



10/24/2016

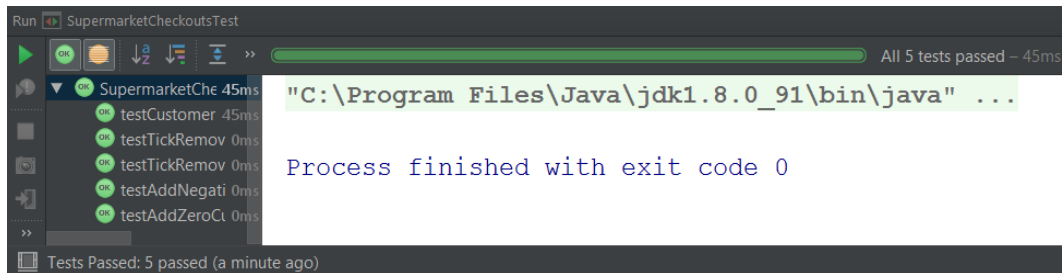
Project 2: Generics, Linked Lists, Stacks, and Apps

CS249

Instructor: Patrick Kelley

Stephen White

The first portion of this project required me to design a supermarket that had the ability to add customers, and simulate a minute of time through ticking, and removing customers when necessary. This would all be done through a queue-based data structure that I designed myself. The easiest way to go about this portion of the project was to split up the work of the SupermarketCheckouts class into two separate classes: Customer, and Clerk. The Customer class replicated what it meant to be a customer, meaning it only worried about itself, and how many items that particular customer had. The Clerk class needed to worry about organizing customers in a queue-like fashion. I decided to utilize an array of type Customer for this data structure, because each queue would have a defined maximum number of customers allowed in each line. All I had to do was keep track of the front and back of the line while adding and removing customers, and I was golden. To make removing a customer simpler, there were two methods that would be utilized in this process: `takeItemFromCustomer()`, and `removeCustomer()`. This way, I did not have to worry about calling `removeCustomer()` directly, as it would be dealt with in my `takeItemFromCustomer()` method. The concept was that if a customer ran out of items, the next time the clerk tried to take an item from them, the customer would leave the queue. Now that I had what it meant to have a single queue of customers defined, I figured an `ArrayList` of type Clerk would be best to have multiple queues. Depending of the specified number of desired lines, a for loop would handle the construction of that many clerks of the desired line length as well. This led to my SuperMarketCheckouts class `addCustomer()` method to be $O(N)$ complexity, as well as my `tick()` method to be $O(N)$ complexity as well, as I needed to go through every clerk one at a time to be able to take an item from them. Below is a screenshot of all unit tests passing:



After I was able to ensure that I could properly construct a supermarket, the next step was to create an application that allowed a user to construct a market based on whatever their specific desires were, and allow them to add customers, tick through their items, display the lines, and quit the program. Below are two screenshots of the running application:

```
Welcome to Stephen's Super Market Express! Press 'c' to continue, or 'q' to quit: c
```

```
ENTERING MARKET...
```

```
Alright boss, how many clerks are working today?: 3
```

```
Awesome! 3 is perfect!
```

```
Now remind me... what's the max number of people allowed in each line?: 3
```

```
Just double checking, you wanted your store to have 3 clerks, and a maximum line capacity of 3 people per line. Is that correct? ('y' for yes, 'n' for no):
```

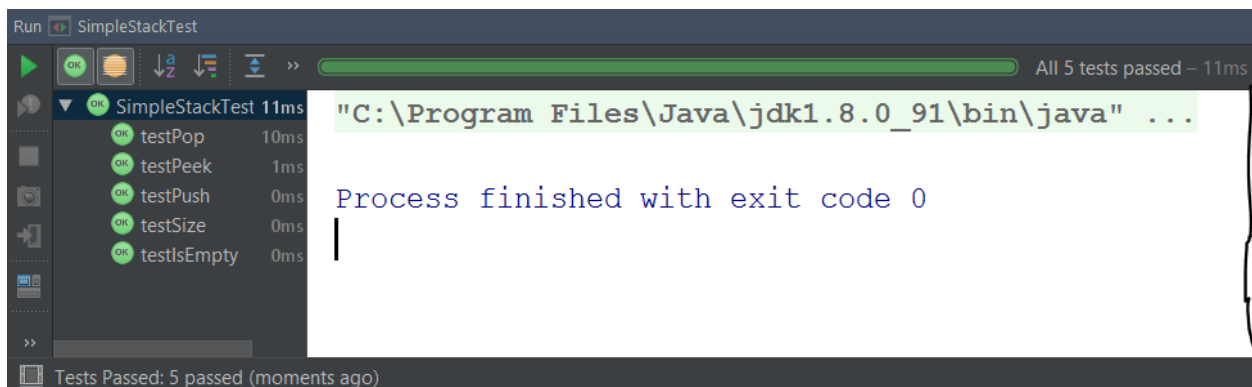
```
Type 'a' to add customers to the line, 't' to tick through the checkout, 'd' to display the lines, and 'q' to quit: a
Type 'a' to add customers to the line, 't' to tick through the checkout, 'd' to display the lines, and 'q' to quit: a
Type 'a' to add customers to the line, 't' to tick through the checkout, 'd' to display the lines, and 'q' to quit: a
Type 'a' to add customers to the line, 't' to tick through the checkout, 'd' to display the lines, and 'q' to quit: d
Here is your current Market:
```

```
Total number of lines in this market: 3
PEOPLE IN THIS LINE: A person with 1 items. NULL NULL
PEOPLE IN THIS LINE: A person with 1 items. NULL NULL
PEOPLE IN THIS LINE: A person with 4 items. NULL NULL
```

```
Type 'a' to add customers to the line, 't' to tick through the checkout, 'd' to display the lines, and 'q' to quit: t
Type 'a' to add customers to the line, 't' to tick through the checkout, 'd' to display the lines, and 'q' to quit: t
Type 'a' to add customers to the line, 't' to tick through the checkout, 'd' to display the lines, and 'q' to quit: d
Here is your current Market:
```

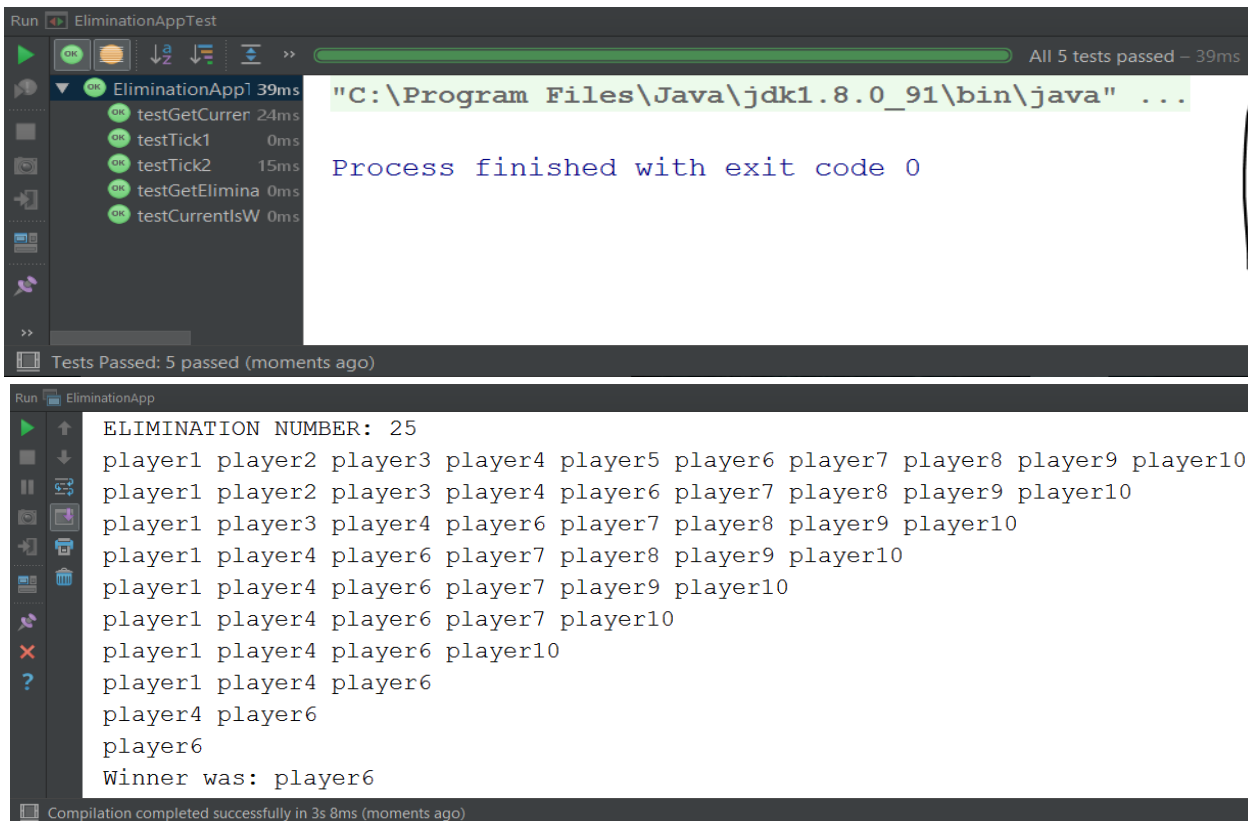
```
Total number of lines in this market: 3
PEOPLE IN THIS LINE: NULL NULL NULL
PEOPLE IN THIS LINE: NULL NULL NULL
PEOPLE IN THIS LINE: A person with 2 items. NULL NULL
```

The second portion of this project required me to design a generic stack from scratch. I would be allowed to utilize a generic array as my underlying data structure, however, I decided to create my own generic singly-linked list as the stack-based data structure. To do this, I first created a Link class, which essentially modeled a Node, in which any object could be attached to. From there, I created MyLinkedList, a class that linked objects together through my Link class in such a way, that it simulated stack-based architecture. This was done through designing addLink(), removeLink(), peek(), getNumLinks(), and isEmpty() methods. After implementing those methods, I created a SimpleStack class that implemented the Stack interface provided, and relied on the MyLinkedList class to create a stack. The stack could add and remove items with $O(1)$ complexity because it only cared about the item on the top of the stack. Pictures of all unit tests passing are provided below:



The last hurdle of this project was to create a data structure that could model a circular singly-linked list, and to develop an elimination application that would utilize the circular singly-linked list to simulate a game of duck-duck-goose without the chasing. To make this happen, I went about constructing my own Node class that took in an object and essentially wrapped it up in a “box” that could be placed in the list. This Node class had two public variables: `nextNode`, and `element`, so I would be able to know what type of object the node contained, as well as the next node in the sequence. The major chunk of this project came in the form of designing my `CircularLinkedList` class. Within this class I designed the logic for adding nodes, removing nodes at a specific “index” (number of nodes from node number one), obtaining the “current” node, obtaining the number of nodes in the list, determining if a winner was present based on the number of people in the list, and the ability to determine if the list was empty. The most important factors that shaped my design was the overwriting of `firstNode` and `lastNode` when either was removed, and keeping track of the “current” node. Adding a node could be done with $O(1)$ complexity as a node would only be added to the end of the list, but removing a node required N operations to find it, therefore making my remove method $O(N)$ complexity. Once my data structure was in place, I simply moved my methods over into the elimination app that modeled an elimination game by masking its methods for `tick()`, `init()`, `getCurrentPlayer()`, `currentIsWinner()`, and `getEliminationNumber()` with my own by creating a `CircularLinkedList` object which I named “circle.” Screenshots of all the unit tests passing as well as a photo

displaying an elimination game being simulated with an elimination number of 25 are shown below:



The image contains two screenshots of a Java IDE's run window. The top screenshot shows the results of running 'EliminationAppTest', with a list of five tests (testGetCurrer, testTick1, testTick2, testGetElimina, testCurrentIsW) all passing. The command prompt shows the Java command and the message 'Process finished with exit code 0'. The bottom screenshot shows the output of 'EliminationApp', which starts with 'ELIMINATION NUMBER: 25' and lists the remaining players in each round until only 'player6' remains as the winner.

```

Run EliminationAppTest
All 5 tests passed - 39ms
"C:\Program Files\Java\jdk1.8.0_91\bin\java" ...
Process finished with exit code 0
Tests Passed: 5 passed (moments ago)

Run EliminationApp
ELIMINATION NUMBER: 25
player1 player2 player3 player4 player5 player6 player7 player8 player9 player10
player1 player2 player3 player4 player6 player7 player8 player9 player10
player1 player3 player4 player6 player7 player8 player9 player10
player1 player4 player6 player7 player8 player9 player10
player1 player4 player6 player7 player9 player10
player1 player4 player6 player7 player10
player1 player4 player6 player10
player1 player4 player6
player4 player6
player6
Winner was: player6
Compilation completed successfully in 3s 8ms (moments ago)

```

This project was not nearly as difficult for me as project one, and as odd as it seems, my process for debugging has improved immensely over the course of these two projects. My planning process has also become quite developed as well. I sat down before writing any code whatsoever, wrote out a plan on paper, and fully developed my thoughts prior to jumping into anything. If I ran into any problems, I would correct them by writing out scenarios that I controlled and knew what the output should be, and this ultimately saved me so much time in the long run. This is not to say that I did not face adversity in the midst of my Circular Linked List class, as I spent a solid day trying to understand where some code had turned out volatile, only to discover little mistakes I had made throughout the development process affected the outcome of

my program as a whole. Other than some little bumps here and there, This project was a great refresher on how linked data structures are made, and how a stack and queue can be made in two completely different ways. I learned one huge lesson however. Chunking problems down into their smallest possible parts is essential in computer science. This became clear to me when I designed several separate classes for each portion of the project to allow me to make the most sense of the project and to split up the responsibility of different classes (i.e. a customer should not have to worry about a clerk, and a clerk should not have to worry about other clerks). I can proudly say I will carry the lessons I learned through this project with me as I progress through not only this course, but this major as a whole.