APCS Problem Set 7: More Iterative Statements

Michael Ferraro
mferraro@balstaff.org
Balboa High School
10 November 2015

Contents

1 Introduction

We have worked with iterative statements in Java before, in the form of `while()` and `for()` loops. While you will be taught a few new Java statements for use with iterative procedures, the focus of this problem set is on building programs that solve problems. As with most programming tasks, there will usually be more than one correct way of solving a problem. Our goal is to do so in the best way possible, where your code is easy to understand and does its job in as few steps as possible.[1]

1.1 Objectives

By completing this problem set, students will learn to...

- use `do-while()` loops to guarantee at least one iteration;
- exit loops using `break` and `return`;
- write and use nested loops; and
- write programs to solve problems.

1.2 Due Date

This problem set is due **before 6th Period on 19 November 2015**. Make sure that you are keeping up with the sections you're learning about in class so that you have enough time for the programming tasks.

1.3 A New Workspace

If you haven't already done so in class, create a new workspace folder in your locker called `workspace2`. All projects you create for this problem set must be present in `workspace2`.

2 Recap of `while()` and `for()`

We've already spent quite a bit of time working with `while()` loops, and more recently you learned how to rewrite those loops in a more convenient and concise way using `for()`.

Read the short methods listed below that are provided in the textbook and make sure you understand how each works. Indicate ``yes'' for methods that you've reviewed and understand. If you're unable to choose ``yes,'' ask peers or your teacher questions!

- `sumUpTo()` -- the `while()` version (p185)
  Understand? ☐
- `sumUpTo()` -- the `for()` version (p187)
  Understand? ☐
- `factorial()` (p187)
  Understand? ☐

save progress

3 `do-while()` Loops

Sometimes we want to guarantee that a loop iterates *at least once* before terminating. One example is when we want to present a text-based menu to a user at least once before quitting. If the user chooses to quit, we terminate the loop that shows them the menu options. (Consider the ATM welcome screen example from a prior class.)

As the reading will point out, you never *have* to use the `do-while()` construct, but sometimes it's more convenient.

3.1 Required Reading

→ Litvin & Litvin, *Java Methods A & AB*: §7.4

3.2 Rewriting `while()` using `do-while()`

Download the code for class `DoWhile` from here and import into a new Eclipse project called `ps7a`. Run the class so you are familiar with how it works. Then, make edits so that the class uses `do-while()` rather than `while()`.

Complete this problem on the paper form.

save progress

4 Exiting Loops & `isPrime()`

In Java, there are three ways to get out of a `while()` or `for()` loop:

- the stopping condition is reached
- a `return` statement, which gets out of the current method
- a `break` statement, which gets out of the current loop

The last two methods for exiting a loop, `return` and `break`, are used when running a loop to its conclusion isn't necessary. This may save time since the CPU doesn't need to run as many instructions.

Before you read the two versions of `isPrime()` in Litvin §7.5, consider how you might write a program to find out whether a number is prime. Let's say you'd like to see if 29 is prime. You'd likely start by seeing if 2 divides 29 without a remainder, and then 3, 4, etc., until you reach 29 itself. If you don't find any numbers that evenly divide 29 by that time, then the number is prime.

It turns out that the approach, while simple, is wasteful (i.e., *inefficient*). As the text will tell you, you only ever need to test up to the highest number whose square is less than or equal to the number you're testing for primality. Consider our example of 29. The largest integer whose square is less than 29 is 5, since $5^2 \leq 29$, and 6^2 is too large. Therefore, you need only see if 2, 3, 4, or 5 are factors. If you haven't found a factor by then, you're done checking and can declare that 29 is in fact prime.

The way that Litvin's `isPrime()` methods work is to break out of the loop testing for factors before the loop would conclude naturally via the stopping condition.

4.1 Required Reading

→ Litvin & Litvin, *Java Methods A & AB*: §7.5

save progress

5 Book Questions Involving Loops

For the problems that follow, place the programs you write in your `ps7a` project.

1. Ch. 7, #6, *Population of Mexico*. This is *first* a math problem! **Do not attempt to write a program to solve the problem** until you understand how to solve the problem using a pencil, paper, and calculator.

   Complete this problem on the paper form.

2. Ch. 7, #8, `addOdds()`: Make sure you follow Livtin's requirements: (a) exactly one `for()` loop, (b) no other iterative or `if()-else` statements, and (c) no use of a formula for computing the sum in a single step.

   Complete this problem on the paper form.

3. Ch. 7, #9, printed sum from 1 to `n`: Pay close attention to the formatting of the printed output; you will not get a sign-off if your printed output doesn't match the appearance of the book's example.

   Complete this problem on the paper form.

save progress

6 Nested Loops

So far, the loops we've written have one variable changing its value with each iteration until a stopping condition is reached. Consider the loop below, in which `x` changes each time until the stopping condition is reached:

```
for ( int x = 0 ; x < 10 ; x++ ) {
    System.out.println(x);
}
```

What if we wanted to write iterative code in which *two* values are changing? Solution: *Nested loops*, or loops within loops. The code below will print out all of the integer coordinates present on a 3×3 plane.

```
for ( int x = 0 ; x < 3 ; x++ ) {
    for ( int y = 0 ; y < 3 ; y++ ) {
        System.out.println( "(" + x + ", " + y + ")" );
    }
}
```

1. Write down the printed output that you **predict** will be produced when the nested `for()` loops are run.

   Complete this problem on the paper form.

2. Write a class called `NestedLoops1` in your `ps7a` project. In the class, have a `main()` method that contains

the code above.

3. Was your prediction in # correct? If not, cross out -- using a single line -- what is incorrect in # and write the correct answer next to it. You will **not** be penalized for having had an incorrect prediction!

Complete this problem on the paper form.

6.1 Required Reading

For the reading in Litvin §7.6, you're only responsible for understanding the code that produces a printed triangle of n rows. The *two-dimensional grid traversal* and *finding duplicates in a list* examples will be explained later.

→ Litvin & Litvin, *Java Methods A & AB*: §7.6

6.2 Book Questions Involving Nested Loops

1. **We'll work on this problem together in class:** Ch. 7, #20.

   Knowing where to start with this problem might be difficult. *Advice: Start with simple cases!* In the space below, draw triangles of n rows for each given value of n.

   Complete this problem on the paper form.

   Now you need to see what patterns are present. In particular, you should look at the following for each row of the triangles you sketched:

   - the row #
   - the # of leading spaces (i.e., how # of spaces before *s)
   - the # of *s printed

   For each of the cases in $n = \{1, 2, 3, 4\}$, construct a table like this one:

n	row #	# spaces	# *s
4	1	3	1
4	2	2	3
...	...	...	...

   (Note that you need to construct **4 tables!**) Use the values in your tables to determine *formulae* that will allow you to predict, given n and the current row, how many leading spaces and *s you need to print.

   Complete this problem on the paper form.

   Once you have a working program, demonstrate for a sign-off.

   Complete this problem on the paper form.

2. **Bonus Problem #1:** For extra credit, solve Ch. 7, #25, *Quarters, Dimes, Nickels, and Pennies*.

Complete this problem on the paper form.

3. **Bonus Problem #2:** For extra credit, finish the partially completed `IsosTriangleRec` class provided here so that the `isosTriangle()` method prints an isosceles triangle of the specified height. The helper methods `isosTriangleHelper()` and `printChars()` must be recursive for credit.

   Complete this problem on the paper form.

save progress

7 Perfect Numbers Lab

Litvin §7.8 defines *perfect number*. Read p196 of that section and address the following:

1. What is a *perfect number*?
2. Create a new project called `ps7b`.
3. Write a class `PerfectNumbers`, which has a `main()` that starts like this:

```
public static void main(String[] args) {
    for ( int n = 1 ; true ; n++ ) {


    }
}
```

   The loop above will never terminate! It will allow us to test every number to see if it's perfect until either (a) we terminate the program or (b) the program runs so many iterations that Java virtual machine tries to store a value in `n` that is too large to be an `int`.
4. Write code to reside within the provided `for()` loop that prints every factor of `n` **including** 1, but **excluding** `n` itself. For example, if `n` is 8, print this:

```
8: 1, 2, 4
```

   (Recall that 8 would be considered *perfect* if $1+2+4=8$.)
5. Improve the code so that each time it finds a factor, it adds its value to a running sum. When all factors have been found, test to see if that sum equals `n`, and report `n` as a perfect number when appropriate.

   Complete this problem on the paper form.

Footnotes

... possible.[1]

Note that solving a problem in the least number of steps *and* making the code easy to understand are two goals that don't always agree!

Michael Ferraro 2015-11-06