CSE143 Midterm, Spring 2021

This is a closed-book/closed-note exam. There is a "cheat sheet" at the end. You are not allowed to access the internet or other sources during the exam.

The exam is not, in general, graded on style and you do not need to include comments. For the stack/queue question, however, you are expected to use generics properly and to declare variables using interfaces when possible. The cheat sheet at the end of the exam mentions important restrictions on Stacks and Queues that you must follow.

You are allowed to abbreviate "compiler error" and "runtime error" for the inheritance question (as in "ce" and "re" or "c.e." and "r.e."), but you should otherwise NOT use any abbreviations on the exam.

You are NOT to use any electronic devices while taking the test, including calculators.

Give yourself 60 minutes to complete the exam and then scan it (preferably as a pdf) and upload it to the course web page.

1. Recursive Tracing, 15 points. Consider the following method:

```
public void mystery(int x) {
    System.out.print("*");
    if (x <= 0) {
        System.out.print(".");
    } else if (x % 2 == 0) {
        System.out.print(x % 10);
        mystery(x / 10);
    } else {
        mystery(x / 10);
        System.out.print(x % 10);
    }
}
```

For each call below, indicate what output is produced:

Method Call	Output Produced
mystery(7);	____________________________________
mystery(246);	_____________________________________
mystery(195);	_____________________________________
mystery(1234);	_____________________________________
mystery(29548);	_____________________________________

2. Recursive Programming, 15 points. Write a recursive method called maxDigits that takes two integers as parameters and that returns a new integer that is composed of the larger digit from each number in corresponding positions. For example, the call maxDigits(3509, 6238) would return 6539, the call maxDigits(70519, 89420615) would return 89470619, and the call maxDigits(5555, 372) would return 5575, as indicated below.

```
              3 5 0 9                   7 0 5 1 9              5 5 5 5
              6 2 3 8             8 9 4 2 0 6 1 5                3 7 2
     (max of) -------      (max of) ---------------      (max of) -------
              6 5 3 9             8 9 4 7 0 6 1 9              5 5 7 5
```

Notice that if one number has more digits than the other, then its leading digits are used in the result (you can think of the corresponding digits in the other number as being 0). Your method should throw an IllegalArgumentException if either number passed as a parameter is negative.

You are not allowed to construct any structured objects to solve this problem (no string, array, ArrayList, StringBuilder, Scanner, etc) and you may not use a while loop, for loop or do/while loop to solve this problem; you must use recursion.

3. Details of inheritance, 20 points. Assuming that the following classes have been defined:

```
    public class Couch extends Table {
        public void method1() {
            System.out.println("Couch 1");
        }
    }

    public class Table extends Chair {
        public void method1() {
            System.out.println("Table 1");
        }

        public void method2() {
            System.out.println("Table 2");
            super.method2();
        }

        public void method3() {
            System.out.println("Table 3");
            method1();
        }
    }

    public class Chair {
        public void method2() {
            System.out.println("Chair 2");
        }
    }

    public class Lamp extends Chair {
        public void method1() {
            System.out.println("Lamp 1");
        }

        public void method2() {
            System.out.println("Lamp 2");
        }
    }
```

And assuming the following variables have been defined:

```
        Table var1 = new Table();
        Chair var2 = new Table();
        Table var3 = new Couch();
        Chair var4 = new Lamp();
        Chair var5 = new Couch();
        Object var6 = new Chair();
```

In the table below, indicate in the right-hand column the output produced by the statement in the left-hand column. If the statement produces more than one line of output, indicate the line breaks with slashes as in "a/b/c" to indicate three lines of output with "a" followed by "b" followed by "c". If the statement causes an error, fill in the right-hand column with either the phrase "compiler error" or "runtime error" to indicate when the error would be detected.

```
        Statement                       Output
        ------------------------------------------------------------

        var1.method2();                 ____________________________

        var2.method2();                 ____________________________

        var3.method2();                 ____________________________

        var4.method2();                 ____________________________

        var5.method2();                 ____________________________

        var6.method2();                 ____________________________

        var1.method1();                 ____________________________

        var2.method1();                 ____________________________

        var3.method1();                 ____________________________

        var4.method1();                 ____________________________

        var1.method3();                 ____________________________

        var2.method3();                 ____________________________

        var3.method3();                 ____________________________

        ((Lamp)var4).method1();         ____________________________

        ((Lamp)var2).method1();         ____________________________

        ((Table)var5).method1();        ____________________________

        ((Couch)var1).method1();        ____________________________

        ((Chair)var6).method2();        ____________________________

        ((Couch)var5).method3();        ____________________________

        ((Table)var5).method3();        ____________________________
```

4. Linked Lists, 15 points. Fill in the "code" column in the following table providing a solution that will turn the "before" picture into the "after" picture by modifying links between the nodes shown. You are not allowed to change any existing node's data field value and you are not allowed to construct any new nodes, but you are allowed to declare and use variables of type ListNode (often called "temp" variables). You are limited to at most two variables of type ListNode for each of the four subproblems below.

   You are writing code for the ListNode class discussed in lecture:

```
     public class ListNode {
         public int data;       // data stored in this node
         public ListNode next;  // link to next node in the list

         <constructors>
     }
```

   As in the lecture examples, all lists are terminated by null and the variables p and q have the value null when they do not point to anything.

before	after	code
p->[1] q->[2]->[3]	p->[1]->[3] q->[2]	
p->[1]->[2] q->[3]	p->[2] q->[1]->[3]	
p->[1]->[2]->[3] q->[4]	p->[4]->[3] q->[2]->[1]	
p->[1]->[2] q->[3]->[4]->[5]	p->[2]->[3] q->[5]->[4]->[1]	

5. Array Programming, 10 points. Write a method called sublist that takes as parameters a "start" index (inclusive) and a "stop" index (exclusive) and that constructs and returns a new ArrayIntList of values that contains the sequence of values at those indexes in a list of integers. For example, suppose that an ArrayIntList called list stores the following:

```
[5, 4, 3, 7, 18, 24, 0, -4, 12, 15, 9]
```

If we make the following call on the method:

```
ArrayIntList result = list.sublist(3, 8);
```

After the call, result should store the following values:

```
[7, 18, 24, 0, -4]
```

Notice that the new list contains values stored at indexes 3 through 7 of the original list. This is similar to the substring method where the second parameter is always one higher than the highest index you want to include. Your method should not change the original list of values.

You are writing a method for the ArrayIntList class discussed in lecture:

```
public class ArrayIntList {
    private int[] elementData; // list of integers
    private int size;          // current # of elements in the list

    <methods>
}
```

You may assume that the indexes passed to the method will be legal. Start and stop will both be greater than or equal to 0, start will be less than or equal to stop, and stop will be less than or equal to the size of the list. You may use the zero-argument constructor for ArrayIntList and you may assume that it will construct an array of sufficient capacity to store the result. If start and stop are equal, you should return an empty list.

You may call the ArrayIntList constructor, but otherwise you may not call any other methods of the ArrayIntList class to solve this problem. You are not allowed to define any auxiliary data structures other than the new ArrayIntList you are constructing (no array, String, ArrayList, etc). Your solution must run in O(n) time where n is the length of the original list.

6. Stacks/Queues, 25 points. Write a method called alternatingReverse that takes a stack of integers as a parameter and that rearranges the values so that every other value starting from the bottom of the stack is reversed in order. For example, if a variable s stores these values:

```
     bottom [1, 2, 3, 4, 5, 6, 7, 8] top
             ^     ^     ^     ^
             |     |     |     |
             +-----+-----+-----+
             sequence to reverse
```

Starting from the bottom of the stack and looking at every other value, we find the sequence of numbers 1, 3, 5, 7. This sequence should be reversed while the other values should stay in the same positions. If we make the following call:

```
     alternatingReverse(s);
```

the stack should store the following values after the call:

```
     bottom [7, 2, 5, 4, 3, 6, 1, 8] top
             ^     ^     ^     ^
             |     |     |     |
             +-----+-----+-----+
              reversed sequence
```

This example uses sequential integers to make it easier to see the sequence, but you should not assume anything about the sequence. For example, if s instead stored this sequence:

```
     bottom [7, 1, 4, 18, 23, 0, -5, 12] top
```

then after the method is called, it would store this sequence:

```
     bottom [-5, 1, 23, 18, 4, 0, 7, 12] top
```

Your method should throw an IllegalArgumentException if the number of elements in the stack is not an even number.

You are to use one queue as auxiliary storage to solve this problem. You may not use any other auxiliary data structures to solve this problem, although you can have as many simple variables as you like. You also may not solve the problem recursively. Your solution must run in O(n) time where n is the size of the stack. Use the Stack and Queue structures described in the cheat sheet and obey the restrictions described there (recall that you can't use the peek method or a foreach loop or iterator).

Space is provided on the next page for your answer.

Please write your answer to alternatingReverse below.

CSE143 Cheat Sheet

`Math` Methods (3.2) *mathematical operations*

`Math.abs(`*value*`)`	absolute value
`Math.min(`*v1, v2*`)`	smaller of two values
`Math.max(`*v1, v2*`)`	larger of two values
`Math.round(`*value*`)`	nearest whole number
`Math.pow(`*b, e*`)`	b to the e power

`Stacks and Queues (14.2)` ***(LIFO and FIFO structures)***

Queues should be constructed using the Queue<E> interface and the LinkedList<E> implementation (you may not pass any arguments to the constructor). For example, to construct a queue of String values, you would say:

```
Queue<String> q = new LinkedList<>();
```

Stacks should be constructed using the Stack<E> class (there is no interface):

```
Stack<Integer> s = new Stack<>();
```

For Stack<E>, you are limited to the following operations (no iterator or foreach loop):

`push(`**value**`)`	pushes the given value onto the top of the stack
`pop()`	removes and returns the top of the stack
`isEmpty()`	returns `true` if this stack is empty
`size()`	returns the number of elements in the stack

For Queue<E>, you are limited to the following operations (no iterator or foreach loop):

`add(`**value**`)`	adds the given value at the end of the queue
`remove()`	removes and returns the front of the queue
`isEmpty()`	returns `true` if this queue is empty
`size()`	returns the number of elements in the queue

`String` Methods (3.3) ***(An object for storing a sequence of characters)***

`length()`	returns the number of characters in the string
`charAt(`**index**`)`	returns the character at a specific index
`equals(`**other**`)`	returns true if this string equals the other
`toUpperCase()`	returns a new string with all uppercase letters
`toLowerCase()`	returns a new string with all lowercase letters
`startsWith(`**other**`)`	returns true if this string starts with the given text
`substring(`**start, stop**`)`	returns a new string composed of characters from start index (inclusive) to stop index (exclusive)
`substring(`**start**`)`	returns a new string composed of characters from start index (inclusive) to the end of the string