Question 1

1.

```
public class Person {
    // Question 1) 1.

    private final String firstName;
    private final String lastName;

    public Person(){
        this.firstName = "";
        this.lastName = "";
    }

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    @Override
    public String toString() {
        return "firstName='" + firstName + '\'' +
                ", lastName='" + lastName + '\'';
    }
}
```

2.

```
public class Student extends Person{
    private final String studentID;
    private float GPA;

    public Student(String firstName, String lastName, String studentID,
float GPA) {
        super(firstName, lastName);
        this.studentID = studentID;
        this.GPA = GPA;
    }

    public String getStudentID() {
        return studentID;
    }

    public float getGPA() {
        return GPA;
    }

    public void setGPA(float GPA) {
        this.GPA = GPA;
    }
```

```
    @Override
    public String toString() {
        return super.toString()+" studentID='" + studentID + '\'' +
                ", GPA=" + GPA;
    }


}
```

3.

```
import java.util.ArrayList;

public class Course {
    private String courseName;
    private String year;
    private ArrayList<Student> students;

    public Course(String courseName, String year, ArrayList<Student>
students) {
        this.courseName = courseName;
        this.year = year;
        this.students = students;
    }

    // Add a new student
    public void addStudent(Student newStudent){
        this.students.add(newStudent);

    }

    // Remove a student based on studentID
    public void removeStudent(String studentID){

        for (Student student : students){
            if (student.getStudentID().equals(studentID)){
                students.remove(student);
                students.trimToSize();
            }

        }
    }

    public double avgGPA(){
        double total = 0.0;
        for (Student student : students){
            total += student.getGPA();
        }
        return (total/students.size());
    }


    public String getCourseName() {
        return courseName;
    }

    public void setCourseName(String courseName) {
        this.courseName = courseName;
    }
```

```
    public String getYear() {
        return year;
    }

    public void setYear(String year) {
        this.year = year;
    }

    public ArrayList<Student> getStudents() {
        return students;
    }

    public void setStudents(ArrayList<Student> students) {
        this.students = students;
    }
}
```

Strategy Pattern:

```
import java.util.ArrayList;

public interface DisplayHandle {
    public boolean outOfOrder(int index, ArrayList<Student> array);
}
```

```
import java.util.ArrayList;

public class Display {
    private DisplayHandle thisDisplayHandle = null;


    public Display(DisplayHandle thisDisplayHandle) {
        this.thisDisplayHandle = thisDisplayHandle;
    }

    public void display(ArrayList<Student> array){
        sort(array);

        String all = "";
        for (Student student : array){
            all += student.toString()+". ";
        }

        System.out.println(all);

    }

    private int sort(ArrayList<Student> array){
        int operations = 0;
        int length = 0;

        length = array.size();
        operations = 0;
        if (length <= 1) {
            return operations;
        }
```

```
        for(int nextToLast = length - 2; nextToLast >=0;nextToLast--){
            for (int index=0; index <= nextToLast; index++){
                if (thisDisplayHandle.outOfOrder(index, array)){
                    swap(index, array);
                }
                operations++;
            }
        }
        return operations;
    }


    private void swap(int index, ArrayList<Student> array) {
        Student temp = array.get(index);
        array.set(index, array.get(index+1));
        array.set((index+1), temp);
    }
}
```

```
import java.util.ArrayList;

public class LastNameDisplayHandle implements DisplayHandle{

    public boolean outOfOrder(int index, ArrayList<Student> array) {
        Student current = array.get(index);
        Student next = array.get(index+1);

        // returns negative if next comes before previous
        if (next.getLastName().compareTo(current.getLastName())<0){
            return true;
        }
        return false;
    }



}
```

```
import java.util.ArrayList;

public class GPADisplayHandle implements DisplayHandle{


    @Override
    public boolean outOfOrder(int index, ArrayList<Student> array) {
        Student current = array.get(index);
        Student next = array.get(index+1);

        // returns negative if next comes before previous
        if (next.getGPA()<current.getGPA()){
            return true;
        }
        return false;
    }

}
```

Question 2

```
public static int num_discs(int n, int[] s){
    ArrayList<Integer> discs = new ArrayList<Integer>();

    for (int file : s){
        boolean added = false;

        for (int index=0; index<discs.size(); index++){
            // if the file can fit onto the disc
            if ((discs.get(index)+file)<=n){
                discs.set(index, discs.get(index)+file);
                added = true;
                break;
            }
        }

        // if unable to fit onto any disc
        if (!added){
            discs.add(file);
        }


    }
    //System.out.println(discs.toString());
    return (discs.size());
}
```

Question 3

1.

```
// Question 3 1.

import java.util.Objects;

public class Pair <T1,T2>{
    private T1 first;
    private T2 second;


    public Pair(T1 first, T2 second) {
        this.first = first;
        this.second = second;
    }

    public T1 getFirst() {
        return first;
    }

    public void setFirst(T1 first) {
        this.first = first;
    }

    public T2 getSecond() {
        return second;
    }
```

```
    public void setSecond(T2 second) {
        this.second = second;
    }

    @Override
    public String toString() {
        return "(" + first.toString() +
                ", " + second.toString() +
                ")";
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Pair<T1, T2> pair = (Pair<T1, T2>) o;
        return Objects.equals(getFirst(), pair.getFirst()) &&
                Objects.equals(getSecond(), pair.getSecond());
    }

    @Override
    public int hashCode() {
        return Objects.hash(getFirst(), getSecond());
    }
}
```

2.

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Set;

// Question 3 2.

public class TwoDHashMap <K1, K2, V>{
    HashMap<Pair<K1,K2>, V> map;

    // Constructor
    public TwoDHashMap() {
        this.map = new HashMap<Pair<K1,K2>, V>();
    }

    public boolean containsKey (K1 k1, K2 k2){
        Pair<K1,K2> key = new Pair<K1,K2>(k1, k2);

        if (this.map.containsKey(key)){
            return true;
        }

        return false;
    }

    public V get (K1 k1, K2 k2){
        Pair<K1,K2> key = new Pair<K1,K2>(k1, k2);

        if (this.map.containsKey(key)){
            return (this.map.get(key));
        }
        return null;
    }
```

```
    public void put (K1 k1, K2 k2, V v){
        Pair<K1,K2> key = new Pair<K1,K2>(k1, k2);
        this.map.put(key, v);
    }

    public ArrayList<Pair<K1,K2>> getAllKeys(){
        Set<Pair<K1, K2>> keySet = map.keySet();
        ArrayList<Pair<K1, K2>> listOfKeys = new ArrayList<Pair<K1,
K2>>(keySet);

        return (listOfKeys);

    }

}
```

Question 5

1. Abstraction is the process of creating self-contained units that allows a software solution to be parametised.
   Inheritance is a form of abstraction in which child classes inherit properties and behaviours from existing parent classes.
   Delegation is the process of a class delegating responsibilities, behaviours and tasks to other classes through containership and association.

2. Abstraction supports good design as it allows code to be re-used. This because abstraction is the process of creating self-contained units of code (for example, classes or interfaces) that are well-suited to a purpose and thus, can be reused in other applications.

   Furthermore, inheritance supports good design as it allows for existing software solutions and code to be extended upon. This is because inheritance is the process of creating new child classes that can inherit properties and behaviours from existing parent classes, thus building upon existing code and design. This not only allows code to be reused and extended upon, but also reduces the need for unnecessary duplication of code.

   Finally, delegation supports good design as it allows the delegation of responsibilities to other classes more suited for that purpose, and thus the adaptability of a design. For example, a class can contain or use the functionality of another class in other to adapt its current purpose to the required application.

3.