Com S 227
Fall 2023
Assignment 1
100 points

Due Date: Friday, September 15, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm Sept 14)

10% penalty for submitting 1 day late (by 11:59 pm Sept 16)

No submissions accepted after Sept 16, 11:59 pm

This assignment is to be done on your own. See the Academic Integrity policy in the syllabus, 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.

Contents

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 conforms to the specification. Along the way you will get lots of practice making decisions about choosing what instance variables are really needed to represent the internal state of an object.

For this assignment you will implement one class, called **`AirportVan`**, that models the activity of a shared-ride van service, in particular the problem of tracking the driver's earnings. The basic operation of the van, as time passes, is simulated by invoking one of the methods

```
public void drive(int miles, int minutes)
```

or

```
public void waitAround(int minutes)
```

The driver earns an hourly wage for all time that has passed according to the "minutes" parameter of the methods above. The wage is set as a constructor parameter. In addition, there are "bonus points" earned based on the distance driven and the time passed. The bonus points depend on the number of riders in the van along with two values, called the "mileage bonus" and the "time bonus" that are set via constructor parameters. The rules are basically as follows:

bonus points based on mileage =
miles driven x mileage bonus x number of riders (*)

bonus points based on time =
time passed x time bonus x number of riders (**)

However, there are two significant exceptions, marked by (*) and (**) above:

(*) if the van is *empty*, the number of riders is treated here as 1
(**) if the van has *one or more riders*, the number of riders is treated here as 1

In addition, a rider may leave the driver a tip when getting dropped off.

Here is a detailed example. Suppose we construct a van where the hourly wage is $12, the mileage bonus is 10 pts per mile, and the time bonus is 5 pts per minute. The following actions take place.

Action	Wages	Tips	Mileage bonus pts	Time bonus pts
Drive 8 miles in 10 minutes (no riders)	$2		80 (8 miles x 10 pts)	0
Wait around 5 minutes (no riders)	$1		0	0
Pick up 1 rider				
Drive 25 miles in 30 minutes (1 rider)	$6		250 (25 miles x 10 pts)	150 (30 x 5 pts)
Pick up 1 rider				
Pick up 1 rider				
Wait around 15 minutes (3 riders)	$3		0	75 (15 x 5 pts)
Drive 20 miles in 1 hour (3 riders)	$12		600 (3 x 20 miles * 10)	300 (60 x 5 pts)
Drop off 1 rider, leaves $2 tip		$2		
Totals	$24	$2	930 pts	525 pts

The driver has $24 in wages, $2 in tips, and 1455 bonus points. The total pay the driver earns depends on the value of each bonus point, which we presume could vary depending on things like experience and seniority. In this case, if we suppose $0.02 per bonus point, the driver's total pay is $24 + $2 + 1455 * 0.02 = $55.10. Since a total of two hours have passed, the average hourly pay is 55.10 / 2.0 or $27.55.

Please note that you should not use any conditional statements (i.e. "if" statements, ternaries, or loops), or anything else we haven't covered, for this assignment. There will be several 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 be penalized slightly for using conditional statements, because you are just being lazy and making your code longer and more complicated.*)

Specification

The specification for this assignment includes this pdf along with any "official" clarifications announced on Canvas.

There is one public constructor:

```
public AirportVan(double givenHourlyRate, int givenMileageBonus,
                  int givenTimeBonus, int givenMaxRiders)
```

Constructs a new **`AirportVan`**. Initially there are no riders and the driver's wages, tips, and bonus points are zero.

There are the following public methods, listed in alphabetical order:

```
public void drive(int miles, int minutes)
```

Simulates driving the van for the given number of miles over the given number of minutes

```
public void dropOff(double tip)
```

Reduces the number of riders by 1 (not going below zero) and collects the given tip. This method does nothing if the number of riders is already zero. The tip amount is given in dollars.

```
public double getAverageHourlyPay(double dollarsPerPoint)
```

Returns the average hourly pay for the driver (total pay / time) since this van was constructed. "Total pay" includes wages, tips, and bonus points.

```
public int getRiderCount()
```

Returns the number of riders currently in this van.

```
public int getTotalMiles()
```

Returns the total miles driven since this van was constructed.

```
public double getTotalPay(double dollarsPerPoint)
```

Returns the total pay earned by the driver since this van was constructed, including wages, tips, and bonus points, using the given multiplier to convert bonus points to dollars.

```
public int getTotalPoints()
```

Returns the total bonus points earned by the driver since this van was constructed.

```
public int getTotalTime()
```

Returns the total time spent by the driver since this van was constructed, in minutes.

```
public double getTotalTips()
```

Returns the total tips earned by the driver since this van was constructed, in dollars.

`public void pickUp()`

Increases the number of riders by 1; however, this method does nothing if the number of riders is already at the maximum for this van.

`public void waitAround(int minutes)`

Simulates the passage of time without the van moving. Equivalent to calling **`drive(0, minutes)`**.

Where's the main() method??

There isn't one! Like most Java classes, 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 class, you can write a test class with a main method like the examples below in the "getting started" section.

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 below.

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.*

Tip: the sample test cases shown below can be found in the file SimpleTests.java, posted on Canvas. Please keep your test code in the default package, not in the **`hw1`** *package.*

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

1. Create a new, empty project and then add a package called **`hw1`**. ***Be sure NOT to create module-info.java***.

2. Create the **`AirportVan`** class in the **`hw1`** package and put in stubs for all the required methods and the constructor. 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. ***DO NOT COPY AND PASTE directly from this pdf (or online javadoc, if any)!*** *This leads to insidious bugs caused by invisible characters that are sometimes generated by sophisticated document formats.)*

3. Briefly 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. The class javadoc should include an **@author** tag with your name. (Repeating phrases from the method descriptions here is perfectly acceptable; however, we strongly advise that you ***DO NOT COPY AND PASTE directly from this pdf or online Javadoc!*** *This leads to insidious bugs caused by invisible characters that are sometimes generated by sophisticated document formats.)*

4. Look at each method. Mentally classify it as either an *accessor* (returns some information without modifying the object) or a *mutator* (modifies the object, usually returning **`void`**). The accessors will give you a lot of hints about what instance variables you need.

5. It seems like the tricky parts involve calculating the bonus points, so let's ignore those to start. Take something simpler like keeping track of the total miles. There is an accessor method **`getTotalMiles()`** that returns the total miles driven, and the way that the miles can change is by calling **`drive()`**. So we could start with a simple test case like this.

```
AirportVan v = new AirportVan(0, 0, 0, 0);
v.drive(20, 0);
System.out.println(v.getTotalMiles());
System.out.println("Expected 20");
```

(We can pass in zeros for the constructor parameters, since they should have no effect on the accumulation of miles.) It is not hard to deduce that in order for **`getTotalMiles()`** to return the correct value, that value probably has to be stored as an instance variable.

6. The same reasoning should apply to **`getTotalTime()`**, right? Try extending the test:

```
v = new AirportVan(0, 0, 0, 0);
v.drive(20, 60);
v.drive(10, 15);
System.out.println(v.getTotalMiles());
System.out.println("Expected 30");
System.out.println(v.getTotalTime());
System.out.println("Expected 75");
```

and add the code to keep track of the time as well.

7. Another relatively easy bit to try next is to keep track of the number of riders. Again, there is an accessor method **`getRiderCount()`**, but in this case the number of riders isn't modified by

the **`drive()`** method, it's modified by **`pickUp()`** and **`dropOff()`**. Try writing a simple test like this:

```
// constructs a van with max of 3 riders
v = new AirportVan(0, 0, 0, 3);
v.pickUp();
v.pickUp();
System.out.println(v.getRiderCount());
System.out.println("Expected 2");
v.pickUp();
System.out.println(v.getRiderCount());
System.out.println("Expected 3");
```

Here we are constructing a van with a max of 3 riders, because that gives us a chance to test the requirement that **`pickUp()`** will never add a rider if the van already has the maximum number of riders. The above test could be continued like this:

```
v.pickUp(); // does nothing, can't go over max riders
System.out.println(v.getRiderCount());
System.out.println("Expected 3");
```

How to implement it? First, you need to know the max number of riders. This is given as a constructor parameter, but in order to refer to that value later in the code, it needs to be saved in an instance variable. That is a common action in a constructor: save the parameter values so they can be used later. Look at the way we saved the "givenDiameter" parameter in the Basketball class from lab 2.

Second, if the addition of a new rider would push the number of riders over the max, how can you check? Basically, you want to take two numbers: the current number of riders plus 1, and the max number of riders, and take the smaller of the two. This is an ideal application of the method **`Math.min()`**.

8. Then, try dropping them off and make sure the number goes down, but not below zero. For now we can use a tip value of zero for the argument of **`dropOff()`**. (Continuing the example above):

```
v.dropOff(0);
System.out.println(v.getRiderCount());
System.out.println("Expected 2");
v.dropOff(0);
System.out.println(v.getRiderCount());
System.out.println("Expected 1");
v.dropOff(0);
System.out.println(v.getRiderCount());
```

```
System.out.println("Expected 0");
v.dropOff(0); // does nothing, can't go below 0
System.out.println(v.getRiderCount());
System.out.println("Expected 0");
```

9. The calculation of bonus points is unavoidable in order to proceed further. To keep things simple, let's try to look separately at the bonus points for time vs the bonus points for mileage. How about starting with time? Here, we need some nonzero constructor parameters for the bonus values. Since no bonus points for time are earned when there are no riders, let's add one passenger:

```
// bonus points for waiting when there is 1 rider
// try with 5 points per minute
v = new AirportVan(0, 0, 5, 3);
v.pickUp();
v.drive(0, 100);
System.out.println(v.getTotalPoints());
System.out.println("Expected 500");
v.drive(0, 50);
System.out.println(v.getTotalPoints());
System.out.println("Expected 750");
```

As with the max riders, the time bonus, which is given as a constructor parameter will need to be saved as an instance variable in order to get this to work.

This is a good place to offer a reminder: *accessor methods should never modify instance variables*, which tells us that the bonus points need to be updated in the mutator method **`drive()`**, not in **`getTotalPoints()`**.

10. What about if there are multiple riders, or no riders? As always, write a simple test first:

```
// bonus points for waiting when there are three riders
v = new AirportVan(0, 0, 5, 3);
v.pickUp();
v.pickUp();
v.pickUp();
v.drive(0, 100);
System.out.println(v.getTotalPoints());
System.out.println("Expected 500");
v.drive(0, 50);
System.out.println(v.getTotalPoints());
System.out.println("Expected 750");
```

```
// bonus points for waiting when there are no riders
v = new AirportVan(0, 0, 5, 3);
v.drive(0, 100);
System.out.println(v.getTotalPoints());
System.out.println("Expected 0");
v.drive(0, 50);
System.out.println(v.getTotalPoints());
System.out.println("Expected 0");
```

The potentially tricky bit is, how to you make the result the same when there is one rider vs when there are multiple riders? Not hard. You just need to multiply by the *smaller* of two numbers:

- The actual number of riders (possibly zero), or
- 1

11. We should be able to do something similar with the bonus points for miles, except that when there are multiple riders, we multiply the miles by the number of riders, and when there are no riders, we multiply by 1. Try a few test cases similar to above.

```
// bonus points for miles with one rider,
// using 10 bonus points per mile
v = new AirportVan(0, 10, 5, 3);
v.pickUp();
v.drive(2, 0);
System.out.println(v.getTotalPoints());
System.out.println("Expected 20"); // 2 * 10
v.drive(5, 0);
System.out.println(v.getTotalPoints());
System.out.println("Expected 70"); // above plus 5 * 10

// bonus points for miles with three riders
v = new AirportVan(0, 10, 5, 3);
v.pickUp();
v.pickUp();
v.pickUp();
v.drive(2, 0);
System.out.println(v.getTotalPoints());
System.out.println("Expected 60"); // 3 * 2 * 10
v.drive(5, 0);
System.out.println(v.getTotalPoints());
System.out.println("Expected 210"); // above + 3 * 5 * 10

// bonus points for miles with no riders
v = new AirportVan(0, 10, 5, 3);
v.drive(2, 0);
System.out.println(v.getTotalPoints());
System.out.println("Expected 20"); // same is for one rider
v.drive(5, 0);
```

```
System.out.println(v.getTotalPoints());
System.out.println("Expected 70");
```

How to get that last one to work? This is a problem of finding the *larger* of two numbers:

- The actual number of riders, or
- 1

12. How about tips? Start with a simple case:

```
// tips
v = new AirportVan(0, 10, 5, 3);
v.pickUp();
v.pickUp();
v.dropOff(5);
System.out.println(v.getTotalTips());
System.out.println("Expected 5");
v.dropOff(2);
System.out.println(v.getTotalTips());
System.out.println("Expected 7");
```

But what should happen if **`dropOff()`** is invoked when there are no riders? The spec says that it should have no effect, and that includes *not affecting the total tips*. Continuing the example above,

```
// tricky bit - there isn't a rider, so dropOff should do nothing
v.dropOff(10);
System.out.println(v.getTotalTips());
System.out.println("Expected 7");
```

By now, you can probably figure out how to do this: in calculating the time bonus, we came up with an expression (involving **`Math.min()`**) that has value 0 if there are no passengers, and value 1 if there is at least one passenger. Use that to multiply the tip amount before you add it!

13. So far we've ignored the method **`waitAround()`**. If you think about it, this should all the same things as **`drive()`**, except that the mileage is always zero. So you can easily get the effect of **`waitAround(x)`** by invoking **`drive(0, x)`**.

14. Last but not least, you need to implement the methods that return the driver's total pay and average pay per hour. You could use the example on page 3 to construct a test case. An interesting question is: do you need another instance variable to store the driver's wages, or can you calculate that value from data you already have? Remember, we want to avoid keeping redundant information in the instance variables.

The SpecChecker

You can find the SpecChecker on Canvas. Import and run the SpecChecker just as you practiced in Lab 1. 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", link #12 on our Canvas front page, if you are not sure what to do.

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 conditional statement when you could just be using **`Math.min`** or modular arithmetic? 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, in addition to the two specific bullet points above.

- **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 repeating the descriptions from the documentation. *However:* ***do***

not literally copy and paste from this pdf*! This leads to all kinds of weird bugs due to the potential for sophisticated document formats like Word and pdf to contain invisible characters.*

- 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 `hw1`. 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 `hw1`. *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 announcements from the instructors on Canvas 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 Canvas. (We promise that no official clarifications will be posted within 24 hours of the due date.)

What to turn in

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.

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, **`AirportVan.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", link #11 on our Canvas front page.

> 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 file **AirportVan.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. The Assignment Submission HOWTO includes detailed instructions for creating a zip file if you need them.