IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

TP5 – Interface et itérateur

Informatique – MAPD

Correction

Objectifs

- Réaliser des implémentations d'une interface (`Iterator`)
- Spécialiser une classe (extension avec cache)

Cet exercice est un exemple d'utilisation d'`Iterator`.
Il permet de voir des obligations de programmation.
Une généralisation est envisageable quand les lambdas (réification de fonction) auront été vues.

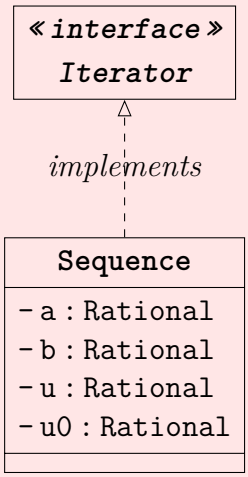
Suite de rationnels

Exercice 1 (*Suite*)

Le but de cet exercice est de réaliser la notion de suite mathématique. Nous l'appliquerons à des suites de rationnels de la forme $u_{i+1} = au_i + b$ avec $u_0 \in \mathbb{Q}$.

▷ **Question 1.1 :**
Écrire une classe représentant une suite (classe `Sequence`) qui implémente `java.util.Iterator`. Tester.

1. **Définie par 3 rationnels (a, b, u_0)**
2. **Ayant toujours un élément $i + 1$**
3. **Calculant l'élément suivant sur demande**



Voir le code dans la réponse à la question suivante.
Le point important est dans la définition de la classe :
`public class Sequence implements Iterator<Rational>`
Les obligations de programmation en découlent...

▷ **Question 1.2 :**
Compléter la classe avec une méthode permettant d'accéder à n'importe quel u_i. La valeur du i courant ne change pas.

```
package sequence;

import java.util.Iterator;

/**
 * The Class Sequence.
 *
 * @author beugnard
 *
 *          I implement a sequence u0, u1, ..., un ,... of Rational.
 *
 *          The definition is fixed and is u(i+1) = a*u(i)+b A more general
 *          definition will be possible once the lamba has been seen.
 */
public class Sequence implements Iterator<Rational> {

      /** The u 0. */
      private Rational a, b, u, u0;

      /**
       * Instantiates a new sequence.
       *
       * @param r1 the r 1
       * @param r2 the r 2
       * @param r3 the r 3
       */
      public Sequence(Rational r1, Rational r2, Rational r3) {
```

```
        a = r1; //
        b = r2; //
        u = r3; // current value
        u0 = r3; // remind initial value
    }

    /**
     * Checks for next.
     * it is infinite
     * @return true, if successful
     */
    @Override
    public boolean hasNext() {
        return true;
    }

    /**
     * Next : with the normal ordering
     *
     * @return the rational
     */
    @Override
    public Rational next() {
        return compute(u);
    }

    /**
     * Compute.
     *
     * @param p the p
     * @return the rational
     */
    private Rational compute(Rational p) {
        Rational up1 = b.sum(a.times(p));
        u = up1;
        return up1;
    }

    /**
     * Index: with a random ordering
     *
     * @param ind the ind
     * @return the rational
     */
    // a complementary function...
    public Rational index(int ind) {
        Rational ui, res;
        ui = u; // stock état courrant
        u = u0;
        for (int i = 0; i < ind; i++) {
            this.next();
```

```
            }
            res = u; // à retourner
            u = ui; // remise état courrant
            return res;
      }

      /**
       * The main method.
       *
       * @param a the arguments
       */
      public static void main(String[] a) {
            Rational r1, r2, init;
            r1 = new Rational(3, 2);
            r2 = new Rational(1, 2);
            init = new Rational(0, 1);
            Sequence s = new Sequence(r1, r2, init);
            System.out.println("0 : " + init);
            for (int i = 1; i < 15; i++) {
                  System.out.println(i + " : " + s.next());
            }
            System.out.println("Direct access");
            System.out.println(s.index(3)); // 19/8 expected
            System.out.println(s.index(18)); // 387158345/262144 expected
            System.out.println(15 + " : " + s.next()); // 15 ?
      }
}
```

La solution précédente n'est sans doute pas efficace, puisqu'il faut recalculer des termes déjà calculés.

▷ **Question 1.3 :**
Proposer une solution qui stocke les résultats déjà calculés. Cette technique s'apparente à la notion de *cache* ; pour des raisons de performance, on stocke des résultats de calcul car, le coût pour retrouver un calcul est moindre que celui de le refaire. Il faut cependant faire attention au coût de stockage. . .

Pas intensivement testé. . .mais c'est l'idée.

```
package sequence;

import java.util.LinkedList;

/**
 * The Class CachedSequence.
 */
public class CachedSequence extends Sequence {

    /** The cache. */
    private LinkedList<Rational> cache;
```

```java
/**
 * Instantiates a new cached sequence.
 *
 * @param r1 the r 1
 * @param r2 the r 2
 * @param r3 the r 3
 */
public CachedSequence(Rational r1, Rational r2, Rational r3) {
  super(r1, r2, r3);
  cache = new LinkedList<Rational>();
  cache.add(r3);
}

/**
 * Next.
 *
 * @return the rational
 */
@Override
public Rational next() {
  Rational u = super.next();
  cache.add(u);
  return u;
}

/**
 * Index.
 *
 * @param ind the ind
 * @return the rational
 */
public Rational index(int ind) {
  Rational res;
  try {
      res = cache.get(ind);
  } catch (Exception e) {// not yet computed
      for (int i = cache.size(); i < ind + 1; i++) {
        this.next();
      }
      res = cache.get(ind);
  }
  return res;
}

/**
 * The main method.
 *
 * @param a the arguments
 */
public static void main(String[] a) {
  Rational r1, r2, init;
```

```
      r1 = new Rational(3, 2);
      r2 = new Rational(1, 2);
      init = new Rational(0, 1);
      CachedSequence s = new CachedSequence(r1, r2, init);
      System.out.println("0 : " + init);
      for (int i = 1; i < 15; i++) {
          System.out.println(i + " : " + s.next());
      }
      System.out.println("Direct access");
      System.out.println(s.index(3)); // 19/8 expected
      System.out.println(s.index(18)); // 387158345/262144 expected
      System.out.println(19 + " : " + s.next()); // 19!
    }
}
```

Une variante avec une méthode normale (sans utiliser les exceptions).

```
package sequence;

import java.util.LinkedList;

/**
 * The Class VarianteCachedSequence.
 */
public class VarianteCachedSequence extends Sequence {

    /** The cache. */
    private LinkedList<Rational> cache;

    /**
     * Instantiates a new variante cached sequence.
     *
     * @param r1 the r 1
     * @param r2 the r 2
     * @param r3 the r 3
     */
    public VarianteCachedSequence(Rational r1, Rational r2, Rational r3) {
      super(r1, r2, r3);
      cache = new LinkedList<Rational>();
      cache.add(r3);
    }

    /**
     * Next.
     *
     * @return the rational
     */
    @Override
    public Rational next() {
      Rational u = super.next();
      cache.add(u);
      return u;
```

```
    }

    /**
     *
     * Ici on utilise du code normal sans exception.
     *
     * @param ind the ind
     * @return the rational
     */
    public Rational index(int ind) {
      Rational res;
      if (ind >= cache.size()) {
          // not yet computed
          System.out.println("I need to compute a bit");
          for (int i = cache.size(); i < ind + 1; i++) {
            this.next();
          }
          res = cache.get(ind);
      } else {
          res = cache.get(ind);
      }
      return res;
    }

    /**
     * The main method.
     *
     * @param a the arguments
     */
    public static void main(String[] a) {
      Rational r1, r2, init;
      r1 = new Rational(3, 2);
      r2 = new Rational(1, 2);
      init = new Rational(0, 1);
      VarianteCachedSequence s = new VarianteCachedSequence(r1, r2, init);
      System.out.println("0 : " + init);
      for (int i = 1; i < 15; i++) {
          System.out.println(i + " : " + s.next());
      }
      System.out.println("Direct access");
      System.out.println(s.index(3)); // 19/8 expected
      System.out.println(s.index(18)); // 387158345/262144 expected
      System.out.println(19 + " : " + s.next()); // 19!
    }
}
```

▷ **Question 1.4 :**
Les `int` sont bornés. Comment prendre en compte cette contrainte ?

Ne pas retourner **`true`** tout le temps mais **`false`** au-delà de `Integer.MAX_VALUE`, mais il faut

gérer un index entier et l'incrémenter avec `next`.

Série de rationnels (optionnel)

Exercice 2 (*Série*)

Le but de cet exercice est de réaliser la notion de série mathématique. Les éléments de la série sont : $s_n = \sum_{i=0}^{n} u_i$ avec $u_{i+1} = au_i + b$ et $u_0 \in \mathbb{Q}$.

▷ **Question 2.1 :**
Réaliser la classe `Serie`. Tester.

Pas intensivement testé...mais c'est l'idée.

```
package mapd;

import java.util.Iterator;

/**
 *
 * @author beugnard
 *
 * I implement a serie u0, u1, ..., un ,... over a sequence of Rational.
 *
 * The definition is fixed and is $U_n = \sum_0^n{u_i}$
 */
public class Serie implements Iterator<Rational> {

    private Sequence s;
    private Rational sum;

    public Serie (Rational r1, Rational r2, Rational r3) {
      s    = new Sequence(r1, r2, r3);
      sum = new Rational (0,1);
    }

    @Override
    public boolean hasNext() {
      return true;
    }

    @Override
    public Rational next() {
      return compute();
    }

    private Rational compute() {
      sum = sum.sum(s.next());
      return sum ;
    }
```

```
    public static void main (String [] a) {
      Rational r1, r2, init;
      r1 = new Rational(3,2);
      r2 = new Rational (1,2);
      init = new Rational (0,1);
      Serie s = new Serie (r1, r2, init);
      System.out.println(init);
      for (int i=1; i< 10; i++) {
          System.out.println(s.next());
      }
    }
}
```

▷ **Question 2.2 :**
Que pensez-vous de la classe `java.util.Random` ? `https://docs.oracle.com/javase/8/docs/api/java/util/Random.html`

Elle aurait pu implanter l'interface `java.util.Iterator`.
Des raisons pour ne pas le faire ?
Exercice potentiel : `https://codereview.stackexchange.com/questions/162035/iterator-producing-unique-random-numbers-in-a-specified-range`

▷ **Question 2.3 :**
Comment pourrait-on généraliser à d'autres formes de suites et de séries ?

Ce serait bien si on pouvait passer une fonction au constructeur.
Sinon, faire une classe par fonction...