Homework 4: Calculators and Sets, Revisited

Due Monday, May 22 at 11:30pm

Turn in your homework via the course web page as an updated version of the `hw4.java` text file that I have provided.

Make sure the file can be successfully compiled with `javac`. There should be *no compilation errors or warnings*; if not you get an automatic 0 for the homework!

Recall the CS131 Academic Honesty Policy! You must list whom you discussed the assignment with at the top of your assignment, and also what other resources you used.

This assignment is meant to give you some exposure to the power of the object-oriented paradigm, in particular its separation of interface from implementation: objects communicate only by sending messages to one another, without access to the internal state of other objects. To make this style work, you have to think carefully about your design: the different classes that exist, their interfaces, and their relationships to one another. This assignment is also meant to allow you to concretely compare the object-oriented paradigm to the functional style that you've used in the first half of the course.

As usual, good style matters. **Here are the rules:**

- **Do not modify the names of any types, classes, instance variables, and methods that our code defines or uses in `hw4.java`. We are relying on them for testing purposes; your code will get no credit in cases where the test script can't find what it's looking for.**
- Use interfaces as types and classes as their implementations, rarely if ever using a class directly as a type.
- *Never* use type-unsafe features of Java, like casts and the `instanceof` expression. Similarly, never build your own version of `instanceof`, such as a method that returns `true` if an object has a particular class. If you ever need to figure out the class of some object, then your design is not as object-oriented as it should be.
- *Never* use the `null` value for anything.
- Always compare objects with their `equals` method rather than with the `==` operator.
- You may use any number of helper methods in classes that you require. Make those `protected` or `private` as you see fit.
- If `javac` gives you a warning that looks like this:
  ```
  Note: hw4.java uses unchecked or unsafe operations.
  Note: Recompile with -Xlint:unchecked for details.
  ```
  this likely means that you're missing some type parameters for polymorphic types (e.g., you're using the type `List` instead of something like `List<Instr>`). Recompile with the flag mentioned above to get a useful error message. The code you turn in should not have compiler warnings.

A few other tips:

- You will find the Java API Documentation useful to understand how various data structures from the standard library work.
- Since we are now in an imperative language, you can make use of side effects (e.g., updating a variable's value). Feel free to do this when it feels natural to you, as long as your code works as intended.
- Write comments where useful to tell the reader what's going on.
- Test your code on several inputs, including corner cases -- we will be doing the same.

Now on to the assignment!

Problem #1: Calculators

a. File `hw4.java` declares an interface `Exp` that is the analogue of the `exp` type in OCaml from Homework 2. We have also declared two classes that meet this interface, which correspond to the two cases in the definition of the OCaml `exp` type: `Num` represents a single number of type `double`, and `BinOp` represents a binary arithmetic operation whose operands are arbitrary arithmetic expressions. The binary operator `Op` associated with such an operation is defined via an enum, which in Java is essentially a class with a finite number of instances (in this case, named `ADD`, `SUB`, etc.). The `calculate` method that I've provided should come in handy.

   For starters implement a constructor for the `Num` and `BinOp` classes. For example, the arithmetic expression (1.0 + 2.0) * 3.0 would be represented as

   ```
   new BinOp(new BinOp(new Num(1.0), Op.ADD, new Num(2.0)), Op.MULT, new Num(3.0))
   ```

   Now uncomment the `eval` function in `Exp` and implement this method for `Num` and `BinOp` to evaluate arithmetic expressions. In a binary operation, the operands should be evaluated from left to right.

   Uncomment the first test case in `CalcTest.main` for an example of what you should support, and add more test cases to gain confidence in your code. After compiling your file with `javac` you can execute this test case with the command `java -ea CalcTest`. The `-ea` flag is used to enable asserts (which are disabled by default).

   I've also provided `equals` and `toString` functions for the implementations of `Exp`, which can come in handy for debugging purposes. (Unfortunately there's no easy way to write an `equals` method in Java without using `instanceof` tests and casts! But your code should **never** use either of these.)

b. The interface `Instr` along with the classes `Push` and `Calculate` in `hw4.java` represent the stack instructions that were defined by the OCaml type `instr` in Homework 2 (but we are ignoring the `Swap` instruction this time). For starters implement a constructor for the `Push` and `Calculate` classes. For example, `new Push(1.0)` and `new Calculate(Op.ADD)` should be legal expressions for creating instructions.

   The class `Instrs` represents a list of `Instr` instructions. Uncomment its `execute` method and implement it; it should have the same behavior as the function of the same name from Homework 2. As in that function, you should use an auxiliary stack as storage space for the execution. Java has a Stack class in the standard library that should be useful.

   To maintain good OO style, each instruction should "know" how to execute itself. In other words, you will want to define a method in `Instr` for executing a single instruction. *If implemented properly, you will not need to use `instanceof` tests or casts, and more generally the `Instrs` class will never have to find out what particular instructions are in its instruction list*. See the second test case in `CalcTest` for a simple example of what your code should do.

   Aside from ordinary `for` and `while` loops, Java has a special "for-each" loop for iterating over arrays and other collections, similar to the `for` loop in Python. Check out this page for an example with lists. Use this loop instead of a regular `for` or `while` loop whenever it is natural.

c. Uncomment the method `compile` in `Exp` and implement it; this method should behave like the function of the same name from Homework 2. Uncomment the third test case in `CalcTest` for a simple example of what is expected.

Problem #2: Sets

The interface `Set` in `hw4.java` represents the type of a set of strings, similar to what we saw in class. Uncomment the method declarations in `Set` and complete the class `ListSet`, which implements sets using an underlying linked list that you will also implement.

For efficiency, your implementation should store the elements in sorted order from least to greatest. See the `compareTo` method of `String`, which you can use for this purpose. So `add` should add the new element in the right place in the sorted order (and due to the set semantics, should not add anything if the element is already in the set), and `contains` should optimize its search for an element using the fact that the elements are in sorted order.

Implementing the methods of `ListSet` will involve adding methods to `SNode` and to the two classes `SEmpty` and `SElement` that implement `SNode`. A `ListSet` has a field `head` representing the head node of the list, and that field has type `SNode`. An `SNode` is either an `SEmpty`, representing the end of the list, or an `SElement`, which represents a node containing an element of the set. I've provided a constructor for `ListSet` that simply initializes its `head` to an `SEmpty` object, as well as appropriate constructors for `SEmpty` and `SElement`.

If you set things up correctly, the node methods will be doing most of the work. In particular:

- **You should never use `instanceof` tests or casts, or write your own methods that do something equivalent. For example, the `ListSet` should never need to know whether its `head` is an instance of `SEmpty` or `SElement`.**
- **The `ListSet` should also never need to get access to the internal state of any nodes (e.g., their `elem` or `next` fields). So you should not create getter/setter methods or their equivalents.**
- **Your implementation should not use any loops at all!**
- **You may not use any of the data structures from the standard library in your implementation.**

An interesting design challenge is how to implement `add` as a single pass through the list. Your implementation should be able to achieve this, without requiring the `ListSet` to access implementation details of the nodes. *Hint: The `add` method on nodes should return a new node, representing the current node's replacement in the list after the addition of a new element.*