|  |
| --- |
| Option #1 |

**Program Corrections:**

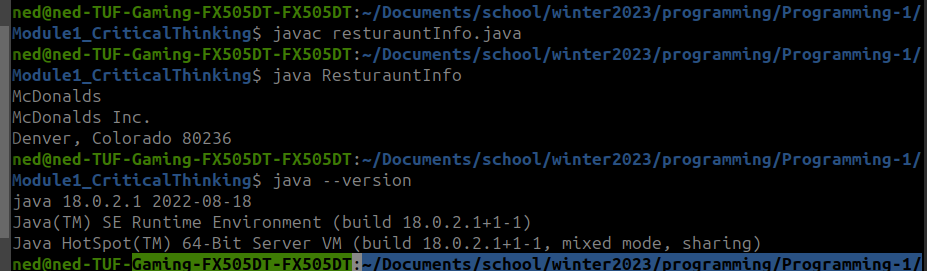
**Module 1: Critical Thinking: (Option 2)**

Change Summary:

* Corrected the spelling of ‘restaurant’ in variable names, no functional changes needed.

**Java Code: ResturantInfo.java**

*/\*\*  
 \* @author Alec Mcdaugale  
 \* December 17,2023  
 \*\*/*class RestaurantInfo {  
 public static void main(String[] args) {  
 final String restaurantName = "McDonalds";  
 final String businessName = "McDonalds Inc.";  
 final String city = "Denver";  
 final String state = "Colorado";  
 final int zipCode = 80236;   
 System.*out*.println(restaurantName);  
 System.*out*.println(businessName);  
 System.*out*.println(city + ", " + state + " " + zipCode);  
 System.*exit*(0);  
 }  
}

**Execution:**

**Github:**

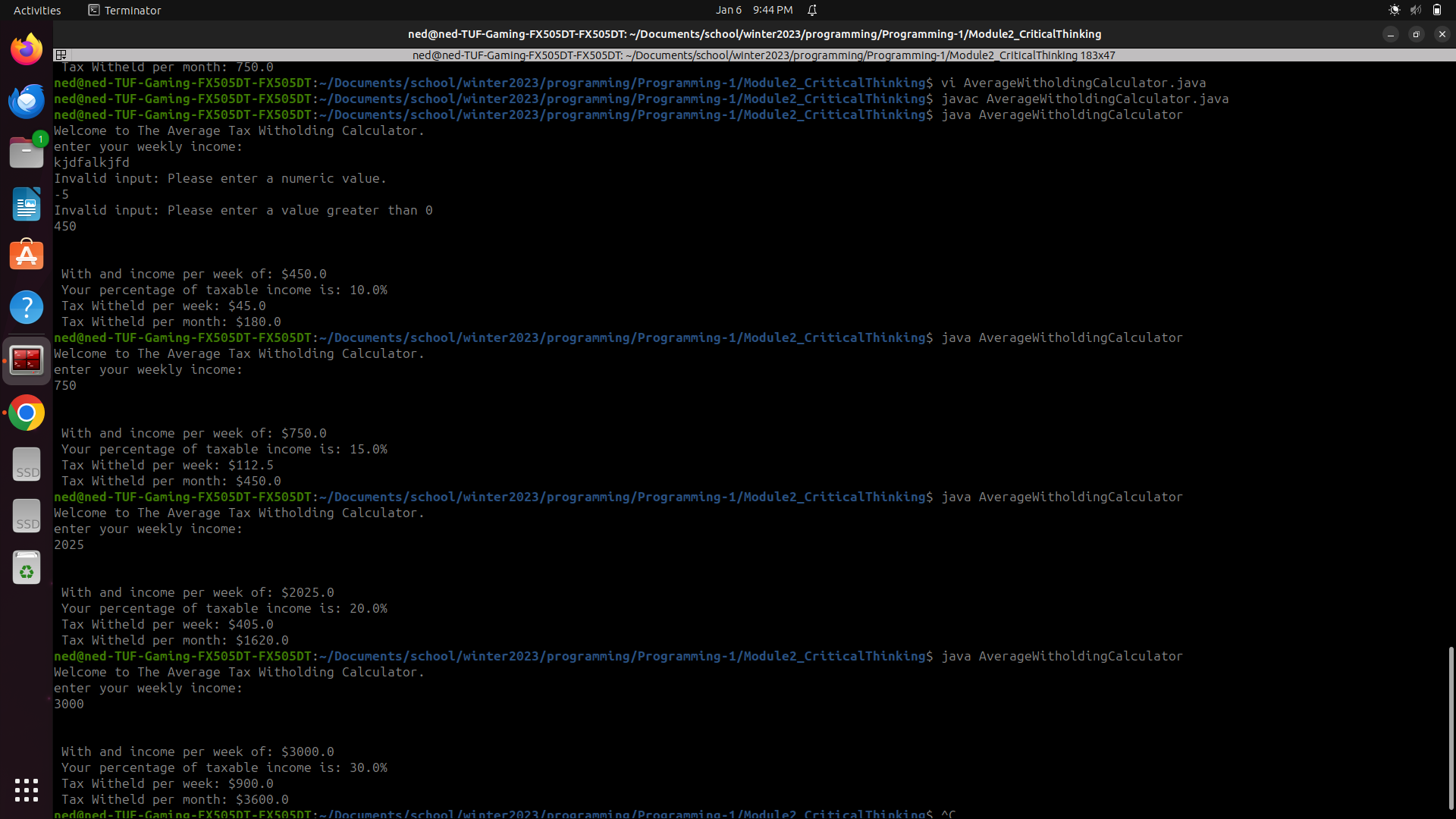
<https://github.com/neoHax05555/Programming-1/tree/main/Module1_CriticalThinking>

**Module 3: Critical Thinking:**

Change Summary:

* None!
  + I believe that the conditions were sufficiently explained in comments and output to user was clear that your tax rate was due to your given income.
  + Code executes properly and produces the correct output.

import java.util.Scanner;  
  
public class AverageWitholdingCalculator {  
 public static void main(String[] args) {  
 double income = 0;  
 double taxWitheld = 0;  
 double percentage = 0;  
 String incomeInput = "";  
 boolean incomeAccepted = false;  
 Scanner scanner = new Scanner(System.*in*);  
  
 System.*out*.println("Welcome to The Average Tax Witholding Calculator.");  
 System.*out*.println("enter your weekly income:");  
   
 // Take input and verify is numeric and is value greater than 0.  
 while (!incomeAccepted) {  
 incomeInput = scanner.next();  
 try {  
 income = Double.*parseDouble*(incomeInput);  
 if (income < 0) {  
 System.*out*.println("Invalid input: Please enter a value greater than 0");  
 } else {  
 incomeAccepted = true;  
 }  
 } catch (NumberFormatException e) {  
 System.*out*.println("Invalid input: Please enter a numeric value.");  
 }  
 }  
  
 if (income < 500) {  
 // Income less than $500: tax rate 10%  
 percentage = .10;  
 } else if (income >= 500 && income < 1500) {  
 // Incomes greater/equal to $500 and less than $1500: tax rate 15%  
 percentage = .15;  
 } else if (income >= 1500 && income < 2500) {  
 // Incomes greater/equal to $1500 and less than $2500: tax rate 20%  
 percentage = .20;  
 } else {  
 // Assumed Incomes greater than/equal to $2500: tax rate 30%  
 percentage = .30;  
 }  
  
 // Calculate witholdings.  
 taxWitheld = income \* percentage;  
  
 System.*out*.println("\n");  
 System.*out*.println(" With and income per week of: $" + income);  
 System.*out*.println(" Your percentage of taxable income is: " + (percentage \* 100) + "%");  
 System.*out*.println(" Tax Witheld per week: $" + taxWitheld);  
 System.*out*.println(" Tax Witheld per month: $" + (taxWitheld \* 4));  
 }  
}

**Github:**

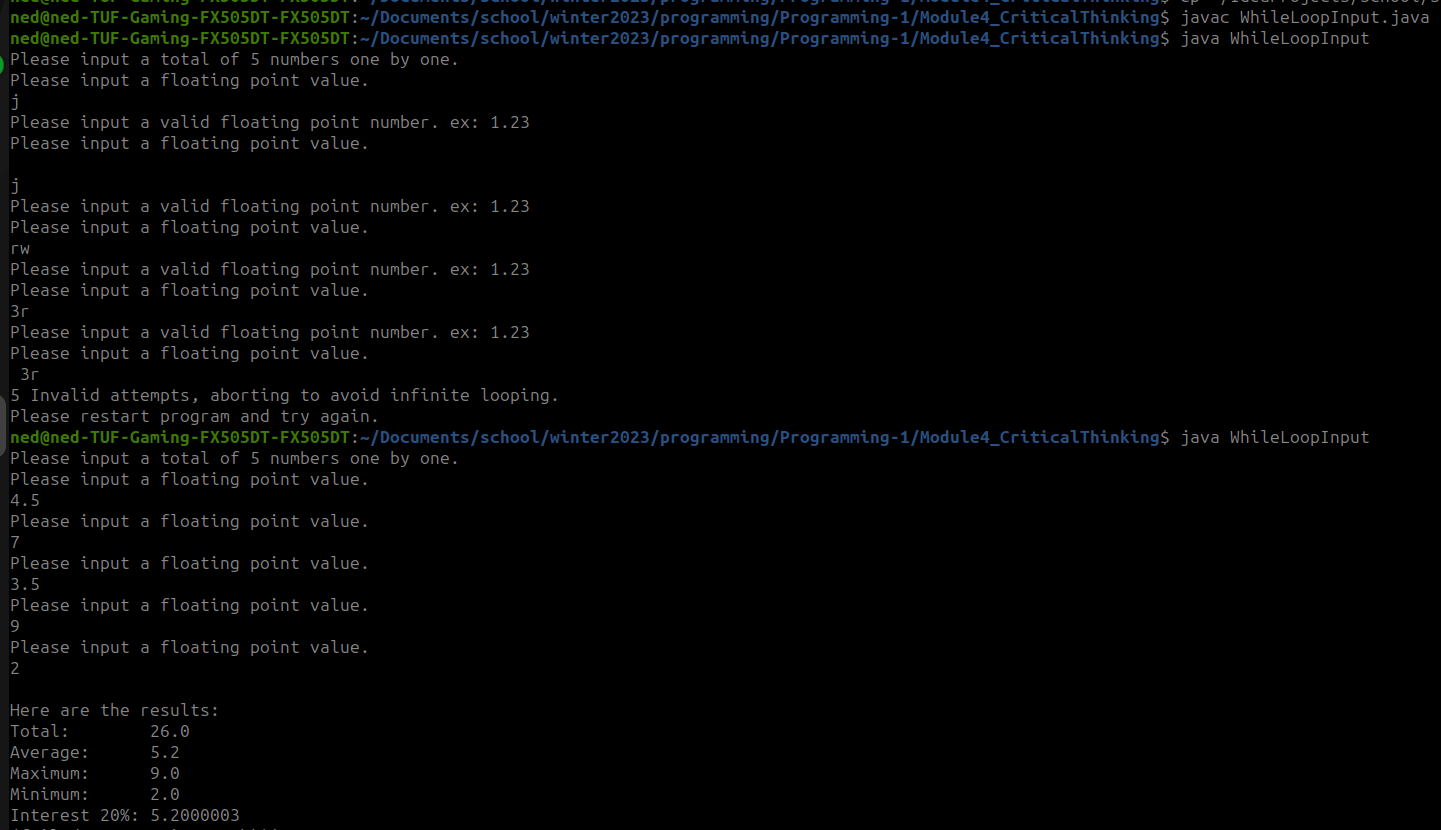
<https://github.com/neoHax05555/Programming-1/tree/main/Module3_CriticalThinking>

**Module 4: Critical Thinking:**

Change Summary:

* Added comment to explain how validation works in try catch block.
* No functional changes needed, program sanitizes input and outputs as expected.

import java.util.ArrayList;  
import java.util.Scanner;  
  
*/\*\*  
 \* @author Alec McDaugale  
 \* @Date: 01-13-24  
 \* @Course: CSC320-1  
 \* <br><br>  
 \* Option #1: Looping Construct with Floating Point Numbers<br><br>  
 \* Write a program that utilizes a while-loop to read a set of five floating-point values from user input. Include code  
 \* to prevent an endless loop. Ask the user to enter the values, then print the following data:  
 \* ------------------------------------------------------------------------------------------  
 \* <ul>  
 \* <li>Total</li>  
 \* <li>Average</li>  
 \* <li>Maximum</li>  
 \* <li>Minimum</li>  
 \* <li>Interest on total at 20%</li>  
 \* </ul>  
 \* ------------------------------------------------------------------------------------------  
 \* Compile and submit your pseudocode, source code, and screenshots of the application executing the application, the  
 \* results and GIT repository in a single document.  
 \*/*public class WhileLoopInput {  
 private static final float *INTEREST\_RATE* = 0.20f;  
  
 public static void main(String[] args) {  
 ArrayList<Float> inputValues = new ArrayList<>();  
 String rawInput;  
 boolean done = false;  
 boolean invalid = false;  
 int acceptedCount = 0;  
 int invalidCount = 0;  
 Scanner scanner = new Scanner(System.*in*);  
  
 System.*out*.println("Please input a total of 5 numbers one by one.");  
 do {  
 /\* Validate input is a float by caching NumberFormatException when  
 \* casting input to a float. If an exception is thrown then we know  
 \* that the input value was invalid. We allow for a maximum of 5  
 \* attempts before giving up and forcing the program to exit.  
 \*/  
 try {  
 System.*out*.println("Please input a floating point value.");  
 rawInput = scanner.next();  
 inputValues.add(Float.*parseFloat*(rawInput));  
 acceptedCount++;  
 invalidCount = 0;  
 if (5 == acceptedCount) {  
 done = true;  
 }  
 } catch (NumberFormatException e) {  
 invalidCount++;  
 if (5 == invalidCount) {  
 done = true;  
 invalid = true;  
 System.*out*.println("5 Invalid attempts, aborting to avoid infinite looping.");  
 } else {  
 System.*out*.println("Please input a valid floating point number. ex: 1.23");  
 }  
 }  
 } while (!done);  
  
 if (!invalid) {  
 float total = 0;  
 float average = 0;  
 // init max and min to first value to ensure initialized for first comparison.  
 float max = inputValues.get(0);  
 float min = inputValues.get(0);  
 float interest = 0;  
 int count = 0;  
  
 for (float value : inputValues) {  
 count++;  
 //Total  
 total += value;  
 //Maximum/Minimum  
 max = (value > max) ? value : max;  
 min = (value < min) ? value : min;  
 }  
  
 //Average  
 average = total / count;  
 //Interest on total at 20%  
 interest = total \* *INTEREST\_RATE*;  
  
 System.*out*.println();  
 System.*out*.println("Here are the results:");  
 System.*out*.println("Total: " + total);  
 System.*out*.println("Average: " + average);  
 System.*out*.println("Maximum: " + max);  
 System.*out*.println("Minimum: " + min);  
 System.*out*.println("Interest 20%: " + interest);  
 } else {  
 System.*out*.println("Please restart program and try again.");  
 }  
 }  
}



**Github:**

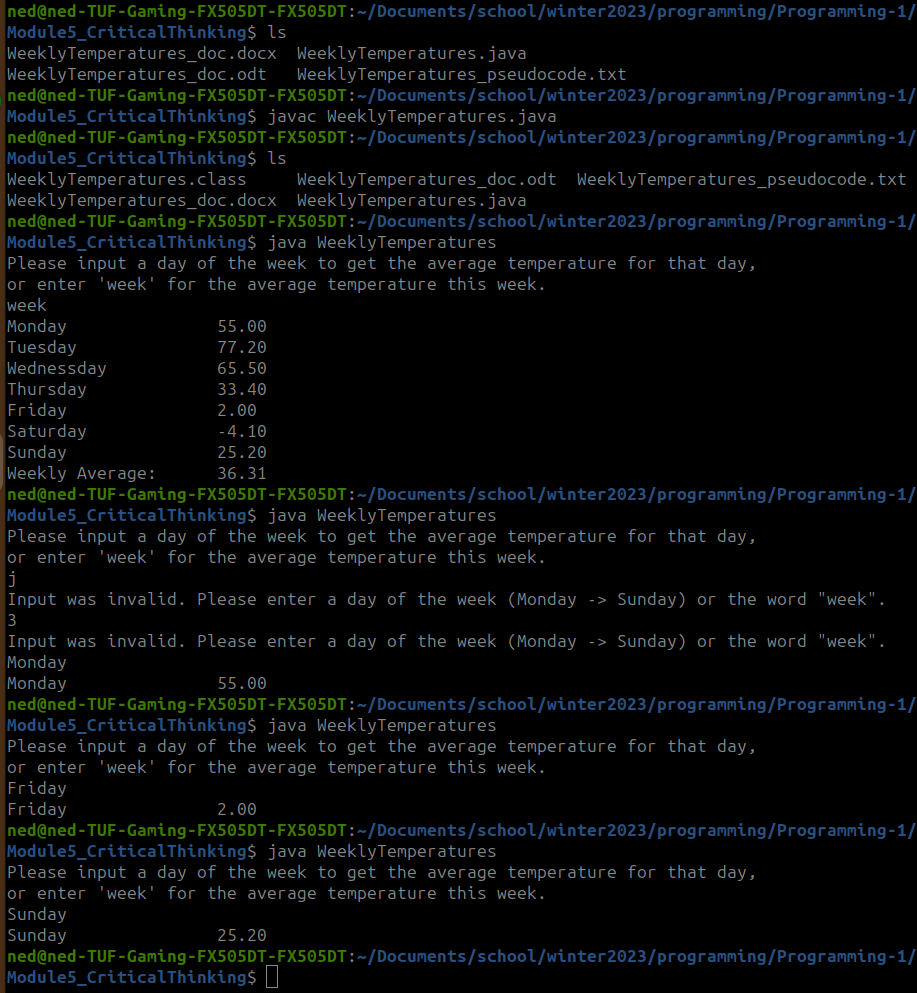
<https://github.com/neoHax05555/Programming-1/tree/main/Module4_CriticalThinking>

**Module 5: Critical Thinking:**

Change Summary:

* Corrected spelling of temperature. (I am a consistent misspeller. Lol)
* Enhanced formatting of output to all align.
  + Moved printing results to a new method to make this more potable.

import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.Scanner;  
  
*/\*\*  
 \* @author Alec McDaugale  
 \* @Date: 01-13-24  
 \* @Course: CSC320-1  
 \* <br><br>  
 \* Option #1: Get Weekly Temperatures<br><br>  
 \* Develop a Java program that will store data in the form of daily average temperatures for one week. Store the day and  
 \* average temperature in two different arraylists. Your program should prompt the user for the day of the week (Monday  
 \* through Sunday) and display both the day and temperature for each day. If "week" is entered, the output for your  
 \* program should provide the temperature for each day and the weekly average. Use the looping and decision constructs  
 \* in combination with the arrays to complete this assignment.  
 \*  
 \* ------------------------------------------------------------------------------------------  
 \* Compile and submit your pseudocode, source code, and screenshots of the application executing the application, the  
 \* results and GIT repository in a single document.  
 \*/*public class WeeklyTemperatures {  
 */\*\*  
 \* Initial state for weekdayIndex, represents an invalid value to use to determine if weekdayIndex was set.  
 \*/* private static final int *INVALID\_WEEKDAY* = -1;  
 */\*\*  
 \* Max weekday, above range of valid indexes to represent all values for the week.  
 \*/* private static final int *ALL\_WEEKDAY* = 7;  
  
 public static void main(String[] args) {  
 ArrayList<String> weekDay = new ArrayList<>(Arrays.*asList*("Monday", "Tuesday", "Wednessday",  
 "Thursday", "Friday", "Saturday", "Sunday"));  
 ArrayList<Double> temperature = new ArrayList<>(Arrays.*asList*(55.0, 77.2, 65.5, 33.4, 2.0, -4.1, 25.2));  
 boolean done = false;  
 // Default to -1 until set.  
 int weekdayIndex = *INVALID\_WEEKDAY*;  
 String rawInput;  
 Scanner scanner = new Scanner(System.*in*);  
  
 System.*out*.println("Please input a day of the week to get the average temperature for that day, " +  
 "\nor enter 'week' for the average temperature this week.");  
 // Accept user input.  
 do {  
 rawInput = scanner.next();  
 if ("week".equalsIgnoreCase(rawInput)) {  
 weekdayIndex = *ALL\_WEEKDAY*;  
 } else {  
 for (String day : weekDay) {  
 if (day.equalsIgnoreCase(rawInput)) {  
 // If found get the index of the given day.  
 weekdayIndex = weekDay.indexOf(day);  
 }  
 }  
 }  
  
 if (*INVALID\_WEEKDAY* == weekdayIndex) {  
 System.*out*.println("Input was invalid. Please enter a day of the week (Monday -> Sunday) or the word \"week\".");  
 }  
 } while (*INVALID\_WEEKDAY* == weekdayIndex);  
  
 // Print results.  
 if (*ALL\_WEEKDAY* == weekdayIndex) {  
 // If 'week' was entered, we need to print averages.  
 double average = 0;  
 double weeklySum = 0;  
 double temp;  
  
 // Sum temps for each day.  
 for (String day : weekDay) {  
 temp = temperature.get(weekDay.indexOf(day));  
 weeklySum += temp;  
 *printTwoColumnOutput*(day, temp);  
 }  
 // Calculate average.  
 average = weeklySum/*ALL\_WEEKDAY*;  
 *printTwoColumnOutput*("Weekly Average: ", average);  
 } else {  
 // Else a weekday was entered, so print that data out for that index.  
 *printTwoColumnOutput*(weekDay.get(weekdayIndex), temperature.get(weekdayIndex));  
 }  
 }  
  
 */\*\*  
 \* Prints output formatted in 2 columns for consistent output to end user.  
 \*  
 \* @param column1 String value, usually a label for column 2.  
 \* @param column2 Float value to represent a dollar amount.  
 \*/* private static void printTwoColumnOutput(String column1, double column2) {  
 System.*out*.printf("%-20s %2.2f\n", column1, column2);  
 }  
}



**Github:**

[https://github.com/neoHax05555/Programming-1/tree/main/Module5\_CriticalThinking](https://github.com/neoHax05555/Programming-1/tree/main/Module4_CriticalThinking)

**Lessons Learned:**

Throughout this Programming 1 course we have learned essentially the same things that I learned in Computer Science 1 before transferring credits from my Associates program through the community college system. While I found this to be a bit of a drag, sometimes it is nice and even advantageous to go back an revisit the basics. In this course we covered everything from variables and data types, decision control structures,repetitive control structures, arrays and ArrayLists, methods and exceptions, memory management, and finally classes and object oriented concepts.

In java we have eight primitive data types. These are int, byte, short, long, float, double, boolean, and char. The importance of knowing these types is that they are not treated as Objects in Java as every other data type is treated as an Object. One fact that is interesting is that String is an Object that we typically think of as a primitive. To truly have a primitive type to represent a String we would need to use a character array, though outside of C I have not seen this very often. Another important thing to consider is that once a variable is no longer being used or is de-referenced in Java, the garbage collector will come and clean up the memory consumed by that variable on either the program stack or heap. (This location is dependent on the JVM implementation you go with.)

Control structures are the most fundamental building block to make logical decisions in any programming language. (Branch statements in Assembly are by far my favorite control structure as they can be used to create any control structure.) Java has 3 basic selection structures. These are the if/else, switch, and ternary statements. Java also has four main types of looping structures. These are the for, cor each, do while and while loop. All control structures are controlled by a boolean value. These are usually the result of a logical expression that evaluates to a boolean value. I believe that this is an important thing to keep in mind as the result of an expression can either be evaluated within the structure, or stored in a boolean to use elsewhere. This creates more flexibility for the developer using these values.

Arrays are essentially a means for someone to store multiple values of the same type to the same variable by indexing them. This in turn simplifies our code quite a bit since we do not need to declare multiple variables to store values for some datatype, and instead can define them all in the same variable. The only downside to using an array is we have to decare their size upon initialization, which makes this size impossible to increase without declaring a new array. Java has created the ArrayList type to solve this delima. This Object provides us with a dynamic array that can increase an decrease in size as we decide to add more values to it.

Java IO is one of the most important topics covered in this course. I mention this because I regularly use byte streams on a daily basis in Java to transfer data across networks, between devices, across processes, and even to write files. It is good that we covered the try/catch/finally statement though I see most people use the try with resources statement as long as the stream supports AutoCloseable, if not then the finally block is indeed needed. Memory management is also one of those things that bite people later on in development. We use a library called LeakCanary which when implemented will report when it detects a memory leak within your code to prevent this.

Exceptions and handling them are another essential topic covered in this course. When a method throws an exception, if unhandled, your process will crash. We had this issue with a front end UI application that we developed where a developer was not catching the null pointer exceptions. This is when I was introduced to the ‘Fail Fast’ paradigm. The purpose here was not to catch your exceptions so that the issues like this would be discovered as soon as possible, and forces the developer to add a null check (or avoid using null) where it failed rather than catching the exception and having the underlying issue going unnoticed. However, this issue went undiscovered for a long time and only occurred 1 in every 1000 executions, so went undiscovered during development and was discovered by an end user instead which is not typically who you want to discover the issues.

Classes and methods are the final topic covered in this course, as they were in Computer Science 1. These are what allow us to make our code modular, and do lots of cool things with designs using various patterns. The Object Oriented design pattern is typically the one I have learned about in past courses where we break down concepts into various objects that can make up larger objects. This is where all of the fun begins.

Though it is bitter sweet, I am glad to be done with this course and moving on to the next course Programing2. For me it is the instructor and all of my peers that I will miss. Even though we were all online for only two months, it was a great group of individuals.

**Final Program**

**Source Code:**

**Main**

package com.alec.portfolio;  
  
import java.io.File;  
import java.io.FileOutputStream;  
import java.io.IOException;  
import java.util.Scanner;  
  
public class Main {  
 private static final String *DIR\_NAME* = "tmp";  
 private static final String *FILE\_NAME* = "VehicleInventory.txt";  
  
 public static void main(String[] args) {  
 Vehicle firstCar = null;  
 VehicleInventory vehicleInventory;  
 boolean success = false;  
  
 // #1 & #2  
 /\* Initialize a car using builder method. The builder method calls a parameterized constructor  
 \* private to the Vehicle class.  
 \*/  
 try {  
 firstCar = new Vehicle.VehicleBuilder("Chevrolet", "Impala", 2017)  
 .setColor("Green")  
 .setMileage(135000)  
 .build();  
 } catch (IllegalArgumentException e) {  
 System.*out*.println("Invalid vehicle format.\n" + e);  
 }  
 /\* Check if the first car was created successfully and init inventory with the first car,  
 \* Else if unsuccessful, init with empty inventory.  
 \*/  
 if (null != firstCar) {  
 vehicleInventory = new VehicleInventory(firstCar);  
 } else {  
 vehicleInventory = new VehicleInventory();  
 }  
 // Print what we have in array to user.  
 *printStringArray*(vehicleInventory.listVehicleInfo());  
  
 // #3  
 // Remove vehicle we created @ index 0  
 success = vehicleInventory.removeVehicle(0);  
 // Print result.  
 System.*out*.println("Remove Vehicle: " + (success ? "Successful" : "Failed"));  
  
 // #4  
 success = vehicleInventory.addVehicle("Ram", "2500", "Silver", 2017, 99000);  
 // Print result.  
 System.*out*.println("Add Vehicle: " + (success ? "Successful" : "Failed"));  
 // Print what we have in array to user.  
 *printStringArray*(vehicleInventory.listVehicleInfo());  
  
 // #5  
 success = vehicleInventory.updateVehicle(0, "Ram", "2500", "Red", 2018, 199000);  
 // Print result.  
 System.*out*.println("Update Vehicle: " + (success ? "Successful" : "Failed"));  
 // Print what we have in array to user.  
 *printStringArray*(vehicleInventory.listVehicleInfo());  
  
 // #6  
 System.*out*.println("Would you like to print the current inventory to a file? [y/n]");  
 Scanner scanner = new Scanner(System.*in*);  
 String response = scanner.nextLine();  
 int invalidCounter = 5;  
 success = false; // Set success to false to init for loop.  
  
 do {  
 switch (response) {  
 case "y":  
 case "Y":  
 *printToFile*(vehicleInventory.listVehicleInfo());  
 success = true;  
 break;  
 case "n":  
 case "N":  
 System.*out*.println("Not printing vehicleInfo to file.");  
 success = true;  
 break;  
 default:  
 if (invalidCounter > 0) {  
 System.*out*.println("Invalid input, plese respond with either 'y' or 'n'.");  
 invalidCounter--;  
 } else {  
 System.*out*.println("Invalid input, max attempts reached, forcing exit.");  
 success = true;  
 }  
 }  
 } while (!success);  
  
 // Exit once complete.  
 System.*exit*(0);  
 }  
  
 */\*\*  
 \* Prints an element of a String array on a line.  
 \*  
 \* @param strings String array to print.  
 \*/* private static void printStringArray(String[] strings) {  
 for (String string : strings) {  
 System.*out*.println(string);  
 }  
 }  
  
 */\*\*  
 \* System independent method to write all Vehicles in inventory to a file. In the  
 \* current working directory we will create a directory and file called  
 \* tmp/VehicleInventory.txt  
 \*  
 \* @param vehicleInventory A string format of vehicle inventory to print to file.  
 \*  
 \* @return true - if write was successful.  
 \* false - if failed to write to file.  
 \*/* private static boolean printToFile(String[] vehicleInventory) {  
 /\*  
 \* Get cwd and append the dirname, should be potable across systems,  
 \* i`m on linux you may be on windows based on location you posted.  
 \* (no C:\ drive in linux)  
 \*/  
 File fileLocation = new File(new File(".").getAbsolutePath(), *DIR\_NAME*);  
 File file = new File(fileLocation, *FILE\_NAME*);  
 boolean success = false;  
  
 // Create dir if DNE.  
 if (fileLocation.mkdir()) {  
 if (!fileLocation.exists()) {  
 // Already init to false, so a failure to create dir will return false.  
 return success;  
 }  
 }  
  
 try (FileOutputStream fos = new FileOutputStream(file)) {  
 for (String car : vehicleInventory) {  
 // Append newline to end of each string, else all on one line.  
 fos.write((car + "\n").getBytes());  
 fos.flush();  
 }  
 success = true;  
 } catch (IOException e) {  
 System.*out*.println("Failed to write output.\n" + e);  
 }  
  
 return success;  
 }  
}

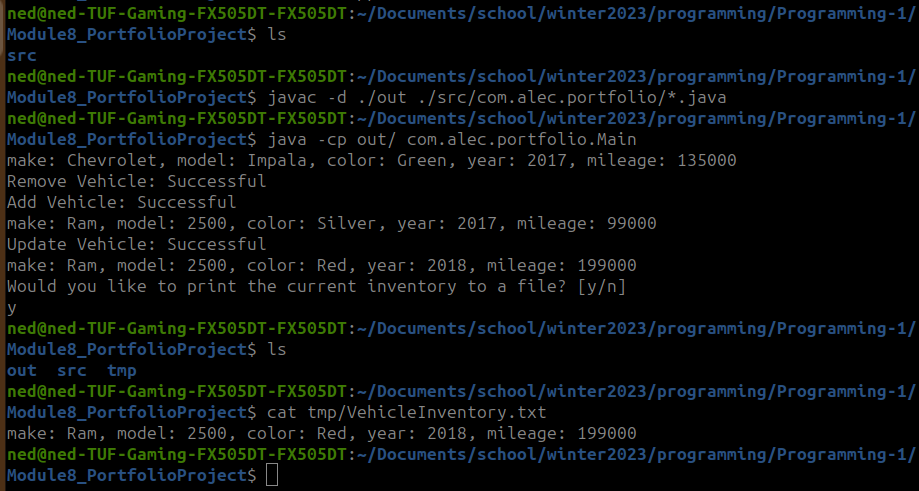
VehicleInventory

package com.alec.portfolio;  
  
import java.util.ArrayList;  
  
*/\*\*  
 \* Vehicle inventory class, vehicles in inventory are stored in an ArrayList called garage.  
 \*/*public class VehicleInventory {  
 private final ArrayList<Vehicle> garage;  
  
 */\*\*  
 \* Default constructor.  
 \*/* VehicleInventory() {  
 this.garage = new ArrayList<>();  
 }  
  
 */\*\*  
 \* Constructor to accept a Vehicle to init in garage.  
 \*  
 \* @param vehicle The vehicle to init in new garage.  
 \*/* VehicleInventory(Vehicle vehicle) {  
 this.garage = new ArrayList<>();  
 garage.add(vehicle);  
 }  
  
 */\*\*  
 \* Constructor to init garage with an arraylist of vehicles.  
 \*  
 \* @param vehicles An arraylist of vehicles to add to garage.  
 \*/* VehicleInventory(ArrayList<Vehicle> vehicles) {  
 this.garage = vehicles;  
 }  
  
 */\*\*  
 \* Add a new vehicle to the garage.  
 \*  
 \* @param make - The company who produced this car.  
 \* @param model - The name to identify the vehicle the company produced.  
 \* @param color Non null String to represent a color.  
 \* @param year - The year the vehicle was manufactured, must be after 1886.  
 \* @param mileage If invalid -1, else a non-negative value representing the vehicles' mileage.  
 \*  
 \* @return true - if vehicle successfully added.  
 \* false - if we failed to add the vehicle to garage.  
 \*/* public boolean addVehicle(String make, String model, String color, int year, int mileage) {  
 boolean success = false;  
 Vehicle.VehicleBuilder vehicle;  
 try {  
 vehicle = new Vehicle.VehicleBuilder(make, model, year);  
 if (null != color && !color.isEmpty()) {  
 vehicle.setColor(color);  
 }  
 vehicle.setMileage(mileage);  
 // Success always false unless this line returns true.  
 success = this.addVehicle(vehicle.build());  
 } catch (IllegalArgumentException e) {  
 System.*out*.println("Failed to add vehicle.\n" + e);  
 }  
  
 return success;  
 }  
  
 */\*\*  
 \* Add a new vehicle to the garage.  
 \*  
 \* @param vehicle The Vehicle object to add.  
 \*  
 \* @return Always true.  
 \*/* public boolean addVehicle(Vehicle vehicle) {  
 //Always true  
 return garage.add(vehicle);  
 }  
  
 */\*\*  
 \* Remove a vehicle at a given index from the garage.  
 \*  
 \* @param index Index of vehicle to remove.  
 \*  
 \* @return true - if vehicle successfully removed.  
 \* false - if we failed to remove the vehicle from garage.  
 \*/* public boolean removeVehicle(int index) {  
 boolean success = false;  
 try {  
 success = removeVehicle(garage.get(index));  
 } catch (IndexOutOfBoundsException e) {  
 System.*out*.println("Index was out of bounds.\n" + e);  
 }  
 return success;  
 }  
  
 */\*\*  
 \* Remove a vehicle with a parameterized approach.  
 \*  
 \* @param make - The company who produced this car.  
 \* @param model - The name to identify the vehicle the company produced.  
 \* @param color Non null String to represent a color.  
 \* @param year - The year the vehicle was manufactured, must be after 1886.  
 \* @param mileage If invalid -1, else a non-negative value representing the vehicles' mileage.  
 \*  
 \* @return true - if vehicle successfully removed.  
 \* false - if we failed to remove the vehicle from garage.  
 \*/* public boolean removeVehicle(String make, String model, String color, int year, int mileage) {  
 boolean success = false;  
 Vehicle.VehicleBuilder vehicleBuilder;  
  
 try {  
 vehicleBuilder = new Vehicle.VehicleBuilder(make, model, year);  
 if (null != color && !color.isEmpty()) {  
 vehicleBuilder.setColor(color);  
 }  
 vehicleBuilder.setMileage(mileage);  
 // Success always false unless this line returns true.  
 success = this.removeVehicle(vehicleBuilder.build());  
 } catch (IllegalArgumentException e) {  
 System.*out*.println("Failed to generate vehicle to remove.\n" + e);  
 }  
 return success;  
 }  
  
 */\*\*  
 \* Remove a vehicle form the garage based on a Vehicle Object.  
 \*  
 \* @param vehicle - Vehicle to remove.  
 \*  
 \* @return true - if vehicle successfully removed.  
 \* false - if we failed to remove the vehicle from garage.  
 \*/* public boolean removeVehicle(Vehicle vehicle) {  
 // If the object exists, will be true, else false.  
 return garage.remove(vehicle);  
 }  
  
 */\*\*  
 \* Replaces the vehicle in garage ArrayList at given index with the Vehicle  
 \* Object generated from parameters passed in.  
 \*  
 \* @param index Index of vehicle to replace.  
 \* @param make - The company who produced this car.  
 \* @param model - The name to identify the vehicle the company produced.  
 \* @param color Non null String to represent a color.  
 \* @param year - The year the vehicle was manufactured, must be after 1886.  
 \* @param mileage If invalid -1, else a non-negative value representing the vehicles' mileage.  
 \*  
 \* @return true - if vehicle successfully updated.  
 \* false - if we failed to update the vehicle in the garage.  
 \*/* public boolean updateVehicle(int index, String make, String model, String color, int year, int mileage) {  
 boolean success = false;  
 Vehicle.VehicleBuilder vehicle;  
  
 try {  
 vehicle = new Vehicle.VehicleBuilder(make, model, year);  
 if (null != color && !color.isEmpty()) {  
 vehicle.setColor(color);  
 }  
 vehicle.setMileage(mileage);  
 // Success always false unless this line returns true.  
 success = this.updateVehicle(index, vehicle.build());  
 } catch (IllegalArgumentException e) {  
 System.*out*.println("Failed to add vehicle.\n" + e);  
 }  
  
 return success;  
 }  
  
 */\*\*  
 \* Replaces the vehicle in garage ArrayList at given index with the Vehicle  
 \* Object passed in.  
 \*  
 \* @param index Index of vehicle to replace.  
 \* @param vehicle Vehicle object to replace it with.  
 \*  
 \* @return true - if vehicle successfully updated.  
 \* false - if we failed to update the vehicle in the garage.  
  
 \*/* public boolean updateVehicle(int index, Vehicle vehicle) {  
 boolean success = false;  
 Vehicle oldVehicle;  
  
 try {  
 oldVehicle = garage.set(index, vehicle);  
 // If we get to this point, and the element at this index was not  
 // previously the same, then it was replaced.  
 success = (!vehicle.equals(oldVehicle));  
 } catch (ArrayIndexOutOfBoundsException e) {  
 System.*out*.println("Index out of bounds.\n" + e);  
 }  
  
 return success;  
 }  
  
 */\*\*  
 \* Converts the arraylist of vehicles in garage into an array of all  
 \* vehicles in toString format.  
 \*  
 \* @return an array of all vehicles from garage in toString format.  
 \*/* public String[] listVehicleInfo() {  
 String[] vehicles = new String[garage.size()];  
  
 for (Vehicle vehicle : garage) {  
 vehicles[garage.indexOf(vehicle)] = vehicle.toString();  
 }  
  
 return vehicles;  
 }  
}

**Vehicle class**

package com.alec.portfolio;  
  
*/\*\*  
 \* Class to represent a vehicle in inventory, use provided builder class to initialize.  
 \*/*public class Vehicle {  
 public static final int *INVALID\_INT\_VALUE* = -1;  
 private final String make;  
 private final String model;  
 private final String color;  
 private final int year;  
 private final int mileage;  
  
 */\*\*  
 \* Default constructor, made private to avoid use outside of this class.  
 \*/* private Vehicle() {  
 this("", "", "", *INVALID\_INT\_VALUE*, *INVALID\_INT\_VALUE*);  
 }  
  
 */\*\*  
 \* Parameterized constructor, used by builder class. Made private to avoid use outside of this class.  
 \*  
 \* @param make - The company who produced this car.  
 \* @param model - The name to identify the vehicle the company produced.  
 \* @param color Non null String to represent a color.  
 \* @param year - The year the vehicle was manufactured, must be after 1886.  
 \* @param mileage If invalid -1, else a non-negative value representing the vehicles' mileage.  
 \*/* private Vehicle(String make, String model, String color, int year, int mileage) {  
 this.make = make;  
 this.model = model;  
 this.color = color;  
 this.year = year;  
 this.mileage = mileage;  
 }  
  
 */\*\*  
 \* Version to accept builder type. Made private to avoid use outside of this class.  
 \* @param vehicleBuilder The instance of the builder class to init this object.  
 \*/* private Vehicle(VehicleBuilder vehicleBuilder) {  
 this(vehicleBuilder.MAKE, vehicleBuilder.MODEL, vehicleBuilder.color, vehicleBuilder.YEAR, vehicleBuilder.mileage);  
 }  
  
 public String getMake() {  
 return make;  
 }  
  
 public String getModel() {  
 return model;  
 }  
  
 public String getColor() {  
 return color;  
 }  
  
 public int getYear() {  
 return year;  
 }  
  
 public int getMileage() {  
 return mileage;  
 }  
  
 @Override  
 public String toString() {  
 return "make: " + this.getMake() + ", model: " + this.getModel() + ", color: "  
 + ((this.getColor().isEmpty()) ? "N/A" : this.getColor())  
 + ", year: " + this.getYear() + ", mileage: "  
 + (((*INVALID\_INT\_VALUE* == this.getMileage()) ? "N/A" : this.getMileage()));  
 }  
  
 @Override  
 public boolean equals(Object obj) {  
 boolean isEqual = false;  
 Vehicle car;  
  
 try {  
 // Check if object is an instance of Vehicle.  
 if (obj instanceof Vehicle) {  
 car = (Vehicle) obj;  
 // If all values are equivalent, then is equal.  
 isEqual = (this.getMake().equals(car.getMake())  
 && this.getModel().equals(car.getModel())  
 && this.getColor().equals(car.getColor())  
 && this.getYear() == car.getYear()  
 && this.getMileage() == car.getMileage());  
 }  
 } catch (ClassCastException e) {  
 System.*out*.println("Failed to cast, not equal.");  
 }  
  
 return isEqual;  
 }  
  
 */\*\*  
 \* I decided to use a builder pattern to init my car Object, and make all constructors private  
 \* to the Vehicle class to prevent invoking those outside of this builder class.  
 \*/* public static class VehicleBuilder {  
 /\* Constants. \*/  
 private static final int *YEAR\_OF\_FIRST\_CAR* = 1886;  
  
 /\* Required values, only aloud to set once since these should never change. \*/  
 private final String MAKE;  
 private final String MODEL;  
 private final int YEAR;  
  
 /\* Optional values. \*/  
 private String color;  
 private int mileage;  
  
 */\*\*  
 \* Minimally required attributes.  
 \*  
 \* @param make - The company who produced this car.  
 \* @param model - The name to identify the vehicle the company produced.  
 \* @param year - The year the vehicle was manufactured, must be after 1886.  
 \*  
 \* @throws IllegalArgumentException - In the event an illegal value was passed in.  
 \*/* public VehicleBuilder(String make, String model, int year) throws IllegalArgumentException {  
 if (null == make || null == model) {  
 throw new IllegalArgumentException("Null values not allowed in constructor.");  
 } else if (make.isEmpty() || model.isEmpty()) {  
 throw new IllegalArgumentException("empty values not allowed in constructor.");  
 }else if (year < *YEAR\_OF\_FIRST\_CAR*) {  
 throw new IllegalArgumentException("Cars did not exist before: " + *YEAR\_OF\_FIRST\_CAR*);  
 }  
  
 // Set Required values from parameters.  
 this.MAKE = make;  
 this.MODEL = model;  
 this.YEAR = year;  
  
 // initialize all non-required values.  
 this.color = "";  
 this.mileage = *INVALID\_INT\_VALUE*;  
 }  
  
 */\*\*  
 \* Sets the color of the car.  
 \*  
 \* @param color Non null String to represent a color.  
 \*  
 \* @return This object in current state to comply to builder pattern.  
 \* @throws IllegalArgumentException Value of color was null.  
 \*/* public VehicleBuilder setColor(String color) throws IllegalArgumentException {  
 if (null == color) {  
 throw new IllegalArgumentException("Null values not allowed for color.");  
 }  
 this.color = color;  
 return this;  
 }  
  
 */\*\*  
 \*  
 \* @param mileage If invalid -1, else a non-negative value representing the vehicles' mileage.  
 \*  
 \* @return This object in current state to comply to builder pattern.  
 \* @throws IllegalArgumentException Value of mileage was not -1 or non-negative value.  
 \*/* public VehicleBuilder setMileage(int mileage) throws IllegalArgumentException {  
 if (mileage < 0 && mileage != *INVALID\_INT\_VALUE*) {  
 throw new IllegalArgumentException("Mileage cannot be less than 0.");  
 }  
 this.mileage = mileage;  
 return this;  
 }  
  
 */\*\*  
 \* Generates a Vehicle object from this builder object.  
 \*  
 \* @return Vehicle with values form this class.  
 \*/* public Vehicle build() {  
 return new Vehicle(this);  
 }  
 }  
}

**Screenshot of Execution**



**Github:**

<https://github.com/neoHax05555/Programming-1/tree/main/Module8_PortfolioProject>