3 Methods for Containment, Unions, and Self Referential Data

Methods and Containment Arrows

3.1 Problem (11.1.1)

Recall the problem of writing a program that assists a book store manager (see exercise 1.2). Add the following methods to the *Book* class:

1. *currentBook*, which accepts a year and returns true only when the book was published in the given year;
2. *thisAuthor*, which accepts an author and returns true only when the book was writted by the given author;
3. *sameAuthor*, which accepts another book and returns true only when the two books have identical authors.∎

3.2 Problem (11.1.2)

Exercise 1.4 provides the data definition for a weather recording program. Design the following methods for the *WeatherRecord* class:

1. *withinRange*, which determines whether today's *high* and *low* were within the normal range;
2. *rainyDay*, which determines whether the precipitation is higher than some given value;
3. *recordDay*, which determines whether the temperature broke either the high or the low record.
4. *same*, which determines whether two different records represent the same information.∎

Methods and Unions of Classes

3.3 Problem (12.1.1)

Figures 3.1 and 3.2 show a part of a class hierarchy for a family of shapes. Construct examples of the *Square* class, and test cases for each method declared in the abstract class. Next, extend the hierachy to include the *Circle* and *Dot* classes described in the textbook. Be sure to provide examples of data and test cases for these functions too.∎

3.4 Problem (12.1.2)

Draw a class diagram for the classes of problem 3.3.∎

3.5 Problem (12.1.3)

Extend and the class hierarchy in figures 3.1 and 3.2 to include *Rectangle*s. The extension must implement all the abstract methods in *AShape*.∎

3.6 Problem (12.1.4)

Implement an extension of the class hierarchy of problem 3.5 to include a perimeter method.∎

```
// represents an abstract shape
abstract class AShape {
CartPt loc;

// to compute the area of this shape
abstract double area();

// to compute the distance of
// this shape to the origin
abstract double distTo0();

// is the given point is within
// the bounds of this shape
abstract boolean in(CartPt p);

// compute the bounding box
// for this shape
abstract Square bb();
}
```

```
// represents a square (this.loc at top left)
class Square extends AShape {
  int size;

  Square(CartPt loc, int size) {
    ... // omitted
  }

  double area() {
    return this.size * this.size;
  }

  double distTo0() {
    return this.loc.distTo0();
  }

  boolean in(CartPt p){
    return
      this.between(this.loc.y, p.y, this.size)
      &&
      this.between(this.loc.x, p.x, this.size);
  }

  Square bb() {
     return this;
  }

  // is x in the interval [lft,lft+wdth]?
  boolean between(int lft, int x, int wdth) {
    return lft <= x && x <= lft + wdth;
  }
}
```

Figure 1: Classes for geometric shapes with methods (part 1)

```
class CartPt {
  int x;
  int y;

  CartPt(int x, int y) { ... // omitted ... }

  // to compute the distance of this point to the origin
  double distTo0(){
     return Math.sqrt( (this.x * this.x) + (this.y * this.y));
  }

  // compute the distance between this CartPt and p
  double distanceTo(CartPt p){
     return
       Math.sqrt((this.x - p.x) * (this.x - p.x) + (this.y - p.y) * (this.y - p.y));
  }

  // create a new point that is deltaX, deltaY off-set from this point
  CartPt translate(int deltaX, int deltaY) {
    return new CartPt(this.x + deltaX, this.y + deltaY);
  }
}
```

Figure 2: Classes for geometric shapes with methods (part 2)

3.7 Problem (12.4.2)

A software house that is working with a grocery chain receives this problem statement:

> Develop a program that keeps track of the items in the grocery store. For now, assume that the store deals only with ice cream, coffee, and juice. Each of the items is specified by its brand name, weight (grams) and price (cents). Each coffee is also labeled as either regular or decaffeinated. Juice items come in different flavors, and can be packaged as frozen, fresh, bottled, or canned. Each package of ice cream specifies its flavor.

Design the following methods:

1. *unitPrice*, which computes the unit price (cents per gram) of some grocery item;
2. *lowerPrice*, which determines whether the unit price of some grocery item is lower than some given amount;
3. *cheaperThan*, which determines whether one grocery item is cheaper than some other, given in terms of the unit cost. ∎

3.8 Problem

Recall Problem 2.2 that dealt with reading lists. Design to following methods for these classes:

1. *hasOldBooks* which determines whether there are any old books in the list of books (books published before 1950).
2. *allNewBooks* which check if all books in the list are new books.
3. *aCount* which computes how many books in the list were written by a given author. ∎

3.9 Problem

Recall Problem 2.3 that dealt with a maze game. We revise the game a little bit, so that the girl starts the game with no fortune in a room with elf. We also add the requirement that each wizard has a unique name. The wizards are are parameterized by two magic numbers a and b and compute the girl's final fortune as a function of her incoming fortune x, using the formula $ax + b$. As before, the game ends when a monster leaves the girl with no fortune, or when the girl lands in a room containing a wizard.

Each move in the game consists of the following steps:

- The elf, the monster, or the wizard adjusts girl's fortune.
- A check is made whether the game is over.
- If the game is not over, the girl moves to the next room of her choice (either left or right).

First, modify your existing class hierarchy to match the updated design specification.
Second, design and implement the following methods for the class hierarchy:

1. *reward* which determines the girl's new fortune, based on her current fortune and the fact that she is currently in 'this' room.
2. *gameOver* which checks if the game is over - again, it needs to know the girl's current fortune and the room.
3. *countRooms* which counts how many rooms are in the maze.
4. *countElfRooms*, *countMonsterRooms* and *countWizardRooms* which count how many rooms of each kind are there in the maze.