Com S 227
Fall 2015
Assignment 1
250 points

Due Date: Tuesday, September 29, 11:59 pm (midnight)

"Late" deadline (25% penalty): Wednesday, September 30, **8:00 pm**

*(**Note late deadline**. A sample solution will be posted by 9:00 pm on Wednesday. Remember that Exam 1 is Thursday, October 1.)*

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 acknowledgement* on the Homework page on Blackboard. 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.

*Note: Our first exam is Thursday, October 1, which is just two days after the due date for this assignment. **It will not be possible to have the assignments graded and returned to you prior to the exam**. The exam dates are arranged by the registrar, not the instructors. We can post a sample solution on September 30, but we strongly encourage you to test your code carefully as described in the section below, "Testing and the SpecCheckers".*

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

Introduction

In this homework you will get some practice designing classes and using conditional statements. This homework requires you to write four classes: `Ticket`, `TicketMachine`, `TicketUtil`, and `Turnstile`. All four of these classes should be in a package named `hw1`.

Overview

For this assignment you'll create some classes for modeling tickets for a city transit system. The type of system we are modeling has these main features:

- Stations are grouped in *zones*. Each zone is represented by a positive integer. The cost for a ride depends on how many zones are crossed. There is a fixed cost, referred to as the *ride cost*, to travel within a zone. In addition, the rider pays an additional cost, called the *zone cost*, which is multiplied by the difference between the zones. For example, if the ride cost is 2.00 and the zone cost is 1.75, then traveling from zone 5 to zone 2 would be 2.00 + 3 * 1.75 = 7.25, since 3 is the difference between 5 and 2. It would be the same to travel from zone 2 to zone 5, of course.

- The tickets have a magnetic stripe in which some information is encoded, including a *balance*. As riders pass through the turnstile to enter the station, they swipe their cards through the card reader on the turnstile. The zone in which the turnstile is located is recorded on the card. When a rider exits the station at his or her destination, the card is swiped at the exit turnstile, and the cost for the trip is subtracted from the balance on the card, if possible. When the balance on the card isn't enough to pay for the trip, the turnstile doesn't open and the card's balance isn't modified.

- There is a discounted fare for seniors, a fact that is also recorded on the ticket.

Here is an overview of the roles of the four classes in this system.

A **`Ticket`** models a "smart ticket" with a magnetic stripe as described above. A ticket has a balance in cents, an indication of whether the user is entitled to discounted fares, and what the start zone is, if a trip is in progress.

A **`TicketMachine`** creates new **`Tickets`** according to various criteria. Users can purchase tickets by selecting a start and end zone, by specifying an amount for the initial balance, or by specifying a number of rides with a given start and end zone. A ticket machine also keeps track of the total number of tickets sold and the total amount of money collected.

The **`Turnstile`** type models a turnstile as described above. Each turnstile has a zone in which it is located and has methods **`swipeIn()`** for entering the station and **`swipeOut()`** for exiting.

The `TicketUtil` class is a utility for calculating ride costs. It is used by the `TicketMachine` when dispensing tickets, and it is used by the `Turnstile` class in order to determine how much to subtract from the balance on a ticket when someone exits.

Note that there is no main() method in any of these classes. In order to test your code, you'll need to create a main class that constructs some of these objects and checks that they work correctly, but your test code does not have to be turned in.

Specification for Ticket

There is a public constructor `Ticket(int value, boolean discounted)`, where `value` is the initial balance on the ticket, and `discounted` indicates whether discounted fares should be charged.

The Ticket class has public methods as follows:

`void beginTrip(int zone)`

Sets the start zone for this ticket. Caller is responsible for ensuring that the argument is a valid zone number. After this method is called, subsequent calls to `isInTransit()` should return true.

`int getStartZone()`

Returns the start zone for this ticket. If this method is called before beginTrip (i.e., when `isInTransit()` is false) this method must return `TicketUtil.INVALID_ZONE`.

`boolean isDiscounted()`

Returns true if this ticket will be charged discounted fares.

`int getBalance()`

Returns the balance on this ticket, in cents.

`boolean charge(int rideCost)`

Reduces the balance on this ticket by the given amount, if possible. If the balance is less than the ride cost, this method returns false without modifying the ticket balance; otherwise reduces the balance by the given amount and returns true. Note that whenever this method is successful, subsequent calls to `isInTransit()` should return false and subsequent calls to `getStartZone()` should return `TicketUtil.INVALID_ZONE`.

`boolean isInTransit()`

Returns true if the ticket has been successfully swiped in but not yet successfully swiped out.

Specification for Turnstile

There is a public constructor **`Turnstile(int zone)`**, where **`zone`** represents the zone for the station in which the turnstile is located.

The **`Turnstile`** class has public methods as follows:

`boolean swipeIn(Ticket ticket)`

Records this turnstile on the given ticket as the start zone. If successful, returns true (i.e., the turnstile opens). However, there are some additional restrictions. If the balance on the ticket is not enough for *any* ride (according to **`TicketUtil.getMinimumFare()`**), then false is returned and the ticket is not modified. If the status of the ticket is already "in transit" (the rider swiped in at some point and then never swiped out) then the turnstile *first* attempts to charge the ticket for a ride to this turnstile's zone, based on the start zone recorded on the ticket. If that is unsuccessful, false is returned without modifying the ticket. If successful, action proceeds as above.

> (Example: Assume the cost of a 2-zone ride is 5.50 and the minimum fare is 2.00. Then suppose a ticket's status is "in transit" with a start zone of 5 and the balance on the ticket is 6.00. If the rider attempts to swipe it in at a turnstile in zone 3, the turnstile first charges the ticket 5.50 for the cost of a ride from zone 5 to zone 3 and records that the ticket is no longer in transit; it then checks that the remaining 0.50 is not enough for the minimum fare, and so it leaves 0.50 on the ticket and returns false.)

`boolean swipeOut(Ticket ticket)`

Attempts to charge the ticket it for a ride from the ticket's start zone to this turnstile's location. If the balance is not enough for the cost of the ride, returns false without modifying the ticket. If the status of the ticket is not "in transit", returns false without modifying the ticket.

`int getEntranceCount()`

Returns the number of times a ticket has successfully swiped in at this turnstile.

`int getExitCount()`

Returns the number of times a ticket has successfully swiped out at this turnstile, including successful charges made during execution of **`swipeIn()`**.

Specification for TicketMachine

There is one public constructor with no arguments.

The **`TicketMachine`** class has public methods as follows:

`Ticket purchaseTicket(int startZone, int endZone, boolean discounted)`

Returns a new ticket with exactly the balance needed for a ride between **`startZone`** and **`endZone`**, using a discounted fare if **`discounted`** is true. The discounted status is stored in the ticket.

`Ticket purchaseTicket(int numRides, int startZone, int endZone, boolean discounted)`

Returns a new ticket with exactly the balance needed for a **`numRides`** rides between **`startZone`** and **`endZone`**, using a discounted fare if **`discounted`** is true. The discounted status is stored in the ticket.

`Ticket purchaseTicket(int amount, boolean discounted)`

Returns a new ticket whose balance is **`amount`**. The discounted status is stored in the ticket.

`int totalTickets()`

Returns the total number of tickets sold from this machine.

`int totalCost()`

Returns the total cost for all tickets sold from this machine.

Specification for TicketUtil

This is a "utility" class whose methods are all static. Static methods can be used without constructing an instance of the class. They are called by putting the class name before the dot, like this:

```
double cost = TicketUtil.calculateRideCost(2, 5, false);
```

There is one **`private`** constructor with no arguments. By making the constructor **`private`**, we ensure that no one will attempt to create an instance of **`TicketUtil`**, which would be a programming error.

The **`TicketUtil`** defines five public constants.

```
/**
 * Fare (in cents) for a trip within a zone.
 */
public static final int RIDE_COST = 200;

/**
 * Discounted fare (in cents) for a trip within a zone.
 */
public static final int RIDE_COST_DISCOUNTED = 150;

/**
 * Additional fare (in cents) for travel between zones.
 */
public static final int ZONE_COST = 175;

/**
 * Additional discounted fare (in cents) for travel between zones.
 */
public static final int ZONE_COST_DISCOUNTED = 120;

/**
 * Constant representing a nonexistent start zone.
 */
public static final int INVALID_ZONE = -1;
```

The TicketUtil class has public methods as follows:

```
static int calculateRideCost(int startZone, int endZone,
                             boolean discounted)
```

Returns the cost for for a ride between **`startZone`** and **`endZone`**, using a discounted fare if **`discounted`** is true.

```
static int calculateRideCost(int numRides, int startZone, int endZone,
                             boolean discounted)
```

Returns the cost for for **`numRides`** rides between **`startZone`** and **`endZone`**, using a discounted fare if **`discounted`** is true.

```
static int getMinimumFare(boolean discounted)
```

Returns the cost of a ride within a zone, using a discounted fare if **`discounted`** is true.

Suggestions for getting started

1. Create a new Eclipse project and within it create a package **`hw1`**.

2. Go ahead and create the four required classes in the **`hw1`** package. Put in stubs for all the methods and constructors described above. (At this point there should be no compile errors in the project)

3. Define the five public constants in **`TicketUtil`**.

4. Run the specchecker. It should print "5 out of 5 tests pass." This will verify that you have the package, class names, method names, and parameter types correct. It will save you a lot of time if you get all this right at the beginning.

5. Javadoc the classes and methods. This is a required part of the assignment, and doing it now will help clarify for you what each method is supposed to do before you begin the actual implementation.

6. You could start with the static methods in the **`TicketUtil`** class. This is pretty easy since there are no instance variables to worry about. Create a test class with a main() method and make up some test cases, for example:

   ```
   System.out.println(TicketUtil.calculateRideCost(5, 2, false));
   System.out.println("Expected: 725);
   ```

7. Think about the instance variables needed for the **`Ticket`** class. For example, what information do you have to store in order to correctly implement the accessor methods **`getBalance()`**, **`isDiscounted()`**, and **`getStartZone()`**? For a newly created **`Ticket`**, what should these values be? Write some test cases, for example:

```
Ticket t = new Ticket(100, false);
System.out.println(t.getBalance());
System.out.println("Expected 100");
System.out.println(t.isDiscounted());
System.out.println("Expected false");
System.out.println(t.getStartZone());
System.out.println("Expected " + TicketUtil.INVALID_ZONE);
```

8. Once you have `getBalance()` and `isDiscounted()` working correctly, you can add some test cases to verify that `TicketMachine` is actually creating the right tickets.

9. Implement the `beginTrip()`, `isInTransit()`, and `charge()` methods for `Ticket`. Write some test cases to check that all the logic is right (e.g., after you call `beginTrip()`, `isInTransit()` should return true).

10. Next you might implement the `TicketMachine` class. Note that most of the "work" of this class is actually done by calls to `TicketUtil`, but a `TicketMachine` is also responsible for keeping track of the total number and cost of the tickets it sells. Define some instance variables for doing so. Add some test cases to your test to see whether the tickets are being counted correctly and created correctly

11. Once you are confident that your `Ticket` methods are correct, think about the `Turnstile` class. To start with, how would you test it? Here are a couple of usage examples:

```
Turnstile zone2 = new Turnstile(2);
Turnstile zone5 = new Turnstile(5);
t = new Ticket(1000, false);
zone2.swipeIn(t);
System.out.println(zone2.getEntranceCount())
System.out.println("Expected 1");
System.out.println(t.isInTransit());
System.out.println("Expected true");
zone5.swipeOut(t);
System.out.println(t.getBalance());
System.out.println("Expected 275");
```

Testing and the SpecCheckers

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

Since the test code 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 1

Your class must conform precisely to this specification. The most basic part of the specification includes the class name and package, the required constants, 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 the following message in the console output:

```
5 out of 5 tests pass.
```

(This SpecChecker will not offer to create a zip file for you). Remember that your instance variables should always be declared **`private`**, and if you want to add any additional "helper" methods that are not specified, they must be declared **`private`** as well. *Specchecker 1 will be available on the 21st of September.*

SpecChecker 2

In addition, since this is the first assignment and we have not had a chance to discuss unit testing very much, we will also provide a specchecker that will run some simple functional tests for you. This is similar to the specchecker you used in Labs 1 and 2. It will also offer to create a zip file for you to submit. *Specchecker 2 should be available around the 24th of September. Please do not wait until that time to start testing!*

Please note that we are also going to read over your code and make sure that things are done sensibly. The automated tests will count for 20% to 30% of the total points. *You may lose points even if your code passes all the specchecker's functional tests*. Hopefully you will find the feedback useful in the long run, and we will attempt to assign some partial credit even if there are mistakes.

> See the document "SpecChecker HOWTO", which can be found in the Piazza pinned messages under "Syllabus, office hours, useful links" if don't remember how to import and run a SpecChecker.

Style and documentation

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

- Use instance variables only for the "permanent" state of the object, use local variables for temporary calculations within methods.
    - You will lose points for redundant instance variables
    - All instance variables should be **`private`**.

- **Accessor methods should not modify instance variables**.

- Your code should not be producing console output. You will likely add **`println`** statements when debugging, but you need to remove them before submitting the code.

- Use the defined constants (**`RIDE_COST`**, etc.). Don't embed numeric literals in your code.
- 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).

- 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. 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 many 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, 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 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

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

Please submit, on Blackboard, 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 four files:

```
Ticket.java
TicketMachine.java
TicketUtil.java
Turnstile.java
```

Please LOOK at the zip file you upload and make sure it is the right one!

Submit the zip file to Blackboard 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 four files listed above. Make sure the 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.