Com S 227
Spring 2019
Assignment 1
100 points

Due Date: Monday, February 11, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm Feb 10)

10% penalty for submitting 1 day late (by 11:59 pm Feb 12)

No submissions accepted after February 12, 11:59 pm

This assignment is to be done on your own. See the Academic Dishonesty policy in the syllabus, http://www.cs.iastate.edu/~cs227/syllabus.html **, for details.**

You will not be able to submit your work unless you have completed the *Academic Dishonesty policy questionnaire* on the Assignments page on Canvas. Please do this right away.

If you need help, see your instructor or one of the TAs. Lots of help is also available through the Piazza discussions.

Please start the assignment as soon as possible and get your questions answered right away. It is physically impossible for the staff to provide individual help to everyone the afternoon that the assignment is due!

This is a "regular" assignment so we are going to read your code. Your score will be based partly (about a third) on the specchecker's functional tests and partly on the grader's assessment of the quality of your code. See the "More about grading" section.

Tips from the experts: How to waste a lot of time on this assignment

1. Start the assignment the night it's due. That way, if you have questions, the TAs will be too busy to help you and you can spend the time tearing your hair out over some trivial detail.
2. Don't bother reading the rest of this document, or even the specification, especially not the "Getting started" section. Documentation is for losers. Try to write lots of code before you figure out what it's supposed to do.
3. Don't test your code. It's such fun to remain in suspense until it's graded!

Overview

The purpose of this assignment is to give you some practice with the process of implementing a class from a specification and testing whether your implementation is correct.

For this assignment we will implement one class, called **`UberDriver`**. It is slightly more complex than the **`Atom`** or **`Basketball`** classes that you saw in Lab 2, so be sure you have done and understood Lab 2.

The **`UberDriver`** class simulates the activity of a driver for a shared-ride service similar to Uber or Lyft. The **`UberDriver`** is configured with a *per-mile rate* and a *per-minute rate*. When driving with one or more passengers, the driver is credited according to the number of miles driven and also the number of minutes spent driving, multiplied by the number of passengers currently in the vehicle. When there are zero passengers in the vehicle, the driver earns nothing. The money earned is accumulated in the *total credits*. For example, suppose a driver earns 1.00 per mile and .20 per minute, and performs the following actions. The right-hand column shows the effect on the total credits.

Action	Driver Credits
Drive 2 miles over a period of 5 minutes	+0
Wait around 3 minutes	+0
Pick up a passenger	+0
Drive 5 miles in 10 minutes	+5.00 for 5 miles, +2.00 for 5 minutes
Pick up another passenger	+0
Drive 2 miles at 10 miles per hour	+4.00 for 2 miles * 2, +4.80 for 12 minutes * 2
Drop off one passenger	+0
Sit in traffic for 15 minutes	+0 for miles, +3.00 for 15 minutes

The driver credits above total 18.80. However, to figure out how much the driver actually earns, we also have to account for the fact that it costs money for gas and maintenance on the vehicle, which we assume is a constant amount of .50 per mile. To determine the driver's profit, we can take the total credits and subtract the operating cost times the number of miles. In this example, the driver has gone 9 miles, so the operating cost is 9 * .50, or 4.50, and so the driver's profit is 14.30. The driver has been working for 45 minutes, so the profit per hour is 60 * 14.30 / 45, or approximately 19.07 per hour. (Note we are ignoring the potential operating costs associated with just the minutes, e.g. idling or sitting in traffic. Note also that in real life, the fare paid by a passenger is significantly higher than the amount credited to the driver, since the passenger fare includes booking fees and commissions.)

Please note that you do not need any conditional statements (i.e. "if" statements) or anything else we haven't covered, for this assignment. There will be a couple of places where you need to choose the larger or smaller of two numbers, which can be done with the methods

`Math.max()` or **`Math.min()`**. (You will probably not be penalized for using conditional statements instead, but your code will become longer and more complicated!)

Specification

Your **`UberDriver`** class must include, and use, the following constant definitions:

```
/**
 * Maximum number of passengers allowed in the vehicle at one time.
 */
public static final int MAX_PASSENGERS = 4;

/**
 * Cost to operate the vehicle per mile.
 */
public static final double OPERATING_COST = 0.5;
```

There is one public constructor and eleven public methods:

`public UberDriver(double givenPerMileRate, double givenPerMinuteRate)`
Constructs an UberDriver with the given per-mile rate and per-minute rate.

`public int getTotalMiles()`
Returns the total miles driven since this UberDriver was constructed.

`public int getTotalMinutes()`
Returns the total minutes driven since this UberDriver was constructed.

`public void drive(int miles, int minutes)`
Drives the vehicle for the given number of miles over the given number of minutes.

`public void waitAround(int minutes)`
Simulates sitting in the vehicle without moving for the given number of minutes. Equivalent to **`drive(0, minutes)`**.

`public void driveAtSpeed(int miles, double averageSpeed)`
Drives the vehicle for the given number of miles at the given speed. Equivalent to **`drive(miles, m)`**, where m is the actual number of minutes required, rounded to the nearest integer. Caller of method must ensure that **`averageSpeed`** is positive.

`public int getPassengerCount()`
Returns the number of passengers currently in the vehicle.

`public void pickUp()`

Increases the passenger count by 1, not exceeding **`MAX_PASSENGERS`**.

`public void dropOff()`

Decreases the passenger count by 1, not going below zero.

`public double getTotalCredits()`

Returns this UberDriver's total credits (money earned before deducting operating costs).

`public double getProfit()`

Returns this UberDriver's profit (total credits, less operating costs).

`public double getAverageProfitPerHour()`

Returns this UberDriver's average profit per hour worked. Caller of method must ensure that it is only called when the value of **`getTotalMinutes()`** is nonzero.

Where's the main() method??

There isn't one. This isn't a complete program and you can't "run" it by itself. It's just a single class, that is, the definition for a type of object that might be part of a larger system. To try out your **`UberDriver`** class, you can write a test class with a main method, analogous to **`AtomTest`** in Lab 2. See the "Suggestions for getting started" section below for some examples. (You can also find these examples in the class SimpleTest.java, which is posted along with the homework spec on Piazza.)

There is also a specchecker (see below) that will perform a lot of functional tests, but when you are developing and debugging your code at first you'll always want to have some simple test cases of your own as in the getting started section.

Suggestions for getting started

*Smart developers don't try to write all the code and then try to find dozens of errors all at once; they work **incrementally** and test every new feature as it's written. Since this is our first assignment, here is an example of some incremental steps you could take in writing this class.*

0. Be sure you have done and understood Lab 2.

1. Create a new, empty project and then add a package called **`hw1`**.

2. Create the **`UberDriver`** class in the **`hw1`** package and put in stubs for all the required methods, the constructor, and the required constants. Remember that everything listed is declared **`public`**. For methods that are required to return a value, just put in a "dummy" return statement that returns zero or false. **There should be no compile errors**.

3. Javadoc the class, constructor and methods. This is a required part of the assignment anyway, and doing it now will help clarify for you what each method is supposed to do before you begin the actual implementation. (Copying method descriptions from this document is perfectly acceptable; don't forget the @param and @return tags for completeness.)

4. Work through the calculation of driver credits and driver profit on page 2. Use a pencil and paper and write the calculations down.

5. To begin thinking about instance variables, look at the accessor methods. It should be clear from the example that the driver credits and profit depend on the miles driven and the minutes spent driving, as well as the number of passengers. So it might make sense to start with those three values. Take **`getTotalMiles`**. In order to return the total miles, your class has to keep track of that value, suggesting that you need an *instance variable* to store the total miles. Start with a simple test case showing exactly what you expect the method to do:

```
import hw1.UberDriver;

public class SimpleTest
{
  public static void main(String[] args)
  {
    UberDriver d = new UberDriver(1.00, 0.20);
    d.drive(10, 25);
    d.drive(7,  35);
    System.out.println(d.getTotalMiles()); // expected 17
  }
}
```

Then define an appropriate instance variable, initialize it in the constructor, and implement the **`getTotalMiles`** method to return the value and the **`drive`** method to update it. (*Tip: you can find these sample test cases in the file SimpleTest.java which is posted along with the homework spec.)*

6. Next, try **`getTotalMinutes`**, which should be similar. Again, write a simple test case first:

```
    d = new UberDriver(1.00, 0.20);
    d.drive(10, 25);
    d.drive(7,  35);
    System.out.println(d.getTotalMinutes()); // expected 60
```

7. While thinking about it, you might as well do the two other methods that affect miles and minutes. The **waitAround** method is very simple - note you can implement it just by calling **drive**, using 0 as the argument for miles. Here is a simple test:

```
d = new UberDriver(1.00, 0.20);
d.drive(10, 25);
d.waitAround(5);
System.out.println(d.getTotalMiles()); // expected 10
System.out.println(d.getTotalMinutes()); // expected 30
```

For **driveAtSpeed**, work out an example by hand first, e.g., if we drive 10 miles at 20 miles per hour, it's clear that it takes 10/20 = .5 hour, which is 30 minutes. How about if we drive 10 miles at 48 miles per hour?

```
d = new UberDriver(1.00, 0.20);
d.driveAtSpeed(10, 48);
System.out.println(d.getTotalMinutes()); // expected 13
System.out.println(d.getTotalMiles()); // expected 10
```

The answer comes out to 12.5 minutes; here you have to read the javadoc carefully to notice that the number of minutes should be rounded to the *nearest* integer, which is 13. To round, you can use the method **Math.round**, the usage of which requires an extra "cast" to type **int**. Here is an example of usage:

```
int roundedMinutes = (int) Math.round(exactMinutes);
```

Again, once you know the minutes, you can finish the implementation just by calling **drive**.

8. For the number of passengers, again start with a simple test case:

```
d = new UberDriver(1.00, 0.20);
d.pickUp();
d.pickUp();
System.out.println(d.getPassengerCount()); // expected 2
d.dropOff();
System.out.println(d.getPassengerCount()); // expected 1
```

The catch here is the requirement that **pickUp** will never allow the passenger count to go above the constant **MAX_PASSENGERS**, and **dropOff** will not allow the count to go below zero. That is, when the **pickUp** is called, the new passenger count should be either

- the existing count, plus 1, or
- MAX_PASSENGERS,

whichever is *smaller*. The easy way to accomplish this is with the method **Math.min**, which returns the smaller of two numbers. There is a similar method **Math.max**.

9. Now it is time to start the calculation of driver credit. As always, begin with a very simple test case, say, with just one passenger:

```
d = new UberDriver(1.00, 0.20);
d.drive(5, 15);  // has no effect on credits
d.pickUp();
d.drive(10, 25);
System.out.println(d.getTotalCredits());  // expected 15.0
```

It's important to notice that in general, the total credits cannot be *calculated* within the **`getTotalCredits`** method - the value has to be kept in an instance variable that is updated when **`drive`**, **`driveAtSpeed`**, or **`waitAround`** is called. The **`getTotalCredits`** method is just an *accessor* and therefore should not modify any instance variables. Note also that in order to figure out the credit to add in **`drive`**, we need to know the per-mile rate (here 1.00) and the per-minute rate (here 0.20). Those values are available as parameter variables within the constructor, but in order to use the values in other methods, they have to be stored in instance variables.

*You should **not** use conditional statements to check whether there is a passenger in the vehicle! Just determine the credit and multiply by the number of passengers.*

11. Test that it works when there are multiple passengers.

```
d = new UberDriver(1.00, 0.20);
d.pickUp();
d.drive(10, 25);
d.pickUp();
d.drive(7, 35);
System.out.println(d.getTotalCredits());  // expected 43.0
```

12. For **`getProfit`**, you need to know the total credits, and you need to know the total miles driven, from which you can get the operating cost. Since these values are already available in the instance variables you already have, **you should not define another instance variable for the profit** - just calculate the result and return it. This is an accessor method that should not modify any instance variables. The same applies to **`getAverageProfitPerHour`**. Do a sample calculation to create a simple test case.

```
d = new UberDriver(1.00, 0.20);
d.pickUp();
d.drive(10, 25);
System.out.println(d.getProfit());  // expected 15.0 - 5.0 = 10.0
System.out.println(d.getAverageProfitPerHour());  // expected 24.0
```

13. Think about other scenarios or combinations of operations that you have not already tested. E.g., do you get the correct credit if you **`waitAround`** when there are passengers in the vehicle? Do you get the correct average profit per hour? Do you get the correct credits for different combinations of rates?

14. At some point, download the SpecChecker, import it into your project as you did in labs 1 and 2, and run it. *Always start reading error messages from the top.* If you have a missing or extra public method, if the method names or declarations are incorrect, or if something is really wrong like the class having the incorrect name or package, any such errors will appear *first* in the output and will usually say "Class does not conform to specification." **Always fix these first.**

The SpecChecker

You can find the SpecChecker online; see the Piazza Homework post for the link. Import and run the SpecChecker just as you practiced in Labs 1 and 2. It will run a number of functional tests and then bring up a dialog offering to create a zip file to submit. Remember that error messages will appear in the *console* output. There are many test cases so there may be an overwhelming number of error messages. ***Always start reading the errors at the top and make incremental corrections in the code to fix them.*** When you are happy with your results, click "Yes" at the dialog to create the zip file. See the document “SpecChecker HOWTO”, which can be found in the Piazza pinned messages under “Syllabus, office hours, useful links,” if you are not sure what to do.

> Note that one of the specchecker test cases refers to a "longer scenario". For your reference, it is testing a driver with a per-mile rate of 2.5 and a per-minute rate of 0.30 performing the following actions:
>
> drive 5 miles in 20 minutes
> wait around 5 minutes
> pick up a passenger
> drive five miles in 20 minutes
> drop off passenger
> drive 2 miles in 10 minutes
> pick up a passenger
> drive 8 miles at 15 mph
> pick up another passenger
> drive 3 miles in 10 minutes
> drop off a passenger
> drive 2 miles in 10 minutes
> drop off passenger
> wait around for 10 minutes

More about grading

This is a "regular" assignment so we are going to read your code. Your score will be based partly (about a third) on the specchecker's functional tests and partly on the TA's assessment of the quality of your code. This means you can get partial credit even if you have errors, and it also means that even if you pass all the specchecker tests you can still lose points. Are you doing things in a simple and direct way that makes sense? Are you defining redundant instance variables? Are you using a loop for something that can be done with integer division? Some specific criteria that are important for this assignment are:

- Use instance variables only for the "permanent" state of the object, use local variables for temporary calculations within methods.
    - You will lose points for having unnecessary instance variables
    - All instance variables should be **`private`**.
- **Accessor methods should not modify instance variables**.

See the "Style and documentation" section below for additional guidelines.

Style and documentation

Roughly 15% of the points will be for documentation and code style. Here are some general requirements and guidelines:

- **Each class, method, constructor and instance variable, whether public or private, must have a meaningful javadoc comment**. The javadoc for the class itself can be very brief, but must include the **`@author`** tag. The javadoc for methods must include **`@param`** and **`@return`** tags as appropriate.
    - Try to briefly state what each method does in your own words. However, there is no rule against copying and pasting the descriptions from the online documentation.
    - Run the javadoc tool and see what your documentation looks like! (You do not have to turn in the generated html, but at least it provides some satisfaction :)

- All variable names must be meaningful (i.e., named for the value they store).
- Your code should not be producing console output. You may add **`println`** statements when debugging, but you need to remove them before submitting the code.

- Internal (//-style) comments are normally used inside of method bodies to explain *how* something works, while the Javadoc comments explain *what* a method does. (A good

rule of thumb is: if you had to think for a few minutes to figure out how something works, you should probably include an internal comment explaining how it works.)
    - Internal comments always *precede* the code they describe and are indented to the same level. In a simple homework like this one, as long as your code is straightforward and you use meaningful variable names, your code will probably not need any internal comments.

- Use a consistent style for indentation and formatting.
    - Note that you can set up Eclipse with the formatting style you prefer and then use Ctrl-Shift-F to format your code. To play with the formatting preferences, go to Window->Preferences->Java->Code Style->Formatter and click the New button to create your own "profile" for formatting.

If you have questions

For questions, please see the Piazza Q & A pages and click on the folder **`assignment1`**. If you don't find your question answered, then create a new post with your question. Try to state the question or topic clearly in the title of your post, and attach the tag **`assignment1`**. *But remember, do not post any source code for the classes that are to be turned in.* It is fine to post source code for general Java examples that are not being turned in. (In the Piazza editor, use the button labeled "pre" to have Java code formatted the way you typed it.)

If you have a question that absolutely cannot be asked without showing part of your source code, make the post "private" so that only the instructors and TAs can see it. Be sure you have stated a specific question; vague requests of the form "read all my code and tell me what's wrong with it" will generally be ignored.

Of course, the instructors and TAs are always available to help you. See the Office Hours section of the syllabus to find a time that is convenient for you. We do our best to answer every question carefully, short of actually writing your code for you, but it would be unfair for the staff to fully review your assignment in detail before it is turned in.

Any posts from the instructors on Piazza that are labeled "Official Clarification" are considered to be part of the spec, and you may lose points if you ignore them. Such posts will always be placed in the Announcements section of the course page in addition to the Q&A page. (We promise that no official clarifications will be posted within 24 hours of the due date.)

What to turn in

Please submit, on Canvas, the zip file that is created by the SpecChecker. The file will be named **`SUBMIT_THIS_hw1.zip`**. and it will be located in whatever directory you selected when you ran

the SpecChecker. It should contain one directory, **`hw1`**, which in turn contains one file, **`UberDriver.java`**. Please LOOK at the file you upload and make sure it is the right one!

Submit the zip file to Canvas using the Assignment 1 submission link and verify that your submission was successful. If you are not sure how to do this, see the document "Assignment Submission HOWTO" which can be found in the Piazza pinned messages under "Syllabus, office hours, useful links."

> We recommend that you submit the zip file as created by the specchecker. If necessary for some reason, you can create a zip file yourself. The zip file must contain the directory **hw1**, which in turn should contain the files **UberDriver.java**. You can accomplish this by zipping up the **src** directory of your project. **Do not zip up the entire project**. The file must be a zip file, so be sure you are using the Windows or Mac zip utility, and NOT a third-party installation of WinRAR, 7-zip, or Winzip.

Note: You will need to complete the "Academic Dishonesty policy questionnaire," found on the Assignments page on Canvas, before the submission link will be visible to you.