Übung 02: Abstrakte Klassen, Interfaces und dynamische Bindung

Abgabetermin: 19.03.2015, 8:15

Name: _____ **Matrikelnummer**: _____

Informatik: ❑ G1 (Prähofer) ❑ G2 (Prähofer) ❑ G3 (Grimmer) ❑ G4 (Grimmer)

WIN: ❑ G1 (Khalil) ❑ G2 (Kusel) ❑ G3 (Kusel)

Aufgabe	Punkte	abzugeben schriftlich	abzugeben elektronisch	korr.	Punkte
Übung 2	24	Java-Programm, Ausgabe der Tests	Java-Programm	❑	

Expression Tree (24 Points)

Abstract syntax trees can be used to represent arithmetic expressions. Numbers are converted to leaf nodes and operators to nodes with two children. Your task is to create classes for the elements of this tree that can visualize an abstract syntax tree as shown in Figure 1.

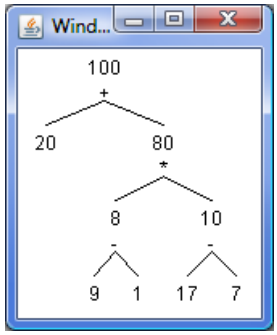

Figure 1: Expression tree generated from the input "20+(9-1)*(17-7)".

We provide Java source code such that you can focus on the essential parts of the implementation. In the package `expression` you can find three predefined public types that you should use:

- `Expression` is an interface that every node in the tree should implement. The interface defines the following methods:
  - **`int`** `evaluate()` – evaluates the expression and returns its value.
  - **`void`** `draw(`**`int`** `x, `**`int`** `y)` – draws the expression subtree at position x/y.
  - **`int`** `getWidth()` – computes the total width of a node. Note that the width of a node must take child nodes into account (see Figure 2).
  - **`int`** `getCenter()` – determines the center coordinate of the expression. For a nice symmetric drawing, a node should always be placed centered above its child nodes (see Figure 2).
- `ExpressionFactory` is an interface, whose implementer is able to create instances of the concrete expression classes. It defines the following methods:
  - `Expression createConstant(`**`int`** `value)` – creates a new expression instance that represents a constant value.
  - `Expression createBinary(`**`char`** `op, Expression left, Expression right)` – creates a new expression instance that represents the specified operation and has `left` and `right` as child expressions.

- `ExpressionParser` allows you to convert a text to an abstract syntax tree using an `ExpressionFactory` object. Look at the source code documentation for an example.

Based on those classes you should define a class hierarchy for the nodes of the abstract syntax tree. You should define concrete classes `ConstantExpression`, `AddExpression`, `SubExpression`, and `MulExpression`. For code reuse you should create an abstract base class for all expressions that take two arguments called `BinaryExpression`. Implement the methods of the interface `Expression` appropriately. Furthermore, define a class named `BaseExpressionFactory` that implements the interface `ExpressionFactory` and creates the instances of the expression classes.

Graphical Output:

Implement method `draw(int x, int y)` of the expression nodes with positioning of the nodes as illustrated in Figure 2.

- A constant is drawn centered in a box. The width of the box is determined by method `getWidth`, the height of the box is given by a constant. Method `getCenter` provides the middle position of the box.
- The width of a binary operation node is determined by the width of its two children (note the recursive definition). The height of the box is again given by a constant. The output for a binary operator node consists of the operator sign and the value computed by method `evaluate`. This output is drawn at the horizontal position determined by the method `getCenter` which is computed as the middle of centers of its child nodes (note again the recursive definition).

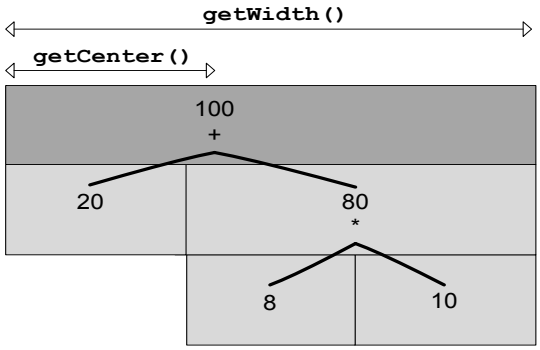

Figure 2: Geometric structure of the tree.

For displaying the graph on the screen you should use the following methods of the class `Window` (for more information on those methods read the comments for them):

- `Window.getTextHeight()` – returns the text height on the screen (see Figure 3).
- `Window.getTextWidth(String)` - returns the width that a certain string needs on the screen (see Figure 3).

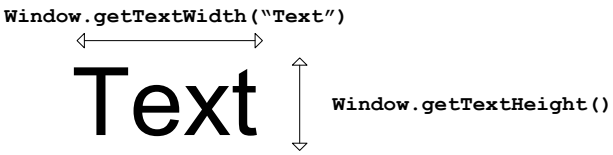

Figure 3: Obtaining the size of a string on the screen.

- `Window.drawLine(int startX, int startY, int endX, int endY)` - draws a black line connecting the points `(startX/startY)` and `(endX/endY)`.
- `Window.drawTextCentered(String text, int x, int y, int width)` – draws a string horizontally centered (see Figure 4).

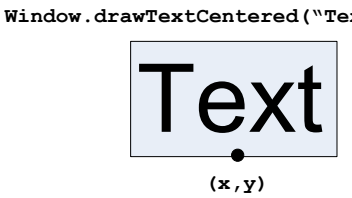

Figure 4: Drawing text horizontally centered.

Testing:

Write several test cases that check the evaluated result of the expression and display the expression tree. Include at least one screen shot in your submission. For example, graphical output for expression "20+(9-1)*(17-7)" should look similar to Figure 1. The main class to generate this example is as follow:

```
package tree.test;

import expression.Expression;
import expression.ExpressionFactory;
import expression.ExpressionParser;
import tree.BaseExpressionFactory;

public class TreeTest {
   public static void main(String[] args) {

      final ExpressionFactory fact = new BaseExpressionFactory();

      final Expression expression =
         new ExpressionParser().createExpression("20+(9-1)*(17-7)", fact);
      expression.draw(0, 0);

   }
}
```