From: **Alan Liang** alan.5100@yahoo.com
Subject: freaking lab10
Date: November 3, 2016 at 4:00 PM
To: Winnie Liang winnieliang27@yahoo.com

AL

lab10 a53

lab10

Due today at 11:59 pm

Part 1

We'll start off this lab by adding onto assignment07's mathematical expressions portion by adding the ability to print out expressions.
For instance, the expression represented by the object `new Add(new Mul(new Num(3), new Num(4)), new Neg(new Num(2)))` woud print "((3 * 4) + neg(2))". We will again use recursion to accomplish this.
Feel free to use your solution to this as a starting point, however, you may also download our solution as a starting point.
Start by editing our Expr abstract class to also have an abstract toString method. This is what we'll use to print expressions and thus we must demand any subclasses to implement this method. Then implement toString for each of the expression subclasses.
Use the following tester, called `ExprTester`, to test your code. See how it expects the toStrings of the expressions to be to know what your toStrings should return. Don't forget to make a new junit test case, not a normal class, when copying/pasting this code.

```
package lab10;

import static org.junit.Assert.*;
import org.junit.Test;

public class ExprTester {

    @Test
    public void simpleNum() {
        Expr e = new Num(4);
        assertEquals("4", e.toString());
    }

    @Test
    public void simpleAdd() {
        Expr e = new Add(
            new Num(5),
            new Num(5)
            );
        assertEquals("(5 + 5)", e.toString());
    }

    @Test
    public void simpleNeg() {
        Expr e1 = new Neg(
            new Num(5)
            );
        assertEquals("neg(5)", e1.toString());
    }

    @Test
    public void complexAdd() {
        Expr e = new Add(
            new Add(
                new Num(1),
                new Add(
                    new Num(10),
                    new Num(1)
                    )
                ),
            new Add(
                new Add(
                    new Add(
                        new Num(3),
                        new Num(3)
                        ),
                    new Num(7)
                    ),
                new Num(1)
                )
            );
        assertEquals("((1 + (10 + 1)) + (((3 + 3) + 7) + 1))", e.toString());
    }

    @Test
    public void complexMixed() {
        Expr a = new Add(
            new Add(
                new Num(1),
                new Add(
                    new Num(10),
                    new Num(1)
                    )
                ),
            new Add(
                new Add(
                    new Add(
                        new Num(3),
                        new Num(3)
                        ),
                    new Num(7)
                    ),
                new Num(1)
                )
            );
        Expr b = new Mul(
            new Add(
                new Num(2),
                new Add(
                    new Num(3),
                    new Mul(
                        new Num(2),
                        new Num(3)
                        )
                    )
                ),
            new Mul(
                new Neg(
                    new Num(0)
                    ),
                new Num(10)
                )
            );
        Expr c = new Mul(
            new Neg(
                new Neg(
                    new Num(1)
                    )
                ),
            new Num(10)
            );
        Expr all = new Add(
            new Add(a,
                    b),
            c
            );
        assertEquals("((((1 + (10 + 1)) + (((3 + 3) + 7) + 1)) + ((2 + (3 + (2 * 3))) * (neg(0) * 10))) + (neg(neg(1)) * 10))", all.toString());
    }
}
```

Part 2

Some words on interfaces

One feature of Java OOP we have yet to see in action is **interfaces**. Like abstract classes, interfaces are a contract among programmers. If you would like to be a subclass (concrete implementer) of `Expr`, then you must implement the methods it demands, namely `eval` and `toString`. In implementing these methods, holding up your end of the deal, any programmer who wishes to use your code knows any subtype of `Expr` offers two methods for him/her to use. Like abstract classes, interfaces cannot be constructed yet they cannot have constructors, they are just a collection of methods and possibly variables.
Expr as an interface would like this:

```
public interface Expr {
    int eval();
    String toString();
}
```

Two differences to note:

- the keywords `abstract class` have been replaced with the `interface` keyword
- the `public` modifier of the methods have been removed - interface methods are `public` by default
    - what are class methods by default?

A class wishing to be part of the Expr interface would implement it like so

```
public class Add implements Expr {
    // ...
    public int eval() { return left.eval() + right.eval(); }
    public String toString() { /* ... implementation ... */ };
}
```

The biggest difference between interfaces and abstract classes is that a class can implement many interfaces, but can only extend one class (be it abstract or not).
Interfaces are typically used to represent ability and specify the *what*, whereas abstract classes also specify the *what* but are also a bit more flexible in what can be provided.
When you make an interface, you are creating a new type. When you have a class implement an interface, that class is a subtype of the interface type and follows the normal subtyping rules.
Thus, the following is still valid and will use dynamic dispatch.

```
Expr e = new Add(/*...*/);
int result = e.eval();
```

Putting interfaces to use

One common interface is `Comparable` which represents the ability to be compared. Thus, a class that implements `Comparable` must implement the methods it requires, which is one: `int compareTo(T other)`. Any class that implements `Comparable` can thus be ordered and can use the pre-existings `Collections.sort` function to sort a collection of instances of that class.
We'll create a Person class that implements this `Comparable` interface.
Create a Person class with this skeleton.

```
public class Person implements Comparable<Person> {

    /**
     * Compares this person to other based on lastname then firstname
     *   (if lastnames are equal, compare firstnames)
     * @param other the other Person to compare this Person to
     * @return  < 0 if this Person is less than other
     *            0 if this Person is equal to other
     *          > 0 if this Person is greater than other
     */
    public int compareTo(Person other) {
        // your code here
    }
}
```

Notice we're implementing the Comparable interface for the Person type. Thus, we have provided a compareTo method for you to fill in. More on that in a bit.
Give Person the following **private** fields:

- String firstname
- String lastname

Have eclipse generate the constructor that accepts these two fields and getters for both.
Override toString to return the lastname, then comma and a space, then firstname. For example, "Smith, Joe".
Back to compareTo. As the javadoc states, compareTo, no matter who implements it for what reason, follows the same convention of return values. For `a.compareTo(b)`, a number less than 0 is returned iff a < b, 0 is returned iff a == b, and a number greater than 0 is returned iff a > b. Note that the symbols '<', '==', and '>' are used loosely here, it is up to the implementing class to define how its instances are ordered. These mathematical symbols don't even make sense to Java outside of numbers.
Our Person class will order its instances based on lastname than firstname. For example, "Zack Johnson" will come before (i.e. is less than) "Alex Smith". "Alex Smith" will come before "Zack Smith" since they have the same lastname. Fortunately for us, the String class implements Comparable so our compareTo can be defined in terms of the compareTo's of lastname and maybe firstname. Implement compareTo now.

Create a Driver. In the main method, create 6 Person objects. Make up names for them to have. Add these people to a list. `List` is an interface while `ArrayList` is an implementation of `List`. `ArrayList` is a subtype of `List` which is why we can write `List<Person> people = new ArrayList<>();`.

Create a method `public static <T extends Comparable<T>> void sortCopy(List<T> list)`. Create a new list, `copy` and send in `list` to its constructor. The `Collections` class offers a static utility method `sort` that sorts Collections of Comparable objects, of which List is one. Because the method knows the collection will hold things that can be compared, it calls the compareTo we wrote to know how to sort the items. Print the copy. Now sort the copy with `Collections.sort(copy)`. Print the copy again.

Back in the main method, call sortCopy with people to verify the contents are now sorted how we defined them to be - by lastname then firstname. Make some of the lastnames the same to verify your compareTo function is working properly.

Create a Student class with the following **private fields**

- Person person
- int GPA

Have eclipse generate a constructor that accepts these 2 fields as well as getters.

Override toString to return the toString of the person, followed by a space and then GPA in parentheses. For example: "Smith, Joe (3)".

Make Student implement the Comparable interface for Students, much like we did for Person. Students will be compared by their Person object they hold. Thus our compareTo can delegate to Person's compareTo method. Implement compareTo.

In your driver, make a list of 6 Students. Use the people you made previously along with a GPA to populate students. In other words, the ith student is made from the ith person with an arbitrary GPA.

Call sortCopy with students to see our compareTos in action.

Its nice to be able to sort by only implementing the Comparable interface but its fairly fixed. We have to pick the one way we want our instances to be compared and if we want that to change, we need to re-implement and re-compile it. What we really want is to be able to specify how to sort them at runtime; that way we can use whatever comparison method we want at that moment.

Comparable has a relative, Comparator, also an interface, which requires one method

```
int compare(T o1, T o2)
```

Where Comparable has `a.compareTo(b)`, Comparator has `compare(a, b)`. If a class would like to implement Comparator, it must implement `compare`.

We'll now write several different ways to compare people and students.

Copy this method into Student

```
public static int compareByGPA(Student a, Student b) {
    return a.GPA - b.GPA;
}
```

Follow this pattern and create a method `compareByNameLength` in Student where instead of comparing a and b using the GPA attribute, they're compared using the length of their person's firstname and lastname. It may be helpful to add a helper method to Person or Student that computes the length of the full name.

Back to the Driver.

Copy this method, which is like the sortCopy we wrote, but also accepts a Comparator to use to sort the elements.

```
public static <T> void sortCopyBy(List<T> list, Comparator<? super T> comp) {
    List<T> copy = new ArrayList<>(list);
    System.out.println(copy);
    Collections.sort(copy, comp);
    System.out.println(copy);
}
```

In the main method, we can sort students with

```
Comparator<Student> byStudents = Student::compareByGPA;
sortCopyBy(students, byStudents);
```

Since aside from its name, compareByGPA has the signature and Comparator's one compare method, compareByGPA is a subtype of the Comparator interface and can thus be used to compare Student's by GPA.

Now sort the students by the length of their full names.

Bonus

Comparator offers a bunch of cool functions that allow us to combine comparators.

1 bonus point if you can sort the Students by descending GPA then ascending length of full name **without writing any new methods**, use the already existings comparators. Since this is bonus for you, the TAs will not offer much assistance.

Add.java Expr.java Mul.java Neg.java Num.java