CSE 373: CSE 143 Review Document

Definitions Recap

- **Objects** : Objects have states and behaviors. Objects are instances of classes.
- **Class** : A class can be defined as a template/ blueprint that describes the behaviours/states that object of its type support.
- **Method** : A method is basically a behaviour. A class can contain many methods. It is in methods where the logic is written, data is manipulated and all the actions are executed.
- **Instance Variables** : Also referred to as a "field" or "member". Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.
- **Example** :

```
Class example {//defining a new class
    public int variable = 5; //Instance Variable specific to each instance.
    public static void main(String[] args) { // defining a new method
        System.out.println("Hello");
    } // closing method
} //Closing class.

Class caller {
    // Creating a seperate class
    public static void main(String[] args) {
        example objectOne = new example(); //making an object of example class.
        System.out.println(objectOne.variable); //Printing out the instance Variable specific to the
            object
    }
}
```

Programming Constructs

- **Conditionals** Conditionals are used to ensure that a certain piece of code is executed only when a certain condition is true. Some conditionals are if, else, if else, etc. For example,

```
int x = 5;
if( x == 5) { //if is a conditional
   System.out.println("5");
}
```

- **Loops** : These are used to repeat a certain block of code multiple times based on a certain condition. There are 2 types of loops :
  1. **For Loops** : General Syntax (based on loop variable)

```
for(instantion; condition; updation) {
    // block of code
}

For example :

for(int i = 0; i < 5; i ++) {
    System.out.println(i);
}
```

```
//Prints out all numbers from 0 till 4 on separate lines.
```

For loops can also be used to iterate over all objects of a Collections object that has a predefined iterator. For example like Sets and Maps are a Collection implementation, hence one can iterate over all elements in a set by using the for loop as (Called a for each loop) :

```
// Prints out all elements in the set inputted with each element on a different line.
public static void printElements(Set<String> input) {
    for(String a : input) {
        System.out.println(a);
    }
}
```

2. **While loops** : These loops are used when you wish to keep continuing executing a block of code until a certain condition is true.
   General Syntax:

```
while(condition) {
    //block of code;
}

For example:

int x = 10;
while( x > 5) {
    System.out.println(x);
    x --;
}
```

- **Inheritance** : Java objects can use inheritance. Avoids redundant code/logic by allowing subclasses to use their superclass' code or behavior (public or protected fields, methods, etc.)
  For example :

```
public class Dog {
    public void bark() {
        System.out.println("Woof");
    }
}

public class Husky extends Dog {
    //empty class. No methods
}

However when we make the following calls:

Dog dog = new Husky(); // making a new object of Husky class.
dog.bark(); // This prints "Woof" although a bark method is not implemented for the Husky class. The
    Husky class in this case inherits the method from it's super class - Dog.
```

- **Arrays** : These are basically data structures used to store data in a index based fashion.
  General functions used : Suppose we have the following array : 1,3,5,7,9 named arrayOne.

1. Length : arrayOne.length // this returns 5.
2. Getting an Element : (Use zero based indexing) arrayOne[index];
   For example : arrayOne[0] = 1 and arrayOne[4] = 9.

- **Recursion** : Recursion is the practice of a function calling itself to solve a smaller subset of the problem. The idea behind Recursion is to call the same function from inside a function and provide a similar but smaller problem to next function call. In a lot of cases, (such as in Binary Trees) this makes the problem much simpler to complete. Usually one must make the choice of whether to solve a problem iteratively or recursively. For example, suppose we need to calculate the factorial of a number, we can do it in two ways :

```
1. Iteratively :
public static void factorial(int n) {
    int product = 1;
    for(int i = 1; i <= n; i ++) {
        product *= i;
    }
    System.out.println(product);
}

2. Recursively :
public static void factorial(int n) {
   System.out.println(helperFactorial(n));
 }

public static int helperFactorial(int n) {
    if(n == 0) {
        return 1;
    } else {
        return n * helperFactorial( n - 1);
    }
}
```

Data Structure Revision

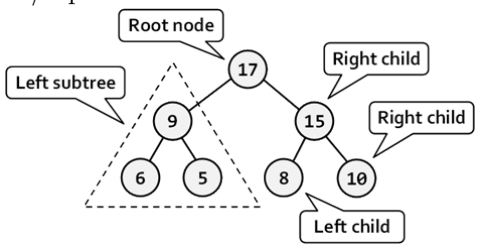
1. **Singly Linked List** : Data Structures used to store data in the form of chains with nodes connected to each other. Can think of these as train carriages. Linked lists are not restricted by length however they do not support index based referencing i.e. it is not possible to get the 5th element in a linkedlist by saying linkedList.get(5) or linkedList[5]. Hence we must always keep track of the starting of the list in addition to other nodes we wish to track.
Typically the nodes are internally defined private class objects that can store the value/element as well as a reference to the next link in the linked list. For example suppose we want a linked list storing strings :

```
private class LinkedNode {

    String element;
    LinkedNode next;

    public LinkedNode(String input, LinkedNode step) {
        element = input;
        next = step;
    }
} // end of Node class

public class LinkedList {
    private LinkedNode front; // this will store the starting of the LinkedList at all points.

    public LinkedList(String first) {
        front = new LinkedNode(first, null); // initializing the first node in the linked list.
    }
}
```

To iterate over a linked list we must start at the starting of the Linked List and go over to till where we want to stop. However, we must be very careful about when to change the front node of the linked list. For example, if I wish to print out all the elements in the linked List :

```
ListNode current = front; //storing reference to front that we will keep changing. Front should not
     change in this case.

while(current != null) {
    System.out.println(current.element);
    current = current.next; //iterating current over the linked list.
}
```

2. **Simple Binary Trees** : Binary Trees are basically a graph based data structure that store data in the form of nodes on a tree with two branches to each node. To iterate over the various nodes in a Binary Search Tree, we usually keep track of the root/top node of a tree and use it to move through the tree.



Ways of moving through a tree. For example, consider the tree shown above and the Node class defined containing three public/protected instance variables, String element, Node Left and Node right.

(a) **In Order** : Left - Center - Right.

```
public void inOrder(Node root) {
    if(root != null) {
        inOrder(root.left);
        System.out.println(root.element);
        inOrder(root.right);
    }
}

This prints out the tree in the format : 6,9,5,17,8,15,10.
```

(b) **Pre Order** : Center - Left - Right.

```
public void preOrder(Node root) {
    if(root != null) {
        System.out.println(root.element);
        preOrder(root.left);
        preOrder(root.right);
    }
}

This prints out the tree in the format : 17,9,6,5,15,8,10.
```

(c) **Post Order** : Left - Right - Center.

```
public void postOrder(Node root) {
    if(root != null) {
        System.out.println(root.element);
        postOrder(root.left);
        postOrder(root.right);
    }
}

This prints out the tree in the format :
6,5,9,8,10,15,17.
```

3. **Stacks and Queues** : **Stacks** are LIFO(Last in First out) Structures. What this means is suppose I put in all numbers from 1 to 10 in order into a stack (1, 2, ...) and then if I pop once from the stack I'll get the element that was entered last (10) then if I pop once again, I'll get 9 , etc. Some stack methods are :

(a) Initialize :

$$Stack\langle DataType\rangle name = newStack\langle DataType\rangle();$$

For example, a stack of integers would be initialized as

$$Stack\langle Integer\rangle stackUno = newStack\langle Integer\rangle();$$

(b) push() : To put in elements into the stack. stackUno.push(1000);

(c) pop() : Returns the element following the LIFO pattern. stackUno.pop() returns an Integer(int).

(d) size() : Returns the size of the stack.

Now let's look at Queues.

Queues are FIFO(First In First Out) structures. What this means is suppose I put in all numbers from 1 till 10 into a queue in order(1, 2, ...) and I remove from it, I will receive 1 first then if I remove again I will get 2, etc. Some functions for Queues are :

(a) Initialize :

$$Queue\langle DataType\rangle name = newLinkedList\langle DataType\rangle();$$

For example, a queue of integers would be initialized as

$$Queue\langle Integer\rangle queueOne = newLinkedList\langle Integer\rangle();$$

(b) enqueue() : To push in elements into the queue.

(c) dequeue() : To remove elements from the queue. Returns which element was removed.

(d) size() : Returns the size of the queue.

Stacks and Queues are really nice because they have very very fast times for addition and removal if we only need to remove from the ends(FIFO or LIFO).

Computational Complexity

The Big O complexity of an algorithm is basically used to compare different algorithms based on how long/ how many steps they take. Some common complexities are:

1. **O(1)**: Any algorithm that takes a really small amount of time or a countable number of steps is said to have O(1) Big O complexity. For example, here are some examples of algorithms with O(1) complexity :

```
1. System.out.println(5);
2. int x = 4;
3. for(int i = 0; i < 5; i ++) {//Can count that loop runs exactly 5 times in every case.
       System.out.println(i);
   }
```

2. **O(n)**: Any algorithm that takes a linear multiple of the input value amount of time/steps is said to have O(n) complexity. For example, here are some examples of algorithms with O(n) complexity.

```
1. for(int i = 0; i < n; i ++) {
       System.out.println(n);
   }

2.  public static void recurse(n) {
       if(n == 1) {
           System.out.println(n);
       }
       recurse(n-1);
   }
```

3. **O(n^2)** : Any algorithm that takes a quadratic multiple of the input value amount of time/steps is said to have O(n^2) complexity. For example, here are some examples of algorithms with O(n^2) complexity.

```
1. for(int i = 0; i < n*n; i ++) {
       System.out.println(n);
   }

2. for(int i = 0; i < n; i ++) {
       for(int j = 0; j < n; j++) {
           System.out.println(i+j);
       }
   }
```