2 Self Referential Data and Functional Methods

Self-referential Data

2.1 Problem (5.1.1)

In the textbook we have already defined the following objects:

```
Date d1 = new Date(5, 6, 2003);
Date d2 = new Date(6, 6, 2003);
Date d3 = new Date(23, 6, 2003);

Entry e1 = new Entry(d1, 5.3, 27, "Good");
Entry e2 = new Entry(d2, 2.8, 24, "Tired");
Entry e3 = new Entry(d3, 26.2, 150, "Exhausted");

ALog l1 = new MTLog();
ALog l2 = new ConsLog(e1,l1);
ALog l3 = new ConsLog(e2,l2);
ALog l4 = new ConsLog(e3,l3);
```

Translate these two objects of type *ALog*

```
ALog l5 = new ConsLog(e3,l1);
ALog l6 = new ConsLog(e3,l2);
```

into the runner's world of logs. Assume these examples were constructed in the context of the four examples above.

Represent the following runner's log as objects:

1. on June 5, 2004: 15.3 miles in 87 minutes, feeling great;
2. on June 6, 2004: 12.8 miles in 84 minutes, feeling good;
3. on June 23, 2004: 26.2 miles in 250 minutes, feeling dead;
4. on June 28, 2004: 26.2 miles in 150 minutes, good recovery;

Create the object via several definitions.

You will need to copy the class definitions for these classes.∎

```
       +-----------------+
       | AReadingList    |<-----------------+
       +-----------------+                  |
              / \                           |
              ---                           |
               |                            |
               |                            |
     +---------+---------+                  |
     |                   |                  |
+---------+          +-------------------+  |
| MTLoB   |          | ConsLoB           |  |
+---------+          +-------------------+  |
             +-------| Book fst          |  |
             |       | AReadingList rst  |--+
             |       +-------------------+
             v
     +------------------+
     | Book             |
     +------------------+
     | String author    |
     | String title     |
     | int price        |
     | int year         |
     +------------------+
```

Figure 1: A class diagram for reading lists

2.2 Problem (5.1.3)

Consider a revision of the problem 1.5

> Develop a program that assists a bookstore manager with reading lists for local schools. . . .

The diagram in figure 1 represents the data definitions for classes that represent reading lists. Implement the definitions with classes. Create two book lists that contain at least one of the books in problem 1.5 plus one or more of your favorite books.∎

2.3 Problem

You are designing an adventure game for your younger sister. Her task is to traverse a maze, while trying to increase her treasure. She starts with some amount of gold, and as she traverses the maze, she may add to her treasure, or lose some of it (or all). The maze consists of rooms, each occupied by one of three creatures: a monster, an elf, or a wizard. In each room with an elf, the elf will increase the treasure by the amount written in his books. In the rooms with the monster, your sister will lose the amount written in blood on the room's wall. Each room with an elf or with a monster leads to two other rooms, but there is only one path to each room, and you can never visit the same room twice. Finally, the maze (and game) ends in a room with a wizard, who adds some more to the treasure using a secret formula. The game may also end in a room with a monster, if your sister loses all her treasure.

Figure 2 shows an example of a maze. Follow the design recipe to define the class hierarchy that represents the maze.∎

2.4 Problem (6.2.1)

Collect the class definitions in this section and evaluate them in ProfessorJ. Create an instance of *Examples* and inspect the object representation. Instances of *UFOWorld*, *AUP*, *UFO*, and *Shot* should each contain default fields for which the constructor does not consume an argument.∎

2.5 Problem (6.2.2)

Take a look at *w* in figure 3. It is an instance of *UFOWorld* without any shots. Think of it as fresh world that has just been created. Write down a new world like *w* assuming that the UFO in *w* has dropped by 3 pixels, that the AUP has remained at the same place, and that the player has fired one shot. A new shot is located 5 pixels above the AUP right in the middle. The width of an AUP is 20 pixels.∎

```
                           Start
                       10 lbs of gold
                             |
            -----------------------------------
            |                                 |
        Monster                              Elf
        loose 5 lbs                         gain 4 lbs
            |                                 |
     --------------                  --------------------
     |            |                  |                  |
  Wizard         Elf             Monster             Wizard
double your   gain 8 lbs       loose 6 lbs         add 9 lbs
 fortune          |                  |
          ----------------         ------------------
          |              |         |                |
       Wizard         Wizard      Elf            Wizard
      add 10 lbs     add 3 lbs   gain 2 lbs     add 1 lb
                                   |
                          ------------------
                          |                |
                       Wizard           Wizard
                      add 5 lbs     tripple your fortune
```

Figure 2: A maze

2.6 Problem (6.2.3)

Add fields to *AUP* that specify how far an instance moves to the left or right when the player hits an arrow key.

Also add fields to *UFO* that specify how far an instance can drop in one time slice of the game and how far it can move to the left or right during the same time.∎

```
class Examples {
  // an anti-UFO platform placed in the center:
  AUP a = new AUP(100);

  // a UFO placed in the center, near the top of the world
  UFO u = new UFO(new Posn(100,5));

  // a UFO placed in the center, somewhat below u
  UFO u2 = new UFO(new Posn(100,8));

  // a Shot, right after being fired from a
  Shot s = new Shot(new Posn(110,490));

  // another Shot, above s
  Shot s2 = new Shot(new Posn(110,485));

  // an empty list of shots
  AShots le = new MtShots();

  // a list of one shot
  AShots ls = new ConsShots(s,new MtShots());

  // a list of two shots, one above the other
  AShots ls2 = new ConsShots(s2,new ConsShots(s,new MtShots()));

  // a complete world, with an empty list of shots
  UFOWorld w = new UFOWorld(u,a,le);

  // a complete world, with two shots
  UFOWorld w2 = new UFOWorld(u,a,ls2);
}
```

Figure 3: Some Sample Objects in the World of UFOs

Methods for Classes

2.7 Problem (10.2.1)

Remember the class *Image* from exercise 1.6 for creating Web pages. Design the following methods for this class:

1. *isPortrait*, which determines whether the image is taller than wider;
2. *size*, which computes how many pixels the image contains;
3. *isLarger*, which determines whether one image contains more pixels than some other image.

Also draw a complete class diagram.∎

2.8 Problem (10.4.2)

Take a look at this following class:

```
// represent information about an image
class Image {
  int width;
  int height;
  String source;

  Image(int width, int height, String source) {
    this.width = width;
    this.height = height;
    this.source = source;
  }
}
```

Design the method *sizeString* for this class. It produces one of three strings, depending on the number of pixels in the image:

1. "small" for images with 10,000 pixels or fewer;
2. "medium" for images with between 10,001 and 1,000,000 pixels;
3. "large" for images that are even larger than that.

Remember that the number of pixels in an image is determined by the area of the image.∎

2.9 Problem

Design a method that computes how long it will take to download an image at our internet access speed (given in bytes per second).∎

2.10 Problem

Design a method that determines whether you can download an image within the limited time, again, knowing the download speed.∎

2.11 Challenge Problem (10.4.3)

Your physics professor would like to simulate an experiment involving bouncing balls. Design a class that represents a ball that is falling on a 10 x 100 canvas at a rate of *DELTA*. That is, each time the clock ticks, the ball drops by *DELTA* pixels.

When the ball reaches the bottom of the canvas, it bounces, i.e., it reverses course and travels upwards again. The bounce is perfect. This means that when the ball always travels the full distance. As long as it is far enough away from the ground, it drops the full distance. If it is too close, it drops by whatever is left and then travels upwards by the remaining number of pixels. Also, when it travels upwards it travels at the same rate as when it is falling.

Design the method *move*, which simulates one step in the movement of the ball.∎