Introduction to Computers and Engineering Problem Solving
Spring 2012
Problem Set 5: MBTA routes
Due: 12 noon, Friday, March 23

Problem statement

a. Overview

The Massachusetts Bay Transportation Authority (MBTA) operates routes using three basic technologies:

- Buses, which operate as single units with a driver
- Urban rail, such as the Red, Orange, Blue and Green Lines, which operate trains consisting of individual, powered cars; a train has an operator and may have additional on-train staff.
- Commuter rail, such as lines to Lowell or Providence, which operate trains consisting of one engine and a number of unpowered cars; a train has an engineer in the engine, and additional staff on the cars.

A route must operate on at least one right-of-way (ROW) type:

- Dedicated right of way, which has tracks or a busway separated from the street system, and stops only at stations. Most urban rail, some bus, and all commuter rail routes operate on a dedicated ROW.
- Shared right of way, which operate together with or near private vehicle traffic on streets. Most bus routes and some urban rail routes operate on a shared ROW.
- Mixed right of way, which is a hybrid of dedicated and shared. For example, the Silver Line bus to the airport operates in its own tunnel at South Station, which is a dedicated right of way, as well as on public streets to and from the airport, which is a shared right of way. Green Line routes operate on shared right of way at their outer ends, as well as in dedicated right of way (tunnel) on their inner portions.

For this homework, you will design and implement a set of classes and interfaces and use them to calculate the MBTA's resource requirements. Your classes will be quite short and will have little computation. You would generally not use a full inheritance structure for a problem this simple, but if you were computing everything that the MBTA monitors on each route, this structure would be appropriate. This homework can be viewed as the initial design for a much larger system.

b. Data members

Each route has the following data:

- Route name (String).
- L= route length (double, miles)

- v= average speed (double, mph)
- h= headway, or interval between buses or trains (double, minutes)

Every bus or train on a route has one operator: bus driver, urban rail operator, or commuter train engineer.

Each urban rail route and commuter rail route has additional data members:

- c= train length, measured in the number of vehicles in a train, assumed constant all day (int)
- k= number of additional staff per train (e.g., conductors, ticket collectors) (int)

Bus routes have no data members for these items; there are no trains or extra staff on buses.

Each route with dedicated right of way has, in addition:

- b= number of stations (int)
- m= number of staff per station (int)

Each route with shared right of way has, in addition:

- a= number of stops (int); these are very simple, and are different than stations
- d= additional vehicles assigned to the route because of traffic delays; this is expressed as a multiplier from a base, e.g., 1.25 (double).

c. Methods: getVehicles() and getEngines()

For each route, you must implement a method called getVehicles(), which computes the number of vehicles required using the following equation:

$$e = 2 * 60 * L * c * d / (v * h) \qquad \text{(buses, urban rail cars, commuter rail cars)}$$

The factor of 2 is because vehicles operate round trip on the route, and the factor of 60 converts between speed (mph) and headway (minutes) units. For buses, where the bus is a single unit, omit variable c in the equation above. For dedicated right of way services, omit d. (Do not just set c=1 or d=1.) For hybrid right of way services, do not omit d.

Additionally, each commuter rail route must have a method getEngines(), which returns the number of engines using an equation that is similar to the equation above. For commuter rail engines, since there is one engine per train, omit variable c in the equation above. Also, For dedicated right of way services, omit d. (Again, do not just set c=1 or d=1.) Therefore, commuter rail routes will have both a getVehicles() and a getEngines() method. getEngines() is:

$$e_1 = 2 * 60 * L / (v * h) \qquad \text{(commuter rail engines)}$$

d. Method: getStaff()

For each route, you must have a method getStaff(), which computes the number of staff as the sum of the number of operators, other train staff, and station staff

The number of operators f for a bus, urban rail, or commuter rail route is:

$f = e/c$ (urban rail routes)

$f = e_1$ (commuter rail routes)

f = e (bus routes)

The number of other staff g on the train for urban rail and commuter rail is:

$g = \frac{k*e}{c}$ (urban rail routes)

$g = k * e_1$ (commuter rail routes)

where k is specific to each route, as shown in the table below, and is not defined for bus.

The number of station staff j for bus, urban rail or commuter rail routes that operate on a dedicated ROW is:

j= b*m (for any route with a dedicated ROW)

The getStaff() method returns n= f + g + j.

e. Data values

You are asked to calculate and output the number of vehicles required and the number of staff required for five MBTA routes using the data in the table below. "na" means not applicable.

Variable	Definition	Bus 77	Silver Line bus	Red Line	Green Line B	Lowell Line
L	Route length	5	7	16	12	25
v	Avg speed	12	20	18	10	30
a	Nbr stops	40	5	na	19	na
b	Nbr stations	na	3	17	11	6
h	Headway	7	10	8	10	30
c	Train length	na	na	6	2	5 cars, 1 engine
	Operator/train	1 driver	1 driver	1 operator	1 operator	1 engineer
k	Additional staff/train	na	na	1 other	1 other	2 other
m	Staff/station	na	1	2	2	1
d	Addl vehicles	1.20	1.15	na	1.25	na
	Route Type	Bus	Bus	Urban	Urban	Commuter
	ROW Type	Shared	Hybrid	Dedicated	Hybrid	Dedicated

f. Program

Your program must meet the following guidelines:

- You must use at least one abstract class, with appropriate abstract and final methods, and at least one interface; you may wish to have several abstract classes and interfaces. You must have at least one abstract method and one final method in at least one class.
- You may not represent the route type (bus, urban rail or commuter rail) or the right of way type (shared or dedicated) as a data member.
- Each subclass must inherit its data and methods from a superclass or an interface to the maximum extent possible. Override superclass methods as needed.
- Each subclass must not contain data that is not relevant. For example, the Red Line may not have stops or additional vehicles, and the 77 bus may not have stations, staff per train, vehicles per train, or staff per station. A subclass must not have any data marked "na" in the table above. You may not set "na" variables equal to one in a subclass that does not use them; they must not be present.
- If a route type has a certain data member, your design must require that it be present. For example, your design must require all shared ROW routes to have a number of stops. All urban rail and commuter rail routes must have a number of stations, staff/station and staff/train.
    - An interface cannot force an implementing class to have a data member, but if the interface has a method getX(), the implementing method will need to either have a data member X or compute X. This meets the requirement of the problem set.
- You may use protected access for data members to be inherited by subclasses, except that any data member managed by a final method must be private.
- Data members and methods must be defined in the highest level of the inheritance tree possible. For example, the data members that exist for all routes must be defined in a Route class, not in BusRoute, UrbanRailRoute or CommuterRail classes.
    - All of your methods will be very simple; they should all be just one line.
- You must write classes or subclasses for the combinations of Route and ROW listed at the in the table above. You should not write any other combination of Route and ROW.
    - You do not need to have a class for bus routes just on dedicated ROW
    - You do not need to have a class for urban rail routes just on shared ROW
- You must write a class RouteTest with only a main() method to test your software. In main(), write code to:
    - Create objects for five routes: bus 77, Silver Line/Airport, Red Line, Green Line B and Lowell Line
    - Compute and output (System.out.println) the number of vehicles and staff required for each route, including the number of engines for commuter rail routes. You will have fractional values, which is fine.
    - Output the number of stops and/or stations for each route.
    - Compute and output the total number of vehicles for bus routes, for urban rail routes and for commuter rail routes. (This will be three totals) You must use the instanceOf keyword in your loop to determine what kind of vehicle is used on a route, since all routes have the same getVehicle() method. Include commuter rail engines, using getEngine().
    - Compute and output the total staff across all routes.

- Remember to let Eclipse help you. Use its Source-> Generate… feature to write constructors and inherited method signatures, as used in lecture. Use the @Override annotation to let the compiler detect errors. Eclipse (and Java annotations) can save you a lot of time. Almost all of your code in this homework can be generated by Eclipse, except for class RouteTest.

Sample output

```
77         Staff: 8.57     Vehicles: 8.57  Stops: 40
silver     Staff: 7.83     Vehicles: 4.83  Stations: 3     Stops: 5
red        Staff: 60.67    Vehicles: 80.0  Stations: 17
green      Staff: 58.0     Vehicles: 36.0  Stations: 11    Stops: 19
lowell     Staff: 16.0     Vehicles: 16.67  ngines: 3.33   Stations: 6

Total staff: 151.07
Buses: 13.40
Urban rail vehicles: 116.0
Total commuter rail vehicles: 16.67
Total commuter rail engines: 3.33
```

Turn In

1. Place a comment with your full name, section, TA name and assignment number at the beginning of the .java file for your solution.
2. Place all of the files in your solution in a single zip file.
    a. Do not turn in copies of compiled byte code or backup (.class or .java~ files)
    b. Do not turn in printed copies of your solution.
3. Submit the single zip file on the 1.00 Web site under the appropriate section. For directions see How To: Submit Homework on the 1.00 Web site.
4. Your uploaded files should have a timestamp of no later than noon on the due date.
5. After you submit your solution, please recheck that you submitted your .java file. If you submitted your .class file, you will receive zero credit.

Penalties

- 30 points off if you turn in your problem set after Friday noon but before noon on the following Monday. You have one no-penalty late submission per term for a turn-in after Friday noon and before Monday noon.
- No credit if you turn in your problem set after noon on the following Monday.

1.00 / 1.001 / 1.002 Introduction to Computers and Engineering Problem Solving
Spring 2012