Problem Set 4
Subtyping and Inheritance

Out: 20 September
Due: **Friday, 6 October**
(beginning of class)

Collaboration Policy. For this problem set, you may either work alone and turn in a problem set with just your name on it, or work with one other student in your section of your choice. If you work with a partner, you and your partner should turn in one assignment with both of your names on it.

Regardless of whether you work alone or with a partner, you are encouraged to discuss this assignment with other students in the class and ask and provide help in useful ways. You may consult any outside resources you wish including books, papers, web sites and people. If you use resources other than the class materials, indicate what you used along with your answer.

Reading: Chapter 7 and Bertrand Meyer's *Static Typing and Other Mysteries of Life*.

Purpose

- Learn to use subtyping in a safe and useful way.
- Learn to design using inheritance.
- Gain some familiarity with graphical user interface programming.
- Add features to an image processing application (and use them to make some pretty pictures)

In the first part of this assignment (questions 1-4), you will do some exercises that develop your understanding of type hierarchies and behavioral subtyping. In the second part (questions 5-11), you will enhance an image processing application using subtyping and inheritance.

Subtyping

Liskov's Chapter 7 and Meyer's *Static Typing and Other Mysteries of Life* describe very different rules for subtypes.

Liskov's substitution principle requires that the subtype specification supports reasoning based on the supertype specification. When we reasoned about a call to a supertype method, we reasoned that if a callsite satisfies the preconditions in the requires clause, it can assume the state after the call satisfies the postconditions in the effects clause. This means the subtype replacement for the method cannot make the precondition stronger since then our reasoning about the callsite may no longer hold (that is, the callsite may not satisfy the stronger precondition). Hence, the type of the return value of the subtype method must be a *subtype* of the type of the return value for the supertype method; the types of the parameters of the subtype method must be *supertypes* of the types of the parameters of the supertype method. This is known as *contravariant* typing.

Bertrand Meyer prefers *covariant* typing: the subtype replacement method parameter types must be *subtypes* of the types of the parameters of the supertype method. We will generalize his rules to apply to preconditions and postconditions also: the subtype method preconditions must be stronger than the supertype method precondition ($\texttt{pre}_{sub}$ => $\texttt{pre}_{super}$) and the subtype postconditions must be stronger than the supertype postconditions ($\texttt{post}_{sub}$ => $\texttt{post}_{super}$). Note that unlike the corresponding Liskov substitution rule, ($\texttt{pre}_{super}$ && $\texttt{post}_{sub}$) => $\texttt{post}_{super}$, there is no need for $\texttt{pre}_{super}$ in the covariant rule since $\texttt{post}_{sub}$ => $\texttt{post}_{super}$.

The signature rule in Java is stricter: subtype methods must have the exact same return and parameter types of the method they override, although they may throw fewer exception types. Java does not place constraints on the behavior of methods, however, since the compiler is not able to check this.

Consider the minimal `Tree` class and its `BinaryTree` subtype, both specified on the next page. The Java compiler will not allow the `getChild` method of `BinaryTree`. Here is the error message:

```
BinaryTree.java:29: getChild(int) in BinaryTree cannot override getChild(int) in Tree;
   attempting to use incompatible return type
      found   : BinaryTree
      required: Tree
```

```
public class Tree
   // OVERVIEW: A Tree is a mutable tree where the nodes are int values.
   //   A typical Tree is < value, [ children ] >
   //      where value is the int value of the root of the tree
   //       and children is a sequence of zero or more Tree objects
   //         that are the children of this tree node.
   //   A Tree may not contain cycles, and may not contain the same
   //   Tree object as a sub-tree in more than one place.

   public Tree (int val)
        // EFFECTS: Creates a tree with value val and no children:
        //             < value, [] >

   public void addChild (Tree t)
        // REQUIRES: t is not contained in this.
        // MODIFIES: this
        // EFFECTS: Adds t to the children of this, as the rightmost child:
        //   this_post = < this_pre.value, children >
        //    where children = [ this_pre.children[0], this_pre.children[1], ...,
        //                  this_pre.children[this_pre.children.length - 1], t]
        //   NOTE: the rep is exposed!

   public Tree getChild (int n)
        // REQUIRES: 0 <= n < children.length
        // EFFECTS: Returns the Tree that is the nth leftmost child
        //   of this.
        //   NOTE: the rep is exposed!


public class BinaryTree extends Tree
   // OVERVIEW: A BinaryTree is a mutable tree where the nodes are int values
   //   and each node has zero, one or two children.
   //
   //   A typical BinaryTree is < value, [ children ] >
   //      where value is the int value of the root of the tree
   //       and children is a sequence of zero, one or two BinaryTree objects
   //         that are the children of this tree node.
   //   A BinaryTree may not contain cycles, and may not contain the same
   //   BinaryTree object as a sub-tree in more than one place.

   public BinaryTree (int val)
        // EFFECTS: Creates a tree with value val and no children:
        //             < value, null, null >

   public void addChild (BinaryTree t)
        // REQUIRES: t is not contained in this and this has zero or one children.
        // MODIFIES: this
        // EFFECTS (same as supertype):
        //   Adds t to the children of this, as the rightmost child:
        //   this_post = < this_pre.value, children >
        //     where children = [this_pre.children[0], this_pre.children[1], ...,
        //                  this_pre.children[this_pre.children.length - 1], t]

   public BinaryTree getChild (int n)
        // REQUIRES: 0 <= n < 2
        // EFFECTS: If this has n children, returns a copy of the BinaryTree
        //   that is the nth leftmost child of this.  Otherwise, returns null.
```

1. (10) Does the `getChild` method in `BinaryTree` satisfy the Liskov substitution principle? Explain why or why not.

2. (10) Does the `addChild` method in `BinaryTree` satisfy the Liskov substitution principle? Explain why or why not.

3. (10) Does the `getChild` method in `BinaryTree` satisfy the Eiffel subtyping rules? Explain why or why not.

4. (10) Does the `addChild` method in `BinaryTree` satisfy the Eiffel subtyping rules? Explain why or why not.

Note that the Java compiler will allow the `addChild` method, but it *overloads* instead of *overrides* the supertype `addChild` method. That is, according to the Java rules the `BinaryTree` class now has two `addChild` methods — one is the `addChild (Tree)` method inherited from `Tree`, and the other is the `addChild (BinaryTree)` method implemented by `BinaryTree`. This can be quite dangerous since the overloaded methods are resolve based on apparent types, not actual types. For example, try this program:

```
static public void main (String args[]) {
  Tree t = new BinaryTree (3);
  BinaryTree bt = new BinaryTree (4);

  t.addChild (bt);                    // Calls the addChild(Tree) method
  bt.addChild (new BinaryTree (5));   // Calls the addChild (BinaryTree) method
  bt.addChild (new Tree (12));        // Calls the addChild (Tree) method
}
```

Note that the first call uses the inherited `addChild(Tree)` because the apparent type of `t` is `Tree`, even though its actual type is `BinaryTree`.

Rhocasa

For the second part of this assignment, you will extend an image processing application (somewhat similar to Picasa, but a bit less sophisticated). Rhocasa does have one feature not provided by Picasa, which is to support filters that can involve multiple images.

Download: ps4.zip — this file contains all the source code for Rhocasa, and some sample images.

Create a new project `ps4` in Eclipse, containing the extracted files from `ps4.zip`. Create a `Run` configuration with the Main class as `ps4.GUI`. To avoid running out of memory, we need to increase the maximum size of the Java VM heap. Do this by selecting the `Arguments` pane, and adding a VM argument (the bottom window in the pane):

```
-Xmx512m -ea
```

The first argument sets the maximum size of the VM heap to 512 megabytes. The second argument turns on assertion checking.

Rhocasa provides a graphical user interface (GUI) for manipulating a set of images by applying filters to generate new images. Every filter must be a subtype of the `Filter` datatype. The filters provided are shown in the class hierarchy below:

```
java.lang.Object
 └ps4.Filter
     └ps4.PointFilter
         └ps4.BrightenFilter
         └ps4.GreyscaleFilter
     └ps4.BlurFilter
     └ps4.FlipFilter
     └ps4.MultiFilter
         └ps4.AddFilter
         └ps4.TileFilter
```

The `Filter` class is a subtype of `java.lang.Object`, the ultimate supertype of all Java object types. The `Filter` class provides methods for examining and manipulating an image including the `filter` abstract method. Since `filter` is an *abstract method*, the `Filter` abstract class provides no implementation of `filter`. To make a useful `Filter` object, we need a subclass of `Filter` that provides an implementation of the `filter` method.

We provide two abstract subtypes of `Filter`:

- `PointFilter` — support filters where the resulting color of each pixel can be determined only from the previous color of that pixel. The `PointFilter` class overrides `filter` with a `final` method (this means subtypes of `PointFilter` cannot override `filter`).
- `MultiFilter` — a filter that is applied to a primary image and one or more secondary images

For details on the provided subtypes, see the specifications in the provided code.

5. (10) Develop a new filter that replaces each pixel in the image with its negative (that is, the red value of the new image is 255 - the red value of the old image, and similarly for blue and green). To work with the provided GUI code, you should add the name of your filter to the `effects` array (initialized at the top of the `GUI` class. The GUI will load the class named ps4.*effect*Filter for the name selected from the menu, so if you name your class `NegativeFilter` in the `ps4` package, you should add `"Negative"` to the `effects` array.

For the next two questions, you are to implement new filters. The behavior of these filters is not specified precisely; you can determine in your implemention a good way to provide the effect described.

6. (10) Develop a new filter that tints an image so the pixels near the top of the image (low row numbers) become gradually more blue.

7. (10) Develop a new filter that produces an image that is the "average" of two or more images.

Behavioral Subtyping

8. (10) How well does the provided `Filter` type hierarchy follow the behavioral subtyping rules? Your answer should consider the `PointFilter` and `MultiFilter` abstract classes and their important methods, as well as the other filter subclasses, explaining whether or not they satisfy the substitution principle.

Adding Parameters

Many of the provided filters would benefit from allowing the user to set additional options. For example, the `BlurFilter` has an integer field `repeats` that controls the number of repetitions of the fliter loop, and the `TileFilter` has an integer field `tileSize` that controls the size of the tile. It would be more useful if filter parameters could be set by the user. Your goal is to modify the design of the Rhocasa implementation to support filters that take a parameter in the cleanest, simplest way possible.

9. (10) Describe at least two substantially different approaches to supporting parameterized filters. You may assume that the filter parameter value is a single integer. For each of your proposed approaches, draw the modified class hierarchy. Discuss the tradeoffs between the different designs — which design involves the most changes to the code? which design is easiest to implement?

10. (10) Modify the application to suppoed the parameterized filters. In addition to modifying the filter classes, you will need to modify the GUI to allow a user to enter the parameter for a parameterized filter. This will involve modifying the `GUIHandler.actionPerformed` method defined in `GUI.java`. Hint: look at how the `MultiFilter` is handled.

For question 10, you will need to create a new dialog box to obtain user input for the parameter value. The Swing tutorial, *How to Make Dialogs* provides some useful documentation and examples for this.

11. (10) Develop a new filter of your choice, and use it (as well as the provided filter) to create an interesting image. Be creative!

Turn-in Checklist: You should turn in your answers to questions 1-10 on paper at the beginning of class on Friday, 6 October. including your code (but not unnecessary printouts of the provided code). Also, submit the image you produced for question 11 as a JPG and a zip file containing all of your code by email to *evans@cs.virginia.edu*. There may be a token prize for the best image created.