CMSC 202 — Fall 2013
The Transport Train - All Aboard!

Assigned	Tuesday, November 26, 2013
Program Due	Tuesday, Decemeber 10, 2013, 8:00am
Weight	10%
Updates	

Objectives

To gain experience

- implementing a generic class,
- using text file input and output,
- using command line arguments, and
- implementing and using exceptions.

Project Description

A train pulls boxcars, which contain “things” like cargo and people, from one destination to another. A train can pull many boxcars containing "things" of different types at the same time. For example, a train may pull five boxcars of people and ten boxcars of cargo, with the boxcars in any order. Note, however, that an individual boxcar will contain only one type of "thing" at any given time (for example, all people or all cargo).

A train has some basic behaviors no matter how many and what type of boxcars it is pulling, such as departing a station, arriving at a station, loading, unloading, adding boxcars, and removing boxcars. Because all boxcars have the same behaviors, it is appropriate to design and implement a generic boxcar class.

In this project, you will use text file I/O to read train commands from a file. These commands will tell a train what operations (e.g. unload) to perform. The operations that the train performs, along with associated information (see below), will be written to what we will call a ***log file***. A log file is essentially a history of the operations that the train has performed.

You will also be designing and using your own exception classes. The conditions under which exceptions are thrown are listed in the details below.

Project Specification Details

Project 5 is invoked with the following command line arguments ***in the following order***.

1. Name of the train's city of origin
2. The train's minimum allowable speed
3. The train's maximum allowable speed
4. Maximum allowable number of boxcars
5. The name of the train's command file
6. The name of the train's log file

You may assume that the command line arguments are of the appropriate types, integers are not negative, and that train speeds are consistent (i.e. maximum speed >= minimum speed).

Your Tasks

1. Design and implement a generic class to model a box car
2. Design and implement a class to model cargo
3. Design and implement a class to model a person
4. Design and implement a class to model a train that can simulataneously pull both cargo and person boxcars
5. Write main( ) and all its helper methods in a class named Project5.java
6. Design, implement, and appropriately use exception classes to detect and handle error conditions (see the specific conditions below)
7. Design and implement any other classes/interfaces you feel are necessary

The Command File

The operations that the the train is to perform will be read from a text file, the name of which is a command line argument. ***Each command and each data item will be on a separate line in the command file***. The formats of the PRINT, ARRIVE, DEPART, SPEEDUP, SLOWDOWN, ADDCAR, REMOVECAR, LOAD, UNLOAD, and QUIT commands are described below. City names, person names, and unique cargo labels may be multi-word strings. Commands and unique government-issued IDs do not contain spaces.

1. PRINT – Outputs the following information to the log file.

A. The train's current speed

B. The train's minimum speed

C. The train's maximum speed

D. The train's current position (e.g. "Traveling from New York to Chicago")

E. The train's current number of boxcars that it is towing

F. The train's maximum number of boxcars that it can tow

G. Details of each boxcar being towed by the train. Each boxcar will give

i. It's unique boxcar ID, which is really just its index in the train (e.g. the 1st boxcar being towed has an ID of 0, the 2nd has an ID of 1, etc.)

ii. The details of each "thing" being transported ***in sorted order,*** or a message that the boxcar is empty (hint: find a sort method in the Java API to use -- don't write your own)

a. Cargo items are listed in sorted order by their unique string label. All attributes of the cargo item must be listed on the same line.

b. People are listed in sorted order by their unique government-issued ID.All attributes of the person must be listed on the same line.

2. ARRIVE – The train reaches its destination and is parked, ready for loading, unloading, adding boxcars, and removing boxcars.

3. DEPART <city> -- The train begins moving at its minimum speed towards its destination.

4. SPEEDUP <miles per hour> -- The train increases its speed by the specified number of miles per hour.The resulting speed is the current speed increased by the "speed up" miles per hour.

5. SLOWDOWN <miles per hour> -- The train decreases its speed by the specified number of miles per hour.The resulting speed is the current speed decreased by the "slow down" miles per hour.

6. ADDCAR <CARGO | PERSON> <maximum number of elements> -- The train gets a new boxcar added to it at its end. The only valid types are CARGO and PERSON. The maximum number of elements is unrelated to any attributes. For example, one piece of cargo is simply one piece, regardless of its size.

7. REMOVECAR <boxcar ID> -- Removes a boxcar from the train as long as the boxcar is empty. The boxcar ID is the index in the train (e.g. the 2nd boxcar being towed will have an ID of 1.). If a car is removed from the middle of the train, then all boxcars behind it should be reassigned a new ID based on their new position in the train.

8. QUIT – Indicates the end of the command file. The program exits gracefully.

The formats for LOAD and UNLOAD are different for each type of "thing" the boxcar is carrying.

1. LOAD (CARGO | PERSON)
    A. Boxcar carrying cargo: LOAD CARGO <boxcar ID> <unique cargo ID> <weight> <height> <width> <length>
        i. Where the boxcar ID is a valid integer, cargo ID is a string, and the other attributes are positive integers. The unit of measure for weight, height, width, and length is irrelevant.
    B. Boxcar carrying people: LOAD PERSON <boxcar ID> <unique government-issued ID> <person name> <age>
        i. Where the boxcar ID is a valid integer, government ID and name are strings, and age is a positive integer.
2. UNLOAD
    A. Boxcar carrying cargo: UNLOAD <boxcar ID> <unique cargo ID>
        i. Where the boxcar ID is an integer, and the cargo ID is a String
    B. Boxcar carrying people: UNLOAD <boxcar ID> <unique government-issued ID>
        i. Where the boxcar ID is an integer, and the government ID is a String

You can see a sample command file here. But please note this is just a ***sample*** command file!!!

The Logfile File

All required output and error messages from train commands must be ritten to a logfile using text file I/O. The name of the logfile is a command line argument. The required output is described in the Command File section above. The PrintWriter class can be used to write data to a file. Here's an example.

```
1  PrintWriter out = new PrintWriter(new File(filename));
2
3  out.printf("%10s%10s%10s\n", "Hello", "Output", "file");
4  out.printf("%10s%10s%10s\n", "Hello2", "Output2", "file2");
5  out.printf("%-10s%-10s%-10s\n", "Hello", "Output", "file");
6  out.printf("%-10s%-10s%-10s\n", "Hello2", "Output2", "file2");
7  out.close();
```

All text printed must be neatly aligned so as to be readable (see the sample log file given below).

A sample logfile has been generated based on the sample command file.

Exception Conditions

Your code must handle the following exception conditions. Any other error conditions that your code introduces should also be checked. Be sure to use appropriate exceptions together with try/catch blocks to separate error detection from error handling.

1. If there are any errors with the command line arguments, display an appropriate error message to the screen and exit your program. There are two command line argument errors that you should check for: that the number of command line arguments is correct and that each of the files (command and log) can be opened appropriately. You need not check for any other command line arguments.
2. If an attempt is made to load an item onto a full boxcar, log an appropriate message and continue to the next command in the file.
3. If an attempt is made to load the wrong type of item into a boxcar (i.e. a person into a cargo boxcar), log an appropriate message and continue to the next command in the file.
4. If the train is commanded to move to faster then it's maximum speed, log an appropriate error message, and continue to the next command in the file.
5. If the train is commanded to slow down below it's minimum speed, log an appropriate error message, and continue to the next command in the file.
6. If an attempt is made to unload a non-existent “thing”, log an appropriate message and continue to the next command in the file.
7. If the train is commanded to do something that makes no logical sense given its current state, the command should be disregarded (but still logged), an error message should be logged, and the next command read from the file.

Requirements, Restrictions, and Assumptions

1. All classes in this project should be in a package named proj5.
2. There are **NO** weight or volume restrictions when loading “things.” The only restriction is on the number of items loaded.
3. “things” do NOT automatically unload when the train arrives.
4. You may assume that all command file data is valid. In particular, this means that
   A. the format of the command file will be valid as specified above,
   B. the values of all data in the command file will be valid (e.g. positive), and
   C. the command file will contain no invalid commands.

   It DOES NOT mean that all commands make common sense. See Exception Conditions above.
5. All commands and their parameters must be logged to the logfile as they are being executed, even if the command is not actually carried out.

6. Appropriate object-oriented design is required. As discussed in class throughout the semester, this includes, but is not limited to:
   A. private instance variables only
   B. limited public interfaces for all classes - don't write unnecessary methods
   C. maximum code reuse
   D. use of try-throw-catch for exceptions
7. Appropriate top-down design for `main( )` and its helper methods is required.
8. Sanity Check: How much code in your train class, boxcar class, cargo class, and person class would have to change if a third kind of "thing" to be carried (say cattle or goats) were added to this project? If the answer isn't ***"NONE!!"*** then you're doing something wrong.

Project Policy

This project is considered a ***CLOSED*** project. This means ...

1. You should try as much as possible to complete the project by yourself.
2. You may get assistance only from the TAs, CS Help Center tutors, or instructors.
3. You must document all outside help you get as part of your file header comment.

As usual, you ***MAY NOT***...

1. copy anyone else's code,
2. have someone else write your code for you,
3. submit someone else's code as your own,
4. look at someone else's code, or
5. have someone else's code in your possession at any time.

Such offenses are violations of the student code of academic conduct.

Grading

See the course website for a description of how your project will be graded.

Project Submission

1. **submit** the .java files for all classes you implement. To submit a single .java file (for example, your Project5.java file), do so as follows.

```
submit cs202 Proj5 Project5.java
```

To submit more than one file at once, simply list them.

```
submit cs202 Proj5 Project5.java Person.java Another.java
```

To submit all of your .java files at once,

```
submit cs202 Proj5 *.java
```

The order in which the files are submitted doesn't matter. However, you must make sure that all files necessary to compile your project are listed. You need not submit all files at the same time. You may resubmit your files as often as you like, but only the last submittal will be graded and will be used to determine if your project is late. For more information, see the Projects page on the course website.

You can check to see what files you have submitted by typing

```
submitls cs202 Proj5
```

Remember — if you make ***any*** change to your program, no matter how insignificant it may seem, you should recompile and retest your program before submitting it. Even the smallest typo can cause compiler errors and a reduction in your grade.

Avoid unpleasant surprises!