CS100M Fall 2005 Project 6 due Thursday 12/1 at 6pm

Submit your three files `CarQueue.java`, `CarDEQueue.java`, and `RailYard.java` on-line in CMS under Project 6 before the project deadline. For java code, be careful to submit the .java file, not the .class file. Both correctness and good programming style contribute to your project score.

You must work either on your own or with one partner. You may discuss background issues and general solution strategies with others, but the project you submit must be the work of just you (and your partner). If you work with a partner, you and your partner must register as a group in CMS and submit your work as a group.

Important note on using DrJava
Turn off the file backup feature of DrJava! Go to menu item **Edit-->Preferences**, choose the last category, "**Miscellaneous**," then **un**check the box for "**Keep Emacs-style Backup Files**."

Objectives and Background

In this project, you will learn how to use arrays and inheritance in Java. Once again, practice *incremental development and testing* (develop and test one class, or even one method, at a time).

We all have stood in lines before, at the grocery store, at the bank, etc. Another word for this kind of line is the *queue*, where each person *enters a queue from the back* and *gets out (gets served) at the front of the queue*. The relative order of the people in a queue does not change (assuming that no one is rude or leaves before getting served). In the fields of Computer Science and Operations Research, we call such queues first-in-first-out, or FIFO. You may have observed another kind of queues at work at a rail yard: the train (or the track) can be considered a queue of rail cars. Operations at a rail yard may include *adding a car to the back or the front* of a train on the track. Similarly, a car may be *removed from the front or the back* of the train. We call this kind of queues where add/remove operations can be performed at both the front and the back a *doubly-ended queue*. Notice the "is-a" relationship? A doubly-ended queue *is a* queue.

In this project, you will work with queues of rail cars (trains) and do a simulation of a simplified rail yard where some tracks are FIFO and some permit operations at both the front and the back of a train. Class Car represents a rail car and is given. Class CarQueue is a queue of rail cars (an array of objects) and is partially implemented. You will implement the entire class CarDEQueue, which represents a doubly-ended queue of rail cars. You need to complete the RailYard class for simulating the operations and you should write additional code in the RailYard class (or create another class) to test your CarQueue and CarDEQueue. Class DataEmptyException is for handling "exceptions," e.g., the error when one tries to get an object from an empty queue. Don't worry about the code in DataEmptyException, but follow the examples in class CarQueue for "raising an error" related to an empty queue when you write class CarDEQueue. The set of skeleton code compiles but does not execute. As in previous projects, your submitted code should at least compile.

The study of queues is very important. For example, when you plan/design a layout of a bank, how do you decide on the number of teller counters? Should you have one queue that feeds several counters or should each counter have its own queue? Computer simulations of queues are often used to help answer such questions. The variables are the number and types of queues, the rate of service, demand, and customer behavior (some people wait in the same queue until served, others change to a different queue at the slightest perception that another queue is moving faster, yet others will leave and come back later). Many of the variables have probabilistic properties and the resulting queue simulations

can be quite complex! Courses in Operations Research, Computer Science, and Management Science examine queues from different viewpoints.

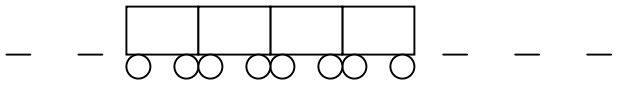
Class Car

Read the code for this given class. Each Car has a type (passenger, dinning, or cargo) and a unique id. Notice that the unique ids are generated automatically using a static variable. Note also that we've given you a toString method. This will be handy later for testing.

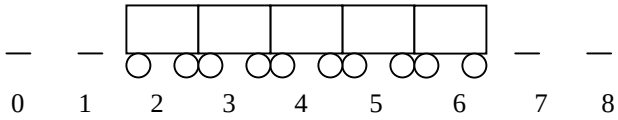
Class CarQueue

A CarQueue is an array of Car objects in a normal queue. Remember that in a regular queue, FIFO, so you only add to the rear and remove from the front. The skeleton for this class is given and you will need to complete the implementation. Below are some diagrams to explain how a queue works.

Let  represent a Car object. Now suppose we have a CarQueue with a capacity of 9. An intermediate picture of the queue where there are 4 Cars may be

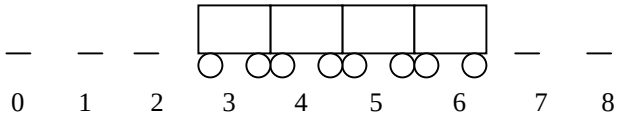


We keep track of the front of the queue with the variable *f* so *f* = 2 in this case. We will use variable *r* to indicate the position after the rear of the queue so *r* = 6. Now if you add a Car to the queue, then the queue looks like



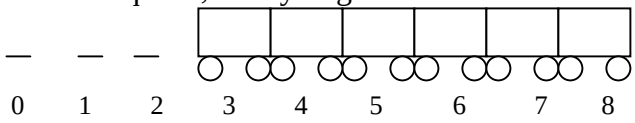
f = 2, *r* = 7

Note that you *do not shift the objects forward*. Simply update the variables *f* and *r* to keep track of the front and rear of the queue. If you remove a Car from the queue, then

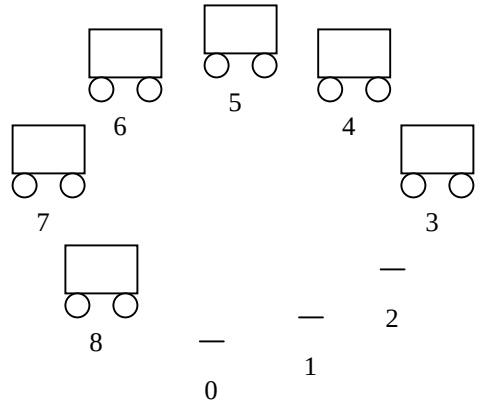


f = 3, *r* = 7

Say you add 2 more Cars to the queue, then you get

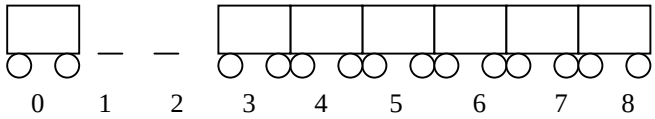


with 6 Cars in the queue but the queue isn't full yet! Its capacity is 9 so there're 3 more spots that can be filled. This means that we need to implement the queue in a *circular* fashion!



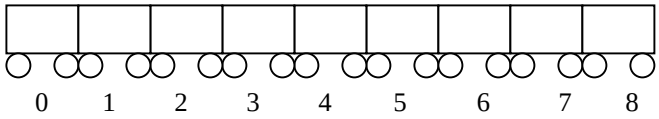
f = 3, ***r* = 0**

Note that the queue is still a normal 1-d array—we just *think* of the arrangement as circular. So if you add a Car to the (back of the) queue, the newly added Car will actually occupy the spot at index 0 and the picture will be



$f = 3, r = 1$

The definitions of variables *f* and *r* have not changed; they still mark the front and the rear as defined originally. Now if you add two more cars, then the queue is full:

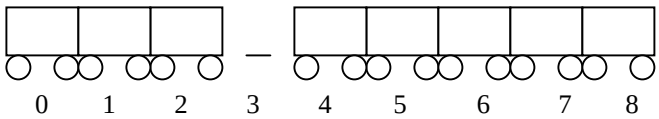


$f = 3, r = 3$

But wait, isn't the status $f = r$ for a full queue the same as the status for an empty queue?! (Think about the initialization: when you create a queue that is initially empty, you'll set $f = 0$ and $r = 0$.) *You need to come up with an implementation strategy to make sure that you can distinguish an empty queue from a full queue.* There're different ways to do it—we leave the decision up to you. Please be sure to comment your code clearly to explain your implementation.

If you need to add a Car to a full queue you will first double the capacity of the queue. (This is a convention: double the capacity when the queue is full; don't add capacity one or two or some other arbitrary number of spots at a time.) When you double the capacity, in actual fact you are creating a new array with twice the capacity and then copying over all the elements while *maintaining the relative order of the objects.*

Back to the current picture above… If we now remove a car from the queue, as usual we remove the Car from the front of the queue so the picture will become



$f = 4, r = 3$

The specification of the entire class in given in the skeleton code. Please read and follow the specifications carefully.

Test your class! Now create a main method in a Test class, instantiate a CarQueue object, and start adding (removing) objects to (from) the queue. Below are some example statements that you may use in the main method of the Test class. Add the tests to the main method one at a time so that it's easy to identify any errors. Write more test code beyond our suggestions below!

```
// Make queue with capacity 3
  CarQueue q= new CarQueue(3);
// Insert 3 Cars to get a full queue.  The 3 Cars have id 0, 1, 2
  for (int k=0; k<3; k++)
    q.insertLast(new Car());
// See the first Car in the queue, which as id 0
  System.out.println(q.first());
// Remove the first Car in the queue
  Car junk= q.removeFirst();
// Now the first Car in the queue is the one with id 1, display it
  System.out.println(q.first());
```

```
/* Now insert another Car.  This should work if your circular implementation
 * is correct.
 */
  q.insertLast(new Car());
/* Now insert another Car.  This should work if your queue doubles its
 * capacity correctly.
 */
  q.insertLast(new Car());
```

Class CarDEQueue

A CarDEQueue is an array of Car objects in a doubly-ended queue. Remember that a doubly-ended queue is one where you can add/remove objects from the front and the rear. A doubly-ended queue is a queue so use inheritance in Java to implement this class. The specification details are given in the file CarDEQueue.java. You need to write the entire class! (We can't let you get out of CS100 knowing only how to fill in the blanks!)

The indexing of the doubly-ended queue works in the same way as described for the normal queue. Again, you must implement the queue in a "circular" fashion.

Test your CarDEQueue *class!* Add more test code to the Test class you have created for testing. For example, instantiate a CarDEQueue object, add and remove a number of Car objects. Be sure to test both the inherited and locally defined methods.

Class RailYard

A RailYard has an array of CarQueue objects representing the tracks. Read the provided code carefully and fill in the five (short) methods we have left blank for you. Some of the methods have to do with *polymorphism*. The main method in the railYard class starts a simple simulation. Following the given code, you can (and should) add more operations at the RailYard to fully test your classes CarQueue and CarDEQueue.

Eample output from the RailYard simulation using our example solution is given in the file sampleOutput.txt.

Note: We have nested a class Instruction inside class RailYard. You are not responsible for nesting classes in CS100.

Final Words

Throughout the project we have made suggestions for testing your code. Testing is done incrementally—as you develop the code—and not all at the end. The idea is that you want to make sure class X is correct before you write a class Y that depends on X. Sometimes it is tempting to rush through coding without stopping to test, but that will burn more time in the long run because you will have to deal with many confounding errors in your buggy code when you finally get around to trying to run your program. We hope you have learned how to develop some simple test code. Remember *incremental development and testing* in your future work!