Chapter 5 Labs

Although there are only two labs, each one has multiple 'checkpoints'. When you reach them, make sure to get checked off before moving on.

Mathematical Expressions

This exercise, although short, demonstrates the flexibility afforded by the generalizability of interfaces. The goal with this lab is to use objects to represent mathematical expressions. These expressions will range from very simple values (e.g., `5`) and binary operations (e.g., `31 / 7`) to arbitrarily complex and nested expressions. To represent this using a type (in this case, 'type' refers to either a class or an interface) hierarchy, we will use a single, root `Expression` interface and several concrete classes (mostly operators) that implement this interface.

Your first task is to create an `Expression` interface with a single `evaluate()` method that takes no arguments and returns a double. This interface, as well as the other classes in this lab, should go in the `math` package.

Next, we'll implement a basic type `Value` to represent a single floating point value. This class should implement the interface, have a `double` instance variable, and use a single-argument constructor to initialize that variable. For the evaluate method, it can simply return the value. This class should also override the `toString()` method of the object class and return the result of `String.valueOf()` called with the instance variable as its argument.

To test this, create an expression in the `MathTest` class using this statement: `Expression expr = new Value(5);`. Next, add a print statement like this: `System.out.println(expr + " = " + expr.evaluate());`. When you run this program, it should print `5.0 = 5.0`.

Now that we've finished implementing a basic `Value` type, we will move on to some basic operations. Create a new `Addition` class that represents the sum of two `Expression`s. The `evaluate()` method should return the sum of the two child expressions and the `toString()` method should return `(<expression 1> + <expression 2>)` with the bracketed expressions replaced accordingly.

After you've finished the `Addition` implementation, create an `Expression` to represent the sum `32 + 7` and a corresponding print statement.

Checkpoint 1

Finally, add the following operation types: `Subtraction`, `Multiplication`, `Division`, `Negation`, and `Exponentiation`, `AbsoluteValue`. Use a caret for `Exponentiation.toString()` (e.g., `<expression 1>^<expression 2>`) and pipes for `AbsoluteValue.toString()` (e.g., `|<expression 1>|`). Use these new types to construct and display the following expressions and their results:

- `(3 * (5 + 2))`
- `(2 / ((9 - 7) * 3))`
- `-(1 / 3^2)`
- `(|-2| + |-3|)`

Checkpoint 2

Priorities

In this lab, you will be creating a program to represent tasks with priorities. Each task is an instance of the `Task` class. Priorities will range from 1-10, 1 being the lowest and 10 being the highest.

1. Design an interface called `Priority` that has two methods: `setPriority()` and `getPriority()`. This interface should also have three static constants: `MIN_PRIORITY`, `MED_PRIORITY`, and `MAX_PRIORITY` set to 1, 5, and 10, respectively.
2. Implement a concrete class `Task` the implements the `Priority` interface. The `Task` class should have all of the following components:
    1. A variable to hold the priority
    2. A variable to hold the name of the task
    3. A constructor that accepts the name of the task as an argument
    4. A method to return the name of the task
    5. A method to set the priority of the task
    6. A method to return the priority of the task
    7. A `toString()` method which returns a `String` with the task name and priority (e.g., `8: Sleep`).
3. Fill in the `main()` method of `TaskTest` with your own personal priorities for the following tasks:
    1. Eat
    2. Sleep
    3. Finish APCS programs
    4. Exercise
    5. Watch TV

    Whenever possible, use the constants in the interface to specify the priority.
4. Print out each task and it's priority.

Checkpoint 3

5. Modify the `Task` class to implement the `Comparable` interface. Use the priorities as the basis for comparison.
6. Write a helper static method in `TaskTest` called `comparePriorities()` which takes two `Task objects` and prints out an appropriate message based on the results of calling `compareTo()`. Then, add the following calls to your main method:

```
comparePriorities(t1, t2);
comparePriorities(t1, t3);
comparePriorities(t1, t4);
```

This should print out something similar to the following after the complete task/priority listing:

```
"Finish APCS programs" has a higher priority than "Eat"
"Finish APCS programs" has a lower priority than "Sleep"
"Finish APCS programs" has an equal priority to "Exercise"
```

Checkpoint 4

7. Using the `int[]`-specific insertion sort `sortInts` from Chapter 6, fill in the body of `sort()` such that it works for any array of `Comparable`s.

8. Create an array of your tasks using an initializer list: `Task[] tasks = {t1, t2, ...};`. Then call the `sort()` method on this list and print out every task-priority pair with their new sorted order.

Checkpoint 5