Fundamental Programming Structures in Java

IFT 194: Lab 2

Brandon Doyle
bdoyle5@asu.edu
1215232174

Dr. Usha Jagannathan
Usha.Jagannathan@asu.edu

July 8, 2018

Pre-Lab Exercises

A. Textbook Sections 5.1–5.3

1. We are tasked with rewriting various conditions in valid Java syntax.

   (a) The condition `x > y > z` may be written in Java as `x > y && y > z`, i.e. we need to join the two comparisons by the ∧–logical operator. This is a result of the type of objects the relational operators act upon; because `x > y` returns a `boolean` type, we receive a compile-time error (invalid types).

   Interestingly enough, this *is* valid Python syntax due its recursive comp_op Grammar definition, so we may (hypothetically) write an inifinite sequence `expr comp_op ... expr comp_op expr`. ∧–logical operators are automatically inserted.

   (b) The statement "x and y are both less than 0" may quite simply be expressed as `x < 0 && y < 0`.

   (c) The statement "neither x nor y are less than 0" may be expressed as `x >= 0 && y >= 0`, or the negation of the previous predicate, i.e. `!(x < 0 && y < 0)`. I think the former is more readable, however.

   (d) The statement "x equals y but not z" may be written as `x == y && x != z`.

2. We are tasked with writing an `if-then` statement to state whether a student has made the Dean's list. Please see Figure 3 for my solution.

3. We are tasked with completing/fixing an example program that computes the raise an employee will receive based on their performance value. Please see Figure 4 for my solution.

Textbook Section 5.4

1. Suppose we have a loop as follows.

```java
package lab_2;

public class SimpleLoop {
    public static void main(String[] args) {
        final int LIMIT = 10;   // immutable
        int count = 1;          // mutable
        while (count <= LIMIT) {
            System.out.print(count + " ");
            count++;
        }
        System.out.println();
    }
}
```

Figure 1: SimpleLoop.java.

View the source of this document on GitHub.

This program outputs the sequence `1..10` when it's executed. Reversing the order of the statements `count++` and `System.out.println(count + " ")` will thus print the sequence `2..11`. This is the case because we are then incrementing each value in `1..10` *prior* to printing.

As a quick comparison, because I think it's awesome how some languages (or paradigms) are better at expressing certain concepts than others, I've written the same program in Haskell with monads in Figure 5. (I think most of the complexity is introduced by immutability.)

2. Next we will consider the following program/loop.

```
package lab_2;

public class TraceLoop
{
    public static void main(String[] args)
    {
        final int LIMIT = 16;
        int count = 1, sum = 0, nextValue = 2;

        while (sum < LIMIT)
        {
            sum += nextValue;
            nextValue += 2;
            count++;
        }
        System.out.println("The sum of your integers is " + sum);
    }
}
```

Figure 2: TraceLoop.java.

When run, the output of this program is the statement "The sum of your integers is 20." Below is a table containing the values of `count`, `sum`, and `nextVal` as this program is executing.

`count`	`sum`	`nextVal`
1	0	2
2	2	4
3	6	6
4	12	8
5	20	10

If we wished instead to add the first *count* nonzero integers, we could modify the `count` variable to start at some positive integer, such as `5`, and the `while`-loop's predicate to instead state `count-- > 0` (while removing the following `count++`), which outputs the following.

`count`	`sum`	`nextVal`
4	0	2
3	2	4
2	6	6
1	12	8
0	20	10
-1	30	12

3. We're tasked next with writing a loop that will print the statement "I love computer science!!" 100 times. See Figure 6 for my solution. Also, this type of loop is count-controlled, and I would much rather have used a `for`-loop.

4. In this question we're asked to write a program that accepts integers from the user and adds them all up. See Figure 7 for my solution.

   Also note that I've used a regular expression to match user input – this loop is *not* "count-controlled," it will exit only if the user decides to terminate the loop. The following is an example input.

   ```
   Enter the next integer: 1
   Intermediate total: 1
   Keep going? [y|n]: y
   Enter the next integer: 13
   Intermediate total: 14
   Keep going? [y|n]: 7
   Total: 14
   Input integers: 2
   *** Exiting
   ```

5. We are asked to write a loop that counts backward from 10 to 1. The code provided has two issues, namely that the `while`-loop's predicate should be `count > 0` (we'd like to stop at 1, not 0), and the variable `count` should be decremented, not incremented, as `count--`. See Figure 8 for my solution.

Conclusion

```
package lab_2;

public class DeansList
{
    public final static double DEANS_LIST_CUTOFF = 3.5;

    /**
     * Determine if a GPA is eligible for the Dean’s list.
     *
     * @param args Ideally contains a single number. If more than one argument is
     *             provided, only the first is taken.
     */
    public static void main(String[] args)
    {
        if (args.length < 1) {
            System.out.println("Please provide your GPA");
            System.exit(0);
        }

        double gpa = 0.0;

        try {
            gpa = Double.parseDouble(args[0]);
        } catch (NumberFormatException e) {
            System.out.println("Please provide a float");
            System.exit(0);
        }

        if (gpa >= DeansList.DEANS_LIST_CUTOFF) {
            System.out.println("Congratulations -- you made the Dean’s list");
        } else {
            System.out.println("Sorry you didn’t make the Dean’s list");
        }
    }
}
```

Figure 3: DeansList.java. I decided to turn this program into a super simple command line utility to test the usage of `args` in the `main` function.

```java
package lab_2;

import java.util.InputMismatchException;
import java.util.Scanner;

public class Salary
{
    /**
     * Compute the salary of a worker based on their performance rating.
     *
     * @param args Not used.
     */
    public static void main(String[] args)
    {
        // ‘try with resources’, since Scanner implements AutoCloseable
        try (var scnr = new Scanner(System.in)) {
            double currentSalary = 0.0, raiseAmount = 0.0;
            int employeeRating = 0;

            while (true) {
                System.out.print("Enter the current salary: ");
                try {
                    currentSalary = scnr.nextDouble();
                } catch (InputMismatchException ex) {
                    System.out.println("*** ERROR: Please enter a float");
                    scnr.next();
                    continue;
                }
                if (currentSalary < 0.0)
                    System.out.println("Please enter a positive float");
                else
                    break;
            }

            while (true) {
                System.out.print("Enter the employee performance rating: ");
                try {
                    employeeRating = scnr.nextInt();
                } catch (InputMismatchException ex) {
                    System.out.println("*** ERROR: Please enter an integer");
                    scnr.next();
                    continue;
                }
                if (employeeRating < 1 || employeeRating > 3)
                    System.out.println("Please enter a number in [1, 2, 3]");
                else
                    break;
            }

            switch (employeeRating) {
                case 1: raiseAmount = (0.06 * currentSalary);
                        break;
                case 2: raiseAmount = (0.04 * currentSalary);
                        break;
                case 3: raiseAmount = (0.015 * currentSalary);
                        break;
            }

            currentSalary += raiseAmount;

            System.out.println("Amount of your raise: $" + raiseAmount);
            System.out.println("Your new salary: $" + currentSalary);
        }
    }
}
```

Figure 4: Salary.java. See also the documentation on AutoCloseable, which provides a nice interface for closing files like Python’s context managers.

```haskell
{- increment.hs -}

import Control.Monad

inc :: Int -> Int
inc = (+ 1)

addSpace :: Show a => a -> String
addSpace el = show el ++ " "

-- Increment prior to printing
priorIncrement :: Int -> Int -> IO ()
priorIncrement start stop = if stop < start then print stop
                            else (mapM_ (putStr . addSpace . inc) [start..stop]) >> putStrLn ""

-- Increment after printing
postIncrement :: Int -> Int -> IO ()
postIncrement start stop = if stop < start then print stop
                           else foldM unit start [start..(stop - 1)] >>= print
  where
    unit :: Int -> Int -> IO Int
    unit i acc = (putStr . addSpace $ acc) >> return (inc i)

main :: IO ()
main = (postIncrement 1 10) >>= (\() -> priorIncrement 1 10)
```

```
$ ghc --make increment.hs
$ ./increment
1 2 3 4 5 6 7 8 9 10
2 3 4 5 6 7 8 9 10 11
```

Figure 5: increment.hs.

```java
package lab_2;

public class CSLoop
{
    /**
     * Print a statement a hundred times.
     *
     * @param args  Not used.
     */
    public static void main(String[] args)
    {
        int i = 100;
        while (i-- > 0)
            System.out.println("I love computer science!! ");
    }
}
```

Figure 6: CSLoop.java.

```java
package lab_2;

import java.util.Scanner;
import java.util.InputMismatchException;

public class Loop
{
    /**
     * Add numbers from user input.
     *
     * TODO: Handle size limit of int type (wrapping).
     *
     * @param args  Not used.
     */
    public static void main(String[] args)
    {
        try (Scanner scnr = new Scanner(System.in)) {
            int sum = 0, nextVal = 0, inputs = 0;
            String keepGoing = "y";

            while (!keepGoing.matches("^[^yY]"))
            {
                System.out.print("Enter the next integer: ");

                try {
                    nextVal = scnr.nextInt();
                } catch (InputMismatchException ex) {
                    System.out.println("*** Error: please enter an integer");
                    scnr.next();
                    continue;
                }

                sum += nextVal;
                inputs++;

                System.out.println("Intermediate total: " + sum);
                System.out.print("Keep going? [y|n]: ");
                keepGoing = scnr.next();
            }

            System.out.println("Total: " + sum);
            System.out.println("Input integers: " + inputs);
            System.out.println("*** Exiting");
        }
    }
}
```

Figure 7: Loop.java.

```
package lab_2;

public class Decrement
{
    /**
     * Print 10..1 to the console.
     *
     * @param args  Not used.
     */
    public static void main(String[] args)
    {
        int count = 11;
        while (count-- > 1)
            System.out.println(count);
    }
}
```

Figure 8: Decrement.java.