# FIL1 – Modularité et typage – 2013/2014
# Évaluation

2 décembre 2013

Pour répondre, compléter le fichier associé `Test.java`, à récupérer sur Campus. Finalement, le déposer sur Campus.

Remarques

– Le langage de programmation utilisé est Java, version 5 au moins.
– On utilise la bibliothèque `java.util` de l'API (*Application Programming Interface*) de Java. Voir l'annexe à la fin pour les détails utiles.
– Dans la suite, l'expression `@@@` représente du code supposé connu.

On se donne neuf interfaces génériques correspondant à des structures algébriques classiques : semi-groupes, monoïdes, groupes, additifs et multiplicatifs, anneaux et corps.

```java
interface SemiGroupeAdd<T> {
    T somme(T x);
}
interface MonoideAdd<T> extends SemiGroupeAdd<T> {
    T zero();
}
interface GroupeAdd<T> extends MonoideAdd<T> {
    T oppose();
}
interface SemiGroupeMul<T> {
    T produit(T x);
}
interface MonoideMul<T> extends SemiGroupeMul<T> {
    T un();
}
interface GroupeMul<T> extends MonoideMul<T> {
    T inverse();
}
interface Anneau<T> extends GroupeAdd<T>, SemiGroupeMul<T> {}
interface AnneauUnitaire<T> extends Anneau<T>, MonoideMul<T> {}
interface Corps<T> extends AnneauUnitaire<T>, GroupeMul<T> {}
```

Tous les tests seront réalisés dans la fonction principale de la classe `Test`.

### Quelques calculs génériques

**Exercice 1 (Un module de calculs)** Dans un module de calculs, appelé `Calculs`, on développe des fonctions génériques de calculs opérant sur les interfaces précédentes.

**1.1** Définir une fonction `T soustraction(T x, T y)` calculant la soustraction de `y` à `x` (autrement dit, `x` moins `y`). On veillera à donner au paramètre `T` le majorant le plus grand possible (relativement à la relation de sous-typage), afin de rendre la fonction `soustraction` la plus générale possible.

```
class Calculs {
  @@@
  public static <T extends GroupeAdd<T>>
    T soustraction(T x, T y){
      return x.somme(y.oppose());
  }
  @@@
}
```

**1.2** Généraliser l'opération `somme` en définissant une fonction générique `T sommeNAire(List<T> l)`, spécifiée ainsi :

- si la liste `l` est vide, la fonction déclenche une erreur `IllegalArgumentException` [1],
- sinon, si le premier élément de la liste n'est pas l'élément neutre de l'addition, la fonction déclenche une erreur `IllegalArgumentException`,
- sinon, la fonction renvoie la somme de tous les éléments présents dans la liste `l`.

Pour itérer sur la liste, se reporter à l'annexe. On veillera à donner au paramètre `T` le majorant le plus grand possible (relativement à la relation de sous-typage), afin de rendre la fonction `sommeNAire` la plus générale possible.

```
class Calculs {
  @@@
  public static <T extends MonoideAdd<T>>
  T sommeNAire(List<T> l) {
    if (l.isEmpty())
      throw new IllegalArgumentException();
    T tete = l.get(0);
    if (!tete.equals(tete.zero()))
      throw new IllegalArgumentException();
    T r = tete;
    for (T e : l) {
      r = r.somme(e);
    }
    return r;
  }
  @@@
}
```

1. Pour lancer une exception `E` : **throw new** `E();`.

**Quelques structures algébriques** Toutes les structures implémentées dans les exercices suivants sont immutables.

**Exercice 2 (Les entiers naturels)** L'ensemble des entiers naturels $\{0, 1, \ldots\}$ peut être considéré comme un monoïde additif et comme un monoïde multiplicatif. Pour implémenter cette structure algébrique, on utilise le patron de conception "Composite", qui traduit directement la définition inductive suivante : un entier naturel est soit nul soit le successeur d'un entier naturel. On définit donc une interface, `Nat`, et deux classes d'implémentation, `Zero`, une classe singleton représentant zéro, et `Succ`, une classe implémentant les entiers naturels ayant un prédécesseur.

**2.1** Compléter la définition de l'interface `Nat` de manière à hériter des interfaces génériques `MonoideAdd` et `MonoideMul`, ainsi que de l'interface `FabriqueNats`, permettant de fabriquer des entiers naturels.

```
interface FabriqueNats {
  Nat creer(); // Crée un entier valant zéro
  Nat creer(Nat pred); // Crée un entier naturel égal
                       //   au successeur de pred
}
interface Nat extends
FabriqueNats, MonoideAdd<Nat>, MonoideMul<Nat> {
  boolean estZero(); // Teste à zéro l'entier naturel
  Nat predecesseur(); // Donne le prédécesseur s'il existe
  int val(); // Convertit l'entier naturel en int
  boolean equals(Object o); // Renvoie false
                            //   si o n'est pas de type Nat,
                            // teste l'égalité
                            //   des entiers naturels sinon
  String toString(); // Représente l'entier renvoyé par val
}
```

**2.2** Implémenter les méthodes hors fabriques (soit les accesseurs, les opérations algébriques, et autres services) dans la classe d'implémentation `Zero`, un singleton représentant l'entier nul (zéro) implémenté à l'aide d'une classe `enum`. La classe ne déclare pas d'attributs. Pour l'accesseur `predecesseur`, on déclenchera une erreur `UnsupportedOperationException`. Pour les quatre opérations algébriques, on veillera à utiliser les fabriques (méthodes `creer`). Il est inutile d'implémenter la méthode `equals` (et d'ailleurs impossible).

```
enum Zero implements Nat {
  SINGLETON;
  public boolean estZero() {
    return true;
  }
  public Nat predecesseur() {
    throw new UnsupportedOperationException();
  }
  @@@
```

```
  public Nat somme(Nat x) {
    return x;
  }
  public Nat zero() {
    return creer();
  }
  public Nat produit(Nat x) {
    return this;
  }
  public Nat un() {
    return creer(creer());
  }
  public int val() {
    return 0;
  }
  public String toString() {
    return "" + val();
  }
}
```

**2.3** Implémenter les méthodes hors fabriques (soit les accesseurs, les opérations algébriques, et autres services) dans la classe d'implémentation `Succ`, dont les instances représentent les entiers non nuls, soit les successeurs, entiers ayant un prédécesseur. Cette classe a un attribut, de type `Nat`. Pour les quatre opérations algébriques, on veillera à utiliser les fabriques (méthodes `creer`).

```
final class Succ implements Nat {
  private Nat pred;
  @@@
  public boolean estZero() {
    return false;
  }
  public Nat predecesseur() {
    return pred;
  }
  @@@
  public Nat somme(Nat x) {
    return creer(predecesseur().somme(x));
  }
  public Nat zero() {
    return creer();
  }
  public Nat produit(Nat x) {
    return x.somme(predecesseur().produit(x));
  }
  public Nat un() {
    return creer(creer());
  }
  public int val() {
    return 1 + predecesseur().val();
  }
```

```
  public boolean equals(Object o) {
    if (!(o instanceof Nat))
      return false;
    Nat x = (Nat) o;
    if (x.estZero())
      return false;
    return predecesseur().equals(x.predecesseur());
  }
  public String toString() {
    return "" + val();
  }
}
```

**2.4** Définir les constructeurs et les fabriques des classes `Zero` et `Succ`.

```
enum Zero implements Nat {
  SINGLETON;
  @@@
  public Nat creer() {
    return Zero.SINGLETON;
  }
  public Nat creer(Nat pred) {
    return new Succ(pred);
  }
  @@@
}
final class Succ implements Nat {
  @@@
  public Succ(Nat pred) {
    this.pred = pred;
  }
  @@@
  public Nat creer() {
    return Zero.SINGLETON;
  }
  public Nat creer(Nat pred) {
    return new Succ(pred);
  }
  @@@
}
```

**2.5** Tester les deux classes `Zero` et `Succ` en suivant le scénario suivant.
- Créer une fabrique de type `FabriqueNats`.
- Initialiser trois entiers naturels à zéro, un et deux, en utilisant la fabrique.
- Calculer la somme de deux et deux, et le produit de quatre fois deux.
- Afficher tous les entiers créés.

```
  FabriqueNats fab = Zero.SINGLETON;
  Nat zero = fab.creer();
  Nat un = fab.creer(zero);
```

```
  Nat deux = fab.creer(un);
  Nat quatre = deux.somme(deux);
  Nat huit = deux.produit(quatre);
  System.out.println("zéro : " + zero);
  System.out.println("un : " + un);
  System.out.println("deux : " + deux);
  System.out.println("quatre : " + quatre);
  System.out.println("huit : " + huit);
```

Par la suite, nous utiliserons une classe générique `Couple<T1, T2>` permettant de représenter des couples. Nous utiliserons aussi une interface générique `Representation`, permettant de représenter les éléments des structures algébriques avec des structures de données particulières (des couples pratiquement).

```
final class Couple<T1, T2> {
  public Couple(T1 c1, T2 c2) {
    pi1 = c1;
    pi2 = c2;
  }
  public final T1 pi1;
  public final T2 pi2;
}
interface Representation<Rep> {
  Rep val();
}
```

Les constructions suivantes utilisent toutes la technique de l'agrégation avec délégation.

**Exercice 3 (Les entiers relatifs)** À partir des entiers naturels, on peut définir les entiers relatifs. Une implémentation évidente consiste à rajouter à un entier naturel un signe. Nous étudions ici une implémentation plus raffinée, fondée sur une opération dite de symétrisation. Un entier relatif $z$ est alors représenté par un couple d'entiers naturels $(p, n)$, dont la différence vaut l'entier relatif : $z = p - n$. Comme il existe une infinité de tels couples, on doit quotienter l'ensemble des couples : deux couples sont équivalents s'ils correspondent aux mêmes différences.

$$(p, n) \equiv (p', n') \Longleftrightarrow p + n' = p' + n.$$

Cette équivalence sera implémentée par la méthode `equals` en Java.

**3.1** Compléter l'interface `Z` de manière à hériter des interfaces suivantes :
- l'interface `Representation<Couple<Nat, Nat>>` permettant de représenter un entier relatif par un couple d'entiers naturels,
- l'interface `FabriqueRelatifs<Z, Nat>` permettant de fabriquer des entiers relatifs à partir de deux entiers naturels,
- l'interface générique `AnneauUnitaire`.

```
interface FabriqueRelatifs<T, N> {
  T creer(N positif, N negatif); // Crée un entier relatif
```

```
                                 //  correspondant à la
                                 //  différence positif - négatif
}
interface Z extends
  Representation<Couple<Nat, Nat>>,
  FabriqueRelatifs<Z, Nat>,
  AnneauUnitaire<Z>
{
  boolean equals(Object o); // Renvoie false
                            //   si o n'est pas de type Z,
                            // teste l'égalité
                            //   des entiers relatifs sinon
  String toString(); // Représente l'entier relatif
                     //   sous la forme d'une différence
}
```

**3.2** Compléter la classe `Relatif` implémentant l'interface `Z`.

```
class Relatif implements Z {
  private Nat positif;
  private Nat negatif;
  public Relatif() {
    this(Zero.SINGLETON, Zero.SINGLETON);
  }
  public Relatif(Nat positif, Nat negatif) {
    this.positif = positif;
    this.negatif = negatif;
  }
  public Couple<Nat, Nat> val() {
    return new Couple<Nat, Nat>(positif, negatif);
  }
  public Z creer(Nat positif, Nat negatif) {
    return new Relatif(positif, negatif);
  }
  public Z somme(Z x) {
    Couple<Nat, Nat> rep = this.val();
    Couple<Nat, Nat> repX = x.val();
    Nat positif = rep.pi1.somme(repX.pi1);
    Nat negatif = rep.pi2.somme(repX.pi2);
    return creer(positif, negatif);
  }
  public Z zero() {
    Couple<Nat, Nat> rep = this.val();
    return creer(rep.pi1, rep.pi1);
  }
  public Z oppose() {
    Couple<Nat, Nat> rep = this.val();
    return creer(rep.pi2, rep.pi1);
  }
  public Z produit(Z x) {
    Couple<Nat, Nat> rep = this.val();
    Couple<Nat, Nat> repX = x.val();
```

```
    Nat positif = rep.pi1.produit(repX.pi1)
                .somme(rep.pi2.produit(repX.pi2));
    Nat negatif = rep.pi2.produit(repX.pi1)
                .somme(rep.pi1.produit(repX.pi2));
    return creer(positif, negatif);
  }
  public Z un() {
    Couple<Nat, Nat> rep = this.val();
    return creer(rep.pi1.un(), rep.pi1.zero());
  }
  public boolean equals(Object o) {
    if (!(o instanceof Z))
      return false;
    Z x = (Z) o;
    Couple<Nat, Nat> rep = this.val();
    Couple<Nat, Nat> repX = x.val();
    return rep.pi1.somme(repX.pi2).equals(
           rep.pi2.somme(repX.pi1));
  }
  public String toString() {
    return val().pi1 + " - " + val().pi2;
  }
}
```

**3.3** Tester la classe `Relatif` suivant le scénario suivant.

- Créer une fabrique de type `FabriqueRelatifs<Z, Nat>`.
- Initialiser trois entiers relatifs à zéro, moins un et moins deux, en utilisant la fabrique.
- Calculer le produit de moins deux fois moins deux.
- Afficher tous les entiers relatifs créés.

```
FabriqueRelatifs<Z, Nat> fabZ = new Relatif();
Z zeroZ = fabZ.creer(zero, zero);
Z moinsUn = fabZ.creer(un, deux);
Z moinsDeux = fabZ.creer(deux, quatre);
Z quatreZ = moinsDeux.produit(moinsDeux);
System.out.println("zéro Z : " + zeroZ);
System.out.println("moins un : " + moinsUn);
System.out.println("moins deux : " + moinsDeux);
System.out.println("quatre Z : " + quatreZ);
```

**3.4** Il apparaît que l'interface `Z` et la classe `Relatif` sont génériques en `Nat` : elles ne dépendent que des propriétés de monoïdes de cette structure. C'est en réalité un exemple d'une construction tout à fait générale, dite de symétrisation, permettant de définir un anneau unitaire à partir d'une structure qui est à la fois un monoïde additif et multiplicatif (un pré-anneau précisément). Compléter la déclaration de l'interface générique `Symetrise<T>` définissant le symétrisé d'un type `T` correspondant à un monoïde additif et à un monoïde multiplicatif. Cette interface hérite

de l'interface `FabriqueRelatifs` et aussi de l'interface `Representation`, afin d'exprimer qu'un relatif est représenté par un couple de deux `T`. Compléter enfin la classe `Diagonale`, classe générique implémentant l'interface `Symetrise` et utilisant un couple d'attributs correspondant à une différence.
Remarque : la méthode `equals`, telle que spécifiée, engendre un avertissement.

```
interface Symetrise<T extends MonoideAdd<T> & MonoideMul<T>>
extends
  Representation<Couple<T, T>>,
  FabriqueRelatifs<Symetrise<T>, T>,
  AnneauUnitaire<Symetrise<T>>
{
  boolean equals(Object o); // Renvoie false
                            //   si o n'est pas de type
                            //     Symetrise<T>,
                            // teste l'égalité des relatifs
                            //   sinon
  String toString(); // Représente le relatif
                     // sous la forme d'une différence
}

class Diagonale<T extends MonoideAdd<T> & MonoideMul<T>>
implements Symetrise<T> {
  private T positif;
  private T negatif;
  public Diagonale(T positif, T negatif) {
    this.positif = positif;
    this.negatif = negatif;
  }
  public Couple<T, T> val() {
    return new Couple<T, T>(positif, negatif);
  }
  public Symetrise<T> creer(T positif, T negatif) {
    return new Diagonale<T>(positif, negatif);
  }
  public Symetrise<T> somme(Symetrise<T> x) {
    Couple<T, T> rep = this.val();
    Couple<T, T> repX = x.val();
    T positif = rep.pi1.somme(repX.pi1);
    T negatif = rep.pi2.somme(repX.pi2);
    return creer(positif, negatif);
  }
  public Symetrise<T> zero() {
    Couple<T, T> rep = this.val();
    return creer(rep.pi1, rep.pi1);
  }
  public Symetrise<T> oppose() {
    Couple<T, T> rep = this.val();
    return creer(rep.pi2, rep.pi1);
  }
  public Symetrise<T> produit(Symetrise<T> x) {
    Couple<T, T> rep = this.val();
```

```
    Couple<T, T> repX = x.val();
    T positif = rep.pi1.produit(repX.pi1).somme(
                rep.pi2.produit(repX.pi2));
    T negatif = rep.pi2.produit(repX.pi1).somme(
                rep.pi1.produit(repX.pi2));
    return creer(positif, negatif);
  }
  public Symetrise<T> un() {
    Couple<T, T> rep = this.val();
    return creer(rep.pi1.un(), rep.pi1.zero());
  }
  public boolean equals(Object o) {
    try {
      @SuppressWarnings("unchecked")
      Symetrise<T> x = (Symetrise<T>) o;
      Couple<T, T> rep = this.val();
      Couple<T, T> repX = x.val();
      return rep.pi1.somme(repX.pi2).equals(
             rep.pi2.somme(repX.pi1));
    } catch (ClassCastException e) {
      return false;
    }
  }
  public String toString() {
    Couple<T, T> rep = this.val();
    return rep.pi1.toString() + " - " + rep.pi2.toString();
  }
}
```

**3.5** Tester la classe `Diagonale<Nat>` suivant le scénario précédent.

- Créer une fabrique de type `FabriqueRelatifs`.
- Initialiser trois entiers relatifs à zéro, moins un et moins deux, en utilisant la fabrique.
- Calculer le produit de moins deux fois moins deux.
- Afficher tous les entiers relatifs créés.

```
FabriqueRelatifs<Symetrise<Nat>, Nat> fabS =
  new Diagonale<Nat>(zero, zero);
Symetrise<Nat> zeroS = fabS.creer(zero, zero);
Symetrise<Nat> moinsUnS = fabS.creer(un, deux);
Symetrise<Nat> moinsDeuxS = fabS.creer(deux, quatre);
Symetrise<Nat> quatreS = moinsDeuxS.produit(moinsDeuxS);
System.out.println("zéro S: " + zeroS);
System.out.println("moins un S: " + moinsUnS);
System.out.println("moins deux S: " + moinsDeuxS);
System.out.println("quatre S: " + quatreS);
```

**Exercice 4 (Les rationnels)** À partir de l'anneau unitaire formé par les entiers relatifs, on peut définir les rationnels. Tout rationnel peut être représenté par une fraction

$$\frac{p}{q}$$

où $p$ est un entier relatif et $q$ un entier relatif non nul. Comme pour un rationnel donné il existe une infinité de telles fractions, on doit comme précédemment quotienter l'ensemble des fractions : deux fractions sont équivalentes si elles correspondent aux mêmes rapports.

$$\frac{p}{q} \equiv \frac{p'}{q'} \Longleftrightarrow p.q' = p'.q.$$

Cette équivalence sera implémentée par la méthode `equals` en Java.

**4.1** Compléter l'interface `Q` de manière à hériter des interfaces suivantes :
- l'interface `Representation<Couple<Z, Z>>` permettant de représenter un rationnel par un couple d'entiers relatifs,
- l'interface `FabriqueFractions<Q, Z>` permettant de fabriquer des rationnels à partir de deux entiers relatifs,
- l'interface générique `Corps`.

```
interface FabriqueFractions<T, A> {
  T creer(A numerateur, A denominateur); // Crée le rationnel
                                         // numerateur/denominateur
}
interface Q extends
  Representation<Couple<Z, Z>>,
  FabriqueFractions<Q, Z>,
  Corps<Q>
{
  String toString(); // Représente le rationnel
                     // sous la forme "numérateur/dénominateur"
  boolean equals(Object o); // Renvoie false
                            //   si o n'est pas de type Q,
                            // teste l'égalité des rationnels
                            //   sinon
}
```

**4.2** Compléter la classe `Rationnel` implémentant l'interface `Q`.

```
class Rationnel implements Q {
  private Z numerateur;
  private Z denominateur;
  public Rationnel(Z numerateur, Z denominateur) {
    this.numerateur = numerateur;
    this.denominateur = denominateur;
  }
  private Z getNumerateur() {
    return numerateur;
  }
  private Z getDenominateur() {
    return denominateur;
```

```java
  }
  public Couple<Z, Z> val() {
    return new Couple<>(getNumerateur(), getDenominateur());
  }
  public Q creer(Z numerateur, Z denominateur) {
    return new Rationnel(numerateur, denominateur);
  }
  public Q somme(Q x) {
    Z numerateurArg = x.val().pi1;
    Z denominateurArg = x.val().pi2;
    Z n = getNumerateur().produit(denominateurArg).somme(
            numerateurArg.produit(getDenominateur()));
    Z d = getDenominateur().produit(denominateurArg);
    return creer(n, d);
  }
  public Q zero() {
    return creer(getNumerateur().zero(), getNumerateur().un());
  }
  public Q oppose() {
    return creer(getNumerateur().oppose(), getDenominateur());
  }
  public Q produit(Q x) {
    Z numerateurArg = x.val().pi1;
    Z denominateurArg = x.val().pi2;
    return creer(getNumerateur().produit(numerateurArg),
                 getDenominateur().produit(denominateurArg));
  }
  public Q un() {
    return creer(getNumerateur().un(), getNumerateur().un());
  }
  public Q inverse() {
    return creer(getDenominateur(), getNumerateur());
  }
  public String toString() {
    return "(" + getNumerateur() + ")/("
                + getDenominateur() + ")";
  }
  public boolean equals(Object o) {
    if (!(o instanceof Q))
      return false;
    Q x = (Q) o;
    Z numerateurArg = x.val().pi1;
    Z denominateurArg = x.val().pi2;
    return getNumerateur().produit(denominateurArg).equals(
              numerateurArg.produit(getDenominateur()));
  }
}
```

**4.3** Tester la classe `Rationnel` suivant le scénario suivant.

– Créer une fabrique de type `FabriqueFractions<Q, Z>`.
– Initialiser deux rationnels à un demi et à deux, en utilisant la fabrique.

– Calculer la somme de un demi et de un demi, ainsi que leur produit.
– Afficher tous les rationnels précédents.
– Calculer l'inverse de deux, et tester l'égalité avec un demi.

```
FabriqueFractions<Q, Z> fabQ = new Rationnel(zeroZ, zeroZ);
Q unDemi = fabQ.creer(moinsUn, moinsDeux);
Q deuxQ = fabQ.creer(moinsDeux, moinsUn);
Q unQ = unDemi.somme(unDemi);
Q unQuart = unDemi.produit(unDemi);
System.out.println("un demi : " + unDemi);
System.out.println("deux : " + deuxQ);
System.out.println("un : " + unQ);
System.out.println("un quart : " + unQuart);
System.out.println("un demi égal à inverse de deux: "
                   + unDemi.equals(deuxQ.inverse()));
```

**4.4** Comme pour le symétrisé de `Nat`, il apparaît que l'interface `Q` et la classe `Rationnel` sont génériques en `Z` : elles ne dépendent que des propriétés d'anneau unitaire de cette structure. C'est de nouveau un exemple d'une construction tout à fait générale, permettant de définir le corps des fractions à partir d'un anneau unitaire (et intègre [2]). Compléter la déclaration de l'interface générique `Fraction<T>` définissant le corps des fractions d'un type `T` correspondant à un anneau unitaire. Cette interface hérite de l'interface `FabriqueFractions` et aussi de l'interface `Representation`, afin d'exprimer qu'une fraction est représentée par un couple de deux `T`. Compléter enfin la classe `Rapport`, classe générique implémentant l'interface `Fraction` et utilisant un couple d'attributs correspondant à un rapport (une fraction).
Remarque : la méthode `equals`, telle que spécifiée, engendre un avertissement.

```
interface Fraction<T extends AnneauUnitaire<T>> extends
  Representation<Couple<T, T>>,
  FabriqueFractions<Fraction<T>, T>,
  Corps<Fraction<T>>
{
  String toString(); // Représente le rationnel
                     // sous la forme "numerateur/denominateur"
  boolean equals(Object o); // Renvoie false
                            //   si o n'est pas de type
                            //     Fraction<T>,
                            // teste l'égalité des rationnels
                            //   sinon
}

class Rapport<T extends AnneauUnitaire<T>>
implements Fraction<T> {
  private T numerateur;
  private T denominateur;
  public Rapport(T numerateur, T denominateur) {
```

2. L'anneau ne doit pas avoir de diviseurs de zéro. Cette propriété reste implicite ici, et correspond à une spécification à vérifier.

```
    this.numerateur = numerateur;
    this.denominateur = denominateur;
  }
  public Couple<T, T> val() {
    return new Couple<>(this.getNumerateur(),
                        this.getDenominateur());
  }
  private T getNumerateur() {
    return numerateur;
  }
  private T getDenominateur() {
    return denominateur;
  }
  public Fraction<T> creer(T numerateur, T denominateur) {
    return new Rapport<T>(numerateur, denominateur);
  }
  public Fraction<T> somme(Fraction<T> x) {
    T numerateurArg = x.val().pi1;
    T denominateurArg = x.val().pi2;
    T n = getNumerateur().produit(denominateurArg).somme(
          numerateurArg.produit(getDenominateur()));
    T d = getDenominateur().produit(denominateurArg);
    return creer(n, d);
  }
  public Fraction<T> zero() {
    return creer(getNumerateur().zero(), getNumerateur().un());
  }
  public Fraction<T> oppose() {
    return creer(getNumerateur().oppose(), getDenominateur());
  }
  public Fraction<T> produit(Fraction<T> x) {
    T numerateurArg = x.val().pi1;
    T denominateurArg = x.val().pi2;
    return creer(getNumerateur().produit(numerateurArg),
                 getDenominateur().produit(denominateurArg));
  }
  public Fraction<T> un() {
    return creer(getNumerateur().un(), getNumerateur().un());
  }
  public Fraction<T> inverse() {
    return creer(getDenominateur(), getNumerateur());
  }
  public String toString() {
    return "(" + getNumerateur() + ")/("
           + getDenominateur() + ")";
  }
  public boolean equals(Object o) {
    try {
      @SuppressWarnings("unchecked")
      Fraction<T> x = (Fraction<T>) o;
      T numerateurArg = x.val().pi1;
      T denominateurArg = x.val().pi2;
```

```
      return getNumerateur().produit(denominateurArg).equals(
             numerateurArg.produit(getDenominateur()));
    } catch (ClassCastException e) {
      return false;
    }
  }
}
```

**4.5** Tester la classe `Rapport<Symetrise<Nat>>` suivant le scénario précédent.
- Créer une fabrique de type `FabriqueFractions`.
- Initialiser deux rationnels à un demi et à deux, en utilisant la fabrique.
- Calculer la somme de un demi et de un demi, ainsi que le produit.
- Afficher tous les rationnels précédents.
- Calculer l'inverse de deux, et tester l'égalité avec un demi.

```
FabriqueFractions<Fraction<Symetrise<Nat>>, Symetrise<Nat>>
  fabFS = new Rapport<Symetrise<Nat>>(zeroS, zeroS);
Fraction<Symetrise<Nat>>
  unDemiFS = fabFS.creer(moinsUnS, moinsDeuxS);
Fraction<Symetrise<Nat>>
  deuxFS = fabFS.creer(moinsDeuxS, moinsUnS);
Fraction<Symetrise<Nat>>
  unFS = unDemiFS.somme(unDemiFS);
Fraction<Symetrise<Nat>>
  unQuartFS = unDemiFS.produit(unDemiFS);
System.out.println("un demi : " + unDemiFS);
System.out.println("deux : " + deuxFS);
System.out.println("un : " + unFS);
System.out.println("un quart : " + unQuartFS);
System.out.println("un demi égal à inverse de deux: "
                   + unDemiFS.equals(deuxFS.inverse()));
```

## Annexe

Le paquet `java.util` contient l'interface générique `List` et une classe générique d'implémentation `LinkedList`, possédant un constructeur sans paramètre. Pour parcourir une liste `l` de type `List<T>`, on procède ainsi.

```
for(T e : l){
  @@@ // Code utilisant e
}
```