CPSC150L Lab 13
Object Interactions

Fall 2017

1 Introduction

In previous labs, we saw that classes can invoke methods implemented in other classes. We did that when creating a `Scanner` object to read input from the user.

However, this type of interaction is of a casual type, one in which functionality is invoked as needed, with no dependency between classes beyond these invocations. Objects in most sophisticated object-oriented programs are linked to each other for longer periods of time. Using the `PersonName` class in the `Person` class is a good example of this.

In this exercise, we will use objects with this type of relationship; in this case cars and persons.

Cars seat people as passengers, and can hold as many passengers as its capacity allows. One of their passengers can be the driver if he/she is 16 years of age or older. Passengers can get inside a car if its capacity allows; and they can leave the car when pleasing. Since we are not simulating car traveling, a car can be left with passengers but no driver (quite an inconvenience in real life!).

Another new facet of this problem is the utilization of arrays of objects. When implementing the `Car` class, you will use arrays of `Person` objects to implement the passengers that will occupy the vehicle.

This is a 2 week lab.

2 Exercises

Fork and clone the `cpsc150l-lab13` repository in the 150 students group. You will need a completely functioning `Person` and `PersonName` class in order to complete this lab. You may use your `Person` and `PersonName` classes from labs 11 and 12 or the ones bundled in the lab 13 repository.

2.1 The `Car` Class

Create a class named `Car`.

This class models a car with a driver and several passengers; therefore it should have fields for the driver (of type `Person`) and passengers (an array of `Person` objects). The length of the array indicates the cars capacity (that is, its maximum number of passengers after the car is created). The cars cannot be created with a size less than the minimum of 4 passengers or a size greater than the maximum of 8 passengers. The driver is also a passenger whose age is 16 or older and should also be in the "passengers" array.

The class has two constructors and ten methods, which should be defined exactly as shown below.

- `public Car()`
  A constructor that creates a small car seating up to 4 people. It initializes the passenger array to that size, and the driver to `null`.
- `public Car(int aCapacity)`
  A constructor that creates a car whose capacity is set by the given parameter. If the parameter is less that 4, the capacity is set to the allowable minimum. If the parameter is greater than 8, the capacity is set to the allowable maximum. It initialiazes the passenger array and the driver as above.
- `public int getCapacity()`
  Returns the capacity of the `Car`.

- `public int getOccupancy()`
  Returns the number of `Person` objects currently in the `Car`.

- `public boolean hasRoom()`
  Returns whether or not the occupancy of the `car` is equal to the capacity of the `Car`.

- `public Person getDriver()`
  Returns the `Person` driving the `Car` or `null` if the `Car` has no driver.

- `public boolean hasPassenger(Person person)`
  It receives a person and indicates whether this person is a passenger. Any two `Person` objects are the same if they bear the same name and age.

- `public boolean setDriver(Person person)`
  If allowed, it assigns the person as the driver and returns `true`. The person can become the driver if the person is a passenger or can become a passenger, and if the person is of driving age. Remember, two persons are the same if they have the same name and age. The method fails and returns `false` if the person is not a passenger and cannot become a passenger (due to full occupancy), or the person is not of driving age. If there is a current driver, that driver is not removed from the array of passengers.

- `public Person[] getPassengers()`
  Returns the list of passengers in the car. The array returned should be a shallow copy of the one kept in the car (thus changes in the copy are not reflected in the original). To make a shallow copy, create a new array of persons, copy all passengers to the new array, and return this copy at the end of the method. Be aware that returning a reference to the list of passengers does not result in a copy. This array should be an exact copy of the passengers array, that is if an entry in passengers is `null`, the corresponding entry in the returned array should also be `null`.

- `public boolean addPassenger(Person person)`
  It adds a person as a passenger. The method fails if the person to add is `null`, if the person is a passenger already, or if there is no room left in the car to add it. The person can be added to the first available seat, i.e., the first seat that is `null`.

- `public boolean removePassenger(Person person)`
  It receives a person and removes it as a passenger. If the person is also the driver,

then the driver seat becomes available (that is, it is set to `null`). The method fails if the person is `null`, or if he/she is not a passenger. It may be helpful to move all the occupied seats (non-`null` `Person` objects) to the front of the array.

- `public boolean canDrive(Person person)`
  It indicates whether a person is of driving age (that is, 16 years or older). The method returns false if the person is `null` or if the person's age is less than 16.

Review Lab 8 to see which methods in `Person` and `PersonName` will be most useful. Test your code against `CarTest.java`.

3 Common Mistakes

The solutions to some common mistakes are as follows.

1. Be sure to check for `null`!!!
2. Pay attention to the maximum and minimum capacities of a `Car`. That is, how many people should `new Car(2)` hold? How about `new Car(6)` and `new Car(10)`?
3. Be careful when removing the driver of a `Car`.