CS61B, Fall 2011 **HW #4** **P. N. Hilfinger**

Due: Monday, 26 September 2011

Create a directory to hold your answers. There is a skeleton for your solutions in the repository under `$STAFFREPOS/hw4`, and also in the directory `~cs61b/code/hw4`. Use the usual command sequence to copy your final solution to a `hw4-`*N* entry in your tags repository directory.

The purpose of this week's homework is to get you thinking about and working on the first project. Trust me: you do *not* want to wait until the last minute to do this thing!

1. The Project #1 skeleton suggests the use of an abstract class `Expr`, whose subtypes would correspond to specific kinds of drawing expression: `rect`, `sin`, `draw`, etc. Since most expressions have essentially the same syntax, you might expect that there should be common code shared by all commands for use in parsing commands. Indeed, this is strongly suggested by the class Combination, which is the supertype of all Expressions other than symbols and numerals. Each child of `Combination` has access to its operands, and will have a constructor such as

```
class ColorExpr extends Expr {
      Expr (List<Expr> operands) {
          super (operands);
          maybe check to see if operands are valid for me.
      }

      ...
}
```

For this to work, you need to be able to take a `StreamTokenizer` and read `Expr`s from it. In the skeleton file for the class `draw.ExprReader`, we've suggested having a `read` method that does this. In this homework, you'll provide an implementation for a stripped-down version of the project skeleton. You can freely adapt it to your project if you choose.

Actually, we go even further in our suggested structure. As it stands, you can imagine writing pseudo-code for `ExprReader.read` that resembles this:

```
If the next expression is a Combination:
    Read the '(';
    Read the operator symbol into variable op;
    Read the operands into list args;
    Read the ')';
    if (op.equals("color")):
        return new ColorExpr(args);
    else if (op.equals("draw")):
        return new DrawExpr(args);
    etc.
```

For our particular problem this is perhaps not too bad, but just to get you thinking about the possibilities inherent in object-oriented mechanisms, our skeleton has suggested a possible alternative. You can throw it away if you like, but at least you should understand it.

You may have noticed that our `Combination` class has an abstract instance method `create` that takes a list of operands and returns a new `Expr`. The idea is that this method would be implemented in each subtype to simply call its constructor. Now consider the following method:

```
static Expr makeAnExpr(Combination type, List<Expr> operands) {
     return type.create(operands);
}
```

and the call

```
return makeAnExpr(new ColorExpr(null), List<Expr> operands);
```

We a kind of constructor in which the type to construct is represented by a (dummy) object of that type, and is an ordinary parameter of `makeAnExpr`. The sample call above is not very interesting, but now that we've "made types (classes) into objects" we can do some rather interesting things. For example, we can store dummy objects (like `new ColorExpr(null)` above) in a look-up table, so that the big if statement in our pseudocode becomes instead just

```
type = look-up op in table;
return makeAnExpr(type, args);
```

With the right library data structures, that first line of pseudo-code is a single call.

Using these ideas (or whatever subset of them appeals to you), implement the `ExprReader` part of your project. Start with the skeleton `hw4` directory in the staff repository or in `~cs61b/code/hw4`. Fill in the indicated places to make '`java draw.HW4`' work (it should read a command containing only '`+`' and '`*`' operators and numerals from a file and print the result of evaluating it).

In case of boredom. Actually, you *don't* need to write `create` methods for each of the subtypes of `Combination`, nor do you need a special data structure to look up dummy instances. Instead, you can use an advanced feature of Java known as *reflection* to do all of this *without even knowing all the operator names!* See the documentation for `java.lang.Class` (especially the methods `forName` and `getConstructor`) and for `java.lang.reflect.Constructor` (especially the method `newInstance`) if you are curious and see if you can figure out how to do this. No, I do *not* expect everyone to do this; do it only if you need to quench an insatiable curiosity.

2. The project #1 statement indicates that you must provide tests of your implementation. Provide a reasonable draft set of tests (`.drw` files and at least some corresponding `.std` files for the `draw-tests` directory.) Of course, you might reasonably wonder how you will know the precise values of numeric quantities without the exact implementations, but make a reasonable effort. In problem #1, above, we added a `HW4` class for doing a kind of unit testing, producing output the program would not normally provide. How could you instead write `.drw` and `.std` files that test the arithmetic operations by comparing Postscript output without using this special extra testing code?