

Designing Methods for Simple Classes

Due: 2/1/2011

Portfolio Problems

Work out the following exercises from the textbook as complete programs. You do not need to work out all the methods, but make sure 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

Read through the `JavaWorld` tutorial, accessible from the Assignment main page, or at:

`http://www.ccs.neu.edu/course/cs2510/Assignments/Assignment3/WorldTutor/tutorial.html`

Download the new JARS (`FunJava-2.jar` and `JavaWorld-2.jar`) and add them to your project's build-path for this assignment. Try out some of the example programs and review the documentation. You don't need to submit the examples, we'll be using `Image`s and `World.display(...)` later in this assignment.

3.1 Problem

Start with your file `Cities.java` from Problem 2.1 in the previous assignment. In your Trac/SVN directory `Assignment-03` save these problems (again) in `Cities.java`.

A. Adjust your data definitions so that you can represent the cities given in the *capitals.txt* file.

   In Java you cannot represent zip-codes, such as `02115`, as numbers (**`int`** or **`double`**) for two reasons. First, with the leading zero removed the number is displayed as `2115`. Second, and more importantly, numbers that start with the character `0` are considered to be in *octal* notation (base-8). The zip code `02115` when translated from base-8 to base-10 becomes:

$$(2 * 512) + (1 * 64) + (1 * 8) + 5 = 1101$$

   To avoid these issues, represent zip-codes as `String`s, (e.g., `"02115"`).

Design the following methods for your class that represents a `City`:

B. the method `isSouthOf` that determines whether this `City` is located South of a given `City`.

C. the method `distanceTo` that computes the distance in *miles* from this `City` to a given `City`. (See the problem *1.1, D* for help with computing the distance.)

D. the method, `toPosn`, that produces a `Posn` that corresponds to the location of this `City` in a `Scene`. Add:

```
import image.*;
import world.*;
```

   statements to the beginning of your file to use the `Scene` and `Posn` classes from the `JavaWorld` library.

   The `Scene` class contains methods called `width()` and `height()`, which can be used to scale the location. `Posn` is a simple class with `x` and `y` integer fields, that can be constructed with an expression like:

```
new Posn(5, 7)
```

E. the method `place` that adds this `City` (as a small `Circle` or `Star`) to a given `Scene` at the correct location (translated latitude and longitude). You may also want to add the name of the `City` using a `Text` image.

   You can test your method(s) using `World.display(...)` in a `testDisplay` (or similarly named) test method in your `Examples` class.

Complete the design of the following interfaces, classes, and methods:

F. Define the interface and classes needed to represent a list of cities. Our convention is to name these `ILoCity`, `MtLoCity`, and `ConsLoCity`.

   Make examples of at least three lists: the empty-list, a list with a single city, and one with at least three cities.

3.2 Problem

The file `Banking.java` contains the definitions of classes the represent bank accounts. In your Trac/SVN directory `Assignment-03` add to the file and save these problems in `Banking.java`.

A. Make examples of the following accounts:

   - A *checking* account for Adam Smith with an ID of 123, a current balance of $150, and a minimum balance of $50.
   - A *savings* account for Betty Jones with an ID 456, a balance of $120, and interest rate of 2.5%.
   - A *certificate of deposit* account for Pat Malloy with an ID of 789 and a balance of $300 that has not yet matured.

B. Design the method `amtAvailable` for these interface and classes that produces the amount the customer can currently withdraw from this `IAccount`.

C. Design the method `moreAvailable` that determines whether this `IAccount` has more available for withdrawal than a given `IAccount`.

D. Design the method `withdraw` that produces a new `IAccount` with the `balance` reduced by the given amount. If the given amount exceeds the available amount, then no money should be withdrawn (i.e., return **`this`**).

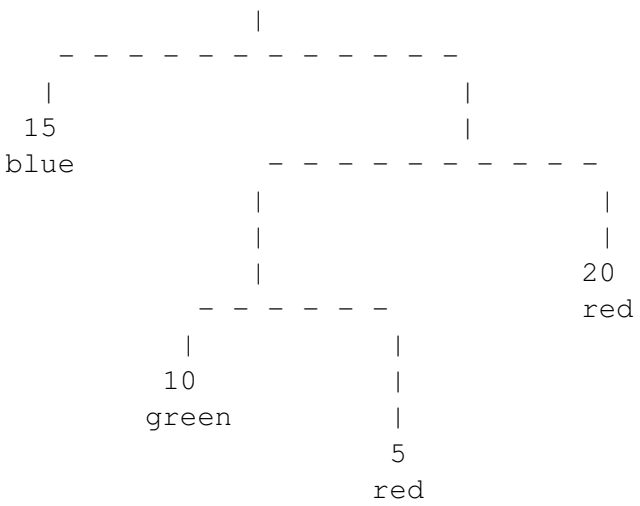
Note: Later we will (*graduate*, and) learn how to signal that the transaction is *invalid*.

3.3 Problem

The file `Mobiles.java` contains the data definitions, examples of data, and the method `countWeights`. Review the file, and in your Trac/SVN directory
`Assignment-03` add to this file and save your problems in
`Mobiles.java`.

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

A. Make an additional Mobile example that represents the following Mobile. The number of dashes in the struts and lines represents their length:



B. Design the method `totalWeight` that computes the total weight of this Mobile. The `length` and strut (`leftside`/`rightside`) weights are given by their lengths (a `rightside` of 5 adds a weight of 5).

C. Design the method `height` that computes the height of this Mobile. Only the `length`s contribute to the height of a Mobile, not the struts.

D. Design the method `isBalanced` that determines whether or not this Mobile is balanced.

   A `Simple` Mobile is always balanced. A `Complex` Mobile is balanced if every sub-mobile is balanced and the weight on the left multiplied by the length of the left strut equals the weight on the right multiplied by the length of the right strut.

E. Design the method `place` that places this Mobile into a given `Scene`. Remember to **`import`** `image.*;` at the top of your file. *Hint*: You'll want accumulate the current `x` and `y` locations to add sub-mobiles to the `Scene` in the right spot. Feel free to represent the `weight` of a `Simple` Mobile by the size of a `Circle` or `Rectangle`. See the documentation for `addLine(...)` and `placeImage(...)` in the `Scene` class.

   Test your method by placing **`import`** `world.*;` at the top of your file and using
   `World.display(...)` on one of your `Complex` examples. Adjust the width/height of the initial `EmptyScene`, `x`, and `y` and scale the lengths/struts to make it look nice.

   Here's my resulting placement of the example above in a 300 x 180 `Scene`, starting at `(120, 20)`.