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 $\wedge$–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`. $\wedge$–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 2 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 3 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.

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 4. (I think most of the complexity is introduced by immutability.)

Conclusion

View the source of this document on GitHub.

```
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 2: 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 3: 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 4: increment.hs.