Compuer Science 62
Assignment 5

Due 6:00pm on Friday, March 5, 2010

This assignment is mostly about stacks, but it will also give you practice with iterators and graphics. When you complete the assignment, you will have a graphics application that simulates a calculator.

Please read Section 10.5 of Bailey's book in preparation for the assignment. The important parts are the definitions of the stack operations. Notice also that our project differs from Bailey's in that our stack contains only integers. The terms `add`, `exch`, and so on are operations on the stack.

Each of the four steps is designed to be straightforward, but each step depends on its predecessors. We strongly suggest that you complete one step before moving on to the next.

1 IntArrayIterator

We will need to iterate, possibly in either direction, across a segment of an array. Write a class named `IntArrayIterator` which implements `Iterator<Integer>` that has the three methods required of an iterator (with `remove` doing nothing, as usual) and one constructor:

```
public IntArrayIterator(int[] data, int start, int stop)
```

The iterator produced by the constructor has two different functionalities; it can iterate up *or* iterate down. Interchanging the values passed to `start` and `stop` changes the direction in which the array is traversed, but it does not change the portion of the array under consideration. The iterator produces values from the array `data` according to the following specifications:

- If `start` $<$ `stop`, the iterator produces values from `data[start]` up through `data[stop-1]`. The values are produced in the order in which they appear in the array.

- If `stop` < `start`, the iterator emits the values from `data[start-1]` *downward* through `data[stop]`. The values are produced in reverse order from their appearance in the array—from high indices to low.
- Obviously, if `start` = `stop`, the iterator produces no values at all.
- The iterator will, without any help from you, generate an error if any of the indices are outside the bounds for the array. You are not required to check those bounds, but your documentation should warn the user. And, later, when you are using this class, you must provide parameters that are within bounds.

2 ArrayStack

The next class is an array-based stack of integers. Write a class `ArrayStack` which implements `Iterable<Integer>`. Work directly with an **array** and indices; do not use any of the list, vector, or stack classes from `structure5` or the Java API. You must have one constructor

```
public ArrayStack(int capacity)
```

which creates an array with `capacity` elements. Besides the array, you will need at least one other instance variable. Implement these standard stack methods, with `assert` statements to guard against pushing onto a full stack, or popping or peeking with an empty one.

```
void clear()
int peek()
int pop()
void push(int value)
boolean isEmpty()
boolean isFull()
int size()
String toString()
Iterator<Integer> iterator()
```

The `size` method returns the number of elements currently in the stack. The `toString` method returns a string that lists the elements in the stack, like these:

```
<ArrayStack:>
<ArrayStack: 0 1 2>
```

The convention for `toString` is that the stack grows to the right. The `iterator` method returns an object of type `IntArrayIterator` that produces all the elements in the stack, running from the base to the top.

Suggestions: Keep it simple. With the *possible* exception of `toString`, the methods will only be two or three lines long. Use the fact that the class is `Iterable` to help with `toString`.

3 PostfixStack

Write a class `PostfixStack` which `extends ArrayStack`. Give it two constructors:

```
public PostfixStack(int capacity)
public PostfixStack()
```

The first will create a stack of the specified capacity, and the second will create a stack with the default capacity of 64.

By extending `ArrayStack`, `PostfixStack` will inherit all the methods from `ArrayStack`. In addition, implement the methods below. Use the public methods available to you via `ArrayStack`. You should NOT be directly modifying the underlying array, etc. of `ArrayStack`.

Each method represents an operation that removes one or two elements from the stack, computes a new value, and pushes that value back onto the stack. The specifications are given in Section 10.5 of Bailey's book. Be sure that you understand, in the cases of subtraction and division, the order of the arguments. Use `assert` statements to guard against empty- or full-stack errors.

```
void dup()
void exch()
void abs()
void neg()
void add()
void sub()
void mul()
void div()
```

Override the `toString` method from `ArrayStack` so that it prints the name `PostfixStack` instead of `ArrayStack`.

Add one additional method of your choice that carries out an operation on the top one or two elements of the stack. Possibilities include square, factorial, and x^y. Give your method a descriptive name.

You are *not* asked to implement the full language simulator described in Section 10.5. Simply provide a class with the above primitives for arithmetic and manipulation of elements on the stack.

For testing, you might write a `main` method to exercise your calculator. If `pc` is a `PostfixStack` object, then the following sequence should produce a stack whose only element is 47. Remember to remove the `main` method when you are finished testing.

```
pc.push(2); pc.dup(); pc.push(5); pc.exch(); pc.dup();
pc.dup(); pc.mul(); pc.mul(); pc.mul(); pc.add();
pc.push(7); pc.push(2); pc.sub(); pc.add();
```

Suggestions: As with `ArrayStack`, the constructors and methods are short and simple. Be sure that `PostfixStack` works correctly before moving on to part 4. It will be very difficult to sort out errors if they occur in the underlying calculator *and* the graphical interface.

4 GraphicsCalculator

For the last part you will complete a graphics shell for your calculator. You can view a sample implementation in the lab by giving the command

```
/common/cs/cs062/assignments/assignment5/bin/calculator
```

in a terminal window. Notice that the calculator uses postfix notation: to multiply 4 by 7 you would press these keys:

4, Enter, 7, Enter, *

There are files `Calculator.java` and `OpButtonListener.java` which do the majority of the work. You can find them in

```
/common/cs/cs062/assignments/assignment5/src/
```

You will need to do the following:

a. Your main task is to add the facility for processing digits. Create a class `DigitButtonListener` be modeled on `OpButtonListener`. The `DigitButtonListener` must have a constructor to match the use in `Calculator`, and it must make a call to the method `addDigit` in `Calculator`.

b. Implement the `addDigit` method in `Calculator`.

c. Add two new buttons between Clear and / which add new facilities from `PostfixStack`. One of them should correspond to the method that you chose to add to `PostfixStack`. The other should correspond to one of the other `PostfixStack` operations, like `dup`, `neg`, or `exch`. Give your buttons descriptive labels.

We must specify precisely the behavior of calculator and its the buttons. First of all, we adopt the convention that the stack is never "too empty." Zeroes appear at the bottom of the stack whenever more elements are needed. Do not change the underlying `PostfixStack`; simply have `Calculator` ensure that the stack always has one zero at the bottom. If an operation like Pop or Clear empties the stack, we push a zero back onto it. When two elements are needed and only one is available, we push another zero and exchange it with the existing element. This convention has already been built into the code for `Calculator`.

Next, the digit buttons operate on the top of the stack. The first time a digit button is pressed, a new value is pushed onto the stack. Subsequent digits change the element on top of the stack, via a pop then a push. The calculator uses the boolean variable `numberInProgress` to remember whether the most recent button is a digit. Pressing a button that is not a digit ends the accumulation of digits and carries out the operation of that button. Therefore, it is not necessary to press Enter after the 7 in the example above.

You will implement the facility for collecting digits in the method `addDigit`. Remember that the way to compute the value of an integer when a new digit is appended on the right is to multiply the original integer by ten and add the value of the digit.

Finally, the display at the top of the window will normally contain the value of the integer on the top of the stack. It must be updated after each change to the stack with a call to `display.setText`. The display will have an error message when there is an attempt to divide by zero.

Submission

Submission. Write your code carefully and clearly. Follow the specifications precisely, because your classes may be tested in ways that you do not suspect. As previously, we will look at your Javadoc documentation.

Submit the final versions of these files, with the names specified, in the usual way.

```
IntArrayIterator.java
ArrayStack.java
PostfixStack.java
Calculator.java
DigitButtonListener.java
```

Grading

You will be graded based on the following criteria:

criterion	points
IntArrayIterator: iterates both up and down appropriately	3
ArrayStack: method functionality (must be array based!)	3
ArrayStack: appropriately handle errors with asserts	1
PostfixStack: method functionality	3
PostfixStack: appropriately handle errors with asserts	1.5
Calculator	3
Extra method/calculator functionality	.5
followed specification	2
appropriate comments (including JavaDoc)	3
style and formatting	2
submitted correctly	1

Notes

Here are some additional notes that are interesting but not directly relevant to the assignment:

The listeners. Because the operations are so fast, we "cheat" a bit on this assignment. Instead of queuing button events to the `Calculator` class, the listener objects execute the code for operations directly. Generally, your listener methods should get in and get out, and not hang around in long methods.