la valeur de π avec 3000 décimales (sur un ordinateur relativement récent) :

```
pi=3.
14159265358979323846264338327950288419716939937510582097494459230781640628620899862834825342117679821488651328230664793844609550582231725359
40812848111745028410270193852110555964462294895493038196442881097566593344612847564823378678316527120190914564856692346034861045432664821339
36072602491412737245876606315588174881520920962829254917153643678925903600113305305488204665213841469519415116094335727036575959195392186117381932611793105118548744623799627495673518857527248912279381831194912983367336244065664386213949463952247371907021798609437027705392171762931767523846748184676694051320568127145263560827785771342757789609173637178721468440901224953430146549585371507922796892589235421995611212921960864344181598136297747713099605187072113499999983729784995159731732816963185952445945534690830264252230825334468535261931188171103137838752886587533208381420617177669147303598253490428755468731159562863882353787593751957781857780532171226806613192787661119590921642019893895257201654858632788659361533818279682303019523530185296899577362259941389124972177528347913151557485724245415069595082953311686172785588975983817546374649393192556040092770167113998488241285836160356370766010471018194295559619894676783744944825537977472684710447534646208466842596949129331367728989152104752162569662458381501935112533824303558764024749647326391419927260426992279678235478163693417216412199245863153028618297455570674983850549458858692699569092721079750930295532116534498720275596023648066549911988183479775356636980742654252786255181841757467289097777279380081647601614524919217321721477235014144197356854816136115735255213347574184946843852332390739414333454776241686251898356948556299219222184272550254256887671790494601653466804988627232791786857843838279679766814541009538837863609506806422512520511739298489684128488626945604241965285222106611863067442786223919494547123713786960956364371917287467764657573962413890865832645995813390478275909946576478951269468398352595709825822620522489407726719478268482601476990902640136394437455305682034962524517493996514314298091906592509372216964615157098583874105978859772975498930161753928468138268683868942774155991855925245953959431499725246808459872736446958486538367362226269912460851243884390451243149972524680845987273644695848653836736222626991246085124388439045124413654976278079771569143599770012961689441694868555848406353422722582848864815845628506168427394522674676788952521385225499546667278239864565961163548862305774564983559363456817432411251576694794519659694252278959772975498930161753928468138268683868942774155991855925245953959431499725246808459872736446958486538367362226269912460851243884390451244136549762780797715691435997700129616894416948685558484063534227222582848864815845628506168427394522674676788952521385225499546667278239864565961163548862305774564983559363456817432411251576694794519659694252288797108931456691368672287489405601015033086179286809208747609178249385899714909675985261365549781893129784821682998948722658804857564142704775551323796414515237462343645428584447952658678215114135473573952311342716610213596953623144295248493718711014576540359027993440374273105785396219838744780847848968332144571386875194350643021845319148481537061468067491927819119793995261419663428754440643745123718192179998391159195618146751426912397489409071864942319 16
```

Chapitre 6

Les `Collections`

1 Principe

L'idée de base des **collections** est simplement de généraliser la notion de liste. Une liste permet de stocker un ensemble d'éléments, mais attribue à chacun d'eux un numéro. Dans la pratique, ce numéro est parfois inutile. De plus, une liste peut contenir plusieurs fois le même élément (à des numéros distincts), ce qui est parfois en contradiction avec les besoins pratiques (si on souhaite représenter un ensemble au sens mathématique). On peut se dire que l'utilisation d'un numéro n'est pas très grave : rien n'oblige à le prendre en compte. Malheureusement, la numérotation n'est pas anodine : elle oblige à utiliser certaines structures (les tableaux dynamiques ou les listes chaînées) qui possèdent des défauts (l'accès à un élément est relativement lent pour les listes chaînées, alors que c'est l'ajout d'un élément qui est lent dans le cas des tableaux dynamiques). De plus, il existe des techniques permettant de représenter un ensemble de façon beaucoup plus efficace qu'avec un tableau dynamique ou une liste chaînée : ce sont par exemple les techniques à base de table de hachage. Or, ces techniques ne peuvent pas donner simplement un numéro à chaque élément contenu dans l'ensemble qu'elles représentent.

Pour tenir compte de tout ceci, `Java` définit une interface `Collection` qui représente des collections d'éléments. Une collection est un groupe fini d'éléments. La notion recouvre à la fois les ensembles (une occurrence d'un élément donné dans le groupe, pas de numéro) et les listes (éléments numérotés, plusieurs occurrences possibles pour chaque élément). La figure 1 récapitule les différentes interfaces et classes liées à `Collection`.

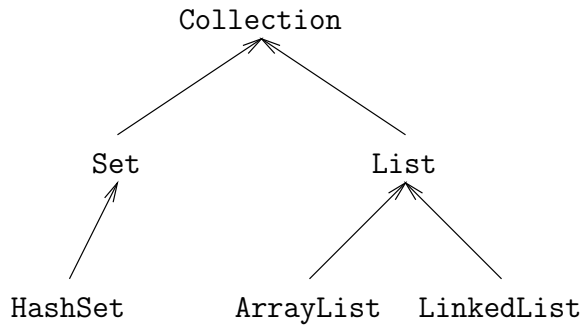


FIG. 1 – L'interface `Collection`, ses sous-interfaces et leurs implantations

2 L'interface `Collection`

L'interface `Collection` propose les méthodes suivantes :

`boolean add(Object o)`

Ajoute à la collection appelante l'objet `o`. Renvoie `true` si l'ajout s'est bien effectué, et `false` si l'objet était déjà contenu dans la collection et qu'elle refuse de le contenir en double.

`boolean addAll(Collection c)`

Ajoute à la collection appelante tous les éléments contenus dans la collection paramètre `c`. Renvoie `true` si et seulement si un élément au moins a été ajouter à la collection appelante (en d'autres termes si la collection appelante a été modifiée).

`void clear()`

Supprime tous les éléments contenus dans la collection appelante.

`boolean contains(Object o)`

Renvoie `true` si et seulement si la collection appelante contient au moins une occurrence de `o` (au sens de la méthode `equals`).

`boolean containsAll(Collection c)`

Renvoie `true` si et seulement si la collection appelante contient tous les éléments contenus dans la collection paramètre (teste donc l'inclusion de `c` dans la collection appelante).

`boolean isEmpty()`

Renvoie `true` si et seulement si la collection appelante est vide.

`Iterator iterator()`

Renvoie un `Iterator` permettant de parcourir tous les éléments de la collection appelante. L'ordre de parcours n'est pas garanti en général (i.e., le premier élément obtenu n'a aucune raison d'être le premier élément inséré dans la collection). De plus, l'ajout d'un élément peut parfaitement modifier radicalement l'ordre de parcours de la collection.

`boolean remove(Object o)`

Supprime une apparition de `o` (au sens de `equals`) dans la collection appelante. Renvoie `true` en cas de suppression effective et `false` si la collection ne contenait pas d'occurrence de `o`.

`boolean removeAll(Collection c)`

Supprime de la collection appelante tous les éléments contenus dans la collection paramètre `c`. Après l'appel la collection appelante le contiendra plus aucun élément en commun avec la collection `c`. Renvoie `true` si et seulement si la collection appelante a été modifiée par l'appel.

`boolean retainAll(Collection c)`

Conserve dans la collection appelante les éléments qui apparaissent aussi dans la collection paramètre. Supprime tous les autres éléments. Renvoie `true` si et seulement si la collection appelant a été modifiée par l'appel.

`int size()`

Renvoie le nombre d'éléments contenus dans la collection appelante.

`Object[] toArray()`

Fabrique un tableau contenant exactement les mêmes références que la collection appelante.

On retrouve bon nombre des méthodes de l'interface `List`, ce qui est parfaitement normal : l'interface `List` est obtenue par extension de l'interface `Collection`. Certaines méthodes semblent nouvelles car dans la présentation de `List` nous avons omis pour des raisons pédagogiques les méthodes faisant intervenir des `Collection`s ou des `Iterator`s.

On remarque bien sûr qu'aucune méthode de `Collection` n'utilise d'indice pour repérer les objets. De ce fait, on peut ajouter des éléments, supprimer des éléments (mais seulement au sens de l'égalité, c'est-à-dire supprimer une occurrence de `o`, pas l'élément numéro `i`), tester si un élément est contenu dans la collection, et surtout **parcourir** la collection, grâce à la méthode `iterator`.

Du point de vue pratique, on ne perd par grand chose par rapport à une `List`, en particulier si le but est le stockage d'objets, sans tenir compte de l'ordre dans lequel ils sont ajoutés à la collection. Par contre, on mesure ici l'importance de l'itération abstraite : c'est le seul moyen de parcourir les éléments d'une collection, en l'absence de numérotation des éléments. L'utilisation d'une collection n'est donc véritablement possible que si on passe par les `Iterator`s.

Remarque 1. Toutes les classes de la hiérarchie `Collection`, c'est-à-dire pour nous les classes `ArrayList`, `LinkedList` et `HashSet` (cf section 4), proposent par tradition un constructeur dont l'unique paramètre est une `Collection`. Ce constructeur fabrique un objet du type considéré (par exemple une `LinkedList`) contenant les mêmes objets que la collection paramètre. Ceci permet une forme de "conversion" d'une implantation des collections vers une autre.

3 Exemples d'utilisation

Comme `Collection` est une interface, il n'est pas possible de créer directement de `Collection`. De ce fait, les collections sont avant tout utilisées comme paramètres de méthodes. Dans `Java`, elles apparaissent par exemple comme paramètres des méthodes `max` et `min` de la classe `Collections`. Voici un exemple d'utilisation de l'interface `Collection` et des `Iterator`s pour afficher le contenu d'une collection et trouver son plus petit élément :

Exemple 1 :

```
import java.util.*;
public class TestCollection {
        public static void display(Collection c) {
                for(Iterator i=c.iterator();i.hasNext();)
                        System.out.print(i.next()+" ");
                System.out.println();
        }
        public static Object min(Collection c) {
        // on suppose que les objets sont comparables deux à deux
                Iterator i=c.iterator();
                Comparable currentMin=(Comparable)i.next();
                Comparable tmp;
                while(i.hasNext()) {
                        tmp=(Comparable)i.next();
                        if(currentMin.compareTo(tmp)>0)
                                currentMin=tmp;
                }
                return currentMin;
        }
        public static void main(String[] args) {
        // test avec une LinkedList (par exemple)
                Collection c=new LinkedList();
                c.add(new Double(-2.3));
                c.add(new Double(-3));
                c.add(new Double(5.3));
```

```
                c.add(new Double(0.2));
                c.add(new Double(-5.27));
                c.add(new Double(7));
                display(c);
                System.out.println(min(c));
        }
}
```

L'affichage produit par ce programme est bien entendu le suivant :

```
-2.3 -3.0 5.3 0.2 -5.27 7.0
-5.27
```

Remarque 2. On voit que l'écriture de `min` est un peu plus complexe que celle qu'on pourrait proposer pour une `List`. En effet, la méthode `next` des `Iterator`s "consomme" l'élément, ce qui n'est pas le cas d'un accès du genre `l.get(i)`. On doit donc conserver chaque élément lu dans une variable jusqu'à ce qu'il devienne inutile.

Pour la détermination du minimum et du maximum d'une collection, il est bien sûr inutile de programmer soit même des méthodes, puisqu'on peut utiliser celle de la classe `Collections`. On pourrait ainsi remplacer l'appel `min(c)` par `Collections.min(c)` (on peut aussi utiliser un `Comparator`).

4 Les ensembles (`Set`)

4.1 Principe

Comme nous l'avons dit dans la présentation des `Collection`s, un des modèles possible pour la notion de collection est celui d'ensemble fini au sens mathématique. Il s'agit ici d'une collection finie d'objets distincts deux à deux. Pour faire la distinction entre les collections au sens large et les ensembles, `Java` définit l'interface `Set` comme extension de l'interface `Collection`. Cependant, cette interface n'ajoute aucune méthode à l'interface `Collection`. Il s'agit simplement d'une interface de marquage : si une classe implante simplement l'interface `Collection`, on ne peut pas faire l'hypothèse que c'est la représentation d'un ensemble. Au contraire, si une classe préfère implanter l'interface `Set` (qui propose donc exactement les mêmes méthodes que `Collection`), on peut supposer qu'elle implante un ensemble. La seule véritable différence se situe donc au niveau du comportement : certaines collections peuvent contenir plusieurs fois le même élément, ce qui n'est pas possible pour les ensembles.

4.2 Implantation

La meilleure technique de programmation d'une classe représentant des ensembles est celle des tables de hachage (que nous avons évoquées dans la section 6, page 81). En `Java`[1], les ensembles sont programmés de cette manière par la classe `HashSet`. Cette classe possède exactement les méthodes de l'interface `Set` (i.e., celles de l'interface `Collection`) et ajoute simplement les constructeurs suivants :

`HashSet()`
: Construit l'ensemble vide.

`HashSet(Collection c)`
: Construit un ensemble contenant les éléments contenus dans la collection `c`.

[1] La méthode `hashCode` de la classe `Object` trouve ici son intérêt : elle permet de disposer d'une "traduction" de n'importe quel objet en `int`, préalable à la construction d'une table de hachage.

Il existe d'autres constructeurs permettant de régler les paramètres qui déterminent le fonctionnement fin de la table de hachage, mais ceci dépasse le cadre de ce cours.

L'utilisation des `HashSet`s est élémentaire : un `HashSet` est en effet une `Collection`. Le gros avantage est que les `HashSet`s sont très efficaces pour l'ajout, la recherche ou la suppression d'un élément (mais il est impossible de numéroter simplement les éléments d'un `HashSet`).

5 Comparaison des performances

5.1 Principe

Dans ce chapitre, nous avons vu trois programmations différentes de l'interface `Collection` : deux `List`s (`LinkedList` et `ArrayList`) et un `Set` (`HashSet`). On peut légitimement se demander si une programmation est plus efficace qu'une autre. Pour répondre à cette question, nous introduisons d'abord une classe `Chrono` rudimentaire, permettant de déterminer le temps d'exécution d'une méthode :

```
public class Chrono {
        private long start;
        public Chrono() {
        }
        public void start() {
                start=System.currentTimeMillis();
        }
        public long stop() {
                return System.currentTimeMillis()-start;
        }
}
```

La méthode `currentTimeMillis` de la classe `System` donne la date précise au millième de seconde, sous une forme qui dépend du système d'exploitation. On utilise cette classe pour chronométrer les méthodes de la classe `Benchmark` (dont le code est donné dans la section 6, page 91). Voici une description des méthodes de cette classe :

`create`
: Teste la vitesse de création d'une collection contenant `nb` éléments (le paramètre `base` est simplement utilisé pour pouvoir ajouter ensuite des éléments distincts en rappelant `create`). En fait, on teste plus précisément l'efficacité de l'ajout à la fin d'une collection (pour les `List`s).

`createStart`
: Même principe que la méthode précédente, sauf que l'ajout se fait au début (ne fonctionne donc que pour les `List`s). Cette méthode est en théorie défavorable aux `ArrayList`s : chaque ajout au début implique la recopie complète de tout le contenu du tableau.

`iterate`
: Réalise le parcours de tous les éléments d'une `List`. Défavorable aux `LinkedList`s pour lesquels l'accès à l'élément de place `i` est lent.

`iterate2`
: Même principe, mais avec l'utilisation d'un `Iterator`, indispensable pour les `HashSet`s et relativement efficace pour les `LinkedList`s.

`suppression`
: Suppression d'un élément choisi au hasard puis ajout de cet élément en dernière position. Teste l'accès direct à un élément par son numéro et les suppressions au milieu d'une liste.

`rotation`
: Suppression du premier élément d'une liste et ajout de celui-ci à la fin. Même principe que la méthode précédente, mais sans test de l'accès direct. Devrait être favorable aux `LinkedList`s.

`search`
: Recherche d'un élément donné. Devrait être favorable aux `HashSet`s.

`suppression2`
: Suppression d'un élément donné. Devrait être très défavorable aux `ArrayList`s (en raison des suppressions au milieu de la liste qui implique) et très favorable aux `HashSet`s.

`suppression3`
: Cette méthode supprime des éléments grâce à la méthode `remove` de l'interface `Iterator`. Ce genre de méthode devrait être favorable aux `LinkedList`s pour lesquels le parcours par `Iterator` est particulièrement adapté et qui sont efficaces pour l'ajout et suppression d'éléments.

5.2 Résultats

Le programme `Benchmark` a été exécuté sur un Pentium 90[2], sous Linux[3], grâce à la machine virtuelle `Java` de `Sun`, en version 1.2[4]. Toutes les optimisations (*threads* natifs et compilation au vol) ont été désactivées. Voici un tableau qui résume les résultats. Ce tableau indique le temps d'exécution de chaque méthode en seconde (les millièmes affichés par le programme sont arrondis au centième le plus proche). *NA* signifie non applicable.

méthode	`ArrayList`	`LinkedList`	`HashSet`
`create`	0.22	0.31	0.75
`iterate`	0.07	23.30	NA
`iterate2`	0.13	0.13	0.15
`suppression`	0.75	2.14	NA
`rotation`	1.36	0.03	NA
`search`	35.92	40.14	0.08
`suppression2`	119.71	34.45	0.09
`createStart`	4.12	0.07	NA
`suppression3` (1 sur 100)	0.26	0.15	0.16
`suppression3` (25 sur 100)	1.97	0.19	0.19

L'utilisation des optimisations (en particulier la compilation au vol) est assez délicate car elle modifie les performances d'une façon très complexe qui dépend énormément de l'ordre dans lequel les méthodes sont exécutées, etc. Cependant, on constate une nette amélioration des performances (temps d'exécution divisé par 2 ou 3 selon la méthode). Le plus important est surtout que les différences entre les trois classes ne sont pas modifiées par l'utilisation des optimisations de la machine virtuelle. Certains résultats catastrophiques (comme par exemple le temps d'exécution de `iterate` pour les `LinkedList`s) sont très largement améliorés, sans pour autant qu'une classe devienne meilleure qu'une autre alors qu'elle était moins bonne sans optimisation. J'ai préféré ne

[2] si, si, ça existe encore !
[3] cf `http://www.linux.com/`
[4] cf `http://www.blackdown.org/`

pas inclure les résultats numériques obtenus avec optimisation car ils sont moins significatifs que les résultats bruts.

On peut tirer d'importants enseignements des résultats obtenus :

- l'ajout en fin d'un tableau dynamique est très efficace (méthode `create`) ;
- l'utilisation de l'accès numéroté dans les `LinkedLists` est une véritable catastrophe (méthode `iterate`) : le parcours complet d'une liste chaîné de 10000 éléments prend 300 fois plus de temps que celui d'un tableau dynamique de même taille !
- l'ajout au début d'un tableau dynamique est assez inefficace (surtout sur un gros tableau, comme l'illustre la méthode `createStart`) ;
- la recherche d'un élément particulier est catastrophiquement lente pour les `Lists`. Une `HashSet` est 500 fois plus rapide sur un collection à `10000` éléments (méthode `search`).

5.3 Et la mémoire ?

Les résultats du chronométrage sont particulièrement favorables aux tables de hachage, qui apparaissent comme une structure de données très efficace. Ce n'est bien sûr pas une illusion, mais il faut tempérer l'efficacité en **temps d'exécution** au regard des résultats en **occupation mémoire**. Des constructions `Java` particulières (voir la section 6) permettent de mesurer approximativement[5] la mémoire occupée par chaque représentation d'une collection. Pour le test effectué, on fabrique une liste de 10 000 entiers (représentés par des `Integers`). Voici la mémoire occupée (en octets) par chaque représentation :

Classe	mémoire
`ArrayList`	208 352
`LinkedList`	400 928
`HashSet`	585 248

On voit que la table de hachage occupe énormément de mémoire (presque trois fois plus que le `ArrayList`), alors que la liste chaînée a une occupation mémoire intermédiaire. Pour bien comprendre l'occupation mémoire, j'ai écrit le programme suivant, qui permet de mesurer l'occupation mémoire d'un simple tableau :

```
public class BenchmarkMemory {
        public static void main(String[] args) {
                Runtime info=Runtime.getRuntime();
                long baseMemory,tmp;
                info.gc();
                baseMemory=info.freeMemory();
                int[] t=new int[10000];
                info.gc();
                tmp=info.freeMemory();
                System.out.println("Mémoire occupée par le int[] : "+
                        (baseMemory-tmp));
                baseMemory=tmp;
                Integer[] t2=new Integer[10000];
                info.gc();
                tmp=info.freeMemory();
                System.out.println("Mémoire occupée par le Integer[] vide : "
                        +(baseMemory-tmp));
```

[5] L'occupation mémoire exacte dépend de la machine virtuelle utilisée.

```
                for(int i=0;i<t2.length;i++)
                        t2[i]=new Integer(i);
                info.gc();
                tmp=info.freeMemory();
                System.out.println("Mémoire occupée par le "
                        +"Integer[] et son contenu : "
                        +(baseMemory-tmp));
        }
}
```

L'affichage produit par ce programme est le suivant (pour l'explication concernant la mesure de la mémoire, voir la section 6) :

```
Mémoire occupée par le int[] : 40016
Mémoire occupée par le Integer[] vide : 40440
Mémoire occupée par le Integer[] et son contenu : 200304
```

On voit tout d'abord que le tableau de `int` occupe la mémoire attendue. En effet, chaque entier occupe 4 octets, d'où 40 000 octets occupés par les 10 000 entiers. De plus, le tableau contient un entier qui indique sa longueur, soit 4 octets de plus. Les 12 octets restants correspondent à la représentation de l'objet `int[]` dans le tas. Le tableau de `Integer[]` occupe un peu plus de place en mémoire car c'est un tableau d'objets, et il n'a donc pas exactement la même nature. L'occupation approximative reste la même car chaque case du `Integer[]` contient une *référence* (et pas un objet). Or, une référence occupe 4 octets, comme un `int`.

L'occupation mémoire globale du `Integer[]` est importante. En fait, 159 864 octets proviennent des 10 000 objets de type `Integer`. Ceci donne une occupation d'un nombre d'octets moyen non entier, ce qui n'est pas très normal. En fait, les méthodes d'étude de la mémoire de `Java` ne sont pas parfaitement précises. On peut donc dire que chaque `Integer` occupe en gros 16 octets, ce qui correspond à 4 octets pour le `int` représenté et `12` octets de représentation de l'objet dans le tas (comme pour le `int[]`).

On voit que l'occupation mémoire d'un `ArrayList` est très raisonnable : elle correspond presque à un `Integer[]`. Par contre, les classes `LinkedList` et `HashSet` ne sont pas économes en mémoire. La `LinkedList` occupe deux fois plus de mémoire qu'un `Integer[]` de même taille, alors que le rapport est de trois pour le `HashSet`.

L'occupation mémoire d'une `LinkedList` n'est pas facile à réduire. En effet, c'est la structure de données qui est intrinsèquement coûteuse. Pour stocker un objet dans un tableau, il faut seulement conserver une référence vers cet objet, ce qui consomme 4 octets. Comme le `ArrayList` est basé sur un tableau, on peut dire qu'il consomme en gros 4 octets par objet stocké. Pour une `LinkedList`, on doit placer la référence vers l'objet à stocker dans un autre objet, qui contient des références vers le précédent et le suivant dans la liste. Comme un objet occupe au minimum 12 octets en `Java`[6], une `LinkedList` consomme au moins 24 octets (soit l'objet et 3 références) par objet stocké, ce qui est énorme.

Pour une table de hachage, la situation est très complexe et dépend en particulier des objets stockés. On peut seulement dire qu'un `HashSet` consommera à coup sûr au moins autant de mémoire qu'un `ArrayList`, en général plus, et parfois beaucoup plus.

Remarque 3. A l'heure actuelle, les ordinateurs personnels possèdent entre 128 et 1024 Mo de mémoire (soit entre 128 et 1024 millions d'octets). De ce fait, un programme `Java` peut normalement consommer sans problème une vingtaine de Mo. Il faut donc confronter l'occupation mémoire des structures étudiées à ce genre de chiffre. On voit donc qu'on dispose quand même d'une certaine latitude.

[6] avec la *machine virtuelle* de `Sun`.

5.4 Conclusions

Voici donc des heuristiques d'utilisation des trois classes :

`ArrayList`

Parfaitement adaptée pour l'accès numéroté à son contenu. Très efficace pour l'ajout des éléments en fin de liste. N'est pas adaptée à des modifications au début de la liste ou provoquées par un `Iterator`. Occupation mémoire minimale.

`LinkedList`

Adaptée pour les modifications provoquées par un `Iterator` (par exemple la suppression des éléments qui vérifient une certaine propriété). Efficace quand on veut numéroter les éléments et en ajouter (ou supprimer) en début de liste. Occupation mémoire importante.

`HashSet`

Très efficace pour l'accès par valeur. Construction relativement lente, mais modifications rapides. Occupation mémoire importante (parfois très importante).

Dans la pratique, voici quelques exemples concrets d'utilisation :

- **Représentation d'un ensemble** : on utilise alors `HashSet`. En pratique, ceci est particulièrement utile quand on doit par exemple conserver une occurrence unique de chacun des éléments d'une liste. On peut aussi utiliser un ensemble pour déterminer combien d'éléments distincts apparaissent dans une collection (voir l'exemple 2), etc.
- **Représentation d'une file d'attente** : on utilise alors `LinkedList`. En effet, dans une file d'attente, le premier arrivé est le premier servi : les nouveaux éléments sont ajoutés en fin de liste, alors que les éléments traités sont supprimés en début de liste.
- **Représentation d'une liste** : on utilise en général `ArrayList`. Quand on veut faire une utilisation basique des listes, avec numérotation, le plus simple est d'utiliser les `ArrayList`s.

Exemple 2 :

Voici un exemple d'utilisation d'une table de hachage pour compter le nombre d'éléments distincts contenus dans un `ArrayList` :

```
import java.util.*;
public class BenchmarkUnique {
        public static void display(long time,Collection c) {
                System.out.println(c.getClass().getName()+" "+time/1000d);
        }
        public static int compteUnique(Chrono chrono,List c) {
                chrono.start();
                int nb=0;
                for(int i=0;i<c.size();i++) {
                        nb++;
                        Object o=c.get(i);
                        for(int j=0;j<i;j++)
                                if(o.equals(c.get(j))) {
                                        nb--;
                                        break;
                                }
                }
                display(chrono.stop(),c);
                return nb;
        }
```

```
        public static void main(String[] arg) {
                Runtime info=Runtime.getRuntime();
                long baseMemory,tmp;
                Chrono c=new Chrono();
                info.gc();
                baseMemory=info.freeMemory();
                ArrayList al=new ArrayList();
                for(int i=0;i<10000;i++)
                        al.add(new Integer(i%1000));
                int unique1=compteUnique(c,al);
                System.out.println(unique1);
                info.gc();
                tmp=info.freeMemory();
                System.out.println("Mémoire occupée par le ArrayList : "
                        +(baseMemory-tmp));
                baseMemory=tmp;
                c.start();
                HashSet hs=new HashSet(al);
                int unique2=hs.size();
                display(c.stop(),hs);
                System.out.println(unique2);
                info.gc();
                tmp=info.freeMemory();
                System.out.println("Mémoire occupée par le HashSet : "
                        +(baseMemory-tmp));
        }
}
```

Le programme affiche les temps d'exécution[7] en secondes, ainsi que la mémoire occupée en octets :

```
java.util.ArrayList 24.959
1000
Mémoire occupée par le ArrayList : 208440
java.util.HashSet 0.176
1000
Mémoire occupée par le HashSet : 113448
```

La méthode par table de hachage est donc ici 140 fois plus rapide que la méthode directe. Bien entendu, tout ceci dépend de la taille du tableau de départ et du nombre d'éléments uniques, mais l'exemple reste éloquent.

L'occupation mémoire est elle aussi éloquente ! En effet, le `HashSet` ne contient ici que 1000 objets. De plus, la mémoire occupée par ces objets est déjà prise en compte dans la mémoire occupée par le `ArrayList`. De ce fait, le `HashSet` utilise ici plus de 100 octets par objet stocké, ce qui est vraiment très important (pour mémoire, le `ArrayList` consomme 4 octets par objet).

[7] Dans ce test, j'ai activé les optimisations, pour illustrer un cas réel d'utilisation.

6 Code de comparaison des performances

6.1 Programmation basique de `Benchmark`

Voici une programmation simple mais peu élégante de la classe `Benchmark`. Pour l'évaluation de la mémoire occupée, ce programme utilise la méthode de classe `getRuntime` de la classe `Runtime`. Cette méthode renvoie un objet de type `Runtime` qui décrit l'environnement d'exécution du programme `Java`. La méthode d'instance `gc` demande à `Java` de mettre en marche son *garbage collector* (c'est-à-dire son éboueur), qui se charge de supprimer les objets inutiles. La méthode d'instance `freeMemory` renvoie la mémoire libre, ce qui permet par simple soustraction de déduire la mémoire occupée par une structure de données. Voici le code de la classe `Benchmark` :

```
import java.util.*;
public class Benchmark {
        public static void display(long time,Collection c) {
                System.out.println(c.getClass().getName()+" "+time/1000d);
        }
        public static void create(Chrono chrono,Collection c,int nb,int base) {
                chrono.start();
                for(int i=0;i<nb;i++)
                        c.add(new Integer(i+base));
                display(chrono.stop(),c);
        }
        public static void createStart(Chrono chrono,List c,int nb,int base) {
                chrono.start();
                for(int i=0;i<nb;i++)
                        c.add(0,new Integer(i+base));
                display(chrono.stop(),c);
        }
        public static void iterate(Chrono chrono,List l) {
                chrono.start();
                Object o;
                for(int i=0;i<l.size();i++)
                        o=l.get(i);
                display(chrono.stop(),l);
        }
        public static void iterate2(Chrono chrono,Collection c) {
                chrono.start();
                Object o;
                for(Iterator i=c.iterator();i.hasNext();)
                        o=i.next();
                display(chrono.stop(),c);
        }
        public static void suppression(Chrono chrono,List l,int nb) {
                chrono.start();
                for(int i=0;i<nb;i++) {
                        int k=(int)(Math.random()*l.size());
                        l.add(l.remove(k));
                }
                display(chrono.stop(),l);
```

```
}
public static void rotation(Chrono chrono,List l,int nb) {
        chrono.start();
        for(int i=0;i<nb;i++) {
                l.add(l.remove(0));
        }
        display(chrono.stop(),l);
}
public static void search(Chrono chrono,Collection c,int nb) {
        chrono.start();
        for(int i=0;i<nb;i++) {
                int k=(int)(Math.random()*c.size());
                Integer tmp=new Integer(k);
                boolean b=c.contains(tmp);
        }
        display(chrono.stop(),c);
}
public static void suppression2(Chrono chrono,Collection c,int nb) {
        chrono.start();
        for(int i=0;i<nb;i++) {
                int k=(int)(Math.random()*c.size());
                Integer tmp=new Integer(k);
                c.remove(tmp);
        }
        display(chrono.stop(),c);
}
public static void suppression3(Chrono chrono,Collection c,int howMany) {
        chrono.start();
        Object o;
        int nb=0;
        for(Iterator i=c.iterator();i.hasNext();) {
                o=i.next();
                if (nb%100==0)
                for(int k=0;k<howMany;k++) {
                        i.remove();
                        if(i.hasNext())
                                i.next();
                        else
                                break;
                }
                nb++;
        }
        display(chrono.stop(),c);
}
public static void main(String[] arg) {
        Chrono c=new Chrono();
        // creation
        System.out.println("Création");
        Runtime info=Runtime.getRuntime();
```

```
info.gc();
long baseMemory=info.freeMemory();
ArrayList al=new ArrayList();
create(c,al,10000,0);
info.gc();
long tmp=info.freeMemory();
System.out.println("Mémoire occupée par le ArrayList : "
        +(baseMemory-tmp));
baseMemory=tmp;
LinkedList ll=new LinkedList();
create(c,ll,10000,0);
info.gc();
tmp=info.freeMemory();
System.out.println("Mémoire occupée par la LinkedList : "
        +(baseMemory-tmp));
baseMemory=tmp;
HashSet hs=new HashSet();
create(c,hs,10000,0);
info.gc();
tmp=info.freeMemory();
System.out.println("Mémoire occupée par le HashSet : "
        +(baseMemory-tmp));
baseMemory=tmp;
System.out.println("--------------------");
// itération
System.out.println("Itération");
iterate(c,al);
iterate(c,ll);
System.out.println("--------------------");
// itération par iterateur
System.out.println("Itération avec itérateur");
iterate2(c,al);
iterate2(c,ll);
iterate2(c,hs);
System.out.println("--------------------");
// suppression et ajout
System.out.println("Suppression et ajout");
suppression(c,al,1000);
suppression(c,ll,1000);
System.out.println("--------------------");
// rotation
System.out.println("Rotation");
rotation(c,al,1000);
rotation(c,ll,1000);
System.out.println("--------------------");
// accès par valeur
System.out.println("Recherche d'un élément");
search(c,al,1000);
search(c,ll,1000);
```

```
                search(c,hs,1000);
                System.out.println("--------------------");
                // suppression par valeur
                System.out.println("Suppression par valeur");
                suppression2(c,al,1000);
                suppression2(c,ll,1000);
                suppression2(c,hs,1000);
                System.out.println("--------------------");
                System.out.println("Ajout de nouveaux éléments,"
                        +" au début pour les listes");
                createStart(c,al,2500,10000);
                createStart(c,ll,2500,10000);
                create(c,hs,2500,1000);
                System.out.println("--------------------");
                // suppression par itérateur
                System.out.println("Suppression par itérateur "
                        +"(1 tout les 100)");
                suppression3(c,al,1);
                suppression3(c,ll,1);
                suppression3(c,hs,1);
                System.out.println("--------------------");
                System.out.println("Suppression par itérateur "
                        +"(25 tout les 100)");
                suppression3(c,al,25);
                suppression3(c,ll,25);
                suppression3(c,hs,25);
                System.out.println("--------------------");
        }
}
```

6.2 Programmation élégante

Pour donner un exemple d'application concrète de ce que nous avons étudié dans ce chapitre, nous pouvons proposer une version plus élégante de la classe `Benchmark`. Cette classe est particulièrement peu satisfaisante : chaque méthode répète le même début et la même fin qui n'a rien à voir avec elle-même (c'est le chronométrage). On souhaiterait pouvoir écrire une seule fois le code de chronométrage et pouvoir *insérer* entre le démarrage et l'arrêt du chrono l'appel à la méthode étudiée. Une solution simple consiste à utiliser la technique de représentation d'un algorithme par une classe. En effet, chaque méthode à chronométrer correspond à un algorithme simple, qui prend comme paramètre une `Collection` et un ou deux entiers. On peut donc utiliser l'interface suivante :

```
import java.util.Collection;
public interface MethodeCollection {
        public void algo(Collection c,int param1,int param2);
}
```

Voici maintenant quelques exemples d'implantation de l'interface :

```
import java.util.*;
public class CreateCollection implements MethodeCollection {
        public void algo(Collection c,int nb,int base) {
```

```
                for(int i=0;i<nb;i++)
                        c.add(new Integer(i+base));
        }
}

import java.util.*;
public class IterateList implements MethodeCollection {
        public void algo(Collection c,int notUsed1,int notUsed2) {
                if(c instanceof List) {
                        List l=(List)c;
                        Object o;
                        for(int i=0;i<l.size();i++)
                                o=l.get(i);
                }
        }
}
```

Pour pouvoir automatiser complètement le lancement des diverses méthodes, il faut pouvoir stocker une méthode à chronométrer et ses paramètres. Pour ce faire, on crée une classe simpliste :

```
public class MethodeAndParam {
        public MethodeCollection méthode;
        public int premier;
        public int second;
        public MethodeAndParam(MethodeCollection m,int param1,int param2) {
                méthode=m;
                premier=param1;
                second=param2;
        }
}
```

La classe `Benchmark` devient alors :

```
import java.util.*;
public class Benchmark2 {
        public static void runOne(MethodeAndParam m,Collection c) {
                Chrono chrono=new Chrono();
                chrono.start();
                m.méthode.algo(c,m.premier,m.second);
                long l=chrono.stop();
                System.out.println(c.getClass().getName()+" "+l/1000d);
        }
        public static void runMany(MethodeAndParam[] mt,Collection[] ct) {
                for(int j=0;j<mt.length;j++) {
                        System.out.println("Test de "+
                                mt[j].méthode.getClass().getName());
                        for(int i=0;i<ct.length;i++)
                                runOne(mt[j],ct[i]);
                        System.out.println("--------------------");
                }
        }
```

```
        public static void main(String[] args) {
                Collection[] ct={new ArrayList(),new LinkedList(),new HashSet()};
                MethodeAndParam[] mt={
                        new MethodeAndParam(new CreateCollection(),10000,0),
                        new MethodeAndParam(new IterateList(),0,0)};
                runMany(mt,ct);
        }
}
```

La solution retenue n'est pas parfaite, en particulier à cause des paramètres des méthodes qui ne sont pas les mêmes pour tous les algorithmes à chronométrer. Ceci étant, on obtient une solution beaucoup plus lisible que la première, et surtout dans laquelle on a évité au maximum de recopier plusieurs fois le même code.