Title : **Car Data Analysis**
Modules : **05 Advanced Control Flow & 06 Arrays & File I/O**
Point Value : **50**
Deadline : **Due before 11:00 pm on Thursday 20th October**

Topics Covered:

Nested decision and repetition statements, one dimensional arrays, file input and output, more writing and calling subroutines. [You absolutely may not use ArrayLists or other Java Collection classes in this assignment.]

Project Description:

In this project, you'll explore loading a real-world dataset containing information about cars available for auction in the US, namely, the US Cars Dataset available from Kaggle.com. Don't worry about downloading the dataset; we will include it with the starter code for this project.

The data is stored in a common file format called CSV (comma-separated values.) These files can be used to represent a dataset arranged in rows and columns. Each column represents a feature of the data we want to store; in our car dataset, these include a car's *brand*, *year*, *mileage* and *price*. Each row represents a car entry in the dataset and is stored in plain text, each column in the entry being separated by commas (hence the name of the file type.)

The overall goal of this project will be to parse the file and load the data we care about into arrays, each of which will represent a whole column of data. Your program will then present a menu allowing the user to perform several data-analysis options on the loaded data.

Program Operation:

The overall behavior of your program will be to present the user with a menu-driven interface that allows the user to repeatedly select a task, until they decide to quit. Before the menu action begins, your program will prompt the user for the name of a car data file to load. This way, the program can be used to analyze other car data files that follow the same format. This includes, for example, if you'd like to make a smaller version of the dataset with only a few rows for testing purposes.

Once the car data has been loaded, your program will present the user with a menu of options. Each menu item corresponds to a different action:

1. **Average price of brand:** The program will prompt the user for a car brand and an output file name. It will then output the number of entries in the loaded data whose brand matches the one given and their average price. The program will also save all matching rows to a CSV file named according to the input given where the columns are *each match's row index in the original dataset*, *brand*, *year*, *mileage*, and *price*. You do not need to include the line of headers (column titles.) This ford.csv file shows the output file that should be generated using "Ford" as the query for this option when loading USA_cars_datasets.csv file at the start of the program. [EDIT 10/19] You should open the csv file with a plain text editor (like WordPad or TextEdit) in order to see the expected output format. Here is a snippet of the first five lines:

```
1, ford, 2011, 190552.0, 2899
3, ford, 2014, 64146.0, 25000
9, ford, 2017, 63418.0, 5250
14, ford, 2013, 100757.0, 20700
18, ford, 2017, 35714.0, 5210
```

2. **Two highest prices:** The program will output the two highest prices contained in the [entire] loaded dataset.
3. **Average price in year and mileage range:** given *inclusive* low- and high-end values for a car's year and mileage, the program will output how many entries in the [full] loaded data are within that range and their average price.
4. **Best value:** The program will prompt the user for *non-inclusive* minimum mileage and price thresholds. The program will then output the "best (highest) value" entry's year, brand, miles, and price from among those exceeding the given thresholds. For a given car with year *y*, mileage *m*, and price *p*, the value is calculated as $y - m/13{,}500 - p/1{,}900$.
5. **Quit:** The program should display "Thank you for using the program!" and then stop running.

If an invalid menu option is entered, the program must display "Invalid option." and continue, displaying the menu again. However, you may assume the user always enters an integer.

We provide a sample run below to give you an idea of the program execution interface. **Your user interface must match the sample runs below exactly.** We have provided (link below) a starter code file named `CarDataAnalysis.java` that contains the menu portion of the program already implemented. It also contains constant definitions (using the `static final` keywords) which represent the indices of the columns to be loaded

from the file; *e.g.*, `BRAND` has value 2 because the car brand is held in the third column in the file, which has index 2.

Assumptions:

- When searching for brand matches, your program must ignore differences in capitalization.
- You may assume that the user enters data of the correct type for the expected input. For the range query, you may also assume that the provided lower bounds are indeed lower than provided upper bounds.
- You may assume that the input dataset file always exists in the same folder as the program, is properly formatted, and contains at least one entry.

Project Implementation:

For most of you, this assignment will probably take at least as long to complete as Project 2 (perhaps longer), so you should start working on this project as soon as possible. We strongly recommend developing this solution in an incremental manner. Here are stages of program completion that you should aim for, in the order given here to match the concepts covered in each module:

- Make a folder on your hard drive for this project. Download the proj3.zip file into that folder and decompress it. You should see a `CarDataAnalysis.java`, which contains the starter code, and a `USA_cars_datasets.csv` file, which contains the dataset.
- The starter code already contains a main method that asks for the input filename and then repeatedly displays the menu, gathering the user's choice and ending when they choose to quit. It also contains a helper method to display the menu, a helper method stub for counting the lines in the given file, and constant declarations at the top of the file. The latter are to be used as indices into the columns loaded from the dataset to extract those columns which are of interest. Compile and run the file to confirm it works before you move on.
- Complete the `countFileLines` method and confirm it's working by comparing the program's output with the loaded lines shown in the sample run below.
- Use the result from `countFileLines` to allocate arrays of the appropriate type and size to hold the *brand*, *year*, *mileage*, and *price* values for each car. Then, write a helper method to load these columns from the given file.

  This method should read the file one line at a time, skipping the file's header line. With each line, it should use the `.split` string method, passing "," as argument so that entries are split by commas. This method will return a string array of entries, one for each column. For example, if the variable `x` contains the string "ack, bork, clack", then the call `x.split(",")` will return an array of length three, containing "ack" at index 0, "bork" at index 1, and "clack" at index 2.

Use the `Integer.parseInt` and `Double.parseDouble` as needed to convert strings holding numeric entries to their appropriate types before placing them in the arrays. For example, if the variable `x` holds the string "123", then the call `Integer.parseInt(x)` will return the integer value 123. Analogously, if the variable `x` holds the string "1.23", then the call `Double.parseDouble(x)` will return the double value 1.23.

- For each menu item (except Quit), make at least one helper method (potentially more) to handle the operation. Include javadoc comments for each. Once ready, compile and run your program to test that the implemented method is now working correctly. Debug as needed before proceeding to the next menu item implementation. Your submitted solution must have **at least 5 helper methods** beyond those provided in the starter code.
- Test, test, test. And fix any lingering checkstyle errors.

Note: the starter file uses **FileNotFoundException**. If you close the Scanner object used to read from a file (which in turn closes the associated file stream), this should suffice. If instead you close the file stream (as done in the class notes and zyBook), then you should change your code to use **IOException** instead.

Sample Runs (user input in bold)**:**

```
Welcome to the car dataset analysis program.
Please enter input csv filename: USA_cars_datasets.csv
File has 2499 entries.

[1]: Average price of brand.
[2]: Two highest prices.
[3]: Average price in year and mileage range.
[4]: Best value.
[5]: Quit.
Please select an option: 1
Please enter a car brand: Ford
Please enter an output filename: ford.csv
There are 1235 matching entries for brand Ford with an average
price of $21666.89.

[1]: Average price of brand.
[2]: Two highest prices.
[3]: Average price in year and mileage range.
[4]: Best value.
[5]: Quit.
```

Please select an option: **1**
Please enter a car brand: **Citroen**
Please enter an output filename: **citroen.csv**
There are no matching entries for brand Citroen.

[1]: Average price of brand.
[2]: Two highest prices.
[3]: Average price in year and mileage range.
[4]: Best value.
[5]: Quit.
Please select an option: **2**
The two highest prices are $84900.00 and $74000.00.

[1]: Average price of brand.
[2]: Two highest prices.
[3]: Average price in year and mileage range.
[4]: Best value.
[5]: Quit.
Please select an option: **3**
Please enter the year lower bound: **2010**
Please enter the year upper bound: **2015**
Please enter the mileage lower bound: **25000**
Please enter the mileage upper bound: **50000**
There are 59 matching entries for year range [2010, 2015] and mileage range [25000, 50000] with an average price of $19722.71.

[1]: Average price of brand.
[2]: Two highest prices.
[3]: Average price in year and mileage range.
[4]: Best value.
[5]: Quit.
Please select an option: **4**
Please enter lower mileage threshold: **1000**
Please enter lower price threshold: **500**
The best-value entry with more than 1000.0 miles and a price higher than $500 is a 2019 jaguar with 20849.0 miles for a price of $2800.

[1]: Average price of brand.
[2]: Two highest prices.
[3]: Average price in year and mileage range.

```
[4]: Best value.
[5]: Quit.
Please select an option: 5
Thank you for using the program!


Process finished with exit code 0
```

General Project (Hard) Requirements:

- **We reserve the right to assign a 0 grade for any submission that does not compile.**
- **You absolutely may not use ArrayLists or any other Java Collection classes in this assignment.**
- Your solution must be named **`CarDataAnalysis.java`** (capitalization matters!)
- Your file must have a Javadoc class comment that briefly explains the purpose of the program, the author's name, JHED, and the date.
- The program must be fully checkstyle-compliant using our course required `check112.xml` configuration file. This means that the checkstyle audit completes without reporting any errors or warnings.
- You must also use good style with respect to class/method/variable names, etc.
- Submit **only the `.java` file named `CarDataAnalysis.java`** to the Project 3 assignment on Gradescope before the deadline date. [Do not submit the car dataset file.]
- You must submit the code before the deadline (with consideration of the grace period and late penalty.) You can resubmit as many times as you want before the deadline. We will only grade your most recent submission.

Grading Breakdown: The following points are awarded only if your submitted files compile, since **assignments that don't compile may earn a 0 grade.**

- [29] Program functionality
- [10] Creation and use of methods (at least 5, including file loader method)
- [5] Method documentation (javadoc comments)
- [4] General style
- [2] Submission & runtime errors