3 Methods for Simple Classes

Practice Problems

Practice problems help you get started, if some of the lab and lecture material is not clear. You are not required to do these problems, but make sure you understand how you would solve them. Solving them on paper is a great preparation for the exams.

Work out as complete programs the following exercises from the textbook. You need not work out all the methods, but make sure you stop only when you see that you really understand the design process.

Problems:

1. Problem 10.3 on page 97
2. Problem 10.4 on page 97
3. Problem 11.2 on page 116
4. Problem 12.1 on page 129
5. Problem 12.4 on page 131

Pair Programming Assignment

3.1 Problem

Start with the file **City.java** from Problem 2.1 from the previous assignment.

Design the following methods for the class that represents one city:

A. the method `sameState` that determines whether a city is in the given state.

B. the method `isSouthOf` that determines whether one city is located South of another city.

C. Design the method `distanceTo` that computes the distance from one city to another. (See the problem 1.1 C) for help with figuring out how to compute the distance.)

D. Design the method `totalDistance` for the class hierarchy `IRoute` that computes the total distance of traveling along this route.

E. Design the method `toPosn` that produces a `Posn` that corresponds to the location of this city in a 100 x 100 Canvas. (Add *import geometry.*;* statement to the beginning of your program.)

F. Design the method `draw` that shows this city as a small disk in the given `Canvas`. Assume the `Canvas` has the size 100 x 100. You may want to also show the name of the city.

Note: These is no way one can test this method. However, include the code that will display at least three cities in a `Canvas`.

3.2 Problem

Start with the file **ExcelCells.java**.

For this problem you will use the classes that represent the values in the cells of a spreadsheet. For each cell we record the row and column where the cell is located, and the data stored in that cell. The data can either be a numerical (integer) value or a formula. Each formula can be one of three possible functions: `+` (representing addition), `mn` (producing the minimum of the two cells), or `*` (computing the product) and involves two other cells in the computation.

A. Make an example of the following spreadsheet segment:

	A	B	C	D	E
1	8	3	4	6	2
2	mn A1 E1	+ B1 C1			* B2 D1
3	* A1 A2	+ B2 B1			mn A3 D1
4		+ B3 B2			mn B4 D1
5		+ B4 B3			* B5 E4

B. Draw on paper this spreadsheet and fill in the values that should show in each cell.

C. Design the method `value` that computes the value of this cell.

D. Design the method `countFun` that computes the number of function applications needed to compute the value of this cell.

E. Design the method `countPlus` that computes the number of `Plus` applications needed to compute the value of this cell.

Make sure you design templates, use helper methods, and follow the containment and inheritance arrows in the diagram.

3.3 Problem

Creative Project

Start working on this, but hand in the complete work with the next assignment. You may hand in the work you have done to get back comments on your preliminary design and implementation.

Design the following methods for this simple version of your game:

A. the method `draw` that will display the world state in the given `Canvas`.

Include in the `Examples` class a *visual test* that shows the initial *world* and a *world* at some point during the game. The lab sample program *DrawFace.java* shows you how to make this happen.

B. Add to your class the following method:

```
// signal the end of the world and display the final message
MyWorld endOfWorld(String message){
return this;
}
```

(do not change anything here, other than the name of your *world* class. Invoke it in the two methods you define below, when the conditions for the ending of the game are satisfied.

C. the method `onKeyEvent` that consumes a `String` and produces a new instance of your *world* in response to the given key. The arrow keys are defined as `"left"`, `"right"`, `"up"`, and `"down"`, the space bar is defined as `"space"`.

Note: If some key event leads to the end of the game, that case should return `this.endOfWorld("end of world message")`.

D. the method `onTick` that produces a new instance of your *world* after one clock tick elapsed.

   Note: If on tick we determine that the game ends, that case should return `this.endOfWorld("end of world message)"`.

Design one method at a time, make sure you follow the *Design Recipe*, and once all the parts are there, you are almost ready to run the game.

Note: I will be more impressed with a well designed simple game than with a game that has all kinds of fancy options, but the code is not readable, methods are jumbled together, there are no tests, and there are no purpose statements.

When you are ready to run the game do the following:

- Change the line that defines you `GameWorld` class to be:

  ```
  class GameWorld extends World{
  ```

  Of course, you will use whatever is the name of your class that defines your world. If you have named it just `World`, you need to change its name to something different.

- Comment out your method the method `endOfWorld`.
- Make the return type for your methods `onKeyEvent` and `onTick` be just `World`.
- Include on your `Examples` class

  ```
  boolean go = this.myInitialWorld.bigBang(200, 300, 0.1);
  ```

  — assuming you have defined `myInitialWorld` in the `Examples` class, want your `Canvas` to be 200 pixels wide and 300 pixels tall, and want the clock tick at every 0.1 second. Of course, you choose your own names, sizes, and the speed.

- Run the program as usual.
- Have fun.