Congratulations on making it to the end in Java. For those where this was their first course in programming, nicely done getting to the end. This will likely be the hardest programming course you will take as you had to learn both programming and syntax. In future languages, knowing the specific syntax and nuances of other languages will be the hardest.

In this lab, you will be creating a license registration tracking system for the Country of Warner Brothers for the State of Looney Tunes. You will create four classes: Citizen, CarOwner, RegistrationMethods, and RegistrationDemo. You will build a CitizenInterface and CarOwnerInterface and implement CitizenInterface and CarOwnerInterface for Citizen and CarOwner classes respectively. You will create RegistrationMethods class that implements RegistrationMethodsInterface(provided).

They are several ways to approach this lab in a systematic manner. My suggestion would be to come up with the interfaces first (Citizen and CarOwner) and then build the Citizen and CarOwner classes that implement these. Or you can start with CitizenInterface.java and then Citizen.java before completing CarOwnerInterface.java and then CarOwner.java.

The UMLs for Citizen and CarOwner will be key to completing the four java files mentioned above.

Citizen
- firstName: String - lastName: String
+setFirstName(String inFirst): void +getFirstName(): String +setLastName(String inLast): void +getLastName(): String +toString(): String

CitizenInterface

Create the javadocs intro section with a short description along with the author, version, and since tags. Then write out the method headers and the applicable javadocs for each method. Each method should have a short Javadoc description along with any applicable @param tags and any applicable @return tag. As a reminder each public method from the Citizen UML should be included in the interface. The interface method headers should not include the word public since only public methods are included.

Citizen Class

- Citizen will implement CitizenInterface. This means that all methods in CitizenInterface will need to be incorporated into Citizen. The only way around this is to create abstract methods forcing Citizen to be an abstract class. However, all methods should be created for Citizen. The easiest way to do this is to copy CitizenInferface, paste into a Citizen.java file, and then add the constructors and build out the methods.
- Since we are saving CarOwner objects to binary files and CarOwner extends Citizen, it will be necessary to import java.io.Serializable and to also implement it along with CitizenInterface. Remember you can only extend one class, but can implement multiple separated by commas.
- It will also be necessary to create two constructors: (1) default no arg constructor (2) constructor that accepts two String params to update firstName and lastName instance field vars. Create applicable javadocs including @param tags for the constructors.

- .toString() should return firstName, a space, and lastName.

CarOwner
- license: String - month: int - year: int
+setLicense(String inLicense): void +getLicense (): String +setmonth(int inMonth): void +getMonth(): int +setYear(int inYear): void +getYear(): int +compareTo(Object o): int +toString(): String

CarOwnerInterface

- **Extends Comparable**
- Create the javadocs intro section with a short description along with the author, version, and since tags. Then write out the method headers and the applicable javadocs for each method.
- Since we want to be able to sort CarOwner objects, we will need to have a compareTo( ) that will be used to determine if one CarOwner is less than another, more than another, or equal to another based on registration year and month. Since we have compareTo( ), the interface will need to extend Comparable. If you do not extend Comparable, a CarOwner [ ] will not sort. Since CarOwner is implementing CarOwnerInterface, the Comparable extension will flow to CarOwner.

CarOwner Class

- CarOwner will implement CarOwnerInterface. This means that all methods in CarOwnerInterface will need to be incorporated into CarOwner. The easiest way to do this is to copy CarOwnerInterface, paste into a CarOwner.java file, and then add the constructors and build out the methods.
- Since we are saving CarOwner objects to binary files, it will be necessary to import java.io.Serializable and to also implement it along with CarOwnerInterface.
- It will also be necessary to create two constructors: (1) default no arg constructor (2) constructor that accepts three String params and two integers. You need to update firstName and lastName instance field vars. HINT – Use super( ) along with any arguments to update first and last. Set instance field vars license, month, and year. Create applicable javadocs including @param tags for the constructors.
- .toString() overrides and returns firstName, a space, and lastName, license, month/year. HINT – using super.toString( ) along with "\t" and "/" will help. NOTE – Sometimes it is necessary to use two tabs to ensure that you go over enough space before writing the next item to line everything up. Pitfall – Sometimes the text output looks misaligned. If you print to pdf, you will see that the columns do line up.
- See the FinalProject file for the compareTo( ). Make sure you read the javadocs to understand what is happening in compareTo( ).

RegistrationMethods

This is where most of the logic of the DMV is happening and where most of the coding effort is at for the Final Project. Key is to keep this modular. One method at a time, do not get overwhelmed. Recommend completing the methods in the order of what is required in RegDemo. This is also the order of methods in the provided RegistrationMethodsInterface.

1) setFileNames( )
    a. Message the user to provide the path to the registration.csv file.
    b. Use Scanner to create an object and use an applicable method to collect user input
    c. Recommend using a while pit or try/catch like SalesReport.java pg713+ to ensure that you have a valid file path to set the applicable instance field var
    d. Message the user to enter a path for the output.txt file, collect the path, and set the applicable instance field var. NOTE – Java will create a new file.
    e. Message the user to enter a path for the binFile.dat file, collect the path, and set the applicable instance field var. NOTE – Java will create a new file.
2) processTextToArrayList (ArrayList<CarOwner> inList)
    a. Set up a File object tied to inputFileName and pass to a Scanner constructor to set up a stream. You will need to do this in a try/catch.
    b. Use nextLine() to get to the 2nd line since the first line is the column headers.
    c. Use a while loop to test for hasNextLine( ). While true
        i. Get the line
        ii. Use .split(???) to tokenize. You have to replace the ??? with the correct item. HINT-How are items in a csv file separated across a line
        iii. Assign the token pieces to the array values as applicable.
        iv. Create a CarOwner object and then add to the ArrayList passed into this method
    d. Make sure to close out the stream
3) printArrayListToFile(ArrayList<CarOwner> inList, String inMsg)
    a. Set up a File object tied to outFileName and pass to a PrintWriter constructor to set up a stream. You will need to do this in a try/catch. We use File here so that each time you run, you delete the existing output file.
    b. Print inMsg to the stream
    c. Run through the ArrayList either using a for or for each loop to print each item in the ArrayList to the stream
    d. You will want to print an additional line in the file for spacing
    e. Make sure to close out the stream
4) writeListToBinary(ArrayList<CarOwner> inList)
    a. Set up a FileOutputStream object tied to binFileName and pass to an ObjectOutputStream to set up a stream.
    b. Use writeObject( ) to write the inList param to the stream. This writes the entire ArrayList as a single vs individual items as object to the file
    c. Make sure to close out the stream
    d. NOTE - You will need to do the above a try/catch. HINT-You will need to catch two different exceptions.

5) readListFromBinary()
    a. Set up a FileInputStream object tied to binFileName and pass to an ObjectInputStream to set up a stream.
    b. Use readObject( ) and assign to an ArrayList<CarOwner> object. Remember when using readObject(), you have to cast. In this case, you will be casting to (ArrayList <CarOwner>) and grabbing the entire list as one object vs individual object items from the ArrayList.
    c. Make sure to close out the stream
    d. NOTE - You will need to do the above a try/catch. HINT-You will need to catch three different exceptions.
    e. Create a CarOwner[ ] to the size of the ArrayList
    f. Use a for or for each loop to get an item from the ArrayList and assign to a corresponding array element.
    g. Return the CarOwner[ ] created
6) printArrayListToFile(ArrayList<CarOwner> inList, String inMsg)
    a. Set up a FileWriter object tied to outFileName set to append and pass to a PrintWriter constructor to set up a stream. You will need to do this in a try/catch. We need to use FileWriter here since we want to append to what printArrayListToFile wrote to the output file
    b. Print the inMsg to the stream
    c. Run through the ArrayList either using a for or for each loop to print each item in the ArrayList to the stream
    d. You will want to print an additional line in the file for spacing
    e. Make sure to close out the stream
7) flagOverdueOwners(CarOwner[] inArray) - This will be similar to the findAll( ) from Ch7
    a. Create an int with the monthsTotal. Remember each year has 12 months, so if you take REG_YEAR*12 and add REG_MONTH, you have the total months.
    b. Create a counter set to 0
    c. Make one pass using a for loop through inArray to see how many owner have registration is over 12 months old. You can do this by subtracting the inArray[i].getYear*12+inArray[i].getMonth from totalMonths. If >12, increment the counter.
    d. Create a new CarOwner[ ] sized to the counter
    e. Make another pass using a for loop like in step c, but this time, assign the value to index 0, make sure to have an index var that is incremented each time you assign a value.
8) flagAlmostDueOwners(CarOwner[] inArray) – This is like flagOverdueOwners above
    a. Follow the steps for flagOverdueOwners above except in step c, the criteria will be monthsTotal - (inArray[i].getYear()*12+inArray[i].getMonth())>9 &&
        monthsTotal - (inArray[i].getYear()*12+inArray[i].getMonth())<=12

RegDemo

1) Create a RegistrationMethods object called dmv
2) Invoke setFileNames( ) using the object created above
3) Create an ArrayList<CarOwner> collection called ltState.

4) Invoke processTextToArrayList( ) passing in ltState as an argument
5) Invoke printArrayListToFile () passing the appropriate arguments (see RegistrationMethods methods summary above). For the String argument pass the following phrase "Initial Set of Car Owners - Unsorted"
6) Invoke writeListToBinary() using ltState as an argument
7) Invoke readListFromBinary() assigning the result to CarOwner[ ] ltStateCopy.
8) Use Arrays.sort method to sort ltStateCopy
9) Use the appropriate RegistrationMethods method to print a copy of ltStateCopy with the String header "Sorted list based on Registration date"
10) Use flagOverdueOwners( ) to create a new CarOwner[ ] whose registration is over 1 yr old that is assigned to CarOwner[ ] overdue based on passing in ltStateCopy.
11) Use the appropriate RegistrationMethods method to print overdue with the String header "Owners with Expired Registration"
12) Use flagAlmostDueOwners( ) to create a new CarOwner[ ] whose registration will expire within the next 3 months or less, but is not over one year old (registrations that are 10-12 months old) that is assigned to CarOwner[] almostDue based on the array ltStateCopy.
13) Use the appropriate RegistrationMethods method to print almostDue with the String header "Owners with registration expiring in three months or less"