Section A: Question 1
Report

Description of Program

This program is made of 6 java files (App.java, Cars.java, familyCars.java, sportsCars.java, businessDiscounts.java, businessCars.java) and the functionality of this program is exactly as how is written in the question paper. All the car data that are in this program are taken from real life data sources of car manufacturers, and thus resemble genuine data. The source code of the program is also well commented to make it much easier to understand the code and all the unnecessary things are taken out (like excessive line breaks).

Aside from that, this program has all the functionalities mentioned in the question, which are as follows:

- Instantiation of 4 objects and 1 from additional subclass
- Encapsulation of all the necessary attributes
- Inheritance of 3 subclasses from 1 superclass
- Polymorphism using method overloading
- Implementation of 1 abstract method
- Creation of UML class diagram of the program
- Program outputs (images)
- Error handling (and exception handling)
- Source code (given at the end of report)

They will now be discussed further in details.

Instantiation

All the objects that are created/instantiated in the App.java file are examples of the instantiation that is done in the program. Aside from that, making the main program instance itself is also a form of instantiation. Here is a screenshot of the code (App.java) where instantiation was done:

```
public class App {

    public void init() {
        // Creating Car Object Instances (Instantiation Of Classes)
        familyCars familyCar1 = new familyCars("Toyota-Corolla", 7500, 270, 1200);
        familyCars familyCar2 = new familyCars("Hyundai-Kona", 6000); // <= 1 instance using method overloading)
        sportsCars sportsCar1 = new sportsCars("Audi-R8", 80000, 3.9, 320, 5000);
```

Figure 1

Encapsulation

The variables in each class (for example “carModel” in Cars.java, from the source code) are all made private, and they can only be accessed by using the getter and setter methods that are made to get and set the values of those variables. Some variables may only have getters, because some of these values (for example car model, car mileage, car top speed) are constant data, and thus doesn’t need to be changed in the future. The image below shows usage of encapsulation in code:

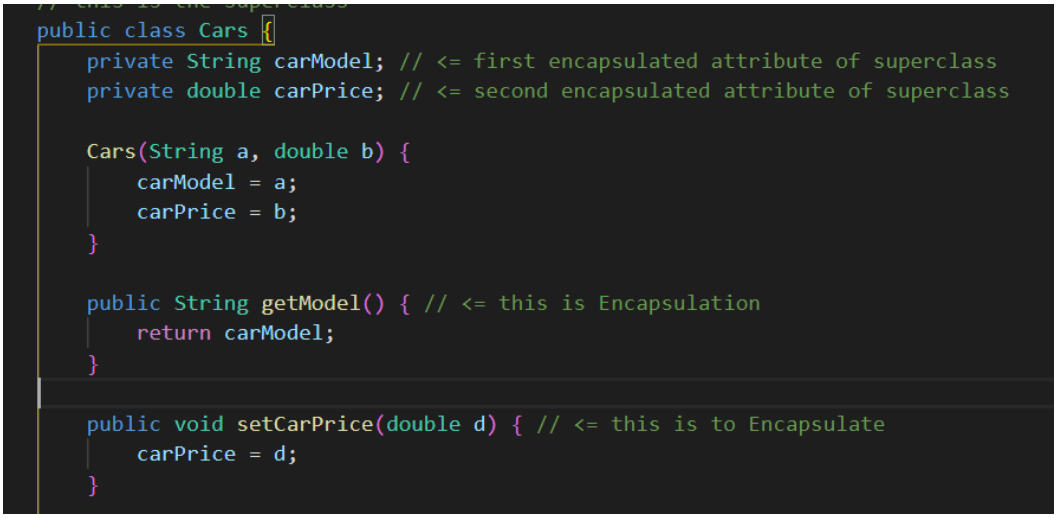

Figure 2

Inheritance

The Cars.java is the main superclass file and its subclasses inherit from it by extending to this class. The subclasses that inherit form this “Cars” superclass (Cars.java) are “familyCars” class (familyCars.java), “businessDiscounts” class (businessDiscounts.java), “sportsCars” class (sportsCars.java), and “businessCars” class (businessCars.java). The image below shows the usage of inheritance in the code:

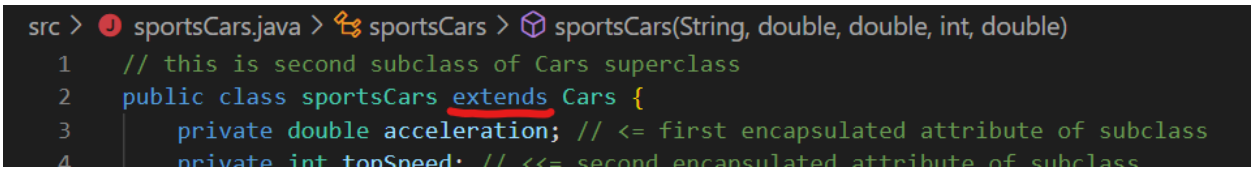

Figure 3: sportCars inheriting from Cars by extending to Cars

Polymorphism

In the familyCars class (familyCars.java), the same constructor method of the class is used twice, meaning after the initial usage it is reused for the second time. By definition this process of reusing

the same method in two different ways is called method overloading. So, polymorphism via method overloading was done successfully. The image of the code is shown below:

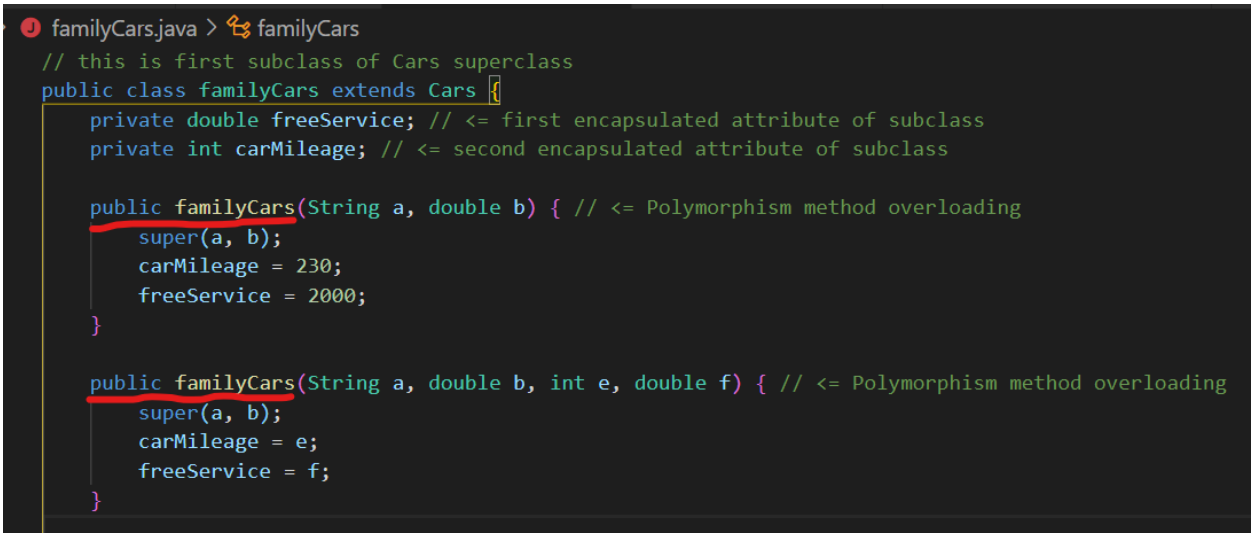

Figure 4: Polymorphism of the "familyCars" constructor method

Abstract Method

The businessDiscount class (businessDiscount.java) is an abstract class. And it has an abstract method called “printBusinessDiscounts()”. This method is declared, but not defined. This method is defined in the end of the businessCars class (businessCars.java) just before using it. So, this method is an abstract method, since it is declared at the abstract class, but not defined until it has been called in another subclass. The image below shows the usage within the code:

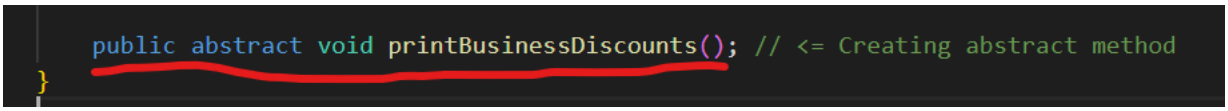

Figure 5: Declaring Abstract Method in businessDiscount class

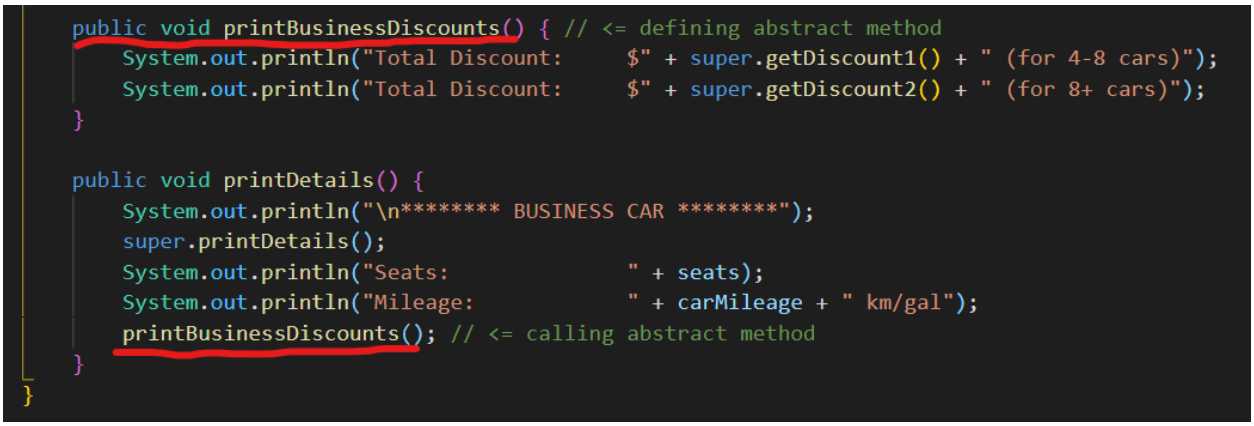

Figure 6: Defining and using the previously-declared abstract method in the businessCars class

UML Class Diagram

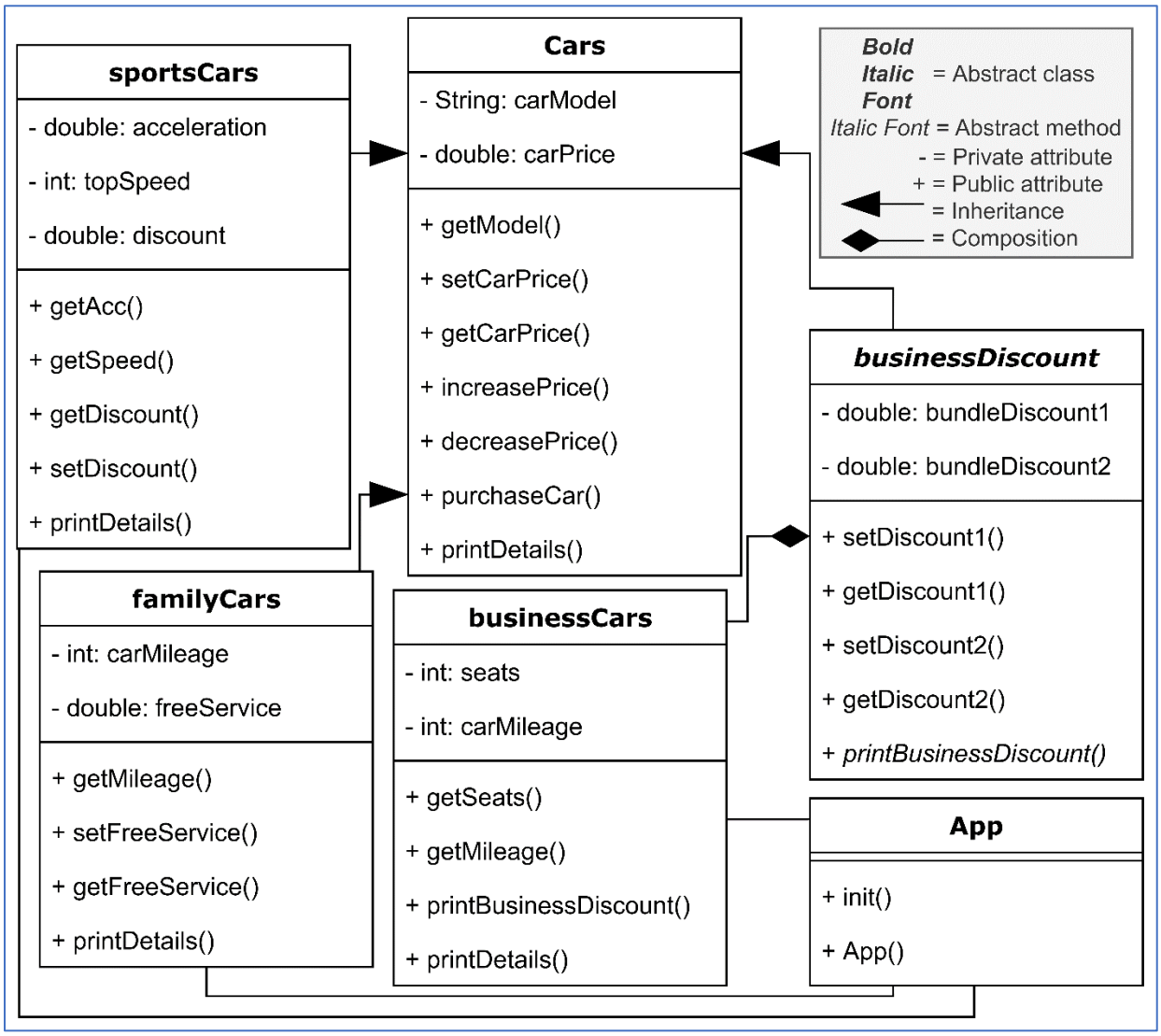

Figure 7: UML Class Diagram

Code Outputs (Terminal Output Screenshots)

```
******** FAMILY CAR ********
Model:              Toyota-Corolla
Price:              $7500.0
Mileage:            270 km/gal
Free Service:       $1200.0

******** FAMILY CAR ********
Model:              Hyundai-Kona
Price:              $6000.0
Mileage:            230 km/gal
Free Service:       $2000.0

******** SPORT CAR ********
Model:              Audi-R8
Price:              $80000.0
Acceleration(0-60): 3.9 second
Top Speed:          320 km/h
Discount:           $5000.0

Increase $2000.0 to Toyota-Corolla - Success

Decrease $7000.0 from Hyundai-Kona - Rejected

Decrease $1000.0 from Hyundai-Kona - Success

Increase $1000.0 to Audi-R8 - Success

Purchase Audi-R8 - price $81000.0

Purchase Bugatti Chiron - price $120000.0
```

Continues to next page…

```
******** FAMILY CAR ********
Model:              BMW-S2
Price:              $8000.0
Mileage:            250 km/gal
Free Service:       $1200.0

******** FAMILY CAR ********
Model:              Ford-Focus
Price:              $10000.0
Mileage:            280 km/gal
Free Service:       $1500.0

******** SPORT CAR ********
Model:              Nissan-GTR
Price:              $98000.0
Acceleration(0-60): 3.4 second
Top Speed:          380 km/h
Discount:           $7000.0

******** SPORT CAR ********
Model:              Bugatti Chiron
Price:              $120000.0
Acceleration(0-60): 2.9 second
Top Speed:          460 km/h
Discount:           $14000.0

******** BUSINESS CAR ********
Model:              Mercedes-Benz AMG
Price:              $15000.0
Seats:              4
Mileage:            170 km/gal
Total Discount:     $7000.0 (for 4-8 cars)
Total Discount:     $12000.0 (for 8+ cars)
```

Source Code

App.java:

```
// main programm
public class App {

    public void init() {
        // Creating Car Object Instances (Instantiation Of Classes)
        familyCars familyCar1 = new familyCars("Toyota-Corolla", 7500, 270, 1200);
        familyCars familyCar2 = new familyCars("Hyundai-Kona", 6000); // <= 1
instance using method overloading)
        sportsCars sportsCar1 = new sportsCars("Audi-R8", 80000, 3.9, 320, 5000);
        familyCars familyCar3 = new familyCars("BMW-S2", 8000, 250, 1200);
        familyCars familyCar4 = new familyCars("Ford-Focus", 10000, 280, 1500);
        sportsCars sportsCar2 = new sportsCars("Nissan-GTR", 98000, 3.4, 380, 7000);
        sportsCars sportsCar3 = new sportsCars("Bugatti Chiron", 120000, 2.9, 460,
14000);
        businessCars businessCar1 = new businessCars("Mercedes-Benz AMG", 15000,
7000, 12000, 4, 170);

        // Printing Car Instance Details
        familyCar1.printDetails();
        familyCar2.printDetails();
        sportsCar1.printDetails();
        // transaction start
        familyCar1.increasePrice(2000);
        familyCar2.decreasePrice(7000);
        familyCar2.decreasePrice(1000);
        sportsCar1.increasePrice(1000);
        sportsCar1.purchaseCar();
        sportsCar3.purchaseCar();
        // transaction end
        familyCar3.printDetails();
        familyCar4.printDetails();
        sportsCar2.printDetails();
        sportsCar3.printDetails();
        businessCar1.printDetails();
    }

    public static void main(String[] args) throws Exception {
        App app = new App(); // <= Instantiation of the programe
        app.init();
    }
}
```

Cars.java:

```
public class Cars { // this is the superclass
    private String carModel; // <= first encapsulated attribute of superclass
    private double carPrice; // <= second encapsulated attribute of superclass

    Cars(String a, double b) {
        carModel = a;
        carPrice = b;
    }

    public String getModel() { // <= this is Encapsulation
        return carModel;
    }

    public void setCarPrice(double d) { // <= this is to Encapsulate
        carPrice = d;
    }

    public double getCarPrice() { // <= this is Encapsulation
        return carPrice;
    }

    public void increasePrice(double inc) {
        try {
            carPrice = (carPrice + inc);
            System.out.println("\nIncrease $" + inc + " to " + carModel + " - Success");
        } catch (Exception e) {
            System.out.println("\nIncrease $" + inc + " to " + carModel + " - Rejected");
        }
    }

    public void decreasePrice(double dec) { // <= this is to Encapsulate
        if ((carPrice - dec) > 1) {
            carPrice = (carPrice - dec);
            System.out.println("\nDecrease $" + dec + " from " + carModel + " -
Success");
        } else {
            System.out.println("\nDecrease $" + dec + " from " + carModel + " -
Rejected");
        }
    }

    public void purchaseCar() {
        try {
            System.out.println("\nPurchase " + carModel + " - price $" + carPrice);
        } catch (Exception e) {
            System.out.println("\nPurchase " + carModel + " - price $" + carPrice + " -
Rrejected");
        }
    }

    public void printDetails() {
```

```
        System.out.println("Model:              " + carModel);
        System.out.println("Price:              $" + carPrice);
    }
}
```

familyCars.java:

```
// this is first subclass of Cars superclass
public class familyCars extends Cars {
    private double freeService; // <= first encapsulated attribute of subclass
    private int carMileage; // <= second encapsulated attribute of subclass

    public familyCars(String a, double b) { // <= Polymorphism method overloading
        super(a, b);
        carMileage = 230;
        freeService = 2000;
    }

    public familyCars(String a, double b, int e, double f) { // <= Polymorphism
method overloading
        super(a, b);
        carMileage = e;
        freeService = f;
    }

    public int getMileage() { // <= this is Encapsulation
        return carMileage;
    }

    public void setFreeService(double g) { // <= this is to Encapsulate
        freeService = g;
    }

    public double getFreeService() { // <= this is Encapsulation
        return freeService;
    }

    public void printDetails() {
        System.out.println("\n******** FAMILY CAR ********");
        super.printDetails();
        System.out.println("Mileage:            " + carMileage + " km/gal");
        System.out.println("Free Service:       $" + freeService);
    }
}
```

sportsCars.java:

```java
// this is second subclass of Cars superclass
public class sportsCars extends Cars {
    private double acceleration; // <= first encapsulated attribute of subclass
    private int topSpeed; // <<= second encapsulated attribute of subclass
    private double discount; // <= third attribute

    public sportsCars(String a, double b, double e, int f, double g) {
        super(a, b);
        acceleration = e;
        topSpeed = f;
        discount = g;
    }

    public double getAcc() { // <= this is Encapsulation
        return acceleration;
    }

    public int getSpeed() { // <= this is Encapsulation
        return topSpeed;
    }

    public void setDiscount(double j) { // <= this is to Encapsulate
        discount = j;
    }

    public double getDiscount() { // <= this is Encapsulation
        return discount;
    }

    public void printDetails() {
        System.out.println("\n******** SPORT CAR ********");
        super.printDetails();
        System.out.println("Acceleration(0-60): " + acceleration + " second");
        System.out.println("Top Speed:          " + topSpeed + " km/h");
        System.out.println("Discount:           $" + discount);
    }
}
```

businessDiscounts.java:

```java
// this is third subclass of Cars superclass, and also the abstract class
public abstract class businessDiscounts extends Cars {
    private int bundleDiscount1; // <<= third encapsulated attribute of subclass
    private int bundleDiscount2; // <= fourth encapsulated attribute of subclass

    public businessDiscounts(String a, double b, int c, int d) {
        super(a, b);
        bundleDiscount1 = c;
        bundleDiscount2 = d;
    }

    public void setDiscount1(int q) { // <= this is to Encapsulate
        bundleDiscount1 = q;
    }

    public int getDiscount1() { // <= this is Encapsulation
        return bundleDiscount1;
    }

    public void setDiscount2(int r) { // <= this is to Encapsulate
        bundleDiscount2 = r;
    }

    public int getDiscount2() { // <= this is Encapsulation
        return bundleDiscount2;
    }

    public abstract void printBusinessDiscounts(); // <= Creating abstract method
}
```

businessCars.java:

```
// this is the subclass of businessDiscounts superclass
public class businessCars extends businessDiscounts {
    private int seats; // <= first encapsulated attribute of subclass
    private int carMileage; // <= second encapsulated attribute of subclass

    public businessCars(String a, double b, int c, int d, int k, int l) {
        super(a, b, c, d);
        seats = k;
        carMileage = l;
    }

    public int getSeats() { // <= this is Encapsulation
        return seats;
    }

    public int getMileage() { // <= this is Encapsulation
        return carMileage;
    }

    public void printBusinessDiscounts() { // <= defining abstract method
        System.out.println("Total Discount:     $" + super.getDiscount1() + " (for 4-8
cars)");
        System.out.println("Total Discount:     $" + super.getDiscount2() + " (for 8+
cars)");
    }

    public void printDetails() {
        System.out.println("\n******** BUSINESS CAR ********");
        super.printDetails();
        System.out.println("Seats:              " + seats);
        System.out.println("Mileage:            " + carMileage + " km/gal");
        printBusinessDiscounts(); // <= calling abstract method
    }
}
```

0340938
Alister Animesh Baroi

Section A Question 2

```java
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.SpringLayout.Constraints;
import java.awt.*;

public class GUI {
    public GUI() {
        JFrame frame = new JFrame();
        JPanel panel = new JPanel();
        panel.setBorder(BorderFactory.createEmptyBorder(30, 30, 10, 30));
        panel.setLayout(new GridLayout(0, 1));

        frame.add(panel, BorderLayout.CENTER);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setTitle("Alisters Car Dealer System");
        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String args[]) {
        new GUI();
    }
}
```

Section B Question 1A

Code Segment:

```
public interface Asia {
    public abstract void Malaysia();
    public abstract void Singapore();
}

public class Main implements Asia {
    public void Malaysia() {
        System.out.println("Method Malaysia");
    }
    public void Singapore() {
        System.out.println("Method Singapore");
    }
    public static void main(String[] args) throws Exception {
        Main app = new Main();
        app.Malaysia();
        app.Singapore();
    }
}
```

The advantages and disadvantages of using an interface method as opposed to using an abstract method are as follows:

Advantage: Interface allows us to do multiple inheritance, while abstract method only allows one. A class can also implement multiple interfaces, but it can only extend one abstract class. Interface is also beneficial in cases where a class shares multiple behaviors with other classes.

Disadvantage: Interface has security issues. They do not have any access modifiers. Aside from that, interface cannot have its own constructor or destructors. So, this means that everything (including abstract methods) in interface is public, due to the low level of restrictions, and thus can cause data leakage. This is a serious security risk for using interface.

Section B Question 1B

Code Segment

```
public class App {
    public static <A> A advancedProgramming(A b) {
        return b;
    }

    public static Number advancedProgramming(Number kl) {
        return kl.intValue() + 2;
    }

    // Method accepting Integer parameter
    public static Integer advancedProgramming(Integer sg) {
        return sg + 1;
    }

    public static void main(String[] args) {
        Integer a = new Integer(1);
        String c = "Hello";
        Number b = new Integer(1);

        Number ZInteger = advancedProgramming(a);
        Number ZNumber = advancedProgramming(b);
        String ZString = advancedProgramming(c);

        System.out.println("advancedProgramming(a): " + ZInteger);
        System.out.println("advancedProgramming(b): " + ZNumber);
        System.out.println("advancedProgramming(c): " + ZString);
    }
}
```

Section B Question 1C

Essentially anything with final prefix is made in the intention to prevent alteration of it. For example, if we use “Final Method”, this means we cannot override the method. Or when we use “Final Class”, it means other subclasses cannot extend to this “Final Class”, and thus cannot inherit from “Final Class”. Essentially, using final prefix adds security to the code to prevent access or alteration of data.

Section B Question 2A

"IS-A" is a term that can be used to easily distinguish the relationship between two or more things. This term can be used to justify if an object is an instance of a certain class. Or whether a certain class is a superclass or a subclass of another class.

Essentially this term is a very powerful guide and a tool to accurately figure out the relationship/links between classes, especially when creating UML class diagrams. For example, a boy is a student, so the class of boy (superclass) can be related to the class of student (subclass), thus we can confirm that there is a relationship between them.

Section B Question 2B

Java may sometime require to have multiple inheritance. On a normal circumstance it is not possible to extend from multiple super classes or abstract classes.

This can usually be tackled in java by using interface. Since interface allows multiple implement and can allow classes to share multiple behaviors with other classes, it is relatively easy (compared to other methods in Java) to overcome the absence of multiple inheritance. The simplest way to do it using interface is by just implementing (and not extending) to the interface.

Section B Question 2C

Overloading and Overriding:

Similarity: Both overloading and overriding are methods of polymorphism. They both do the objective of having the same method being used in multiple ways.

Differences: While overloading works by having multiple different methods with the same method name, which makes the method work in two different ways (depending on the situation), overriding works by overwriting the previous method with new method, basically updating the definition/functionality of the method.

0340938
Alister Animesh Baroi

Section B Question 3

Involuntary Method

```java
import java.io.FileNotFoundException; // involuntary way
import java.io.IOException;
import java.io.FileInputStream;

public class Main {
    public static void main(String args[]) {
        FileInputStream myFile = null;
        try {
            myFile = new FileInputStream("/file.txt");
        } catch (FileNotFoundException e) {
            System.out.println("File not found");
        }
        int x;
        try {
            while ((x = myFile.read()) != -1) {
                System.out.print((char) x);
            }
            myFile.close();
        } catch (IOException e) {
            System.out.println("I/O error occurred: " + e);
        }
    }
}
```

Voluntary Method

```java
public class Main { // voluntary way
    public static void main(String args[]) {
        try {
            int index[] = { a, s, d, f, g };
            System.out.println(index[8]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Index not found");
        }
    }
}
```

Java exception handling needs to be voluntarily because Java we well equipped to terminate executions. But if we handle the exception, the java compiler/interpreter will skip just 1 line to proceed with the code execution, instead of directly terminating the whole process. So, this was, if there are some exceptions, handling it voluntarily will result in less program crashes and more stability.

0340938
Alister Animesh Baroi

Section B Question 4

```java
import java.io.*;
import java.io.Serializable;

public class Main implements Serializable {
    int age;
    String name;
    double height;

    public Main(String name, int age, double height) {
        this.name = name;
        this.age = age;
        this.height = height;
    }

    public static void main(String args[]) {
        try {
            Main info = new Main("Alister", 20, 5.8);
            FileOutputStream fout = new FileOutputStream("myinfo.txt");
            ObjectOutputStream out = new ObjectOutputStream(fout);
            out.writeObject(info);
            out.flush();
            out.close();
            System.out.println("Task Completed Successfully");
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```