Software and Programming II (SP2) — Lab sheet 9

Recursive Data Structures

2018/19

1. **Part of the solution for this question has already been implemented. You can find the code of this partial implementation on Moodle. Your task is to download and complete this code.**

   In the lecture we saw an example of a class `Employee` that had the employee's line manager (another employee) as an attribute:

```
package employee;

public class Employee {
    private String name; // the Employee's name; non-null
    private Employee lineManager; // the Employee's direct manager; null means none

    public Employee(String name, Employee lineManager) {
        this.name = name;
        this.lineManager = lineManager;
    }
    public Employee(String name) {  // constructor for the CEO
        this(name, null);  // the CEO has no line manager
    }
    public Employee getLineManager() {
        return this.lineManager;
    }

    @Override
    public String toString() {
        return this.name; // omits the line manager
    }
}
```

   We will now work a bit with this recursive data structure. An example for creating objects of the class is given by the following program snippet (Erica is Derek's line manager, Derek is Celia's line manager, and Celia is Barry's line manager):

```
Employee erica = new Employee("Erica");
Employee derek = new Employee("Derek", erica);
Employee celia = new Employee("Celia", derek);
Employee barry = new Employee("Barry", celia);
```

   1. Add a recursive instance method

      `public Employee getCEO()`

      to the class `Employee` that returns the highest-ranked manager reachable from this employee. For example, the code snippet

`System.out.println(barry.getCEO());`

should print:

```
Erica
```

2. Now write a non-recursive instance method

   `public Employee getCEOLoop()`

   (using loops instead of recursion) with the same functionality.

3. We would like to record the "chain of command" from a particular employee in an object of type `ArrayList<Employee>` via a method

   `public ArrayList<Employee> getCommandChain()`

   that uses recursion.

   The first entry in the returned list is the current employee, the second entry is their line manager, etc. So for

   `System.out.println(barry.getCommandChain());`

   we would get the output

   ```
   [Barry, Celia, Derek, Erica]
   ```

   Hint: Do not make `getCommandChain()` itself recursive. Instead, write a recursive *helper method*

   `private void fillCommandChain(ArrayList<Employee> chain)`

   that you call from the method `getCommandChain()` with a suitable argument.

4. Now write a non-recursive instance method

   `public ArrayList<Employee> getCommandChainLoop()`

   (using loops instead of recursion) with the same functionality.

2. **Part of the solution for this question has already been implemented. You can find the code of this partial implementation on Moodle. Your task is to download and complete this code.**

   In this question we are going to combine recursive data structures with inheritance. We will use both concepts together to represent (a simple form of) arithmetic *expressions*.

   (An extended version of the code in this question might be used as part of a compiler for programming languages such as Java or C. The compiler uses a symbolic representation of the Java expressions that it reads and may evaluate expressions where all values are known already at compile time.)

   We define the set of all expressions as the minimal set such that:

   - A constant of type `int` is an expression.
   - If s and t are expressions, then also $(s + t)$ is an expression.
   - If s and t are expressions, then also $(s * t)$ is an expression.

   Now we want to create an inheritance hierarchy to represent such expressions. To this end, we introduce an interface `Expression` with two methods `int computeValue()` and `int numberOfNodes()`. The method `computeValue()` is supposed to compute the value of this expression with the usual meaning of the mathematical operators (here: $+$ and $*$). The method `numberOfNodes()` returns the number of sub-expressions in this expression (i.e., its "size").

   Moreover, we want to override the method `public String toString()` method that we inherit from the class Object. The idea is that the text representation of, for example, an expression $((2 + 3) * 4)$ should be the String `"((2 + 3) * 4)"`. (This has already been implemented.)

We want our interface `Expression` to represent the different kinds of expressions listed above. Thus, for each of the three different kinds of expressions listed above, we introduce a corresponding class that implements the interface Expression.

1. `IntConstant`, a class that stores an `int`.
2. `PlusExpression`, an expression $(s + t)$ that represents symbolically that two expressions are added (the "plus" operator is applied to them).
3. `TimesExpression`, an expression $(s * t)$ that represents symbolically that two expressions are multiplied (the "times" operator is applied on them).

The expressions $(s + t)$ and $(s * t)$ have something in common: they have two "sub-expressions" s and t. Thus, we introduce a common superclass `BinaryExpression` for the classes PlusExpression and TimesExpression. (Here the prefix "Binary" is supposed to indicate that these are expressions with *two* sub-expressions). This will allow us to implement common behaviour (i.e., methods that work in exactly the same way) for the two classes PlusExpression and TimesExpression. Moreover, if we should ever decide to add a class, say, DivExpression to represent the mathematical division operation on two sub-expression (s/t), we would inherit some of its functionality from the abstract superclass "for free".

We will make the class BinaryExpression *abstract* so that we do not have to implement all the methods from the interface Expression in BinaryExpression.

We can represent the information also *graphically* in a *UML class diagram* (more on this in a later week). It tells us what instance variables and methods each class has, and it tells us how classes (and interfaces) are related. In addition to the arrows for the "implements" and "extends" relations that we have already seen, there is also a different kind of arrow with a "diamond" at its tip, going from the interface Expression to the class BinaryExpression, and with the number 2 at its source. It tells us that BinaryExpression has 2 instance variables of type Expression.

The diagram, where we have also mentioned the visibilities for the instance variables (all private) and the methods (all public):