TP Noté 2007 - Durée : 2h

IB2 : Informatique de Base 2

Première année

Nancy-Université

Supports et notes de Cours/TD/TP autorisés.

Élements fournis. Vous trouverez dans le répertoire `/home/depot/1A/IB2/TP_NOTE` un ensemble de classes et d'interfaces que vous devrez compléter au fur et à mesure des questions. Recopier ces classes dans le répertoire `/home/depot/TP-NOTES/IB2-<horaire>/<login>`

Tests et Compilation. Parmi les classes fournies, vous trouverez une classe `Test` vous permettant de tester les différentes classes que vous implanterez. Cette classe vous affichera quels sont les tests qui ne s'exécutent pas correctement et vous donnera à titre indicatif un score pouvant s'apparenter à votre note de TP. **À la fin de la séance de TP, il est impératif que toutes vos classes compilent ! Un manquement à cette règle sera fatalement sanctionné.**

Le but du TP est de commencer la modélisation d'expressions dont les constantes et les variables prennent leurs valeurs dans les entiers, les réels, les booléens, etc. Ce genre de modélisation est utilisée dans les formules des tableurs comme MICROSOFT EXCEL ou des logiciels de calcul formel comme MAPLE ou MATHEMATICA. Par exemple, dans MICROSOFT EXCEL l'évaluation de l'expression `SI(21<5 ;3 ;"xxx")` retournerait en résultat la chaîne de caractères `"xxx"`, alors que dans le cas ou l'expression conditionnelle eût été vraie, le résultat aurait été la valeur entière 3.

L'ensemble des valeurs possibles pour les constantes et les variables s'appelle le *domaine*. On notera `Domaine` notre domaine. Le travail demandé dans cette partie se limitera à l'implantation des domaines booléens et entiers. `Domaine` est une interface appartenant au package `expressions` dont la définition est la suivante :

expressions/Domaine.java

```
1  package expressions;
2
3  public interface Domaine {
4      public abstract Domaine plus(Domaine x);
5      public abstract Domaine fois(Domaine x);
6      public abstract Domaine moins(Domaine x);
7      public abstract Domaine div(Domaine x);
8
9      public abstract boolean isLogique();
10     public abstract boolean isNumerique();
11     public abstract boolean isErreur();
12 }
```

Les opérations `plus` et `fois` auront leur sens habituel dans les entiers décrits par la classe `Entier` . Ces opérations seront traduites respectivement par le `OU` et le `ET` dans le type booléen `Booleen`. Les opérations `moins` et `div` correspondront à la soustraction et à la division (entière comme en Java dans la classe `Entier`). Dans le domaine `Booleen`, l'expression `a.moins(b)` devra signifier `a OU NON b`, tandis que l'expression `a.div(b)` retournera une erreur.

On voit donc apparaître la nécessité d'avoir une classe `Erreur` qui est la classe des valeurs retournées par des calculs impossibles.

expressions/Erreur.java

```
1  package expressions;
2
3  public class Erreur implements Domaine {
4      protected Erreur() { }
5      private static final Erreur erreur = new Erreur();
6      public static Erreur getInstance() { return Erreur.erreur; }
7
8      public Domaine plus(Domaine x)  { return this; }
9      public Domaine fois(Domaine x)  { return this; }
10     public Domaine moins(Domaine x) { return this; }
11     public Domaine div(Domaine x)   { return this; }
12
13     public boolean isLogique()   { return false; }
14     public boolean isNumerique() { return false; }
15     public boolean isErreur()    { return true;  }
16
17     public String toString() { return "#valeur"; }
18 }
```

Cette classe s'utilise de la manière suivante :

```
if ( /* condition rendant le calcul erroné */ ) {
   return Erreur.getInstance();
}
```

▷ **Question 1. Implantation de la classe `Booleen`.**

(a) Observer attentivement le squelette de la classe `Booleen` qui vous est fournie (dont une version simplifiée vous est donnée ci-dessous) :

expressions/Booleen.java

```
package expressions;

public class Booleen implements Domaine {
     // la valeur logique enveloppée :
    private boolean valeur;

    // Constructeur
    public Booleen(boolean valeur) {
        this.valeur = valeur;
    }

    public boolean toBoolean() {
        return this.valeur;
    }

    public Booleen neg() {
        return new Booleen(! this valeur);
    }

    // À compléter
    // ...
}
```

(b) Implanter les méthodes provenant de l'interface `Domaine`. Pour chaque méthode, il est nécessaire de tester que l'argument est bien un `Booleen`, et dans le cas contraire il faut retourner une valeur d'erreur. Les méthodes qui n'ont pas de raisons d'être définies dans la classe `Booleen` doivent retourner également une valeur d'erreur.

(c) Redéfinir la méthode `public boolean equals(Object o)` pour qu'elle retourne vrai si le paramètre `o` est une instance de `Booleen`, et si `this` et `o` traduisent la même valeur logique.

(d) Redéfinir la méthode `public String toString()` pour qu'elle retourne la chaîne de caractères `"Vrai"` ou `"Faux"` selon la valeur logique enveloppée.

(e) Compiler et exécuter la classe `Test` fournie après avoir pris soin de décommenter uniquement le code relatif à la *Partie I* de la série de tests.

▷ **Question 2. Implantation de la classe `Entier`.**

(a) Observer le squelette de la classe `Entier` fournie :

expressions/Entier.java

```
package expressions;

public class Entier extends Numerique {
     // la valeur enveloppée :
    private int valeur;

    // Constructeur
    public Entier(int valeur) {
        this.valeur = valeur;
    }

    // À compléter
    // ...
}
```

(b) Définir ou redéfinir les méthodes en testant que l'argument est bien un `Entier` , et en retournant la valeur d'erreur dans le cas contraire. En cas de divisions par zéro, une erreur doit également être retournée.

(c) Redéfinir la méthode `public boolean equals(Object o)` pour qu'elle teste si le paramètre `o` est une instance de `Entier`, et si `this` et `o` traduisent la même valeur.

(d) Redéfinir la méthode `public String toString()` pour qu'elle retourne la traduction en une chaîne de caractères de la valeur. La chaîne de caractères retournée ne doit absolument pas contenir d'espace ni avant, ni après, ni entre le signe `'-'` et des chiffres si la valeur est négative.

(e) Compiler et exécuter la classe `Test` fournie après avoir pris soin de décommenter le code relatif à la *Partie II* de la série de tests.

▷ **Question 3. Implantation de la classe `Const`.**

(a) On a simplifié l'interface `Expr` étudiée en TD en la définissant de la manière suivante :

expressions/Expr.java

```
1 package expressions;
2
3 public interface Expr {
4     public abstract Domaine evaluer();
5     public abstract String decompiler();
6     public abstract Expr dupliquer();
7 }
```

Observer le squelette de la classe `Expr` fournie en notant bien que le domaine des valeurs calculées par la méthode `evaluer()` est maintenant de type `Domaine`

(b) Observer la classe `Const` fournie que nous allons utiliser pour modéliser les constantes :

expressions/Const.java

```
1  package expressions;
2
3  public class Const implements Expr {
4
5      // valeur de la constante
6      private Domaine valeur;
7
8      // Constructeur
9      public Const(Domaine valeur) {
10         this.valeur = valeur;
11     }
12
13     // À compléter
14     // ...
15 }
```

(c) Compléter cette classe en implantant les méthodes provenant de l'interface `Expr`. Pour réaliser la méthode `decompiler()` on pourra s'aider du code de la classe `Test`.

(d) Redéfinir la méthode `public boolean equals(Object o)` pour qu'elle teste si le paramètre `o` est une instance de `Const`, et si `this` et `o` ont des valeurs égales.

(e) Compiler et exécuter la classe `Test` fournie après avoir pris soin de décommenter le code relatif à la *Partie III* de la série de tests.

▷ **Question 4. Implantation de la classe `Plus`.**

(a) Examiner attentivement le code source de la classe `Binaire` fournie, tout particulièrement le profile de la méthode `public abstract Domaine calculer(Domaine x, Domaine y)`. Cette méthode retournera le résultat dans `Domaine` du calcul effectué sur les valeurs des deux fils lors de l'évaluation.

(b) Compléter la classe `Plus` qui hérite de la classe `Binaire`, et qui définit ou redéfinit ce qu'il faut. On fera en sorte que la duplication d'une instance soit une copie en profondeur.

(c) Redéfinir la méthode `public boolean equals(Object o)` pour qu'elle teste si le paramètre `o` est une instane de `Plus`, et si `this` et `o` ont des fils égaux.

(d) Compiler et exécuter la classe `Test` fournie après avoir pris soin de décommenter le code relatif à aux *Parties IV et V* de la série de tests. Procéder par étapes (*Partie IV* puis *Partie V*).

▷ **Question 5. Implantation de la classe `Si`.**

(a) Observer attentivement le squelette de la classe `Si` qui vous est fournie. Cette classe implante directement l'interface `Expr`. Chaque instance de cette classe possède trois attributs de type `Domaine`.

expressions/Si.java

```
1  package expressions;
2
3  public class Si implements Expr {
4
5      // les attributs
6      private Expr c; // la condition
7      private Expr vsv; // la valeur si c est vraie
8      private Expr vsf; // la valeur si c est fausse
9
10     // Constructeur
11     public Si(Expr c, Expr vsv, Expr vsf) {
12         this.c = c;
13         this.vsv = vsv;
14         this.vsf = vsf;
15     }
16
17     // À compléter
18     // ...
19 }
```

(b) Compléter la classe `Si` en contrôlant le type de l'évaluation de la condition `c` lors de l'évaluation de `this`. Si ce n'est pas une constante logique, on retournera une valeur d'erreur. Sinon, la valeur retournée est celle de `vsv` si `c` est évaluée en vraie, ou `vsf` dans le cas contraire. On ne compare pas ici les types de `vsv` et `vsf`. Comme dans la classe `Plus`, la duplication d'une instance se fera en profondeur.

(c) Redéfinir la méthode `public boolean equals(Object o)` comme dans les questions précédentes.

(d) Compiler et exécuter la classe `Test` fournie après avoir pris soin de décommenter le code relatif à aux *Parties VI* de la série de tests.