CS2030 (2310) Practical Assessment #1

Tags & Categories

Tags:

Categories:

Related Tutorials

Task Content

Public Transport

A public transport system comprises buses and trains. Passengers pay a fare according to the bus and/or train rides within a trip. The fare is based upon a prescribed fare system, and there are many fare systems available.

Task

Your task is to track the different rides within a trip and compute the fare for the entire trip based on a given fare system. Although this exercises only focuses on buses and trains, your program should be able to support other rides.

Take note!

There are altogether five levels. You are required to complete ALL levels.

You should keep to the constructs taught in class, as well as the programming discipline instilled throughout the course.

- Write each `class/interface` in its own file; you are NOT allowed to use `enum` or `record`.
- Ensure that ALL declarations of object properties and class constants include the `private` and `final` modifiers.
- Ensure that your classes are NOT cyclic dependent.
- You are NOT allowed to use java libraries, other than those from `java.lang`, `java.util.List`, `java.util.Comparator` and `java.util.Iterator`.
- If necessary, use only the `ImList` class and `Pair` class provided; do NOT create your own.
- `instanceof` can only be used in a class to check `instanceof` the same class, e.g. within a typical `equals(Object)` method.
- You are NOT allowed to use java keywords/literals: `null`.
- You are NOT allowed to use Java reflection, i.e. `Object::getClasses` and other methods from `java.lang.Class`
- You are NOT allowed to use * wildcard imports.
- You are NOT allowed to use explicit typecasts to convert between types.
- You are NOT allowed to define anonymous inner classes or lambdas.
- You are NOT allowed to define array constructs, e.g. `String[]` or using ellipsis, e.g.`String...`

You may assume that all tests provide valid arguments to the methods; hence there is no need to validate method arguments.

Level 1

The following interface `Ride` is provided:

```
interface Ride {
    public int distance();
    public boolean isSameRide(Ride otherRide);
}
```

Define the class `Bus` as an implementation of `Ride` with a constructor that takes in an alphanumeric bus number as a `String`, an integer boarding stage and an integer alighting stage. You may assume that each stage is 800 metres, and that the boarding stage is strictly less than the alighting stage.

Include the `distance` method that returns the distance (in metres) between the two stages of the bus ride.

Include an appropriate `toString` method that returns a `String` representation of the bus ride. As we not testing `isSameRide` in this level, you may implement the method in any way you wish.

```
$ javac your_java_files
$ jshell your_java_files_in_bottom-up_dependency_order
jshell> Bus bus = new Bus("96A", 4, 7)
bus ==> [Bus 96A: 4 -- 7]

jshell> bus.distance()
$.. ==> 2400

jshell> Ride ride = bus
ride ==> [Bus 96A: 4 -- 7]
```

Level 2

A trip involves a series of rides. In this level, the only rides are bus rides.

Write a class `Trip` with a constructor that takes in the first ride to start the trip.

Include the method `next` that takes in a new ride, and appends it to the end of the current trip.

Include an appropriate `toString` method that returns a `String` representation of the bus trip.

```
$ javac your_java_files
$ jshell your_java_files_in_bottom-up_dependency_order
jshell> new Trip(new Bus("96A", 4, 7))
$.. ==> [Bus 96A: 4 -- 7]

jshell> Trip trip = new Trip(new Bus("96A", 4, 7))
trip ==> [Bus 96A: 4 -- 7]

jshell> trip.next(new Bus("95", 3, 10))
$.. ==> [Bus 96A: 4 -- 7][Bus 95: 3 -- 10]

jshell> trip
trip ==> [Bus 96A: 4 -- 7]

jshell> trip.next(new Bus("96A", 6, 10))
$.. ==> [Bus 96A: 4 -- 7]
```

Note from the last test above that a new ride with the same bus number as the last bus ride cannot be appended to the trip, regardless of the boarding and alighting stages. You will need to define the `isSameRide` method to serve this purpose.

Level 3

We now include the train. Write a class `Train` with a constructor that takes two integers: the boarding train station and the alighting train station, where the boarding station is strictly less than the alighting station. For now, you may assume that there are only five stations in the current rail system with the distance (in metres) between stations indicated by the table below:

- From station 1 to station 2: `1500`
- From station 2 to station 3: `2400`
- From station 3 to station 4: `1200`
- From station 4 to station 5: `900`

Include the `distance` method that returns the distance (in metres) traveled between the two stations, and an appropriate `toString` method that returns a `String` representation of the train ride.

Moreover, a train ride can now be included in a trip.

```
$ javac your_java_files
$ jshell your_java_files_in_bottom-up_dependency_order
```

```
jshell> Train train = new Train(1, 2)
train ==> [Train: 1 -- 2]

jshell> train.distance()
$.. ==> 1500

jshell> train = new Train(1, 3)
train ==> [Train: 1 -- 3]

jshell> train.distance()
$.. ==> 3900

jshell> Ride ride = train
ride ==> [Train: 1 -- 3]

jshell> Trip trip = new Trip(train)
trip ==> [Train: 1 -- 3]

jshell> trip.next(new Bus("96A", 4, 7))
$.. ==> [Train: 1 -- 3][Bus 96A: 4 -- 7]

jshell> trip.next(new Train(3, 5))
$.. ==> [Train: 1 -- 3]

jshell> trip.next(new Train(3, 5)).next(new Bus("96A", 4, 7)).next(new Bus("96A", 6, 10))
$.. ==> [Train: 1 -- 3][Bus 96A: 4 -- 7]
```

Notice from the last two test cases above that consecutive train rides where the passenger alights at a station and boards another train at the same station is not allowed.

Level 4

We are now ready to include the fare system. For simplicity, we assume that the same fare system is applied to all types of rides.

We start with the staged fare system that is based on separate fares for each ride within a trip. The total fare can then be computed as a sum of the fares.

The following interface `Fare` is provided:

```
interface Fare {
    public int computeFare(Iterable<Ride> trip);
}
```

Notice that the `computeFare` method takes in a trip as an `Iterable<Ride>`. Although it may seem that the `computeFare` method can take in a trip as a `Trip`, you will realize later that `Trip` would have to be dependent on `Fare`, and hence a cyclic dependency arises.

Write the `StagedFare` class to represent the table of fares (in cents), e.g.

- up to 4000m: `90`
- up to 7000m: `110`
- up to 10000m: `130`
- ...

Rather than hard-coding the fare table, it should be constructed "on-the-fly". This is useful since higher fares for longer distances can be added to an existing fare table when the need arises, e.g. due to inflation.

Write a class `StagedFare` that implements `Fare`. Include a constructor that takes the initial entry of the fare table as two integer arguments: the distance (in metres) and the fare (in cents). As an example, `new StagedFare(4000, 90)` will initialize a new bus fare table with a fare of 90 cents up to 4000 metres. However, as there is no further entry, distances greater than 4000 will also be 90 cents.

To add another entry into the table, include the `add` method that takes in a distance and a fare. For example,

```
new StagedFare(4000, 90).add(7000, 110)
```

will create a fare table where the fare is 90 cents up to 4000m, and 110 cents up to 7000m. Since there is no further entry, distances above 7000 will also be 110 cents.

You may assume that longer distances have correspondingly higher bus fares. However, you cannot assume that entries will be added to the fare table in increasing distances or fares.

Include a `toString` that returns a `String` representation of the fare with the corresponding fare range (i.e. lowest and highest fares) in the table.

```
$ javac your_java_files
$ jshell your_java_files_in_bottom-up_dependency_order
jshell> new StagedFare(4000, 90)
$.. ==> Staged fare from 90 to 90

jshell> new StagedFare(4000, 90).add(7000, 110)
$.. ==> Staged fare from 90 to 110

jshell> new StagedFare(7000, 110).add(4000, 90)
$.. ==> Staged fare from 90 to 110
```

How do we compute the fare of a trip? In the `Trip` class, include the method `fare` that takes in a fare system and returns the total fare over all rides within the trip. Doing this will allow us to compute the fare of a trip under different fare systems.

```
jshell> StagedFare sfare = new StagedFare(7000, 110).add(4000, 90)
sfare ==> Staged fare from 90 to 110

jshell> Fare fare = sfare
fare ==> Staged fare from 90 to 110

jshell> Trip trip = new Trip(new Train(1, 5))
trip ==> [Train: 1 -- 5]

jshell> trip.fare(sfare)
$.. ==> 110

jshell> trip.next(new Bus("96A", 4, 7)).fare(sfare)
$.. ==> 200
```

Level 5

We would like to offer a rebate for each transfer within the trip. The rebate is offered based on an existing staged fare system.

Write a `RebateFare` class with a constructor that takes in a `StagedFare` and an integer rebate amount so as to offer a rebate for all transfers from one ride to another. Note that rebate can only be applied to a staged fare system. You may also assume that the rebate amount will not exceed the minimum fare of the system.

Include a `toString` method to indicate that a rebate is applied to the fare system.

```
jshell> StagedFare sfare = new StagedFare(7000, 110).add(4000, 90)
sfare ==> Staged fare from 90 to 110

jshell> RebateFare rfare = new RebateFare(sfare, 25)
rfare ==> Staged fare from 90 to 110 (with rebate 25)

jshell> Trip trip = new Trip(new Train(1, 5))
trip ==> [Train: 1 -- 5]

jshell> trip.fare(rfare)
$.. ==> 110

jshell> trip.next(new Bus("96A", 4, 7)).fare(rfare)
$.. ==> 175
```

Notice from the last test case above that a rebate is applied only upon a transfer, and not for every ride.

Lastly, include another distance-based fare system. As an example,

- 91 cents up to 3200m, followed by
- 10 cents for every 1000m fare bands, up to
- 40200m (excluding the first 3200m)

Write a class `DistanceFare` and include a constructor that takes five arguments: the flat fare up to the first flat fare distance, the band fare for each band fare distance, and the maximum distance. Using the above example, the corresponding `DistanceFare` created is:

```
new DistanceFare(91, 3200, 10, 1000, 40200)
```

Include an appropriate `toString` method to return the `String` representation of the distance-based fare.

```
jshell> DistanceFare dfare = new DistanceFare(91, 3200, 10, 1000, 40200)
dfare ==> Distance fare: 91 up to 3200; 10 every 1000 up to 40200

jshell> Trip trip = new Trip(new Train(1, 5))
trip ==> [Train: 1 -- 5]

jshell> trip.fare(dfare)
$.. ==> 121

jshell> trip.next(new Bus("96A", 4, 7)).fare(dfare)
$.. ==> 151
```

Note that a rebate cannot be offered to distance-based fares. Also a fare system that has a rebate cannot be offered a further rebate.

```
jshell> StagedFare sfare = new StagedFare(7000, 110).add(4000, 90)
sfare ==> Staged fare from 90 to 110

jshell> RebateFare rfare = new RebateFare(sfare, 25)
rfare ==> Staged fare from 90 to 110 (with rebate 25)

jshell> new RebateFare(dfare, 25)
|  Error:
|  incompatible types: DistanceFare cannot be converted to StagedFare
|  new RebateFare(dfare, 25)
|                 ^---^

jshell> new RebateFare(rfare, 25)
|  Error:
|  incompatible types: RebateFare cannot be converted to StagedFare
|  new RebateFare(rfare, 25)
|                 ^---^
```

And finally, just to make sure...

```
jshell> Fare fare = sfare
fare ==> Staged fare from 90 to 110

jshell> Fare fare = rfare
fare ==> Staged fare from 90 to 110 (with rebate 25)

jshell> Fare fare = dfare
fare ==> Distance fare: 91 up to 3200; 10 every 1000 up to 40200
```