Q1 Consider the output of this program:

```
public static void main(String[] args)  {

    Set<String> s1 = new HashSet<>(); // HashSet implements Set

    s1.add("Apple");
    s1.add("Bear");
    System.out.println(s1); //Output#1: [Apple, Bear]

    s1.add("Cat");
    s1.add("Dog");
    s1.add("Apple");
    s1.add("Bear");
    System.out.println(s1); //Output#2: [Apple, Cat, Bear, Dog]
}
```

(a) Why does Output#2 show [Apple, Cat, Bear, Dog] instead of [Apple, Bear, Cat, Dog, Apple, Bear] ?

(b) What if we change to Treeset?

```
Set<String> s1 = new TreeSet<>(); // TreeSet implements Set
```

Q2 Using Comparator in Collections.sort()

What are the outputs of the following program? Complete the blank.

```
class Product {
    private int part_number;  //sometimes we want to compare part_number
    private String product_name; //sometimes we want compare product name
    public Product(String pName, int pNum) {part_number=pNum; product_name=pName;}
    public int getPNum() {return part_number;}
    public String getPName() {return product_name;}
    public String toString() {return String.format("%s(%d)",product_name,part_number);}
}

public class MainArrayList{

    public static void main(String [] args)
    {
        ArrayList<Product> arr = new ArrayList<>();
        arr.add(new Product("Widget",4562));
        arr.add(new Product("Modem",9912));
        arr.add(new Product("Toaster",1234));

        Collections.sort(arr, new Comparator<Product>() {
            public int compare(Product a, Product b) {
                return Integer.compare(a.getPNum(),b.getPNum());
            }
        });

        System.out.println(arr); //Output: [Toaster(1234), Widget(4562), Modem(9912)]

        Collections.sort(arr, new Comparator<Product>() {
            public int compare(Product a, Product b) {
                return a.getPName().compareTo(b.getPName());
            }
        });

        System.out.println(arr); //Output: _____
    }
}
```

Q3 Using Disjoint - Rewrite the `hasBeenClassmateOf` method we wrote in Lab11 Q1a.

```
public class Student {

    private String id;

    ArrayList<Offering> studies;

    public Student(String id) {
        this.id = id;
        studies = new ArrayList<Offering>();
    }

    public void takeCourse(Course course, String sem) {
        studies.add(course.getOffering(sem));
    }

    public boolean hasBeenClassmateOf(Student s2) {
        for (Offering o:studies)
            if (s2.hasTakenCourse(o))
                return true;
        return false;
    }

    private boolean hasTakenCourse(Offering target) {
        for (Offering o:studies)
            if (o==target)
                return true;
        return false;
    }
}
```

```
public boolean hasBeenClassmateOf(Student s2) {



}
```

Q4 How to work out Mock Midterm Q1

```
public static void main(String[] args)
{

    try {
        String report = analyze();
        System.out.println(report);









}
----------------------------------------------------------------
private static String analyze() throws FileNotFoundException
{

    int total=0;

    Scanner f = null;

}
```

Case 1: The file *profits.txt* has 12 numbers:

```
Total: 1212820000
From main(): Reporting is OK.
```

```
113890000
101200000
98040000
97670000
94740000
102990000
114920000
113180000
82950000
119060000
80620000
93560000
```

Case 2: The file *profits.txt* does not exist.

```
Cannot open input file!
```

Case 3: The file *profits.txt* has less than 12 numbers.

```
Insufficient data!
Total: [cannot be calculated]
```

Case 4: The file *profits.txt* has non-integer like "oneMillion"

```
Wrong data!
Total: [cannot be calculated]
```

Q5 How to work out Mock Midterm Q2?

```
public class Main {
  public static void main(String[] args) {
    Student a = new Student("Ada", 'F', "CS"); //name, gender, programme
    Student b = new Student("Billy", 'M', "Math");
    Student c = new Student("Candy", 'F', "CS");
    Student d = new Student("Daisy", 'F', "CS");
    Student e = new Student("Emily", 'F', "Math");

    Rule rG = new RuleSameGender();
    Rule rBk = new RuleSimilarBackground();

    a.listTargets(); //[Output] Ada: Billy Candy Daisy Emily
    b.listTargets(); //[Output] Billy: Ada Candy Daisy Emily

    a.addRule(rG);
    a.listTargets(); //[Output] Ada: Candy Daisy Emily
    a.addRule(rBk);
    a.listTargets(); //[Output] Ada: Candy Daisy

    Student f = new Student("Fanny", 'F', "CS");
    a.listTargets(); //[Output] Ada: Candy Daisy Fanny
    f.listTargets(); //[Output] Fanny: Ada Billy Candy Daisy Emily
  }
}
```

How does the solution achieve the *Open-close principle* of OO design? Briefly explain in at most 40 words.