Com S 227
Spring 2019
Assignment 1
100 points

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

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

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

No submissions accepted after February 12, 11:59 pm

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

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

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

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

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

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

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

Overview

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

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

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

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

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

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

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

Specification

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

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

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

There is one public constructor and eleven public methods:

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

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

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

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

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

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

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

`public void pickUp()`

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

`public void dropOff()`

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

`public double getTotalCredits()`

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

`public double getProfit()`

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

`public double getAverageProfitPerHour()`

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

Where's the main() method??

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

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

Suggestions for getting started

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

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

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

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

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

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

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

```
import hw1.UberDriver;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The SpecChecker

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

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

More about grading

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

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

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

Style and documentation

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

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

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

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

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

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

If you have questions

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

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

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

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

What to turn in

Please submit, on Canvas, the zip file that is created by the SpecChecker. The file will be named **`SUBMIT_THIS_hw1.zip`**. and it will be located in whatever directory you selected when you ran the SpecChecker. It should contain one directory, `hw1`, which in turn contains one file, `UberDriver.java`. Please LOOK at the file you upload and make sure it is the right one!

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

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

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

Com S 227
Spring 2019
Assignment 2
200 points

Due Date: Thursday, February 21, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm Feb 20)

10% penalty for submitting 1 day late (by 11:59 pm Feb 22)

No submissions accepted after February 22, 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 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.

Note: Our first exam is Monday, February 25, which is just four 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****. We can post a sample solution on February 23 if anyone is interested.*

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

Introduction

The purpose of this assignment is to give you lots of practice working with conditional logic. For this assignment you'll create one class called `CricketGame` that is a simplified model of the game of *cricket*.

Whoa, you say, but I don't know anything about cricket. Do not worry, until the day before yesterday neither did we! We just read about the rules on Wikipedia, made some simplifications, and wrote up a specification. In the past we have made assignments out of games like baseball and American football, so this semester we thought we'd level the playing field (haha) for the non-US students.

As with our previous assignment, all you really need to do is implement the methods as specified and you'll be fine. Write simple test cases to make sure you know what you want the code to do. Remember you can always post your *test* code on Piazza for comments if you are not entirely sure you have the right idea of what the spec is saying.

Overview and specification

The specification for this assignment is provided in the form of an online javadoc:

http://web.cs.iastate.edu/~cs227/assignments/hw2/doc/

which also includes a concise overview of the game as relevant for this assignment. If you want a more in-depth understanding, a good place to look is the article, "Cricket Explained (An American Viewpoint)":

http://static.espncricinfo.com/db/ABOUT_CRICKET/EXPLANATION/CRICKET_EXPLAINED_AMERICAN.html

The `api` package

NOTE: The two types `Outcome` and `Defaults` in the `api` package are provided for you. Do not modify them and do not add any code to the `api` package. Your `CricketGame` class needs to be in a package named `hw2`.

In Assignment 1, recall that we defined two *constants* as `public static final` values in the `UberDriver` class. Here we are also defining some constants. One set of constants is defined in the class `Defaults` as `public static final` values. You will need to use these when you implement the no-argument constructor of `CricketGame`. The only new thing to learn is that since they are not defined within `CricketGame`, you need to qualify them with the class name when you use them. For example, you'd write something like

```
someInstanceVariable = Defaults.DEFAULT_NUM_INNINGS;
```

after importing `api.Defaults`.

The other set of constants are defined as an `enum` type called `Outcome`. This type is used as the argument type for the `bowl` method. They are really just like the numeric constants we have seen before, with the advantage that they are *type-safe*: it is impossible to accidentally pass an invalid or out of range value to the `bowl` method, since the argument *must* be one of the constants listed in the `enum` type `Outcome`. The way you use them is just like integer constants. For example, in your implementation of the `bowl` method you might have code such as

```
if (outcome == Outcome.WICKET)
{
  playersOut = playersOut + 1;
  // etc...
```

Tip: Of course, you need an import for `api.Outcome.` If you ALSO add the line:

```
import static api.Outcome.*;
```

to the top of your `CricketGame` class, you don't have to keep typing "`Outcome.`" in front of each one. For more information on enums see "Special Topic 5.4" which is in Section 5.5 of the textbook.

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 a message such as `"2 out of 2 tests pass"` in the console output. (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 2

In addition, since this is the second 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 Assignment 1. It will also offer to create a zip file for you to submit. ***Specchecker 2 should be available February 17. Please do not wait until that time to start testing!***

Getting started

Remember to work ***incrementally*** *and test new features as you implement them. Always start with simple test cases or usage examples. The "getting started" section from Assignment 1 provided a detailed, step-by-step example of this process. For this assignment, we will provide some general guidance below, but we expect that by now you can start writing your own test cases.*

1. Create a new, empty project and then add two packages called **`api`** and **`hw2`**. In the **`api`** package, create two classes named **`Defaults`** and **`Outcome`**. Copy and paste the given sample code for Defaults.java and Outcome.java. (You might notice that **`Outcome`** is declared as an **`enum`**, not a **`class`**; this is explained on page 2.)

2. In the **`hw2`** package, create a class named **`CricketGame`**. Based on the javadoc, put in stubs for all the required methods and the two constructors. Make sure there are no compile errors. You'll need an import for **`api.Outcome`**. Write a brief javadoc comment for each method. At this point you can download and run specchecker #1 to make sure you have all the method declarations correct. **Do not add to or modify the code in the `api` package.**

3. If you look over the accessor methods, you can get an idea what instance variables you are going to need - you'll need to keep track of the score for each side, the outs, the bowls, and so on. Remember to avoid redundancy. For example, since side 0 always starts, you don't need a separate variable to keep track of which side is batting - you can get it from the number of innings. You will probably also have to store the values given in the constructor. Each time you define an instance variable, name it carefully and write a brief comment stating what it represents.

4. You can't go much further without starting on the **`bowl`** method. It will have a number of cases based on the argument. The simplest to begin with might be the cases for **`Outcome.WIDE`**, and **`Outcome.NO_BALL`**, which affect only the batting side's score. You could write a simple test case like this:

```
CricketGame g = new CricketGame(2, 3, 4, 6);
g.bowl(Outcome.WIDE);
g.bowl(Outcome.WIDE);
System.out.println(g.getScore(true)); // expected 2
```

(The boolean argument to **getScore** indicates whether you want the batting side's score or the other side's score. For the moment, you can just return the batting side's score and worry about the other team later. Similarly, the **bowl** method is specified to do nothing if the game is over or if the ball is already in play. You can also add this check later on.)

5. The cases for **Outcome.BOUNDARY_SIX** and **Outcome.BOUNDARY_FOUR** also affect the score, but also count towards the number of *bowls*. A simple test might look like this:

```
g = new CricketGame(2, 3, 4, 6);
g.bowl(Outcome.BOUNDARY_SIX);
System.out.println(g.getScore(true)); // expected 6
System.out.println(g.getBowlCount()); // expected 1
```

6. Once you start counting bowls, you also need to worry about the *overs*. Once the number of bowls reaches a certain value (2 in the example above, the first argument to the constructor), you increment the overs and start counting bowls again.

```
g = new CricketGame(2, 3, 4, 6);
g.bowl(Outcome.BOUNDARY_SIX);
g.bowl(Outcome.BOUNDARY_SIX);
g.bowl(Outcome.BOUNDARY_SIX);
g.bowl(Outcome.BOUNDARY_SIX);
g.bowl(Outcome.BOUNDARY_SIX);
System.out.println(g.getBowlCount()); // expected 1
System.out.println(g.getOverCount()); // expected 2
```

7. Some of the cases in **bowl** immediately get the batsman out, such as **Outcome.WICKET**. Try incrementing the *outs* for the batting side. Write some test cases.

8. When the number of overs reaches a certain amount, or when the number of outs reaches a certain amount, you have to increment the *innings* and switch to the other side. (In order to test your code you'll need to have implemented **whichSideIsBatting** and make sure your **getScore** returns the correct value for either side.). Try this:

```
g = new CricketGame(2, 3, 4, 6);
g.bowl(Outcome.BOUNDARY_SIX);
g.bowl(Outcome.BOUNDARY_SIX);
g.bowl(Outcome.BOUNDARY_SIX);
g.bowl(Outcome.BOUNDARY_SIX);
g.bowl(Outcome.BOUNDARY_SIX);
g.bowl(Outcome.BOUNDARY_SIX);
System.out.println(g.getBowlCount()); // expected 0
System.out.println(g.getOverCount()); // expected 0
System.out.println(g.getCompletedInnings()); // expected 1
System.out.println(g.getScore(true)); // expected 0
System.out.println(g.getScore(false)); // expected 36
```

9. Next you might continue with `Outcome.HIT`. Again, start with simple test cases that you can check:

```
g = new CricketGame(2, 3, 4, 6);
g.bowl(Outcome.HIT);
System.out.println(g.isInPlay()); // expected true
g.safe();
System.out.println(g.isInPlay()); // expected false
```

Continue in a similar way, writing some simple tests for `tryRun` and `runOut`, for example,

```
g = new CricketGame(2, 3, 4, 6);
g.bowl(Outcome.HIT);
g.tryRun();
g.tryRun();
g.tryRun();
g.runOut();
System.out.println(g.getScore(true));  // expected 2
System.out.println(g.getOuts());       // expected 1
```

10. Other details include detecting when the game has ended, making sure that `bowl` does nothing when the game is over, and so on.

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 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**.

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

Using "helper" methods

The public API for your class must exactly match the javadoc, but you can define additional **`private`** methods if you wish to simplify your implementation. For example, incrementing the score for the batting side means having to check which side is batting and then adding to the appropriate score variable. Instead of having this check in multiple places, create a private method that does it. If you look at your code, you'll likely see several other places where the same logic is duplicated; for example, after many operations, you'll need to check the number of bowls and see whether it's time to increment overs, check the overs or outs to see whether it's time to switch innings. When you see duplicated logic in several places, ask yourself whether it makes sense to move the common code into a helper method.

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.

- 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 `assignment2`. 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 `assignment2`. *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 "code" to have the editor keep your code formatting. You can also use "pre" for short code snippets.)

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

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

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

What to turn in

Please submit, on Canvas, the zip file that is created by the SpecChecker. The file will be named **`SUBMIT_THIS_hw2.zip`**. and it will be located in whatever directory you selected when you ran the SpecChecker. It should contain one directory, **`hw2`**, which in turn contains one file, **`CricketGame.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 2 submission link and verify that your submission was successful. If you are not sure how to do this, see the document "Assignment Submission HOWTO", which can be found in the Piazza pinned messages under "Syllabus, office hours, useful links."

> We recommend that you submit the zip file as created by the specchecker. If necessary for some reason, you can create a zip file yourself. The zip file must contain the directory **hw2**, which in turn should contain the file **CricketGame.java.** If you are using Eclipse, 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.

Com S 227
Spring 2019
Miniassignment 1
40 points

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

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

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

No submissions accepted after March 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#ad , 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 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.

Note: This is a miniassignment and the grading is automated. If you do not submit it correctly, you will receive at most half credit.

Also note: we expect to have one more miniassignment before break, which should be available by March 9, which is prior to the due date for this one. So start early!

Overview

This is a short set of practice problems involving writing loops. You will write eight methods for the class `mini1.FromLoopToNuts`. All of the methods are static, so your class will not have any instance variables (or any static variables, for that matter). There is a constructor, but it is declared `private` so the class cannot be instantiated.

For details and examples see the online javadoc.

You do *not* need arrays or ArrayLists for this assignment, though you will not be penalized for using them. None of the problems requires nested loops.

Advice

Before you write any code for a method, **work through the problem with a pencil and paper on a few concrete examples**. Make yourself write everything down; in particular, write down things that you need to remember from one step to the next (such as indices, or values from a previous step). Try to explain what you are doing in words. Write your algorithm in pseudocode.

Another key problem-solving strategy is to try solving *part* of the problem, or solving a *related, simpler problem*. For example, here are some ideas for getting started on each problem using this strategy:

- To start solving **`countMatches`**, can you
  a. iterate over a string and just print the characters one at a time?
  b. given two strings the same length, print the characters of *both* strings one at a time?
  c. repeat (b) but print "boo" each time the strings have the same character at some index i?
  d. figure out the largest index that is valid in both strings?

- To start solving **`isArithmetic`**, can you
  a. parse the string and just print the numbers one at a time? (*Tip: use a Scanner, and call the method* **`setDelimiter(",")`** *to split the tokens at commas)*
  b. determine whether the string contains two or more values?
  c. find the difference between the first two values?
  d. parse the string and print out the difference between each value and the previous one?

- To start solving **`eliminateRuns`**, can you
  a. iterate over a string and just print the characters one at a time?
  b. iterate over a string and append each character onto a new string? *(Tip: see the section below on creating a string with a loop)*
  c. iterate over a string and append each character onto a new string, only if it isn't already the last character of the new string?

- To start solving **`threeInARow`**, can you
  a. write a loop to generate random numbers until you get a 7?
  b. write a loop to generate random numbers until you get a value that matches the previously generated number?

- To start solving **`findEscapeCount`**, can you
  a. write statements to update a and b for just one iteration? *(See the javadoc for some sample values.)*
  b. do it for two iterations? *(See the javadoc for some sample values.)*

- To start solving **`differByOneSwap`**, can you
  a. iterate over a string and print each character one at a time?
  b. iterate over a string and print out each pair of adjacent characters?
  c. do the same for two strings?
  d. do the same for two strings, and print "boo" whenever the two pairs have the same characters in the opposite order?
  e. count the number of times that (d) happens?

- To start solving **`countSubstringsWithOverlap`**, can you
  a. use the **`indexOf`** methods to check whether one string is a substring of another?
  b. given strings s and t, if t occurs at index i in s, make a substring containing everything in s that is *after* index i?
  c. do (b) in a loop until t does not occur?

- To start solving **`countSubstringsWithOverlap`**, can you
  a. use the **`indexOf`** methods to check whether one string is a substring of another?
  b. given strings s and t, if t is a substring of s, make a substring containing everything in s that is *after* the occurrence of t?
  c. do (b) in a loop until t does not occur?

My code's not working!!

Developing loops can be hard. If you are getting errors, a good idea is to take a simple concrete example, and trace execution of your code by hand (as illustrated in section 6.2 of the text) to see if the code is doing what you want it to do. You can also trace what's happening in the code by temporarily inserting **`println`** statements to check whether variables are getting updated in the way you expect. (Remember to remove the extra **`println`**'s when you're done!)

Overall, the best way to trace through code with the debugger, as we are practicing in Lab 6. Learn to use the debugger effectively, and it will be a lifelong friend.

Always remember: one of the wonderful things about programming is that within the world of your own code, you have absolute, godlike power. If the code isn't doing what you want it to do, you can decide what you really want, and make it so. **You are in complete control**!

(If you are not sure what you *want* the code to do, well, that's a different problem. Go back to the "Advice" section.)

What do you mean, a private constructor?

Just put this declaration at the top of your class:

```
/**
 * Private constructor disables instantiation.
 */
private FromLoopToNuts() { }
```

How do I make a string with a loop, as needed for `eliminateRuns`?

Start with an empty string and concatenate additional characters in each iteration. For example, here is one way to create the reverse of a given string:

```
public static String reverse(String s)
{
  String result = "";  // start with empty string
  for (int i = s.length() - 1; i >= 0; i = i - 1)
  {
    result += s.charAt(i); // add on characters one at a time
  }
  return result;
}
```

As an aside, experienced Java programmers would probably use a **`StringBuilder`** object, which works like a mutable type of string:

```
private static String reverse(String s)
{
  StringBuilder sb = new StringBuilder();
  for (int i = s.length() - 1; i >= 0; i = i - 1)
  {
    sb.append(s.charAt(i));
  }
  return sb.toString();
}
```

Testing and the SpecChecker

A SpecChecker will posted shortly that will perform an assortment of functional tests. As long as you submit the assignment correctly, your score will be exactly the score reported by the specchecker.

However, when you are debugging, it is much more helpful if you have a simpler test case of your own that reproduces the error you are seeing.

Remember that to call a static method, you prefix it with the *class* name, not with an object reference. For example, here is simple test case for the **`countMatches`** method:

```
import mini1.FromLoopToNuts;

public class SimpleTest
{
  public static void main(String[] args)
  {
    int result = FromLoopToNuts.countMatches("abcde", "xbydcazzz");
    System.out.println(result);
  }
}
```

You can save yourself from having to type "**`FromLoopToNuts`**" over and over again by using the Java feature **`import static`**, which allows you to invoke a static method without typing the class name:

```
import static mini1.FromLoopToNuts.*;

public class SimpleTest
{
  public static void main(String[] args)
  {
    int result = countMatches("abcde", "xbydcazzz");
    System.out.println(result);
  }
}
```

Since no test code is being turned in, you are welcome to post your tests on Piazza for others to use and comment on.

Documentation and style

Since this is a miniassignment, the grading is automated and in most cases we will not be reading your code. Therefore, there are no specific documentation and style requirements. However, writing a brief descriptive comment for each method will help you clarify what it is you are trying to do.

If you have questions

For questions, please see the Piazza Q & A pages and click on the folder **`miniassignment1`**. 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 **`miniassignment1`**. *But remember, do not post any source code for the classes that are to be turned in.* It is fine to post source code for general Java examples that are not being turned in. (In the Piazza editor, use the button labeled "pre" to have Java code formatted the way you typed it.)

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

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

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

What to turn in

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 Canvas, the zip file that is created by the SpecChecker. The file will be named **`SUBMIT_THIS_mini1.zip`**. and it will be located in the directory you selected when you ran the SpecChecker. It should contain one directory, **`mini1`**, which in turn contains one file, **`FromLoopToNuts.java`**.

Always LOOK in the zip file the file to check what you have submitted!

Submit the zip file to Canvas using the Miniassignment1 submission link and verify that your submission was successful. If you are not sure how to do this, see the document "Assignment

Submission HOWTO" which can be found in the Piazza pinned messages under “Syllabus, office hours, useful links.”

We strongly recommend that you just submit the zip file created by the specchecker. If you mess something up and we have to run your code manually, you will receive at most half the points.

> We strongly 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 **mini1**, which in turn should contain the file **`FromLooptoNuts.java`**. You can accomplish this 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.

Optional reading: What's the point of that method `findEscapeCount`?

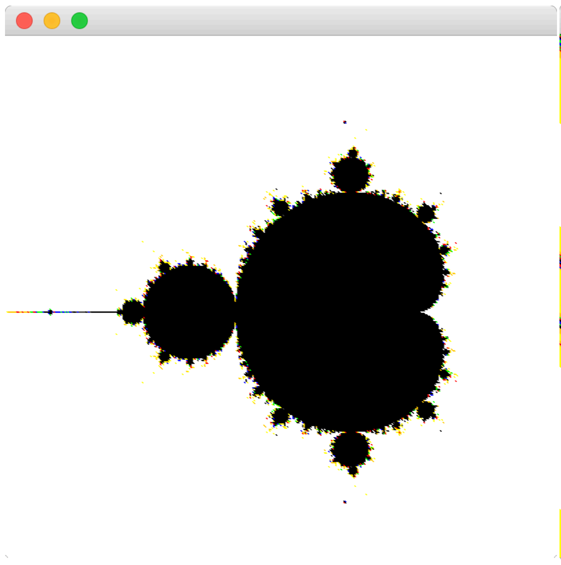
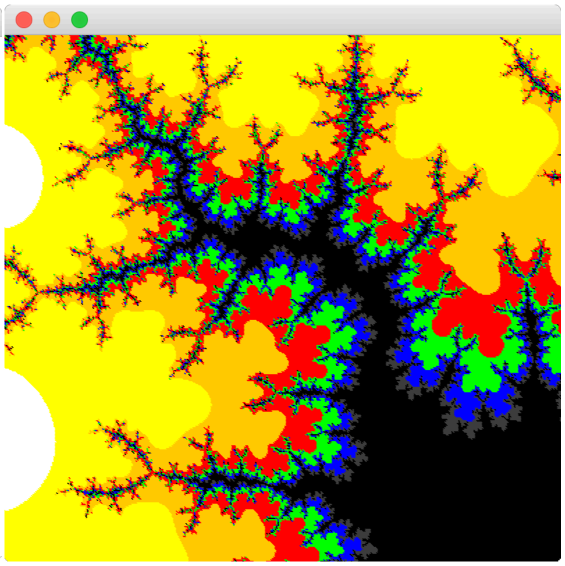
This section is not part of the assignment, but could be fun to investigate. It describes a graphical application that can be used to visualize the effect of your **`findEscapeCount`** *method.*

The short algorithm you are implementing in **`findEscapeCount`** defines a representation of what is known as the *Mandelbrot set*, which can be loosely defined[1] as the set of points (x, y) for which the value $\sqrt{a^2 + b^2}$ is always less than or equal to 2, no matter how many times the given steps are repeated. If this value ever goes over 2, we say it has "escaped" the bound. To approximate this set of (x, y) pairs, we pick a maximum number of iterations (say 100) and see whether the value escapes within that many iterations. If not, we assume it is part of the set. The number of iterations it actually takes to escape for a given (x, y) is an interesting piece of information, and the number is often used to study the *boundary* of the Mandelbrot set, which exhibits a detailed self-similar or "fractal" structure (looks the same no matter how much you zoom in). People typically create a graphical representation using different colors to depict different ranges of escape values.

To experiment with what this means, take a look at the application MandelbrotTest.java. This is a simple GUI that depicts the Mandelbrot set, based on your code for findEscapeCount. Points that are in the set are colored black, and points on the boundary are given a color depending on the escape value. You can select an area with the mouse to repeatedly zoom in. The initial screenshot on the left depicts the whole set, in a 3 x 3 region at the origin. The image on right is zoomed in to an area of roughly .005 x .005 on the top of the upper "bulb". You can edit the

[1] More precisely, it's the set of *complex* numbers $c = x + yi$ such that the modulus of z_n never exceeds 2 in any number of iterations of the steps $z_{n+1} = z_n^2 + c$. In our case we're doing the same arithmetic using the notation $c = x + yi$ and $z = a + bi$ (where i is a symbol representing the square root of -1).

CUTOFFS array near the top of the file to change the way the colors are assigned (see the getColor method).

The GUI code is based on the Java Swing libraries. This style of programming is specialized (not to mention somewhat tedious) and we don't cover it in this course. However, you might want to investigate it on your own. Much of what you find about Swing on the internet is out of date or incorrect. The official Oracle tutorial, however, is detailed and accurate,

http://docs.oracle.com/javase/tutorial/uiswing/start/index.html

and there is a also simpler collection of examples on Steve's web page,

http://web.cs.iastate.edu/~smkautz/

(scroll down to "Other stuff").

COM S 227
Spring 2019
Miniassignment 1: A simple computer
40 points

Due Date: Friday, March 15, 11:59 pm (midnight)
5% bonus for submitting 1 day early (by 11:59 pm March 14)
10% penalty for submitting 1 day late (by 11:59 pm March 16)
No submissions accepted after March 16, 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#ad , 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 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.

Note: This is a miniassignment and the grading is automated. If you do not submit it correctly, you will receive at most half credit.

Overview

In this assignment you will implement a simple instruction set for a toy accumulator architecture. An *accumulator architecture* is a class of computer architecture in which most or all instructions work on an accumulator register. This is in contrast with a *register architecture*, which has many registers that may be selected and used for computation and control.

Accumulator architectures were very common in the early computers from the 1940s through the 1960s, but as the science of computing matured, we came to understand that register architectures were superior in many ways, so accumulator architectures disappeared from the computing landscape.

But that accumulator architectures no longer exist in practice does not imply that they are not interesting machines! There is much to be learned by both neophyte and experienced computer scientists about the practice of computing by looking at the design of an accumulator architecture, and by programming one.

Our architecture, called CS227COMP, is digital (base 10, as opposed to binary, base 2) and has only 13 instructions, and yet that is sufficient to solve non-trivial problems; in fact, it's theoretically as "powerful" as any modern computer!

Implementing this processor architecture will give you practice working with arrays and loops. The javadoc provides details and examples of the implementation.

Advice

Before you write any code for a method, **work through the problem with a pencil and paper on a few concrete examples**. Make yourself write everything down; in particular, write down things that you need to remember from one step to the next (such as indices, or values from a previous step). Try to explain what you are doing in words. Write your algorithm in pseudocode.

Another key problem-solving strategy is to try solving part of the problem, or solving a related, simpler problem. Often it's useful to break a problem into one or more sub-problems that can be solved in methods of their own. "Helper" methods, like these, are for internal (inside a class) use only and should have *private* access; you, the author of the class call them to help you implement the functionality of the class, but allowing the user of the class to call them could potentially corrupt class state. Our CS227COMP implementation has 17 private methods to handle things like executing each of the instructions, updating the instruction counter, and dealing with integer overflow. This is not to say that you should have helper methods! Only to get you to think about how a good design may look.

Here are some ideas for getting started:

- Among other initialization, your constructors should allocate your memory array.
- Remember that accessors ("getters") should get and mutators ("setters") should set. Don't *combine* the functionality!
- `runProgram()` simply runs the next instruction in a loop!
- To get started on `loadMemoryImage()`, can you
  1. determine the size of the smaller of two arrays?
  2. iterate over an array, checking values and copying them?
- To get started on loadProgramFromConsole(), can you
  1. prompt for input from a user?
  2. read an unknown number of integers from a scanner until you see a sentinel?
  3. check that the integers are in the representable range?
- To get started on loadProgramFromFile(), see `loadProgramFromConsole()`. This is essentially a simpler version without prompts.
- `nextInstruction()` is the meat of this program. It may seem intimidating when you consider that you have to correctly execute any of the 13 instructions, but, done well, this method should be simpler (longer, true, but simpler) than either of the two load methods. For instance, as mentioned above, the instructions themselves are most cleanly implemented in private helper functions. Given that, this method is a matter of figuring out which instruction is being executed and calling the appropriate helper.
  Ideally, instruction counter update would occur at exactly one place directly within `nextInstruction()`; however, the jump instructions throw a wrench in the works. It's not impossible to special-case the jumps and do your instruction counter update here, but its probably cleaner to do it in each of your instruction helpers instead.
  Helper methods to handle instruction overflow and arithmetic overflow can be called here or in your instruction helpers.
  Crashes can occur here *and* in your load methods. It might be useful to have a helper that handles crashes for you that you can simply call when you need it.

My code's not working!!

Developing loops can be hard. If you are getting errors, a good idea is to take a simple concrete example, and trace execution of your code by hand (as illustrated in section 6.2 of the text) to see if the code is doing what you want it to do. You can also trace what's happening in the code by temporarily inserting `println` statements to check whether variables are getting updated in the way you expect. (Remember to remove the extra `println`'s when you're done!)

Overall, the best way to trace through code with the debugger, as we are practicing in Lab 6. Learn to use the debugger effectively, and it will be a lifelong friend.

Always remember: one of the wonderful things about programming is that within the world of your own code, you have absolute, godlike power. If the code isn't doing what you want it to do, you can decide what you really want, and make it so. **You are in complete control!**

(If you are not sure what you want the code to do, well, that's a different problem. Go back to the "Advice" section.)

Testing and the SpecChecker

A SpecChecker will posted shortly that will perform an assortment of functional tests. As long as you submit the assignment correctly, your score will be exactly the score reported by the specchecker.

However, when you are debugging, it is much more helpful if you have a simpler test case of your own that reproduces the error you are seeing. For example:

```
import mini2.CS227Comp;

public class SimpleTest
{
 public static void main(String[] args)
 {
   String msg = "(LOAD␣4)␣For␣machine␣with␣IC=0,␣"         +
                "AC=0,␣memory=[3004,␣0,␣0,␣0,␣42,␣5],␣"    +
                "after␣nextInstruction(),␣AC␣should␣be␣42.";
   int[] arr = {3004, 0, 0, 0, 42};
   comp = new CS227Comp(0, 0, arr);
   comp.nextInstruction();
   System.out.println(msg + (42 == comp.getAC()));
 }
}
```

Since no test code is being turned in, you are welcome to post your tests on Piazza for others to use and comment on.

Documentation and style

Since this is a miniassignment, the grading is automated and in most cases we will not be reading your code. Therefore, there are no specific documentation and style requirements. However, writing a brief descriptive comment for each method will help you clarify what it is you are trying to do.

If you have questions

For questions, please see the Piazza Q & A pages and click on the folder `miniassignment2`. If you dont 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 `miniassignment2`. *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 whats 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 Canvas, the zip file that is created by the SpecChecker. The file will be named `SUBMIT_THIS_mini2.zip`. and it will be located in the directory you selected when you ran the SpecChecker. It should contain one directory, `mini2`, which in turn contains one file, `CS227Comp.java`.

Always LOOK in the zip file the file to check what you have submitted!

Submit the zip file to Canvas using the Miniassignment2 submission link and verify that your submission was successful. If you are not sure how to do this, see the document "Assignment Submission HOWTO" which can be found in the Piazza pinned messages under Syllabus, office hours, useful links.

We strongly recommend that you just submit the zip file created by the specchecker. If you mess something up and we have to run your code manually, you will receive at most half the points.

> We strongly 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 `mini2`, which in turn should contain the file `CS227Comp.java`. You can accomplish this 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

Com S 227
Spring 2019
Assignment 3
300 points

Due Date: Friday, April 5, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm April 4)

10% penalty for submitting 1 day late (by 11:59 pm April 6)

No submissions accepted after April 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 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.

Note: Our second exam is Monday, April 8, which is just a few 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****.*

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

Introduction

The purpose of this assignment is to give you some practice writing loops, using arrays and lists, and most importantly to get some experience putting together a working application involving several interacting Java classes.

For this assignment you will implement an *assembler* for the simple computer from Miniassignment 2. An assembler is a program that translates *assembly language* into machine code. An assembly language is similar in structure to machine code, but allows you to use symbolic names for opcodes, data locations (variables), and jump targets. There is a detailed example below.

There are two classes for you to implement. **`CS227Asm`** is the assembler itself, and **`AsmFileUtil`** is a utility containing some static methods for working with files. As always, your primary responsibility is to implement these classes according to the specification and test them carefully.

Some additional types you will need are provided in the **`api`** package, including **`Instruction`**, **`NVPair`**, and **`SymbolTable`**. These are described in more detail below. You will need to become familiar with the **`api`** package code in order to use it. These classes are not complex and you'll find that source code is pretty easy to read.

The two classes you write go in package **`hw3`**. You should not modify any of the code in the **`api`** package.

Specification

The specification for this asssignment includes

- this pdf,
- any "official" clarifications posted on Piazza, and
- the online javadoc

The assembly language

An assembly language program is a text file. However, in our implementation, the file-reading is done separately, so that the actual input to the assembler is an ArrayList of strings, one for each line of the file. The file is defined as follows:

- Programs have three sections, *data*, *labels*, and *instructions*, in that order.
- All three sections are required, even if empty.
- Each section starts with a semantic marker on a line by itself: "data:", "labels:", or "instructions:", respectively.
- The *data* section is a list of zero or more name/value pairs of variable names and their initial integer values. Initial values are required even if the program will write the variable before reading it. These name/value pairs occur one per line until the "labels:" semantic marker.

- The *labels* section consists of a series of jump target names. All jump instructions used in the instructions section must have a target that is named in the labels section. The jump target names are listed one per line until the "instructions:" semantic marker.
- The *instructions* section consists of a series of instructions and labels representing jump targets.
    - There are two I/O instructions, "read" and "write", which have a variable as a parameter (that is, they must be followed by a variable name previously defined in the data section).
    - There are two memory instruction, "load" and "store", which have a variable as a parameter. There are five arithmetic instructions, "add", "sub", "div", "mul",
    - and "mod", which have a variable as a parameter.
    - There are three jump instructions, "jump", "jumpn", and "jumpz", which have a jump target as a parameter (i.e., must be followed by a label that was previously defined in the labels section.)
    - There is a "halt" instruction which has no parameter.
    - Jump targets are given by a jump target name (per the labels section) appearing on a line in lieu of an instruction.
- Blank lines are not allowed.
- All lines may include comments that should be ignored. For lines that contain semantic markers, label declarations, labels (jump targets in the instruction section), and the "halt" instruction, comments begin with the *second* non-whitespace token. Variable declarations and instruction other than "halt" all require a parameter, so comments begin with the *third* non-whitespace token.

Here is an example of a CS227Asm program that prints out numbers from 10 down to 1. (The remarks on the right are comments.) Read through the code and see how it works.

```
data:
count 10        count = 10
const_one 1
labels:
point_a
point_b
instructions:
load count
point_a
jumpz point_b   while count != 0
write count         print the value of count
sub const_one       count = count - 1
store count
jump point_a        (go back to top of loop)
point_b
halt
```

Parsing and code generation

The sample program above is assembled into the following machine code. Note that the original instructions (though not the comments) are included in the result as descriptions following the machine instructions, and locations of jump targets are labeled (text in parentheses).

```
+3007 load count
+6306 jumpz point_b (label point_a)
+2007 write count
+5108 sub const_one
+4007 store count
+6001 jump point_a
+8000 halt (label point_b)
+0010 count
+0001 const_one
-99999
```

Here is how it works.

1. First the data section is parsed, creating a "symbol table" for the variables to be used in the program. The symbol table is just a list of name-value pairs that associates each variable with its initial value. In this example, the table would contain the string "count" with initial value 10 and the string "const_one" with initial value 1.

2. Next the label section is parsed, creating a symbol table for the labels. Initially, each label is associated with value 0. When we parse the instruction section, we'll adjust the label values to correspond to the actual memory address at which the label would occur when the instructions are correctly placed into memory.

3. Next the instruction section is parsed, creating a list of `Instruction` objects *and* completing the symbol table for labels using the locations of the jump targets. The `Instruction` type is defined for you in the `api` package. Each `Instruction` encapsulates an opcode, an operand, and a description. The way it works is that the instructions will be placed in memory starting at address 0. By keeping a count of how many instructions have been added to the list so far, whenever we find a label we know what the address associated with that label is. Note that the labels themselves do not increment the instruction count and are not included in the instruction list:

```
data:
count 10
const_one 1
labels:
point_a
point_b
instructions:
load count          <-- instruction 0
point_a             <-- this label gets value 1
jumpz point_b       <-- instruction 1
write count         <-- instruction 2
sub const_one       <-- instruction 3
store count         <-- instruction 4
jump point_a        <-- instruction 5
point_b             <-- this label gets value 6
halt                <-- instruction 6
```

4. The next thing to do is go through the instructions and set the numeric value of the operands. You'll notice that when in the constructor for an `Instruction` object, we record the symbolic name for the operand (either a label for jump instructions, or a variable name for the other instructions), but we don't yet have the numerical value for the operand. How we get that value will differ depending on whether it's a jump instruction or not. For a jump instruction, we find the label in the symbol table for labels and use the address we previously stored there in step 3. As an example, note the line

```
+6306 jumpz point_b
```

in the generated machine code. The operand part, 06, is the memory address of the instruction at the label `point_b` in the assembly code.

All the other instructions (except halt), refer to a variable, so we need the address for the variable. The variables will be laid out in memory immediately following the instructions, so all we need to do is find the index of the variable in the data symbol table, and add the size of the instruction list. As an example, note the line

```
+5108 sub const_one
```

in the generated machine code. The operand part, 08, is the memory address to be associated with the variable `const_one`. We know that because the index of `const_one` in the symbol table is 1 (i.e., it's the second data declaration), the length of the instruction list is 7, and 7 + 1 = 8.

5. Next, we want to include jump target locations in the instruction descriptions. For example, in the sample machine code above, the text in parentheses as seen in the line

```
+8000 halt (label point_b)
```

shows that the jump target represented by label **`point_b`** is memory location 6 (the seventh line of the machine code). This is not hard: in the symbol table for the labels, the value of each label is the index of the instruction whose description should have the label information appended. The **`Instruction`** class has a method **`addLabelToDescription`** to make this easy.

6. Finally, we need to write the machine code as a list of strings. The instructions come first, then the variables. The **`Instruction`** class has a built-in method **`toString()`** that converts the **`Instruction`** into the desired string format for the output. For the variables, just iterate over the symbol table for data. Getting the value to be represented as a four-digit string with the leading plus or minus sign can be done with **`String.format`**, which works the same as **`printf`** as used in Miniassignment 2.

The tasks above are all specified as public methods to simplify testing, so that your **`assemble()`** method can literally be implemented like this:

```
public ArrayList<String> assemble()
{
  parseData();
  parseLabels();
  parseInstructions();
  setOperandValues();
  addLabelAnnotations();
  return writeCode();
}
```

The `AsmFileUtil` class

The **`CS227Asm`** class does not read input or write output. The source program is provided to the constructor as a list of strings, and the machine language program is returned as a list of strings. In practice, an assembler would read its source from a file and write the machine code to another file. The necessary file operations are to be implemented by you as static methods in the utility class **`AsmFileUtil`**. For example, if the sample program above is in a file named "test1.asm227", then the following main method would produce the machine code as console output:

```
public static void main(String[] args) throws FileNotFoundException
{
  ArrayList<String> result = AsmFileUtil.assembleFromFile("test1.asm227");
  for (String s : result)
  {
    System.out.println(s);
  }
}
```

And if you have a working version of the machine simulator CS227Comp.java from Miniassignment 2 (you can download a sample solution), you could actually run the program too:

```
public static void main(String[] args) throws FileNotFoundException
{
  int[] code = AsmFileUtil.createMemoryImageFromFile("test1.asm227");
  CS227Comp comp = new CS227Comp(10);
  comp.loadMemoryImage(code);
  comp.runProgram();
}
```

The `NVPair` and `SymbolTable` classes

In addition to the **`Instruction`** class discussed above, the **`api`** package includes two types **`NVPair`** and **`SymbolTable`**. An **`NVPair`** is a simple type encapsulating a string name and an associated int value, such as a variable name and associated initial value or a jump target label and its associated address. A **`SymbolTable`** is basically a list of **`NVPair`** objects, along with some useful methods for looking things up. Take a look at the code to see what the methods actually do.

Remember: do not modify any code in the **`api`** package.

Notes on error checking

Although our CS227Comp program had a configurable amount of memory, it could only address up to 100 words. This is because of the two-digit operand. Any program + data that ends up greater than 100 words long is invalid. Of course, there are many other things that can invalidate a program that we haven't discussed here as well; for instance, using a variable or label which hasn't been defined, using a label more than once, etc. Handling all possible errors adds a not-insignificant extra level of complexity on top of the assembler. Suffice to say that we (the teaching staff) will not perform tests using deliberately broken programs on your assembler. You may assume that all input programs will be valid. This is not to say that you may not inadvertently (or on purpose) test your program using invalid programs of your own!

In most cases, some type of exception will be thrown whether or not you explicitly check for invalid code. You are not expected to perform error checking, but you may notice exceptions being "thrown" at some points in the **`api`** code. For example, in the in the **`SymbolTable`** method **`findByName`**, if there is no symbol with the given name, it will reach the line

```
throw new IllegalArgumentException(...)
```

which will, not surprisingly, cause your program to exit and display the stack track on the console. You'll find that this is useful for tracking down bugs in your code. (We will study the exception mechanism in some detail at the end of the semester.(

Testing and the SpecChecker

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

We will provide a basic SpecChecker, but **it will not perform any functional tests of your code**. At this point in the course, you are expected to be able to read the specfications, ask questions when things require clarification, and write your own unit tests. 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.**

The SpecChecker will verify the class names and packages, the public method names and return types, and the types of the parameters. If your class structure conforms to the spec, you should see a message similar to this in the console output:

```
x out of x tests pass.
```

This SpecChecker will also offer to create a zip file for you that will package up the two required classes. 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.

> 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.

Importing the sample code

1. Download the zip file. You don't need to unzip it.
2. In Eclipse, go to File -> Import -> General -> Existing Projects into Workspace, click Next.
3. Click the radio button for "Select archive file".
4. Browse to the zip file you downloaded and click Finish.

If you have an older version of Java (below 8) or if for some reason you have problems with this process, or if the project does not build correctly, you can construct the project manually as follows:

1. Unzip the zip file containing the sample code.
2. In Windows Explorer or Finder, browse to the src directory of the zip file contents
3. Create a new empty project in Eclipse
4. In the Package Explorer, navigate to the src folder of the new project.
5. Drag the **`api`** folder from Explorer/Finder into the **`src`** folder in Eclipse.

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 functional tests that we run and partly on the grader's assessment of the quality of your code. 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**.
- Avoid code duplication.
- Internal (//-style) comments are normally used inside of method bodies to explain *how* something works, while the Javadoc comments explain *what* a method does. Use internal comments where appropriate to explain how your code works. (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.)

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 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.
- Try not to embed numeric literals in your code. Use the defined constants wherever appropriate.
- 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 **`assignment3`**. 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 **`assignment3`**. *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 "code" to have the editor keep your code formatting. You can also use "pre" for short code snippets.)

If you have a question that absolutely cannot be asked without showing part of your source code, change the visibility of the post to "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.)

Getting started

At this point we expect that you know how to study the documentation for a class, write test cases, and develop your code incrementally to meet the specification. *In particular, for this assignment we are not providing a specchecker that will perform any functional tests of your code.* It is up to you to test your own code (though you are welcome to share test cases on Piazza). Here are some basic operations:

- Be sure you are familiar with the basic machine operations as implemented in Miniassignment 2. In the Piazza homework post, you can find source code for a sample Miniassignment 2 solution that you can use to try out your machine code.
- Be sure you have done lab 8, which covers reading and writing files as well as the use of ArrayLists.
- Familiarize yourself with the code in the **`api`** package, since you will need to use all of it in one way or another.
- All the operations in **`AsmFileUtil`** depend on having a working **`CS227Asm`**, so you need to start with **`CS227Asm`**.

Each of the six public methods described previously (in "Parsing and code generation") is dependent on the previous one, so it makes sense to begin implementing them in the order given. As always start by writing simple test cases. For example, here is an easy way to create an ArrayList of strings for the sample program from the "Parsing and code generation" section, so you can start testing the assembler without worrying about reading and writing files:

```
String[] temp = {
  "data:",
  "count 10",
  "const_one 1",
  "labels:",
  "point_a",
  "point_b",
  "instructions:",
  "load count",
  "point_a",
  "jumpz point_b",
  "write count",
  "sub const_one",
  "store count",
  "jump point_a",
  "point_b",
  "halt"
};
```

```
    ArrayList<String> program = new ArrayList<>();
    for (String s : temp)
    {
      program.add(s);
    }
```

Then, you could write a simple test that calls **`parseData`** and then checks the symbol table to be sure it worked:

```
    CS227Asm asm = new CS227Asm(program);
    asm.parseData();
    SymbolTable dataTable = asm.getData();
    for (int i = 0; i < dataTable.size(); i += 1)
    {
      NVPair p = dataTable.getByIndex(i);
      System.out.println(p);
    }
```

which should produce the output:

```
count:10
const_one:1
```

See the sample program SimpleTests.java for some more ideas along these lines.

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_hw3.zip`**. and it will be located in whatever directory you selected when you ran the SpecChecker. It should contain one directory, **`hw3`**, which in turn contains two files, **`CS227Asm.java`** and **`AsmFileUtil.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 3 submission link and verify that your submission was successful. If you are not sure how to do this, see the document "Assignment Submission HOWTO", which can be found in the Piazza pinned messages under "Syllabus, office hours, useful links."

> We recommend that you submit the zip file as created by the specchecker. If necessary for some reason, you can create a zip file yourself. The zip file must contain the directory **hw3**, which in turn should contain the two files **`CS227Asm.java`** and **`AsmFileUtil.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.

Com S 227
Spring 2019
Miniassignment 3
30 points

Due Date: Thursday, April 18, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm April 17)

10% penalty for submitting 1 day late (by 11:59 pm April 19)

No submissions accepted after April 19, 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#ad , 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 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.

Note: This is a miniassignment and the grading is automated. If you do not submit it correctly, you will receive at most half credit.

Overview

Imagine you are going to a Sandwich place for lunch. A sandwich in this restaurant can be customarily built from a number of ingredients, such as bread, meats, cheeses, and so on. Each ingredient has a number of options to choose from. For example there could be three options of bread, 2 options of meats, 2 options of cheeses and so on. Your job in this mini-assignment is to list out all the different combinations of ingredients that a customer can have when building a sandwich.

Specifically, you need to write a single recursive method that computes all the combinations. The number of options for each and every ingredient is stored in an array called `options`. That is, the length of `options` is the same as the number of the ingredients and `options` [i] stores the number of options available for ingredient *i*.

For example, a sandwich that uses three ingredients (bread, meat, and cheese) that can be chosen from three types of bread, 2 types of meats, and 2 types of cheeses can be encoded in the following `options` array:

```
int options = {3, 2, 2};
```

All the different combinations of ingredients that a customer can use to build a sandwich are:

`[0, 0, 0],` // use type 0 bread, type 0 meat, type 0 cheese
`[0, 0, 1],` // use type 0 bread, type 0 meat, type 1 cheese
…
`[2, 0, 1],` // use type 2 bread, type 0 meat, type 1 cheese
`[2, 1, 1],` // use type 2 bread, type 1 meat, type 1 cheese

Altogether, there are 3*2*2 = 12 different combinations of ingredients a customer can have. The method you are going to write should return all the different combinations in a 2-D array, with each row of which representing a unique combination of the ingredients. In our example above, the returned 2-D array will have 12 rows and 3 columns (12 combinations of the 3 ingredients).

> *Tip:* it is useful to remember that in Java, what we call a 2-D array is really just an array of arrays. In the above example, the result is a 12-element array, and each of the 12 elements ("rows") is a 3-element `int` array. For example, if `combos` is the 2-D array returned in this example, an easy way to print it row by row is:

```
for (int[] row : combos)
{
  System.out.println(Arrays.toString(row));
}
```

Think recursively

To solve this problem recursively, you may think in the following way. Suppose there are n ingredients and there are m_n options to choose from for the last ingredient. What you can do is to find all the combinations to build a sandwich using only the first $n - 1$ ingredients, which can be achieved by making a recursive call. Now let `int[ ][ ] combos =` … be the returned combinations of the first n-1 ingredients of the recursive call. `combos` should be a 2-D array with n-1 columns and $m_1 \times m_2 \times ... \times m_{n-1}$ rows, where m_i represents the number of options available for the i^{th} ingredient.

For example, using the options {3, 2, 2} above, your recursive call for just the first two options {3, 2} would return the 2D array `combos` consisting of

```
[0, 0]
[0, 1]
[1, 0]
[1, 1]
[2, 0]
[2, 1]
```

Your next step is to include the last ingredient. Recall that there are m_n options for the last ingredient, so you will need `combos.length` times m_n rows for the 2-D array that contains all the combinations of all the n ingredients. To fill it, you can take each array within `combos` and duplicate it m_n times, increasing the length by 1, each time setting a different choice of the last ingredient. In the example above, each array is copied twice since the number of third options m_3 is 2. The result you return would be the 2D array:

```
[0, 0, 0]
[0, 0, 1]
[0, 1, 0]
[0, 1, 1]
[1, 0, 0]
[1, 0, 1]
[1, 1, 0]
[1, 1, 1]
[2, 0, 0]
[2, 0, 1]
[2, 1, 0]
[2, 1, 1]
```

> *Tip*: The static method **`Arrays.copyOf`** is an easy way to duplicate an array while increasing the length. For example, if **`arr`** is `[2, 4, 6]`, then the method call **`Arrays.copyOf(arr, arr.length + 1)`** returns the array `[2, 4, 6, 0].`

Your class should be called **`Combinations`**, placed in a package called **`mini3`**. It should have a single method which **must** be implemented using recursion. The single method should be called:

```
public static int[][] getCombinations(int[] choices) {

}
```

You may assume that the parameter **`choices`** contains at least one **`int`** value and all the values are positive (1 or higher).

Sorting your combinations

In order to produce consistent outputs and also for the SpecChecker to correctly evaluate your method, you need to sort your combinations (a 2-D array) at the end of your method before returning it. You are provided in this mini-assignment with a class called **`ArrayComparator`** that you will need to sort the combinations. To use **`ArrayComparator`** to sort, do the following:

```
Arrays.sort(combinations, new ArrayComparator());
```

where **`combinations`** is the 2-D array you are returning.

The class **`ArrayComparator`** is provided for you and you should not modify it.

Testing and the SpecChecker

A SpecChecker will posted shortly that will perform an assortment of functional tests. As long as you submit the assignment correctly, your score will be exactly the score reported by the specchecker.

However, when you are debugging, it is much more helpful if you have a simpler test case of your own that reproduces the error you are seeing.

Since no test code is being turned in, you are welcome to post your tests on Piazza for others to use and comment on.

Documentation and style

Since this is a miniassignment, the grading is automated and in most cases we will not be reading your code. Therefore, there are no specific documentation and style requirements. However, writing a brief descriptive comment for each method will help you clarify what it is you are trying to do.

If you have questions

For questions, please see the Piazza Q & A pages and click on the folder **`miniassignment3`**. 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 **`miniassignment3`**. *But remember, do not post any source code for the classes that are to be turned in.* It is fine to post source code for general Java examples that are not being turned in. (In the Piazza editor, use the button labeled “pre” to have Java code formatted the way you typed it.)

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

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

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

What to turn in

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 Canvas, the zip file that is created by the SpecChecker. The file will be named **`SUBMIT_THIS_mini3.zip`**. and it will be located in the directory you selected when you ran the SpecChecker. It should contain one directory, **`mini3`**, which in turn contains two files, **`Combinations.java`**, **`ArrayComparator.java.`**

Always LOOK in the zip file the file to check what you have submitted!

Submit the zip file to Canvas using the Miniassignment3 submission link and verify that your submission was successful. If you are not sure how to do this, see the document "Assignment Submission HOWTO" which can be found in the Piazza pinned messages under “Syllabus, office hours, useful links.”

We strongly recommend that you just submit the zip file created by the specchecker. If you mess something up and we have to run your code manually, you will receive at most half the points.

We strongly 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 **mini3**, which in turn should contain two files, **`Combinations.java`**, **`ArrayComparator.java.`** You can accomplish this 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.

Com S 227
Spring 2019
Assignment 4
300 points

Due Date: Friday, May 3, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm May 2)

NO LATE SUBMISSIONS - EVERYTHING MUST BE IN FRIDAY NIGHT

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 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!

Introduction

The purpose of this assignment is to give you some experience working with inheritance and abstract classes in a realistic setting. You'll also get some more practice with 2D arrays and lists.

A portion of your grade on this assignment (roughly 15% to 20%) will be determined by how effectively you have been able to use inheritance to minimize duplicated code in the Piece hierarchy.

Summary of tasks

1. Create implementations of five concrete subtypes of the **`Piece`** interface:
   - **`CornerPiece`**
   - **`DiagonalPiece`**
   - **`IPiece`**
   - **`LPiece`**
   - **`SnakePiece`**
2. Implement the abstract class **`AbstractPiece`** containing the common code for the concrete **`Piece`** classes above (optionally, you can implement additional abstract classes extending **`AbstractPiece`**, if you find that doing so improves your design).
3. Implement a class **`BlockAddiction`** extending the **`AbstractGame`** class, in which you provide implementations of the method **`determinePositionsToCollapse()`** and the constructors.
4. Finish the implementation of the **`BasicGenerator`** class

All of your code goes in the package **`hw4`**.

Overview

In this project you will complete the implementation of a Tetris-style or “falling blocks” type of video game. This particular game, which we’ll call **`BlockAddiction`**, is a sort of a mix of Tetris with a game like "Bejeweled". If you are not familiar with such games, examples are not hard to find on the internet.

The basic idea is as follows. The game is played on a 2D grid. Each position in this grid can be represented as a (*row, column*) pair. We typically represent these positions using the simple class **`api.Position`**, which you'll find in the **`api`** package. At any stage in the game, a grid position may be empty (null) or may be occupied by an *icon* (i.e., a colored block), represented by the class **`api.Icon`**. In addition, a *piece* or *shape* made up of a combination of icons falls from the top of the grid. This is referred to as the *current piece* or *current shape*. As it falls, the current piece can be

- **shifted** from side to side
- **transformed**, which may rotate or flip it or something else
- **cycled**, which will change the relative positions of the icons within the piece, without changing the cells it occupies

When the currently falling piece can’t fall any further, its icons are added to the grid, and the game checks whether it has completed a *collapsible set*. In **`BlockAddiction`**, a collapsible set is formed of three or more adjacent icons that match (have the same color). All icons in a collapsible set are then removed from the grid, and icons above them are shifted down an equivalent amount. (Note that there is no "gravity" and there can be empty spaces remaining below an icon in the grid.) The new icon positions may form new collapsible sets, so the game will iterate this process until there are no more collapsible sets. This behavior is already implemented in the class **`AbstractBlockGame`**, and your main task is to implement the abstract method **`determineCellsToCollapse()`**, which finds the positions in collapsible sets.

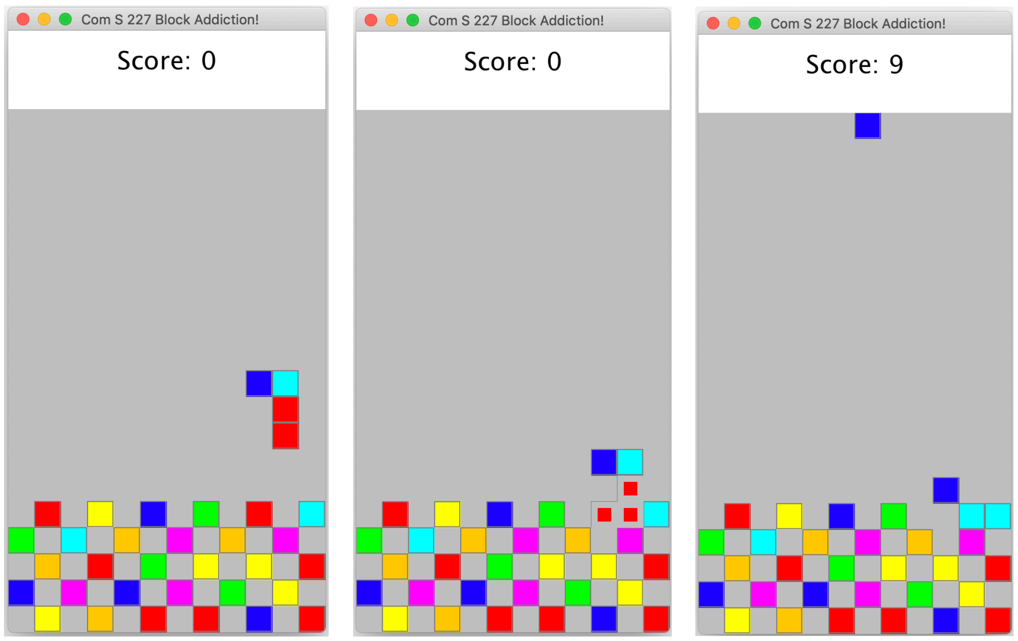

Com S 227 Block Addiction!
Score: 0
Com S 227 Block Addiction!
Score: 0
Com S 227 Block Addiction!
Score: 9

Figure 1: *An LPiece falling, and the collapsible set of three matching adjacent red icons disappears, and the blocks above them shift down.*

Specification

The specification for this assignment includes

- this pdf,
- any "official" clarifications posted on Piazza, and
- the online javadoc

The `Piece` interface and the five concrete `Piece` types

See the javadoc for the **`Piece`** interface.

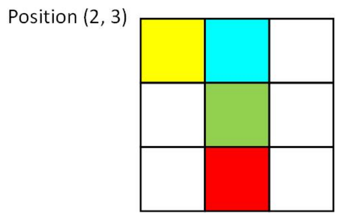
The currently falling piece is represented by an object that implements the **`Piece`** interface. Each piece has a state consisting of:

- The position in the grid of the upper-left corner of its bounding square, represented as an instance of **`Position`** (which can change as a result of calling methods such as **`shiftLeft()`**, **`shiftRight()`**, or **`transform()`**).

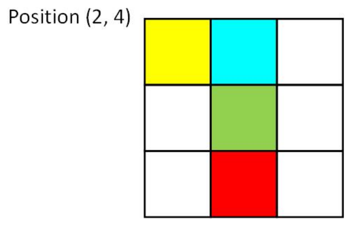
- The icons that make up the piece, along with their relative positions within the bounding square.

The position of a piece within a grid is always described by the upper left corner of its bounding square. Most importantly, there is a **`getCellsAbsolute()`** method that enables the caller to obtain the *actual* positions, within the grid, of the icons in the piece. The individual icons in the piece are represented by an array of **`Cell`** objects, a simple type that encapsulates a **`Position`** and an **`Icon`**. A **`Position`** is just a (row, column) pair, and an **`Icon`** just represents a colored block.

For example, one of the concrete **`Piece`** classes you will implement is call the **`LPiece`**. One is shown below in its initial (non-transformed) configuration. The method **`getCells()`** returns four cell objects with the positions relative to the upper left corner of the bounding square, namely (0, 0), (0, 1), (1, 1), and (2, 1), in that order, where the ordered pairs represent *(row, column)*, NOT *(x, y)*. (The colors are shown for illustration, and are normally assigned randomly by the *generator* for the game.) Suppose also that the piece's position (upper left corner of bounding square) is row 2, column 3. Then the **`getCellsAbsolute()`** method should return an array of four cell objects with positions (2, 3), (2, 4), (3, 4), and (4, 4), in that order.

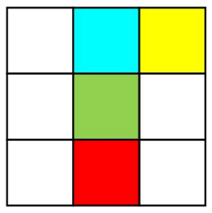


If the method **`shiftRight()`** is called on this piece, the position is updated, but **`getCells()`** would still return the same cells, since the positions are relative to the upper left corner. But the **`getCellsAbsolute()`** method would now return (2, 4), (2, 5), (3, 5), and (4, 5).



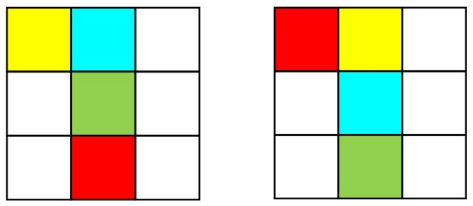
Each piece defines a particular behavior associated with the **`transform()`** operation. For the **`LPiece`**, the **`transform()`** operation just flips it across its vertical centerline. The position of the bounding square does not change, but the positions of the cells within the bounding square are modified, as shown below. This time the **`getCells()`** method should return an array of cells with positions (0, 2), (0, 1), (1, 1), and (2, 1), and the **`getCellsAbsolute()`** method should return an array of four cell objects with positions (2, 5), (2, 4), (3, 4), and (4, 4), in that order.

Position (2, 3)



Likewise, if the **`cycle()`** method is invoked, the positions for the cells stay the same but the icons associated with the cells will change. The illustration below shows the result of invoking **`cycle()`** on the first figure:

Position (2, 3)

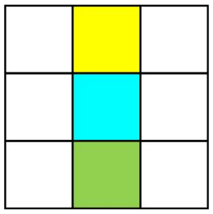


Each icon shifts forward to the next cell, and the last icon is placed in the first cell. (The ordering of the cells always the same, even if transformed.)

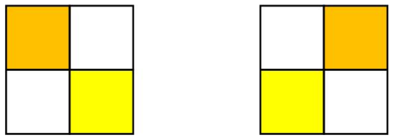
Altogether you will need to create five concrete classes, described below, that (directly or indirectly) extend **`AbstractPiece`** and, therefore, implement the **`Piece`** interface. It is up to you to decide how to design these classes, but a portion of your grade will be based on how well you use inheritance to avoid duplicated code. Remember that in the descriptions below, an ordered pair is *(row, column)*, NOT *(x, y)*:

1. The one illustrated above, called the **`LPiece`**. Initially the icons are at (0, 0), (0, 1), (1, 1), and (2, 1), in that order. The **`transform()`** method flips the cells across the vertical centerline.

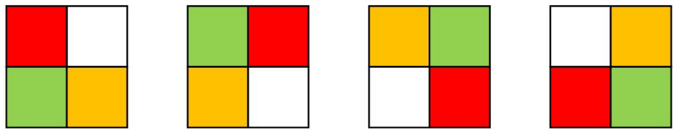
2. The **`IPiece`**, which has a 3 x 3 bounding square with the icons down the center in the order (0, 1), (1, 1), and (2, 1). For **`IPiece`**, the **`transform()`** method does nothing:



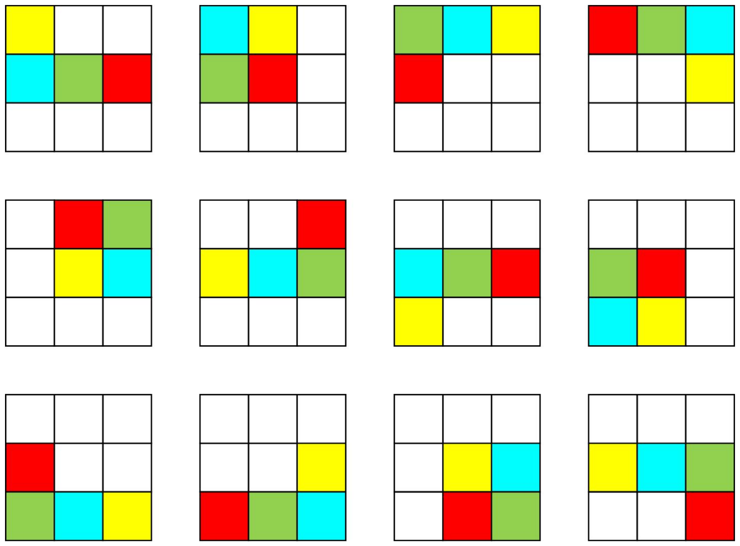
3. The **`DiagonalPiece`**, which has a 2 x 2 bounding square, shown below with its initial configuration on the left. The icon positions are in the order (0, 0), (1, 1). The **`transform()`** method flips the cells across the vertical centerline, as shown on the right (same as for the **`LPiece`**).



4. The **`CornerPiece`**, which has a 2 x 2 bounding square and initial cell positions (0, 0), (1, 0), and (1, 1) in that order. The **`transform()`** method performs a kind of rotation through four different states as shown below, e.g., after calling **`transform()`** once, the cell positions are (0, 1), (0, 0), and (1, 0), and after calling **`transform()`** four times they are back in the initial positions.



5. The **`SnakePiece`**, which has a 3 x 3 bounding square and initial cell positions (0, 0), (1, 0), (1, 1), and (1, 2) in that order. The **`transform()`** method transitions through twelve different states and back to the original, following a snake-like pattern as shown below (reading left-to-right and top-to-bottom):



When you implement these five concrete types, pay careful attention to code reuse, and implement common code in an abstract superclass called **`AbstractPiece`**. **Part of your score will be based on how well you have taken advantage of inheritence to reduce code duplication among the three concrete types.**

Each of these types has a constructor that takes an initial position and an array of **`Icon`** objects:

```
public CornerPiece(Position position, Icon[] icons)
public DiagonalPiece(Position position, Icon[] icons)
public IPiece(Position position, Icon[] icons)
public LPiece(Position position, Icon[] icons)
public SnakePiece(Position position, Icon[] icons)
```

The given icon objects are placed in the initial cells in the order given. Each constructor should throw **`IllegalArgumentException`** if the given array is not the correct length.

There are some additional notes below about implementing **`transform()`** for the **`SnakePiece`** and for implementing **`clone()`**.

The Game interface and AbstractGame class

See the javadoc for the **`Game`** interface.

The class **`AbstractGame`** is a partial implementation of the **`Game`** interface. A client such as the sample UI interacts with the game logic only through the interface **`Game`** and does not depend directly on the **`AbstractGame`** class or its subclasses. **`AbstractGame`** is a general framework for any number of Tetris-style games. It is specialized by implementing the abstract method:

`List<Position> determinePositionsToCollapse()`

Examines the grid and returns a list of positions to be collapsed.

(Remember that **`List<T>`** is the interface type that **`ArrayList<T>`** implements, so your method can return an **`ArrayList<Position>`**.)

The key method of **`Game`** is **`step()`**, which is called periodically by the UI to transition the state of the game. The **`step()`** method is fully implemented in **`AbstractGame`**, and it is not necessary for you to read it in detail unless you are interested. You will just need a basic understanding of how it interacts with the **`determinePositionsToCollapse()`** method that you will implement. In **`BlockAddiction`**, the task of **`determinePositionsToCollapse()`** is to identify the positions of icons in all "collapsible sets", that is, sets of three or more adjacent icons with matching color.

If you are interested: The way that **`determinePositionsToCollapse()`** is invoked from **`AbstractGame`** is the following. Whenever the current shape cannot fall any further, the **`step()`** method calls **`determinePositionsToCollapse()`**. If the returned list is nonempty, then the list is stored in the state variable **`positionsToCollapse`** and the game transitions into the COLLAPSING state. On the next call to **`step()`**, the method **`collapsePositions()`** of **`AbstractGame`** is invoked to actually perform the modification of the grid to remove the icons and shift down the icons above them. In many of these kinds of games, collapsing some positions may create additional collapsible sets of positions, possibly starting a chain reaction, so the logic of the COLLAPSING state is basically the following:

```
while (game state is COLLAPSING)
{
  collapsePositions(positionsToCollapse);
  positionsToCollapse = determinePositionsToCollapse();
  if (positionsToCollapse.size() == 0)
  {
    generate a new current shape
    change game state to NEW_SHAPE
  }
}
```

Remember, the class **`AbstractGame`** is fully implemented and you should not modify it.

The BlockAddiction class

You will create a subclass of **`AbstractGame`**, called **`BlockAddiction`**, that implements the game described in the introduction. The method **`determinePositionsToCollapse()`** must be declared **`public`** even though it is declared **`protected`** in **`AbstractGame`** (yes, Java allows this). This requirement is to make it easier to test your code.

There are two constructors declared as follows:

```
public BlockAddiction(int height, int width, Generator gen, int preFillRows)
public BlockAddiction(int height, int width, Generator gen)
```

(The second one is equivalent to calling the first one with zero for the fourth argument.)

The height, width, and generator are just passed to the superclass constructor. If **`preFillRows`** is greater than zero, your constructor should initialize the bottom **`preFillRows`** rows in a checkerboard pattern, using random icons obtained from the generator. The checkerboard pattern should place an icon at *(row, col)* in the grid if both *row* and *col* are even, or if both *row* and *col* are odd. See the illustration on page 3.

Implementing the method `determineCellsToCollapse()`

The method **`determineCellsToCollapse()`** is potentially tricky, and you can waste a lot of time on it if you don't first think carefully. A collapsible set is defined to be any set of three or more adjacent icons with the same color, so it could potentially contain many icons. Given an icon in a collapsible set, it is a hard problem to find *just* the cells that are part of that set. However, notice that **you do not have to solve that problem**. You have an easier problem, which is to return a list including *all* cells that are part of *any* collapsible set in the grid. What makes a cell part of a collapsible set? Well, either it has two or more neighbors that match its color, or else it must be a neighbor of such a cell. Therefore, it is enough to iterate over the grid and do the following for each array element:

If the element is non-null and has two or more neighbors that match, add it to the list, and also add all its matching neighbors to the list.

The list may contain many duplicates, which is not hard to deal with - just create a new list, copy the positions over, but ignore any that you have already found (the list method **`contains()`** is useful here). Finally, the list needs to be sorted. Fortunately, the **`Position`** class implements the **`Comparable`** interface so you can just call **`Collections.sort`** on your list.

The BasicGenerator class

See the javadoc for the **`Generator`** interface.

One important detail for the challenge and playability of the game is to configure what pieces are generated and the probability of getting each type. The role of the **`Generator`** interface is to make these features easy to configure. The **`AbstractGame`** constructor requires a **`Generator`** instance to be provided to its constructor. There is a partially implemented skeleton of the class

`BasicGenerator` class implementing the `Generator` interface. Ultimately, it should return one of the five concrete piece classes, selected according to the probabilities given in the javadoc comments. As you develop your Piece classes, you can just add statements to generate the ones you have completed and tested. (An easy way to generate pieces with varying probabilities: generate a random number from 0 to 100, and if it's in the range 0 up to 10 return an `LPiece`, if it's in range 11 up to 35 return a `DiagonalPiece`, and so on.) See the `BasicGenerator` javadoc comments for more details.

The UI

There is a graphical UI in the `ui` package. The controls are the following:

left arrow - shift the current falling piece left, if possible
right arrow - shift the current piece right, if possible
down arrow - make the current piece fall faster
up arrow - apply the transform() method
space bar - cycle the blocks within the piece
'p' key - pause/unpause

The main method is in `ui.GameMain`. You can try running it. It will start up a small instance of the partly-implemented class `example.SampleGame`. See the "Getting started" section for more details. To run the UI with your `BlockAddiction` game once you get it implemented, you'll need to edit `ui.GameMain`.

The UI is built on the Java Swing libraries. This code is complex and specialized, and is somewhat outside the scope of the course, but of course you are welcome to read it. *It is provided for fun, not for testing your code! It is not guaranteed to be free of bugs.*

Testing and the SpecChecker

As always, you should try to work incrementally and write tests for your code as you develop it. For the Piece hierarchy you can follow the examples in ExamplePieceTests.java in the sample code.

Do not rely on the UI code for testing! Trying to test your code using a UI is very slow, unreliable, and generally frustrating. *In particular, when we grade your work we are NOT going to run the UI, we are going to verify that each method works according to its specification.* **In the default package in the sample code there are some examples with ideas for how to test.**

We will provide a basic SpecChecker, but **it will not perform any functional tests of your code**. At this point in the course, you are expected to be able to read the specifications, ask questions when things require clarification, and write your own unit tests. 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.**

The SpecChecker will verify the class names and packages, the public method names and return types, and the types of the parameters. If your class structure conforms to the spec, you should see a message similar to this in the console output:

```
x out of x tests pass.
```

This SpecChecker will also offer to create a zip file for you that will package up the two required classes. Remember that your instance variables should always be declared **`private`**, and if you want to add any additional methods that are not specified, they must be declared **`private`** or **`protected`**.

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.

Special documentation and style requirements

- You may not use **`protected`**, **`public`** , or package-private instance variables. Normally, instance variables in a superclass should be initialized by an appropriately defined superclass constructor. You can create additional **`protected`** getter/setter methods if you really need them.

Style and documentation

See the special requirements above.

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

- 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**.
- 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.
  - **When a class implements or overrides a method that is already documented in the supertype (interface or class) you normally do not need to provide additional Javadoc,** unless you are significantly changing the behavior from the description in the supertype. You should include the `@Override` annotation to make it clear that the method was specified in the supertype.

- 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 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 `assignment4`. 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 `assignment4`. *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 "code" to have the editor keep your code formatting. You can also use "pre" for short code snippets.)

If you have a question that absolutely cannot be asked without showing part of your source code, change the visibility of the post to "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.

Getting started

General advice about using inheritance: It may not be obvious how to decide what code belongs in **`AbstractPiece`**. A good way to get started is to forget about inheritance at first. That is, pick one of the required concrete types, such as **`LPiece`**, add the clause **`implements Piece`** to its declaration, and just write the code for all the specified methods. (That is, temporarily forget about the requirement to extend **`AbstractPiece`**.) Then, choose another concrete type, and write all the code for that one. At this point you'll notice that you had to write lots of the same code twice. Move the duplicated code into **`AbstractPiece`**, and change the declaration for **`LPiece`** so it says **`extends AbstractPiece`** instead of **`implements Piece`**.

That's a good start, and you can use a similar strategy as you implement the other piece types. Remember you'll need to include one or more protected constructors for **`AbstractPiece`** in order to initialize it.

At this point we expect that you know how to study the documentation for a class, write test cases, and develop your code incrementally to meet the specification. There are some example tests provided with the Piazza homework links to give you some ideas about how to test your code. The comments should explain everything pretty well. You can work independently on the **`Piece`** classes and on the **`BlockAddiction`** class. The **`BasicGenerator`** is not directly needed until you want to actually try to play the game.

In addition, in the **`examples`** package you'll find a simplified, partial implementation of the **`Shape`** interface, called **`SamplePiece`**, and a simplified, partial subclass of **`AbstractGame`** called **`SampleGame`**. If you run the main class **`ui.GameMain`**, it will start up this game and you should see a two-block shape falling from the top. Try implementing the **`shiftLeft`** and **`shiftRight`** methods of **`SamplePiece`** to make the block shift from side to side with the arrow keys. That will be a rudimentary Tetris game.

To run the UI with your **`BlockAddiction`** game once you get it implemented, you'll need to edit the **`create`** method of **`ui.GameMain`**.

Implementing transform() for `SnakePiece`

Please do not just write twelve different cases! There are a few ways to approach this, as always, but here is one idea. First, imagine that there is just one cell that starts at (0, 0) and the transform method should shift it to (0, 1), then to (0, 2), then to (1, 0), and so on (picture the yellow icon in the illustration above). Rather that using a bunch of if-statements, you can just create an array of **`Position`** objects in the order you want:

```
private static final Position[] sequence =
{
  new Position(0, 0),
  new Position(0, 1),
  new Position(0, 2),
  new Position(1, 2),
  new Position(1, 1),
  new Position(1, 0),
  new Position(2, 0),
  new Position(2, 1),
  new Position(2, 2),
  new Position(1, 2),
  new Position(1, 1),
  new Position(1, 0),
};
```

and then maintain a counter that goes from 0 to 11 and wraps around to 0. At each call to **`transform()`**, update the counter and set the cell position using the counter as an index into your **`sequence`** array. How about the remaining three cells? Notice that the positions you want for them are always the three preceding ones in the **`sequence`** array, where again, if your index goes down from 0 you wrap around to 11.

(And here is an interesting challenge for you: could you reuse this same strategy for **`CornerPiece`**?)

Note about the `clone()` method

The clone() method is also discussed in "Special Topic 10.2" of the textbook.

TL;DR In most cases you can just copy and paste the **`clone()`** method from **`SamplePiece`** into **`AbstractPiece`**, changing the **`SamplePiece`** cast to **`AbstractPiece`**.

A **`Piece`** must have a **`clone()`** operation that returns a deep copy of itself. This is actually a critical part of the implementation, since the mechanism that the **`AbstractGame`** uses to detect whether a shift or transform is possible is by cloning the shape, performing the shift or transform first on the clone, and checking whether it overlaps other blocks or extends outside the grid.

The method **`clone()`** is defined in **`java.lang.Object`** as follows:

```
protected Object clone() throws CloneNotSupportedException
```

The default implementation of **`clone()`** makes a copy of the object on which it's invoked, and although it is declared to return type **`Object`**, it always creates an object of the same runtime type. However, it is a "shallow" copy, i.e., it just copies the instance variables, not the objects they refer to. Therefore, in order to use it, you have to add your own code to deep-copy the objects that your instance variables refer to (for the **`Shape`** classes, you might have a **`Position`** and an array of **`Cell`**s, for example). In addition, due to some technical nonsense we have to put the call to **`clone()`** in a "try/catch" block. We have not seen try/catch yet, but do not worry, you just have to follow the pattern below, as seen in **`examples.SampleShape`**:

```
@Override
public Piece clone()
{
  try
  {
    // call the Object clone() method to create a shallow copy...
    SamplePiece s = (SamplePiece) super.clone();

    // ...then make it into a deep copy
    // (note there is no need to copy the position,
    // since Position is immutable, but we have to deep-copy the cell array
    // by making new Cell objects
    s.cells = new Cell[cells.length];
    for (int i = 0; i < cells.length; ++i)
    {
      s.cells[i] = new Cell(cells[i]);
    }
    return s;
  }
  catch (CloneNotSupportedException e)
  {
    // can't happen, since we know the superclass is cloneable
    return null;
  }
}
```

In most cases, you'll implement **`AbstractPiece`** using instance variables for the position and cell array as in **`SamplePiece`**, and in that case you can just copy and paste the **`clone()`** method, changing the **`SamplePiece`** cast to **`AbstractPiece`**. Basically, you need to make *copies* of any mutable objects that your instance variables refer to, so that the original and the clone don't share references to the same objects. So if you have added more reference variables to **`AbtractPiece`**, consider whether they need to be copied when you make a clone.

The `clone()` mechanism in Java tends to be a fragile and often trouble-prone, and it only works on objects that are specifically implemented with cloning in mind. Java programmers tend to avoid using `clone()` unless there is a good reason. Notice that in the example above, we're not using `clone()` to make a copy of the the `Cell` - it's simpler just to use the copy constructor. But for the piece classes, by taking advantage of the built-in `clone()` mechanism you can implement `clone()` just once in the superclass and allow the subclasses to inherit it. (Of course, if any of your subclasses have any additional, non-primitive instance variables beyond those defined in your superclass, you'll need to override `clone()` again for them in order to get a deep copy.)

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_hw4.zip`**. and it will be located in whatever directory you selected when you ran the SpecChecker. It should contain one directory, **`hw4`**, which in turn contains a minimum of eight files, including:

```
AbstractPiece.java
BasicGenerator.java
BlockAddiction.java
CornerPiece.java
DiagonalPiece.java
IPiece.java
LPiece.java
SnakePiece.java
```

(plus any other abstract class you've added to the hierarchy).

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

Submit the zip file to Canvas using the Assignment 4 submission link and verify that your submission was successful. If you are not sure how to do this, see the document "Assignment Submission HOWTO", which can be found in the Piazza pinned messages under “Syllabus, office hours, useful links.”

> We recommend that you submit the zip file as created by the specchecker. If necessary for some reason, you can create a zip file yourself. The zip file must contain the directory **hw4**, which in turn should contain the the required .java files. You can accomplish this by zipping up the **src** directory within 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.

Labs for the course can be found here: http://web.cs.iastate.edu/~cs227/labs/