Homework 14: Train Data Structure

Release date: Sunday, April 22nd, 2018

Due date: Sunday, April 29th, 2018

Goals:

- Create a data structure that adds elements using references
- Manipulate a data structure to change or remove elements
- Traverse a LinkedList data structure to find elements

Prerequisites:

- Understand the LinkedList data structure

Introduction:

Dynamic data structures are one of the most helpful tools for programmers managing both the planning process and the management of memory that they use for their programs. As you've seen in lecture, the **ArrayList** allows the user to add elements without needing to worry about the size of the underlying array: if the array needs more space, a new array will be created to handle new data without our guidance. Similarly, a **LinkedList** allows the user to add elements without needing to worry about the constraints of an array: the LinkedList allows for *virtually* unlimited elements to be added without changing constraints (limited, of course, by your computer's resources.

This homework assignment introduces a dynamic data structure that combines Arrays and LinkedLists. The **Train** is a type of LinkedList that allows the user to connect standalone units of **Cars**.

Description

The Train is a LinkedList composed of two different objects: the Car and the Train.

Car

The Car object is, itself, a *static data structure*. Each Car object holds a single array of elements referred to as **cargo**. Like a standard train car, this array will remain the same size for a single instance of a Car object, hence the static description for this structure. A position in the cargo array is considered "empty" if it is null.

Each Car also has a reference to the car behind it, if one exists, referred to as the **nextCar**. This will be your means of creating a Train, as a train is composed of a sequence of train cars linked together.

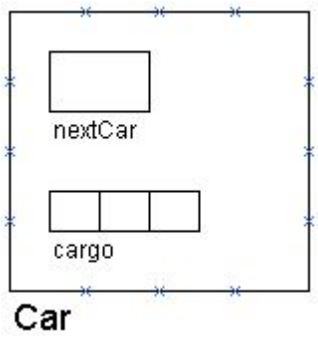

Figure 1. Simple diagram of a Car object

Methods

A Car is capable of performing actions without being connected to a Train, so we will be providing it with some methods of its own. The table below describes the functionality of the Car's methods; see the skeleton for more details including edge cases.

Method	Description
Car	The Car's constructor simply starts the instance with an empty array capable of holding a number of elements according to the parameter.
addCargo	Adds the passed object's reference to the Car's cargo array. Cargo should be added to the first "empty" (*null*) space in the Car. You should not move any of the other cargo.
getCargo	Returns a reference to this Car's array of cargo.
capacity	Returns the maximum capacity of this Car (the length of the cargo array).
size	Returns the current number of objects stored in this Car.
get	For a passed index, this method returns a reference to the object at that location in the Car's cargo array.

set	For a passed index and object reference, this method replaces the object at the provided index in the Car's cargo array with the passed object (null is permitted).
isFull	Returns a boolean value based on whether each position in the Car's cargo array is non-empty.
setNextCar	Sets this Car's nextCar to reference the Car that will be following this Car..
getNextCar	Returns the Car linked to by this Car.

Train

The Train object will be our *dynamic data structure*. Unlike Cars, Trains are limited to holding objects of type Car. However, the bonus is that different Cars can hold different types of objects, meaning you can have a Train carrying many kinds of cargo.

Like the LinkedList you've learned about, the Train maintains two references: a **head** and a **tail**, referred to respectively as the **engine** and the **caboose**. The engine will refer to the first car in the Train, while the caboose will refer to the last car in the Train. Both will refer to null if no cars exist yet.

Lastly, we'll want the Train to be capable of adding cargo without requiring the user to manually create new Cars. To do so, we specify a **newCarSize** for Cars that are created automatically when there is no space on the Train.

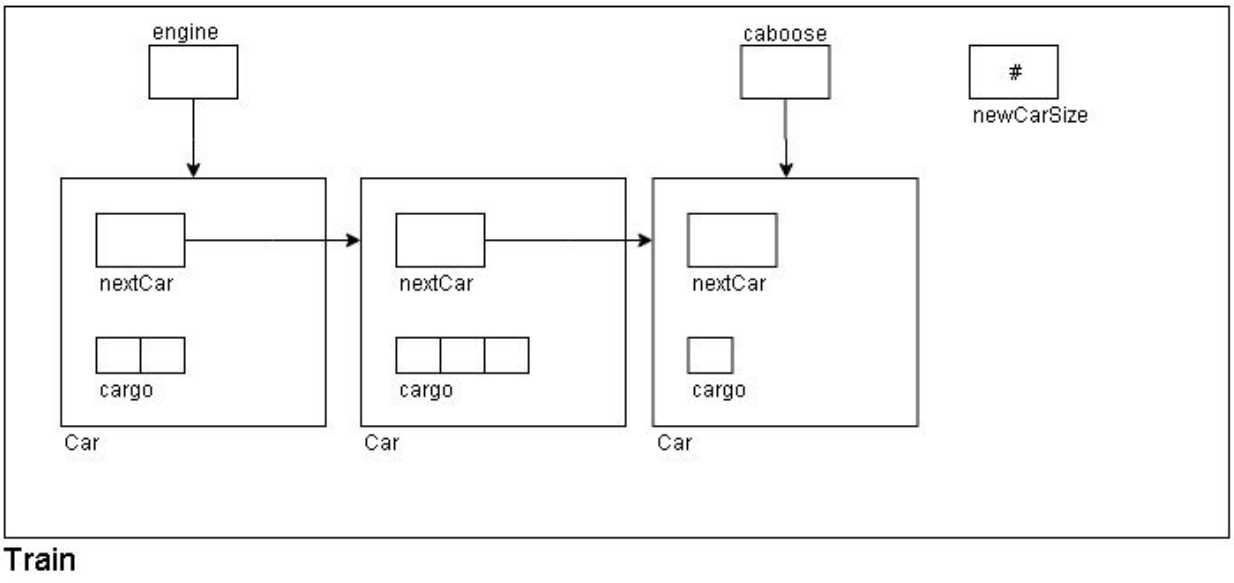

Figure 2. Simple Diagram of a Train Object with three cars

Methods

The Train will have some of the expected functionalities of a LinkedList, but with the added twist of working with Cars. The table below describes the functionalities of the Train's methods; see the skeleton code for more details, including edge cases.

Method	Description
Train (no arguments)	Creates a new Train with no Cars. engine and caboose should point to null.
Train (one argument)	Creates a new Train with the array of Cars provided. The Cars will be added to the Train in the same order that they are presented in the array. You may assume that none of the Cars are already connected to another Car.
getNewCarSize	Returns the size of new Cars created by the Train.
setNewCarSize	Sets the size of new Cars created by the Train.
getEngine	Returns the first Car in the Train.
getCaboose	Returns the last Car in the Train.
size	Returns the number of Cars in the Train. You will need to iterate through the Cars in this Train to count them.
get (one argument)	Returns the Car at the given position in the Train.
get (two arguments)	Returns the cargo in the Car at the given position in the Train.
appendCar (car argument)	Adds the passed car to the end of the Train, making it the new caboose, and the Car previously linked to that car should be removed. You do not need to check if this Car is already a part of the Train.
appendCar (no argument)	Adds an empty new Car to the end of the Train according to the size specified by newCarSize.
set	For a passed index and object reference, this method replaces the object for the specified Car at the provided index in the Car's cargo array with the passed object.
addCargo	Adds the parameter to the first open space in a Car on the Train. You do not need to check if the parameter is already on the Train. Creates a new Car if there is no Car on this Train that can store the cargo.

emptyTrainCars	Removes references to all objects from each Car on the Train. The Cars should remain connected and the Train should continue to have access to them.
boilerUp	Prints “Toot toot!\n”.

FullCarException

FullCarException is an exception class that has been implemented for you. You will be required to throw the FullCarException when attempting to add cargo to a full car. The message for the exception is up to you.

Constraints

To receive credit for this assignment, you must abide by the following constraints:

- You may not create any additional classes besides Car, Train, and FullCarException.
- You may not create additional fields for any of the Car, Train or FullCarException classes.
- You may not modify the declarations of the fields for either Car or Train.

Tips

- The methods are presented above and in the skeleton in the order by which the assignment is most easily completed.
- It is recommended that you try each of your methods after you develop them.
- Many of the methods for Train will require you to iterate through elements that are linked by object references rather mapped to positions in an array. Once you figure this out, you will be able to complete all the methods. Consider how you can keep track of your location on the Train and know when you’ve reached the end as you traverse it.

Grading

Submission
For this assignment, you are required to submit **ONLY** the following source files for automatic grading:

- Car.java
- Train.java

Your code must compile or you will not receive credit for this assignment.

Distribution
Points for this assignment will be distributed as follows:

Car:	20%
Train	80%

Methodology
Methods for Car and Train will be automatically tested. As an added challenge, the same Cars and Train will running non-stop through all your test cases. This means that if you fail a test, there is a significant chance your remaining test cases will *derail* and consequently fail. You'll want to fix the first test case failure reported before resubmission.

The testing system used may report these failures out of order, but they have been numbered in the case this should occur (with test 01 occurring first, then 02, and so on).