SWE 300-
Spring 2020
Exam 2
4/12- 4/13
Time Limit: `Optional.empty()` **Minutes**

Name: Andrew Januszko

This exam contains 6 pages (including this cover page) and 5 questions. Total of points is 43. There are 3 bonus points.

Grade Table (for instructor use only)

Question	Points	Bonus Points	Score
Name on each page	0	3	
Variables	11	0	
Names	8	0	
Table Driven	10	0	
Order, Association	14	0	
Total:	43	3	

1. (3 points (bonus)) Write your name on the header of each page.

2. (_____/*11 points*) Consider the following C code snippet. The scope of a variable is a range (of lines) where the variable can be accessed. For example, the scope of `mole` is line 2 to line 26. A span is the number of lines between two references of a variable. A reference can be a `def` (e.g., being on the left hand side of an assignment, `a := 0`) or a `use` (e.g., the value is accessed, `a := b` where `b` is used). The live time is the number of lines between the first reference and the last reference.

```
1  #include <stdio.h>
2
3  #define UNDER 18.5
4  #define NORMAL_HI 24.9
5  #define OVER_HI 29.9
6
7  int
8  main(void)
9  {
10   int weight, height;
11   double bmi;
12
13   printf("Enter weight in pounds: ");
14   scanf("%d", &weight);
15   printf("Enter height in inches: ");
16   scanf("%d", &height);
17
18   bmi = 703.0 * weight / (height * height);
19
20   if (bmi < UNDER)
21     printf("%.1f Underweight.\n", bmi);
22   else if (bmi <= NORMAL_HI)
23     printf("%.1f Normal.\n", bmi);
24   else if (bmi <= OVER_HI)
25     printf("%.1f Overweight.\n", bmi);
26   else
27     printf("%.1f Obese.\n", bmi);
28
29   return (0);
30 }
```

   (a) (6 points) Complete the following table:

variable	live time (lines)	span (average) (lines)	scope (line range)
weight	9	3	10,30
height	9	3	10,30
bmi	17	1	11,30

   (b) (2 points) For the code above, suggest a change so that it would be easier for programmers to track `bmi`.

   initialize bmi on line 18 with 703.0 * weight / (height * height);

(c) (3 points) Indicate the binding time of variable x for the following scenarios. Use *when code is written*, *at compile time*, or *at runtime*,

i. JAVA:

```
int x = readUserChoice();
```

at runtime

ii. JAVA:

```
int x = 12;
```

when code is written

iii. C:

```
#define MAX 12

int main(void) {
  int x = MAX;
}
```

at compile time

3. (_____/*8 points*) Questions related to the names of variables.

(a) (2 points) According to Clean Code, the author would suggest you change the Java class name `ShipmentAddress` to addr.

(b) (3 points) Choose **3** rows and complete them in the following table.

Operator Symbol	Operation Name	Operand 1 Name	Operand 2 Name	Result Name
÷	division	dividend	divisor	quotient
+	addition	augend	addend	sum
−	subtraction	minuend	subtrahend	difference
×	multiplication	multiplicand	multiplier	product

(c) (3 points) Give an example of using solution domain names other than the visitor pattern. And explain.

inputParseInt states that the code is used to parse input for integers

(d) 3 The convention for (Linux) C code for naming variables is under_scores. Show an example below.

```
#define LITERS_TO_MILLI 1000
```

4. (_____/*10 points*) The amount of college tuition to be paid depends on the student's family income. The follow Java code depicts the relation.

```
//assume income has its value and is between 10,000 and 200,000

int tuition;
if (income < 55000)
```

```
  tuition = 7999;
else if (income < 65000)
  tuition = 8499;
else if (income < 75000)
  tuition = 9899;
else if (income < 85000)
  tuition = 11999;
else if (income < 95000)
  tuition = 13999;
else
  tuition = 16250;
```

(a) (2 points) In general, why does the table driven method have the better performance over the use of branches?

Because it does not have to check each conditional.

(b) (4 points) Rewrite the code snippet using table driven method.

```
int[] tuition = {7999,8499,9899,11999,13999,16250};
int max = Math.max(0, Math.round((income/10000.0)-5));
System.out.print(tuition[(int) Math.min(5, max)]);
```

(c) (4 points) Consider the code in unknown language

```
Select inputCharacter
Case "+", "="
  ProcessMathSymbol( inputCharacter )
Case "0" To "9"
  ProcessDigit( inputCharacter )
Case ",", ".", ":", ";", "!", "?"
  ProcessPunctuation( inputCharacter )
Case " "
  ProcessSpace( inputCharacter )
Case "A" To "Z", "a" To "z"
  ProcessAlpha( inputCharacter )
Case Else
  ProcessError()
```

Rewrite this selection statement in Java or C using table-driven method (indirectly). You can use a table to return a classification and dispatch the method to process the character. The second step does not need to be table driven.

```
char table[][2] = {
        {'+', 'A'},
        {'=', 'A'},
        {'0', 'B'},
        {'1', 'B'},
        {'2', 'B'},
        {'3', 'B'},
        {'4', 'B'},
        {'5', 'B'},
        {'6', 'B'},
        {'7', 'B'},
        {'8', 'B'},
        {'9', 'B'},
        {',', 'C'},
        {'.', 'C'},
        {':', 'C'},
        {';', 'C'},
        {'!', 'C'},
        {'?', 'C'},
        {'a', 'D'},
        {'b', 'D'},
        {'c', 'D'},
        ...
        {'Z', 'D'}
        {' ', 'E'}
    };
```

```
char getClassification(char inputCharacter)
    {
        for (int i = 0; i < 71; ++i)
        {
            if (inputCharacter == table[i][0])
            {
                return table[i][1];
            }
        }
        return 'F';
    }
```

```
switch (classification)
    {
    case 'A':
        ProcessMathSymbol(inputCharacter);
        break;
    case 'B':
        ProcessDigit(inputCharacter);
        break;
    case 'C':
        ProcessPunctuation(inputCharacter);
        break;
    case 'D':
        ProcessAlpha(inputCharacter);
        break;
    case 'E':
        ProcessSpace(inputCharacter);
        break;
    default:
        ProcessError();
        break;
    }
```

5. (_____/*14 points*) Consider the following method `add`.

```
static int add(int a, int b) {
  return b == 0 ? a : add(a+1, b-1);
}
```

(a) (3 points) Demonstrate the evaluation of `add(1,3)`.

```
3 ≠ 0
add(1 + 1, 3 - 1)
2 ≠ 0
add (2 + 1, 2 - 1)
1 ≠ 0
add (3 + 1, 1 - 1)
0 == 0
return 4
```

(b) (3 points) Demonstrate the evaluation of `add(3,1)`.

```
1 ≠ 0
add (3 + 1, 1 - 1)
0 == 0
return 4
```

(c) (2 points) Which parameter, a or b, determines the number of recursive calls? b

(d) (3 points) Which should execute faster, `add(3, add(3, 3))` or `add(add(3, 3), 3)`? Why?

add(add(3,3),3) is faster because it has less recursive calls.

(e) (3 points) Assume f, g are difference lists constructed from three linked lists, l1, l2, l3.

```
var f = toDiff(l1);
var g = toDiff(l2);
```

Let `h = DiffList.append(f,g)`. Demonstrate that the evaluation of `fromDiff(h)` step by step.

```
apply the DiffList function to null
f.apply(null);
apply that to DiffList h
h.f.apply(null)
The resulting list should be null.
```