Com S 227
Fall 2020
Assignment 1
100 points

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

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

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

No submissions accepted after September 12, 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 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.

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 night 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 "Sample Usage" or "Getting started" sections. 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 incrementally developing a class from a specification and testing whether your implementation is correct. In particular, the "Sample Usage" and "Suggestions for getting started" sections provide enough detail to walk you through a process of developing the code in small steps and testing each step as you go.

None of the code should be very complicated; most methods are just one line, and the longest one, the `visit()` method, can be done in 6 to 8 lines. **In particular, you do not need any conditional statements (i.e. "if" statements)** or anything else we haven't covered. You won't be penalized if you use them, but you'll just be making things more complicated.

For this assignment you will implement two classes, called `Location` and `Backpacker`. The specification for this assignment includes this pdf along with any "official" clarifications posted on Piazza. Details are in the next two sections. Here is a brief overview:

A `Location` object just represents a place name along with a cost per night for lodging. There is a constructor for initializing the location name and lodging cost and there are several accessor methods. There are no mutator methods. The lodging cost is represented as an integer; the units are unspecified, but we can think of them as euros.

A `Backpacker` object models a student bumming around Europe, possibly with very little money. She has a certain amount of money and a current location, represented by a `Location` object. We assume that she has a rail pass so there is no cost to go from one place to another. (We also do not consider the need to *eat* in this model!) The basic operation is to visit a location for a given number of nights. The name of the location, and the number of nights spent there, are appended to a string referred to as the "journal". Each location has a certain lodging cost for staying there. If the backpacker doesn't have enough money for lodging, she has to sleep at the train station for some of those nights. The backpacker's funds can be replenished by calling home. However, the backpacker's parents are stuck at home working full time and may not have much sympathy for the backpacker running out of money while gallivanting around Europe like the free spirit that she is. The amount they actually send is proportional to the number of postcards the backpacker has sent to them since the last time she asked for money. Of course, sending a postcard costs money too! If things are so bad that the backpacker doesn't even have enough money to send a postcard home from her current location, we say that the backpacker is "SOL".

Specification for the `Location` class

There is one public constant:

`public static final double RELATIVE_COST_OF_POSTCARD = 0.05;`

The cost to mail a postcard from a location is this fraction times the location's lodging cost (rounded to the *nearest* integer).

There is one public constructor:

`public Location(String givenName, int givenLodgingCost)`

Constructs a new Location with the given name and lodging cost per night.

There are the following public methods:

`public String getName()`

Returns this location's name.

`public int lodgingCost()`

Returns this location's lodging cost per night.

`public int costToSendPostcard()`

Returns the cost to send a postcard from this location. The value is a percentage of the lodging cost specified by the constant RELATIVE_COST_OF_POSTCARD, rounded to the nearest integer.

`public int maxLengthOfStay(int funds)`

Returns the number of nights of lodging in this location that a backpacker could afford with the given amount of money.

`public int maxNumberOfPostcards(int funds)`

Returns the number of postcards that a backpacker could afford to send from this location with the given amount of money.

Specification for the `Backpacker` class

There is one public constant:

`public static final int SYMPATHY_FACTOR = 30;`

Proportionality constant when calling home for more money: the amount of money received is this constant times the number of postcards the backpacker has sent home (since the last time she called home for money).

There is one public constructor:

`public Backpacker(int initialFunds, Location initialLocation)`
Constructs a backpacker starting out with the given amount of money in the given location. The journal is initially a string consisting of "*locationname*(start)", where *locationname* is the name of the starting location.

There are the following public methods:

`public String getCurrentLocation()`
Returns the name of the backpacker's current location.

`public int getCurrentFunds()`
Returns the amount of money the backpacker currently has.

`public String getJournal()`
Returns the backpacker's journal. The journal is a string of comma-separated values of the form *locationname(number_of_nights)* containing the cities visited by the backpacker, in the order visited, along with the number of nights spent in each. The first value always has the form *locationname(start)* for the starting location. For example, if a backpacker starts in Paris, spends 5 nights in Rome, and then spends 2 nights in Prague, the journal string would be:
`Paris(start),Rome(5),Prague(2)`

`public boolean isSOL()`
Returns true if the backpacker does not have enough money to send a postcard from the current location.

`public int getTotalNightsInTrainStation()`
Returns the number of nights the backpacker has spent in a train station when visiting a location without enough money for lodging.

`public void visit(Location c, int numNights)`
Simulates a visit by this backpacker to the given location for the given number of nights. The backpacker's funds are reduced based on the number of nights of lodging purchased. When the funds are not sufficient for **`numNights`** nights of lodging in the location, the number of nights spent in the train station is updated accordingly. The journal is updated by appending a comma, the location name, and the number of nights in parentheses.

`public void sendPostcardsHome(int howMany)`
Sends the given number of postcards, if possible, reducing the backpacker's funds appropriately and increasing the count of postcards sent. If there is not enough money, sends as many postcards as possible without allowing the funds to go below zero.

`public void callHomeForMoney()`
Increases the backpacker's funds by an amount equal to SYMPATHY_FACTOR

times the number of postcards sent since the last call to this method, and resets the count of the number of postcards sent back to zero.

Where's the main() method??

There isn't one! Like most Java classes, these are not complete programs and you can't "run" them by themselves. A class is just 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 such the example below.

Sample usage

The best way to think about a specification like this is to try to write some simple test cases and think about what behavior you expect to see. Make up some sample values and work out what the results should be with pencil and paper. Once you figure something out by hand, you can usually write the code using the same sequence of steps.

What follows is an annotated example of this process. You might start with the **`Location`** class, since it is pretty simple and the **`Backpacker`** class depends on it.

```
public class LocationTest
{
  public static void main(String[] args)
  {
    Location c = new Location("Paris", 75);

    // We should see the values with which we constructed it
    System.out.println(c.getName());     // expected "Paris"
    System.out.println(c.lodgingCost()); // expected 75
```

For **`maxLengthOfStay`**, suppose we have 500 euros. 75 goes into 500 6 times, so with 500 euros we could stay a maximum of 6 nights. On the other hand, if we only have 50 euros, we can stay zero nights.

```
    System.out.println(c.maxLengthOfStay(500));  // expected 6
    System.out.println(c.maxLengthOfStay(50));   // expected 0
```

Next, what about **`costToSendPostcard`**? We're supposed to multiply the lodging cost by the given percentage **`RELATIVE_COST_OF_POSTCARD`**, defined as 0.05. So 0.05 * 75 is 3.75. This needs to be rounded to the nearest integer, which is 4.

```
   System.out.println(c.costToSendPostcard());  // expected 4
```

Finally, we need **`maxNumberOfPostcards()`**. If we have 50 euros, and the cost to send each one is 4 euros, then we can send 12 postcards.

```
System.out.println(c.maxNumberOfPostcards(50)); // expected 12
```

For testing the **`Backpacker`** class, there is a bit more going on: you have the current location, the journal, the funds, the postcards. *You don't have to think about it all at once*. You could start out just looking at part of the problem, say, how the location changes as the backpacker visits new places. You can observe the location via the **`getCurrentLocation()`** and **`getJournal()`** accessor methods. What should you observe when you call those methods?

```
public class BackpackerTest
{
  public static void main(String[] args)
  {
    // a few places to visit
    Location paris = new Location("Paris", 75);
    Location rome = new Location("Rome", 50);

    // start out in Paris
    Backpacker t = new Backpacker(500, paris);

    // initial state
    System.out.println(t.getCurrentLocation()); // expected "Paris"
    System.out.println(t.getJournal());     // expected "Paris(start)"

    // try going to Rome
    t.visit(rome, 2);
    System.out.println(t.getCurrentLocation()); // expected "Rome"
    System.out.println(t.getJournal());    // expected "Paris(start),Rome(2)"

    // back to Paris for a week
    t.visit(paris, 7);
    System.out.println(t.getCurrentLocation()); // expected "Paris"
    System.out.println(t.getJournal());    // "Paris(start),Rome(2),Paris(7)"
```

Next, what happens to the funds if we follow the same itinerary as above? For Rome, we can determine (from the **`Location`** object) that we have enough money for 10 nights. We're staying 2 nights, which is less than 10, so we buy lodging for 2 nights. That costs 2 * 50, or 100 euros, so we expect to have 400 left.

```
    t = new Backpacker(500, paris);  // start over

    // initial state
    System.out.println(t.getCurrentFunds()); // expected 500

    // visit a location
    t.visit(rome, 2);
    System.out.println(t.getCurrentFunds()); // expected 400
```

When we go back to Paris for a week, it's a bit different. We determine (from the **`Location`** object) that 400 euros is enough for only 5 nights. The desired number is 7 nights, and the smaller of these values is 5. So we buy lodging for only 5 nights, at a cost of 5 * 75 = 375 euros. We should have 25 euros left. This also implies that we slept in the train station for 7 - 5 = 2 nights.

```
t.visit(paris, 7);
System.out.println(t.getCurrentFunds()); // expected 25
System.out.println(t.getTotalNightsInTrainStation()); // expected 2
```

Paris is a lot of fun, so we stay another week. This time we discover we have only enough money for zero nights lodging. The smaller of zero and 7 is zero, so we purchase zero nights lodging at a cost of zero times 75. So we still have 25 euros, but we've slept another 7 -0 = 7 nights in the train station.

```
t.visit(paris, 7);
System.out.println(t.getCurrentFunds()); // expected 25
System.out.println(t.getTotalNightsInTrainStation()); // expected 9
```

Ok, we're tired, but are we SOL? The cost of a postcard from Paris is 4 euros. Is that more than we have left? The expression 4 > 25 is **`false`**, so we are not yet SOL. Let's send a postcard home! It will cost us just 4 euros.

```
t.sendPostcardsHome(1);
System.out.println(t.getCurrentFunds()); // expected 21
```

Before we call home and ask for more money, let's really get on Mom and Dad's good side. Try sending 12 postcards! With 21 euros, the maximum number of postcards we can send is 5. The smaller of 5 and 12 is 5, so that's the number we actually send. We should see our funds go down by 5 times the postcard cost, or 20 euros, leaving us one euro.

```
t.sendPostcardsHome(12);
System.out.println(t.getCurrentFunds()); // expected 1
```

At this point we're technically SOL: we have 1 euro, the cost of sending a postcard is 4 euros, and the expression 4 > 1 is **`true`**.

```
System.out.println(t.isSOL()); // expected true
```

But, we've sent a total of 6 postcards home. When we call and ask for money, our funds should go up by 6 times **`SYMPATHY_FACTOR`**, 6 * 30 = 180, leaving us sitting pretty with 181 euros.

```
t.callHomeForMoney();
System.out.println(t.getCurrentFunds()); // expected 181
```

If we ask for money again now, nothing should happen - Mom and Dad will have no more sympathy for us until we send more postcards.

```
t.callHomeForMoney();
System.out.println(t.getCurrentFunds()); // still just 181
```

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 examples above.

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 a detailed guide for how an experienced coder might go about creating a class such as this one:*

0. Be sure understand the basics of defining a class as in Sections 3.1 - 3.3 of the text and as practiced in Lab 2.

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

2. Create the **`Location`** class in the **`hw1`** package and put in stubs for all the required methods and the constructor. (For methods that are required to return a value, just put in a "dummy" **`return`** statement that returns zero or **`false`**.) Do the same for the **`Backpacker`** class. Make sure you have no compile errors (red error markers). Write a brief javadoc comment for each method. Try to mentally classify each one as an *accessor* or *mutator*.

3. Write a simple test class for the **`Location`** class, as illustrated in the previous section. Make sure it runs, even though most of the answers will be wrong.

4. Think about instance variables for **`Location`**. One rule of thumb is to look at the accessors: for example, in order for **`getName()`** to return the correct value, that information has to be present within the object. That suggests you need an instance variable to store the name. Make sure to initialize it to the correct value in the constructor. There is a similar situation for the lodging cost. What about the cost of sending a postcard? Do you need another instance variable for that? Not really, since it is calculated from the lodging cost. It is a good practice to **avoid redundant instance variables**.

5. Once you have instance variables defined and correctly initialized, it is pretty straightforward to implement the methods. If you are not sure what to do, work out a few more examples by hand as in the previous section.

> Tip: to round a floating-point number to the nearest integer, you can use the method **`Math.round`**. For example, after the statement
>
> ```
> int i = (int) Math.round(3.75);
> ```
>
> the variable i has value 4.
>
> Aside: there is a minor technicality seen above, which is that **`Math.round`** actually returns a type **`long`** (another type of whole number), not an **`int`**. To use it as an **`int`** we have to "cast" it, which is where the **`(int)`** in parentheses comes from.

6. You can start with the **`Backpacker`** class as long as you at least have the **`getName()`** and **`lodgingCost()`** methods of the **`Location`** class working (though you won't be able to implement everything until you have the remaining methods correct as well). Again, start with some simple test cases as described in the Sample Usage section above. *Remember, you don't have to implement it all at once. Implement one feature at a time, and test as you go.*

7. In the first step of the sample test code we called the method **`getCurrentLocation()`**, which is supposed to return the name of the location that the backpacker is currently in. In order to return the correct value, you'll have to keep track of the backpacker's current location, suggesting you need an instance variable to store the current location whenever **`visit()`** is called. How should it be initialized? We then called the method **`getJournal()`** that returns a string with names of *all* the cities visited. So we need a String variable for the journal that we can update each time we visit a new location. How is it supposed to be initialized? Once you have these two variables correctly initialized, you can implement the two accessors easily and write the statements in **`visit()`** that will keep them updated. After you get this much done, run your simple test and make sure you get the values you expect.

8. Next it makes sense to look at money. We have an accessor, **`getCurrentFunds()`**, suggesting that we need an instance variable to keep track of the funds. Be sure to initialize it in the constructor. For updating the funds in **`visit()`**, work carefully through the examples in the previous section. In particular, look what happened when we had 400 euros and visited Paris for a week:

> *maximum number of nights we can afford with 400 euros* = 5
> *number of nights we're staying* = 7
> *number of nights lodging we actually purchase* = **the smaller of these two numbers**
>
> Tip: To implement the logic of the third step, you can use the method **`Math.min`**, which returns the smaller of two given numbers:
>
> ```
> int x = Math.min(3, 4); // now x is 3
> ```

9. There is also an accessor **`getTotalNightsInTrainStation()`**. There doesn't seem to be any way to calculate this value from the instance variables we already have (location, journal, funds) so we probably need another one. Add the code to update it in **`visit()`**.

10. The remaining two methods are slightly more subtle. Notice that for **`callHomeForMoney()`** to do the right thing, it has to know how many postcards had been sent home. Can this be calculated from information we already have? No - we need another instance variable to count postcards. This variable is used (and then zeroed) in **`callHomeForMoney()`**, but it has to be updated in **`sendPostcardsHome()`**. The logic can be implemented with **`Math.min`** (as in step 8).

11. For **`isSOL`**, you're returning a boolean value. It should be **`true`** if the current funds are less than the cost to send a postcard, and **`false`** otherwise. You might think you need an "if" statement here, but you don't. If your instance variable for the current location is, say, **`currentLocation`**, you can just return the expression,

```
return getCurrentFunds() < currentLocation.costToSendPostcard();
```

12. Download the specchecker, import it into your project as you did in lab 1, and run it. If there are error messages, *always start reading 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 Canvas Assignments page for the link. 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”, which can be found linked on the 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? 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**.

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 must include the **`@author`** tag, and method javadoc 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 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 reading input or producing console output. You may add **`println`** statements when debugging, but you need to remove them before submitting the code.

- Use the defined constant **`RELATIVE_COST_OF_POSTCARD`**. Don't embed the number 0.05 in your code. Use the defined constant **`SYMPATHY_FACTOR`**. Don't embed the number 30 in your 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 interal 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" and select "Instructors" so only the staff. 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 document on the Canvas front page 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. (We promise that no official clarifications causing a change in the spec 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 two files, **`Location.java`** and **`Backpacker.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 by downloading the zip file you submitted and looking inside it. If you are not sure how to do this, see the document "Assignment Submission HOWTO" which is linked on the 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 two files **Location.java** and **Backpacker.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. Detailed instructions can be found in the Assignment Submission HOWTO.