Project 1: Lounge Airlines
Profit Maximization

CmpE 160, Introduction to Object Oriented Programming, Spring 2022

Instructor: Tuna Tuğcu
TA: Yiğit Yıldırım
SA: Bahadır Gezer
bahadir.gezer@boun.edu.tr

Due: **11.05.2022, 23.55**

1 Introduction

Modern aviation is a great way for safe, cost-effective, and fast transport for the public. The aviation industry has grown significantly over the past century from small propeller planes that operate whenever to vast fleets of modern aircraft with rigorous timetables. This growth was partly fueled by advancements in aircraft manufacturing and partly by computer-driven planning and pricing.

Price is the most important factor for acquiring success in the aviation industry. The modern traveler will prioritize prices over anything else. Not even brand equity is relevant when it comes to modern air travel. With websites where passengers can monitor prices for each route, each air carrier, time slot, and any other variable, airlines need to be very aware of their pricing to stay competitive. Specialized software and algorithms set aircraft schedules, analyze routes, price tickets, and manage resources.

In this project, you will try to implement an acceptable version of these systems that work for Lounge Airlines.

You will be the airline which has some aircraft. Like any well-functioning company, you, too, have a profit incentive. So, your task is simple: **Generate as much profit as possible.**

2 Class Hierarchy

There will be five main containers: “airline”, “airport”, “passenger”, “interfaces” and “executable”. Inside “airline,” there will be the “aircraft” container, and inside the “aircraft” container, there will be the “concrete” container. You should not create any additional java files.

You must design the accessibility of your code by proper encapsulation techniques. For example, some methods and fields will have predetermined access modifiers. Other than those, you should determine access modifiers in a way that removes unrelated or potentially harmful access within your code.

src:
- airline container:
  - Airline.java
  - aircraft container:
    - Aircraft.java
      - Abstract class, implements AircraftInterface
    - PassengerAircraft.java
      - Abstract class, extends Aircraft implements PassengerInterface
    - concrete container:
      - JetPassengerAircraft.java
        - Extends PassengerAircraft
      - RapidPassengerAircraft.java
        - Extends PassengerAircraft
      - WidebodyPassengerAircraft.java
        - Extends PassengerAircraft
      - PropPassengerAircraft.java
        - Extends PassengerAircraft
- airport container:
  - Airport.java
    - Abstract class
  - HubAirport.java
    - Extends Airport
  - MajorAirport.java
    - Extends Airport
  - RegionalAirport.java
    - Extends Airport
- passenger container:
  - Passenger.java

Abstract class

EconomyPassenger.java

Extends Passenger

BusinessPassenger.java

Extends Passenger

FirstClassPassenger.java

Extends Passenger

LuxuryPassenger.java

Extends Passenger

interfaces container:

AircraftInterface.java

PassengerInterface.java

executable container:

Main.java

3 Project Details

The implementation of this project has two parts. First, the functionality part is where you create all the classes and necessary methods. The second part is where you will try out different algorithms and interesting heuristics to maximize profits.

Since you are “the airline”, your executable should not interact with any method outside the Airline.java class (Except aircraft configuration functions). This class will be the gateway to the functional part of the code.

3.1 Airline:

This class is the only class that should be imported to the main. So, all your functions should meet here. First, you will be given a few basic functionalities. Then, you can build upon these to create methods suited to your needs.

You must not import aircraft classes into the main class, so there should not be any aircraft objects in the main class. Instead, all aircraft objects should be held in the airline class. Thus, when choosing which aircraft to use, you should give an integer, which should be the index of that aircraft in your aircrafts ArrayList.

Revenue and expenses will also be calculated inside the airline class. Therefore, you should hold both values separately and return the profit when requested.

Necessary fields:

- **int maxAircraftCount**
  Maximum number of aircrafts this airline can hold. Will be given as an input.
- **double operationalCost**
  Operational cost value for this airline. Will be given as an input.

Primitive Operations:

- **boolean fly(Airport toAirport, int aircraftIndex)**

  This method will let you fly the aircraft from its current airport to toAirport. aircraftIndex is the index at which the aircraft is stored in the aircrafts ArrayList. This method should return true if the flight operation is successful and return false if the flight operation cannot be completed.

  There is a running cost for every airline, which increases as the airline gets larger. This must be considered. A running cost of an airline is the operationalCost times the number of aircraft the airline has. Running cost should be added automatically to the expenses every time this method is called.

  A flight cannot happen if the plane cannot reach the destination airport because of range limitations, or the other airport is full. There are some edge cases for invalid flights. You should think of them yourself; they are very easy to figure out.

  If the aircraft can fly, there is a flight cost associated with every flight. This flight cost consists of the operational cost of the airplane, landing fee, and departure fee. Calculation of this cost will be explained in the aircraft and airport classes.

- **boolean loadPassenger(Passenger passenger, Airport airport, int aircraftIndex)**

  This method will let you load a passenger from a given airport to an aircraft of your choice. Passenger is the passenger to load; the airport is the airport where the passenger is, and aircraftIndex is the index of the aircraft to which the passenger will be loaded.

  This method should return true if the load operation is successful and return false if the load operation cannot be completed.

  A load operation cannot happen when the aircraft and the passenger are not in the same airport or the passenger cannot be loaded into the aircraft. A passenger cannot be loaded into an aircraft if the aircraft does not have seats for that passenger or if the aircraft exceeds the maximum weight limit with the addition of that passenger. Passenger weight calculation will be explained in the passenger class. Passengers cannot sit in seats of higher class. However, if they have to, they can sit in lower-class seats. Do not forget that the priority is always the seat of higher class when a passenger is assigned a seat.

  If a passenger can be loaded, the loading cost should be calculated and added to expenses. Therefore, every load operation has a loading cost associated with it. The calculation of this cost will be explained in the aircraft classes.

- **boolean unloadPassenger(Passenger passenger, int aircraftIndex)**

  This method will let you unload a passenger from a given aircraft. Passenger is the passenger to unload, aircraftIndex is the index of the aircraft which the passenger will be unloaded.

  This method should return true if the unload operation is successful and return false if the unload operation could not be completed.

  An unload operation cannot happen if the passenger cannot disembark at the aircraft's airport. A disembarkation check will be done in the passenger classes. Passengers can disembark if the airport they want to disembark at is a future destination that they want to visit. Passengers cannot go back to lower destinations in the destinations list.

  Ticket revenue is earned when a passenger disembarks; transfers and loading operations do not generate revenue. The ticket price calculation details will be given in the aircraft and passenger classes. Ticket price should be added to the revenue in this function.

- **Refueling functions:**

  Refueling operations should be done through this class. You can implement any refueling function you desire so long as you write the amount of fuel bought, from which airport for which aircraft to the output log. For example, you could have a refuel(int aircraftIndex, double fuel) or fulle(int aircraftIndex); it is up to you. Do not forget to change the weight of the aircraft when adding fuel. Fuel weight for a given fuel amount is calculated by multiplying the fuelWeight constant by the fuel amount. So, you can think of the fuel amount as a volume.

  Since fuel weight is a big part of the aircraft's overall weight, you might want to dump fuel; in this case, you do not get any refunds for dumped fuel. Any fuel dumping should also be written to the log.

- **Aircraft creation functions:**

  As stated earlier, all aircraft should be added using the airline, not by importing any aircraft into the main. Thus, you should have a function for adding every type of aircraft. Every aircraft created should be appended to the aircraft ArrayList. Your aircraft count should not exceed the maximum aircraft count given in the inputs.

- **Seat setting functions:**

  After creating the aircraft, you should specify the seat configuration you want on that aircraft. Seat setting and resetting details will be given in aircraft classes.

3.2 Passenger:

3.2.1 Passenger.java

- **public boolean board(int seatType)**

  Loads the passenger to an aircraft. Passenger class does not need to keep track of which aircraft is in. Instead, we need to define a new value called seatMultiplier; this value will be used when calculating the ticket price. For example, if the seat assigned is an economy seat, the seatMultiplier is 0.6. If the seat assigned is a business seat, the seatMultiplier is 1.2. Finally, if the seat assigned is a first-class seat, the seat multiplier is 3.2.

- **public double disembark(Airport airport, double aircraftTypeMultiplier)**

  Disembarks the passenger to the airport. Returns the ticket price. This method will do the necessary disembarkation operations, and it will be where ticket money is calculated and returned.

  If the airport is not a future destination, this method should return 0 and not perform any operation. If not, it should call calculateTicketPrice() and return this price. Before returning, it should set the airport and reset the necessary multipliers.

- **public boolean connection(int seatType)**

  Transfers the passenger to another plane. Connects the flight. If the new seat is an economy seat, the seatMultiplier should be multiplied by 0.6. If the new seat is a business seat, the seatMultiplier should be multiplied by 1.2. If the new seat is a first-class seat, the seatMultiplier should be multiplied by 3.2.

  We need to define a new value called connectionMultiplier; this value will be sued when calculating the ticket price. The connectionMultiplier should be "1" upon boarding since there is no connection yet. Each time connection() is called, connectionMultiplier should be multiplied by 0.8. This value will be used when calculating ticket prices.

- **abstract double calculateTicketPrice(Airport airport, double aircraftTypeMultiplier)**

  This method must be implemented in the child classes of Passenger.java; it is used in the disembark() method.
- **public abstract double landAircraft(Aircraft aircraft)**

  Does the necessary landing operations and returns the landing fee.

Necessary fields:

- **private final int ID**
  Unique ID of the passenger.
- **private final double weight**
  Weight of the passenger.
- **private final int baggageCount**

The number of pieces of baggage the passenger has. Used when calculating ticket price.

- **private ArrayList<Airport> destinations**
  ArrayList of destinations. Used when determining whether a passenger can disembark or not. Passengers can disembark at future destinations of this ArrayList. A future destination means a destination with a higher index. When a passenger disembarks, the list "shrinks"; now, the passenger can disembark in airports higher up or equal to the airport of disembarkation.

The ticket price is calculated when a passenger disembarks.

Each passenger will have an ID, weight, and baggageCount. These will be provided for you in the inputs. There will also be a list of destinations the passenger wants to visit.

Passenger.java will be an abstract class. Since the voluntary action of the consumer determines the final price in a free market, ticket prices will be calculated in the passenger class. Ticket revenue will be collected only when the passenger disembarks. Connections and boardings do not generate any revenue.

3.2.2 EconomyPassenger.java

- **protected double calculateTicketPrice(Airport toAirport, double aircraftTypeMultiplier)**

  This method calculates and returns the ticket price. It is calculated with the help of a few values.

  First, we have the airport multiplier. This depends on the airport at which the passenger disembarked previously and to toAirport. Suppose the previous disembarkation airport is a hub airport. If the passenger goes to another hub airport, the airport multiplier is 0.5, or if the other is a major airport, the airport multiplier is 0.7, or if the other airport is a regional airport, the airport multiplier is 1. If the previous disembarkation airport was a major airport and the passenger goes to another major airport, the airport multiplier is 0.8, or if the other airport is a hub airport, the airport multiplier is 0.6, or if the other airport is a regional airport, then the airport multiplier is 1.8. If the previous disembarkation airport was a regional airport and the passenger goes to another regional airport, the airport multiplier is 3.0, or if the other airport is a hub airport, then the airport multiplier is 0.9, or if the other airport is a major airport, then the airport multiplier is 1.6.

  Then we have the passenger multiplier; for this type of passenger, this value is 0.6. Now we can calculate the ticket price. You need to multiply the distance between the airport of previous disembarkation and the toAirport, aircraftTypeMultiplier, connectionMultiplier, airport multiplier, and passenger multiplier. For the last step, you take the ticket price above and increase it by 5% for every piece of baggage the passenger has. This is your final ticket price.

3.2.3 BusinessPassenger.java

- **protected double calculateTicketPrice(Airport toAirport, double aircraftTypeMultiplier)**

  This method calculates and returns the ticket price. It is calculated with the help of a few values.

  First, we have the airport multiplier. This value is the same as the one for economy passengers.

  Then we have the passenger multiplier; for this type of passenger, this value is 1.2. The ticket price calculation is the same as we did for economy passengers; we just needed new constants.

3.2.4 FirstClassPassenger.java

- **protected double calculateTicketPrice(Airport toAirport, double aircraftTypeMultiplier)**

  This method calculates and returns the ticket price. It is calculated with the help of a few values.

  First, we have the airport multiplier. This value is the same as the one for business passengers.

  Then we have the passenger multiplier; for this type of passenger, this value is 3.2. The ticket price calculation is the same as we did for business passengers; we just needed new constants.

3.2.5 LuxuryPassenger.java

- **protected double calculateTicketPrice(Airport toAirport, double aircraftTypeMultiplier)**

  This method calculates and returns the ticket price. It is calculated with the help of a few values.

  First, we have the airport multiplier. This value is the same as the one for first-class passengers.

  Then we have the passenger multiplier; for this type of passenger, this value is 15. The ticket price calculation is the same as we did for first-class passengers; we just needed new constants.

3.3 Airport:

3.3.1 Airport.java

- **public abstract double departAircraft(Aircraft aircraft)**

  Does the necessary departure operations and returns the departure fee.

- **public abstract double landAircraft(Aircraft aircraft)**

  Does the necessary landing operations and returns the landing fee.

Necessary fields:

- **private final int ID**

  Unique ID of the airport.

- **private final double x, y**
  Coordinates of the airport. Used when calculating distance.
- **protected double fuelCost**
  Price of fuel in this airport.
- **protected double operationFee**
  The fee paid to this airport for certain operations.
- **protected int aircraftCapacity**
  The maximum number of aircraft this airport can hold.

You can add any helper functions of your liking to this class. Some helpful functions are: boolean equals(Airport other), boolean isFull(), double getDistance(Airport airport), void addPassenger(Passenger passenger), void removePassenger(Passenger passenger)…

Passenger objects should be held either in airport objects or aircraft objects. You can use any collection you like to store passenger objects.

3.3.2 HubAirport.java

- **public double departAircraft(Aircraft aircraft)**

  This method should do the necessary departure operations. Returns the departure fee.

  To calculate the departure fee, we need to define two values. The first one is called the fullness coefficient. The fullness coefficient is obtained by multiplying 0.6 with e -the mathematical constant- raised to the power of aircraft ratio. The aircraft ratio for an airport is the total number of aircraft divided by the total aircraft capacity of that airport. The second value we need to define is the aircraft weight ratio; it is simply the ratio of weight to maxWeight for the aircraft given in the parameters. Therefore, to calculate the departure fee, you must multiply operationFee, aircraft weight ratio, fullness coefficient, and a constant value of 0.7.

- **public abstract double landAircraft(Aircraft aircraft)**

  This method should do the necessary landing operations. Returns the landing fee.

  To calculate the landing fee, we need to define two values. The first one is called the fullness coefficient, the same as the one used in departAircraft(). The second value is also the same as departAircraft(). Therefore, to calculate the departure fee, you must multiply operationFee, aircraft weight ratio, fullness coefficient, and a constant value of 0.8.

3.3.3 MajorAirport.java

- **public double departAircraft(Aircraft aircraft)**

  This method should do the necessary departure operations. Returns the departure fee.

  To calculate the departure fee, we need to define two values. The first one is called the fullness coefficient. The fullness coefficient is obtained by

multiplying 0.6 with -the mathematical constant- raised to the power of aircraft ratio. The aircraft ratio for an airport is the total number of aircraft divided by the total aircraft capacity of that airport. The second value we need to define is the aircraft weight ratio; it is simply the ratio of weight to maxWeight for the aircraft given in the parameters. Therefore, to calculate the departure fee, you must multiply operationFee, aircraft weight ratio, fullness coefficient, and a constant value of 0.9.

- **public abstract double landAircraft(Aircraft aircraft)**

  This method should do the necessary landing operations. Returns the landing fee.

  To calculate the landing fee, we need to define two values. The first one is called the fullness coefficient, which is the same as the one used in departAircraft(). The second value is also the same as departAircraft(). Therefore, you must multiply operationFee, aircraft weight ratio, and fullness coefficient to calculate the departure fee.

3.3.4 RegionalAirport.java

- **public double departAircraft(Aircraft aircraft)**

  This method should do the necessary departure operations. Returns the departure fee.

  To calculate the departure fee, we need to define two values. The first one is called the fullness coefficient. The fullness coefficient is obtained by multiplying 0.6 with -the mathematical constant- raised to the power of aircraft ratio. The aircraft ratio for an airport is the total number of aircraft divided by the total aircraft capacity of that airport. The second value we need to define is the aircraft weight ratio; it is simply the ratio of weight to maxWeight for the aircraft given in the parameters. Therefore, to calculate the departure fee, you must multiply operationFee, aircraft weight ratio, fullness coefficient, and a constant value of 1.2.

- **public abstract double landAircraft(Aircraft aircraft)**

  This method should do the necessary landing operations. Returns the landing fee.

  To calculate the landing fee, we need to define two values. The first one is called the fullness coefficient, which is the same as the one used in departAircraft(). The second value is also the same as departAircraft(). Therefore, to calculate the departure fee, you must multiply operationFee, aircraft weight ratio, fullness coefficient, and a constant value of 1,3.

3.4 Aircraft:

Like the airline, you have access to different aircraft, each with its stats. Therefore, you must choose the correct aircraft with the correct seat configuration for the correct passenger base and routes.

3.4.1 Aircraft.java

You can implement other helper functions of your liking.

- protected double fly(Airport toAirport)

  This function flies the aircraft to toAirport. It should do the necessary fuel calculations using getFuelConsumption(), which will be explained later. It should also calculate the flight cost and return this value.

  Flight cost consists of the operational cost of the aircraft, the departure fee, and the landing fee. Departure and landing fees will be explained in the airport classes. Operational cost is different for each type of aircraft, and it is based on the fullness of the aircraft and the distance of the flight. Full details of this calculation will be given in the concrete aircraft classes.

  Since fly() always returns the flight cost, you should implement a boolean function for fly(), which checks if a flight can happen or not. This boolean function and fly() should be used together to ensure that any invalid operations do not happen.

Necessary fields:

- **protected Airport currentAirport**
  The airport which the plane is currently in.
- **protected double weight**
  This is the current weight of the aircraft. Every aircraft has an empty weight, so this will be given different starting values depending on the aircraft. This value should not exceed maxWeight.
- **protected double maxWeight**
  This is the maximum allowable weight of the aircraft. The aircraft should not exceed this weight.
- **protected double fuelWeight**
  This field is equal to 0.7; it is the constant we use to convert fuel volume to fuel weight.
- **protected double fuel**
  This field holds the amount of fuel the aircraft has at any moment. It should not exceed fuelCapacity. fuelCapacity and this field are both volumes, so you should multiply fuel amount with fuelWeight when you want to get the weight for some amount of fuel. This value is zero upon initialization.
- **protected double fuelCapacity**
  Fuel capacity of the airplane. It is different and fixed for every type of aircraft.

Refueling functions:

These functions should add fuel to the aircraft. You can choose what kind of refueling function you want to implement. However, all refueling functions must add the appropriate fuel weight along with the fuel. Every refueling function must also return a double, which should be fuel cost. Fuel cost is calculated by multiplying the fuel amount by the airport's fuel cost which the aircraft is refueling at. These functions will be used by the matching caller functions of the airline.

3.4.2 PassengerAircraft.java

This class will hold functions related to passenger operations for aircraft. Passenger objects must be held in this class if they are loaded in. Passenger objects should switch between airport and aircraft objects; they should not be stored anywhere else. You can use any collection you like to store passenger objects.

- **double loadPassenger(Passenger passenger)**

  This method loads the passenger to the appropriate seat. It should return the loading fee. Loading fees are based on the operationFee, which is a fee specific to each type of aircraft. If the loading operation cannot be completed return value should only be the operationFee. If the loading operation is completed, the complete loading fee is returned.

  Loading fee is calculated by multiplying operationFee with aircraftTypeMultiplier, the detail of which will be given in the concrete aircraft classes, and a constant. This constant depends on the type of seat selected for that passenger. If the seat is a first-class seat, the constant is 2.5; if the seat is a business seat, the constant is 1.5; and if the seat is an economy seat, the constant is 1.2. Remember that passengers cannot sit in seats with a higher class than their class. LuxuryPassengers act the same way as FirstClassPassengers.

  If you do not want to pay the operationFee every time you do an invalid loading operation, you can create a boolean method that will check if the operation is valid. If you use both methods together, you can avoid unnecessary expenses.

- **double unloadPassenger(Passenger passenger)**

  This method unloads the passenger. A passenger can be unloaded if it can disembark at the aircraft's airport. This is where the ticket revenue will be calculated and collected. Ticket price is calculated by multiplying disembarkation revenue with the seat constant. Disembarkation revenue will be calculated in the passenger classes.

  If the passenger was seated in economy, the seat constant is 1.0; if the passenger was seated in business, the seat constant is 2.8; if the passenger was seated in first-class, the seat constant is 7.5.

  If the passenger cannot disembark, you should return the operationFee. So, this method can return both revenue and expense, do not forget to consider this.

- **double transferPassenger(Passenger passenger, PassengerAircraft toAircraft)**

  This method transfers the passenger from the current aircraft to the toAircraft. If the passenger cannot disembark in an airport, you can use this method. When moving a passenger, it acts as a bypass to the unloadPassenger() operation.

  The implementation of this method is very similar to the loadPassenger operation; the only difference is this operation is between aircraft rather than between an aircraft and an airport.

  If the transfer operation is invalid, you should return the operationFee as an expense. If the transfer operation is valid, then you should return the loading fee. This loading fee is what you get when you use the loadPassenger() method above.

So, as a small summary, you can load passengers anytime. If you loaded them incorrectly, you could disembark them in the same airport, and the revenue function will not generate any revenue. Still, you will have to pay some fee for the loading and unloading operations. When you go to a new airport, if that airport is a future destination of the passenger, then the passenger can disembark and leave the aircraft. However, if the passenger cannot disembark at that airport, you have two options. You can fly the aircraft to an airport that the passenger can disembark in, or you can transfer the passenger to another aircraft in the same airport.

Necessary fields:

- **protected double floorArea**
  The total floor area of the aircraft, this field is different for each type of aircraft, so it should be initialized in the concrete aircraft classes.
- **private int economySeats, businessSeats, firstClassSeats**
  Count of seats assigned for this aircraft for each seat type.
- **private int occupiedEconomySeats, occupiedBusinessSeats, occupiedFirstClassSeats**
  Count of seats that are occupied for each seat type.

Seat operations:

Operations related to seat configuration will be done in this class. Each type of aircraft has a predetermined floor area, and each type of seat also has a predetermined area. You must make sure that the total area of seats assigned does not exceed the floor area of the plane. To calculate the occupied area, you must multiply each seat count with the respective seat area and add them.

Economy seats have an area of 1, business seats have an area of 3, and first-class seats have an area of 8.

You can set and reset seats as much as you want; however, you should ensure that the seat you are deleting is empty. If not, you should unload that passenger first.

3.4.3 PropPassengerAircraft.java

This class represents a small propeller aircraft designed for short routes and minor destinations.

- **getFlightCost(Airport toAirport)**

  This method calculates and returns the flight cost. It is used in the fly() function of Aircraft.java. The flight cost comprises three costs: landing cost, departure cost, and flight operation cost.

  We need three values to calculate flight operation cost: fullness, distance, and a constant. Distance is the distance between the current airport and the toAirport. Fullness is the ratio of occupied seats to all seats. The constant for this type of aircraft is 0.1. Flight operation cost is the multiplication of these three values. Departure and landing cost is explained in Airport classes.

- **getFuelConsumption(double distance)**

  This method calculates the fuel consumption of the aircraft for a given distance; it should return the total fuel needed for the given distance.

  Real aircraft have a bathtub curve for fuel consumption. This project uses a similar curve.

  To calculate fuel consumption, we must define two values. The first one is the distance ratio. The distance ratio for this type of aircraft is distance -given as a parameter- divided by 2000. The second value is called the bathtub coefficient. This value is calculated by putting the distance ratio in the curve ($y = 25.9324\ x^4 - 50.5633\ x^3 + 35.0554\ x^2 - 9.90346\ x + 1.97413$). Fuel consumption has two parts: a takeoff and a cruise part. Fuel consumption is the addition of these two values. Takeoff fuel consumption is weight times a constant divided by fuelWeight. We must divide by fuelWeight because we are calculating the volume of fuel. The constant for this type of aircraft is 0.08. Cruise fuel consumption is fuelConsumption times bathtub coefficient times distance.

Necessary fields:

- **protected double weight**
  The actual weight of the aircraft. Declared in Aircraft.java, the initial value -which corresponds to the empty weight- is 14000. Weight should never drop below 14000 if the program runs correctly.
- **protected double maxWeight**
  The maximum weight this type of aircraft can have, Declared in Aircraft.java, is equal to 23000. Weight should never exceed maxWeight.
- **protected double floorArea**
  Declared in PassengerAircraft.java, equal to 60. This field is used in seat area calculations. The total seat area of your aircraft should never exceed floorArea.

- **protected double fuelCapacity**
  Fuel capacity of the aircraft. Declared in Aircraft.java, equal to 6000. The fuel inside the aircraft should never exceed fuelCapacity. You can consider this value as fuel volume; fuel should be multiplied with fuelWeight to get the weight of a given fuel volume.
- **protected double fuelConsumption**
  Fuel consumption of the aircraft will be used in the getFuelConsumption() method and declared in Aircraft.java, equal to 0.6.
- **protected double aircraftTypeMultiplier**
  A constant to balance the usage of different aircraft. This field was used when loading the passenger. Declared in Aircraft.java, equal to 0.9.

3.4.4 WidebodyPassengerAircraft.java

This class represents a large jet aircraft designed for very long routes and major destinations. It represents the Airbus A330.

- **getFlightCost(Airport toAirport)**
  This method calculates and returns the flight cost. It is used in the fly() function of Aircraft.java. Flight cost comprises three costs: landing cost, departure cost, and flight operation cost.
  We need three values to calculate flight operation cost: fullness, distance, and a constant. Distance is the distance between the current airport and toAirport. Fullness is the ratio of occupied seats to all seats. The constant for this type of aircraft is 0.15. Flight operation cost is the multiplication of these three values. Departure and landing cost is explained in Airport classes.

- **getFuelConsumption(double distance)**
  This method calculates the fuel consumption of the aircraft for a given distance; it should return the total fuel needed for the given distance.
  Real aircraft have a bathtub curve for fuel consumption. This project uses a similar curve.
  To calculate fuel consumption, we must define two values. The first one is the distance ratio. The distance ratio for this type of aircraft is distance -given as a parameter- divided by 14000. The second value is called the bathtub coefficient. This value is calculated by putting the distance ratio in the curve ($y = 25.9324\ x^4 - 50.5633\ x^3 + 35.0554\ x^2 - 9.90346\ x + 1.97413$). Fuel consumption has two parts: a takeoff and a cruise part. Fuel consumption is the addition of these two values. Takeoff fuel consumption is weight times a constant divided by fuelWeight. We must divide by fuelWeight because we are calculating the volume of fuel. The constant for this type of aircraft is 0.1. Cruise fuel consumption is fuelConsumption times bathtub coefficient times distance.

Necessary fields:

- **protected double weight**
  The actual weight of the aircraft. Declared in Aircraft.java, the initial value -which corresponds to the empty weight- is 135000. Weight should never drop below 135000 if the program runs correctly.
- **protected double maxWeight**
  The maximum weight this type of aircraft can have, Declared in Aircraft.java, equals 250000. Weight should never exceed maxWeight.
- **protected double floorArea**
  Declared in PassengerAircraft.java, equal to 450. This field is used in seat area calculations. The total seat area of your aircraft should never exceed floorArea.
- **protected double fuelCapacity**
  Fuel capacity of the aircraft. It was declared in Aircraft.java, equal to 140000. The fuel inside the aircraft should never exceed fuelCapacity. You can consider this value as fuel volume; fuel should be multiplied with fuelWeight to get the weight of a given fuel volume.
- **protected double fuelConsumption**
  Fuel consumption of the aircraft will be used in the getFuelConsumption() method, declared in Aircraft.java, equal to 3.0.
- **protected double aircraftTypeMultiplier**
  A constant to balance the usage of different aircraft. This field was used when loading the passenger. Declared in Aircraft.java, equal to 0.7.

3.4.5 RapidPassengerAircraft.java

This class represents a medium-sized aircraft designed for medium-range routes and major destinations. It represents the Concorde, which was a supersonic passenger aircraft.

- **getFlightCost(Airport toAirport)**
  This method calculates and returns the flight cost. It is used in the fly() function of Aircraft.java. Flight cost comprises three costs: landing cost, departure cost, and flight operation cost.
  We need three values to calculate flight operation cost: fullness, distance, and a constant. Distance is the distance between the current airport and toAirport. Fullness is the ratio of occupied seats to all seats. The constant for this type of aircraft is 0.2. Flight operation cost is the multiplication of these three values. Departure and landing cost is explained in Airport classes.

- **getFuelConsumption(double distance)**
  This method calculates the fuel consumption of the aircraft for a given distance; it should return the total fuel needed for the given distance.
  Real aircraft have a bathtub curve for fuel consumption. This project uses a similar curve.

To calculate fuel consumption, we must define two values. The first one is the distance ratio. The distance ratio for this type of aircraft is distance -given as a parameter- divided by 7000. The second value is called the bathtub coefficient. This value is calculated by putting the distance ratio in the curve ($y = 25.9324\ x^4 - 50.5633\ x^3 + 35.0554\ x^2 - 9.90346\ x + 1.97413$). Fuel consumption has two parts: a takeoff and a cruise part. Fuel consumption is the addition of these two values. Takeoff fuel consumption is weight times a constant divided by fuelWeight. We must divide by fuelWeight because we are calculating the volume of fuel. The constant for this type of aircraft is 0.1. Cruise fuel consumption is fuelConsumption times bathtub coefficient times distance.

Necessary fields:

- **protected double weight**
  The actual weight of the aircraft. Declared in Aircraft.java, the initial value -which corresponds to the empty weight- is 80000. Weight should never drop below 80000 if the program runs correctly.
- **protected double maxWeight**
  The maximum weight this type of aircraft can have, Declared in Aircraft.java, equals 185000. Weight should never exceed maxWeight.
- **protected double floorArea**
  Declared in PassengerAircraft.java, equal to 120. This field is used in seat area calculations. The total seat area of your aircraft should never exceed floorArea.
- **protected double fuelCapacity**
  Fuel capacity of the aircraft. Declared in Aircraft.java, equal to 120000. The fuel inside the aircraft should never exceed fuelCapacity. You can consider this value as fuel volume; fuel should be multiplied with fuelWeight to get the weight of a given fuel volume.
- **protected double fuelConsumption**
  Fuel consumption of the aircraft will be used in the getFuelConsumption() method and declared in Aircraft.java, equal to 5.3.
- **protected double aircraftTypeMultiplier**
  A constant to balance the usage of different aircraft. For example, this field was used when loading the passenger. Declared in Aircraft.java, equal to 1.9.

3.4.6 JetPassengerAircraft.java

This class represents a private jet designed for short to medium-range routes. It has a small floor area, so it should be used for squeezing the maximum amount of profit from luxury passengers.

- **getFlightCost(Airport toAirport)**

This method calculates and returns the flight cost. It is used in the fly() function of Aircraft.java. Flight cost comprises three costs: landing, departure, and flight operation costs.

We need three values to calculate flight operation cost: fullness, distance, and a constant. Distance is the distance between the current airport and toAirport. Fullness is the ratio of occupied seats to all seats. The constant for this type of aircraft is 0.08. Flight operation cost is the multiplication of these three values. Departure and landing cost is explained in Airport classes.

- **getFuelConsumption(double distance)**

  This method calculates the fuel consumption of the aircraft for a given distance; it should return the total fuel needed for the given distance.

  Real aircraft have a bathtub curve for fuel consumption. This project uses a similar curve.

  To calculate fuel consumption, we must define two values. The first one is the distance ratio. The distance ratio for this type of aircraft is distance -given as a parameter- divided by 5000. The second value is called the bathtub coefficient. This value is calculated by putting the distance ratio in the curve ($y = 25.9324\ x^4 - 50.5633\ x^3 + 35.0554\ x^2 - 9.90346\ x + 1.97413$). Fuel consumption has two parts: a takeoff and a cruise part. Fuel consumption is the addition of these two values. Takeoff fuel consumption is weight times a constant divided by fuelWeight. We must divide by fuelWeight because we are calculating the volume of fuel. The constant for this type of aircraft is 0.1. Cruise fuel consumption is fuelConsumption times bathtub coefficient times distance.

Necessary fields:

- **protected double weight**
  The actual weight of the aircraft. Declared in Aircraft.java, the initial value -which corresponds to the empty weight- is 10000. Weight should never drop below 10000 if the program runs correctly.
- **protected double maxWeight**
  The maximum weight this type of aircraft can have, Declared in Aircraft.java, equals 18000. Weight should never exceed maxWeight.
- **protected double floorArea**
  Declared in PassengerAircraft.java, equal to 30. This field is used in seat area calculations. The total seat area of your aircraft should never exceed floorArea.
- **protected double fuelCapacity**
  Fuel capacity of the aircraft. Declared in Aircraft.java, equal to 10000. The fuel inside the aircraft should never exceed fuelCapacity. You can consider this value as fuel volume; fuel should be multiplied with fuelWeight to get the weight of a given fuel volume.
- **protected double fuelConsumption**

Fuel consumption of the aircraft will be used in the getFuelConsumption() method and declared in Aircraft.java, equal to 0.7.

- **protected double aircraftTypeMultiplier**
  A constant to balance the usage of different aircraft. This field was used when loading the passenger. Declared in Aircraft.java, equal to 5.

3.5 Interfaces:

3.5.1 AircraftInterface

Methods this interface must have, you can add more methods if you want to.

- **double fly(Airport toAirport)**
- **boolean addFuel(double fuel)**
- **boolean fillUp() OR boolean fulle()**
  Refuels the aircraft to its full capacity.
- **boolean hasFuel(double fuel)**
  Checks if the aircraft has the specified amount of fuel.
- **double getWeightRatio()**
  Returns the ratio of weight to maximum weight.

3.5.2 PassengerInterface

Methods this interface must have, you can add more methods if you want to.

- **double transferPassenger(Passenger passenger, PassengerAircraft toAircraft);**
- **double loadPassenger(Passenger passenger);**
- **double unloadPassenger(Passenger passenger);**
- **boolean isFull();**
  Checks whether the aircraft is full or not
- **boolean isFull(int seatType);**
  Checks whether a certain seat type is full or not.
- **boolean isEmpty();**
  Checks whether the aircraft is empty or not.
- **public double getAvailableWeight();**
  Returns the leftover weight capacity of the aircraft.
- **public boolean setSeats(int economy, int business, int firstClass);**
- **public boolean setAllEconomy();**
  Sets every seat to economy.
- **public boolean setAllBusiness();**
  Sets every seat to business.
- **public boolean setAllFirstClass();**
  Sets every seat to first class.
- **public boolean setRemainingEconomy();**
  Does not change previously set seats, sets the remaining to economy.
- **public boolean setRemainingBusiness();**
  Does not change previously set seats, sets the remaining to business.

- **public boolean setRemainingFirstClass();**
  Does not change previously set seats, sets the remaining to first class.
- **public double getFullness();**
  Returns the ratio of occupied seats to all seats.

4 Input & Output Format

4.1 Input

Input will be given as a text file. The file will be given as the first command-line argument. Input format is given below. Details on how inputs are generated will be given later.

The first line of the input has ***M*** maximum number of aircraft allowed for the airline, followed by ***A*** number of airports, and ***P*** number of passengers. The following ***A*** lines will be airports and the next ***P*** lines will be passengers.

Each airport line consists of $ID_{airport}$***, X, Y, fuelCost, operationFee, aircraftCapacity*** in this order, all separated by whitespaces. ***ID*** is the unique airport ID. ***X*** and ***Y*** are the coordinates of the airport. The other values correspond to the fields specified with the same name in the airport class.

Each passenger line consists of $ID_{passenger}$***, weight, baggageCount,*** followed by a list of integers in this order, all separated by whitespaces. ***ID*** is the unique passenger ID. The other values correspond to the fields specified with the same name in the passenger class. The following integers are a list of airport IDs. These generate the destinations ArrayList for the passenger. Do not change the input order for airport IDs when putting them in the ArrayList.

Constraints:

$1 \leq M, A \leq 10^5$

$0 \leq P \leq 10^9$

$-10^9 \leq X, Y \leq 10^9$

$40 \leq weight \leq 200$

$0 \leq baggageCount \leq 10$

$0 \leq fuelCost, operationFee, aircraftCapacity \leq 1000$

$100000 \leq ID_{airport} \leq 999999$

$10^9 \leq ID_{passenger} \leq 10^{10} - 1$

4.2 Output

Since you determine what operations are done and in what order, there is no single output file for this project. You will output a log for each operation you do, and in

the last line you will write the total revenue of your airline. The output file will be given as the second command line argument when running your program.

Please do not forget to output all the information. Otherwise, I will not be able to confirm your revenue output, and you will not get any points.

All double values must be written without any formatting.

Output log format:

- ### Aircraft Creation:

  “0 <initial_airport> <aircraft_type>”

  This line must be written when an aircraft is added to the airline.

  <initial_airport> : ID of the airport the aircraft will start at.
  <aircraft_type> : type value of the aircraft. This value should be 0 for PropPassengerAircraft, 1 for WidebodyPassengerAircraft, 2 for RapidPassengerAircraft, 3 for JetPassengerAircraft.

- ### Flight Operation:

  “1 <destination_airport> <aircraft>”

  This line must be written when an airplane flies to another airport.

  <destination_airport> : ID of the airport which the plane flies to.
  <aircraft> : ID of the aircraft which flies to another airport. Aircraft IDs depend on their order of creation. In other words, it is their index in the aircrafts ArrayList.

- ### Seat Assignment:

  “2 <aircraft> <economy> <business> <first_class>”

  This line must be written when an aircraft configures its seats.

  <aircraft> : ID of the aircraft which does the seat assignment.
  <economy> : Number of economy seats assigned to the aircraft.
  <business> : Number of business seats assigned to the aircraft.
  <first_class> : Number of first-class seats assigned to the aircraft.

- ### Fuel Loading:

  “3 <aircraft> <fuel_amount>”

This line must be written when an aircraft loads fuel. If you plan on dumping fuel, give the <fuel_amount> with a negative sign in front of it.

<aircraft> : ID of the aircraft which the fuel is loaded onto.
<fuel_amount> : Amount of fuel that is loaded. This a volume not the weight.

- **Passenger Loading:**
  “4 <passenger> <aircraft> <airport>”

  This line must be written when a passenger is loaded.

  <passenger> : ID of the passenger.
  <aircraft> : ID of the aircraft the passenger is loaded onto.
  <airport> : ID of the airport the passenger is loaded from.
  <seat> : Type of seat the passenger is seated in. 0 for economy, 1 for business, 2 for first class.

- **Passenger Unloading:**
  “5 <passenger> <aircraft> <airport>

  This line must be written when a passenger is unloaded

  <passenger> : ID of the passenger.
  <aircraft> : ID of the aircraft the passenger is unloaded from.
  <airport> : ID of the airport the passenger is unloaded to.

- **Passenger Transfer:**
  “6 <passenger> <from_aircraft> <to_aircraft> <airport>

  This line must be written when a passenger is transferred.

  <passenger> : ID of the passenger.
  <from_aircraft> : ID of the aircraft the passenger is transferred from.
  <to_aircraft> : ID of the aircraft the passenger is transferred to.
  <airport> : ID of the airport the passenger is transferred at.

5 Submission

Submit your codes via TeachingCodes plugin before the deadline. You can submit multiple times as you proceed with the project. Note that only your last submission will be graded. We will apply a penalty for late submissions as explained in the next section.

6 Grading

The grading of this project is based on the automatic compilation and run and the success of your code in test cases. If your code compiles and runs with no error, then you will get 10/100, implementing this project using proper object oriented design principles will account for 10/100, Javadoc documentation of Airline.java will account for 10/100, test cases will account for 70/100.

There will be bonus points. We will run every program and take every total revenue. The one program with the maximum total revenue will get 10 extra points and the one with the least revenue will not get any points. Every program between the min and max will be given some grade based on the distribution of total revenues.

Late submission: If you submit within 1 day after the deadline, your grade will be calculated over 100 and then multiplied by 0.75. If you submit within the 2nd day after the deadline, your grade will be multiplied by 0.5. We will not accept submissions after 2 days.

7 Warnings

- Any revenue generated or fee paid should be written to the output. You will not get any points if your output and operation log do not match.
- This is an individual project.
- All source codes are checked automatically for similarity with other submissions. Make sure you write and submit your own code. Any sign of cheating will be penalized.
- Make sure you document your code with necessary inline comments and use meaningful variable names. Do not over-comment or make your variable names

unnecessarily long. This is especially important in the main class. These will be taken into account for partial grading.

- Please do not forget that providing the proper visibility and accessibility is important. You should not implement everything as public. Use proper access modifiers.

- You need to document the Airline.java class with proper Javadoc style documentation.

- If you want to implement a method for every child, ensure that the parent class enforces the implementation for that method.

- Every method must be generated in the appropriate class. Do not pass down unnecessary fields in every method.

- Please direct all questions to the email address given in the first page.

- I suggest you start the project right away, do not leave it to the last week.