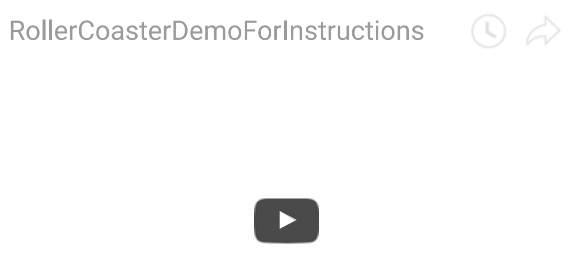
PROJECT 4 ROLLER COASTER SPRING 2016

Home 〉CS 2114 Spr 2016 〉March 14 - March 20 〉Project 4 Roller Coaster Spring 2016

Project 4 Roller Coaster Spring 2016

Due Date: Thursday, April 7 @ 8:00 AM.



Introduction

For this project, your job is to write a program which tracks and visualizes how people ride a roller coaster.

Imagine a scenario, the entire CS 2114 class is visiting an amusement park and wants to ride the roller coaster. The class splits up into small groups of friends who want to ride together. We call these groups waiting parties as these groups wait in a line to ride the roller coaster. Our goal as a part of this project is to simulate and visualize how these waiting parties progress through a ride line. A number of scenarios arise. Your job is to think through these scenarios before you begin the implementation. This project aims to help you understand and develop systems where data flows between a number of classes and different classes communicate and collaborate with each other. Below is the description of the basic scenario, try to get the big picture first and then delve into the details of each part.

First, they wait in line in groups called waiting parties. When they try to get in line, their height is checked, and they are turned away if anyone in their party is too short, unless they are willing to leave that person behind.

The parties are seated in the roller coaster in order. Consider a case where the roller coaster has 5 vacant seats and there is a Waiting Party with 7 persons. Two things can happen. There will be some parties who are willing to be split up and will do so to fill the current coaster. If a party is unwilling to split up, then the coaster will roll on without them. In this program, the parties behind them cannot skip ahead! When a willing party gets split up, we assume that the remaining people in that party continue to be willing to split up if needed. A party that is too big for the entire coaster and unwilling to split up would cause the program to hang!

To track this movement of people, we are going to use several data structures that you've learned over the course of this semester, as well as several other topics such as interfaces,

NAVIGATION

Home
- My home
  - My profile
  - Current course
    - CS 2114 Spr 2016
      - Participants
      - March 14 - March 20
        - Reading Quiz 8 Spring 2016
        - Lab 08: Pre-Lab Activity
        - Lab 08: Post-Lab Activity Additional Methods
        - Lecture 11 Queue Introduction
        - Student Music Survey - Data Collection for Project 5
        - Practice Problems 8
        - Lecture 12 Queue Implementation and List Introduction
        - Project 5 Group Project Team Members Submission
        - Project 4 Roller Coaster Spring 2016
        - Project 4 Sample Input Files
  - My courses

SETTINGS

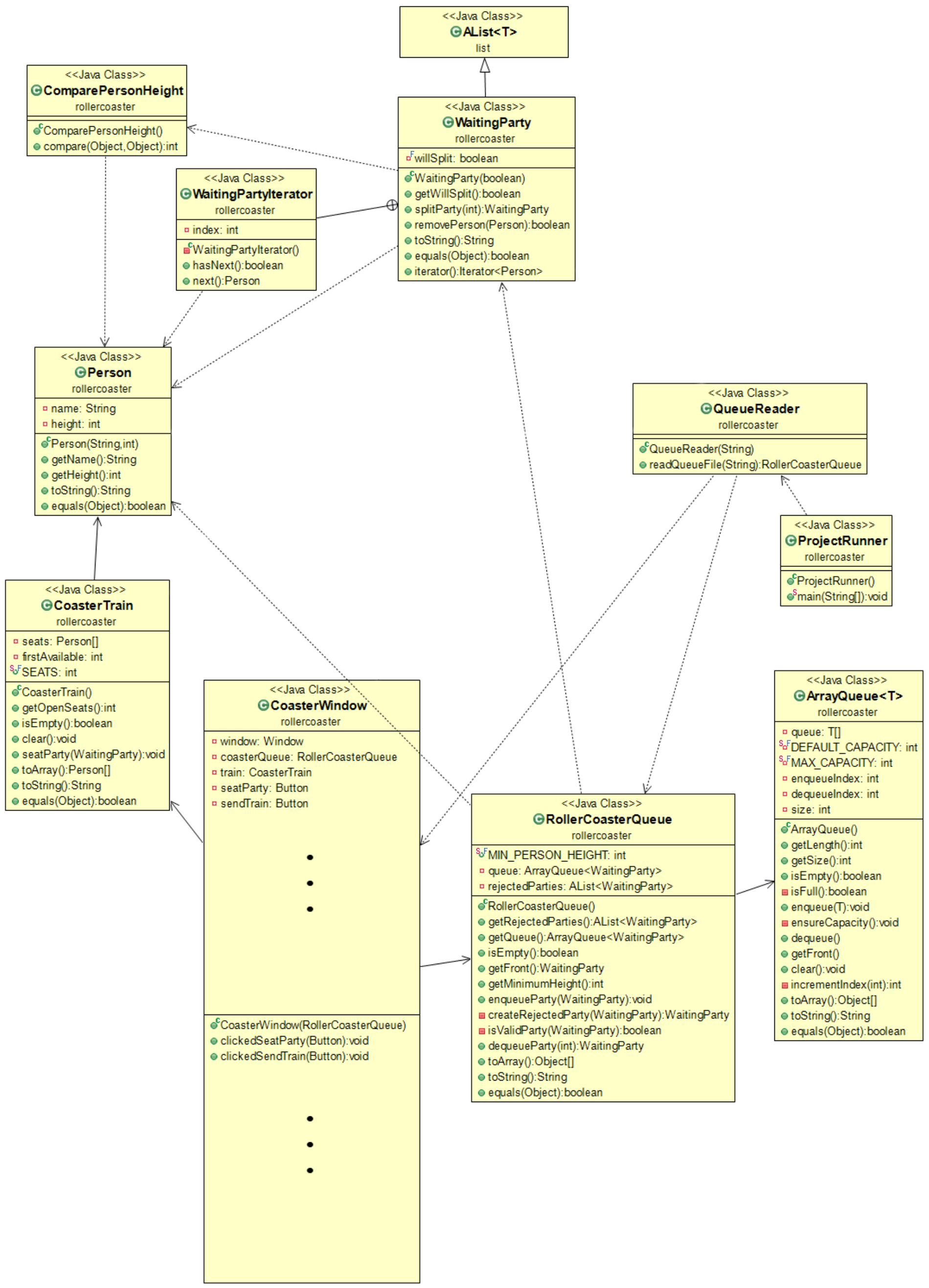
- Course administration
- My profile settings

extension, encapsulation, `Window`, and File I/O.

For this last guided project, we want you to try to figure it out yourself as much as possible. We will start by giving you the broadest overview of the project, and slowly fill in the gaps. It is strongly recommended that you always read through all the instructions, before you start coding.

As you write each class, write its Junit tests. It is recommended that you use Test Driven Development, which requires you to code the tests before you write the class. Fully test each class before you move on to the next one, (i.e., practice piecemeal development). If you don't test from the ground up, it will be difficult to tell where a bug is coming from when combining many classes and functions.

UML



Overview of Classes

Project name: RollerCoaster

package name: rollercoaster

An overview of the classes is given in this section, step-by-step implementation suggestions for each class is provided after the General Guidelines below.

Person: These objects contain a string, for a person’s name, and an int, for their height in centimeters.

ComparePersonHeight: This Comparator class can be used to order persons with height as the primary key and name as the secondary key. Persons will be ordered by height, if any two are the same height then those two are ordered by name.

AList: Import this list class from CarranoDataStructuresLib. You do not implement it.

WaitingParty: WaitingParty extends AList. WaitingParties hold the groups of Persons who want to ride together. A WaitingParty can be split if the people in the party are willing to be separated. Data from the input files will be parsed into WaitingParty objects.

ArrayQueue: This data structure implements QueueInterface with a circular array implementation. It provides default queue behavior, such as enqueue, dequeue, getFront, and isEmpty.

RollerCoasterQueue: This data structure represents the line of WaitingParties for the CoasterTrains. RollerCoasterQueue encapsulates an ArrayQueue. RollerCoasterQueue offers unique queue behavior because depending on how many seats are available on the coaster, it may ask a WaitingParty to split. It also offers roller coaster-specific enqueue behavior by checking for the height of people before allowing them in the roller coaster line.

CoasterTrain: A CoasterTrain holds an array of Persons. It seats people into the array and keeps track of available seats.

CoasterWindow: This object is the front end. Here we build our window, its buttons, and render the RollerCoasterQueue and CoasterTrain on the window in a meaningful way.

QueueReader: The QueueReader parses the input data from a comma separated value, (.csv), file and parse it into WaitingParties then tie the CoasterWindow and RollerCoasterQueue together.

ProjectRunner: The ProjectRunner class begins the program by creating a QueueReader and telling it which file to look at.

General Recommendations

Order of Implementation

Be mindful of the order in which you implement the classes. Write and run tests for each class as you go. Some classes depend on each other, so be sure one is correct before implementing the next. For example, WaitingParty and CoasterTrain depend on Person. Once you have your JUnit tests working, implement parsing an input file and test more data.

Fields

- Should typically be private. If they are final static literals, which could not be changed by giving public access, make them public.
- Make them final if you expect them to be set once and never again.
- Make them static if you are pre-declaring a literal value, (constant), for use in a class, such as information which will always be the same for all objects of a class.

Getters && Setters

- For this course, always provide getters for fields. We expect your getters to be declared as “get”+<fieldname>().
- Only provide setters if you expect those fields will need to be changed, aka if they correspond to fields which are not final and are mutable.

Methods

- Declare methods public when a user of your class will need access to it.
- Declare helper methods private as much as possible. If it's only used inside of the class, make it private.
- If you're copying and pasting code, make a method.
- If you're writing complicated code, which could be error prone, make it a method and test it separately. Examples include large logical statements, or edge cases. Note that you can have multiple test methods for one method.

equals()

For this project, different instances of a class are equal if their fields are logically the same. For example, if you instantiate two different Persons, but give them the same name and height, they are equal.

You are expected to write an `equals()` method for most classes. This is a good programming habit. It is common that you will want to use `equals()` to debug, so you should preemptively have it in your code. Not having `equals()` implemented can easily cause hard to detect silent bugs. It is easy to write code that depends on equals without realizing it's not implemented and is actually testing for identity which seems like it works, but is logically incorrect. Be sure to correctly code your `equals()` methods.

toString()

If you're building a complex string which is dependent on multiple facets of an object instance, use a StringBuilder. StringBuilder is much lighter on memory, and will make your code faster. When you operate only on String objects, since they are immutable, every time you change them a new string is instantiated in memory. That can get to be expensive in terms of memory usage!

You are expected to write a `toString()` method for most classes. This is also a good programming habit. It is common that you will want to use `toString()` to debug, so you should preemptively have it in your code.

The summary of the toString() output for each class is as follows:

Person: Jane Doe 80cm

Waiting Party: Party of size 2 will not split. [Nick Doe 80cm, Alex Doe 76cm]

ArrayQueue: [Nick Doe 80cm, Alex Doe 76cm]

RollerCoasterQueue:(note this one has a newline character in it)
Line with minimum height 65cm.
[Party of size 2 will not split. [Nick Doe 80cm, Alex Doe 76cm], Party of size 4 will not split. [Taylor 120cm, Kendal 80cm, Criminally Tall Man 500cm, Avery 117cm]]

CoasterTrain: [Nick Doe 80cm, Alex Doe 76cm, Taylor 120cm, Kendal 80cm, Criminally Tall Man 500cm, Avery 117cm]

Guard against NullPointerException

If you ever call a method on an instance of an object, be aware that you may need to check whether the object is null and handle it as a special case. A special case where a method returns null can also provide useful information about the state of the object, such as a queue returning null when empty.

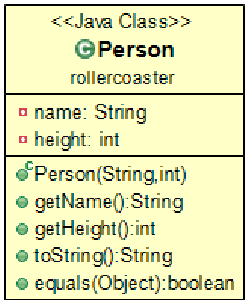
Include a statement similar to the following before you use parameters to avoid NullPointerExceptions.

```
if(parameter == null) {
    ///handle special case
}
```

Implementation Guidelines Class-by-

Class

Person



public String toString()

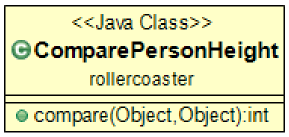
Using a StringBuilder, concatenate the name, the height value, and the abbreviation for centimeters.

Sample output: "Joe Cocker 80cm"

public boolean equals(Object obj)

Two Person objects are considered equal when their name and height is the same. You will need to write your own `equals` method.

ComparePersonHeight

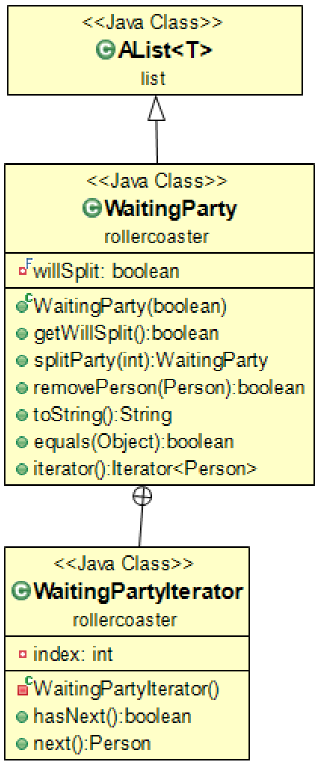


This Comparator object is very similar to the example you will see in the lecture notes. Its generic type is Object so that it can be used on arrays returned from toArray(). Notice if two people have equal height, then they are then ordered based on name.

This class does not need a constructor. This is the only method needed in this class. You will have to add the javadoc.

```java
public class ComparePersonHeight implements Comparator<Object> {
    @Override
    public int compare(Object o1, Object o2) {
        int diff = ((Person)o1).getHeight() - ((Person)o2).getHeight();
        if (diff == 0) {
                diff = ((Person)o1).getName().compareTo(((Person)o2).getName());
        }
        return diff;
    }
}
```

WaitingParty extends AList<Person> implements Iterable<Person>



HINT: When you extend a class, an explicit call to `super()` will reuse the superclass' constructor inside of yours. Likewise you want to use the `AList` superclass' `getEntry()`, `getLength()`, and other members.

public WaitingParty splitParty(int maxSize)

`maxSize` tells the party how many people will be allowed on the ride. If the requested `maxSize` is larger than or equal to this party's size, simply return `this` because the party meets the requirement.

If the party is too large and will not split, return null.

If we know that the party will split and it is larger than maxSize, then we have meaningful work. Make a new `WaitingParty` named `splitParty`. Take `maxSize` number of people from `this` and place them in `splitParty`. Make sure to remove from the front of the list, since those people were "first" in line.

Return `splitParty`. Notice that we are returning a party with some of the people from our original party, and we also have removed them from this original party. We effectively have two parties now, and there should be no person existing in both parties.

public String toString()

Using a StringBuilder, concatenate "Party of size ", length, " will ", optional "not ", "split. " followed by the super's `toString()` method which will print each person in the list. Note that there is a space between the period after split and the second toString(). For example:

"Party of size 2 will not split. [Joe Cocker 80cm, Susy Q. 77cm]" "Party of size 1 will split. [Taylor 120cm]"

public boolean equals(Object obj)

WaitingList inherits an `equals()` method from `AList`, but if we want to ignore the order of the people, we need to override that. Implement your standard `equals()` method and use the following code after you've checked all the special cases. The code below takes advantage of the sorting provided by Java's `Arrays` class so that we can compare two lists in the same order regardless of which order they are stored in WaitingParty. `Arrays` will sort them using a Comparator object before checking for equality.

```
        ...
        Object[] items = this.toArray();
        Object[] otherItems = other.toArray();
        ComparePersonHeight comparer = new ComparePersonHeight();
        Arrays.sort(items, comparer);
        Arrays.sort(otherItems, comparer);
        return Arrays.equals(items, otherItems);
        ...
```

public Iterator iterator()

Notice that in addition to extending AList, WaitingParty implements Iterable. The iterator() method returns a new WaitingPartyIterator that can be used to iterate over Person objects in a WaitingParty. This implements the iterator method expected in the Iterable interface. Client code of WaitingParty has the option of using this iterator to traverse over the Person objects in WaitingParty as seen in Carrano Java Interlude 5, in particular section JI5.12. Note that each call to iterator() returns a new WaitingPartyIterator, so you'll want to store a single iterator in a variable when working through the list.

private class WaitingPartyIterator implements Iterator

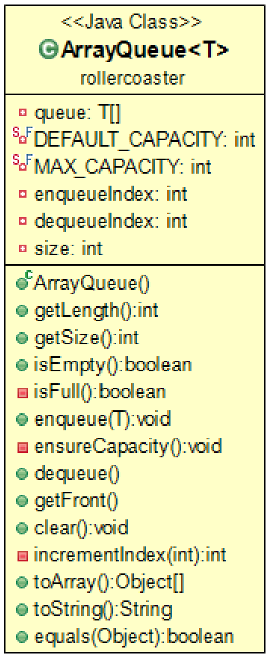
WaitingPartyIterator is a nested class inside of WaitingParty (see UML in above WaitingParty Section). Notice that in addition to extending AList, WaitingParty implements Iterable. So, WaitingPartyIterator is a nested class that implements Iterator. An iterator steps through or traverses a collection of data. Iterators are introduced in Carrano's Java Interlude 5.

private WaitingPartyIterator()
The constructor should initialize the index to 0.

public boolean hasNext()
Checks if the WaitingParty has a next Person object based on whether index < getLength().

public Person next()
Iterates to the next Person Object in the Waiting Party. If hasNext() is true get the value with the the current index and increment index, otherwise throw a NoSuchElementException.

ArrayQueue<T> implements QueueInterface<T>



```
private static final int DEFAULT_CAPACITY = 10;
private static final int MAX_CAPACITY = 100;
```

The queue can initially hold DEFAULT_CAPACITY objects. If more objects need to be added, then the size can be expanded until it reaches MAX_CAPACITY objects. If MAX_CAPACITY is exceeded, then throw an IllegalStateException.

The parties waiting to ride need to be tracked in order, since they are given to us in the order which they have arrived in line,(i.e., FIFO order). As you've recently discussed in class, the best way to track people in line is to use a queue.

See your textbook for information on how to implement Circular Array ArrayQueues! Notice this implementation keeps track of the enqueueIndex (AKA backIndex) and dequeueIndex (AKA frontIndex) and uses those to determine the number of items in the queue. A field for the size can be maintained to avoid the calculation, but is unnecessary and with each change to the queue must be updated which increases risk of error. This project requires the implementation of the methods size(), equals(), toArray() and toString() in addition to those described in the text.

A full ArrayQueue will have one empty slot. This null is to keep your dequeue and enqueue indices from overlapping. They only overlap when the queue is empty. When you create your array, build it using DEFAULT_CAPACITY + 1. This way, the back end array will have an actual size of 11, but will be only able to hold 10 elements.

An ArrayQueue implementation needs to correctly handle the cases where the indices wrap around. Be sure to write test cases for such scenarios, and your textbook has useful examples.

private void ensureCapacity()

This optional helper method can be used to upgrade the size of the queue when the queue is full. The new size is twice as large as the old size. Note that size is different from length. So a queue of length 11 will be full when the size is 10, and after upgrading the size in ensureCapacity(), the new length will be 21 with the capacity to hold 20 elements.

private int incrementIndex(int index)

This is an optional helper method, but will help you with the circular queue wrapping logic. Anywhere you increment an index in the array, you should use this method or similar modular arithmetic.

```
return ((index + 1) % queue.length);
```

public void clear()

Think of clear as hitting the reset button on your object. You can also think of the clear method as a way to re-call the constructor. When you implement clear, try to match your constructor, (i.e., clear should result in the object being in the same state as a newly constructed object).

Example: Say you were clearing a LinkedStack like in project 3. Instead of calling remove over and over until it is empty, simply set your fields to their default values. This executes faster, and has the same logical result.

public Object[] toArray()

Queues generally do not have a method that lets us cycle through the elements like we can in a list. Implementing a `toArray()` method gives client code the option of accessing the data in the queue without interfering with the integrity of the queue.

Your `toArray()` cannot just return the underlying array from `this`. The returned array needs to have its entries start at index 0, which might not be true of the circular array. The returned array should only be as large as this.size, meaning it has no extra empty slots.

public String toString()

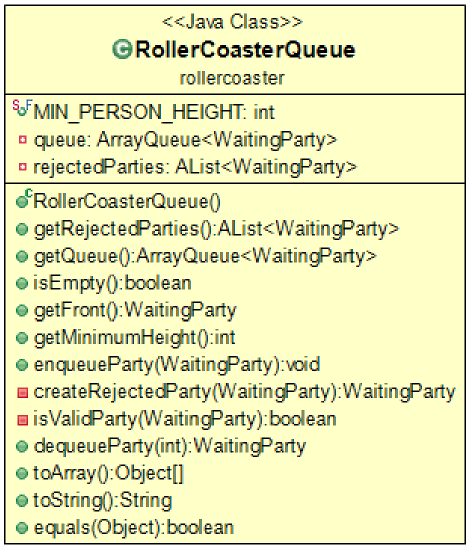
The `toString()` method will also need to iterate through the contents of the queue. Concatenate the `toString()` of each Object in the ArrayQueue separated by a comma and space, except for the last object in the queue. StringBuilder is especially efficient in these situations where we are building a string in a loop. Surround the whole thing with brackets, []. If the queue is empty, return "[]".

For example: [Nick Doe 80cm, Alex Doe 76cm] public boolean equals(Object obj)

For two ArrayQueues to be equal, they have to contain the same elements in the same order. With a circular queue implementation, it is possible the elements are not stored in the same indices in each underlying array. So, compare the sizes of the two ArrayQueues and then iterate over the contents. For example:

```
for (int i = 0; i < size(); i++) {
        T myElement = queue[(frontIndex + i) % queue.length];
        T otherElement = other.queue[(other.frontIndex + i) % other.queue.length];
        if (! myElement.equals(otherElement)) {
                return false;
        }
}
```

RollerCoasterQueue



<<Java Class>>
RollerCoasterQueue
rollercoaster
MIN_PERSON_HEIGHT: int
queue: ArrayQueue<WaitingParty>
rejectedParties: AList<WaitingParty>
RollerCoasterQueue()
getRejectedParties():AList<WaitingParty>
getQueue():ArrayQueue<WaitingParty>
isEmpty():boolean
getFront():WaitingParty
getMinimumHeight():int
enqueueParty(WaitingParty):void
createRejectedParty(WaitingParty):WaitingParty
isValidParty(WaitingParty):boolean
dequeueParty(int):WaitingParty
toArray():Object[]
toString():String
equals(Object):boolean

A RollerCoasterQueue will enqueue waiting parties based on the input file data, and riders who are too short will not be allowed to get in line. When the user clicks the "Seat Parties" button, the RollerCoasterQueue will dequeue a waiting party according to seats available on the current roller coaster train, `CoasterTrain` class, and move parties through the line. We choose to encapsulate ArrayQueue here, because we want to restrict access to its enqueue and dequeue methods. We don't want users of this class to be able to get around the height check, for example.

`public static final int MIN_PERSON_HEIGHT = 96;`

Helper methods

Note that just because we provide less instructions for this class doesn't mean it's a trivial class. You should pay attention to any duplicate or complex code you're writing and put it in a private helper method instead. The helper methods listed in the UML are suggestions and by no means the way you must split up your methods. The `isValidParty(WaitingParty)` method could be used to determine if a given party will be able to be enqueued without splitting. The `createRejectedParty(WaitingParty)` method could be used to separate out the allowed and rejected people from a party.

public void enqueueParty(WaitingParty party)

Before adding a party to the `RollerCoasterQueue`, `enqueueParty()` will check the height of the persons in the party. If anyone is too short, and the party is willing to split, the too-short people will be put in a new `WaitingParty` called `rejectedParty` and the rest of the original party can be enqueued. The `rejectedParty` will be stored in the `AList` of `rejectedParties`. In the case that someone is too short and the party is not willing to split, then the entire party is placed in the `AList` of `rejectedParties` and never gets enqueued, (i.e., never gets to ride).

public WaitingParty dequeueParty(int seatsAvailable)

In this method, we want to check the front `WaitingParty` of the queue to see if they can board the coaster train, and if so, return that boarding party and take them out of the queue.

If the queue is empty, return null. If the front party is too big and unwilling to split, then they cannot board the current roller coaster train, so return null. If the front party is the correct size already, then they can be dequeued and returned. If the front party is too large but is willing to split, then the split party can be returned.

public Object[] toArray()

Queues generally do not have a method that lets us cycle through its elements, but the elements in the queue can be returned in an array so that client code can work with them one by one. The waiting parties are held in this class, and when we want to represent each `WaitingParty` with a circle in our window, we will need to iterate through the queue elements.

public String toString()

Build a string that begins with this introduction specific to Roller Coaster Queues

```
"Line with minimum height " + MIN_PERSON_HEIGHT + "cm.\n"
```

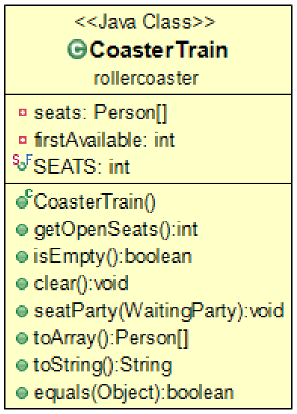
and then use the `toString()` method for the ArrayQueue. Note that this String has a newline character in it, and there is no space after the period and before the newline.

For example:
Line with minimum height 9cm.
[Party of size 2 will not split. [Nick Doe 80cm, Alex Doe 76cm], Party of size 4 will not split. [Taylor 120cm, Kendal 80cm, Criminally Tall Man 500cm, Avery 117cm]]

public boolean equals(Object obj)

Two `RollerCoasterQueue` objects are considered equal if their queue of waiting parties and their `AList` of rejected parties are both equal. Hint: both of those classes have equals methods of their own.

CoasterTrain



```
public static final int SEATS = 20;
```

We will track the first available seat index. It will also be used to calculate the remaining number of empty seats. So that we can use that number to ask the queue for a party with a size which will fit into the coaster.

public void seatParty(WaitingParty party) throws IllegalStateException

If the party provided is too large to seat, throw an IllegalStateException. Otherwise, place everyone in the party in order into this coaster train. Make sure that you do NOT remove them from their original party.

public void clear()

Think of clear as hitting the reset button on your object. You can also think of the clear method as a way to re-call the constructor. When you implement clear, try to match your constructor, (i.e., clear should result in the object being in the same state as a newly constructed object).

public String toString()

Concatenates the items in the underlying array starting at item 0 and separates them by a comma and a single space. The entire `toString()` should be surrounded by square brackets.

For example:
[Nick Doe 80cm, Alex Doe 76cm, Taylor 120cm, Kendal 80cm, Criminally Tall Man 500cm, Avery 117cm]

public boolean equals(Object obj)

Two `CoasterTrain` objects are considered equal if they have people of the same name and height seated in the same order.

Input: Read from File

Our design approach is to read in from a file with values separated by commas. The format of the input file will look something like the following:

```
true
Billy J. Thornton, 176, Bob E. Evans, 172, Joe Cocker, 80
false
Susy Q., 155
```

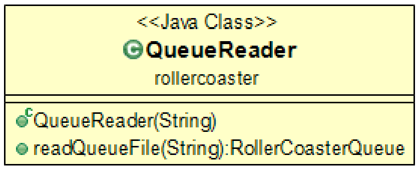
The odd lines tell us whether the party is willing to split and the even lines correspond to the list of people for that waiting party.

Their names are always followed by their height in centimeters.

You can use http://www.fakenamegenerator.com/ to build your own input.

If NumberFormatExceptions are thrown during input, it is a likely indication that input file format is not the same as what the code is expecting.

QueueReader



Place the created "input.txt" file in your root directory. When you look at your project folder's structure, the input file should be on the same level as the src and bin folders. After adding it, you may need to refresh your project file structure by hitting F5 or right clicking the project and choosing Refresh.

The `QueueReader` class will read the input and begin the `CoasterWindow`. In the constructor, you will complete parsing. Pass your parsed input into another class, `CoasterWindow`, to run your program.

You can download some small sample input files from the course page, and place them in your project folder. You need to parse the csv file and extract data from the file. The structure of the csv file is mentioned above in the previous section, (Input: Read from CSV File).The `QueueReader` Class will read from the input file It is responsible for parsing the input about the parties of people waiting to ride on a roller coaster.

public QueueReader(String fileName)

Call readQueueFile to populate a RollerCoasterQueue.

Instantiate a new `CoasterWindow` with the recently filled RollerCoasterQueue as its parameter.

public RollerCoasterQueue readQueueFile(String fileName)

Declare and instantiate a local RollerCoasterQueue, this will store the data in the input file.

Declare a new `Scanner` named file and instantiate it to be a new Scanner, with a new `File` as its parameter. Give your new `File` the String fileName parameter. This should be done inside a try statement because a FileNotFoundException will be thrown if the provided filename is incorrect. In the catch statement, you can print the stack trace of the error and exit the program.

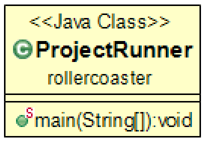
If the filename was correct, you can begin parsing the file. Keep a counter for the lines as you loop through it using hasNextLine(), you can use that counter to know if it's an odd or even line. The odd lines have Booleans to flag whether the party is willing to split. The even lines have a list of people.

You can use String's `split()` function to parse the lines. The delimiter is "," but if you want to also account for spaces you can use a regular expression " *, *" This will ensure your heights can easily be converted with `Integer.valueOf(String)`.

Hint: You may get an empty string if you use your scanner's `nextLine()` method to parse the people's names. That's because after parsing the previous integer, the scanner is still "looking" at the end of the previous line. Call `nextLine()` a second time to get their full names.

Reference your previous labs and textbook for File I/O guidance as needed.

ProjectRunner

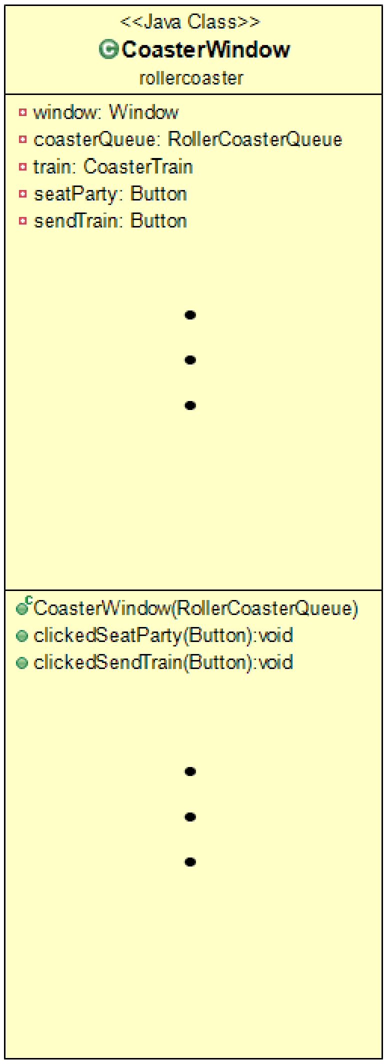


Once again, you will use a ProjectRunner class to begin your program. Typically, your main method will be very small. Instead of parsing input in the main method, it is better practice to place it in a Reader class, in this case `QueueReader`.

public static void main(String[] args)

If a single argument has been provided, we will use that as our input filename. Otherwise, instantiate a `QueueReader` class and provide the default "`input.txt`" filename.

CoasterWindow



<<Java Class>>
CoasterWindow
rollercoaster
window: Window
coasterQueue: RollerCoasterQueue
train: CoasterTrain
seatParty: Button
sendTrain: Button
CoasterWindow(RollerCoasterQueue)
clickedSeatParty(Button):void
clickedSendTrain(Button):void

This class diagram leaves space for where you will write and name your own private variables and methods.

CoasterWindow is responsible for the visualization of the WaitingParties moving through the line which you can see in the Youtube video at the top of this page. A circle is used to represent every WaitingParty in the queue. The size of the circle is proportional to how many Persons are in the WaitingParty. The party's willingness to split up to get on a ride is represented by red (no) and green (yes) filled circle. At any point the user can click the "Send Train" button to move the coaster along so that a empty coaster comes up to be filled. This may be especially useful if you have a stubborn waiting party that will not split at the front of the line!

You will be in charge of determining how to best decompose this problem into public and private methods, field and local variables, etc. The public methods and private field variables shown in the UML are the only methods and variables you are required to have. This section will provide some hints for you about how to split up the work.

Overview

In this class, we build a `Window` and title it 'Roller Coaster Ride'. On the south side, we will have a `Button` named `seatParty`, and a `Button` named `sendTrain`. Every time `seatParty` is clicked, one party will be taken from the queue and seated in the `CoasterTrain`. Every time `sendTrain` is clicked, the currentCoaster will be cleared and reused to seat parties to simulate the departure of a filled coaster and the arrival of an empty coaster. You won't be able to run this class and check your code until you complete the `QueueReader` class.

Field Variables

At minimum, the class will need to keep track of these objects:

- A Window which displays the visuals.

- A RollerCoasterQueue which has the parties waiting in the queue for the ride.
- A CoasterTrain which has people seated and ready to ride.
- Two buttons, one which says "Seat Party" and one which says "Send Train".

You will most likely need more field variables than these. If you find yourself creating many objects of the same kind, such as CircleShapes to display the parties, then having an AList or an array of that object may be helpful, such as an AList called partyCircles which contains a CircleShape for each party displayed in the queue.

As you code, consider the pros and cons of having field variables accessible to the whole class versus local variables. If you need to access a variable from multiple methods, such as your queue, then having a class variable is a good idea. If you only need to interact with the variable temporarily or a single time, such as the vertical separator Shape, then a local variable might be fine.

Constants

It might help your readability if you keep some constants to determine placement and size of Shapes on the screen. For example, instead of setting a CircleShape to have a radius of `10*party.getLength()`, you can have `CIRCLE_RADIUS_FACTOR*party.getLength()` so that any time you need to change the factor of your radius, you only have to change where you set your constant instead of changing every single time you typed "10".

```
public static final int QUEUE_STARTX = 100; //The horizontal position which starts
public static final int QUEUE_STARTY = 150; //The vertical position which starts yo
public static final int CIRCLE_RADIUS_FACTOR = 10; //To multiply with a WaitingPart
```

These are only examples and you should make constants as you see fit. Even if you don't need a constant for a value, it's best to have a variable of some sort rather than hardcoding the number.

public void clickedSeatParty(Button button)

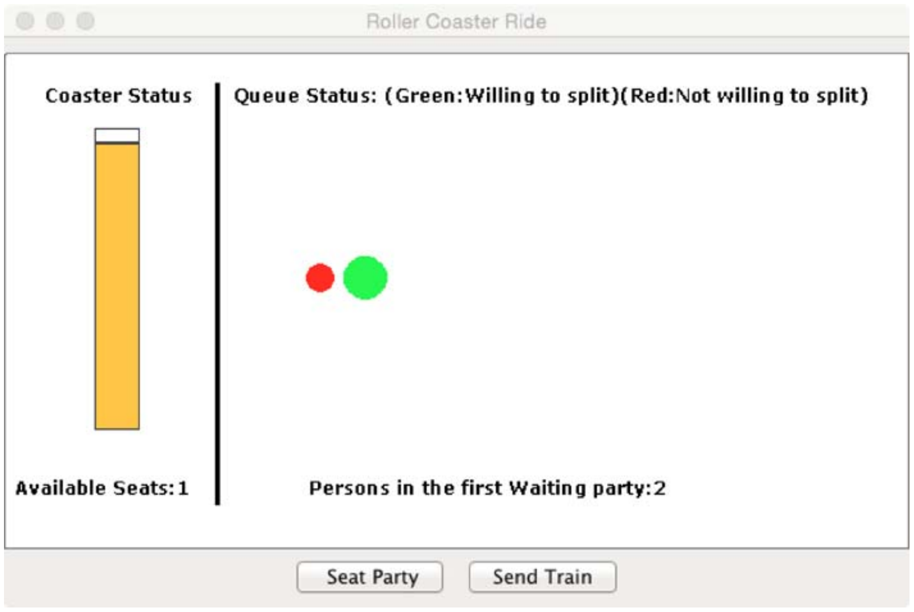
When the Seat Party button is hit, the ride will attempt to seat the `WaitingParty` at the front of the `RollerCoasterQueue`. After a party is seated, the queue, coaster, and window display should all be updated to match.

When the `CoasterTrain` is nearly full, the `RollerCoasterQueue` should handle some of the logic for you in its dequeueParty() method. Consider what it means if a null is returned from dequeueParty(). If the train cannot seat any more people before starting the ride, then the Seat Party button should be disabled so the user can't hit it again. The button should also be disabled if the queue is completely empty, i.e. the ride is over. Watch out here for NullPointerException bugs in your code.

public void clickedSendTrain(Button button)

When the ride is sent, all the people on board can then be cleared off and the train is ready to start boarding more riders from a clean slate. This should be updated visually as well. The user should also be able to hit the Seat Party button again if it was disabled before sending the train.

Note that the user can choose to send the train even if the train is not yet full.


Roller Coaster Ride
Coaster Status
Queue Status: (Green:Willing to split)(Red:Not willing to split)
Available Seats:1
Persons in the first Waiting party:2
Seat Party
Send Train

Drawing Shapes in the window

As the program progresses, it will need to render and update Shapes in the window to represent the underlying data. There are two common approaches to changing how a window updates: keep track of the Shapes in the window and change their position, color, size, and remove them as needed, or erase all objects from your screen and put everything back on in its correct position. Be sure to read the class API on `GraphWindow` to be aware of the methods at your disposal.

In either case, it might be helpful to have helper methods manage some of the updating, especially updating that needs to be done more than once or in more than one case.

The Shapes you will need to have on your window can be seen in the Youtube video and in the screenshot above. Some Shapes, such as the text "Coaster Status", the queue status text, and the vertical black bar, will not change during the program. Others, such as the available seat number, the colored Shapes representing parts of the coaster, and the waiting party size text, will need to be updated often.

The CircleShapes representing the waiting parties should be proportional to how big each party is, should be colored based on willingness to split, and should be centered vertically when compared to each other. The colored Shape representing how full the coaster train is should be proportional to the number of seats open and should fill from the bottom. You might find it useful to use two shapes for this bar and either overlay one on the other or update them both to meet at the right spot.

End of the ride

Once the queue is empty, and all riders have been sent, you've completed your program! Display “Ride Closed!” where your Queue's CircleShapes were, and disable both your seatParty and sendTrain buttons.

Optional: Additional Implementations

You are not expected to implement the following ideas for this project, but you should be starting to think more and more about the design issues of your projects. Here are some concepts to consider:

For the roller coaster we could also use a 2D array to model the rows of seats, but this is not really necessary. Regardless of how we store the seats, we could also allow riders to sit out of order and next to certain people.

Add visualization of the people in the seats of the coaster.

What if we didn't want everyone to be stuck waiting behind a party that wouldn't split? How would we change our design? This could be a practical use of a deque that allows us to dequeue and enqueue on both sides of the queue!

Currently all the people get queued from the file at once. An improvement would be to allow riders to continually join the queue. This would take advantage of our circular array implementation of a queue. Would you use a Button? User provided parties?

Submission status

Submission status	This assignment does not require you to submit anything online
Grading status	Not graded
Due date	Thursday, April 7, 2016, 8:00 AM
Time remaining	The due date for this assignment has now passed