Homework 5: Calculators and Dictionaries Revisited

Due Thursday, February 27 at 11:30pm

Turn in your homework via the course web page as an updated version of the `hw5.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.

For this assignment we will revisit the calculator and dictionaries we saw on Homework 2, in order to concretely compare the functional style we saw in OCaml with the object-oriented style of Java. 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 `hw5.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.
- Use inheritance instead of duplicating code where possible.
- Always compare objects with their `equals` method rather than with the == operator.
- You may use any number of helper methods that you require. Make those `protected` or `private` as you see fit.

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 functions on several inputs, including corner cases -- we will be doing the same.

Now on to the assignment!

Problem #1: Calculators

a. Recall the OCaml type `aexp` of calculator expressions from Homework 2:

```
type op = Plus | Minus | Times | Divide
type aexp = Num of float | BinOp of aexp * op * aexp
```

File `hw5.java` declares an interface `AExp` as the Java version of this type. Uncomment the `eval` function in `AExp` and define two classes `Num` and `BinOp` that implement `AExp`. The `eval` method should behave just like the `evalAExp` function from Homework 2, except here we're using `doubles` instead of `floats`. I've defined an enum, which in Java is essentially a class with a finite number of instances (in this case, named `PLUS`, `MINUS`, etc.), to represent the `op` type. I've also provided `calculate` methods for these objects, which you should find useful.

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.

b. In Homework 2 we also saw the OCaml type `sopn` of stack operations:

```
type sopn = Push of float | Swap | Calculate of op
```

This type is represented by the `Sopn` interface in `hw5.java`. The class `RPNExp` represents a list of `Sopn` instructions. Uncomment its `eval` method and implement it; the method should behave like `evalRPN` from Homework 2. Implementing the method will require you to declare three classes that implement the `Sopn` interface. Feel free to add whatever methods seem useful to the `Sopn` interface. See the second test case in `CalcTest` for an example of what your code should do. *If implemented properly, you will not need to use `instanceof` tests or casts.* You will find the Java `List` type and its implementations as well as the Java `Stack` class to be useful.

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 `toRPN` in `AExp` and implement it; the method should have the same behavior as `toRPN` in Homework 2. Uncomment the third test case in `CalcTest`. You may find it useful to define a `toString()` method in each class implementing `AExp` for testing purposes. For instance, a reasonable output of the given test case is `[Push 1.0, Push 2.0, Calculate PLUS, Push 3.0, Calculate TIMES]`. (In Java, the + operator for string concatenation is highly overloaded, and in particular an invocation `"hi" + o` for any object `o` is equivalent to `"hi" + o.toString()`. Java lists already have a well-defined `toString()` method that recursively converts each element to a string.)

Problem #2: Dictionaries

The interface `Dict` in `hw5.java` represents the type of a dictionary, parameterized by the types for keys and values. Note that the dictionary is updated in place, as is typical for Java: the `put` method has a `void` return type and simply mutates the data structure rather than returning a new one. The `get` method should throw the declared `NotFoundException` if the given key is not in the dictionary.

a. Recall our OCaml type `dict2` from Homework 2, which implements dictionaries as a linked list of entries:

```
type ('a,'b) dict2 = Empty | Entry of 'a * 'b * ('a,'b) dict2
```

The class `DictImpl2` in `hw5.java` is the analogous implementation for Java. It contains a field of type `Node` which represents the root of a linked list of entries. Right now `DictImpl2`'s constructor and methods are broken; you need to fix them. This will also involve adding methods to `Node` and to the two classes `Empty` and `Entry` that implement `Node`.

Since we are now in an imperative language, the `put` method should actually update the existing entry for a key if one already exists in the dictionary. If you set things up well, you can achieve a `put` by a single pass through the entries, which either updates an existing entry for the given key (if one is found) or else adds the new entry as the very last one in the list of entries. *Hint: It will help you achieve this if the `put` method on a node returns a node.*

If implemented properly, you will not need to use `instanceof` tests or casts. For example, the `DictImpl2` should never need to know whether its `root` is empty or not. Uncomment the first test case in `DictTest` for an example usage.

b. Recall `dict3` from Homework 2, in which we represented a dictionary as a function from keys to values. The class `DictImpl3` in `hw5.java` is the analogous implementation for Java. Java doesn't have first-class functions (though they are coming in the next version of Java!) but they can be simulated using objects. Accordingly, `DictImpl3` contains a field of type `DictFun`, which represents an arbitrary function with argument type `K` and result type `V`. The intent is that this function will return the value associated with a given key in the dictionary, and it will throw the `NotFoundException` if the key is not in the dictionary.

Right now `DictImpl3`'s constructor and methods are broken; you need to fix them. You may find Java's syntax for anonymous classes useful. Uncomment the second test case in `DictTest` for an example usage.

c. `FancyDict` is a subtype of `Dict` that includes a few more operations on dictionaries: `clear` removes all entries from the dictionary, `containsKey` check if a key is in the dictionary, and `putAll` adds all of the given key-value pairs (using the `Pair` class we have defined) to the dictionary. Provide an implementation of `FancyDict` called `FancyDict2Impl`, which implements `FancyDict` using the same `Node` interface and associated classes that you defined in Problem 2a for use in the implementation of `Dict2Impl`. *You should not need to modify the code for any existing interfaces or classes.* Uncomment the third test case in `DictTest` for an example usage.