Problem Set 0: Turtle Graphics

Welcome to 6.005!

This course is about three essential properties of software:

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

The purpose of this problem set is to:

- introduce the tools we will use in 6.005, including Java, Eclipse, JUnit, and Git;
- introduce the process for 6.005 problem sets;
- and practice basic Java and start using tools from the Java standard library.

You should focus in this problem set on writing code that is **safe from bugs** and **easy to understand** using the techniques we discuss in class.

Part I

Problem 0: Install and set up

Read and complete the **Getting Started guide (../../getting-started/)** . The guide will step through:

- installing the JDK, Eclipse, and Git
- configuring Eclipse
- configuring Git
- learning and practicing the basics of Git

You need to complete all the steps in the guide before you start working on this problem set.

Problem 1: Initialize the Git Repo

This process will be identical for each problem set.

1. **Initialize your repo.**

   Download the project code here (ps0.zip) . In the terminal, navigate to the folder containing the project code and run:

   ```
   git init
   ```

2. After cloning your repository, **add the project to Eclipse** so you can work on it.

   Note: Getting Started step 2 (../../getting-started/#config-eclipse) has setup you must perform before using Eclipse in 6.005.

   To import a project:

   - In Eclipse, go to *File → Import… → Git → Projects from Git*
   - On "Select Repository Source" page, select "Existing local repository"
   - On "Select a Git Repository," click Add…, and Browse… to the directory of your clone
   - The "Search results" list should show your clone, with ".git" at the end; click "Finish"
   - On "Select a wizard" for importing, choose "Import existing projects"
   - Finally, on "Import projects," make sure ps0 is checked, and click "Finish"

Problem 2: Warm up with `mayUseCodeIn-Assignment`

a. Look at the source code contained in `RulesOf6005.java` in package `rules` . Your warm-up task is to implement:

```
mayUseCodeInAssignment(
       boolean writtenByYourself, boolean availableToOthers,
       boolean writtenAsCourseWork, boolean citingYourSource,
       boolean implementationRequired)
```

You can find the policy under General Information (../../general/) on the course home page (../../) .

b. Once you've implemented this method, run the `main` method in `RulesOf6005.java` .

`public static void main(String[] args)` is the entry point for Java programs. In this case, the `main` method calls the `mayUseCodeInAssignment` method with input parameters. To run `main` in `RulesOf6005` , right click on the file `RulesOf6005.java` in either your Package

Explorer, Project View, or Navigator View, go to the *Run As* option, and click on *Java Application* .

Unit testing

Right now, we can use the `main` method plus some visual inspection to verify that our implementation is correct. More generally, programs will have many dozens of methods that need to be tested; visually inspecting output for each one is fragile, time-consuming, and inherently non-scalable.

Instead, we will use *automated unit testing* , which runs a suite of tests to automatically test whether the implementations are correct. For this problem set, we will write unit tests for methods that do not draw graphics on the screen; unit-testing GUIs is a more complex problem.

Automated unit testing with JUnit

JUnit (//www.junit.org/) is a widely-adopted Java unit testing library, and we will use it heavily in 6.005. A major component of the 6.005 design philosophy is to decompose problems into minimal, orthogonal units, which can be assembled into the larger modules that form the finished program. One benefit of this approach is that each unit can be tested thoroughly, independently of others, so that faults can be quickly isolated and corrected as code is rewritten and modules are configured. Unit testing is the technique of writing tests for the smallest testable pieces of functionality, to allow for the flexible and organic evolution of complex, correct systems.

By writing thoughtful unit tests, it is possible to verify the correctness of one's code, and to be confident that the resulting programs behave as expected. In 6.005, we will use JUnit version 4.

Anatomy of JUnit

JUnit unit tests are written method by method. There is nothing special a class has to do to be used by JUnit; it only need contain methods that JUnit knows to call, which we call *test methods* . Test methods are specified using *annotations* , which may be thought of as keywords (more specifically, they are a type of metadata), that can be attached to individual methods and classes. Though they do not themselves change the meaning of a Java program, at compile- or run-time other code can detect the annotations and make decisions accordingly. Though we will not deeply explore annotations in 6.005, you will see a few important uses of them.

Look closely at `RulesOf6005Test.java` , and note the `@Test` that precedes method definitions. This is an example of an annotation. The JUnit library uses this annotation to determine which methods to call when running unit tests. The `@Test` annotation denotes a test method; there can be any number in a single class. Even if one test method fails, the others will be run.

Unit test methods can contain calls to `assertEquals` , which is an assertion that compares two objects against each other and fails if they are not equal, `assertTrue` , which checks if the condition is true, and `assertFalse` , which checks if the condition is false. Here is a list of all the assertions supported by JUnit (//junit.org/javadoc/latest/index.html?org/junit/Assert.html) . If an assertion in a test method fails, that test method returns immediately, and JUnit records a failure for that test.

c. Run the unit tests.

To run the tests in `RulesOf6005Test` , right click on the `RulesOf6005Test.java` file in either your Package Explorer, Project View, or Navigator View, and go to the *Run As* option. Click on *JUnit Test* , and you should see the JUnit view appear.

If your implementation of `mayUseCodeInAssignment` is correct, you should see a green bar, indicating that all the tests (there's only 1 test, containing 2 assertions) passed.

d. Try *breaking* your implementation and running `RulesOf6005Test` again.

   You should see a red bar in the JUnit view, and if you click on `testMayUseCodeInAssignment`, you will see a *stack trace* in the bottom box, which provides a brief explanation of what went wrong. Double-clicking on a line in the failure stack trace will bring up the code for that frame in the trace. This is most useful for lines that correspond to your code; this stack trace will also contain lines for Java libraries or JUnit itself.

e. Enough breaking: fix your implementation so it's correct again. Make sure the tests pass.

   Passing the JUnit tests we provide does **not** necessarily mean that your code is perfect. You need to review the function specifications carefully, and **always write your own JUnit tests** to verify your code.

Problem 3: Commit and push `RulesOf6005`

After you've finished implementing the function and verified that it is correct, let's do a first commit.

1. First, in the terminal, change directory (`cd`) to your clone, and take a look around with

   ```
   git status
   ```

   which shows you files that have been created, deleted, and modified in the project directory. You should see `RulesOf6005.java` listed under "Changes not staged for commit." This means Git sees the change, but you have not (yet) asked Git to include the change as part of your next commit.

2. You can run the command

   ```
   git diff
   ```

   to see your changes. (**Note** : when the diff is more than one page long, use the arrow keys. Press `q` to quit the diff.)

3. Before committing, files must be *staged* for commit. Staging a file is as simple as

   ```
   git add <filename>
   ```

   so use

   ```
   git add src/rules/RulesOf6005.java
   ```

   to stage the file. You should also stage the test file if you've added more tests.

4. In addition, it's always a good idea to review your commits before committing to them. Run

   ```
   git status
   ```

   again to see that your changes are now listed under "Changes to be committed." If you run

   ```
   git diff
   ```

   those changes are no longer shown! Use

```
git diff --staged
```

to see exactly what Git will record if you commit now.

5. Ready? To perform the commit,

```
git commit
```

will actually commit the changes locally, after opening your default editor to allow you to write a commit message. Your message should be formatted according to the Git standard (//git-scm.com/book/ch5-2.html#Commit-Guidelines) : a short summary that fits on one line, followed by a blank line and a longer description if necessary.

If Git warns you about configuring your default identity or you can't edit your commit message , you did not follow the instructions in the Getting Started guide. Getting Started step 5 (../../getting-started/#config-git) has setup you must perform before using Git.

Now run

```
git status
```

once more, and see that your changes are no longer listed.

6. You can use the command

```
git log
```

to see the history of commits in your project. Right now, you should see two of them: the initial commit to create your problem set repository with the starting code, and the commit you made just now.

Important: only the local history has the new commit at this point; it is not stored in your remote repository. This is one important aspect where Git is different from centralized systems such as Subversion and CVS.

Part II

Turtle graphics and the Logo language

Logo (//en.wikipedia.org/wiki/Logo_%28programming_language%29) is a programming language created at MIT that originally was used to move a robot around in space. Turtle graphics, added to the Logo language, allows programmers to issue a series of commands to an on-screen "turtle" that moves, drawing a line as it goes. Turtle graphics have also been added to many different programming languages, including Python (//docs.python.org/2/library/turtle.html) , where it is part of the standard library.

In the rest of problem set 0, we will be playing with a simple version of turtle graphics for Java that contains a restricted subset of the Logo language:

- `forward(units)`
  Moves the turtle in the current direction by *units* pixels, where units is an integer. Following the original Logo convention, the turtle starts out facing up.

- `turn(degrees)`
  Rotates the turtle by angle *degrees* to the right (clockwise), where degrees is a double precision floating point number.

You can see the definitions of these commands in `Turtle.java`.

Do NOT use any turtle commands other than `forward` and `turn` in your code for the following methods.

Problem 4: `drawSquare`

Look at the source code contained in `TurtleSoup.java` in package `turtle`.

Your task is to implement `drawSquare(Turtle turtle, int sideLength)`, using the two methods introduced above: `forward` and `turn`.

Once you've implemented the method, run the `main` method in `TurtleSoup.java`. The `main` method in this case simply creates a new turtle, calls your `drawSquare` method, and instructs the turtle to draw. Run the method by going to *Run → Run As… → Java Application*. A window will pop up, and, once you click the "Run!" button, you should see a square drawn on the canvas.

Problems 5—10: Polygons and headings

For detailed requirements, read the specifications of each function to be implemented above its declaration in `TurtleSoup.java`. Be careful when dealing with mixed integer and floating point calculations.

You should not change any of the *method declarations* (what's a declaration? (//docs.oracle.com/javase/tutorial/java/javaOO/methods.html)) below. If you do so, you risk receiving **zero points** on the problem set.

Drawing polygons

- Implement `calculateRegularPolygonAngle`
  There's a simple formula for what the inside angles of a regular polygon should be; try to derive it before googling/binging/duckduckgoing.
- Run the JUnit tests in `TurtleSoupTest`
  The method that tests `calculateRegularPolygonAngle` should now pass and show green instead of red.

  If `testAssertionsEnabled` fails, you did not follow the instructions in the Getting Started guide. Getting Started step 2 (../../getting-started/#config-eclipse) has setup you must perform before using Eclipse.
- Implement `calculatePolygonSidesFromAngle`
  This does the inverse of the last function; again, use the simple formula. However, **make sure you correctly round** to the nearest integer. Instead of implementing your own rounding, look at Java's `java.lang.Math` (//docs.oracle.com/javase/8/docs/api/?java/lang/Math.html) class for the proper function to use.
- Implement `drawRegularPolygon`
  Use your implementation of `calculateRegularPolygonAngle`. To test this, change the `main` method to call `drawRegularPolygon` and verify that you see what you expect.

Calculating headings

- Implement `calculateHeadingToPoint`
  This function calculates the parameter to `turn` required to get from a current point to a target point, with the current direction as an additional parameter. For example, if the turtle is at (0,1) facing 30 degrees, and must get to (0,0), it must turn an additional 150 degrees, so `calculateHeadingToPoint(30, 0, 1, 0, 0)` would return `150.0`.
- Implement `calculateHeadings`
  Make sure to use your `calculateHeadingToPoint` implementation here. For information on how to use Java's `List` interface and classes implementing it, look up `java.util.List` (//docs.oracle.com/javase/8/docs/api/?java/util/List.html) in the Java library documentation. Note that for a list of *n* points, you will return *n-1* heading adjustments; this list of adjustments could be used to guide the turtle to each point in the list. For example, if the input lists consisted of `xCoords=[0,0,1,1]` and `yCoords=[1,0,0,1]` (representing points (0,1), (0,0), (1,0), and (1,1)), the returned list would consist of `[180.0, 270.0, 270.0]`.

Problem 11: Personal art

- Implement `drawPersonalArt`
  In this function, you have the freedom to draw any piece of art you wish. Your work will be judged both on aesthetics and on the code used to draw it. Your art doesn't need to be complex, but it should be more than a few lines. Use helper methods, loops, etc. rather than simply listing forward and turn commands.

  For `drawPersonalArt` only, you may also use the `color` method of `Turtle` to change the pen color. You may only use the provided colors.

  Here are some examples (https://www.google.com/search?q=python+turtle+example+images) of the kinds of images you can generate procedurally with turtle graphics, though note that many of them use more commands than what we've provided here. Modify the `main` method to see the results of your function.

MIT EECS

6.005 — Software Construction on MIT OpenCourseWare (https://ocw.mit.edu/ans7870/6/6.005/s16/) | OCW 6.005 Homepage (https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/) **Spring 2016**

Problem Set 1: Tweet Tweet

The purpose of this problem set is to give you practice with test-first programming. Given a set of specifications, you will write unit tests that check for compliance with the specifications, and then implement code that meets the specifications.

Get the code

To get started, download the assignment code (ps1.zip) and initialize your repository. If you need a refresher on how to create your repository, see Problem Set 0 (../ps0/#clone) .

Overview

The theme of this problem set is to build a toolbox of methods that can extract information from a set of tweets downloaded from Twitter.

Since we are doing test-first programming, your workflow for each method should be (*in this order*).

1. Study the specification of the method carefully.
2. Write JUnit tests for the method according to the spec.
3. Implement the method according to the spec.
4. Revise your implementation and improve your test cases until your implementation passes all your tests.

On PS0, we graded only your method implementations. On this problem set, we will also grade the tests you write. In particular:

- **Your test cases should be chosen using the input/output-space partitioning approach** . This approach is explained in the reading about testing (../../classes/03-testing/) .
- **Include a comment at the top of each test suite class describing your *testing strategy*** – how you partitioned the input/output space of each method, and then how you decided which test cases to choose for each partition.
- **Your test cases should be small and well-chosen.** Don't use a large set of tweets from Twitter for each test. Instead, create your own artificial tweets, carefully chosen to test the partition you're trying to test.

- **Your tests should find bugs.** We will grade your test cases in part by running them against buggy implementations and seeing if your tests catch the bugs. So consider ways an implementation might inadvertently fail to meet the spec, and choose tests that will expose those bugs.
- **Your tests must be legal clients of the spec.** We will also run your test cases against legal, variant implementations that still strictly satisfy the specs, and your test cases should not complain for these good implementations. That means that your test cases can't make extra assumptions that are only true for your own implementation.
- **Put each test case in its own JUnit method.** This will be far more useful than a single large test method, since it pinpoints where the problem areas lie in the implementation.
- Again, keep your tests small. Don't use unreasonable amounts of resources (such as `MAX_INT` size lists). We won't expect your test suite to catch bugs related to running out of resources; *every* program fails when it runs out of resources.

You should also keep in mind these facts from the readings and classes about specifications (part 1 (../../classes/06-specifications/) , part 2 (../../classes/07-designing-specs/)):

- **Preconditions.** Some of the specs have preconditions, e.g. "this value must be positive" or "this list must be nonempty". When preconditions are violated, the behavior of the method is *completely unspecified* . It may return a reasonable value, return an unreasonable value, throw an unchecked exception, display a picture of a cat, crash your computer, etc., etc., etc. In the tests you write, do not use inputs that don't meet the method's preconditions. In the implementations you write, you may do whatever you like if a precondition is violated. Note that if the specification indicates a particular exception should be thrown for some class of invalid inputs, that is a *postcondition* , not a precondition, and you *do* need to implement and test that behavior.
- **Underdetermined postconditions.** Some of the specs have underdetermined postconditions, allowing a range of behavior. When you're implementing such a method, the exact behavior of your method within that range is up to you to decide. When you're writing a test case for the method, you must allow the implementation you're testing to have the full range of variation, because otherwise your test case is not a legal client of the spec as required above.

Finally, in order for your overall program to meet the specification of this problem set, you are required to keep some things unchanged:

- **Don't change these classes at all:** the classes `Tweet` , `Timespan` , and `TweetReader` should not be modified *at all* .
- **Don't change these class names:** the classes `Extract` , `Filter` , `SocialNetwork` , `ExtractTest` , `FilterTest` , and `SocialNetworkTest` must use those names and remain in the `twitter` package.
- **Don't change the method signatures and specifications:** The public methods provided for you to implement in `Extract` , `Filter` , and `SocialNetwork` must use the method signatures and the specifications that we provided.
- **Don't include illegal test cases:** The tests you implement in `ExtractTest` , `FilterTest` , and `SocialNetworkTest` must respect the specifications that we provided for the methods you are testing.

Aside from these requirements, however, you are free to add new public and private methods and new public or private classes if you wish. In particular, if you wish to write test cases that test a stronger spec than we provide, you should put those tests in a separate JUnit test class, so that we don't try to run them on staff implementations that only satisfy the weaker spec. We suggest naming those test classes `MyExtractTest` , `MyFilterTest` , `MySocialNetworkTest` , and we suggest putting them in the `twitter` package in the `test` folder alongside the other JUnit test classes.

Problem 1: Extracting data from tweets

In this problem, you will test and implement the methods in `Extract.java`.

You'll find `Extract.java` in the `src` folder, and a JUnit test class `ExtractTest.java` in the `test` folder. Separating implementation code from test code is a common practice in development projects. It makes the implementation code easier to understand, uncluttered by tests, and easier to package up for release.

a. Devise, document, and implement test cases for `getTimespan()` and `getMentionedUsers()`, and put them in `ExtractTest.java`.

b. Implement `getTimespan()` and `getMentionedUsers()`, and make sure your tests pass.

If you want to see your code work on a live sample of tweets, you can run `Main.java`. (`Main.java` will not be used in grading, and you are free to edit it as you wish.)

Hints:

- Note that we use the class `Instant` (//docs.oracle.com/javase/8/docs/api/?java/time/Instant.html) to represent the date and time of tweets. You can check this article on Java 8 dates and times (//java.dzone.com/articles/deeper-look-java-8-date-and) to learn how to use `Instant`.
- You may wonder what to do about lowercase and uppercase in the return value of `getMentionedUsers()`. This spec has an underdetermined postcondition, so read the spec carefully and think about what that means for your implementation and your test cases.
- `getTimespan()` *also* has an underdetermined postcondition in some circumstances, which gives the implementor (you) more freedom and the client (also you, when you're writing tests) less certainty about what it will return.
- Read the spec for the `Timespan` class carefully, because it may answer many of the questions you have about `getTimespan()`.

Commit to Git. Once you're happy with your solution to this problem, commit to your repo! Committing frequently – whenever you've fixed a bug or added a working and tested feature – is a good way to use version control, and will be a good habit to have for your team projects.

Problem 2: Filtering lists of tweets

In this problem, you will test and implement the methods in `Filter.java`.

a. Devise, document, and implement test cases for `writtenBy()`, `inTimespan()`, and `containing()`, and put them in `FilterTest.java`.

b. Implement `writtenBy()`, `inTimespan()`, and `containing()`, and make sure your tests pass.

Hints:

- For questions about lowercase/uppercase and how to interpret timespans, reread the hints in the previous question.
- For all problems on this problem set, you are free to rewrite or replace the provided example tests and their assertions.

Commit to Git. Once you're happy with your solution to this problem, commit to your repo!

Problem 3: Inferring a social network

In this problem, you will test and implement the methods in `SocialNetwork.java`. The `guessFollowsGraph()` method creates a social network over the people who are mentioned in a list of tweets. The social network is an approximation to who is following whom on Twitter, based only on the evidence found in the tweets. The `influencers()` method returns a list of people sorted by their influence (total number of followers).

a. Devise, document, and implement test cases for `guessFollowsGraph()` and `influencers()`, and put them in `SocialNetworkTest.java`. Be careful that your test cases for `guessFollowsGraph()` respect its underdetermined postcondition.

b. Implement `guessFollowsGraph()` and `influencers()`, and make sure your tests pass. For now, implement only the minimum required behavior for `guessFollowsGraph()`, which infers that Ernie follows Bert if Ernie @-mentions Bert.

If you want to see your code work on a live sample of tweets, run `Main.java`. It will print the top 10 most-followed people according to the social network you generated. You can search for them on Twitter to see if their actual number of followers has a similar order.

Problem 4: Get smarter

In this problem, you will implement one additional kind of evidence in `guessFollowsGraph()`. Note that we are taking a broad view of "influence" here, and even Twitter-following is not a ground truth for influence, only an approximation. It's possible to read Twitter without explicitly following anybody. It's also possible to be influenced by somebody through other media (email, chat, real life) while producing evidence of the influence on twitter.

Here are some ideas for evidence of following. Feel free to experiment with your own.

- **Common hashtags.** People who use the same hashtags in their tweets (e.g. `#mit`) may mutually influence each other. People who share a hashtag that isn't otherwise popular in the dataset, or people who share multiple hashtags, may be even stronger evidence.
- **Triadic closure (//en.wikipedia.org/wiki/Triadic_closure) .** In this context, triadic closure means that if a strong tie (mutual following relationship) exists between a pair A,B and a pair B,C, then some kind of tie probably exists between A and C – either A follows C, or C follows A, or both.
- **Awareness** . If A follows B and B follows C, and B retweets a tweet made by C, then A sees the retweet and is influenced by C.

Keep in mind that whatever additional evidence you implement, your `guessFollowsGraph()` must still obey the spec. To test your specific implementation, make sure you put test cases in your own `MySocialNetworkTest` class rather than the `SocialNetworkTest` class that we will run against staff implementations. Your work on this problem will be judged by the clarity of the code you wrote to implement it and the test cases you wrote to test it.

MIT EECS

Problem Set 2: Poetic Walks

The purpose of this problem set is to practice designing, testing, and implementing abstract data types. This problem set focuses on implementing mutable types where the specifications are provided to you; the next problem set will focus on immutable types where you must choose specifications.

Design Freedom and Restrictions

On several parts of this problem set, the classes and methods will be yours to specify and create, but you must pay attention to the **PS2 instructions** sections in the provided documentation.

You must satisfy the specifications of the provided interfaces. In some cases, you are permitted to **strengthen** the provided specifications or **add** new methods, but in other cases you are **not permitted to change them at all** .

Get the code

To get started, download the assignment code (ps2.zip) and initialize a repository. If you need a refresher on how to create your repository, see Problem Set 0 (../ps0/#clone) .

Overview

The Java Collections Framework (//docs.oracle.com/javase/tutorial/collections/index.html) provides many useful data structures for working with collections of objects: lists (//docs.oracle.com/javase/tutorial/collections/interfaces/list.html) , maps (//docs.oracle.com/javase/tutorial/collections/interfaces/map.html) , queues (//docs.oracle.com/javase/tutorial/collections/interfaces/queue.html) , sets (//docs.oracle.com/javase/tutorial/collections/interfaces/set.html) , and so on. It does not provide a **graph** (https://en.wikipedia.org/wiki/Graph_(abstract_data_type)) data structure. Let's implement graphs, and then use them to create timeless art of unparalleled brilliance.

Problems 1-3: the type we will implement is **`Graph<L>`** , an abstract data type for mutable weighted (https://en.wikipedia.org/wiki/Glossary_of_graph_theory#Weighted_graphs_and_networks) directed graphs (https://en.wikipedia.org/wiki/Directed_graph) with labeled vertices.

mutable graph	vertices and edges may be added to and removed from the graph
directed edges	edges go from a source vertex to a target vertex
weighted edges	edges are associated with a positive integer weight
labeled vertices	vertices are distinguished by a label of some immutable type, for example they might have `String` names or `Integer` IDs

Read the **Javadoc documentation for `Graph`** (doc/index.html?graph/Graph.html) generated from `src/graph/Graph.java` .

`Graph<L>` is a generic type (https://docs.oracle.com/javase/tutorial/java/generics/types.html) similar to `List<E>` (//docs.oracle.com/javase/8/docs/api/?java/util/List.html) or `Map<K,V>` (//docs.oracle.com/javase/8/docs/api/?java/util/Map.html) . In the specification of `List` , `E` is a placeholder for the type of elements in the list. In `Map` , `K` and `V` are placeholders for the types of keys and values. Only later does the client of `List` or `Map` choose particular types for these placeholders by constructing, for example, a `List<Clown>` or a `Map<Car,Integer>` .

The specification of a generic type is written in terms of the placeholders. For example, the specification of `Map<K,V>` says that `K` must be an immutable type: if you want to make something the key in a `Map`, it shouldn't be a mutable object because the `Map` may not work correctly.

In our specification for `Graph<L>`, we make the same demand about the immutability of type `L`. Clients of `Graph` who try to use mutable vertex labels have violated the precondition. They cannot expect correct behavior.

For this problem set, we will implement `Graph` twice, with two different reps, to practice choosing abstraction functions and rep invariants and preventing rep exposure. There are many good reasons for a library to provide multiple implementations of a type (for example, the `ArrayList` (//docs.oracle.com/javase/8/docs/api/?java/util/ArrayList.html) and `LinkedList` (//docs.oracle.com/javase/8/docs/api/?java/util/ArrayList.html) implementations of `List` satisfy clients with different performance requirements), and we're following that model.

Problem 4: with our graph datatype in hand, we will implement **`GraphPoet`**, a class for generating poetry using a word affinity graph.

Our poet will be able to take an input like this phrase from Tolkien (https://books.google.com/books?id=aWZzLPhY4o0C&pg=PA232&dq=%22They+spoke+no+more+of+the+small+news+of+the+Shire+far+away%22) :

> They spoke no more of the small news of the Shire far away, nor of the dark shadows and perils that encompassed them, but of the fair things they had seen in the world together

and generate, with help from The Bard (//shakespeare.mit.edu/Poetry/sonnets.html) , a stanza addressed to some mysterious figure:

> They spoke no *love,* more *than* of *all* the small news of *all* the Shire far away,
> nor *thou* of *all* the dark shadows and perils that encompassed them,
> but *love* of *all* the *outward* fair *in* things they had *not* seen in *all* the *wide* world together

Read the **Javadoc documentation for `GraphPoet`** (doc/index.html?poet/GraphPoet.html) generated from `src/poet/GraphPoet.java`.

`GraphPoet` will come in handy when you take 21W.772 (//student.mit.edu/catalog/m21Wb.html#21W.772) next semester.

Problem 1: Test `Graph<String>`

Devise, document, and implement tests for `Graph<String>`.

For now, we'll only test (and then implement) graphs with `String` vertex labels. Later, we'll expand to other kinds of labels.

In order to accommodate running our tests on *multiple implementations* of the `Graph` interface, here is the setup:

- The testing strategy and tests for the **static** `Graph.empty()` method are in **`GraphStaticTest.java`**. Since the method is static, there will be only one implementation, and we only need to run these tests once. We've provided these tests. You are free to change or add to them, but you can leave them as-is for this problem.
- Write your testing strategy and your tests for all the **instance** methods in **`GraphInstanceTest.java`**. In these tests, you must use the `emptyInstance()` method to get fresh empty graphs, *not `Graph.empty()`!* See the provided `testInitialVerticesEmpty()` for an example.

Unlike `GraphStaticTest`, which works like any other JUnit test class we've written, `GraphInstanceTest` is different because you can't run it directly. It has a blank waiting to be filled in: the `emptyInstance()` method that will provide empty `Graph` objects. In the next problem, we'll see two *subclasses (https://docs.oracle.com/javase/tutorial/java/landl/subclasses.html)* of `GraphInstanceTest` that fill in the blank by returning empty graphs of different types.

Your tests in `GraphInstanceTest` must be legal clients of the `Graph` spec. Any tests specific to your implementations will go in the subclasses in the next problem.

Java note

`GraphInstanceTest` is an abstract class (https://docs.oracle.com/javase/tutorial/java/landl/abs. Abstract classes and subclassing (https://docs.oracle.com/javase/tutorial/java/landl/sub have their uses, but in general should be avoided.

Commit to Git. Once you're happy with your solution to this problem, commit to your repo!

Problem 2: Implement `Graph<String>`

Now we'll implement weighted directed graphs with `String` labels — **twice**.

For **all** the classes you write in this problem set:

- **Document the abstraction function and representation invariant**.
- Along with the rep invariant, **document how the type prevents rep exposure**.
- **Implement `checkRep`** to check the rep invariant.
- **Implement `toString`** with a useful human-readable representation of the abstract value.

In the future, every immutable type you write will override `equals` and `hashCode` to implement the `Object` contract as described in class 15 on equality (equality-todo.html#../../classes/15-equality/) , but that's **not** required for this problem set.

All your classes must have clear **documented specifications** . This means every method will have a Javadoc comment, except when you use `@Override` :

- When a method is specified in an interface and then implemented by a concrete class — for example, `add(..)` in `ConcreteEdgesGraph` is specified in `Graph` — **the `@Override` (//docs.oracle.com/javase/tutorial/java/annotations/predefined.html) annotation with no Javadoc comment indicates the method has the same spec** . Unless the concrete class needs to strengthen the spec, don't write it again (DRY).

You can choose either `ConcreteEdgesGraph` or `ConcreteVerticesGraph` to write first. There should be **no dependence or sharing of code** between your two implementations.

2.1. Implement `ConcreteEdgesGraph`

For `ConcreteEdgesGraph` , you must use the rep provided:

```
private final Set<String> vertices = new HashSet<>();
private final List<Edge> edges = new ArrayList<>();
```

You may not add fields to the rep or choose not to use one of the fields.

For class `Edge` , you define the specification and representation; however, `Edge` must be **immutable** .

Normally, implementing an immutable type means implementing `equals` and `hashCode` as described in class 15 on equality (equality-todo.html#../../classes/15-equality/) . That's **not** required for this problem set. Be careful: not implementing equality means you cannot use `equals` to compare `Edge` objects! Specify and implement your own observer operations if you need to compare edges.

After deciding on its spec, devise, document, and implement tests for `Edge` in `ConcreteEdgesGraphTest.java` .

Then proceed to implement `Edge` and `ConcreteEdgesGraph` .

Java note

We require that class `Edge` be declared in `ConcreteEdgesGraph.java` .

However, you may move `Edge` to be a static nested class or instance inner class (https://docs.oracle.com/javase/tutorial/java/javaOO/n of `ConcreteEdgesGraph` if desired.

The identical restriction and permission apply to `ConcreteVerticesGraph` 's `Vertex` .

Be sure to use the `@Override` (//docs.oracle.com/javase/tutorial/java/annotations/predefined.html) annotation on `toString` to ensure that you are correctly overriding the `Object` version of the method, not creating a new, different method.

You should strengthen the spec of `ConcreteEdgesGraph.toString()` and write tests for it in `ConcreteEdgesGraphTest.java` . Don't add those tests to `GraphInstanceTest` , which must test the `Graph` spec. All Java classes inherit `Object.toString()` (https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#toString--) with its very underdetermined spec. `Graph` doesn't specify a stronger spec, but your concrete classes can.

Run the tests by right-clicking on `ConcreteEdgesGraphTest.java` and selecting *Run As* → *JUnit Test* .

Commit to Git. Once you're happy with your solution to this problem, commit to your repo!

2.2. Implement `ConcreteVerticesGraph`

For `ConcreteVerticesGraph` , you must use the rep provided:

```
private final List<Vertex> vertices = new ArrayList<>();
```

You may not add fields to the rep.

For class `Vertex` , you define the specification and representation; however, `Vertex` must be **mutable** .

After deciding on its spec, devise, document, and implement tests for `Vertex` in `ConcreteVerticesGraphTest.java` .

Then proceed to implement `Vertex` and `ConcreteVerticesGraph` .

You should strengthen the spec of `ConcreteVerticesGraph.toString()` and write tests for it in `ConcreteVerticesGraphTest.java` . Don't add those tests to `GraphInstanceTest` , which must test the `Graph` spec.

Run the tests by right-clicking on `ConcreteVerticesGraphTest.java` and selecting *Run As* → *JUnit Test* .

Commit to Git. Once you're happy with your solution to this problem, commit to your repo!

Problem 3: Implement generic `Graph<L>`

Nothing in either of your implementations of `Graph` should rely on the fact that labels are of type `String` in particular.

The spec says that vertex labels "must be immutable" and are "compared using the `equals` (//docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object-) method." Let's change both of our implementations to support vertex labels of *any* type that meets these conditions.

3.1. Make the implementations generic

a. Change the declarations of your concrete classes to read:

```
public class ConcreteEdgesGraph<L> implements Graph<L> { ... }

class Edge<L> { ... }
```

and:

```
public class ConcreteVerticesGraph<L> implements Graph<L> { ... }

class Vertex<L> { ... }
```

b. Update both of your implementations to support any type of vertex label, using placeholder `L` instead of `String`. Roughly speaking, you should be able to find-and-replace all the instances of `String` in your code with `L`!

This refactoring (https://en.wikipedia.org/wiki/Code_refactoring) will make `ConcreteEdgesGraph`, `ConcreteVerticesGraph`, `Edge`, and `Vertex` generic types.

- Previously, you might have declared variables with types like `Edge` or `List<Edge>`. Those will need to become `Edge<L>` and `List<Edge<L>>`.
- Similarly, you might have called constructors like `new ConcreteEdgesGraph()` or `new Edge()`. Those will need to become, for example, `new ConcreteEdgesGraph<String>()` and `new Edge<L>()`. Depending on context, you can use diamond notation (https://docs.oracle.com/javase/tutorial/java/generics/types.html#diamond) `<>` to avoid writing type parameters twice.

When you're done with the conversion, all your instance method tests should pass.

Commit to Git. As always, once you're happy with your solution to this problem, commit to your repo!

3.2. Implement `Graph.empty()`

a. Pick one of your `Graph` implementations for clients to use, and implement `Graph.empty()`. Unlike `ArrayList` and `LinkedList`, where clients who only want a `List` are forced to suffer a dose of rep exposure by calling a concrete constructor, clients of `Graph` should not be aware of how we've implemented it.

b. Because the implementation of `Graph` does not know or care about the actual type of the vertex labels, our test suite with `String` labels is sufficient.

However, since this is our first generic type, to gain confidence that you do indeed support different types of labels, add a few additional tests to the provided **`GraphStaticTest.java`** that create and use `Graph` instances with a couple different types of (immutable) labels.

At this point, all of your code (implementations and tests) must have **no warnings** from the compiler (warnings have a ⚠ symbol, as opposed to the ⊗ for errors) and no `@SuppressWarnings` (//docs.oracle.com/javase/8/docs/api/?java/lang/SuppressWarnings.html) annotations (which would disable the warning, nice try).

Run *all* the tests in the project by right-clicking on the `test` folder and selecting *Run As → JUnit Test*.

At this point in the problem set, we've completed the test-first implementation of `Graph`.

- You should have a green bar from JUnit and excellent code coverage from EclEmma.
- This is a good opportunity for self code review: remove dead code and debugging `println`s, DRY up duplication, and so on.
- And of course, commit to your repo.

Java note

You may use `@SafeVarargs` (//docs.oracle.com/javase/8/docs/api/?java/lang/SafeVarargs.html) where necessary, for example if you write testing helper functions that take a variable number of arguments (https://docs.oracle.com/javase/tutorial/java/javaOO/a.

Problem 4: Poetic walks

Graphs — what are they good for? Poetry!

The specification of **`GraphPoet`** (doc/index.html?poet/GraphPoet.html) explains how a poet is initialized with a corpus and then, given an input, uses a word affinity graph defined by that corpus to transform the input poetically.

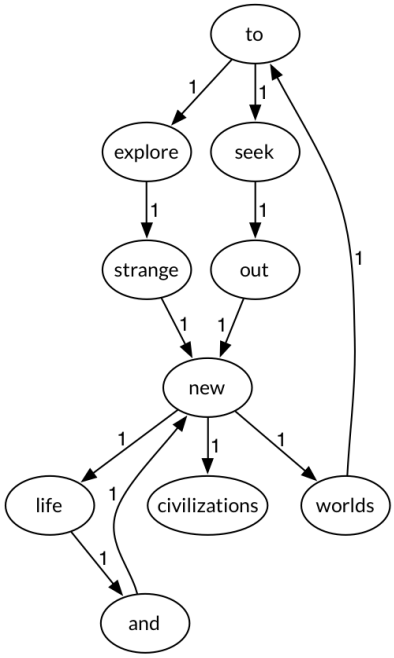
For example, here's a small but inspirational corpus:

> To explore strange new worlds
> To seek out new life and new civilizations
> (https://en.wikipedia.org/wiki/Where_no_man_has_gone_before)

`GraphPoet` will use this corpus to generate a word affinity graph where:

- vertices are case-insensitive words, and
- edge weights are in-order adjacency counts.

The graph is shown on the right. In this example, all the edges have weight 1 because no word pair is repeated.



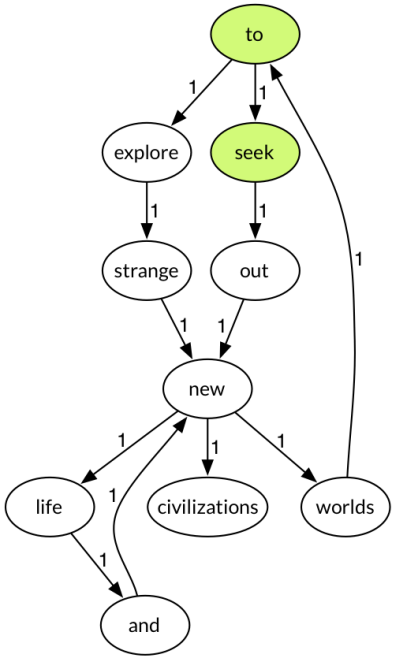
From there, given this dry bit of business-speak input:

> Seek to explore new and exciting synergies!

the poet will use the graph to generate a poem by inserting (where possible) a *bridge word* between each pair of input words:

w_1		w_2	path?	bridge word
seek	→	*to*	no two-edge-long path from *seek* to *to*	—
to	→	*explore*	no two-edge-long path from *to* to *explore*	—
explore	→	*new*	one two-edge-long path from *explore* to *new* with weight 2	*strange*
new	→	*and*	one two-edge-long path from *new* to *and* with weight 2	*life*
and	→	*exciting*	no vertex *exciting*	—
exciting	→	*synergies!*	no vertices *exciting* or *synergies!*	—

Hover over rows above to visualize poem generation on the right. In this example, the outcome is deterministic since no word pair has multiple bridge words tied for maximum weight. The resulting output poem is:



> Seek to explore strange new life and exciting synergies!

Exegesis is left as an exercise for the reader.

4.1. Test `GraphPoet`

Devise, document, and implement tests for `GraphPoet` in **`GraphPoetTest.java`**.

You are free to add additional methods to `GraphPoet`, but you may not change the signatures of the required methods. You are free to strengthen the specifications of the required methods, but you may not weaken them. Note how this might mean your tests for `GraphPoet` are not usable against someone else's implementation, because you've strengthened or added to the spec.

Your tests should make use of one or more sample corpus files. Put those files in the `test/poet` directory — the same directory as `GraphPoetTest.java`. Reference them in the code with, for example:

```
new File("test/poet/seven-words.txt")
```

While you still must test and implement `GraphPoet(File)`, remember that you may add operations to `GraphPoet`. Adding a creator that takes a `String`, for example, might simplify your tests for `poem(..)` at the cost of additional tests for the new creator.

4.2. Implement `GraphPoet`

Implement `GraphPoet` in **`GraphPoet.java`**.

You must use `Graph` in the rep of `GraphPoet`, but the implementation is otherwise entirely up to you.

For reading in corpus files, there are at least three Java APIs to consider:

- `FileReader` (//docs.oracle.com/javase/8/docs/api/?java/io/FileReader.html) is the standard class for reading from a file. Wrapping the `FileReader` in a `BufferedReader` (//docs.oracle.com/javase/8/docs/api/?java/io/BufferedReader.html) allows you to use the convenient `readLine()` method to read whole lines at a time.
- The utility function `Files.readAllLines(..)` (//docs.oracle.com/javase/8/docs/api/?java/nio/file/Files.html) will read all the lines from a `Path` (//docs.oracle.com/javase/8/docs/api/?java/nio/file/Path.html) . Our poet takes a `File` (//docs.oracle.com/javase/8/docs/api/?java/io/File.html) , but `File` provides a `toPath()` method to obtain a `Path` .
- Class `Scanner` (//docs.oracle.com/javase/8/docs/api/?java/util/Scanner.html) is designed for breaking up input text into pieces. It provides many features and you'll need to read its specs carefully.

Java note

Another option, if you want to program in a functional style: use `Files.lines(..)` , which returns a `Stream<String>` (//docs.oracle.com/javase/8/docs/api/?java/util/stream/Stream.html) .

4.3. Graph poetry slam

If you want, update the `main(..)` method in **`Main.java`** with a cool example: an interesting poem from minimal input, a surprising poem given the corpus and input, or something else cool.

Put the corpus for your example in `src/poet` — the same directory as `mugar-omni-theater.txt` .

Before you're done

- Make sure you have **documented specifications** , in the form of properly-formatted Javadoc comments, for all your types and operations.
- Make sure you have **documented abstraction functions and representation invariants** , in the form of a comment near the field declarations, for all your implementations.

  With the rep invariant, also say how the type prevents rep exposure.

  Make sure all types use `checkRep` to check the rep invariant and implement `toString` with a useful human-readable representation of the abstract value.
- To gain static checking of the correct signature, use `@Override` (//docs.oracle.com/javase/tutorial/java/annotations/predefined.html) when you override `toString` .

  Also use `@Override` when a class implements an interface method, to remind readers where they can find the spec.
- Make sure you have a thorough, principled test suite for every type.

MIT EECS

6.005 — Software Construction on MIT OpenCourseWare (https://ocw.mit.edu/ans7870/6/6.005/s16/) | OCW 6.005 Homepage (https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/) Spring 2016

Problem Set 3: Expressivo

In this problem set, we will explore parsers, recursive data types, and equality for immutable types.

Compared to the previous problem sets, we are imposing very few restrictions on how you structure your code. In addition, much of the code that you write for problems in this problem set will depend heavily on how you decided to implement earlier parts of the problem set. We strongly recommend that you read through the entire assignment before writing any code.

Design Freedom and Restrictions

On several parts of this problem set, the classes and methods will be yours to specify and create, but you must pay attention to the **PS3 instructions** sections in the provided documentation.

You must satisfy the specifications of the provided interfaces and methods. You are, however, permitted to strengthen the provided specifications or add new methods. On this problem set, unlike previous problem sets, we will not be running your tests against any other implementations.

Get the code

To get started, download the assignment code (ps3.zip) and initialize a repository. If you need a refresher on how to create your repository, see Problem Set 0 (../ps0/#clone) .

Overview

Wouldn't it be nice not to have to differentiate all our calculus homework by hand every time? Wouldn't it be just lovely to type it into a computer and have the computer do it for us instead? For example, we could interact with it like this (user input in green):

If the output is an expression, your system may output an equivalent expression, including variations in

```
> x * x * x
x*x*x

> !simplify x=2
8

> !d/dx
(x*x)*1+(x*1+1*x)*x

> !simplify x=0.5000
.75

> x * y
x*y

> !d/dy
0*y+x*1
```

spacing, parentheses, simplification, and number representation.

If a number, your system may output an equivalent number, accurate to at least 4 decimal places.

In this system, a user can enter either an **expression** or a **command** .

An *expression* is a polynomial consisting of:

- `+` and `*` (for addition and multiplication)
- nonnegative numbers in decimal representation, which consist of digits and an optional decimal point (e.g. `7` and `4.2`)
- variables, which are case-sensitive nonempty sequences of letters (e.g. `y` and `Foo`)
- parentheses (for grouping)

Order of operations uses the standard PEMDAS (https://en.wikipedia.org/wiki/Order_of_operations#Mnemonics) rules.

Space characters around symbols are irrelevant and ignored, so `2.3*pi` means the same as `2.3 * pi` . Spaces may not occur within numbers or variables, so `2 . 3 * p i` is not a valid expression.

When the user enters an expression, that expression becomes the **current expression** and is echoed back to the user (user input in green):

```
> x * x * x
x*x*x

> x + x * x * y + x
x + x*x*y + x
```

If the output is an expression, your system may output an equivalent expression, including variations in spacing, parentheses, simplification, and number representation.

If a number, your system may output an equivalent number, accurate to at least 4 decimal places.

A *command* starts with `!` . The command operates on the current expression, and may also update the current expression. Valid commands:

`d/d`*`var`
produces the derivative of the current expression with respect to the variable *var* , and updates the current expression to that derivative

`simplify`*`var`*$_1$`=`***`num`***$_1$** **`...`** ***`var`***$_n$**`=`***`num`***$_n$**
substitutes *num* $_i$ for *var* $_i$ in the current expression, and evaluates it to a single number if the expression contains no other variables; does **not** update the current expression

Entering an invalid expression or command prints an error but does not update the current expression. The error should include a human-readable message but is not otherwise specified.

More examples:

```
> x * x * x
x*x*x

> 3xy
ParseError: unknown expression

> !d/dx
(x*x)*1+(x*1+1*x)*x

> !d/dx
(x*x)*0+(x*1+1*x)*1+(x*1+1*x)*1+(x*0+1*1+x*0+1*1)*x

> !d/d
ParseError: missing variable in derivative command

> !simplify
(x*x)*0+(x*1+1*x)*1+(x*1+1*x)*1+(x*0+1*1+x*0+1*1)*x

> !simplify x=1
6.0

> !simplify x=0 y=5
0
```

If the output is an expression, your system may output an equivalent expression, including variations in spacing, parentheses, simplification, and number representation.

If a number, your system may output an equivalent number, accurate to at least 4 decimal places.

The three things that a user can do at the console correspond to three provided method specifications in the code for this problem set:

- `Expression.parse()`
- `Commands.differentiate()`
- `Commands.simplify()`

These methods are used by `Main` to provide the console user interface described above.

Problem 1: we will create the `Expression` data type to represent expressions in the program.

Problem 2: we will create the parser that turns a string into an `Expression`, and implement `Expression.parse()`.

Problems 3-4: we will add new `Expression` operations for differentiation and simplification, and implement `Commands.differentiate()` and `Commands.simplify()`.

Problem 1: Representing Expressions

Define an immutable, recursive abstract data type to represent expressions as abstract syntax trees.

Your AST should be defined in the provided `Expression` interface (in `Expression.java`) and implemented by several concrete variants, one for each kind of expression. Each variant should be defined in its own appropriately-named `.java` file.

Concrete syntax in the input, such as parentheses and whitespace, should not be represented at all in your AST.

1.1 `Expression`

To repeat, your data type must be **immutable** and **recursive** . Follow the recipe for creating an ADT:

- **Spec** . Choose and specify operations. For this part of the problem set, the only operations `Expression` needs are creators and producers for building up an expression, plus the standard observers `toString()` , `equals()` , and `hashCode()` . We are strengthening the specs for these standard methods; see below.
- **Test** . Partition and test your operations in `ExpressionTest.java` , including tests for `toString()` , `equals()` , and `hashCode()` . Note that we will not run your test cases on other implementations, just on yours.
- **Code** . Write the rep for your `Expression` as a data type definition (../../classes/16-recursive-data-types/recursive/#recursive_datatype_definitions) in a comment inside `Expression` . Implement the variant classes of your data type.

Remember to include a Javadoc comment above every class and every method you write; define abstraction functions and rep invariants, and write checkRep; and document safety from rep exposure.

1.2 `toString()`

Define the `toString()` operation on `Expression` so it can output itself as a string. This string must be a valid expression as defined above. You have the freedom to decide how to format the output with whitespace and parentheses for readability, but the expression must have the same mathematical meaning.

Your `toString()` implementation must be recursive, and must not use `instanceof` .

Use the `@Override` annotation to ensure you are overriding the `toString()` inherited from `Object` .

Remember that your tests must obey the spec. If your `toString()` tests expect a certain formatting of whitespace and parentheses, you must specify this formatting in your spec.

1.3 `equals()` and `hashCode()`

Define the `equals()` and `hashCode()` operations on your AST to implement *structural equality* .

Structural equality defines two expressions to be equal if:

a. the expressions contain the same variables, numbers, and operators;
b. those variables, numbers, and operators are in the same order, read left-to-right;
c. and they are grouped in the same way.

For example, the AST for `1 + x` is *not* equal to the AST for `x + 1` , but it is equal to the ASTs for `1+x` , `(1+x)` , and `(1)+(x)` .

For *n* -ary groupings where *n* is greater than 2:

- Such expressions must be equal to themselves. For example, the ASTs for `3 + 4 + 5` and `(3 + 4 + 5)` must be equal.
- However, whether they are equal or not to different groupings with the same mathematical meaning is *not specified* , and you should choose an appropriate specification and implementation for your AST. For example, you must determine whether the ASTs for `(3 + 4) + 5` and `3 + (4 + 5)` are equal.

For equality of numbers, you have the freedom to choose a reasonable definition. Integers that can be represented exactly as a `double` should be considered equal. For example, the ASTs for `x + 1` and `x + 1.000` must be equal.

Remember: concrete syntax, including parentheses, should not be represented in your AST. Grouping, for example, should be reflected in the AST's structure.

Be sure that AST instances which are considered equal according to this definition and according to `equals()` also satisfy observational equality (//web.mit.edu/6.005/www/sp16/classes/15-equality/) .

Your `equals()` and `hashCode()` implementations must be recursive. Only `equals()` can use `instanceof` , and `hashCode()` must not.

Remember to use the `@Override` annotation.

Commit to Git. Once you're happy with your solution to this problem, commit to your repo!

Problem 2: Parsing Expressions

Now we will create the parser that takes a string and produces an `Expression` value from it. The entry point for your parser should be `Expression.parse()` , whose spec is provided in the starting code.

Examples of valid inputs:

```
3 + 2.4
3 * x + 2.4
3 * (x + 2.4)
((3 + 4) * x * x)
foo + bar+baz
(2*x    )+    (    y*x    )
4 + 3 * x + 2 * x * x + 1 * x * x * (((x)))
```

Examples of invalid inputs:

```
3 *
( 3
3 x
```

Examples of optional inputs:

```
2 - 3
(3 * x) ^ 2
6.02e23
```

You may consider these inputs invalid, or you may choose to support additional features (new operators or number representations) in the input. However, *your system may not produce an output with a new feature unless that feature appeared in its input* . This way, a client who knows about your extensions can trigger them, but clients who don't know won't encounter them unexpectedly.

2.1 Write a grammar

Write an ANTLR grammar for polynomial expressions as described in the overview. A starting ANTLR grammar file can be found in `src/expressivo/parser/Expression.g4` . This starting grammar recognizes sums of integers, and ignores spaces.

The file `Configuration.g4` contains some common boilerplate that is imported into `Expression.g4` . You should not edit `Configuration.g4` .

See the reading on parser generators (//web.mit.edu/6.005/www/sp16/classes/18-parser-generators/) for more information about ANTLR, including links to documentation.

2.2 Implement `Expression.parse()`

Implement `Expression.parse()` by following the recipe:

- **Spec** . The spec for this method is given, but you may strengthen it if you want to make it easier to test.
- **Test** . Write tests for `Expression.parse()` and put them in `ExpressionTest.java` . Note that we will not run your tests on any implementations other than yours.
- **Code** . Implement `Expression.parse()` so that it calls the parser generated by your ANTLR grammar. The reading on parser generators (../../classes/18-parser-generators/) discusses how to call the parser and construct an abstract syntax tree from it, including code examples.

Hint: use `reportErrorsAsExceptions` to change how the lexer or parser handles errors.

A general note on precision: you are only required to handle nonnegative decimal numbers in the range of the `double` (//docs.oracle.com/javase/8/docs/api/?java/lang/Double.html) type.

2.3 Run the console interface

Now that `Expression` values can be both parsed from strings with `parse()` , and converted back to strings with `toString()` , you can try entering expressions into the console interface.

Run `Main` . In Eclipse, the Console view will allow you to type expressions and see the result. Try some of the expressions from the top of this handout.

Commit to Git. Once you're happy with your solution to this problem, commit to your repo!

Problem 3: Differentiation

The symbolic differentiation operation takes an expression and a variable, and produces an expression with the derivative of the input with respect to the variable. The result does not need to be simplified.

For example, the following are correct derivatives :

`x*x*x` with respect to **`x`**
`3 * x * x`

`x*x*x` with respect to **`x`**
`x*x + (x + x)*x`

`x*x*x` with respect to **`x`**
`( ( x*x )*1 )+( ( ( x*1 )+( 1*x ) )*x )+( 0 )`

If the output is an expression, your system may output an equivalent expression, including variations in spacing, parentheses, simplification, and number representation.

If a number, your system may output an equivalent number, accurate to at least 4 decimal places.

Incorrect derivatives:

`y*y*y` with respect to **`y`**
`3*y^2` *uses unexpected operator*

`y*y*y` with respect to **`y`**
`0` *d/dx, should be d/dy*

To implement your recursive differentiation operation, use these rules:

$$\frac{dc}{dx} = 0 \qquad \frac{dx}{dx} = 1 \qquad \frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx} \qquad \frac{d(u \cdot v)}{dx} = u\left(\frac{dv}{dx}\right) + v\left(\frac{du}{dx}\right)$$

where *c* is a constant or variable other than the variable we are differentiating with respect to (in this case *x*), and *u* and *v* can be anything, including *x* .

3.1. Add an operation to `Expression`

You should implement differentiation as a method on your `Expression` datatype, defined recursively. The signature and specification of the method are up to you to design. Follow the recipe:

- **Spec** . Define your operation in `Expression` and write a spec.
- **Test** . Put your tests in `ExpressionTest.java` . Note that we will not run your test cases on other implementations, just on yours.
- **Code** . The implementation must be recursive. It must not use `instanceof` , nor any equivalent operation you have defined that checks the type of a variant.

3.2 Implement `Commands.differentiate()`

In order to connect your differentiation operation to the user interface, we need to implement the `Commands.differentiate()` method.

- **Spec** . The spec for this operation is given, but you may strengthen it if you want to make it easier to test.
- **Test** . Write tests for `differentiate()` and put them in `CommandsTest.java` . These tests will likely be very similar to the tests you used for your lower-level differentiation operation, but they must use `Strings` instead of `Expression` objects. Note that we will not run your tests on any implementations other than yours.
- **Code** . Implement `differentiate()` . This should be straightforward: simply parsing the expression, calling your differentation operation, and converting it back to a string.

3.3 Run the console interface

We've now implemented the `!d/d` command in the console interface. Run `Main` and try some derivatives in the Console view.

Commit to Git. Once you're happy with your solution to this problem, commit to your repo!

Problem 4: Simplification

The simplification operation takes an expression and an environment (a mapping of variables to values). It substitutes the values for those variables into the expression, and attempts to simplfy the substituted polynomial as much as it can.

The set of variables in the environment is allowed to be different than the set of variables actually found in the expression. Any variables in the expression but not the environment remain as variables in the substituted polynomial. Any variables in the environment but not the expression are simply ignored.

The only required simplification is that if the substituted polynomial is a constant expression, with no variables remaining, then simplification must reduce it to a single number, with no operators remaining either. Simplification for substituted polynomials that still contain variables is underdetermined, left to the implementer's discretion. You may strengthen this spec if you wish to require particular simplifications in other cases.

For example, the following are correct output for simplified expressions:

> If the output is an expression, your system may output an equivalent expression, including variations in spacing, parentheses, simplification, and number representation.
>
> If a number, your system may output an equivalent number, accurate to at least 4 decimal places.

`x*x*x` with environment **`x=5 , y=10 , z=20`**
125

`x*x*x + y*y*y` with environment **`y=10`**
x*x*x+10*10*10

`x*x*x + y*y*y` with environment **`y=10`**
x*x*x+1000

`1+2*3+8*0.5` with empty environment
11.000

Incorrect simplified expressions:

`x*x*y` with environment **`x=1 , y=3`**
1*1*3 *not a single number*

`x*x*x` with environment **`x=2`**
(8) *includes parentheses*

`x*x*x` with empty environment
x^3 *uses unexpected operator*

Optional simplified expressions:

`x*x*y + y*(1+x)` with environment **`x=2`**
7*y

`x*x*x + x*x*x` with empty environment
2*x*x*x

4.1 Add an operation to `Expression`

You should implement simplification as a method on your `Expression` datatype, defined recursively. The signature and specification of the method are up to you to design. Follow the recipe:

- **Spec** . Define your operation in `Expression` and write a spec.
- **Test** . Put your tests in `ExpressionTest.java` . Note that we will not run your test cases on other implementations, just on yours.
- **Code** . The implementation must be recursive (perhaps by calling recursive helper methods). It must not use `instanceof` , nor any equivalent operation you have defined that checks the type of a variant class.

You may find it useful to add more operations to `Expression` to help you implement the simplify operation. Spec/test/code them using the same recipe, and make them recursive as well where appropriate. Your helper operations should not simply be a variation on using `instanceof` to test for a variant class.

4.2 `Commands.simplify()`

In order to connect your simplify operation to the user interface, we need to implement the `Commands.simplify()` method.

- **Spec** . The spec for this operation is given, but you may strengthen it if you want to make it easier to test.
- **Test** . Write tests for `simplify()` and put them in `CommandsTest.java` . These tests will likely be very similar to the tests you used for your lower-level simplify operation, but they should use

`Strings` instead of `Expression` objects. Note that we will not run your tests on any implementations other than yours.

- **Code** . Implement `simplify()` . This should be straightforward: simply parsing the expression, calling your simplify operation, and converting it back to a string.

4.3 Run the console interface

We've now implemented the `!simplify` command in the console interface. Run `Main` and try using it in the Console view.

Commit to Git. Once you're happy with your solution to this problem, commit to your repo!

Before you're done

- Make sure you have **documented specifications** , in the form of properly-formatted Javadoc comments, for all your types and operations.
- Make sure you have **documented abstraction functions and representation invariants** , in the form of a comment near the field declarations, for all your implementations.

  With the rep invariant, also say how the type prevents rep exposure.

  Make sure all types use `checkRep()` to check the rep invariant and implement `toString()` with a useful representation of the abstract value.
- Make sure you have satisfied the **Object contract (../../classes/15-equality/#the_object_contract)** for all types. In particular, you will need to specify, test, and implement `equals()` and `hashCode()` for all immutable types.
- Use `@Override` (//docs.oracle.com/javase/tutorial/java/annotations/predefined.html) when you override `toString()` , `equals()` , and `hashCode()` , to gain static checking of the correct signature.

  Also use `@Override` when a class implements an interface method, to remind readers where they can find the spec.
- Make sure you have a thorough, principled **test suite** for every type. Note that `Expression` 's variant classes are considered part of its rep, so a single good test suite for `Expression` covers the variants too.

MIT EECS

6.005 — Software Construction on MIT OpenCourseWare (https://ocw.mit.edu/ans7870/6/6.005/s16/) | OCW 6.005 Homepage (https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/) Spring 2016

Problem Set 4: Multiplayer Minesweeper

The purpose of this problem set is to explore multithreaded programming with a shared mutable data type, which you should protect using one or more thread safety strategies.

Design Freedom and Restrictions

On this problem set, you have substantial design freedom.

Your solution **must not** change the name, signature, or specification of the `MinesweeperServer` methods `main()` or `runMinesweeperServer()`; you also should not change the implementation of `main()`. You **may** change any of the other provided code in `MinesweeperServer`.

Also note that the axis definition of your board must match what is defined in *Protocol and specification*. The (x, y) coordinates start at (0, 0) in the top-left corner, extend horizontally to the right in the *x* direction, and extend vertically downwards in the *y* direction.

Your code will be tested against the specification. Changing these interfaces or axes will cause the tests to fail, and you will receive 0 points for the submission. It is your responsibility to examine Didit feedback and make sure your code compiles and runs for grading, but you must do your own testing to ensure correctness.

Beyond these requirements you have complete design freedom. For example, you can add new methods, classes, and packages; rewrite or delete other pieces of provided code; etc.

Get the code

To get started, download the assignment code (ps4.zip) and initialize a repository. If you need a refresher on how to create your repository, see Problem Set 0 (../ps0/#clone) .

Overview

You will start with some minimal server code and implement a server and thread-safe data structure for playing a multiplayer variant of the classic computer game "Minesweeper."

- You can review the **rules of traditional single-player Minesweeper** on Wikipedia (//en.wikipedia.org/wiki/Minesweeper_(video_game)) .
- You can also **play traditional single-player Minesweeper** online (//minesweeperonline.com/) .

  (You may notice that the implementation in the latter link above does something subtle. It ensures that there's never a bomb where you make your first click of the game. You should *not* implement this for the assignment. It would be in conflict with giving the option to pass in a pre-designed board, for example.)

Your final product will consist of a server and no client; it should be fully playable using the `telnet` utility to send text commands directly over a network connection as described below.

Multiplayer Minesweeper server

We will refer to the board as a grid of squares. Each square is either *flagged* , *dug* , or *untouched* . Each square also either contains a bomb, or does not contain a bomb.

Our variant works very similarly to standard Minesweeper, but with multiple players simultaneously playing on a single board. In both versions, players lose when they try to dig an untouched square that happens to contain a bomb. Whereas a standard Minesweeper game would end at this point, in our version, the game keeps going for the other players. In our version, when one player blows up a bomb, they still lose, and the game ends for them (the server ends their connection), but the other players may continue playing. The square where the bomb was blown up is now a *dug* square with no bomb. The player who lost may reconnect to the same game again via `telnet` to start playing again.

Note that there are some tricky cases of user-level concurrency. For example, say user *A* has just modified the game state (i.e. by digging in one or more squares) such that square *i,j* obviously has a bomb. Meanwhile, user *B* has not observed the board state since this update has taken place, so user *B* goes ahead and digs in square *i,j* . Your program should allow the user to dig in that square — a user of Multiplayer Minesweeper must accept this kind of risk.

We are not specifically defining, or asking you to implement, any kind of "win condition" for the game.

Telnet client

`telnet` is a utility that allows you to make a direct network connection to a listening server and communicate with it via a terminal interface. Before starting this problem set, please ensure that you have `telnet` installed. *nix operating systems (including Mac OS X) should have `telnet` installed by default.

Windows users should first check if telnet is installed by running the command `telnet` on the command line.

- If you do not have `telnet` , you can install it via Control Panel → Programs and Features → Turn Windows features on/off → Telnet client. However, this version of `telnet` may be very hard to use. If it does not show you what you're typing, you will need to turn on the localecho option

(//windows.microsoft.com/en-us/windows/telnet-commands#1TC=windows-7) .

- A better alternative is PuTTY: download `putty.exe` (//www.chiark.greenend.org.uk/~sgtatham/putty/download.html) . To connect using PuTTY, enter a hostname and port, select Connection type: Raw, and Close window on exit: Never. The last option will prevent the window from disappearing as soon as the server closes its end of the connection.

You can have `telnet` connect to a host and port (for example, `web.mit.edu:80`) from the command line with:

```
telnet web.mit.edu 80
```

Since port 80 is usually used for HTTP, we can now make HTTP requests through `telnet` . If you now type (where ↵ indicates a newline):

```
GET /↵
↵
```

`telnet` should retrieve the HTML for the webpage at web.mit.edu (//web.mit.edu) . If you want to connect to your own machine, you can use `localhost` as the hostname and whatever port your server is listening on. With the default port of 4444 in this problem set, you can connect to your Minesweeper server with:

```
telnet localhost 4444
```

The server may close the network connection at any time, at which point `telnet` will exit. To close a connection from the client, note the first output printed by telnet when it connects:

```
Trying 18.9.22.69...
Connected to web.mit.edu.
Escape character is '^]'.
```

`^]` means `Ctrl - ]` , so press that. Then enter `quit` to close the connection and exit telnet.

Protocol and specification

You must implement the following protocol for communication between the user and the server.

The messages in this protocol are described precisely and comprehensively using a pair of grammars. Your server must accept any incoming message that satisfies the user-to-server grammar, react appropriately, and generate only outgoing messages that satisfy the server-to-user grammar.

- Wikipedia description of the **grammar notation: Backus–Naur Form** (//en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form)
- Wikipedia description of **regular expressions** , which are used on the right-hand side of grammar productions (//en.wikipedia.org/wiki/Regular_expression)

Messages from the user to the server

Formal grammar

```
MESSAGE ::= ( LOOK | DIG | FLAG | DEFLAG | HELP_REQ | BYE ) NEWLINE
LOOK ::= "look"
DIG ::= "dig" SPACE X SPACE Y
FLAG ::= "flag" SPACE X SPACE Y
DEFLAG ::= "deflag" SPACE X SPACE Y
HELP_REQ ::= "help"
BYE ::= "bye"
NEWLINE ::= "\n" | "\r" "\n"?
X ::= INT
Y ::= INT
SPACE ::= " "
INT ::= "-"? [0-9]+
```

Each message starts with a message type and arguments to the message are seperated by a single SPACE character and the message ends with a NEWLINE. The NEWLINE can be a single character `"\n"` or `"\r"` or the two-character sequence `"\r\n"` , the same definition used by `BufferedReader.readLine()` .

The user can send the following messages to the server:

LOOK message

The message type is the word “look” and there are no arguments.

Example: | l | o | o | k | \n |

Returns a BOARD message, a string representation of the board’s state. Does not mutate anything on the server. See the section below on messages from the server to the user for the exact required format of the BOARD message.

DIG message

The message is the word “dig” followed by two arguments, the X and Y coordinates. The type and the two arguments are seperated by a single SPACE.

Example: | d | i | g | | 3 | | 1 | 0 | \r | \n |

The dig message has the following properties:

1. If either x or y is less than 0, or either x or y is equal to or greater than the board size, or square x,y is not in the *untouched* state, do nothing and return a BOARD message.
2. If square x,y’s state is *untouched* , change square x,y’s state to *dug* .
3. If square x,y contains a bomb, change it so that it contains no bomb and send a BOOM message to the user. Then, if the debug flag is missing (see Question 4), terminate the user’s connection. See again the section below for the exact required format of the BOOM message. Note: When modifying a square from containing a bomb to no longer containing a bomb, make sure that subsequent BOARD messages show updated bomb counts in the adjacent squares. After removing the bomb continue to the next step.
4. If the square x,y has no neighbor squares with bombs, then for each of x,y’s *untouched* neighbor squares, change said square to *dug* and repeat *this step* (not the entire DIG procedure) recursively for said neighbor square unless said neighbor square was already dug before said change.
5. For any DIG message where a BOOM message is not returned, return a BOARD message.

FLAG message

The message type is the word “flag” followed by two arguments the X and Y coordinates. The type and the two arguments are seperated by a single SPACE.

Example: | f | l | a | g | | 1 | 1 | | 8 | \n |

The flag message has the following properties:

1. If x and y are both greater than or equal to 0, and less than the board size, and square x,y is in the *untouched* state, change it to be in the *flagged* state.
2. Otherwise, do not mutate any state on the server.
3. For any FLAG message, return a BOARD message.

DEFLAG message

The message type is the word “deflag” followed by two arguments the X and Y coordinates. The type and the two arguments are seperated by a single SPACE.

Example: | d | e | f | l | a | g | | 9 | | 9 | \n |

The flag message has the following properties:

1. If x and y are both greater than or equal to 0, and less than the board size, and square x,y is in the *flagged* state, change it to be in the *untouched* state.
2. Otherwise, do not mutate any state on the server.
3. For any DEFLAG message, return a BOARD message.

HELP_REQ message

The message type is the word “help” and there are no arguments.

Example: | h | e | l | p | \r | \n |

Returns a HELP message (see below). Does not mutate anything on the server.

BYE message

The message type is the word “bye” and there are no arguments.

Example: | b | y | e | \r | \n |

Terminates the connection with this client.

Messages from the server to the user

Formal grammar

```
MESSAGE ::= BOARD | BOOM | HELP | HELLO
BOARD ::= LINE+
LINE ::= (SQUARE SPACE)* SQUARE NEWLINE
SQUARE ::= "-" | "F" | COUNT | SPACE
SPACE ::= " "
NEWLINE ::= "\n" | "\r" "\n"?
COUNT ::= [1-8]
BOOM ::= "BOOM!" NEWLINE
HELP ::= [^\r\n]+ NEWLINE
HELLO ::= "Welcome to Minesweeper. Board: " X " columns by " Y " rows. Play
ers: " N
          " including you. Type 'help' for help." NEWLINE
X ::= INT
Y ::= INT
N ::= INT
INT ::= "-"? [0-9]+
```

There are no message types in the messages sent by the server to the user. **The server sends the HELLO message as soon as it establishes a connection to the user.** After that, for any message it receives that matches the user-to-server message format, other than a BYE message, the server should always return either a BOARD message, a BOOM message, or a HELP message.

For any message from the user which does not match the user-to-server message format , discard the incoming message and send a HELP message as the reply to the user.

The action to take for each different kind of message is as follows:

HELLO message

In this message:

- *N* is the number of users currently connected to the server, and
- the board is $X \times Y$.

This message should be sent to the user exactly as defined and only once, immediately after the server connects to the user. Again the message should end with a NEWLINE.

Example:

```
Welcome to Minesweeper. Board: 8 columns by 8 rows.
 Players: 12 including you. Type 'help' for help.\r\n
```

BOARD message

The message has no start type word and consists of a series of newline-separated rows of space-separated characters, thereby giving a grid representation of the board's state with exactly one char for each square. The mapping of characters is as follows:

- "-" for squares with state *untouched* .
- "F" for squares with state *flagged* .
- " " (space) for squares with state *dug* and 0 neighbors that have a bomb.
- integer COUNT in range [1-8] for squares with state *dug* and COUNT neighbors that have a bomb.

Here is an example board message with 3 rows and 8 columns:

								2	-	-	\r	\n	
1	1				1	F	-	-	\n				
F	1				1	-	-	-	\n				

Notice that in this representation we reveal every square's state of *untouched* , *flagged* , or *dug* , and we indirectly reveal limited information about whether some squares have bombs or not.

In the printed **BOARD** output, the (x,y) coordinates start at (0,0) in the top-left corner, extend horizontally to the right in the X direction, and vertically downwards in the Y direction. (While different from the standard geometric convention for IV quadrant, this happens to be the protocol specification.) So the following output would represent a flagged square at **(x=1,y=2)** and the rest of the squares being untouched:

```
- - - - - - -
- - - - - - -
- F - - - - -
- - - - - - -
- - - - - - -
- - - - - - -
- - - - - - -
```

In order to conform to the protocol specification, you'll need to preserve this arrangement of cells in board while writing the board messages. If you change this order in either writing the board message or reading the board from a file (Problem 4) your implementation does not satisfy the spec.

BOOM message

The message is the word *BOOM!* followed by a NEWLINE.

Example:

B	O	O	M	!	\r	\n

The last message the client will receive from the server before it gets disconnected!

HELP message

The message is a sequence of non-NEWLINE characters (your help message) followed by NEWLINE.

Example:

R	T	F	M	!	\n

Unlike the above example, the HELP message should print out a message which indicates all the commands the user can send to the server. The exact content of this message is up to you – but it is important that this message not contain multiple NEWLINEs.

Problem 1: set up the server to deal with multiple clients

In `minesweeper.server.MinesweeperServer`, we have provided you with a single-threaded server which can accept connections with one client at a time, and which includes code to parse the input according to the client-server protocol above.

Modify the server so it can maintain multiple client connections simultaneously. Each client connection should be maintained by its own thread. You may wish to add another class to do this.

As always, all your code should be safe from bugs, easy to understand, and ready for change. Be sure the code to implement multiple concurrent clients is written clearly and commented if necessary.

You may continue to do nothing with the parsed user input for now.

When you write tests in `MinesweeperServerTest.java`, you may write tests similar to the **published test** (https://github.com/mit6005/sp16-ps4) we provide, and you may also write tests using `ByteArrayInput / OutputStream` stubs as described in *Sockets & Networking* (../../classes/21-sockets-networking/#testing_clientserver_code) .

Problem 2: design a data structure for Minesweeper

In `minesweeper.Board`, we have provided a starting file for this problem.

Specify, test, and implement a data structure for representing the Minesweeper board (as a Java type, without using sockets or threads). Define the `Board` ADT as well as any additional classes you might need to accomplish this.

You may ignore thread safety for now.

Your `Board` class (and other classes) must have specifications for all methods; abstraction function, rep invariant, and safety from rep exposure written as comments; and the rep invariant implemented as a `checkRep()` method called by your code.

Problem 3: make the entire system thread-safe

a. Come up with a strategy to make your system thread safe. As we've learned, there are multiple ways to make a system thread safe. For example, this can be done by:

   - using immutable or thread-safe data structures
   - using a queue to send messages that will be processed sequentially by a single thread
   - using synchronized methods or the synchronized keyword at the right places
   - or combination of these techniques

b. The specification for `Board` (and any other classes) should state whether that type is thead-safe.

   In addition to the other internal documentation, `Board` (and any other classes) must document their thread-safety argument. Depending on your design, the board (or other classes) may ***not*** be thread-safe. If so, document that.

   You will also document your system thread safety argument in `MinesweeperServer.java` in problem 5.

SFB, ETU, RFC: be sure any code to implement thread safety is written clearly and commented if necessary.

Problem 4: initialize the board based on command-line options

We want our server to be able to accept some command-line options. The exact specification is given in the Javadoc for `MinesweeperServer.main()`, which is excerpted below:

> Usage:
>
> ```
> MinesweeperServer [--debug | --no-debug] [--port PORT]
>                   [--size SIZE_X,SIZE_Y | --file FILE]
> ```
>
> The `--debug` argument means the server should run in debug mode. The server should disconnect a client after a BOOM message if and only if the `--debug` flag was NOT given. Using `--no-debug` is the same as using no flag at all.
>
> *E.g.* `MinesweeperServer --debug` starts the server in debug mode.
>
> `PORT` is an optional integer in the range 0 to 65535 inclusive, specifying the port the server should be listening on for incoming connections.
>
> *E.g.* `MinesweeperServer --port 1234` starts the server listening on port 1234.
>
> `SIZE_X` and `SIZE_Y` are optional positive integer arguments specifying that a random board of size SIZE_X × SIZE_Y should be generated.
>
> *E.g.* `MinesweeperServer --size 42,58` starts the server initialized with a random board of size 42 × 58.
>
> `FILE` is an optional argument specifying a file pathname where a board has been stored. If this argument is given, the stored board should be loaded as the starting board.
>
> *E.g.* `MinesweeperServer --file boardfile.txt` starts the server initialized with the board stored in `boardfile.txt`.

We provide you with the code required to parse these command-line arguments in the `main()` method already existing in `MinesweeperServer`. You should not change this method. Instead, you should change `runMinesweeperServer` to handle each of the two different ways a board can be initialized: either by random, or through input from a file. (You'll deal with the debug flag in Problem 5.)

For a **`--size`** argument: if the passed-in size X,Y > 0, the server's `Board` instance should be randomly generated and should have size equal to X by Y. To randomly generate your board, you should assign each square to contain a bomb with probability .25 and otherwise no bomb. All squares' states should be set to *untouched*.

For a **`--file`** argument: If a file exists at the given path, read the the corresponding file, and if it is properly formatted, deterministically create the `Board` instance. The file format for input is:

```
FILE ::= BOARD LINE+
BOARD := X SPACE Y NEWLINE
LINE ::= (VAL SPACE)* VAL NEWLINE
VAL ::= 0 | 1
X ::= INT
Y ::= INT
SPACE ::= " "
NEWLINE ::= "\n" | "\r" "\n"?
INT ::= [0-9]+
```

In a properly formatted file matching the FILE grammar, the first line specifies the board size, and it must be followed by exactly Y lines, where each line must contain exactly X values. If the file is properly formatted, the `Board` should be instantiated such that square *i,j* has a bomb if and only if the *i* 'th VAL in LINE *j* of the input is 1. If the file is improperly formatted, your program should throw an unchecked exception (either `RuntimeException` or define your own). The following example file describes a board with 4 columns and 3 rows with a single bomb at the location *x=2* , *y=1* .

Example:

4		3	\n				
0		0		0		0	\n
0		0		1		0	\n
0		0		0		0	\n

If neither `--file` nor `--size` is given, generate a random board of size 10 × 10.

Note that `--file` and `--size` may not be specified simultaneously.

Reading from a file: see **tips for reading and writing** below.

Implement the `runMinesweeperServer` method so that it handles the two possible ways to initialize a board (randomly or by loading a file).

Running the server on the command line: this is easiest to do from the `bin` directory where your code is compiled. Open a command prompt, `cd` to your `ps4` directory, then `cd bin` . Here are some examples:

```
cd bin
java minesweeper.server.MinesweeperServer --debug
java minesweeper.server.MinesweeperServer --port 1234
java minesweeper.server.MinesweeperServer --size 123,234
java minesweeper.server.MinesweeperServer --file ../testBoard
java minesweeper.server.MinesweeperServer --debug --port 1234 --size 20,14
```

You can specify command-line arguments in Eclipse by clicking on the drop-down arrow next to "run," clicking on "Run Configurations…", and selecting the "Arguments" tab. Type your arguments into the text box.

Problem 5: putting it all together

a. Modify your server so that it implements our protocols and specifications, using a single instance of `Board` .

Note that when we send a BOOM message to the user, we should terminate their connection if and only if the debug flag (see Problem 4) is missing.

The **tips for reading and writing** below may be useful for reading from and writing to socket streams.

b. Near the top of `MinesweeperServer`, include a substantial comment with an argument for why your system is thread-safe.

SFB, ETU, RFC: be sure the code to implement thread safety is written clearly and commented if necessary.

Commit to Git. Pushing your code to your Git repository on Athena will cause Didit to run a single staff-provided test. You can read and download the **source code for the published test** (https://github.com/mit6005/sp16-ps4) .

You are strongly encouraged to download and run the published test on your own machine.

Do not attempt to debug failures on Didit before you can *run and pass* the published test locally on your own machine.

Tips for reading and writing

- `BufferedReader.readLine()` (//docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html#readLine--) reads a line of text from an input stream, where line is considered to be terminated by any one of a line feed (`\n`), a carriage return (`\r`), or a carriage return followed immediately by a linefeed. This behavior matches the NEWLINE productions in our grammars.
- You can wrap a `BufferedReader` (//docs.oracle.com/javase/8/docs/api/?java/io/BufferedReader.html) around both `InputStreamReader` (//docs.oracle.com/javase/8/docs/api/?java/io/InputStreamReader.html) and `FileReader` (//docs.oracle.com/javase/8/docs/api/?java/io/FileReader.html) objects.
- `PrintWriter.println()` (//docs.oracle.com/javase/8/docs/api/java/io/PrintWriter.html#println--) prints formatted text to an output stream, and terminates the current line by writing the line separator string. It always writes a line separator, even if the input string already ends with one. The line separator is defined by the system property `line.separator` ; you can obtain it using `System.lineSeparator()` . It is `\n` on *nix systems and `\r\n` on Windows. Both of these line endings are allowed by the NEWLINE productions in our grammars.
- You can wrap a `PrintWriter` (//docs.oracle.com/javase/8/docs/api/?java/io/PrintWriter.html) around a `Writer` (//docs.oracle.com/javase/8/docs/api/?java/io/Writer.html) or `OutputStream` (//docs.oracle.com/javase/8/docs/api/?java/io/OutputStream.html) objects.

Beyond telnet

You are now done with the problem set, but you could imagine not having to use `telnet` to play Minesweeper. A GUI would do a lot to bring your program to life. Here is GUI client from past student **Robin Cheng** that you can use to play your Minesweeper game: **Minesweeper GUI** (./minesweeperGUI.jar) .

This GUI waits until you do a LOOK or other operation before updating the view.

It has UI for spawning clients, so you can actually just run the jar once and then use the UI to spawn a number of clients.

It also displays a printout of all the protocol I/O which is going on, making it particularly helpful for debugging.

You should have your server running before trying to start a client connection from this GUI.

Note: This GUI is entirely unofficial! You may find bugs or incorrect behavior. Do not consider it sufficient for testing your code.

If you want practice writing GUIs, we encourage you to write your own Minesweeper GUI and email the source to the staff.

MIT EECS