INF112 – Programmation Orientée Objet en JAVA
Contrôle de connaissances (1h30)
22 avril 2024

Nom :
Prénom :
Groupe :

— Ce sujet d'examen comporte 12 pages.
— Une seule feuille de papier A4 recto-verso est autorisée. Téléphones et ordinateurs éteints.
— Le barème est donné à titre indicatif. Il pourra être revu si les correcteurs l'estiment nécessaire.
— Les réponses aux questions s'écrivent dans les boîtes prévues à cet effet.
— La taille des boîtes de réponses n'est pas forcément proportionnelle à la taille des réponses attendues.
— Si vous constatez une erreur dans l'énoncé, écrivez comment vous le corrigez et continuez votre rédaction.

1. (2 points) En orienté objet, une instance de classe (objet) contient des valeurs qui permettent de caractériser l'objet que l'on veut modéliser. Quel type d'attribut permet aux objets de communiquer entre eux ? Décrivez comment cette communication est réalisée.

(0,5) Référence.

(1,5) Pour chaque type de message que l'objet peut recevoir, la classe de l'objet définie une méthode associée au type de message. Cette méthode est une procédure qui est exécutée par l'objet lorsqu'il reçoit le type de message associé. Une référence sur un objet permet d'appeler la méthode correspondant au message que l'on veut envoyer.

2. (3 points) On vous donne le programme suivant :

```
public class Robot {

    private static int robotCount = 0;

    public static int getRobotCount() {
        return robotCount;
    }

    public Robot(String name,
                 int xPos,
                 int yPos ) {
        super(name, xPos, yPos);

        robotCount++;
    }
}

public class Simulator {

    public static void main(String[] args) {
        Robot robot1 = new Robot("robot_1", 0, 0);
        Robot robot2 = new Robot("robot_2", 5, 6);
        Robot robot3 = new Robot("robot_3", 50, 10);

        System.out.println(robot2.getRobotCount());
    }
}
```

Que signifie le mot clé *static* qui qualifie la variable *robotCount* ? Si on exécute ce programme, qu'est-ce qui s'affichera à la console ? A quoi sert la fonction main ? Pourquoi est-elle déclarée statique ?

(1) Static signifie que la variable appartient à classe et donc que sa valeur est commune à toutes les instances de la classe.

(0,5) 3

(0,5) La fonction main est le point d'entrée du programme.

(1) Puisqu'aucun objet de la classe n'existe au démarrage du programme, la fonction est statique pour que la JVM charge la classe en mémoire, puis appelle la fonction sans avoir à générer au préalable une instance de la classe.

3. (2 points) Un aspect important du paradigme de l'orienté objet est le polymorphisme. Décrivez brièvement de quoi il s'agit et donnez un exemple.

(1,0) C'est lorsqu'un objet peut avoir plusieurs types via l'héritage.

(1,0) Exemple avec les formes vu en cours ou celui-ci :

```java
public abstract class Animal {
    public abstract void makeSound();
}

public class Pig extends Animal {

    @Override
    public void makeSound() {
        System.out.println("groin-groin !");
    }
}

public class Dog extends Animal {

    @Override
    public void makeSound() {
        System.out.println("ouaf ouaf !");
    }
}
```

4. (1 point) Expliquez la différence entre une classe abstraite et une interface.

Une interface ne peut spécifier que des signatures de méthodes (et également des méthodes ayant un corps de type default, mais il n'est pas obligatoire de le mentionner dans la réponse), tandis qu'une classe abstraite peut fournir, en plus de signatures, des corps de ces méthodes ainsi que des attributs.

5. (1 point) Voici les définitions suivantes :

```
public interface Shape {
}

public abstract class AbstractShape implements Shape {
}

public class Circle extends AbstractShape {
}

public class Rectangle extends AbstractShape {
}
```

Parmi les expressions suivantes numérotées de 1 à 5, quelles sont celles qui compilent ?

```
1)  AbstractShape shape = new Circle();

2)  Circle circle = new Shape();

3)  Shape shape1 = new AbstractShape();

4)  Rectangle rect = new Circle();

5)  Shape shape2 = new Rectangle();
```

1 et 5

6. (1 point) Donnez deux raisons pour encapsuler les données. Pourquoi est-ce important lors de la mise en œuvre du MCV ?

(0,5) Deux réponses parmi les suivantes :

- A valider les valeurs des attributs d'un objet.
- A garantir la cohérence des données.
- A rendre transparente la mise en œuvre des classes.
- A garantir la cohérence de l'application.

(0.5) L'encapsulation des données fournies des méthodes de modification des données du modèle dans lesquelles le modèle peut notifier ses observateurs (ou les vues) que ses données ont changées.

8. Exercice de modélisation (10 points) : Programmez des classes servant à modéliser les **expressions arithmétiques** offertes par une calculatrice simple comme celle de la photo suivante :

Cette calculatrice permet de saisir et d'évaluer différentes expressions telles que des constantes, des racines carrées, des fonction inverses, des additions, des soustractions, des multiplications et des divisions.

Chaque expression doit pouvoir être évaluée, ce qui a pour effet de retourner un résultat sous la forme d'un nombre réel. Les expressions doivent également pouvoir être converties en chaîne de caractère, afin de pouvoir être affichées à l'utilisateur, ou encore stockées dans un fichier par exemple.

Les expressions se déclinent en différentes sous-classes d'expressions que vous devez modéliser :

1. Des expressions **constantes**, constituées d'une valeur réelle, servant à modéliser des expressions comme « 112.0 » par exemple. L'évaluation d'une expression constante retourne simplement la valeur de l'expression.
2. Des expressions **unaires**, constituées d'un opérateur unaire, tel que la racine carrée ou la fonction inverse, et d'une sous-expression (*x* par exemple). Cette classe sert à modéliser des expressions telles que la racine carrée de *x* (*√x*), ou la fonction inverse de *x (1/x)*. L'évaluation d'une expression unaire consiste d'abord à évaluer sa sous-expression *x*, puis à appliquer l'opérateur au résultat de cette évaluation.
3. Des expressions **binaires**, constituées d'un opérateur binaire, tel que l'addition, la soustraction, la multiplication ou la division, et de deux sous-expressions (*x* et *y*). Cette classe sert à modéliser des expressions telles que la somme, la soustraction, la multiplication ou la division de *x* et *y*. L'évaluation d'une expression binaire consiste d'abord à évaluer ses deux sous-expressions, puis à appliquer l'opérateur à ces deux résultats.

Pour la suite de l'exercice, concentrez-vous sur les concepts et l'application des bonnes pratiques de modélisation. La syntaxe est secondaire.

8.1 (5 points) On suppose que l'interface Java suivante vous est fournie, telle que requise par l'équipe de développement de l'interface graphique de la calculatrice.

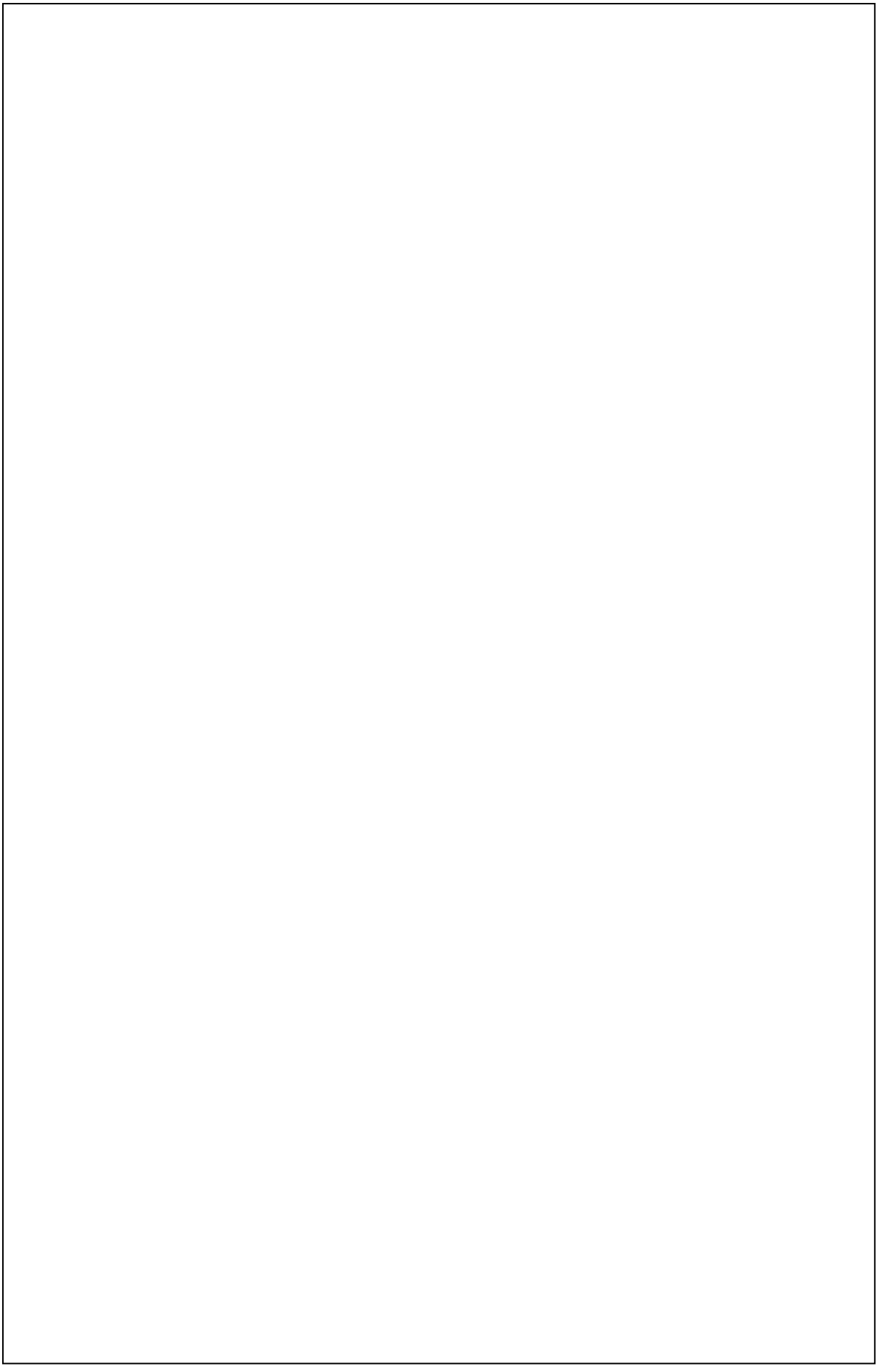
```
package fr.tp.inf112.calculator.expressions;

/**
 * Represents a mathematical expression to be used by the simple calculator software application.
 */
public interface Expression {

    /**
     * Returns the result of evaluating this expression.
     * @return A {@code double} value.
     */
    double evaluate();

    /**
     * Returns a string representation of this expression to be displayed to users or to be stored
     * in a permanent data store such as a file.
     * @return A non {@code null} not empty {@code String} value.
     */
    String getText();
}
```

A partir de cette interface, programmez les classes requises pour modéliser des expressions constantes, ainsi que des expressions de racine carré, d'addition et de multiplication. Vous pouvez utiliser la méthode statique de la classe *Math* (*public static double sqrt(double value)*) pour calculer la racine carrée.

```
public abstract class BasicExpression implements Expression {
}

public class ConstantExpression extends BasicExpression {

    private final double value;

    public ConstantExpression(double value) {
        this.value = value;
    }

    @Override
    public double evaluate() {
        return value;
    }

    @Override
    public String getText() {
        return Double.toString(value);
    }
}
```

```java
public abstract class UnaryExpression extends BasicExpression {

    private final String operator;

    private final Expression expression;

    public UnaryExpression(String operator,
                           Expression expression) {
        super();

        this.operator = operator;
        this.expression = expression;
    }

    public String getOperator() {
        return operator;
    }

    public Expression getExpression() {
        return expression;
    }

    @Override
    public String getText() {
        return getOperator() + "(" + getExpression().getText() + ")";
    }
}

public class SquareRootExpression extends UnaryExpression {

    public SquareRootExpression(Expression expression) {
        super("sqrt", expression);
    }

    @Override
    public double evaluate() {
        final double value = getExpression().evaluate();

        if (value < 0.0) {
            throw new ArithmeticException("Square root of a negative value in expression '" +
                getText() + "'");
        }

        return Math.sqrt(value);
    }
}
```

```
public abstract class BinaryExpression extends BasicExpression {

    private final String operator;

    private final Expression leftExpression;

    private final Expression rightExpression;

    public BinaryExpression(String operator,
                            Expression leftExpression,
                            Expression rightExpression) {
        super();

        this.operator = operator;
        this.leftExpression = leftExpression;
        this.rightExpression = rightExpression;
    }

    public String getOperator() {
        return operator;
    }

    public Expression getLeftExpression() {
        return leftExpression;
    }

    public Expression getRightExpression() {
        return rightExpression;
    }

    @Override
    public String getText() {
        return getLeftExpression().getText() + " " +
            getOperator() + " "  + getRightExpression().getText();
    }
}

public class AdditionExpression extends BinaryExpression {

    public AdditionExpression(Expression leftExpression,
                              Expression rightExpression) {
        super("+", leftExpression, rightExpression);
    }

    @Override
    public double evaluate() {
        return getLeftExpression().evaluate() + getRightExpression().evaluate();
    }

    @Override
    public String getText() {
        return "(" + super.getText() + ")";
    }
}

public class ProductExpression extends BinaryExpression {

    public ProductExpression(Expression leftExpression,
                             Expression rightExpression) {
        super("X", leftExpression, rightExpression);
    }

    @Override
    public double evaluate() {
        return getLeftExpression().evaluate() * getRightExpression().evaluate();
    }
}
```



8.2 (3 points) En supposant que les classes *SubstractionExpression*, *DivisionExpression* et *PowerExpression* sont fournies, complétez le code de création de l'expression suivante :

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Pour les valeurs :

- *a = 3.0*
- *b = 4.0*
- *c = 1.0*

```
final double aValue = 3.0;
final double bValue = 4.0;
final double cValue = 1.0;

// -b
Expression _b = new ConstantExpression(bValue);
Expression __b = new ProductExpression(new ConstantExpression( -1.0 ), _b);

// b^2
Expression _bSquare = new PowerExpression(_b, new ConstantExpression(2.0));
```

```
// 4a
Expression _a = new ConstantExpression(aValue);
Expression _4a = new ProductExpression(new ConstantExpression(4.0), _a);

// 4ac
Expression _c = new ConstantExpression(cValue);
Expression _4ac = new ProductExpression(_4a, _c);

// b^2 - 4ac
Expression _bSquare_4ac = new SubstractionExpression(_bSquare, _4ac);

try {

    // sqrt(b^2 - 4ac)
    Expression _sqrt_bSquare_4ac = new SquareRootExpression(_bSquare_4ac);

    // -b + sqrt(b^2 - 4ac)
    Expression numerator = new AdditionExpression(__b, _sqrt_bSquare_4ac);

    // 2a
    Expression denominator = new ProductExpression(new ConstantExpression(2.0), _a);

    // (-b + sqrt(b^2 - 4ac)) / 2a
    Expression result = new DivisionExpression(numerator, denominator);

    System.out.println(result.getText() + " = " + result.evaluate());
}
catch (ArithmeticException ex) {
    System.out.println(ex.getLocalizedMessage());
}
```

8.3 (1 point) Gestion des exceptions : vérifiez que votre code de la question précédente gère correctement les exceptions arithmétiques pouvant survenir lors de l'évaluation des expressions, par exemple si la valeur de la variable *c* dans la question précédente vaut *2.0* au lieu de *1.0*. Utilisez la classe *ArithmeticException* du JDK, qui hérite de la classe *RuntimeException*, et son constructeur prenant un message de type String en paramètre.

(0,5) pour le code de levage de l'exception dans la classe *SquareRootException*.

```
@Override
public double evaluate() {
    final double value = getExpression().evaluate();

    if ( value < 0.0) {
        throw new ArithmeticException("Square root of a negative value in expression '" +
            getText() + "'");
    }

    return Math.sqrt(value);
}
```

(0,5) pour le code de traitement de l'exception (*try-catch*) dans la question précédente.

FIN DU SUJET