6.1

Explain the difference between a class, an abstract class and an interface in Java:

A class can be instantiated and has methods with definitions and attributes.

An abstract class is a template for a class – which contains methods with definitions and also abstract methods – which are declarations that the class which inherits the abstract class must declare the methods before it can be instantiated. Abstract classes cannot be instantiated. Abstract classes can have attributes.

An interface is a template for a class. It contains no method definitions. Every mathod in an interface is abstract (the abstract keyword is hidden). Interfaces can have attributes.

6.8

1. Create a Java interface for a standard queue (i.e. FIFO).

```
interface QueueInterface{
    void enqueue(int x);
    int dequeue();
    boolean isEmpty();
}
```

2. Implement OOPListQueue, which should use two OOPLinkedList objects as per the queues you constructed in your FoCS course. You may need to implement a method to reverse lists.

```
package uk.ac.cam.hjel2.oop.sv2;

import java.util.NoSuchElementException;

public class OOPListQueue implements QueueInterface{

    private OOPLinkedList headlist;
    private OOPLinkedList taillist;

    OOPListQueue(){
        headlist = new OOPLinkedList();
        taillist = new OOPLinkedList();
    }

    @Override
    public void enqueue(int x) {
        taillist.add(x);
    }

    @Override
    public int dequeue() {
        if (headlist.length != 0){
            return headlist.remove();
        }
        else if(taillist.length != 0) {
            taillist.reverse();
            headlist = taillist;
            taillist = new OOPLinkedList();
            return dequeue();
        }
        else {
            throw new NoSuchElementException();
        }
    }

    @Override
    public boolean isEmpty() {
        return (headlist.length==0 && taillist.length==0);
    }
}
```

3. Implement OOPArrayQueue. Use integer indices to keep track of the head and tail positions.

```
package uk.ac.cam.hjel2.oop.sv2;

import java.util.Arrays;
import java.util.NoSuchElementException;

public class OOPArrayQueue implements QueueInterface{
    private int[] queue;
    private int hd;
    private int tl;
    private int length;

    OOPArrayQueue(){
        queue = new int[10];
        this.length = 10;
        // hd is the first occupied element
        // tl is the next free element
        // if the queue fills up on insertion then we immediately create
        // a new array (with double the length) to be the queue
        // so the only case that hd=tl is when the queue is empty
    }

    @Override
    public void enqueue(int x) {
        if ((tl + 1) % length == hd) {
            System.out.println("called");
            System.out.printf("hd: %s tl: %s len: %s%n", hd, tl, length);
            int[] temp = new int[2 * length];
            System.arraycopy(queue, hd, temp, 0, length - hd);
            System.arraycopy(queue, tl, temp, length - hd + 1, (hd - tl - 1) % length);
            queue = temp;
            hd = 0;
            tl = length;
            length *= 2;
        }
        queue[tl] = x;
        tl++;
        tl %= length;
    }

    @Override
    public int dequeue() {
        if (!isEmpty()) {
            int first = queue[hd];
            hd++;
            hd %= length;
            return first;
        }
        else {
            throw new NoSuchElementException();
        }
    }

    @Override
    public boolean isEmpty() {
        return hd==tl;
    }

    @Override
    public String toString(){
        return Arrays.toString(queue);
    }
}
```

4. For OOPListQueue; dequeue has a $\Theta(1)$ ammortized cost (however an indivual dequeue could have $O(n)$ time). Enqueing and isEmpty are all $\Theta(1)$.

   OOPArrayQueue has a $\Theta(1)$ time for dequeue and isEmpty – and a $\Theta(1)$ ammortized cost for enqueue.

9.3

Write a Java program that reads in a text file that contains two integers on each line, separated by a comma (i.e. two columns in a comma-separated file). Your program should print out the same set of numbers, but sorted by the first column and subsorted by the second.

```
package uk.ac.cam.hjel2.oop.sv2;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.Scanner;

public class OOPFileReader {

    private static class OOPComparator implements Comparator<int[]>{

        @Override
        public int compare(int[] o1, int[] o2) {
            switch (Integer.compare(o1[0], o2[0])){
                case -1 -> {return -1;}
                case 0 -> {return Integer.compare(o1[1], o2[1]);}
                default -> {return 1;}
            }
        }
    }

    public void read(String path) throws FileNotFoundException {
        List<int[]> tuples = new ArrayList<>();
        Scanner scanner = new Scanner(new File(path));
        String[] line;
        while (scanner.hasNext()){
            line = scanner.next().split(",");
            tuples.add(new int[]{Integer.parseInt(line[0]), Integer.parseInt(line[1])});
        }
        tuples.sort(new OOPComparator());
        for (int[] t: tuples){
            System.out.printf("%s,%s%n", t[0], t[1]);
        }
    }
}
```