Com S 227: Summer 2020
Homework 1
100 points

Due Date: Friday, June 5, 11:59 pm (midnight)
Early deadline (5% bonus): Thursday, June 4, 11:59 pm
Late deadline (10% penalty): Saturday, June 6, 11:59 pm

General information

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* on the Homework page on Canvas. Please do this right away.

If you need help, ask your instructor or the TAs. Help is also available through Piazza discussions.

Please start the assignment as soon as possible and get your questions answered right away!

Introduction

In this homework you will have the opportunity to create a system involving several interacting objects. In addition, you'll get some practice using conditional statements. The system consists of six classes:

```
ExitMachine.java
ParkingRateUtil.java
PaymentMachine.java
SimpleClock.java
Ticket.java
TicketDispenser.java
```

You will be implementing *four* of the six classes: `SimpleClock` and `Ticket` are already implemented for you and should not be modified.

The system is a model of the payment system for the ISU Memorial Union parking garage. Here is the scenario:

> When you enter the garage, you *take* a **ticket** from a **ticket dispenser**. The ticket includes a magnetic stripe that can store a small amount of information. The dispenser *sets* the ticket with a timestamp based on its internal **clock**. When you are ready to leave the MU, you stop at a **payment machine** and *insert* your ticket. The machine *displays* a payment amount that depends on how long you have been parked, based on the machine's internal **clock** and the start time

recorded on the ticket. The amount due is calculated using a **rate calculator** that contains an algorithm based on current parking rates. You can *observe* the amount due when looking at the machine. You *make the payment* and then *eject* the ticket from the machine. The ticket is now modified to have one more piece of information: a second timestamp indicating when you made the payment. You now have 15 minutes to get to your car and exit the garage. When you leave the garage, you *insert* the ticket into an **exit machine**. If the exit machine determines (using its internal **clock**) that it has been 15 minutes or less from the time you made payment, then the gate goes up and you exit the garage. Otherwise, you have to go back to the payment machine and pay some more.

In describing the scenario, the primary nouns in the description (in **bold**) become the objects in our design, and the actions on those objects (in *italics*) become methods. The section below specifies the types and methods in detail.

Detailed specification

The class `SimpleClock`

The `SimpleClock` class is fully implemented and you should not modify it. A `SimpleClock` is just a counter used to simulate the passage of time. For our purposes, the time is just an integer, which we think of as the number of minutes that have passed since the initialization of the system. The passage of time is simulated by calling the method `timePasses()`. In a complete system, the three participants that need access to time, namely, the `TicketDispenser`, the `PaymentMachine`, and the `ExitMachine`, will all share the same `SimpleClock` object. Read the code for details; it is extremely simple.

The class `Ticket`

The `Ticket` class is fully implemented and you should not modify it. A `Ticket` is a simple data container that can store two pieces of information: An int representing the time at which it was created, and an int representing the time at which payment was made. Read the code for details; it is extremely simple.

The class `TicketDispenser`

There is one constructor:

`public TicketDispenser(SimpleClock givenClock)`
: Constructs a `TicketDispenser` that uses the given clock.

There is one public method:

`public Ticket takeTicket()`

Constructs and returns a new **`Ticket`** object. The constructed ticket will have a start time based on the current value of the ticket dispenser's clock and a payment time of zero.

The class `ExitMachine`

There is one constructor:

`public ExitMachine(SimpleClock givenClock)`

Constructs an **`ExitMachine`** that uses the given clock and has an initial count of zero.

There are two public methods:

`public boolean insertTicket(Ticket t)`

Simulates inserting a ticket into this machine. If the ticket's payment time is within **`ParkingRateUtil.EXIT_TIME_LIMIT`** minutes of this machine's clock time (*and* is greater than zero), the method returns true. Otherwise the method returns false. The **`Ticket`** object is not modified. If the method returns true, this machine's exit count is incremented. (Note that this method is a mutator method that also returns a value.)

`public int getExitCount()`

Returns a count of the total number of successful exits. A "successful exit" is defined to be a call to **`insertTicket()`** that returns true.

The class `ParkingRateUtil`

This is a "utility" class, that is, it has no instance variables, is never instantiated, and serves only as a container for one or more static methods. (Similar to the class **`Math`**.) Note: in implementing the parking rates (see the section "Current parking rates", below) there are many literal numeric values to deal with. **It's ok to hard-code these literal values.** Your **`ExitMachine`** should use the constant **`EXIT_TIME_LIMIT`**.

There is one constructor, which is declared **`private`** since the class should never be instantiated:

```
private ParkingRateUtil()
```

There is one public constant:

```
public static final int EXIT_TIME_LIMIT = 15;
```

There is one public method:

`public static double calculateCost(int minutes)`

Returns the cost of parking for the given total number of minutes, based on the current rates for the MU garage. See the section "Current parking rates", below, for details.

The class `PaymentMachine`

A **`PaymentMachine`** has methods allowing a **`Ticket`** object to be updated to show when payment is made. There is a method **`insertTicket()`** to simulate inserting a ticket into the machine. At that point a transaction is said to be "in progress", and the **`inProgress()`** method returns true, until a subsequent call to **`ejectTicket()`**. The amount due for parking can be obtained from the method **`getPaymentDue()`**. The method **`makePayment()`** finally updates the **`Ticket`** object to record the time of payment. (We assume that all payments are by credit/debit and are successful.) In addition, the **`PaymentMachine`** includes an accumulator that records the total amount of money paid into the machine since it was initialized.

There is one constructor:

`public PaymentMachine(SimpleClock givenClock)`

Constructs a **`PaymentMachine`** that uses the given clock. Initially, total payments are 0.0.

There are seven public methods:

`public void insertTicket(Ticket t)`

Simulates inserting the given ticket into this machine. This method does not modify the **`Ticket`** object or perform any calculation with it. After calling this method, the **`inProgress()`** method returns true until a subsequent call to **`ejectTicket()`**. Calling **`insertTicket()`** while a transaction is in progress has no effect.

`public Ticket getCurrentTicket()`

Returns a reference to the ticket currently in this machine, or **`null`** if no transaction is in progress.

`public boolean inProgress()`

Returns true if there is currently a ticket in this machine, false otherwise.

`public double getPaymentDue()`

Returns the payment due for the ticket currently in the machine. If no transaction is in progress, returns 0.0. This method does not modify the **`Ticket`** object or update this machine's total payments. The payment due is based the *current time* (according to this machine's clock) and on the *start time* and *payment time* for the current ticket. The basic amount due is the result of calling **`ParkingRateUtil.calculateCost()`** for the difference *current time - start time*.

However, if the *payment time* is nonzero (indicating that some payment was already made), the cost of parking from *start time to payment time* (i.e., the amount that must have already been paid) is subtracted from the amount due.

`public void makePayment()`

Updates the current ticket with the payment time and adds the payment amount to this machine's total. If there is no transaction in progress, this method has no effect.

`public void ejectTicket()`

Simulates ejecting a ticket from this machine, after which another ticket can be inserted. This method does not modify the current ticket object or update this machine's total payments. If there is no transaction in progress, this method has no effect.

`public double getTotalPayments()`

Returns the total payments that have been made at this machine.

Current parking rates

These are taken from http://www.mu.iastate.edu/parking--maps/parking/ (screenshot below)

The first step in implementing an algorithm like this is to work out some concrete examples by hand. For instance:

- how much is it for 10 minutes?
- how much is is for 30 minutes?
- how much is it for 31 minutes (charged as 1 hour)?
- how much is it for 250 minutes (charged as 5 hours)?

- how much is it for 1000 minutes (charged as one day)?
- how much is it for 3000 minutes (charged as two days and 120 minutes)?

Testing and the SpecCheckers

As always, you should try to work incrementally and write simple tests for your code as you develop it.

Since test code that you write is not a required part of this assignment and does not need to be turned in, **you are welcome to post your test code on Piazza for others to check, use and discuss.**

SpecChecker

Your class must conform precisely to this specification. The most basic part of the specification includes the package and class names the public method names and return types, and the types of the parameters. We will provide you with a specchecker to verify that your class satisfies all these aspects of the specification and does not attempt to add any public attributes or methods to those specified. If your class structure conforms to the spec, you should see a message such as "7 out of 7 tests pass" in the console output. The specchecker will also run some simple functional tests for you. This is like the specchecker you used in Lab 1. It will also offer to create a zip file for you to submit.

More about grading

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. 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? 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 lots of unnecessary instance variables
  - All instance variables should be **`private`**.
- **Accessor methods should not modify instance variables**.

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 and complete Javadoc comment. Class javadoc must include the **`@author`** tag, and method javadoc must include **`@param`** and **`@return`** tags as appropriate.
  - Try to state what each method does in your own words, but there is no rule against copying and pasting the descriptions from this document.
  - 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.
- For this assignment it is ok to embed numeric literals in the ParkingRateUtil class.
- 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 a comment explaining how it works.)
  - Internal comments always *precede* the code they describe and are indented to the same level.
- 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, and **for this assignment you are welcome to post and discuss test code**. (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 Webex timeslot 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.)

Suggestions for getting started

Remember to work incrementally and test your code as you go!

1. Import the given project into Eclipse. The simplest way to do this is download it to some convenient location that is *not* within the workspace directory, then right click on the workspace and import.

2. Create the remaining four classes. Put in stubs for the required constructors and methods.

3. If you haven't already done so, read the code for **`Ticket`** and **`Clock`**. It is very simple, and you will need to use both these classes immediately in order to implement anything else. Try a few simple usage examples in a main method of your own, for example,

```
SimpleClock c = new SimpleClock();
System.out.println(c.getTime());         // Expected 0
c.timePasses(10);
System.out.println(c.getTime());         // Expected 10
Ticket t = new Ticket(42);
System.out.println(t.getStartTime());    // Expected 42
System.out.println(t.getPaymentTime()); // Expected 0
```

4. You can work on the four other classes independently of each other and test them separately. (In order to fully complete the **`getPaymentDue`** method of **`PaymentMachine`**, you'll need to have the **`ParkingRateUtil`** class implemented, but that is the only dependency.)

5. The **`ExitMachine`** class is pretty simple and might be a good warm-up. Start by writing the javadoc for it. Then write a simple test case. For example:

```
SimpleClock c = new SimpleClock();
ExitMachine em = new ExitMachine(c);
Ticket t = new Ticket(c.getTime());
t.setPaymentTime(10);
c.timePasses(20);
boolean canExit = em.insertTicket(t);
System.out.println(canExit);              // expected true
Ticket t2 = new Ticket(0);
t2.setPaymentTime(30);
c.timePasses(60);
canExit = em.insertTicket(t2);
System.out.println(canExit);            // expected false
System.out.println(em.getExitCount()); // expected 1
```

6. You can start implementing the **`ParkingRateUtil`** class anytime and begin testing it. The section "Current parking rates" above has some ideas for test cases to try. Do the calculations by hand first, and then you can just write your code to perform the same steps. Notice that you can continue with the other two classes even if you haven't finished this one.

7. The **`TicketDispenser`** does not do very much: The **`takeTicket()`** method just constructs a new **`Ticket`** object and returns it. For example,

```
SimpleClock c = new SimpleClock();
TicketDispenser td = new TicketDispenser(c);
c.timePasses(10);
Ticket t = td.takeTicket();
System.out.println(t.getStartTime());   // Expected 10
System.out.println(t.getPaymentTime()); // Expected 0
```

8. For **`PaymentMachine`**, start with the methods **`insertTicket`**, **`ejectTicket`**, **`getCurrentTicket`**, and **`inProgress`**. These methods are extremely simple and only require one instance variable (think about **`getCurrentTicket`**). Write the javadoc first. Then write a few usage examples, for instance:

```
SimpleClock c = new SimpleClock();
PaymentMachine pm = new PaymentMachine(c);
Ticket t = new Ticket(0);
pm.insertTicket(t);
System.out.println(pm.inProgress());   // expected true
Ticket current = pm.getCurrentTicket();
System.out.println(current == t);      // expected true
pm.ejectTicket();
System.out.println(pm.inProgress());   // expected false
current = pm.getCurrentTicket();
System.out.println(current == null);   // expected true
```

9. For **`getPaymentDue`**, you'll need to have **`ParkingRateUtil`** at least partially working. Carefully write the javadoc for the method first, and then think about some test cases. Remember that this method does not modify the ticket or the **`PaymentMachine`**. For example,

```
SimpleClock c = new SimpleClock();
c.timePasses(10);
PaymentMachine pm = new PaymentMachine(c);
Ticket t = new Ticket(c.getTime());
c.timePasses(60);
pm.insertTicket(t);
System.out.println(pm.getPaymentDue());  // expected 2.00
pm.ejectTicket(); // eject without paying
c.timePasses(75);
pm.insertTicket(t);
```

```
System.out.println(pm.getPaymentDue());  // expected 5.00
```

10. When you have **`getPaymentDue`** working, you can implement **`makePayment`**. This is the method that updates the ticket and updates the total payments in the machine. How would you test it? Again, begin with some simple test cases in a main method of your own.

11. One time I went to the MU workspace and parked in the garage, I was there for about 90 minutes. I took my ticket to the payment machine and paid for 90 minutes parking. Then realized that I left something at the workspace, so I went back there and then I ran into a friend in the stairwell on the way out. Eventually, I had spent 45 minutes more, so, I had to go back into the MU and pay for more parking. What was the amount of the second payment? Have you accounted for this in your **`getPaymentDue`** method? Would it work correctly if I was delayed in the stairwell a second time?

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 second SpecChecker. The file will be named **`SUBMIT_THIS_hw1.zip`**. and it will be located in the directory you selected when you ran the SpecChecker. It should contain one directory, **`hw1`**, which in turn contains six files, four that you created and two that were provided for you:

```
ExitMachine.java
ParkingRateUtil.java
PaymentMachine.java
SimpleClock.java
Ticket.java
TicketDispenser.java
```

Submit the zip file to Canvas using the Assignment 1 submission link and **verify that your submission was successful by checking your submission history page**. 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 six files listed above. Make sure all files have the extension **`.java`**, NOT **`.class`**. You can accomplish this easily by zipping up the **src** directory of your 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.