Project 2: Tarmart Simulation

DUE: Sunday, Mar 6th at 11:59pm

Extra Credit Available for Early Submissions!

Basic Procedures

You must:

- Fill out a readme.txt file with your information (goes in your user folder, an example readme.txt file is provided)
- Have a style (indentation, good variable names, etc.) and pass the automatic style checker (see P0).
- Comment your code well in JavaDoc style AND pass the automatic JavaDoc checker (see P0).
- Have code that compiles with the command: **`javac *.java`** in your user directory.
- Have code that runs with both commands below:
  - `java Tarmart [numLines] [seed]`
  - `java TarmartGUI [seed]`

You may:

- Add additional methods and variables, however these methods **must be private OR protected**.
- Add additional nested, local, or anonymous classes, but any nested classes must be **private**.
- Add methods **required** by parent abstract classes and methods required by interfaces you must implement (duh).

You may NOT:

- Make your program part of a package.
- Add additional ***public*** methods, variables, or classes. You may have public methods in ***private*** nested classes.
- Use any built in Java Collections Framework classes in your program (e.g. no **`LinkedList`**, **`HashSet`**, etc.).
- Create any arrays anywhere in your program. You *may not* call **`toArray()`** methods to bypass this requirement.
- Alter any method signatures defined in this document or the template code. Note: "throws" is part of the method signature in Java, don't add/remove these.
- Add **`@SuppressWarnings`** to any methods unless they are private helper methods for use with a method we provided which already has an **`@SuppressWarnings`** on it.
- Alter provided classes or methods that are complete (**`TarmartGUI`**, **`Tarmart`**, etc.).
- Add any additional import statements (or use the "fully qualified name" to get around adding import statements).
- Add any additional libraries/packages which require downloading from the internet.

Setup

- Download the `p2.zip` and unzip it. This will create a folder `section-yourGMUUserName-p2`;
- Rename the folder following the same instructions from P0 and P1.
- Complete the `readme.txt` file (an example file is included: `exampleReadmeFile.txt`)

Submission Instructions

- Make a backup copy of your user folder!
- Remove all test files, jar files, class files, etc.
- You should just submit your java files and your readme.txt
- Zip your user folder (not just the files) and name the zip `section-username-p2.zip` (no other type of archive) following the same rules for `section` and `username` as described above.
  - The submitted file should look something like this:

```
001-krusselc-p2.zip --> 001-krusselc-p2 --> JavaFile1.java
                                            JavaFile2.java
                                            ...
```

- Submit to blackboard.

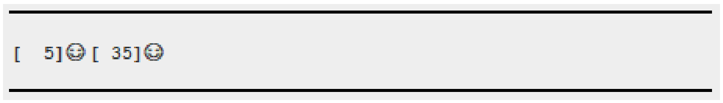
Grading Rubric

Due to the complexity of this assignment, an accompanying grading rubric pdf has been included with this assignment. Please refer to this document for a complete explanation of the grading.

Overview

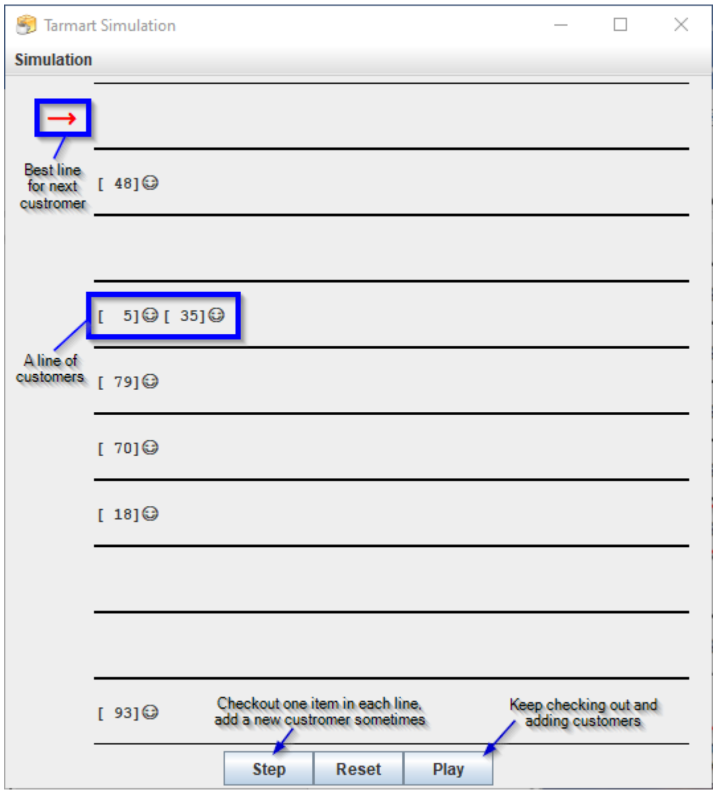
You have been asked to simulate the new Tarmart Priority Checkout System™. This new priority system uses passive RFID tags placed into all Tarmart merchandise, and a large RFID scanner placed at the entrance to the checkout lines, to determine how many items each customer has when they checkout.[1] A survey of Tarmart customers has shown that their customer base is mostly over the age of 55, and that they *emphatically* do not like to bag their own items. This means that they can't just walk out of the store (like at the new Amazway stores), but have to have each item individually bagged by a checkout clerk which is very slow. The new Tarmart Priority Checkout System™ will display a giant red Tarmart Arrow™ overtop the line with the fewest items to be checked out giving their customers a clear indicator of which line they should join.

You are going to write the backbone of the Tarmart store simulation to see the efficiently of customers with carts `[  5]☺` in a line:



```
[  5]☺[ 35]☺
```

As we have discussed, Tarmarts have multiple lines and customers would like to choose the shortest line based on the number of items that need to be checked out:


Tarmart Simulation
Simulation
Best line for next custromer
A line of customers
Checkout one item in each line, add a new custromer sometimes
Keep checking out and adding customers
Step
Reset
Play

```
[ 48]☺

[  5]☺[ 35]☺
[ 79]☺
[ 70]☺
[ 18]☺


[ 93]☺
```

Every minute ("step") the clerk at the front of each line will bag one item from the front customer. When a customer has no more items to bag, they will leave the line. Every so often, another customer will get into line, picking the line chosen by the giant red Tarmart Arrow™.

IMPORTANT: While you may examine code from the textbook and our slides on linked lists, queues, and priority queues you may not *use* the code from the textbook *as* your solution. For this project, all work must be your own implementation based on your personal understanding of the material. Just don't copy code and you'll be fine.

tl;dr You're building a store simulator. You can't use code from anywhere but your brain (textbook and slides != brain).

[1] Tarmart is always striving to passionately fulfil their moto to their customers: **Expect money? Pay better!™**

Step 0a: Planning Your Code

The Tarmart simulation is based on a priority queue of Tarmart checkout lines. [**Please reread that last sentence**.] The checkout lines are queues, and the priority queue is needed to pick the best line for the next person to join. Below is a "plan of attack" for this assignment:

We have a large amount of code for the simulation already written. This code ***depends on the data structures you will be writing***. All of the GUI and other "simulation" features are written in their entirety (and come with complete JavaDocs). The following classes are provided:

- **`Tarmart.java`** - Represents an entire Tarmart and runs a text-based simulation.
- **`TarmartGUI.java`** - A graphical representation of the Tarmart with simulation controls.

There is a large amount of data structure code we will be completing to support the simulation shown. Do not start writing code yet! Start by getting a good sense of what you'll be doing:

- **`TarmartCustomer.java`** - This class represents a single customer with their cart. It has the comments describing each of the methods we need to complete (see Step 1).
- **`ThreeTenList.java`** - This generic class extends Java's **`AbstractList`** class (see Step 2). This is a simple linked list similar to the one we built in class, but it is "doubly linked".
- **`ThreeTenQueue.java`** - This generic class extends **`ThreeTenList`** and implements Java's `Queue` interface (see Step 3). You may want to review how inheritance works in Java using either your CS211 materials or the resources listed in the appendix of this document.
- **`TarmartLine.java`** - This class is a type of **`ThreeTenQueue`** specifically designed to order a group of cutomers (it **`extends ThreeTenQueue<TarmartCustomer>`**) and it requires some additional features (see Step 4). You may want to review the **`Comparable`** interface Java using either your CS211 materials or the resources listed in the appendix of this document.
- **`ThreeTenPriorityQueue.java`** - This class extends **`ThreeTenQueue`** to turn it into a priority queue. In class we have discussed a number of ways to implement a priority queue with a list. This class has Big-O restrictions you must meet which will determine how you implement it.

tl;dr You need to understand the code base before trying to work on this assignment.

Step 0b: Reading the Code Base + JavaDocs

It is HIGHLY RECOMMENDED that you write your JavaDocs for this project during this stage. If you have to learn how the code works, you might as well write it down! If you (1) learn how the code works, (2) finish an entire project, then (3) go to write how the code works, you'll just be redoing work you already did ☹ Writing JavaDocs at the *beginning* means you will have a full understanding of the code base ***as you work***.

Remember you can use the inheritdoc comments for inheriting documentation from interfaces and parent classes! Never heard of inheritdoc? Here's the short version: if you override a method, you can inherit the documentation from the class/interface where the documentation is written. You just combine the @Override annotation with the following JavaDoc comment, like so:

```
/**
 *  {@inheritDoc}
 */
@Override
public String toString() {
      //code here...
}
```

Saves time. Very simple.

Step 1: TarmartCustomer Class

The TarmartCustomer class should be very easy, but you might need to brush up on some material from CS211. See the code skeleton for more information.

Big-O Requirements Summary:

Component	*Big-O*
All methods	O(1)

Step 2: ThreeTenList

- Java Reference: https://docs.oracle.com/javase/8/docs/api/java/util/AbstractList.html

You are going to implement a doubly linked list class called **`ThreeTenList`**. This class extends the Java Collections Framework class **`AbstractList`** which you can read about in the above API reference[2]. We walked through much of the code for singly linked lists in class, so just extend that a little bit to handle the idea of doubly linked lists.

Big-O Requirements Summary:

Component	*Big-O*
Constructor, size(), add(T)	O(1)
All other methods	O(n)

Step 3: ThreeTenQueue

- Java Reference: https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html

Read the above documentation CAREFULLY as it will help you understand what you are trying to do. It's a bit long, but you can do it!

Big-O Requirements Summary:

Component	*Big-O*
All methods	O(1)

Step 4: TarmartLine

This class is a type of **`ThreeTenQueue`** specifically designed to order a group of people (it **`extends ThreeTenQueue<TarmartCustomer>`**) and it requires some additional features.

1. Tarmart lines have ID numbers. Supporting methods:
   - **`TarmartLine(int id)`** - Constructor for a line with a given ID.
   - **`int getId()`** - Returns the ID of the line.
2. Customers are in the line and those customers have items to be checked out. Additionally, a line can be compared to another line based on the number of items which have not been checked out. Supporting methods:
   - **`int itemsInLine()`** - Sums up all items for all people in the line. For example, if there are two people in line, one with 16 items and one with 54 items, this method will return 70.
   - **`int compareTo(TarmartLine otherLine)`** - Compares one grocery line to another based on the number of items each line. If the two lines are tied, it compares them by their id. More explanation can be found in the JavaDoc comments for this method and in the Appendix:Background section of this document.
3. Tarmart clerks can process items from the front of the line. Supporting method:
   - **`void processItem()`** - removes one item from the first person in the line. If the person at the front has no more items, they are removed from the line.

[2] Normally a linked list would extend **`AbstractSequentialList`**, but we have overridden the iterator for you, so that you can practice the basic code for linked lists.

4. Lines can be represented as a string. Supporting method:
   - **`String toString()`** - Converts the line to a string in the following format:
     `[id]: TC([#]) = [#] shopper(s) with [#] item(s)`

     For example, if Line 4 has two people in it (one with 5 items and one with 35):

     ```
     [  5]☺[ 35]☺
     ```

     The string would be:
     `4: TC(5) TC(35) = 2 shopper(s) with 40 item(s)`

     Similarly, an empty line with ID 1 would produce the string:
     `1: = 0 shopper(s) with 0 item(s)`

     For more examples, see **`exampleRun.txt`** (included in the p2 zip).

Big-O Requirements Summary:

Component	*Big-O*
Constructor, getId(),processItem()	O(1)
compareTo()	O(n+m)
All other methods	O(n)

Step 5: ThreeTenPriorityQueue

Since our priority queue is built out of a linked list, you'll need to implement an **`update()`** method which accepts an item and "updates" its place in the queue (i.e. update its priority). Lower priority items should be at the front of the queue (since we'll want to pick the *shortest* lines, not the *longest* lines in our grocery store).

Big-O Requirements Summary:

Component	*Big-O*	*Notes*
element(), peek()	O(1)	
update()	O(n)	
All other methods	---	← Do not ask about this or post publicly on Piazza about this. *You, by yourself, with no one else*, need to determine and implement the best Big-O based on the restrictions given.

tl;dr You're writing the code for the five classes outlined above in Step 1-5. You should look at the provided code skeleton for more guidance and some restrictions on the Big-O.

Requirements Summary

An overview of the requirements are listed below, please see the grading rubric for more details.

- **Implementing the classes** - You will need to implement required classes and fill the provided template files.
- **JavaDocs and Style** - You are required to write JavaDoc comments for all the required classes and methods. Check provided classes for example JavaDoc comments. Style requirements are the same as other projects.
- **Big-O** - Template files provided to you contains instructions on the REQUIRED Big-O runtime for many methods. Your implementation of those methods should NOT have a higher Big-O.

tl;dr You need to: write code, document it, meet the big-O reqs, and test your code.

How To Handle a Multi-Week Project

While this project is given to you to work on over several weeks, you are unlikely to be able to complete this is one weekend. It has been specifically designed such that the "stages" correspond with each of a set of lectures. We recommend the following schedule:

- Step 0-1 (Planning and **`TarmartCustomer`**): Before the first weekend
  - Familiarize yourself with the code and get the **`TarmartCustomer`** class out of the way.
  - Complete the JavaDocs for ALL classes as part of ***this stage***.
- Step 2-3 (**`ThreeTenList`** and **`ThreeTenQueue`**): First weekend
  - At this point you've learned about both linked lists and queues in class and done your readings, so implement the doubly linked list and the queue.
- Step 4 (**`TarmartLine`**): Before the second weekend
  - The Tarmart customer line is a simple extension to the queue class, so do that next and you can visit the GTAs if you got stuck on the **`ThreeTenQueue`** over the weekend.
- Step 5 (**`ThreeTenPriorityQueue`**): Second weekend
  - You've had a lot of practice with queues now and some review in class, so try hitting the priority queue class and wrap up your work! Do lots of testing, testing, testing...

This schedule will leave you with an entire week + weekend to get additional help and otherwise handle the rest of life ☺ Also, notice that if you get done early in the third week, you can get extra credit! Check our grading rubric PDF for details. Otherwise you'll have plenty of time to test and debug your implementation before the due date.

tl;dr Don't try to do this project all at once. Above is a schedule you can use to keep on track.

Testing

The main methods provided in the template files contain useful code to test your project as you work. You can use command like "**`java TarmartCustomer`**" to run the testing defined in **`main()`**. You could also edit **`main()`** to perform additional testing. JUnit test cases will not be provided for this project, but feel free to create JUnit tests for yourself. A part of your grade *will* be based on automatic grading using test cases made from the specifications provided.

When you're are done, you should be able to run either of the following commands to try out the simulation:

```
java Tarmart [numLines] [seed]
java TarmartGUI [seed]
```

The first will run 20 steps of a text-based simulation for a given number of lines and a given seed. For example, if we run:

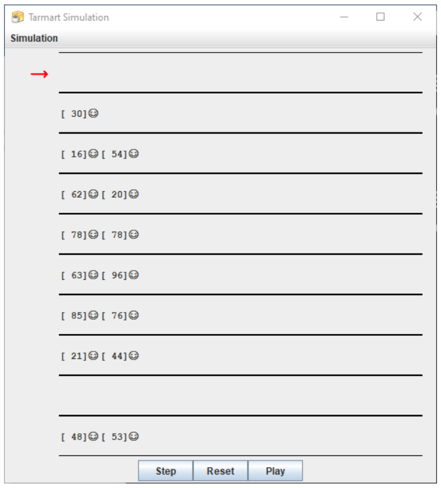
```
java Tarmart 4 0
```

We should get output similar to **`exampleRun.txt`** (included with this document).

If we run:

```
java TarmartGUI 0
```

We should see something like the image on the right:



```
→

[ 30]☺

[ 16]☺[ 54]☺

[ 62]☺[ 20]☺

[ 78]☺[ 78]☺

[ 63]☺[ 96]☺

[ 85]☺[ 76]☺

[ 21]☺[ 44]☺


[ 48]☺[ 53]☺
```

Appendix: CS211 Background

There are some basic Java concepts that we'll need to understand in order to do this project. It is assumed that you acquired this knowledge in the prerequisite classes (e.g. CS211), but this section outlines some material you may need a refresher on.

Inheritance and Generics

- Java Tutorial Inheritance: https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html
- Java Tutorial Generics: https://docs.oracle.com/javase/tutorial/java/generics/index.html

The majority of the classes in our program will be generic and will be using inheritance. Please make sure these are clear before starting this project.

Comparing Interface

- Java Reference: https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html
- Java Reference: https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html
- Helpful YouTube Video: https://www.youtube.com/watch?v=JSvVsOm4oX0

There are two Java interfaces which help explain to Java how objects can be compared. The **`Comparable`** interface helps define the "natural" order of custom data types (classes). An example of a "natural" ordering is with numbers. If I gave you a set of numbers and asked you to sort them, you would use their "natural" order to do so.

The **`Comparator`** interface helps define the order of custom data types that do not have a natural order to them. An example of something that does not have a natural order is a house. If I gave you a set of houses and asked you to sort them, you'd ask for more information (sort by size? price? location? paint color?).

If we make a class and want instances of that class to have the equivalent of **`<`**, **`==`**, and **`>`**, we use one of these two Java features to provide the needed information to the computer. A good video on the different ways to compare instances of a class (including the `Comparable` interface) is linked at the top of this section, but below is a text summary of what we need for this project:

1. Instances can't/shouldn't be compared with the <, >, or == operators.

If we have a class:

```
class Person {
      String name;
      int age;
}
```

And we make two instances:

```
Person p1 = new Person();
Person p2 = new Person();
```

We can't do the following:

```
if(p1 > p2) {
      //do something
}
```

Because Java doesn't know how to "compare" **`p1`** and **`p2`** (by **`name`**? by **`age`**? by memory location?).

2. If the instances have a "natural" order to them (or only ONE order for this particular project), then we should use the `Comparable` interface.

If people should always be compared by age in this program, then we can have the **`Person`** class implement the **`Comparable`** interface to tell Java how to compare people. The **`Comparable`** interface requires one single method: **`compareTo()`**.

```
class Person implements Comparable<Person> {
      String name;
      int age;
      public int compareTo(Person p) {
            return this.age - p.age;
      }
}
```

There are two things of note here. First, `Comparable` is generic, and we tell it what the classes can be compared to (e.g. people can be compared with people, so we implemented `Comparable<Person>`). Second, **`compareTo()`** returns an int. The convention is that `p1.compareTo(p2)` returns the following

If logically …	compareTo() return is…
`p1 > p2`	> 0
`p1 < p2`	< 0
`p1 == p2`	= 0

We still ***<u>can't do this</u>***:

```
if(p1 > p2) {
      //do something
}
```

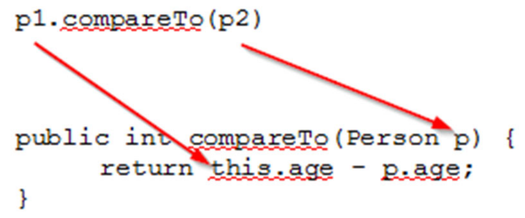
but now we can do this:

```
if(p1.compareTo(p2) > 0) {
      //do something
}
```

In our implementation of **`compareTo()`** above:

```
public int compareTo(Person p) {
      return this.age - p.age;
}
```

If `p1`'s age was 10 and `p2`'s age was 5, then `p1.compareTo(p2)` would return in 5. Note that `p1`'s age becomes `this.age` and `p2`'s age becomes `p.age` due to how we are calling this method:



```
p1.compareTo(p2)


public int compareTo(Person p) {
      return this.age - p.age;
}
```

3. Try this out before continuing with the project.

Make a simple class (like the `Person` class above), have it implement the `Comparable` interface (as shown above), and try calling that method (like above) in a main method. Play around with this to get a good feel for how this works before continuing.

Compiling JavaDocs

- JavaDoc Reference: http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html

The amazing Java API documents we've been reading throughout the semester were generated with a tool called `javadoc`. If we have written good JavaDocs, we can "compile" them into a webpage using the `javadoc` command from the terminal. For this project, if the path variable is set correctly for Java, we should be able to compile all the JavaDocs in the user folder by running the following command from the user directory:

```
javadoc -private -d docs *.java
```

If your system doesn't recognize the `javadoc` command, you may need to reconfigure your path (this is part of setting up Java, so you may Google how to do this if you need).

Once we've run the above command successfully, we will have a directory called "docs" in the user folder. In that folder is an index.html file we can open to see a whole website containing our Java documentation for this project! The `javadoc` compiler can also help us debug JavaDocs (it will give warnings when we forget to do certain things).

Random Numbers

- Java Reference: https://docs.oracle.com/javase/8/docs/api/java/util/Random.html
- Random Seeds: https://en.wikipedia.org/wiki/Random_seed

One of the command line arguments we'll use to run the simulation is a "seed" for a random number generator. Using this we can run the same simulation over and over again or make a new simulation each time. It would be helpful to have at least a vague understanding of pseudorandom numbers before working on this assignment.