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

TP2 – Exception et héritage

Informatique – MAPD

Correction

Objectifs

- Créer et utiliser des exceptions à bon escient dans un programme Java
- Écrire une classe Java qui hérite d'une autre en utilisant les bons éléments de programmation (`super`, chaînage de constructeurs, redéfinition)
- Expliquer les concepts de surcharge et redéfinition et les contraintes associées

1 Manipulation des exceptions

Exercice 1 (*Exception*)

Relire le cours sur la partie gestion des exceptions et reprendre votre solution du TP1 pour les rationnels à la question 1.4.

▷ **Question 1.1 :**
Créez une exception `UndefinedRational` représentant le cas où le dénominateur du rationnel est nul. Puis utilisez-la dans les constructeurs.

UndefinedRational.java

```
package rational;

/**
 * Une exception pour le cas du dénominateur == 0.
 * @author jroyer
 *
 */
public class UndefinedRational extends Exception {
```

```
      // sinon je suis une classe comme les autres

      private String msg = " Vous me copierez 100 fois :\n"
                  + " je ne sais pas définir un rationnel avec un dénominateur nul";

      /**
       * May be better this is inherited from the Throwable superclass.
       */
      public String getMessage() {
            return this.msg;
      }
}
```

Rational.java

```
package rational;

public class Rational {

      private int numerator;
      private int denominator;

      public Rational(int n, int d) throws UndefinedRational {
            if (d==0) {
                  throw new UndefinedRational();
            } else {
                  numerator = n;
                  denominator = d;
            }
      }
```

▷ **Question 1.2 :**
Dans quelle méthode devez-vous utiliser cette exception ? Quelles sont les alternatives pour la gestion d'une exception ? Et dans quels cas utiliser ces alternatives ?

— Dans les méthodes qui vont créer directement ou indirectement des rationnels.
— Soit vous propagez l'exception à la méthode appelant le code ayant généré cette exception :

```
      /**
       * Exemple de propagation de l'exception.
       * La bonne solution ici
       * @param r
       * @return
       * @throws UndefinedRational
       */
      public Rational sum(Rational r) throws UndefinedRational {
            return new Rational(
                        this.numerator * r.denominator + r.numerator * this.denominator,
                        this.denominator * r.denominator);
```

```
}
```

— Soit vous traitez cette exception :

```
/**
 * Exemple de traitement locale de l'exception.
 * @param r
 * @return
 */
public Rational diff(Rational r) {
    try {
        return this.sum(r.neg());
    } catch (UndefinedRational e) {
        e.printStackTrace();
        return null;
    }
}
```

— Les 2 solutions sont très contextuelles . . .. Dans notre cas, `sum`, `diff` et `times` ne peuvent pas produire d'exception si les opérandes sont des rationnels corrects. Le **`catch`** serait la preuve qu'on a pu créer un rationnel avec un dénominateur nul. La seconde approche est à préférer car elle cache l'exception qui ne peut pas pouvoir être levée.

▷ **Question 1.3 :**
Vous avez utilisé des exceptions dites « *checked* », que sont les exceptions « *unchecked* » ?

Hint : `https://www.jmdoudoux.fr/java/dej/chap-exceptions.htm`

Sont *unchecked* `Error`, `RuntimeException` et leurs sous classes : voir `https://docs.oracle.com/javase/8/docs/api/`

2 Héritage

Nous allons commencer par faire hériter votre classe d'une autre classe existante.

Exercice 2 (*héritage*)

▷ **Question 2.1 :**
Cherchez dans la documentation Java la classe `Number`. Quelles sont les obligations de programmation des sous-classes ? Où cette classe est-elle utilisée ?

On peut trouver dans `https://docs.oracle.com/javase/10/docs/api/java/lang/Number.html` les informations.
Les obligations de programmation sont :

— **`abstract double`** `doubleValue()`
Returns the value of the specified number as a double, which may involve rounding.

— **abstract float** floatValue()
Returns the value of the specified number as a float, which may involve rounding.
— **abstract int** intValue()
Returns the value of the specified number as an int, which may involve rounding or truncation.
— **abstract long** longValue()
Returns the value of the specified number as a long, which may involve rounding or truncation.

Faire un Search Number ; on trouve aussi les paramètres de fonction qui utilisent Number (pas si fréquent).

▷ **Question 2.2 :**
Intégrez votre code à la hiérarchie des java.lang.Number de Java en respectant les obligations de programmation qui en découlent. Testez.

```
package rational;
public class RationalNumber extends Number {
    private int numerator;
    private int denominator;

    public RationalNumber(final int n, final int d) throws UndefinedRational {
      if (d ==0) throw new UndefinedRational();
      denominator = d;
      numerator = n;
    }
    @Override
    public String toString() {
      return numerator + (denominator == 1?"":"/" + denominator);
    }

    @Override
    public boolean equals(Object o) {
      if ((o != null) && !(o instanceof RationalNumber)) return false;
      RationalNumber r = (RationalNumber) o;
      return this.numerator * r.denominator == r.numerator * this.denominator;
    }

    public RationalNumber sum(RationalNumber r) {
      try {
          return new RationalNumber(this.numerator * r.denominator + r.numerator * this.denomin
                            this.denominator * r.denominator);
      } catch (UndefinedRational e) {
          // never occur
          return null;
      }
    }

    public RationalNumber diff(RationalNumber r) {
          return this.sum(r.neg());
    }
```

```
public RationalNumber times(RationalNumber r) {
  try {
      return new RationalNumber(this.numerator * r.numerator,
                        this.denominator * r.denominator);
  } catch (UndefinedRational e) {
      // never occur
      return null;
  }
}

public RationalNumber neg(){
  try {
      return new RationalNumber(- this.numerator,this.denominator);
  } catch (UndefinedRational e) {
      // never occur
      return null;
  }
}

@Override
public final int intValue() {
  return numerator / denominator;
}

@Override
public final long longValue() {
  return (long) numerator / (long) denominator;
}

@Override
public final float floatValue() {
  return (float) numerator / (float) denominator;
}

@Override
public final double doubleValue() {
  return (double) numerator / (double) denominator;
}

/**
 * @param args
 */
public static void main(String[] args) throws UndefinedRational {
  RationalNumber r;
  try{
      r = new RationalNumber(1,0);
      System.out.println("error : UndefinedRational must be raised");

  } catch(UndefinedRational e){
```

```
      }
      r = new RationalNumber(1,3);
      System.out.println("intValue "+r.intValue());
      System.out.println("longValue "+r.longValue());
      System.out.println("floatValue "+r.floatValue());
      System.out.println("doubleValue "+r.doubleValue());
    }
}
```

Reprenez maintenant votre classe et définissez une sous-classe qui mémorise le pgcd du numérateur et du dénominateur.

▷ **Question 2.3 :**
Créez la sous-classe permettant de mémoriser le PGCD (`WithPGCD`), celui-ci ne doit être calculé qu'au plus une fois. Utilisez le chaînage de constructeur et redéfinissez `toString` intelligemment.

Pas vraiment de difficulté.
Si vous avez choisi une implantation *non mutable* , ce qui est un bon choix, le pgcd ne change jamais ; il faut donc le calculer une seule fois (à la création) et le stocker (dans un attribut par exemple). Si vous avez fait un choix *mutable*. . ., c'est un peu plus compliqué.

```
package rational;

/**
 * Une sous-classe de  rationnels
 * @author jroyer
 *
 */
public class WithPGCD extends Rational {

    /**
     * The pgcd attribute
     */
    private int pgcd;

    /**
     * To store  the pgcd
     * @param n
     * @param d
     * @throws UndefinedRational
     */
    public WithPGCD(int n, int d) throws UndefinedRational {
      super(n, d);
      // compute gcd
      if (n == 0) {
          this.pgcd = 0;
      } else {
          int a = Math.abs(n);
          int b = Math.abs(d);
          while (a != b) {
```

```
            if (a > b) {
                a = a - b;
            } else {
                b = b - a;
            }
        }
        this.pgcd = a;
    }
}

/**
 * Redefinition with super.
 */
public String toString() {
    return super.toString() + "\n PGCD= " + this.getPGCD();
}

/**
 * A getter.
 * @return
 */
private int getPGCD() {
    return this.pgcd;
}


// ========================= des questions
//1.Est-ce que je peux la redéfinir avec la signature NON
// public Rational sum(Rational r) throws UndefinedRational Exception

//2.Est-ce que je peux la redéfinir avec la signature NON c'est une surcharge
// public WithPGCD sum(WithPGCD r) throws UndefinedRational return null;

//3.Quelle est la nature de la méthode avec la signature UNE redéfinition
// public WithPGCD sum(Rational r) throws UndefinedRational return null;

//4.Est ce une redéfinition correcte NON c'est une surcharge
//public Object sum(WithPGCD r) throws UndefinedRational return null;

//5.Puis-je définir une surcharge de sum : NON
// public Object sum(Rational r) throws UndefinedRational return null;

//6.Puis-je définir une surcharge de sum YES
//public Object sum(WithPGCD r, Object o) throws UndefinedRational return null;

// ========================= des tests
/**
 * Tests locaux (bad practice)
 * @param args
 */
public static void main(String[] args) {
```

```
        WithPGCD r0, r1, r2, r3, r4, r5;
        try {
            //r0 = new WithPGCD(10,2);
            //System.out.println("r0 = " + r0);
            r1 = new WithPGCD(0,6);
            System.out.println("r1 = " + r1);
            r5 = new WithPGCD(2,-6);
            System.out.println("r5 = " + r5);
            r2 = new WithPGCD(-2,3);
            System.out.println("r2 = " + r2);
            r3 = new WithPGCD(1,2);
            System.out.println("r3 = " + r3);
            System.out.println("r1 = r2 ? " + r1.equals(r2));
            System.out.println("r1 = r3 ? " + r1.equals(r3));
            System.out.println("r1+r2 = " + (r1.sum(r2)));
            r4 = new WithPGCD(-8,-4);
            System.out.println("r4 = " + r4);
            System.out.println("r1 * r2 = " + r1.times(r2));
            System.out.println("-(r3 * r4) = " + (r3.times(r4)).neg());
            System.out.println("r3 * r3 = " + (r3.times(r3)));
        } catch (UndefinedRational e) {
            //e.printStackTrace();
            System.err.println("Y'a un dénominateur nul dans mes tests !!!");
        }
    }

}
```

▷ **Question 2.4 :**
Soit la méthode `sum` dans la classe `Rational` ayant pour signature `public Rational sum(Rational r) throws UndefinedRational`.

Dans la classe `WithPGCD` est-il possible de :

1. **redéfinir avec la signature `public Rational sum(Rational r) throws UndefinedRational, Exception` ?**
2. **redéfinir avec la signature `public WithPGCD sum(WithPGCD r) throws UndefinedRational` ?**
3. **redéfinir avec la signature `public WithPGCD sum(Rational r ) throws UndefinedRational` ?**
4. **redéfinir avec la signature `public Object sum(WithPGCD r) throws UndefinedRational` ?**
5. **faire une surcharge avec `public Object sum(Rational r) throws UndefinedRational` ?**
6. **faire une surcharge de `sum` `public Object sum(WithPGCD r, Object o) throws UndefinedRational` ?**

Faites une hypothèse, puis essayez et expliquez votre observation.

1. Est-ce que je peux la redéfinir avec la signature

```
public Rational sum(Rational r) throws UndefinedRational, Exception {
  return null;
}
```

   Réponse : non à cause de la nouvelle `Exception`

2. Est-ce que je peux la redéfinir avec la signature
   `public WithPGCD sum(WithPGCD r) throws UndefinedRational { return null}`
   Réponse : non c'est une surcharge (type du paramètre différent)
3. Quelle est la nature de la méthode avec la signature
   `public WithPGCD sum(Rational r) throws UndefinedRational{ return null}`
   Réponse : c'est une redéfinition (type de retour est un sous type du type de retour initial)
4. Est ce une redéfinition correcte
   `public Object sum(WithPGCD r) throws UndefinedRational {return null;}`
   Réponse : NON c'est une surcharge (type du paramètre différent du type initial)
5. Puis-je définir une surcharge de `sum`
   `public Object sum(Rational r) throws UndefinedRational{ return null;}`
   Réponse : TENTATIVE INCORRECTE de redéfinition et donc ne veut pas considérer ce cas comme surcharge non plus
6. Puis-je définir une surcharge de `sum`

```
public Object sum(WithPGCD r, Object o) throws UndefinedRational{
  return null;
}
```

   Réponse : OUI

La différence entre les deux derniers cas peut s'expliquer ainsi :

Le type de retour ne fait pas partie de la signature officielle en Java, seulement les arguments et la classe de définition (information utile pour la généricité et le mécanisme d'effacement). Donc dans l'avant dernier cas le compilateur identifie une tentative de redéfinition mais n'est pas content car le type de retour n'est pas covariant avec la signature redéfinie.

Par contre dans le dernier cas il identifie une surcharge et donc se moque complètement du type de retour.