Project: Air Ticket – Part 1

Deliverables

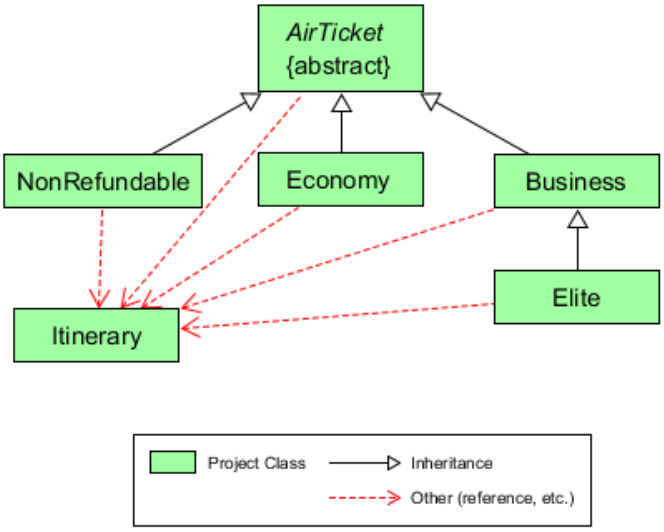
Your project files should be submitted to Web-CAT by the due date and time specified. Note that there is also an optional Skeleton Code assignment which will indicate level of coverage your tests have achieved (there is no late penalty since the skeleton code assignment is ungraded for this project). The files you submit to skeleton code assignment may be incomplete in the sense that method bodies have at least a return statement if applicable or they may be essentially completed files. To avoid a late penalty for the project, you must submit your completed code files to Web-CAT no later than 11:59 PM on the due date for the completed code assignment. If you are unable to submit via Web-CAT, you should e-mail your project Java files in a zip file to your TA before the deadline. The grades for the Completed Code submission will be determined by the tests that you pass or fail in your test files and by the level of coverage attained in your source files as well as usual correctness tests in Web-CAT.

Files to submit to Web-CAT:

- Itinerary.java, ItineraryTest.java
- AirTicket.java
- NonRefundable.java, NonRefundableTest.java
- Economy.java, EconomyTest.java
- Business.java, BusinessTest.java
- Elite.java, EliteTest.java

Specifications

Overview: This project is the first of three that will involve the pricing and reporting of air tickets. You will develop Java classes that represent categories of air tickets including non-refundable, economy, business and business elite, all of which will have an itinerary. Note that there is no requirement for a class with a main method in this project. You will need to create a JUnit test file for the indicated classes and write one or more test methods to ensure the classes and methods meet the specifications. You should create a jGRASP project and add the class and test files as they are created. All of your files should be in a single folder. The UML class diagram at right provides a visual overview of how the classes in the project relate to one another.



You should read through the remainder of this assignment before you start coding.

- **Itinerary.java**

  Requirements: Create Itinerary class that stores trip data and provides methods for get departure date/time, get miles, and toString.

  Design: The Itinerary class has fields, a constructor, and methods as outlined below.

  (1) **Fields**: *instance variables* of type String for "from airport", "to airport", the departure date/time (e.g., "2021/11/21 1430"), arrival date/time, and of type int for miles. These five variables should be private so that they are not directly accessible from outside of the class. These are the only five fields this class should have.

  (2) **Constructor**: Your Itinerary class must contain a constructor that accepts five parameters representing the values to be assigned to the fields above. Below is an example of how the constructor could be used to create an Itinerary object:

```
Itinerary trip = new Itinerary("ATL", "LGA", "2021/11/21 1400",
                               "2021/11/21 1640", 800);
```

  (3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. At minimum you will need the following methods.
  - `getFromAirport`: Accepts no parameters and returns a String representing the "from airport".
  - `getDepDateTime`: Accepts no parameters and returns a String representing the departure date/time.
  - `getMiles`: Accepts no parameters and returns an int miles field.
  - `toString`: Returns a String containing the information in the Itinerary object as shown below.

    `ATL-LGA (2021/11/21 1400 - 2021/11/21 1640) 800 miles`

  Code and Test: As you implement your Itinerary class, you should compile and test it as methods are created by using a combination of interactions and JUnit test methods. For example, as soon you have implemented and successfully compiled the constructor, you should create an instance of Itinerary in interactions. Remember that when you have an instance on the workbench, you can unfold it to see its values. After you have implemented and compiled one or more of the methods, you should begin creating test methods in the corresponding ItineraryTest.java file.

- **AirTicket.java**

  Requirements: Create an *abstract* AirTicket class that stores ticket data and provides methods to access the data.

Design: The AirTicket class has fields, a constructor, and methods as outlined below.

(1) **Fields**: *instance* variables for flight number of type String, trip data of type Itinerary, base fare of type double, and fare adjustment factor of type double. These variables should be declared with the *protected* access modifier so that they are accessible in the subclasses of AirTicket. These are the only fields that this class should have.

(2) **Constructor**: The AirTicket class must contain a constructor that accepts four parameters representing the values to be assigned to the fields above. Since this class is abstract, the constructor will be called from the subclasses of AirTicket using *super* and the parameter list.

(3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. At minimum you will need the following methods.

- `getFlightNum`: Accepts no parameters and returns a String representing the flight number.
- `getBaseFare`: Accepts no parameters and returns a double representing the base fare.
- `getFareAdjustmentFactor`: Accepts no parameters and returns a double representing the fare adjustment factor field.
- `toString`: Returns a String containing the information in the AirTicket object. This method will be called from the toString method in the subclasses of AirTicket using super.toString(). In each of the examples for the toString methods of the subclasses below, the first four lines are produced AirTicket's toString method. Note that you can get the class name for an instance c by calling c.getClass(). The DecimalFormat should use `"$#,##0.00"` as the pattern for dollar amounts.
- `totalFare`: An *abstract* method that accepts no parameters and returns a double representing the total fare. Subclasses of AirTicket must implement this method.
- `totalAwardMiles`: An *abstract* method that accepts no parameters and returns an int representing the award miles. All direct subclasses of AirTicket must implement this method.

- **NonRefundable.java**

Requirements: Derive the class NonRefundable from AirTicket.

Design: The NonRefundable class has a field, a constructor, and methods as outlined below.

(1) **Field**: *instance* variable for discount factor of type double. This variable should be declared with the *private* access modifier. This is the only field that should be declared in this class.

(2) **Constructor**: The NonRefundable class must contain a constructor that accepts four parameters representing the values in the AirTicket class and the one parameter for the field declared in NonRefundable. Since this class is a subclass of AirTicket, the super constructor should be called with values for AirTicket. The field in this class should be set with the last

parameter. Below is an example of how the constructor could be used to create an Itinerary object:

```
NonRefundable nr = new NonRefundable("DL 1860", trip, 450, 0.45, 0.90);
```

(3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. At minimum you will need the following methods.

- `getDiscountFactor`: Accepts no parameters and returns a double representing the discount factor.
- `totalFare`: Accepts no parameters and returns a double representing the total fare calculated by multiplying the base fare by the fare adjustment factor and the discount factor.
- `totalAwardMiles`: Accepts no parameters and returns an int equal to the actual miles from the itinerary.
- `toString`: Returns a String containing the information in the NonRefundable object. This method will call toString method in the parent class using super.toString() and then the additional information declared in NonRefundable as shown below. Note that the first four lines below should come from the toString method in AirTicket. The fifth line is indented three spaces (not tabbed).

```
Flight: DL 1860
ATL-LGA (2021/11/21 1400 - 2021/11/21 1640) 800 miles (800 award miles)
Base Fare: $450.00 Fare Adjustment Factor: 0.45
Total Fare: $182.25 (class NonRefundable)
   Includes DiscountFactor: 0.9
```

Code and Test: As you implement the NonRefundable class, you should compile and test it as methods are created by using a combination of interactions and JUnit test methods. For example, as soon you have implemented and successfully compiled the constructor, you should create an instance of Itinerary in interactions. Remember that when you have an instance on the workbench, you can unfold it to see its values. After you have implemented and compiled one or more of the methods, you should begin creating test methods in the corresponding NonRefundableTest.java file.

- **Economy.java**

Requirements: Derive the class Economy from AirTicket.

Design: The Economy class has fields, a constructor, and methods as outlined below.

(1) **Field**: *static final* variable for economy award miles factor of type double. This field is constant that should be set to 1.5 with the *public* access modifier. This is the only field that should be declared in this class.

(2) **Constructor**: The Economy class must contain a constructor that accepts four parameters representing the values in the AirTicket class. Since this class is a subclass of AirTicket, the super constructor should be called with values for AirTicket. Below is an example of how the constructor could be used to create an Economy object:

```
Economy e = new Economy("DL 1860", trip, 450, 1.0);
```

(3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. At minimum you will need the following methods.

- `totalFare`: Accepts no parameters and returns a double representing the total fare calculated by multiplying the base fare by the fare adjustment factor.
- `totalAwardMiles`: Accepts no parameters and returns an int equal to the actual miles from the itinerary multiplied by the economy award mile factor. Note that the returned value must be cast to int since it is calculated using the economy award mile factor, which is a double.
- `toString`: Returns a String containing the information in the Economy object. This method will call toString method in the parent class using super.toString() and then the additional information declared in Economy as shown below. Note that the first four lines below should come from the toString method in AirTicket. The fifth line is indented three spaces (not tabbed).

```
Flight: DL 1860
ATL-LGA (2021/11/21 1400 - 2021/11/21 1640) 800 miles (1200 award miles)
Base Fare: $450.00 Fare Adjustment Factor: 1.0
Total Fare: $450.00 (class Economy)
   Includes Award Miles Factor: 1.5
```

Code and Test: As you implement the Economy class, you should compile and test it as methods are **created** by using a combination of interactions and JUnit test methods. For example, as soon you have implemented and successfully compiled the constructor, you should create an instance of Itinerary in interactions. Remember that when you have an instance on the workbench, you can unfold it to see its values. After you have implemented and compiled one or more of the methods, you should begin creating test methods in the corresponding EconomyTest.java file.

- **Business.java**

Requirements: Derive the class Business from AirTicket.

Design: The Business class has fields, a constructor, and methods as outlined below.

(1) **Field**: *instance* variables for food&beverages of type double and entertainment of type double. These fields should be declared with the *protected* access modifier. Also, a *static final* variable for business award miles factor of type double. This field is constant that should be set to 2.0 with the *public* access modifier. These are the only fields that should be declared in this class.

(2) **Constructor**: The Business class must contain a constructor that accepts four parameters representing the values in the AirTicket class and two for the fields above. Since this class is a subclass of AirTicket, the super constructor should be called with values for AirTicket. Below is an example of how the constructor could be used to create an Business object:

```
Business b = new Business("DL 1860", trip, 450, 2.0, 50.0, 50.00);
```

(3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. At minimum you will need the following methods.

- `totalFare`: Accepts no parameters and returns a double representing the total fare calculated by multiplying the base fare by the fare adjustment factor and then adding food&beverages and entertainment.
- `totalAwardMiles`: Accepts no parameters and returns an int equal to the actual miles from the itinerary multiplied by the business award miles factor. Note that the returned value must be cast to int since it is calculated using the business award mile factor, which is a double.
- `toString`: Returns a String containing the information in the Economy object. This method will call toString method in the parent class using super.toString() and then the additional information declared in Economy as shown below. Note that the first four lines below should come from the toString method in AirTicket. The DecimalFormat should use `"$#,##0.00"` as the pattern for dollar amounts. The fifth line is indented three spaces (not tabbed).

```
Flight: DL 1860
ATL-LGA (2021/11/21 1400 - 2021/11/21 1640) 800 miles (1600 award miles)
Base Fare: $450.00 Fare Adjustment Factor: 2.0
Total Fare: $1,000.00 (class Business)
   Includes Food/Beverage: $50.00 Entertainment: $50.00
```

Code and Test: As you implement the Business class, you should compile and test it as methods are **created** by using a combination of interactions and JUnit test methods. For example, as soon you have implemented and successfully compiled the constructor, you should create an instance of Itinerary in interactions. Remember that when you have an instance on the workbench, you can unfold it to see its values. After you have implemented and compiled one or more of the methods, you should begin creating test methods in the corresponding BusinessTest.java file.

- **Elite.java**

Requirements: Derive the class Elite from class Business.

Design: The Elite class has fields, a constructor, and methods as outlined below.

(2) **Field**: *instance* variable for communication services of type double. This field should be declared with the *private* access modifier. This is the only field that should be declared in this class.

(4) **Constructor**: The Elite class must contain a constructor that accepts four parameters representing the values in the AirTicket class, two for the fields from the Business class, and one for the field above. Since this class is a subclass of Business, the super constructor should be called with values for Business. Below is an example of how the constructor could be used to create an Business object:

```
Elite be = new Elite("DL 1860", trip, 450, 2.5, 50.0, 50.00, 100.00);
```

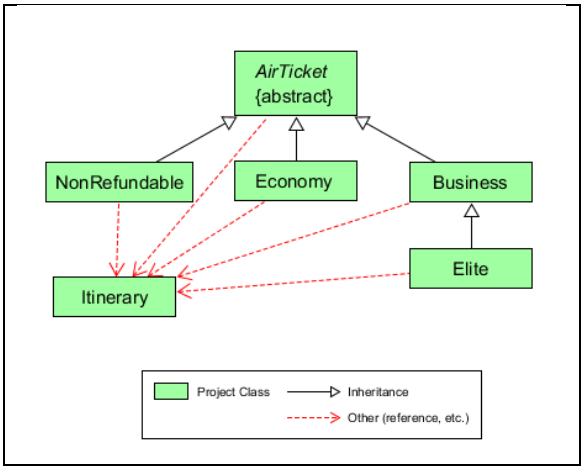
(5) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. At minimum you will need the following methods.

- `totalFare`: Accepts no parameters and returns a double representing the total fare calculated by calling super.totalFare() and then adding communication services.
- `totalAwardMiles`: None – the Elite class will use the method inherited from Business.
- `toString`: Returns a String containing the information in the Economy object. This method will call toString method in the parent class using super.toString() and then the additional information declared in Economy as shown below. Note that the first four lines below should come from the toString method in AirTicket, and the fifth line should come from the toString method in Business. The DecimalFormat should use `"$#,##0.00"` as the pattern for dollar amounts. The fifth and sixth lines are indented three spaces (not tabbed).

```
Flight: DL 1860
ATL-LGA (2021/11/21 1400 - 2021/11/21 1640) 800 miles (1600 award miles)
Base Fare: $450.00 Fare Adjustment Factor: 2.5
Total Fare: $1,325.00 (class Elite)
   Includes Food/Beverage: $50.00 Entertainment: $50.00
   Includes: Comm Services: $100.00
```

Code and Test: As you implement the Elite class, you should compile and test it as methods are **created** by using a combination of interactions and JUnit test methods. For example, as soon you have implemented and successfully compiled the constructor, you should create an instance of Itinerary in interactions. Remember that when you have an instance on the workbench, you can unfold it to see its values. After you have implemented and compiled one or more of the methods, you should begin creating test methods in the corresponding EliteTest.java file.

UML Class Diagram: As you add your classes to the jGRASP project, you should generate the UML class diagram for the project. Once generated, you can use the mouse to select/drag a class to arrange the diagram like the one below (repeated from page 1).


AirTicket
{abstract}
NonRefundable
Economy
Business
Elite
Itinerary
Project Class
Inheritance
Other (reference, etc.)