University of Illinois @ Urbana-Champaign

CI 487: Data Structures for Education

Implementation #1: Vehicle Database

---

1 Objectives and Overview

By the end of this assignment:

- A review of constructing classes and instantiating objects in Java
- A review of overriding functions
- Reading XML files in Java.
- Using the parsed XML data to instantiate objects.

2 Structures and Specifications

This assignment will consist of the following files:

- `vehicles.xml` → A large XML file from the US Department of Energy containing data on vehicles.
- `Vehicle.java` → The object we will use to store information on each vehicle from vehicles.xml.
- `VehicleDatabase.java` → This class will be used to (1) parse in the file, (2) instantiate `Vehicle` objects based on the XML elements that are parsed, and (3) allow the user to use query information in order to search for vehicles.
- `VehicleInfoSearch.java` → Uses the `VehicleDatabase` object to setup a small filter-based, search engine.

In combination, these files will power an app that (1) reads in data from the XML file, (2) cleans that data and constructs a List of objects from it, and (3) returns filtered results based on a user query.

2.1 vehicles.xml

The general format of the original file is quite verbose and contains more data than we need. The process of extracting these fields will be detailed in the next section but for now, here is the list of the ones you should be interested in:

1. `<cylinders> INTEGER </cylinders>` → A count on the number of cylinders.
2. `<drive> STRING </drive>` → Indicates what type of drive the car is (e.g., rear wheel drive).
3. `<fuelType> STRING </fuelType>` → The type of fuel the engine works on (e.g., Gas, Diesel).
4. `<make> STRING </make>` → The brand of vehicle.
5. `<model> STRING </model>` → The model of the vehicle.
6. `<year> INTEGER </year>` → The year of that make and model.
7. `<trans> STRING </trans>` → Indicates the transmission type.
8. `<VClass> STRING </VClass>` → The type of vehicle (e.g., truck, sedan).

2.2 Vehicle.java

To begin with, we will make a `Vehicle` class which will contain information on each of the vehicles we will eventually read in from our XML file.

2.2.1 Attributes

The first thing we will need for our class are some attributes to store information on the vehicles that will eventually be instantiated. Reference Section 2.1 and create attributes for each of the vehicle elements we will be storing information on. Be sure to (1) make each attribute is of the correct type and (2) all attributes are `private` and `final`. The purpose of `private` is to ensure encapsulation and `final` is used to ensure that, once set, the values associated with each attribute cannot be changed.

2.2.2 Constructor

Our constructor should have the following general form:

```
public Vehicle(...){
    // Body here
}
```

Your Task: Complete the constructor such that it takes parameters for each of the vehicles attributes and sets them. Refer to the attributes in section 2.2 for which attributes should be included in the constructor and what their type should be.

2.2.3 Getters

Since our attributes were declared with the `private` keyword they will not be accessible outside of the class. This keyword allows us to control access to attributes by defining "getter" functions to allow instances of our class, as they're used in other classes, access to private variables. Additionally, we can allow other classes to modify the values of private variables using "setter" functions.

For our Vehicle class we will only be needing to define getter functions since all of our attributes are `final` and therefore cannot be modified after their initial assignment. In general, getters function names should be the word "get" followed by the name of the attribute we want to get. For instance, if your class has a private attribute string attribute named `studentName` the associated getter function name should be:

```
public String getStudentName(){
    return studentName;
}
```

Your Task: Define a getter for all of the attributes of your class using this general form.

Hint: Defining getters and setters is tedious so most modern IDEs can auto-generate getters and setter. Explore your IDE and see if you can figure out how to use this functionality.

2.2.4 String Overrides

Finally, we want the ability to print the contents of our class in a pretty, readable manner. By default, if you instantiate a class and then print it you will get the something along the lines of the following output:

Foo.java

```
public class Foo{
    //...
}
```

Main.java

```
public class Main{
    Foo c1 = new Foo();
    System.out.println(c1);
}
```

Terminal:

```
> java Main
Foo@6504e3b2
```

By overriding the `toString` function we can define how this class should be converted to a string. This in turn defines what string should be printed when printing an instance of the class. For instance, if the `Foo` class has an attribute `name` we can provide a `toString` method for that class and our program will now run in the following way:

Foo.java

```
public class Foo{
    public String name = "bar";
    //...
    @Override
    public String toString(){
        return "Foo name: " +
            this.name;
    }
}
```

Main.java

```
public class Main{
    Foo c1 = new Foo();
    System.out.println(c1);
}
```

Terminal:

```
> java Main
Foo name: bar
```

Your Task: Override the `toString` method such that when an instance of the `Vehicle` class is printed it appears in the following form:

```
--Year Make Model--
Cylinders: CylinderCount
Drive: DriveConfig
Fuel Type: FuelType
Transmission: TransmissionType
Class: VehicleClass
```

2.2.5 Comparable Override

In your provided starter files you will notice that the constructor is declared in the following way:

```
public class Vehicle implements Comparable<Vehicle>{
    //...
}
```

`Comparable` is an interface. We will cover interfaces in greater detail later but, for the time being it is sufficient to know that an interface is a construct in Java which contains function headers which have not been implemented. For `Comparable`, one of these functions is the `compareTo` method which is used by functions such as `Collections.sort`. In a later portion of the assignment we will be sorting collections of `Vehicle` instances and, as such, we must implement the `compareTo` method by returning a comparison between attributes in the class for which the `compareTo` method has already been implemented (e.g., `String`, `Integer`). For example:

```
public class Foo implements Comparable<Foo>{
    String name;
    public Foo(String name){
        this.name = name
    }
    @Override
    public int compareTo(Foo other) {
        return name.compareTo(other.name);
    }
}
```

Your Task: Change the class such that it implements `Comparable<Vehicle>` and implement the method `compareTo` method such that it compares vehicle objects based on their year.

2.2.6 Vehicle Class - Checklist

- Class Attributes:
  - ☐ All needed attributes are declared.
  - ☐ Attributess are `private`.
  - ☐ Attributess are `final`.
- Getters
  - ☐ A getter function exists for each attribute.
  - ☐ All getters are of the form `getAttributeName`.
- Overrides
  - ☐ The `tosString` method is present and returns a string of the specified format.
  - ☐ The `compareTo` method is present and returns the comparision between the intance and a `Vehicle` instance that is passed in as a parameter. The comparison should be based on year of the vehicles being compared.

2.3 VehicleDatabase.java

The XML document we are given contains many many fields with various information on each type of vehicle. We want our parse to do the following:

1. Parse the data in using the XML java DOM parser.
2. Filter out the data we are interested in and retrieve the values.
3. Instantiate Vehicle objects with those values.

2.3.1 Attributes

For our class we will need two `private` attributes:

1. `vehicleDatabase` → An `List` of `Vehicle` instances. This will contain all of the vehicles in our database.
2. `makes` → An `List` of strings. This will contain a list of all the unique makes in our database.

You do not need to do anything for this task as those attributes are declared outside of the construction and initialized within it.

2.3.2 Constructor and XML Parsing

For this portion of the assignment, create a constructor that takes a single string, `filepath`. This file path is intended to be either an absolute or relative path to an XML file of the same form as vehicles.xml. The purpose of our constructor is to parse this file and use it's contents to populate our class's attributes.
The general template for parsing an XML file in Java using `javax` is as follows:

```
DocumentBuilderFactor dbf = DocumentBuilderFactory.newInstance();

DocumentBuilder db = null;
try{
    db = dbf.newDocumentBuilder();
} catch(ParserConfigurationException e{
    //...
}

File f = new File(filepath);

Document doc = null;
try{
    doc = db.parse(f);
} catch(SAXException | IOException e){
    //...
}
```

Following this set of operations you will be left with `dbf`. From there you should use the `getElementsByTagName` function. This will produce a `NodeList` which contains instances of the `Element` class where each `Element` is a portion of XML that had the given tag. You can then iterate over that `NodeList` in the following way:

```
NodeList nodes = doc.getElementsByTagName("foo")
for(int i = 0; i < nodes.getLength(); i++){
    Element elem = (Element) nodes.get(i);
    //process elem
}
```

Use this to parse the document in the following way:

1. Get a `NodeList` of all tags in the XML document with the `"vehicle"` tag.
2. Iterate over the list and, at each iteration use to following structure to extract each vehicle attribute we are interested in:

   ```
   elem.getElementsByTagName(??).item(0).getTextContent();
   ```

   Be sure to convert the text content to `Integer` when needed.
3. At each iteration create a new vehicle instance and add that instance to the `vdb` list.

Your Task: Reference section 2.1 to determine what the names for each of the fields is. Go through the provided skeleton code and replace all instances of "??" within the `getElementByTagName` method with the appropriate tag. Additionally, after extracting all the needed values, instantiate a new vehicle class with those values and add it to the database.

2.3.3 Getters

The only getter you will need for this class is for the `makes` attribute. We will be building other query function that will allow the user to interact with the database. This getter (`getMakes`) has been provided for you.

2.3.4 Query Functions

To interact with the "database" we will be implementing the following functions

`public List<String> queryClasses(String userMake){ ... }` This function should search the database, and return a *sorted* List of string containing all the *unique* vehicle classes that are of the specified `userMake`. Be sure to ignore the case of the string `userMake` when making comparisons.

`public List<String> queryModels(String userMake, String user_class){ ... }` This function should search the database and return a `sorted` List of strings containing all the *unique* vehicle models present in the database that have the specified `userMake` and `userModel`. Be sure to ignore the case of the parameters when making comparisons.

`public List<Vehicle> queryVehicles(String userModel){ ... }` Search the database and return an List of all the `Vehicle` instances in the database that match the model specified in the parameter `userModel`. Be sure to ignore the case of `userModel` when making comparisons.

2.3.5 VehicleDatabase Class - Checklist

- Constructor:
  - ☐ Populates the `vdb` and `makes` attributes with data from the XML file.
- Functions to Implement:
  - ☐ `queryClasses`
  - ☐ `queryModels`
  - ☐ `queryVehicles`

2.4 VehicleInfoSearch.java

This class will only have the `main` method and will structure the search functionality of the database. It should be implemented using the following steps:

1. Generate a database using the provided XML file.
2. Present the user with the list of available makes, ask them which they want, and store that input.
3. Use the make they selected to query which classes of vehicle are available with that make, present the result, then ask the user which they want.
4. User the previous make and class results to query which models are available, present the results of that query, and ask the user which they would prefer.
5. User the user input from the previous step to query the list of all those models, iterate over that list, and print each vehicle object to the user.
6. Ask the user if they want to continue. If they type `q` terminate the loop; otherwise, continue.

Your output, once finished, should look something like this:

```
Generating database...
Makes: [AM General, ASC Incorporated, Acura, Alfa Romeo, American
   Motors Corporation, Aston Martin, Audi, ...]
Enter a make: Audi
Classes: [Compact Cars, Large Cars, Midsize Cars, Midsize Station
   Wagons, Midsize-Large Station Wagons, ...]
Enter a class: Midsize Cars
Models: [100, 100 quattro, 200, 200 quattro, 200 quattro 20v, 5000 CS
   Turbo, 5000 CS quattro, ...]
Enter a model: 100

RESULTS:
--1993 Audi 100--
Cyliners: 6
Drive: Front-Wheel Drive
Fuel Type: Premium
Transmission: Automatic 4-spd
Class :Midsize Cars

--1993 Audi 100--
Cyliners: 6
Drive: Front-Wheel Drive
Fuel Type: Premium
Transmission: Manual 5-spd
Class :Midsize Cars

--1994 Audi 100--
Cyliners: 6
Drive: Front-Wheel Drive
Fuel Type: Premium
Transmission: Automatic 4-spd
Class :Compact Cars
...
```

Do note that the above example has replace some of the output for makes, classes, and models with ellipses. This is only to shorted the example. Your code should output each of the lists in their entirety.