3 Designing Methods

3.1 Methods for simple classes and classes with containment

Design the following methods for the classes that represent pets that you have defined during the previous lab.

Follow the Design Recipe!

Do as many of these methods as you need, till you are comfortable with the syntax, with designing the templates, and writing and running the tests.)

1. Method `weighsLessThan` that determines whether the pet weighs less than the given weight limit for flying in the passenger cabin of an airplane. (Each airline has their own limit.)
2. Method `sameOwner` that tells us whether the owner of the pet is the same as the owner of the given pet. Do this for first two variants of the `Pet` class.
3. Method `newWeight` that produces a new `Pet` same as the original one, but with the weight changed to the new weight, as the pet visits the veterinarian and is weighed there.
4. Method `changeOwner` that produces a new `Pet` same as the original one, but with the owner changed to the new owner. Do this for first two variants of the `Pet` class.
5. Method `olderOwner` that determines whether the `Owner` of one `Pet` is older than the `Owner` of another `Pet`. Do this for second variant of the `Pet` class.

3.2 Designing Methods: Unions of Classes

In the previous lab you have designed the class hierarchy that represents the following kinds of pets:

- **cats** where we record whether it is a short-hair cat of a long-hair cat
- **dogs** where we record the breed (e.g. Husky, Labrador, etc., or Mutt — describing an unknown breed)

- **gerbils** where we need to know whether it is a male of female

still keeping track of the name of the animal and of its owner.

1. Design the method `isAcceptable` that determines whether the pet is acceptable for a child that is allergic to long haired cats, gets along only with Labrador and Husky dogs, and should not have a female gerbil pets.
2. Design the method `isOwner` that determines whether this animal's owner has the given name.

3.3 Methods for Self-Referential Data.

1. **Problem: Mobiles**

   An artist is designing *mobiles* that will hang from the ceiling. The simplest *mobile* is just one colored ball hanging on a nylon line of some length. More complex *mobiles* are built by hanging at the end of the nylon line a rigid rod with two other mobiles hanging from from the two ends of the rod. The following drawings illustrate two *mobiles*, a simple one and a complex one.

```
Sample mobiles:
---------------

Simple mobile      |
                   |
                   10
                  blue

Complex mobile         |
                       |
                       |
           ------------+-----
           |                |
     ------+------          |
     |           |          |
     10          |          40
    red          10         green
                blue
```

   Our data definition specialist suggested the following class diagram to define the data that represents *mobiles*:

```
             +---------+
             | IMobile |<---------------+
             +---------+                |
             +---------+                |
                  |                     |
                / \                     |
                ---                     |
                  |                     |
       ---------------------            |
       |                   |            |
+--------------+   +---------------+    |
| Simple       |   | Complex       |    |
+--------------+   +---------------+    |
| int length   |   | int length    |    |
| int weight   |   | int leftside  |    |
| IColor color |   | int rightside |    |
+--------------+   | IMobile left  |----+
                   | IMobile right |----+
                   +---------------+
```

(a) Define the classes and interfaces that implement this data definition in *FunJava*.

(b) Make examples of the two sample *mobiles* shown earlier. Add an additional example of data that represents the following *mobile* (The number of dashes in the rods and lines represents their length):

```
              |
   - - - - - - - - - - - - -
  |                       |
 60                       |
blue           - - - - - - - - - - -
              |                   |
              |                  40
           - - - - - -           red
          |           |
         10           |
        green         5
                     red
```

(c) Design the method `countWeights` that count the number of colored balls hanging on the *mobile*.

(d) Design the method `totalWeight` that computes the total weight of a *mobile*. The weight of the lines and struts is given by their lengths (a strut of length *n* has weight *n*).

(e) Design the method `height` that computes the height of the *mobile*. We would like to hang the *mobile* in a room and want to make sure it will fit in.

Make sure you keep updating the *TEMPLATE* as you go along. (We have already started you on your way.)

B. **Problem: Strings**

For this problem start with the file **Strings.java** that defines a list fo `String`s.

Note: The following method defined for the class `String` may be useful:

```
// does this String come before the given String lexicographically?
// produce value < 0   --- if this String comes before that String
// produce value zero  --- if this String and that String are the same
// produce value > 0   --- if this String comes after that String
int compareTo(String that)
```

(a) Design the method `isSorted` that determines whether the list is sorted in alphabetical order.

Hint: You may need a helper method. You may want remember to the accumulator style functions we have seen in Scheme.

(b) Design the method `merge` that consumes two sorted lists of `String`s and produces a sorted list of `String`s that contains all items in both original lists (including duplicates).

Again, make sure you keep updating the *TEMPLATE* as you go on.

3.4 Using the draw library

Learn how to draw shapes using the *draw* library.

1. Download the program *DrawFace.java*. Run it.

   The program illustrates the use of the `draw` library that allows you to draw shapes on a `Canvas`. The first three lines specify that we will be using three libraries (programs that define classes for us to use). The `colors` library defines a union of six colors (black, white, red, yellow, blue, and green) through the interface `IColor`. The `geometry` library defines a single class `Posn` that has no methods besides the constructor. The `draw` library does the work – allows you to define a `Canvas` of the given size and to draw shapes on the `Canvas`.

Define the class `Picture` that represents a simple picture that will be shown in the `Canvas`. The class only needs to know the current coordinates of some anchor point of the picture (its center, or its top left corner).

Design a picture that consists of at least one of each: a circle, a disk, a rectangle, a line, and a text. Now design the method `draw` in the class `Picture` that draws this picture on the given `Canvas`. Assume the size of the `Canvas` is always 100 by 100.

2. Design the method `moveWithin` that produces a new `Picture` moved by the given `dx` and `dy`, but using a *wrap-around*, i.e, if the picture would disappear to the left, it will re-emerge on the right, etc.

3. Design the method `onKey` that consumes a `String` and produces a new `Picture` moved in the given direction `"up"`, `"down"`, `"left"`, or `"right"` 3 pixels, with the same constraints as in the previous method.

Save the work you have done.