3 Complex Data Definitions; Designing Methods

3.1 Problem: Methods for simple classes and containment

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

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.

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 Problem: 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 Problem: Using the draw library

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

1. Download the program *DrawFace.java*. Run it. You need to include the libraries *draw.jar*, *colors.jar*, and *geometry.jar* in your project, as you have done before for the *funjava.jar* and *tester.jar*.

   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.

3.4 Problem: Strings

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

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

```
// does this String come before that String lexicographically?
// produce value < 0   --- if this String comes before that String
// produce value zero  --- if this 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. *Save the work you have done.*