ASSIGNMENT 3: Polymorphism

COMP 1020 Winter 2022

Due Date: April 8, 2022 11:59 pm

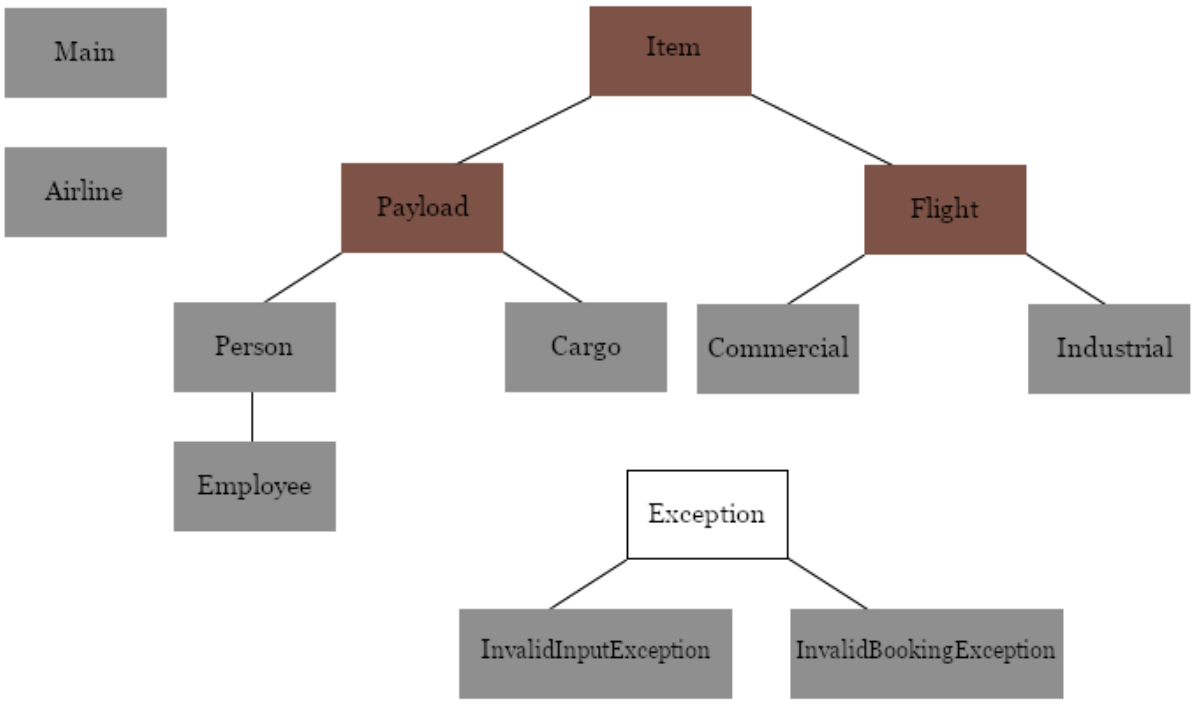
Instructions

- You must complete the "**Blanket Honesty Declaration**" checklist on the course website before you can submit any assignment. You will not be able to see the dropbox if you have not accepted the Honesty Declaration.
- Only submit the **java files**. Do ***not*** submit any other files unless otherwise instructed.
- To submit the assignment, upload the specified files to the **Assignment 3** folder on the course website.
- Assignments must follow the **programming standards** document published on UMLearn.
- After the due date and time, assignments may be submitted but will be subject to a late penalty. Please see the ROASS document published on UMLearn for the course policy for late submissions.
- If you make multiple submissions, only the **most recent version** will be marked.
- These assignments are your chance to learn the material for the exams. Code your assignments independently. We use software to compare all submitted assignments to each other and **pursue academic dishonesty vigorously**.
- Your Java programs must compile and run upon download, without requiring any modifications.

Assignment Overview

You are the database administrator for a small airline. You have the task of handling bookings for the airline's planes, both commercially and industrially. You will have to build a program that reads and processes instructions from a text file, making changes to the system as necessary and printing out error messages when problems occur.

You will implement a set of related **classes of objects**, as seen in the hierarchy below:



- ***Main*** will contain a main method and reads in data files. It will create an instance of the Airline class.
- ***Airline*** is your central class that has ***Item***s that are managed to run an airline.
- ***Item, Payload,*** and ***Flight*** are **abstract** classes.
- ***Exception*** is already defined, so you will only implement its children.

ASSIGNMENT 3: Polymorphism

COMP 1020 Winter 2022

Download the provided .java files for Airline, Item, Payload, Person, Employee, Cargo, InvalidInputException, and InvalidBookingException. These are starting points to help you on your way. You may write your files entirely on your own, or use these files. More code will be added to these files later, but you just need them to successfully compile Phase 1 and 2.

Phase 1: Main

First, implement the Main class. This class will run a simulation of an Airline. To do this, it will read a list of commands from a file, do some parsing of the command, and update an Airline object given the issued command. It should contain:

- String constant **FILE_NAME** containing the name of the text file you want to read
- A **public static void main(String[] args)** method, which will:
    - create an instance of **Airline** using the default constructor, and
    - call the method **readInputFile**, passing the name of the file **FILE_NAME** and the created Airline object as parameters

- Method **public static void readInputFile(String fileName, Airline airline)**, which takes the file name and an Airline object as parameters. This method should:
    - open the file for reading
    - read it line by line
    - Process each line, turning it into an array. The String "The lazy dog" should be split into an array of 3 Strings that look like this ["The", "lazy", "dog"].
    - Then, call the method **parseCommand** or **processComment** to process the array, passing the String array and Airline as parameters (see below). If **parseCommand** throws an exception, it should be caught by the readInputFile method. When that happens, print the message returned by the exception, and move on to the next line in the file. File exceptions will be tested, but **not** as rigorously as they were for Assignment 2. If the current line starts with an asterisk (*), call **processComment** instead of **parseCommand**.

- Method **public static void parseCommand(String [] tokens, Airline air) throws InvalidInputException,** which receives one line in the file as an array of Strings, and an Airline object. It determines which type of command the line contains.
    - The first token (tokens[0]) determines the type of the command.
    - Once the command is identified, you should call the specific method that handles the type of command. Each possible command, once identified, will be handled by a specific method, which is discussed in Phase 4. For phase 1, all input will throw an exception.
    - If parseCommand sees a command it does not know, it should throw an **InvalidInputException**.
- A method **public static void processComment(String data)**, which will handle a comment command. If the first token contains the **"*"** symbol, it is a comment – you do not need check that that **data** starts with *. The * symbol will always be followed by at least 1 space character. Print the rest of the String as it appears in the input text file, starting at the first non-space character. The following is the expected output when you run the program with the TestPhase1.txt file provided.

Expected Output:

```
This is a comment.
This is another comment.
InvalidInputException: Command not found
This is a third comment.
```

Phase 2: Flight

The software is designed to plan a flight, and must create Payloads, and attach Payloads to Flights. The flights will be checked to ensure that takeoff is possible. The first step in this process is to implement the basics of the Flight, Commercial, and Industrial classes to represent the planes. In the next phases, we will implement the various behaviours of these planes.

The key things Flights do is "**book**" some payload to be carried on a Flight. Payloads are valid if: the weight is less than or equal to a maximum weight (MAX_WEIGHT), **and** there is space in the in Flight for the item (MAX_PAYLOAD).

- Create **abstract class** Flight:
  - an ArrayList, or partially filled array of Payload objects
  - An **int** constant, MAX_WEIGHT, set to 200, which is the maximum item weight
  - An **int** constant MAX_PAYLOAD, set to 100, which is the maximum number of items that can be loaded.
  - Method **public void book(Payload payload)**, which accepts a **Payload** object as a parameter. If the payload is valid it should be added to the list of Payload objects. If a payload is not valid, an **InvalidBookingException** should be thrown and the item should not be added to the list.
  - A default **toString()** method which displays information about a Flight. See Test Phase 2 for the expected output.
  - Note that **Flight** inherits an instance variable **ID** from the **Item** class as seen in the Item skeleton file.
- Create two other classes **Commercial** and **Industrial** that are subclasses of **Flight**
  - Commercial and Industrial planes impose **additional** validations to payloads that are passed to the **book** method. If the Payload is determined to be not valid, it should not be added to the list of Payload objects, and an **InvalidBookingException** should be thrown.
    - For Commercial planes, a Payload is valid if
      - it passes the previously defined validation, and
      - the payload is a type of **Person** (including an **Employee**),
    - For Industrial planes, a Payload is valid if
      - it passes the previously defined validation, and
      - it is **Cargo** or an **Employee** (but ***not*** a non-Employee Person).
  - These additional validations should be handled in the **book** methods of Commercial and Industrial. You are welcome to add helper methods if you prefer.

Phase 3: Item

Items are used in the system as a superclass for both Flight and Payloads. All Items are issued a unique identifier within the system, for tracking. The rules for IDs are as follows:

- All ID numbers are 9 digits long
- The first digit changes based on the type of item
  - Commercial Flights start with 1
  - Industrial Flights start with 2
  - Persons start with 3
  - Cargo starts with 4
- The last 8 digits begin at 0 and increase by one for each item created

  - That is, the first Item created will end in 0, the next will end in 1, and so on. Example: creating a Commercial flight would be issued id 100000000, then a Person would be issued 300000001 respectively.

Consider where to add code in the hierarchy (Item, and its subclasses) to generate and store these IDs.

Class Item should also have the following:

- Add method **public static int getTotalNumberOfItems()**, which returns the number of Items that have been created.
- Add **public int getID()** which returns the ID that has been generated for this object.

Phase 4: Airline

Next, implement the start of Airline so we can keep track of our Flights and Payloads. An Airline tracks all Flights and all Payloads in lists. For lists, you can use partially-filled arrays, ArrayLists or any other collection type you deem fit. Airlines will then use the following methods to track the data:

- Method **public String addFlight(char type)** which adds a new Flight. This method takes in a **char** which should be either 'C' (for Commercial Flights) or 'I' (for Industrial Flights). It should then create the appropriate Object and store it in the **flights** array.
  - If the char received is neither 'C' nor 'I', addFlight should throw an InvalidInputException
  - If a Flight is successfully added, this method should return a String saying so (as seen below), which the calling method (in Main) should print.
- Add a method **public Flight getFlight(int id)** which accepts an ID number and returns the Flight from **flights** which has that ID number, or **null** if no such Flight exists.

Now, edit **Main** to take advantage of your new class

- Your main method from Phase 1 creates a new Airline, and reads in a list of commands via readInputFile and parseCommand.
- Update your **parseCommand** to handle the following commands, following the guidelines in Phase 1.
  - `CREATE-FLIGHT typeCode`
    - This creates a new Flight. The second token (typeCode) will be "C" for Commercial Flights and "I" for Industrial Flights. Any other token should throw an **InvalidInputException**.
  - `GET-FLIGHT id`
    - This finds the Flight with the specified ID and prints its data by calling the toString on the flight. If no such Flight exists, it prints an error message instead.

You can test your program by running it with the TestPhase4.txt file provided. You should get the output seen here:

```
Commercial Flight 100000000 has a payload size of 0
Commercial Flight 100000001 has a payload size of 0
Industrial Flight 200000002 has a payload size of 0
InvalidInputException: Flight Creation Command Incorrect
Commercial Flight 100000001 has a payload size of 0
Flight 200000001 not found
```

**can also test everything using TestPhase3.txt

Phase 5: Payload and subclasses

It is time to more fully implement Payload and its various subclasses. Modify the provided files, changing the constructors (including updating the constructor signature), adding methods variables where appropriate, to make the implementation work.

- All Payload items have a **weight**, which should be stored as a double. A Payload's weight will be passed in when the constructor is called (this is already done in the Payload skeleton).
- Persons (including Employees) should have a **firstName** and a **lastName**.
- Employees have (in addition to their ID number), an **empID** number. This should be generated in the same way as ID numbers, except the first digit should be 5. However, the count of Employees should be independent of the number of Items created.
    - That is, the first employee will always have an empID of 500000000 regardless of how many other Persons or Cargo has been created
- Employees also have a **job**, which is passed into their constructor along with their name.
- Create a method **public boolean hasJob(String isItThisJob)** in Employee. This method accepts a String as input. If the String matches the Employee's job, return True. Otherwise, return False.
- Create a **toString** method in Payload to print out a Payload's ID and weight, as seen here. Note that the weights must line up as shown. You may assume nothing will be more than 9999 kg.

```
ID: 400000002, Weight:   50 kg
ID: 400000003, Weight:  150 kg
```

- Create a toString method in Person to add the Person's first name to its output, as seen here

```
ID: 300000004, Weight:   60 kg, Name: Mike
```

- Create a toString method in Employee to add the Employee's empID and job to its output, as seen here

```
ID: 300000005, Weight:   75 kg, Name: Mike, EMP: 500000000, Job: Pilot
```

- In the Payload skeleton file, there is a parameter-less constructor. Now that Payload has been fully implemented, delete it, as it is not needed.

Phase 6: Payloads

Create a factory method in Payload to create Payloads from given data – **public static Payload payloadFactory(String payloadType, String weight, String firstname, String lastname, String job) throws InvalidInputException**. Instantiate appropriate objects given the following rules:

- payloadType will be 'P' for Person, 'E' for Employee, and 'C' for Cargo.
- weight will be expressed as an integer (see TestPhase7.txt for examples)
- Persons have a weight, a first and last name, but no job.
- Employees have a weight, a first and last name, and a job.
- Cargo has a weight, but no first name, last name, or job.

You may assume the input is good and reliable. If the number provided for weight cannot be made into a Double, throw the exception.

In parseCommand, pass null for the input that has not been provided if there is none provided on the line that you are processing. For instance, cargo would have null for job.

Create methods **addPayload** and **getPayload** to class **Airline** to manage Payloads that are to be sent.

- Method **public void addPayload (Payload thePayload)** which adds a new Payload to the list of Payloads stored in Airline.
- Add a method **public Payload getPayload(int id)** which accepts an ID number and returns the Payload from the list of Payloads which has that ID number, or **null** if no such Payload exists.

Phase 7: Expanding Functionality

Now it is time to add some more commands for the system to process. Update your program to be able to process the following commands. Add instance variables and methods as required. Any variation from the template shown below should throw an InvalidInputException.

- `CREATE-PAYLOAD payloadType weight firstName lastName job`
  - Use your factory method to create a payload with this data. The parameters are the same. Keep track of this payload in your Airline object.
- `GET-PAYLOAD id`
  - This command should produce the same result as GET-FLIGHT but for Payloads
  - Use accessor methods you have defined in previous phases
- `ASSIGN-PAYLOAD flightID payloadID`
  - This should call **book** to add the payload with the given ID to the flight with the given ID. If either the flight or the payload does not exist, it should throw an `InvalidInputException` with an appropriate message.

Output of TestPhase7.txt:

```
Commercial Flight 100000000 has a payload size of 0
ID: 300000001, Weight:   50 kg Name: John
ID: 300000002, Weight:   75 kg Name: Jane EMP: 500000000, Job: Pilot
ID: 400000003, Weight:  200 kg
ID: 300000001, Weight:   50 kg Name: John
ID: 300000002, Weight:   75 kg Name: Jane EMP: 500000000, Job: Pilot
InvalidInputException: Payload  200000001 does not exist
InvalidInputException: Can only book Persons for Commercial flights
Commercial Flight 100000000 has a payload size of 1
```

Phase 8: Taking Flight

Finally, implement the ability to make planes take off. Do the following:

- Add a new variable to Flight to track if the flight has taken off. A flight can only take off once.
- Add **public boolean canTakeoff()** in Flight to check if a Flight can take off. A Flight can take off only if all the following are true.
  - It has not already taken off
  - This flight contains at least 2 items
  - It has at least one Employee in its payload
- Add a **public String doTakeoff()** in Flight that will be called to make a plane take off
  - The method should call canTakeoff() to check if the Flight can take off.
  - If it can, this method should make the flight take off
  - It should return a String describing whether the Flight took off (as seen in the test phase)
- Edit Main and Airline as necessary to implement the command: `TAKEOFF flightID`
  - If the flightID does not exist, throw an `InvalidInputException`
  - Otherwise, print the message resulting from the call to the doTakeoff method.

Bonus (5%):

1. Do not allow the same payload to be booked on multiple flights.
2. Do not allow the same payload to be booked twice on the same flight

Make a note in your submission that you accomplished this task. Include a test phase as proof.

ASSIGNMENT 3: Polymorphism

COMP 1020 Winter 2022

Final note

Be sure to use Object Orientation. The only methods that should use "instance of" are **book** in Commercial and Industiral, and **canTakeOff** in Flight. If you are using instanceof in other places, you can rewrite it by using polymorphism!

Assignment Guidelines

Unless specified otherwise, all instance variables must be **`private`** or **`protected`**, and all methods should be **`public`**. Also, unless specified otherwise, there should be ***no*** **println** statements in your classes. Objects usually do not print anything; they only return **String** values from **toString** methods.

Your code should be DRY – don't repeat yourself. If you have a method that already accomplishes a task, or has some logic, that logic should be used. You may add any **`private`** methods you see fit. Only use **instanceof** where it is absolutely necessary – most problems can be solved with polymorphism instead of **instanceof**.

The description of the assignment has methods that are required. There may be suggested names for the parameters, which you may change. Unless specifically noted, provided method signatures are suggestions and you may change them if you wish.

This assignment is more open-ended than the previous two. The parameters you need for methods are not always specified, nor are the types of instance variables. In addition, you will not be told specifically which methods should catch which errors. This is by design; the goal of this assignment is to get you to think carefully about how to design a program given only the abstract specifications.

Testing Your Coding

We have provided sample test files for each phase to help you see if your code is working according to the assignment specifications. These files are starting points for testing your code. Part of developing your skills as a programmer is to think through additional important test cases and to write your own code to test these cases. ***Different, more vigorous tests will be used by the markers to test your program.***

Sample output is provided in some cases, for you to match as closely as possible. Other output is left out, for you to practice reading code, understand a program, and practice seeing if output is correct or not based on inputs.

Hand in

Submit all your Java files. Do not submit .class or .java~ files! You do not need to submit the TestPhaseN.java files that were given to you. If you did not complete all phases of the assignment, use the Comments field when you hand in the assignment to tell the marker which phases were completed, so that only the appropriate tests can be run. For example, if you say that you completed Phases 1-2, then the marker will compile your files with appropriate test cases. If it fails to compile and run, you will lose all the marks for the test runs. The marker will not try to run anything else and will not edit your files in any way. Make sure none of your files specify a package at the top!