Conditions de distribution et de copie

Exercices (IV) : Utilisation d'objets

Fabrice Rossi

15 février 2002

1 Problèmes "pratiques"

Les problèmes suivants peuvent être résolus sur ordinateur car ils se basent sur l'utilisation d'objets déjà définis en standard dans `Java`. On trouve la documentation correspondante sur le site web de `sun`, `http://java.sun.com`.

1.1 Chronométrage

Dans certains exercices et problèmes de cette section, on propose de comparer différentes méthodes par chronométrage. Pour ce faire, il faut utiliser la méthode codecurrentTimeMillis de la classe `System` (cette méthode renvoie un `long` qui correspond à une date exprimée en millisecondes).

On utilise cette méthode de la façon suivante :

ModeleChrono

```
1  public class ModeleChrono {
2    public static void main(String[] args) {
3      long début,fin;
4      début=System.currentTimeMillis();
5      // appel de la méthode à étudier
6      fin=System.currentTimeMillis();
7      // affichage du temps écoulé en millisecondes
8      System.out.println(fin-début);
9    }
10 }
```

Il suffit de remplacer le commentaire de la ligne 5 par un appel à la méthode à chronométrer, ou par les instructions qu'on souhaite étudier. Il est bien sûr possible de chronométrer plusieurs choses dans un même programme.

1.2 Chaînes de caractères

Problème 1.1 :

L'utilisation de l'objet `StringTokenizer` permet de faciliter le traitement des chaînes de caractères. Voici un démonstration de l'objet en question :

Token

```
1  import java.util.*;
2  public class Token {
3    public static void main(String[] args) {
4      String s="1.2   1.5  1.7";
```

```
5       StringTokenizer st=new StringTokenizer(s," ");
6       while(st.hasMoreTokens()) {
7         System.out.println(st.nextToken());
8       }
9       s="1.2, 1.5, 1.7";
10      st=new StringTokenizer(s," ,");
11      while(st.hasMoreTokens()) {
12        System.out.println(st.nextToken());
13      }
14    }
15  }
```

L'affichage produit par ce programme est le suivant :

```
1.2
1.5
1.7
1.2
1.5
1.7
```

Le principe de la classe simple :

- un objet `StringTokenizer` est capable de découper une chaîne de caractères en morceaux appelés *tokens*. Pour ce faire, il considère qu'une chaîne est constituée d'une suite de *tokens* séparés par des caractères spéciaux. La ligne 5 indique par exemple qu'on considère que la chaîne `s` utilise le caractère espace comme délimiteur. Pour la ligne 10, on utilise le caractère espace et la virgule comme délimiteurs ;
- la méthode `hasMoreTokens` renvoie `true` si et seulement si la chaîne à laquelle le `StringTokenizer` est associé contient encore des *tokens* non observés ;
- la méthode `nextToken` renvoie le prochain *token* de la chaîne à laquelle le `StringTokenizer` est associé. Après avoir été renvoyé, le *token* est considéré comme observé et le `StringTokenizer` passe au prochain *token*.

Quelques applications :

1. Ecrire un programme qui calcule la moyenne d'une suite de valeurs réelles saisies par l'utilisateur sous forme d'une chaîne dans laquelle les valeurs sont séparées par des virgules (et d'éventuels espaces), sur le modèle de la ligne 9 du programme d'exemple.
2. Ecrire un programme qui donne des statistiques sur les mots contenus dans une phrase saisie par l'utilisateur : mot le plus long, le plus court, nombre total de mots et taille moyenne des mots.
3. Ecrire une méthode qui recherche un mot dans un texte, au sens strict du terme : on considère par exemple que le mot "chercher" n'apparaît pas de le texte "rechercher un mot", car il n'y apparaît pas en tant que mot, mais en tant que partie d'un mot. La méthode devra renvoyer la position du mot dans le texte, en nombre de mots depuis le début du texte. Par exemple, le premier mot du texte porte le numéro 0, alors que le troisième porte le numéro 2. On pourra aussi chercher à renvoyer la position en nombre de caractères depuis le début du texte.

1.3 Fichiers

Problème 1.2 :

On se propose d'utiliser les classes de manipulation de fichiers afin de calculer des statistiques sur un texte.

Lecture d'un fichier :

Pour lire un fichier, on se base sur le modèle suivant :

Fichier

```
1  import java.io.*;
2  public class Fichier {
3    public static void main(String[] args) throws IOException {
4      BufferedReader lecteur=new BufferedReader(new FileReader("truc.txt"));
5      String s=lecteur.readLine();
6      while(s!=null) {
7        System.out.println(s);
8        s=lecteur.readLine();
9      }
10     lecteur.close();
11   }
12 }
```

Le principe du programme est simple :

- la ligne 1 correspond à l'utilisation du paquet `java.io` qui contient la majorité des outils de manipulation de fichiers ;
- l'ajout de la commande `throws IOException` à la fin de la déclaration de la méthode `main` (ligne 3) prévient le compilateur que nous allons utiliser des méthodes qui peuvent produire des erreurs d'exécution (ces erreurs sont liées à l'utilisation des méthodes de traitement des fichiers) ;
- pour lire le fichier de nom `truc.txt`, on commence par créer un objet `FileReader` (ligne 4), c'est-à-dire un "lecteur de fichier" ;
- pour faciliter la lecture du fichier, on fabrique un objet `BufferedReader` (ligne 4), `lecteur`, à partir de l'objet `FileReader` qu'on vient de créer : la référence `lecteur` permet donc de lire effectivement le fichier de nom `truc.txt` ;
- la lecture proprement dite est réalisée grâce à une seule méthode, `readLine`. Le principe de cette méthode est le suivant : l'ordinateur conserve une position dans le fichier (au départ, cette position est bien entendu le début du fichier). A chaque appel de `readLine`, l'ordinateur réalise les opérations suivantes :
  1. il lit une ligne dans le fichier, à partir de la position courante
  2. il se positionne juste après la ligne qu'il vient de lire
  3. le résultat de l'appel est une `String` qui contient la ligne que l'ordinateur vient de lire.

  Quand on arrive à la fin du fichier, la méthode `readLine` renvoie la référence `null`.
- l'appel `lecteur.close()` ferme le fichier après sa lecture.

La seule difficulté liée aux `BufferedReader` est qu'il est impossible de revenir en arrière dans un fichier. Si on appelle deux fois `readLine`, on lit deux lignes successives, pas deux fois la même ligne !

Si le fichier `truc.txt` a le contenu suivant :

truc.txt

```
1  Coucou
2  Et voilà
```

l'affichage produit par le programme d'exemple est le suivant :

```
Coucou
Et voilà
```

Dans la pratique, on peut éviter d'avoir à écrire le nom du fichier à étudier dans le programme. Pour ce faire, il suffit de passer par un composant graphique spécial, le `JFileChooser`, comme le montre l'exemple suivant :

FichierSelect

```
1  import java.io.*;
2  import javax.swing.*;
3  public class FichierSelect {
4    public static void main(String[] args) throws IOException {
5      JFileChooser chooser = new JFileChooser();
6      int choix;
7      do {
8        choix=chooser.showOpenDialog(null);
9      } while (choix!=JFileChooser.APPROVE_OPTION);
10     FileReader lecteurBase=new FileReader(chooser.getSelectedFile());
11     BufferedReader lecteur=new BufferedReader(lecteurBase);
12     String s=lecteur.readLine();
13     while(s!=null) {
14       System.out.println(s);
15       s=lecteur.readLine();
16     }
17     lecteur.close();
18   }
19 }
```

Le principe du programme est simple :
- la ligne 2 correspond à l'utilisation du paquet `javax.swing` qui contient de nombreux composants graphiques ;
- le composant permettant le choix d'un fichier est un `JFileChooser`, créé ligne 5 ;
- la méthode `showOpenDialog` a pour effet d'afficher une boîte de dialogue permettant le choix d'un fichier. Cette méthode renvoie un entier qui indique l'action choisie par l'utilisateur du programme. Nous attendons ici une action `APPROVE_OPTION` qui correspond au choix effectif d'un fichier ;
- enfin, on obtient le fichier sélectionné par l'appel de la méthode `getSelectedFile` qui renvoie un objet `File` (qu'on peut utiliser directement pour créer un `FileReader`).

On pourrait parfaitement utiliser le même objet `JFileChooser` pour choisir un fichier dans lequel écrire, auquel cas on passe par la méthode `showSaveDialog` (pour un exemple, voir la suite du texte).

Quelques traitements :

1. Ecrire un programme qui affiche le nombre de lignes et de lettres contenues dans un fichier.
2. Ecrire un programme qui demande un mot et affiche les lignes d'un fichier qui contiennent ce mot, en indiquant pour chaque ligne son numéro d'ordre dans le fichier (on pourra utiliser la méthode `indexOf` des `String`s).

3. Ecrire un programme qui affiche la moyenne, le minimum et le maximum des nombres réels contenus dans un fichier (on suppose que le fichier ne contient que des réels pour éviter des difficultés de conversion).
4. En utilisant éventuellement l'objet `StringTokenizer`, écrire un programme qui affiche le nombre de mots contenus dans un fichier.

Ecriture de fichier :

Ecrire dans un fichier ne pose pas vraiment de problème, comme le montre le programme suivant :

Save

```
1  import java.io.*;
2  import javax.swing.*;
3  public class Save {
4    public static void main(String[] args) throws IOException {
5      JFileChooser chooser = new JFileChooser();
6      int choix;
7      do {
8        choix=chooser.showSaveDialog(null);
9      } while (choix!=JFileChooser.APPROVE_OPTION);
10     FileWriter écrivainBase=new FileWriter(chooser.getSelectedFile());
11     PrintWriter écrivain=new PrintWriter(new BufferedWriter(écrivainBase));
12     écrivain.println("Coucou");
13     écrivain.close();
14   }
15 }
```

On remarque quelques différences minimes avec la lecture :
– on utilise la méthode `showSaveDialog` à la place de `showOpenDialog` (ce qui change les messages utlisés dans la boite de dialogue)
– l'objet d'écriture s'obtient à partir de trois objets : un `FileWriter`, puis un `BufferedWriter` et enfin un `PrintWriter`. Ce dernier objet propose toutes les méthodes `print` et `println` qu'on attend habituellement de `System.out`.

Quelques manipulations :

1. Ecrire un programme qui à partir de deux fichiers en écrit un troisième qui contient les lignes qui diffèrent d'un fichier à l'autre (plus précisément, on compare les lignes de même numéro dans les deux fichiers et on écrit dans le fichier résultat la ligne du premier fichier si elle est différente de la ligne de même numéro dans le deuxième fichier).
2. Ecrire un programme qui corrige la typographie d'un fichier, c'est-à-dire écrit un fichier corrigé à partir d'un fichier de départ en respectant les règles suivantes :
   (a) une phrase commence par une majuscule
   (b) les symboles `:`, `;`, `?` et `!` sont suivis et précédés par un espace
   (c) les symboles `.` et `,` sont suivi par un espace, mais ne sont jamais précédés par un espace.

   On aura intérêt à utiliser un `StringBuffer` pour accélérer les traitements.

3. En utilisant l'algorithme de cryptage proposé dans les exercices sur les chaînes de caractères, écrire un programme qui lit un fichier et écrit son contenu crypté dans un autre fichier (voir aussi le problème 1.4).

1.4 Entiers longs

Exercice 1.3 :

La classe `BigInteger` du paquet `java.math` permet de représenter des entiers de longueur quelconque, la principale application étant la cryptographie. Voici une classe de démonstration qui illustre les principales méthodes fournies par la classe `BigInteger` :

DemoBigInteger

```
1  import java.math.BigInteger;
2  public class DemoBigInteger {
3    public static void main(String[] args) {
4      // constructeur à partir de String
5      BigInteger x=new BigInteger("39388828829182");
6      // constructeur à partir de long
7      BigInteger y=BigInteger.valueOf(229299229);
8      // comparaison long/BI
9      x=BigInteger.valueOf(Long.MAX_VALUE);
10     long u=Long.MAX_VALUE;
11     System.out.println(u);
12     System.out.println(2*u);
13     System.out.println(x);
14     // calcul de 2*x en BigInteger (et donc résultat exact)
15     System.out.println(x.multiply(BigInteger.valueOf(2)));
16     // addition
17     System.out.println(x.add(y));
18     // soustraction
19     System.out.println(x.subtract(y));
20     // quotient de la division euclidienne
21     System.out.println(x.divide(y));
22     // reste de la division euclidienne
23     System.out.println(x.remainder(y));
24     // moins unaire
25     System.out.println(x.negate());
26     // puissance (entière)
27     System.out.println(x.pow(3));
28   }
29 }
```

L'affichage produit par le programme est le suivant :

```
9223372036854775807
-2
9223372036854775807
18446744073709551614
9223372037084075036
9223372036625476578
```

```
40224173788
104366355
-9223372036854775807
78463771692333509522426190271025445444293359109474248294 3
```

Questions :

1. Ecrire une méthode `BigInteger fact(long n)` qui calcule la factorielle d'un `long` de façon exacte en utilisant les `BigInteger`s.
2. Ecrire une méthode `BigInteger Cnp(long n,long p)` qui calcule le coefficient $C_n^p = \frac{n!}{p!(n-p)!}$ en utilisant les `BigInteger`s. On comparera les temps de calcul de l'approche simpliste qui se contente d'appliquer la formule à ceux d'approches plus évoluées qui travaillent par récurrence ou qui appliquent une formule simplifiée (cf le la section 1.1) pour le chronométrage.

Problème 1.4 :

On se propose de programmer l'algorithme de cryptage à clé publique RSA (d'après les initiales de ses créateurs, Rivest, Shamir et Adleman). Le principe des algorithmes à clé publique est celui de l'asymétrie : tout le monde connaît la clé publique alors que seul le destinataire du message connaît la clé privée. On utilise la clé publique pour coder le message. En théorie, seule la clé privée permet le décodage.

RSA est basé sur des calculs numériques simples effectués sur des grands entiers. C'est d'ailleurs l'une des principales applications des `BigInteger`s qui sont associés à de nombreuses méthodes liées à la cryptographie.

RSA comporte deux parties distinctes : le choix des clés et les calculs de codage et de décodage.

Calcul des clés

Pour déterminer un couple clé publique/clé privée, on commence par choisir au hasard deux nombres premiers très grands (100 chiffres semble un minimum), p et q (on utilise pour cela le constructeur adapté de `BigInteger`). On choisit ensuite au hasard un entier e premier avec $(p-1)(q-1)$ (en utilisant la méthode `gcd` des `BigInteger`s qui calcule le plus grand commun diviseur de deux nombres). On pose $n = pq$. Le couple (e, n) constitue la clé publique. La clé privée est obtenue en calculant d tel que $ed - 1$ soit divisible par $(p-1)(q-1)$ (en utilisant la méthode `modInverse` des `BigInteger`s). (d, n) est la clé privée associée à la clé publique (e, n).

La première partie du problème consiste à écrire un programme qui calcule un couple clé privée/clé publique en utilisant des nombres premiers de longueur choisie par l'utilisateur. Les clés seront sauvegardées dans deux fichiers distincts (deux lignes pour chaque clé).

RSA est entièrement basé sur le fait qu'il est très difficile (lire très long) de retrouver p et q à partir de n (et de e), et qu'il est donc très difficile de retrouver d à partir de la clé publique. On considère qu'avec des nombres premiers comprenant 1024 bits chacun (c'est-à-dire en gros 300 chiffres), le niveau de sécurité de RSA est militaire (en tout cas avec les moyens informatiques de 2002).

Codage et décodage

Le codage d'un texte est très simple, à condition de le considérer comme une suite de grands entiers. En effet, pour coder un entier m, il suffit de calculer le reste c de la division de m^e

par n. On peut montrer qu'on retrouve m en calculant le reste de la division de c^d par n (à condition que $m < n$). Ces opérations se font grâce à la méthode `modPow` des `BigInteger`s.

Pour coder un texte, il faut donc commencer par le convertir en une suite d'entiers comportant chacun autant de chiffres que n. Pour ce faire, on peut convertir chaque caractère du texte en son code unicode, soit un entier compris entre 0 et 65535. On complète ensuite les codes trop courts pour avoir un code à cinq chiffres pour chaque caractère (en fait, on peut se contenter de 4 chiffres pour les langues occidentales) et on place bout à bout les codes. Par exemple, le texte `Bonjour` devient l'entier suivant :

```
00660111011001060111011701l4
```

Dès qu'on obtient un entier presque aussi long que n, on réalise son codage par la formule indiquée, puis on passe à la suite du texte.

Le décodage se fait par l'opération inverse. Pour faciliter cette opération, il est important de retrouver les entiers qui ont été codés, c'est pourquoi on complète toujours les résultats par des zéros afin d'obtenir une suite d'entiers chacun de longueur exactement égale à celle de n.

La deuxième partie du problème consiste à écrire un programme qui propose codage et décodage d'un texte stocké dans un fichier, à partir d'un fichier contenant la clé à utiliser.

Programmation avancée

La programmation de RSA est basée sur plusieurs méthodes des `BigInteger`s dont les algorithmes sont intéressants et qu'on peut donc reprogrammer pour mieux comprendre la subtilité du cryptage :

Plus grand commun diviseur
: Pour déterminer e, on utilise la méthode `gcd` qui calcule le plus grand commun diviseur de deux `BigInteger`s. Programmer une méthode `pgcd` qui réalise le même calcul sans utiliser `gcd` et en s'appuyant sur l'algorithme d'Euclide (qui se base sur le fait que $pgcd(a, b) = pgcd(b\%a, a)$ où $b\%a$ désigne le reste de la division de b par a).

Reste d'une puissance
: Pour le codage et le décodage RSA, on calcule le reste de a^b divisé par n (en utilisant la méthode `modPow`). Il n'est pas très efficace de calculer a^b puis de faire la division. Il est déjà plus utile de faire la réduction (le calcul du reste) à chaque multiplication par a. Par exemple au lieu de calculer $a^8 \mod n$, on peut faire $(\ldots(((a \mod n) \times a \mod n) \times a \mod n)\ldots)$. Comparer, en les chronométrant, les deux approches.

Reste d'une puissance (version évoluée)
: On peut améliorer grandement la solution proposée précédemment en utilisant la décomposition binaire de b. Considérons par exemple $b = 13$. En binaire, b s'écrit 1101, ce qui veut dire que $13 = 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3$. On a donc $a^{13} = aa^{2^2}a^{2^3}$. Pour obtenir la décomposition en binaire, il suffit de diviser b par 2 successivement. Le reste de chaque division donne le chiffre correspondant de la représentation binaire. En parallèle, on multiple successivement par a une variable qui contient a^{2^k} (par élévation successive au carré). Voici un exemple d'évolution des variables dans un programme qui utilise l'algorithme proposé (à chaque étape, on multiplie le résultat par `a2k` si cela est nécessaire, en fonction du reste de la division de `bsur2k` par deux, on divise `bsur2k` par deux et on élève `a2k` au carré) :

résultat	bsur2k	a2k
1	13	a
a	6	a^2
a	3	a^4
a^5	1	a^8
a^{13}	0	a^{16}

Programmer cette approche et comparer la aux approches précédentes, toujours par chronométrage.

On pourra programmer le calcul de b tel que $ab = 1 \mod n$ en cherchant des références sur l'algorithme d'Euclide étendu (on remplace ainsi la méthode `modInverse`). De même on pourra aussi s'intéresser aux méthodes permettant d'engendrer aléatoirement des nombres premiers (on remplace ainsi le constructeur qui se charge de cette tâche).

1.5 Calcul numérique

Exercice 1.5 :

La classe `BigDecimal` du paquet `java.math` permet de représenter des nombres réels de précision arbitraire. Le principe de base simple : un `BigDecimal` est construit à partir d'un `BigInteger x` et d'une puissance `scale` (cf la section précédente pour les `BigIntegers`). Le réel représenté par ce couple est définit par $\texttt{x/10}^{\texttt{scale}}$. Voici un exemple de manipulation des `BigDecimals` :

DemoBigDecimal

```
1  import java.math.BigDecimal;
2  import java.math.BigInteger;
3  public class DemoBigDecimal {
4    public static void main(String[] args) {
5      BigDecimal x=new BigDecimal("12.3553");
6      BigDecimal y=new BigDecimal(BigInteger.valueOf(459782023),5);
7      System.out.println(x);
8      System.out.println(y);
9      BigDecimal z=x.multiply(y);
10     System.out.println(z);
11     z=z.multiply(z);
12     System.out.println(z.add(y));
13     System.out.println(z.subtract(x));
14   }
15 }
```

On obtient l'affichage suivant :

```
4597.82023
56807.448287719
3227090778.782098345006222961
3227086168.606568345006222961
```

La principale limitation des `BigDecimals` est que les calculs complexes ne sont pas proposés par la classe `BigDecimal`. Le seul calcul évolué est la division qui est réalisée par les méthodes `divide`. Le principe de `divide` est de calculer une approximation du résultat, en fonction d'un mode d'arrondi précisé lors de l'appel. Il y a deux méthodes : la première propose autant de

chiffres après la virgule dans le `BigDecimal` appelant. La seconde méthode permet de choisir le nombre de chiffres. Voici un exemple d'utilisation des deux méthodes :

DemoDivide

```
1  import java.math.BigDecimal;
2  import java.math.BigInteger;
3  public class DemoDivide {
4    public static void main(String[] args) {
5      BigDecimal x=new BigDecimal("3");
6      BigDecimal y=new BigDecimal("2");
7      System.out.println(y.divide(x,BigDecimal.ROUND_HALF_EVEN));
8      System.out.println(y.divide(x,30,BigDecimal.ROUND_HALF_EVEN));
9    }
10 }
```

On obtient l'affichage suivant :

```
1
0.666666666666666666666666666667
```

On voit qu'il est en général plus judicieux de choisir le nombre de chiffres après la virgule.

Questions :

– Les méthodes `divide` utilisent un algorithme exact mais relativement complexe pour effectuer leur calcul. On peut proposer une méthode relativement simple (et itérative) pour calculer $\frac{1}{x}$ qui peut servir de base à une division. On considère la suite définie par :

$$\begin{cases} u_0 \in]0, \frac{2}{x}[\\ u_n = 2u_{n-1} - x\,(u_{n-1})^2 \text{ si } n > 0 \end{cases}$$

On montre que cette suite converge vers $\frac{1}{x}$. Ecrire une méthode de classe qui à un `BigDecimal` x, une précision `nb` et un mode d'arrondi, associe $\frac{1}{x}$ calculé par la suite proposée. On utilisera les remarques suivantes :

- pour u_0, on peut chercher une puissance de dix qui majore x, par exemple en utilisant la méthode `bitLength` des `BigInteger`s qui donne le nombre de chiffres binaires utilisés dans un `BigInteger` (on appliquera cette méthode à la "partie entière" de x, obtenue grâce à la méthode `unscaledValue`) ;
- pour arrêter le calcul, on peut comparer u_n et u_{n-1} : quand la différence entre les deux valeurs devient petite par rapport à la précision souhaitée, on arrête le calcul.

On comparera la méthode proposée à la méthode `divide` correspondante (par chronométrage).

– la classe `BigDecimal` ne propose pas de méthode `sqrt` permettant de calculer la racine carrée d'un `BigDecimal`. Pour faire un tel calcul, la technique la plus simple consiste à passer par une méthode itérative (comme pour $\frac{1}{x}$). Pour ce faire, on considère la suite définie par :

$$\begin{cases} v_0 \in]0, \sqrt{\frac{3}{x}}[\\ v_n = \frac{3v_{n-1} - x(v_{n-1})^3}{2} \text{ si } n > 0 \end{cases}$$

On montre que cette suite converge vers $\frac{1}{\sqrt{x}}$.

Pour appliquer cette technique, on commence par écrire une méthode `div2` qui divise par 2 un `BigDecimal` sans utiliser la méthode `divide`, mais en se basant sur la décomposition

`n/10^scale` et sur le fait que dans celle-ci, le `BigInteger n` est représenté en base 2. On comparera la méthode proposée à une utilisation de `devide` et à une multiplication par le `BigDecimal` 0.5.
Ensuite, on écrit une méthode `sqrtInv` qui renvoie $\frac{1}{\sqrt{x}}$ à une précision donnée, en utilisant la suite proposée. Pour le point de départ de la suite, on pourra utiliser une approximation grossière de $\frac{1}{\sqrt{x}}$ basée sur une puissance de 10.
Enfin, pour calculer $\sqrt{x}$, on multiplie $\frac{1}{\sqrt{x}}$ par x.

– On peut utiliser une technique plus directe pour le calcul de $\sqrt{x}$ à partir de la suite définie par :

$$\begin{cases} w_0 > 0 \\ w_n = \frac{1}{2}\left(w_{n-1} + \frac{x}{w_{n-1}}\right) \text{ si } n > 0 \end{cases}$$

On montre que pour tout point de départ positif w_0, la suite converge vers $\sqrt{x}$. La principale différence avec la méthode précédente est qu'on doit faire une division quelconque (et pas seulement par 2), ce qui risque de ralentir les calculs.
Programmer une méthode `sqrt` qui calcule la racine carrée d'un `BigDecimal` à une précision donnée en utilisant la suite w_n. Comparer les deux solutions par chronométrage.

Problème 1.6 :

Pour calculer une valeur approchée de π, on utilise le développement en série entière de la fonction arctan. On a en effet :

$$\arctan(x) = \sum_{i=0}^{\infty} \frac{(-1)^k x^{2k+1}}{2k+1}$$

Si on sait calculer une valeur approchée de $\arctan(x)$, on peut alors obtenir π par différentes formules :

$$\pi = 4\arctan(1)$$
$$\pi = 16\arctan\left(\frac{1}{5}\right) - 4\arctan\left(\frac{1}{239}\right)$$
$$\pi = 48\arctan\left(\frac{1}{18}\right) + 32\arctan\left(\frac{1}{57}\right) - 20\arctan\left(\frac{1}{239}\right)$$

Tout l'intérêt de ces formules est qu'elles ne font intervenir que des entiers et les opérations de base (addition, soustraction, multiplication et division), à condition bien sûr de savoir calculer arctan.

Calcul simple : on commence par programmer un calcul simple de π, c'est-à-dire un calcul qui se base sur le type fondamental `double` :

1. Écrire d'abord une méthode de classe `double arctan(double x,double p)` qui calcule arctan de `x` en utilisant le développement en série entière, et avec une précision de `p`. On calcule en fait $\sum_{i=0}^{n} \frac{(-1)^k x^{2k+1}}{2k+1}$, n étant déterminé de sorte que $\frac{x^{2n+1}}{2n+1} <$ `p`. Il est interdit d'utiliser la méthode `Math.pow` (en fait, c'est surtout stupide).
2. Écrire une méthode principale qui affiche les temps de calcul des trois formules proposées. On devra permettre à l'utilisateur de choisir la précision `p`.

Calcul évolué : on souhaite maintenant donner un nombre arbitraire de décimales pour π, en utilisant la classe `BigDecimal`. Pour effectuer le calcul de π, on travaille exactement comme dans la section précédente, en commençant par écrire une méthode `BigDecimal arctan(int k,int p)`. Cette méthode doit calculer une valeur approchée de arctan $\left(\frac{1}{k}\right)$, avec `p` chiffres corrects après la virgule. On utilisera ensuite la méthode pour calculer π, mais en se limitant aux deux dernières formules proposées. Comme pour la section précédente, on comparera les deux formules grâce au chronométrage des calculs. Il est important d'afficher les valeurs obtenues pour π afin de vérifier que les premiers chiffres sont corrects.

2 Problèmes "théoriques"

Les problèmes suivants sont à résoudre de façon théorique (sur papier), car `Java` ne propose pas en standard les objets étudiés. On peut bien entendu programmer de tels objets, mais c'est en général un problème à part entière.

Problème 2.1 :

On considère donnée une classe `Tableau` qui permet la création d'objets. Un objet de type `Tableau` représente une zone rectangulaire du plan dans laquelle on peut placer des objets mathématiques. Voici les méthodes d'instance de la classe `Tableau` :

constructeur : `Tableau(double xmin,double xmax,double ymin,double ymax)`

Fabrique un tableau qui représente la zone d'abscisses minimale `xmin` et maximale `xmax`, et d'ordonnées minimale `ymin` et maximale `ymax`.

`void ajouteSegment(double dx,double dy,double ax,double ay)`

Ajoute au tableau appelant un segment de point de départ `(dx,dy)` et de point d'arrivée `(ax,ay)`.

`void ajouteCercle(double x,double y,double r)`

Ajoute au tableau appelant un cercle de centre `(x,y)` et de rayon `r`.

`void efface()`

Efface tous les objets mathématiques contenus dans le tabeau appelant.

Voici un exemple d'utilisation de l'objet `Tableau` :

DemoTableau

```
1  public class DemoTableau {
2    public static void main(String[] args) {
3      // on représente le rectangle [-1,1]x[-1,1]
4      Tableau t=new Tableau(-1,1,-1,1);
5      // on dessine un carré de côté 1 et centré en 0,0
6      t.ajouteSegment(0.5,0.5,0.5,-0.5);
7      t.ajouteSegment(0.5,-0.5,-0.5,-0.5);
8      t.ajouteSegment(-0.5,-0.5,-0.5,0.5);
9      t.ajouteSegment(-0.5,0.5,0.5,0.5);
10   }
11 }
```

La figure 1 illustre le résultat mathématique du programme d'exemple. **Dans tout cet exercice, vous pouvez vous inspirer de ce que vous avez étudié en graphisme. Notez bien que seuls les principes généraux restent valables, car le repère du `Tableau` est un repère mathématique classique et les coordonnées des points sont des `doubles`.**

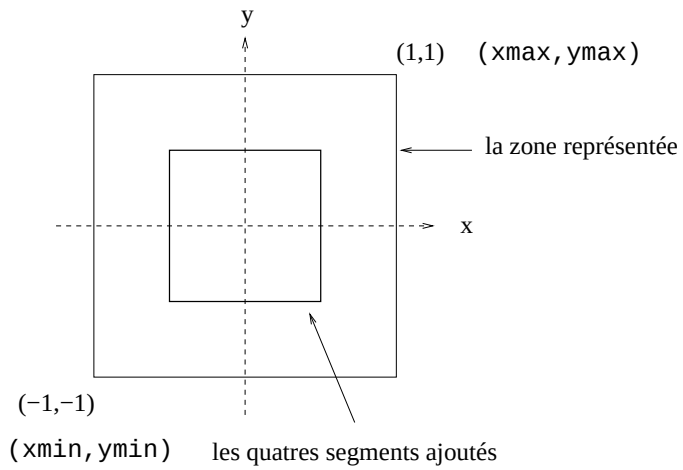

FIG. 1 – Résultat du programme `DemoTableau`

On peut parfaitement répondre aux questions sans aucune connaissance sur le graphisme en `Java`.

1. Ecrire une méthode **de classe** `void grille(Tableau t,int n)` qui efface le `Tableau` paramètre et dessine une grille de `n`×`n` cases. Pour ce faire, il suffit de tracer `n+1` segments horizontaux et autant de segments verticaux. Chaque case doit avoir un côté de longueur 1. La case la plus en bas à gauche a comme coin inférieur gauche l'origine du repère.
2. Ecrire une méthode **de classe** `void ajoutePolygone(Tableau t,double[] x,double[] y)` qui ajoute au `Tableau` paramètre le polygone décrit par les tableaux `x` et `y`. Le tableau `x` donne les abscisses des sommets du polygone, alors que `y` donne les ordonnées. Notez bien qu'il faut fermer le tracé du polygone.

   Si on appelle la méthode proposée de la façon suivante : `ajoutePolygone(t, new double[]{0.5,0.5,-0.5,-0.5}, new double[]{0.5,-0.5,-0.5,0.5})`, on doit obtenir le carré proposé dans l'exemple `DemoTableau`.
3. Ecrire une méthode **de classe** `void tableauxCercle(Tableau t,double[][] x)` qui donne du tableau `x` la représentation suivante. On détermine d'abord `max` la plus grande valeur du tableau. Ensuite, le tableau est représenté par autant de cercles qu'il contient de cases. La case `x[i][j]` est représentée par un cercle de centre `(i,j)` et de rayon `0` si `x[i][j]<=0` et de rayon `x[i][j]/(2*max)` sinon.
4. Ecrire une méthode **de classe** `void trajetDécroissant(Tableau t,double[][] x)` qui ajoute `n-1` segments (où `n` désigne le nombre total de cases de `x`) au `Tableau t` selon le principe suivant. On définit les suites x_i et y_i de la façon suivant : la case `x[`x_0`][`y_0`]` contient le plus grand élément du tableau `x`, la case `x[`x_1`][`y_1`]` contient le deuxième plus grand élément du tableau, etc. jusqu'à la case `x[`$x_{\mathtt{n}-1}$`][`$y_{\mathtt{n}-1}$`]` qui contient le plus petit élément du tableau `x`. On ajoute au `Tableau t` les segments reliant (x_i, y_i) à (x_{i+1}, y_{i+1}) pour i allant de 0 à $\mathtt{n}-2$.

Problème 2.2 :

On suppose donnée une classe `Rationnel`. Un objet de type `Rationnel` représente un rationnel au sens mathématique du terme (c'est-à-dire une fraction $\frac{p}{q} \in \mathbb{Q}$. Voici la documentation de la classe :

constructeur : `Rationnel(int p,int q)`

Fabrique un objet `Rationnel` représentant $\frac{\texttt{p}}{\texttt{q}}$.

`int num()`

Renvoie la valeur du numérateur du rationnel appelant.

`int dén()`

Renvoie la valeur du dénominateur du rationnel appelant.

`double toDouble()`

Renvoie le `double` le plus proche du rationnel représenté par l'objet appelant.

`Rationnel somme(Rationnel r)`

Renvoie un nouvel objet `Rationnel` somme du rationnel appelant et du rationnel `r`.

`Rationnel produit(Rationnel r)`

Renvoie un nouvel objet `Rationnel` produit du rationnel appelant et du rationnel `r`.

`Rationnel différence(Rationnel r)`

Renvoie un nouvel objet `Rationnel` différence du rationnel appelant et du rationnel `r`.

`Rationnel quotient(Rationnel r)`

Renvoie un nouvel objet `Rationnel` quotient du rationnel appelant et du rationnel `r`.

`int compareTo(Rationnel r)`

Renvoie un entier strictement négatif si le rationnel appelant est strictement plus petit que le rationnel `r`. Renvoie zéro si les deux rationnels sont égaux, et un entier strictement positif si le rationnel appelant est strictement plus grand que le rationnel `r`.

`Rationnel abs()`

Renvoie le rationnel valeur absolue du rationnel appelant.

Questions :

1. Ecrire une méthode `public static Rationnel moyenne(int[] t)` qui au tableau d'entiers `t` associe sa moyenne, calculée de façon exacte grâce à la classe `Rationnel`.
2. Ecrire une méthode `public static Rationnel moyenne(Rationnel[] t)` qui au tableau de rationnels `t` associe sa moyenne, calculée de façon exacte.
3. On souhaite résoudre de façon exacte le système d'équations suivant :

$$\begin{cases} ax + by + c = 0 \\ dx + ey + f = 0 \end{cases}$$

On suppose que les coefficients a, b, c, d, e et f sont des entiers. On forme d'abord le déterminant du système $g = ae - bd$. Si $g \neq 0$, le système possède une seule solution donnée par :

$$x = \frac{bf - ce}{g}$$
$$y = \frac{cd - af}{g}$$

Ecrire une méthode `public static Rationnel[] système(int a,int b,int c,int d,int e,int f)` qui résout le sytème d'équations décrit par les paramètres entiers. Si ce système n'a pas une solution unique, la méthode devra renvoyer un tableau vide. Sinon, elle devra renvoyer sous forme d'un tableau x et y calculés exactement.

4. La principale limitation des rationnels est qu'ils ne peuvent pas représenter de façon exacte certains nombres réels. L'exemple le plus simple est $\sqrt{2}$ qui est irrationnel. Pour représenter de façon approchée une racine carrée, on utilise la propriété suivante. Etant donné un réel $x > 0$, la suite $(u_n)_{n\in\mathbb{N}}$ définie comme suit converge (très rapidement) vers $\sqrt{x}$:

$$\begin{cases} u_0 &= \frac{x}{2} \\ u_k &= \frac{1}{2}\left(u_{k-1} + \frac{x}{u_{k-1}}\right) \text{ pour } k > 0 \end{cases}$$

   Ecrire une méthode `public static Rationnel sqrt(Rationnel r,Rationnel p)` qui au `Rationnel r` associe une valeur approchée de $\sqrt{\texttt{r}}$, calculée en utilisant la suite définie plus haut. On ne peut pas savoir à l'avance combien de termes de la suite il faut calculer. On arrêtera donc le calcul quand $|u_{k-1} - u_k| < \texttt{p}$, en renvoyant le rationnel u_k. Le `Rationnel p` représente donc la précision du calcul.

5. En utilisant la méthode `sqrt` qui vient d'être définie, écrire une méthode `public static Rationnel std(Rationnel[] t,Rationnel p)` qui au tableau de `Rationnels t` associe son écart-type (`p` sera utilisé comme paramètre de précision). On rappelle que l'écart-type de la liste de valeurs $(t_0, \ldots, t_{n-1})$ est donné par :

$$\sqrt{\frac{1}{n}\sum_{i=0}^{n-1}(t_i - m)^2},$$

   où m désigne la moyenne de la liste.

6. Ecrire une méthode `public static Rationnel[] trinôme(Rationnel[] coeffs,Rationnel p)` qui calcule la ou les racine(s) réelle(s) du trinôme dont les coefficients sont rangés dans le tableau `coeffs` (`coeffs[`i`]` correspond au coefficient de x^i). S'il n'existe pas de racine réelle, le tableau renvoyé devra être vide. On considérera `p` comme un paramètre de précision.