IUT de Paris Descartes – Base de la Programmation Objet

Travaux Pratiques – Sujet n°9

Objectifs

— Concevoir ses propres classes d'exception
— Lever et attraper des exceptions

Un tableur étendu

1. Nous voulons introduire dans le tableur fait en séance de travaux dirigés un nouveau type de cellule pouvant contenir de simple chaîne de caractères. Ce nouveau type de cellule ne pourra pas intervenir dans des calculs mais pourra uniquement être affiché (via `toString`). En conséquence, l'appel de la méthode `évaluer` sur un objet de ce type devra lever une exception particulière.

   Les sources constituant le point de départ de votre travail sont sur le serveur.

2. Définissez une classe d'exception nommée `CelluleNonEvaluableException`. Pour être une classe d'exception, cette classe doit dériver de la classe `Exception` (ou de toute autre classe d'exception telles que `RuntimeException`, etc).

   Solution: Cette classe doit être déclarée dans le paquetage `tableur` car nous verrons plus tard que toutes les cellules (et donc la classe `Cellule`) doivent s'en servir.

```
package tableur;

@SuppressWarnings("serial")
public class CelluleNonEvaluableException extends Exception {
}
```

3. Définissez un nouveau sous-type de `Cellule` nommé `CelluleTexte` et représentant les cellules textuelles. Vous ferez en sorte qu'une invocation de la méthode `évaluer` lève une exception de type `CelluleNonEvaluableException`. De plus, vous vous assurerez que l'affichage d'une somme (avec la classe `CelluleAddition`) mettant en oeuvre un objet de type `CelluleTexte` produit le caractère 'E' (pour erreur).

   Solution: La méthode `CelluleTexte.évaluer` ne peut lever une exception que si la méthode `Cellule.évaluer` déclare possible cette levée. Nous faisons en sorte que cette dernière méthode ne soit plus abstraite mais lève cette exception. De plus, il est nécessaire de prendre en compte la levée potentielle d'une exception dans la méthode `Cellule.toString` car elle invoque `Cellule.évaluer`.

```
public abstract class Cellule {
  public int évaluer() throws CelluleNonEvaluableException {
    throw new CelluleNonEvaluableException();
  }
```

```
  @Override
  public String toString() {
    try {
      return Integer.toString(évaluer());
    }
    catch (CelluleNonEvaluableException e) {
      return "E";
    }
  }
}
```

La classe `CelluleTexte` (définie dans la paquetage `cellules`) se contente de spécialiser la méthode `toString`.

```
package cellules;

public class CelluleTexte extends Cellule {
  private String s;

  public CelluleTexte(String s) {
    this.s = s;
  }

  @Override
  public String toString() {
    return s;
  }
}
```

Enfin, il est nécessaire de préciser qu'à présent la méthode `CelluleAddition.évaluer` peut lever une exception. Il est important de noter qu'il n'est pas nécessaire de le préciser dans les autres sous-classes de `Cellule` telles que `CelluleVide` ou `CelluleConstante`.

```
public class CelluleAddition extends Cellule {
  ...

  @Override
  public int évaluer() throws CelluleNonEvaluableException {
    return t.get(c1).évaluer() + t.get(c2).évaluer();
  }
}
```

4. Définissez un nouveau sous-type de `Cellule` nommé `CelluleDivision` permettant de définir une cellule comme étant la division entière de deux autres cellules.

Solution: La classe `CelluleDivision` est définie dans le paquetage `cellules`.

```
package cellules;

public class CelluleDivision extends Cellule {
  private Tableur t;
  private Coord c1, c2;

  public CelluleDivision(Tableur t, Coord c1, Coord c2) {
    assert(t.coordCorrecte(c1) && t.coordCorrecte(c2));
    this.t = t;
    this.c1 = c1;
```

```
    this.c2 = c2;
  }

  @Override
  public int évaluer() throws CelluleNonEvaluableException {
    int diviseur = t.get(c2).évaluer();
    if (diviseur != 0)
      return t.get(c1).évaluer() / diviseur;
    throw new CelluleNonEvaluableException();
  }
}
```

5. Faites en sorte que l'affichage d'une cellule mettant en oeuvre une division par zéro produise le caractère 'D'.

Solution: On commence par modifier la classe `CelluleNonEvaluableException` de façon à ce qu'un label (une chaîne de caractère) soit associé à ce type d'exception.

```
@SuppressWarnings("serial")
public class CelluleNonEvaluableException extends Exception {
  public String getLabel() {
    return "E"; // label par défaut
  }
}
```

La méthode `toString` de la classe `Cellule` est modifiée en conséquence.

```
public abstract class Cellule {
  ...

  @Override
  public String toString() {
    try {
      return Integer.toString(évaluer());
    }
    catch (CelluleNonEvaluableException e) {
      return e.getLabel();
    }
  }
}
```

La classe `CelluleDivision` peut maintenant lever son propre type d'exception en spécialisant `CelluleNonEvaluableException`. Vous noterez la syntaxe permettant de déclarer une classe interne à une autre classe. Cette syntaxe a été vue dans la première partie du module.

```
public class CelluleDivision extends Cellule {
  ...
  @SuppressWarnings("serial")
  static private class DivParZéroException
                       extends CelluleNonEvaluableException {
    @Override
    public String getLabel() {
      return "D"; // label spécifique
    }
  }

  @Override
```

```
  public int évaluer() throws CelluleNonEvaluableException {
    int diviseur = t.get(c2).évaluer();
    if (diviseur != 0)
      return t.get(c1).évaluer() / diviseur;
    throw new DivParZéroException(); // erreur spécifique
  }
}
```