**PROGRAMMING PATTERNS**

420-301-VA

#### **LAB-3**

### Instructions

* Labs’ answers must be demoed to the teacher in class.
* Labs are individual works.
* You are supposed to work on this LAB during the session hours otherwise you risk falling behind.
* First, try to solve each problem without assistance except asking clarifying questions, preferably from the teacher.
* Do not look at solutions or get assistance from colleagues, online resources, or AI before spending considerable time trying to do the work from scratch yourself, some problems may require hours to solve, and you may need to look away work on another assignment and get back to the problem with a fresh eye that is normal and differs from individual to individual.
* The lab work is based on the theory sessions and a direct implementation of the concepts. Make sure you understand the concepts and can do the examples, or follow the logic, discussed in class and identify what the lab is asking you to do relative to the discussed concepts.
* Feel free to ask more questions to the teacher or look at different examples that are not the solutions to the lab from any other resource.
* Note that the due date is different for each section.
* Note that if you could not work on a lab by the deadline, you should still work on it, submit it and discuss the case with the teacher.

## Array exercise

1. Write a program with a method called “swap”, given an array as an argument the swap method swaps the values of each two adjacent elements starting from the first element until the last element in the array, and does only one sweep or pass over the array.

Before you implement the program, validate your understanding by specifying the expected output for the given array.

Given the array:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4 | 2 | 6 | 9 | 1 |

What is the resulting output for the given array?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 6 | 9 | 1 | 4 |

## Lambda Expressions

1. Consider the following defined interface:

|  |
| --- |
| **interface Trimmer{**  **public void trimLast(ArrayList<String> arrayList);**  **}** |

We assume that the expected functionality of the ‘trimLast’ method is to remove the last value from the ArrayList.  
Implement the lambda expression that would be used to call the following method:

|  |
| --- |
| **public static void trimList(ArrayList<String> arrayList, Trimmer t){**  **t.trimLast(arrayList);**  **}** |

Implement a fully working example with a sample ArrayList.

## Recursion

1. We can calculate the sum of all integers from 1 to n using a recursive method as per the below:

|  |
| --- |
| **public static int calculateSum(int n) {**  **// Base case: sum of 0 is 0**  **if (n == 0) {**  **return 0;**  **}**  **// Recursive case: add n with the sum of (n-1)**  **return n + calculateSum(n - 1);**  **}** |

The methods call stack for the recursive method when called with the number ‘4’ as such: **calculateSum(4)** would be as follows:

|  |  |  |
| --- | --- | --- |
| return 0 | 0 | 0 |
| return **1 + calculateSum(0);** | 1+0 | 1 |
| return **2 + calculateSum(1);** | 2+1 | 3 |
| return **3 + calculateSum(2);** | 3+3 | 6 |
| return **4 + calculateSum(3);** | 4+6 | 10 |
| return 10 |  |  |

1. Factorial is the product of all numbers preceding a certain number starting from 1 up until and including the number itself and is expressed as Factorial(n) = n\*(n-1)\*(n-2)...\*1

Thus Factorial(6) = 6\*5\*4\*3\*2\*1 = 720.

Implement the recursive method that would calculate factorial, without using a loop.

What would be the method call stack for Factorial(6):

|  |  |  |
| --- | --- | --- |
| Return 1 | 1 | 1 |
| Return 2 \* Factorial(1) | 2\*1 | 2 |
| Return 3 \* Factorial(2) | 3\*2 | 6 |
| Return 4 \* Factorial(3) | 4\*6 | 24 |
| Return 5 \* Factorial(4) | 5\*24 | 120 |
| Return 6 \* Factorial(5) | 6\*120 | 720 |
| Return 720 |  |  |

1. Complete the exercises in the attached [draw.io](http://draw.io) file and insert the responses as images in this word document.
2. Considering the add() method that uses a recursive helper method to add values to the tree:

|  |
| --- |
| **public void add(E e) {**  **root = addRecursive(root, e);**  **}**  **private Node<E> addRecursive(Node<E> current, E e) {**  ***// Base case: If the current node is null, we've found the insertion point***  **if (current == null) {**  **return new Node<>(e);**  **}**  **int comparison = e.compareTo(current.data);**  **if (comparison < 0) {**  ***// If the element is smaller, go left***  **current.left = addRecursive(current.left, e);**  **} else if (comparison > 0) {**  ***// If the element is larger, go right***  **current.right = addRecursive(current.right, e);**  **}**  ***// If the element is a duplicate (comparison == 0), do nothing***  ***// For a simple implementation, you might ignore it or handle it differently***  **return current; *// Return the (potentially new) node***  **}** |

Complete the iterative version of the add() method that would do the same task as the recursive version, which is to add an element to the tree:

|  |
| --- |
| **public void addIterative(E e) {**  **Node<E> newNode = new Node<>(e);**  **if (root == null) {**  **root = newNode;**  **return;**  **}**  **Node<E> current = root;**  **Node<E> parent = null;**  **while (current != null) {**  **parent = current;**  **if (e.compareTo(current.data) < 0) {**  **current = current.left;**  **} else if (e.compareTo(current.data) > 0) {**  **current = current.right;**  **} else {**  ***// Handle duplicates***  **return;**  **}**  **}**  ***// Insert the new node at the empty spot***  **if (e.compareTo(parent.data) < 0) {**  **parent.left = newNode;**  **} else {**  **parent.right = newNode;**  **}**  **}** |