Abstracting with Function Objects

Due: 2/15/2011 10:00pm

Portfolio Problems

Work out the following exercises from the textbook as complete programs. Make sure you read the surrounding text to get a guidance how to proceed with each.

Complete any problems that require drawing using the `JavaWorld` library and `World.display`, i.e., *not* the mentioned `draw` library/`Canvas`.

1. Problem 19.5 on page 271

2. Problems 19.6 - 19.11 on page 276-279

Pair Programming Assignment

5.1 Problem

Save your solutions to the following problems in a file called `BookLists.java`, in your repository's `Assignment-05/src` directory.

During the lectures we have designed methods for lists that `select` elements by different criteria (e.g., the `author` of a `Book`, or `gpa` of a `Student`).

1. Define the class Book and the classes/interface that represent a list of `Book`s. A Book has a title, an author name, a year of publication, and a price.

2. Define the following interface for classes that compare books:

```
// Represent a method for comparing Books
interface ICompareBooks{

   // Does b1 come before b2 in this ordering?
   public boolean compare(Book b1, Book b2);
}
```

3. Define three classes that implement this interface for ordering Books by: *title length* (class `BookOrderByTitleLength`), *author names* (class `BookOrderByAuthor`), and one more criterion of your choice.

4. Design the method `sort` for your list classes that produces this list in sorted order. Your method should accept an instance of `ICompareBooks` to define the appropriate ordering of the `Book`s.

5. Design the method `isSorted` for the classes that represent lists of Books that uses an instance of `ICompareBooks` to determine whether this list of Books is sorted correctly. Make sure your tests use all three ways of comparing books.

   Note: Remember the one task one method rule. This is just a slight modification of the methods you have already designed.

5.2 Problem

Save your solutions to the following problems in a file called `BookBSTs.java`, in your repository's `Assignment-05/src` directory.

You love *binary-search-trees* (BSTs) right? Here we'll work with binary search trees that represent a collection of `Book`s, which is an abstraction of your typical binary search trees that contain *numbers*.

Here's a class diagram for our classes:

```
             +---------------------+
             | abstract class ABST |<-------+
             +---------------------+        |
      +------| ICompareBooks order |        |
      |      +---------+-----------+        |
      |               / \                   |
      |              +---+                  |
      |                |                    |
      |       +--------+--------+           |
      |       |                 |           |
      |   +------+         +------------+   |
      |   | Leaf |         |   Node     |   |
      |   +------+         +------------+   |
      |                    | Book data  |---|--+
      |                    | ABST left  |---+  |
      |                    | ABST right |--/   |
      |                    +------------+      |
      |                                        V
      V                               +---------------+
+-----------------------------------+ |     Book      |
|           ICompareBooks           | +---------------+
+-----------------------------------+ | String title  |
| boolean compare(Book b1, Book b2) | | String author |
+-----------------------------------+ | int    price  |
                                      +---------------+
```

5.2.1 BST Methods

1. Define the classes that represent a binary search tree of `Book`s as shown above.

2. Design the method `insert` that inserts a given `Book` into this binary search tree using the `ICompareBooks` defined in this tree. Test your method with your classes that implement `ICompareBooks` from before.

3. Design the method `getFirst` that produces the *first* `Book` in the binary search tree (as given by the appropriate `ICompareBooks`). In the `Leaf` class this method should have the following body:

   ```
   throw new RuntimeException("No first of a Leaf");
   ```

   Hint: a helper method that determines if a tree is a `Leaf` (or not) will make this much easier.

4. Design the method `getLast` that produces the *last* Book in the binary search tree (as given by the appropriate `ICompareBooks`). As above, the `Leaf` implementation should have the following body:

   ```
   throw new RuntimeException("No last of a Leaf");
   ```

5. Design the method `getRest` that produces a new binary search tree with the first `Book` removed. In the `Leaf` class this method should have the following body:

   ```
   throw new RuntimeException("No rest of a Leaf");
   ```

   Hint: Start with examples of different `Node`s. We usually call this type of method *"tree surgery"*... can you guess why?

5.2.2 BST Sorting and Equality

1. Design a method `sortBST` in your classes that represent lists of `Book`s that sorts this list using a binary tree. *Hint*: think of how you can use `insert` to create an ordered tree, and `getFirst` and `getRest` to pull the elements out in order.

2. Design a method `sameBook` that determines if this `Book` is the same as the given `Book`.

3. Design a method `sameTree` that determines if this `ABST` is the same (in shape and in content) as the given `ABST`. For the sake of tree comparison, you can ignore the `ICompareBooks` when comparing `ABST`s.

   Hint: use the techniques we discussed in class to break this into seperate, less abstract problems, i.e., special helper methods.

4. Design a method `sameData` that determines if this `ABST` contains the same `Book`s as the given `ABST`, though not necessarily in the same tree shape (i.e., insertion order).

   Hint: remember those `getFirst` and `getRest` methods? Think structural recursion.

5.2.3 BST Integrity

Notice that when we insert into an `ABST` starting with a `Leaf` we always get a tree with the correct ordering, but we can easily hand-construct a tree that does not maintain the correct ordering.

Every `Leaf` is correctly ordered, but a `Node` is only correctly ordered, assuming its `left` and `right` trees are correctly ordered, if (1) all the `Book`s in the `left` and `right` trees belong *before* and *after* its `data` (respectively), and (2) the `order` comparison is the same as that of its `left` and `right` trees.

1. Design/modify the constructor of your `Node` class to confirm that the given `data`, `left`, and `right` have these properties.

   Hint-1: remember those `getFirst` and `getLast` methods. See the lecture notes for what should happen when the condition is *not* met.

   Hint-2: For the purposes of this exercise you can/should use `equals` to compare your `ICompareBooks` for equality (e.g., `a.equals(b)`), assuming you only ever use one instance of a particular implementation of `ICompareBooks` (e.g., you never create two `BookOrderByAuthor` instances to construct a tree).

   Note that this version of `equals` will only work for two *identical* objects (i.e., the same exact instance). Later we will use the *singleton pattern* to enforce that only one instance of a particular class is created.