

Problem Set 1 | Problem Set 1 Beta | 6.005.1x Courseware

III openlearninglibrary.mit.edu/courses/course-v1:MITx+6.005.1x+3T2016/courseware/Week_2/ps1-beta

The purpose of this problem set is to:

- introduce the tools we use in 6.005x, including Java, Eclipse, and JUnit;
- introduce the process for 6.005x problem sets;
- and practice reading a function specification and writing basic Java to implement it.

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. You won't be asked to write any test cases of your own on this problem set. Testing will happen in Problem Set 2.

Configure Eclipse

Eclipse is a powerful, flexible, and occasionally frustrating set of tools for developing and debugging programs, especially in Java.

Note: if you prefer, you can do the problem sets in another Java development environment, or using command-line tools. The problem sets require compiling Java, running JUnit, and running Ant, so if you know how to do those things some other way, go for it. The rest of these instructions will talk only about using Eclipse.

You should have already installed Eclipse and Java when you set up the Java Tutor.

When you run Eclipse, you will be prompted for a "workspace" directory, where Eclipse will store its configuration and metadata. The default location is a directory called **workspace** in your home directory. You should not run more than one copy of Eclipse at the same time with the same workspace.

On the left side of your Eclipse window is the Package Explorer, which shows you all the projects in your workspace.

1. Open Eclipse preferences.

Windows & Linux: go to *Window* → *Preferences*.

OS X: go to *Eclipse* → *Preferences*.

2. Make sure Eclipse is configured to use **Java 8**.

1. In preferences, go to *Java* → *Installed JREs*. Ensure that "Java SE 8" or "JDK 1.8.0_71" is the only one checked. If it's not listed, click Search.
2. Go to *Java* → *Compiler* and set "Compiler compliance level" to 1.8. Click OK and Yes on any prompts.

3. Make sure **assertions are always on**. Assertions are a great tool for keeping your code safe from bugs, but Java has them off by default.

In preferences, go to *Java* → *Installed JREs*. Click "Java SE 8", click "Edit...", and in the "Default VM arguments" box enter: `-ea` (which stands for *enable assertions*).

4. Make sure your **Eclipse workspace refreshes automatically** whenever files are changed on the filesystem. This will be important when you run the grader script, so that you see the output files it generates.

In preferences, go to *General* → *Workspace*. Turn on both checkboxes that talk about refresh: "Refresh using native hooks or polling" and "Refresh on access."

5. **Tabs**. If you configure the editor to use spaces instead of tabs, then your code will look the same in all editors regardless of how that editor displays tab characters.

In preferences, go to *Java* → *Code Style* → *Formatter*. Click the "Edit..." button next to the active profile. In the new window, change the Tab policy to "Spaces only." Keep the Indentation size and Tab size at 4. To save your changes, enter a new "Profile name" at the top of the window and click OK.

The [6.005 Eclipse FAQ](#) has some tips and tricks to help you make the most of Eclipse.

Download and Import the Problem Set Code

1. Download the starting code for this problem set:

| [ps1.zip](#)

2. Import the code into Eclipse:

1. In the Eclipse menubar, choose File → Import...
2. In the Select dialog, open General and choose Existing Projects Into Workspace.
3. In the Import Projects dialog, choose Select archive file, then click Browse and use the file dialog to find where you downloaded ps1.zip.
4. Still in the Import Projects dialog, make sure ps1-warmup is listed in the Projects, and that the checkbox next to it is checked
5. On "Import projects," make sure ps1-warmup is checked.

If ps1-warmup is grayed and uncheckable, then the likely reason is that ps1-warmup already exists in your Package Explorer (perhaps because you are following these directions a second time). You need to cancel the import, delete or rename ps1-warmup in your Package Explorer, and try the import again.

6. Click Finish. ps1-warmup should now appear in your Package Explorer.

Warm up with `Quadratic.roots()`

In the Eclipse Package Explorer, open up `ps1-warmup / src / ps1`, and you should find the file `Quadratic.java`. Your warm-up task is to implement the `roots()` method in this class, which finds the roots of a quadratic equation. The method's specification comment describes how it should behave. Pay close attention to this specification while you're implementing. The quadratic formula will be useful in your solution, but be careful! Many things can go wrong.

Before you start writing code, read the rest of this handout, because it describes three ways that you can use to check your implementation: (1) a `main` method, (2) a JUnit test suite, and (3) an autograder. Read about and try running each of these first, using the unfinished `roots()` implementation that we provided you, and then come back and start working on your implementation.

Running `main()`

The `Quadratic` class also contains a `main` method. The method `public static void main(String[] args)` is the standard entry point for a Java program. In this case, the `main` method calls `roots()` on a simple quadratic equation.

To run `main`, right-click on `Quadratic.java` in the Package Explorer, and choose *Run As* → *Java Application*. You will see the results printed in the Console tab at the bottom of the Eclipse window. It will say "not implemented yet" if you haven't started writing `roots()`.

Run JUnit Tests

A `main` method is not a great way to test a program. Much better is automated unit testing, which runs a suite of tests to automatically test whether the implementations are correct.

JUnit is a widely-adopted Java unit testing library, and we will use it heavily in 6.005.

To find the tests for `Quadratic`, open up `ps1-warmup/test/ps1` in the Package Explorer, where you should find the file `QuadraticTest.java`. By convention, JUnit tests are put in a class whose name ends in `Test`, and 6.005 goes a step further by putting them in the `test/` folder rather than the `src/` folder.

To run the tests, right click on `QuadraticTest.java` in the Package Explorer, and choose *Run As* → *JUnit Test*. You should see the JUnit view appear.

If your implementation of `roots()` is correct, you should see a green bar, indicating that all the tests passed.

If your implementation is still broken or unfinished, you should see a red bar, and a list of the tests that were run. Clicking on a test shows a *stack trace* in the bottom box, which provides a brief explanation of what went wrong. Double-clicking on a line in the stack

trace goes to 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.

Use these tests to help you implement roots().

Run the Autograder

Once your implementation is passing all the JUnit tests, you need to run the automatic grader. The autograder is the file `grader.xml` in your project. It is written as an Ant script, where Ant is an example of a *build management tool*, a kind of tool commonly used in software development to compile code, run tests, and package up code for deployment. Our autograder script does all three of these things. Support for running Ant scripts is built into Eclipse.

To run the automatic grader, right click on `grader.xml` in the Package Explorer, and choose *Run As* → *Ant Build*. Choose the plain *Ant Build* rather than *Ant Build...*, because you don't need all the options that the Ant Build... dialog box will offer you.

The grading script will run and display output in the Console tab. At the end, it should tell you that it has created two new files: `my-grader-report.xml`, which is the result of running the grading tests, and `my-submission.zip`, which is your problem set code and grader output packaged up and ready for submission to edX.

Note: if automatic workspace refreshing is not working for some reason, then `my-grader-report.xml` and `my-submission.zip` may not immediately appear in the Eclipse Package Explorer. Try refreshing the project manually by right-clicking on `ps1-warmup` and choosing Refresh. Then you should see the files appear. If they still aren't there, check the Console tab for errors.

To view your autograder results, double-click on `my-grader-report.xml`, and you will see the results in your JUnit pane. The `my-grader-report.xml` file is simply the output of running JUnit on the autograder's tests. For now (Problem Set Beta 1), the autograder tests are identical to the tests you've already been using in `QuadraticTest.java`, so you should see exactly the same tests passing or failing that you see when you ran JUnit yourself as in the previous section. On future problem sets, the autograder will run different tests.

You can change your code and rerun the autograder as often as you want. You have to double-click on `my-grader-report.xml` each time to see the newest results. You don't need to Refresh the project each time, however. Once `my-grader-report.xml` is visible in the Package Explorer, Eclipse will load the latest version whenever you click on it.

Submit the Beta version of your Problem Set

After the autograder runs, it produces `my-submission.zip` in your Eclipse project. This zip file contains your code and your grading report. To get credit for the problem set, you need to upload this zip file to edX before the deadline.

The Problem Set Beta 1 submission page is the last section of this handout. To find it, click the rightmost button on the section bar at the top of this page:



Eclipse FAQ | Problem Set 1 Beta | 6.005.1x Courseware

III openlearninglibrary.mit.edu/courses/course-v1:MITx+6.005.1x+3T2016/courseware/Week_2/ps1-beta

1. [Course](#), [current location](#)
2. [Progress](#)

Eclipse FAQ

Eclipse Tips

Resources

[Java Debugging with Eclipse](#) introduces Eclipse's debug mode, a recommended tool for tracking down bugs using breakpoints. This step-by-step tutorial explains the *Debug perspective* and provides screenshots to decode the mysterious debugger icons in Eclipse.

Tips & Tricks

Navigating

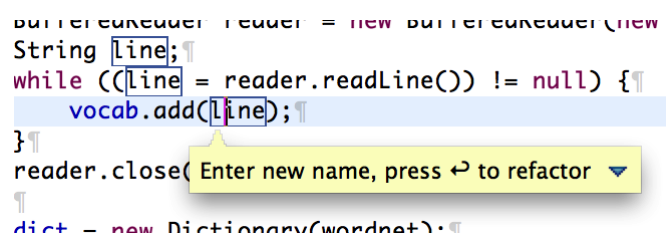
Command-shift-R and command-shift-T bring up a dialog for quickly searching for and **navigating to files** and **classes**, respectively.

Editing

Select an element and use **Refactor** → **Rename** (or its keyboard shortcut) to rename that variable, field, method, class, etc. wherever it appears.

Take advantage of **code completion** to help you use a new API: if you have an object `squiggle`, typing the name and then a period will bring up code completion options.

```
BufferedReader reader = new BufferedReader(new
String line;
while ((line = reader.readLine()) != null) {
    vocab.add(line);
}
reader.close();
dict = new Dictionary(wordnet);
```



Use **Source** → **Organize Imports** or command- (or control-) -shift-O to automatically add and organize Java import statements. Just start using a new class, *Organize Imports*, and you're good to go.

Use **Source** → **Toggle Comment** or keyboard shortcut Command - / to quickly toggle on or off some lines of code.

Add any tips you find helpful in your journey with Eclipse to the forum by clicking Show Discussion below and adding a post!

Problem Set 1 Final

III openlearninglibrary.mit.edu/courses/course-v1:MITx+6.005.1x+3T2016/courseware/Week_3/ps1-final

1. [Course](#), [current location](#)
2. [Progress](#)

Download the final grading tests for Problem Set 1 here:

| [ps1-final-grader.jar](#)

Drag the file into the root folder of your Eclipse ps1 project, and choose Copy File from the dialog.

To run the autograder with the new tests, right-click on `grader.xml` in Eclipse and choose Run As / Ant Build. The grader should automatically find both the beta and final tests, and its output in the Console tab should include the line:

| Your code has been compiled and run against the **BETA + FINAL** grading tests.

If you are still seeing just **BETA**, then make sure you dragged `ps1-final-grader.jar` to the right place. It should appear in Eclipse right after `ps1-beta-grader.jar`.

Problem Set 2 | Problem Set 2 Beta | 6.005.1x Courseware

III openlearninglibrary.mit.edu/courses/course-v1:MITx+6.005.1x+3T2016/courseware/Week_4/ps2-beta

Problem Set 2 Starting Code

Download the starting code for Problem Set 2 here:

| [ps2.zip](#)

The process for doing this problem set is the same as in Problem Set 1, but here are quick reminders:

- To import the starting code into Eclipse, use File → Import... → General → Existing Projects Into Workspace → Select Archive File, then Browse to find where you downloaded ps2.zip. Make sure `ps2-tweets` is checked and click Finish.
- To run JUnit tests, right-click on the `test` folder in Eclipse and choose Run As → JUnit Test.
- This problem set has no `main()` method to run, just tests.
- To run the autograder, right-click on `grader.xml` in Eclipse and choose Run As → Ant Build.
- To view the autograder results, make sure your project is Refreshed, then double-click on `my-grader-report.xml`.
- To submit your problem set, upload `my-submission.zip` to the submission page, which is the last section of this handout, at the end of the section bar.

Problem Set 2: Tweet Tweet

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.

Part of the point of this problem set is to learn how to write good tests. In particular:

- **Your test cases should be chosen using the input/output-space partitioning approach.** This approach is explained in the [reading about 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. The testing reading has an [example of documenting the testing strategy for a method](#).
- **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.** Your test cases will be run 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.** Your test cases will also be run 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 about [specifications](#) and [designing specifications](#):

- **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` and `Timespan` 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.

1. Devise, document, and implement test cases for `getTimespan()` and `getMentionedUsers()`, and put them in `ExtractTest.java`.
2. Implement `getTimespan()` and `getMentionedUsers()`, and make sure your tests pass.

Hints:

- Note that we use the class `Instant` to represent the date and time of tweets. You can check [this article on Java 8 dates and times](#) 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()`.
-

Problem 2: Filtering lists of tweets

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

1. Devise, document, and implement test cases for `writtenBy()`, `inTimespan()`, and `containing()`, and put them in `FilterTest.java`.
2. 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.
-

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).

1. 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.

2. 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.
-

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.** 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 the grader will run against staff implementations.

Understanding the Grader Output

III openlearninglibrary.mit.edu/courses/course-v1:MITx+6.005.1x+3T2016/courseware/Week_4/ps2-beta

1. [Course](#), [current location](#)
2. [Progress](#)

Understanding the Problem Set 2 Grader Output

When you run the grader for Problem Set Beta 2 and view the resulting `my-grader-report.xml`, you will see several categories of tests.

`twitter.staff.Original{Extract,Filter,SocialNetwork}Test` are tests we originally gave you with the problem set. These tests were in `test/twitter/{Extract,Filter,SocialNetwork}Test.java` before you started adding your own tests to those files. The tests are run against your own implementations of the methods.

`twitter.{Extract,Filter,SocialNetwork}Test` are your own tests run against your own implementations of the methods. The grader is running your own test code here, so it should produce the same results as you would get from right-clicking on the `test` folder and running JUnit.

`twitter.staff.{Extract,Filter,SocialNetwork}TestRunner` are running your own tests (from your `test` folder) against staff implementations of the methods. The staff implementations can be either *bad* (not following the spec) or *good* (correctly following the spec). For a bad implementation, the grader test passes if at least one of your tests rejects the bad implementation. For a good implementation, the grader test passes if all of your tests accept the good implementation. Test names in this category have the form `yourTests_staff[Bad|Good]Impl_[hint about how the implementation behaves]`. For example, `yourTests_staffBadImpl_Filter_alwaysReturnsEmpty` means that your tests were run against a bad implementation in which the `Filter` methods always return empty lists.

Problem Set 2 Final

III openlearninglibrary.mit.edu/courses/course-v1:MITx+6.005.1x+3T2016/courseware/Week_6/ps2-final

1. [Course](#), [current location](#)
2. [Progress](#)

Download the final grading tests for Problem Set 2 here:

| [ps2-final-grader.jar](#)

Drag the file into the root folder of your Eclipse ps2 project, and choose Copy File from the dialog.

To run the autograder with the new tests, right-click on `grader.xml` in Eclipse and choose Run As / Ant Build. The grader should automatically find both the beta and final tests, and its output in the Console tab should include the line:

| Your code has been compiled and run against the **BETA + FINAL** grading tests.

If you are still seeing just **BETA**, then make sure you dragged `ps2-final-grader.jar` to the right place. It should appear in Eclipse right after `ps2-beta-grader.jar`.

Problem Set 3 | Problem Set 3 Beta | 6.005.1x Courseware

III openlearninglibrary.mit.edu/courses/course-v1:MITx+6.005.1x+3T2016/courseware/Week_9/ps3-beta

Problem Set 3 Starting Code

Download the starting code for Problem Set 3 here:

| [ps3.zip](#)

The process for doing this problem set is the same as in Problem Set 1, but here are quick reminders:

- To import the starting code into Eclipse, use File → Import... → General → Existing Projects Into Workspace → Select Archive File, then Browse to find where you downloaded ps3.zip. Make sure `ps3-library` is checked and click Finish.
- To run JUnit tests, right-click on the `test` folder in Eclipse and choose Run As → JUnit Test.
- This problem set has no `main()` method to run, just tests.
- To run the autograder, right-click on `grader.xml` in Eclipse and choose Run As → Ant Build.
- To view the autograder results, make sure your project is Refreshed, then double-click on `my-grader-report.xml`.
- To submit your problem set, upload `my-submission.zip` to the submission page, which is the last section of this handout, at the end of the section bar.

Problem Set 3: The Librarians

Overview

The purpose of this problem set is to practice designing, testing, and implementing abstract data types. This problem set includes both immutable and mutable types. For one type (`SmallLibrary`), the representation of the type is specified, and you should implement its methods to match that rep. For other types, the representation is up to you to choose. The problem set also includes an example of two different implementations of the same Java interface. Finally, you'll be expected to implement equality appropriately for all the types.

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

1. Study the specifications of the type's operations carefully.
2. Write JUnit tests for the operations according to the spec.
3. Write the type's rep (its fields), document its rep invariant and abstraction function, and implement the rep invariant in a `checkRep()` method.
4. Implement the type's methods according to the spec.
5. Make an argument about why your rep is safe from rep exposure, and write it down in a comment.
6. Revise your implementation and improve your test cases until your implementation passes all your tests.

As in Problem Set 2, part of the point of this problem set is to learn how to write good tests for abstract data types, so the same expectations apply:

- Your test cases should be chosen using the input/output-space partitioning approach.
- Your test cases should be small and well-chosen.
- Your tests should find bugs. The Final grading test suite will include buggy implementations of the types, so your tests need to find those bugs.
- Your tests must be legal clients of the spec.

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 class names:** the classes `Book`, `BookCopy`, `Library`, `SmallLibrary`, `BigLibrary`, `BookTest`, `BookCopyTest`, `LibraryTest`, and `BigLibraryTest` must use those names and remain in the `library` package.
- **Don't change the method signatures and specifications:** The public methods provided for you to implement in `Book`, `BookCopy`, and `Library` must use the method signatures and the specifications that we provided.
- **Don't include illegal test cases:** The tests you implement in `BookTest`, `BookCopyTest`, and `LibraryTest` must respect the specifications that we provided for the methods you are testing.
- **Don't change the rep for `SmallLibrary`:** One type, `SmallLibrary`, has a required rep that you should not change. For the other types (`Book`, `BookCopy`, and `BigLibrary`) the rep is up to you.

Aside from these requirements, however, you are free to add new public and private methods and new public or private classes if you wish.

Problem 1: Book and BookCopy

The overall theme of this problem set is implementing a book catalog for a lending library. In this problem, you will test and implement the abstract data types `Book` and `BookCopy`. `Book` is an immutable type representing a published book (uniquely identified by its title, author list, and publication year). Since a library may have more than one copy of the

same book, we also have a mutable type `BookCopy` that represents one copy of a book. The operations and specs for `Book` and `BookCopy` are given in their source files, and should not be changed.

You'll find `Book.java` and `BookCopy.java` in the `src` folder, and their corresponding JUnit test classes `BookTest.java` and `BookCopyTest.java` in the `test` folder. as we did in previous problem sets.

1. Devise, document, and implement test cases for the operations of `Book`, and put them in `BookTest.java`.
2. Choose a representation for `Book`, and write down the rep invariant and abstraction function in a comment. These types are basically warmups, so your rep invariant and abstraction function will be very simple. Implement the rep invariant by writing assertions in the `checkRep()` method.
3. Implement the operations of `Book` using your rep. Make sure to call `checkRep()` at appropriate points, i.e. at the end of every creator, producer, and mutator operation. Also make sure to implement `equals()` and `hashCode()` as appropriate for an immutable type.
4. Convince yourself that your type is safe from rep exposure, and write your argument down in a comment after the abstraction function.
5. Finally, run your tests, and revise until your `Book` implementation passes your tests.

Repeat these same steps for `BookCopy`, taking care to note that it is a mutable type where `Book` is immutable:

1. Devise, document, and implement test cases for the operations of `BookCopy`, and put them in `BookCopyTest.java`.
2. Choose a representation for `BookCopy`, and write down the rep invariant and abstraction function in a comment. Again, these will be very simple for this class. Implement the rep invariant by writing assertions in the `checkRep()` method.
3. Implement the operations of `BookCopy`, including calls to `checkRep()` at appropriate points. Also make sure to implement `equals()` and `hashCode()` as appropriate for a mutable type.
4. Convince yourself that your type is safe from rep exposure, and write your argument down in a comment after the abstraction function.
5. Finally, run your tests, and revise until your `BookCopy` implementation passes your tests.

Problem 2: LibraryTest

The library catalog itself is represented by the `Library` type, which is a Java interface. In later problems on this problem set, you will implement two different implementations of this type:

- `SmallLibrary`, a simple brute-force representation that works well enough for small libraries, like a few hundred books owned by a single person.
- `BigLibrary`, a representation that performs better on bigger libraries.

In this problem, we'll just write tests for the `Library` interface itself, without caring which kind of library the tests are run on. The `LibraryTest` JUnit class has been designed so that it automatically runs twice, once using `SmallLibrary` and again using `BigLibrary`.

Devise, document, and implement test cases for the operations of `Library`, and put them in `LibraryTest.java`.

Problem 3: SmallLibrary

Now that we have some test cases, we're ready to implement `SmallLibrary`. The representation for `SmallLibrary` is already given in `SmallLibrary.java`, including its rep invariant and abstraction function.

1. Implement the rep invariant of `SmallLibrary` by writing assertions in the `checkRep()` method.
2. Implement the operations of `SmallLibrary`, including calls to `checkRep()` at appropriate points. Also make sure to implement `equals()` and `hashCode()` as appropriate for a mutable type.
3. Write down an argument that your type is safe from rep exposure.
4. Finally, run your `LibraryTest` tests, and revise until your `SmallLibrary` implementation passes your `LibraryTest` tests. Note that you won't get a full green light yet from JUnit, because it is also running `LibraryTest` against `BigLibrary`, which you haven't implemented yet.

Notes:

- **Don't change the rep.** You may find yourself tempted to add new fields to the `SmallLibrary` rep, but don't. The rep has been chosen to be as simple as possible, to make some Library operations very easy to implement (like `isAvailable()`) even though others are harder (like `find()`). Bear with that, and make it work. Starting with a simple brute-force rep allows you to debug your tests and your understanding of the spec. You'll invent a better rep in the next problem.

- **Ignore performance.** SmallLibrary is designed for very small book collections, so its operations can be very slow or use extra space. It's brute force. Just don't worry about performance, and focus on simplicity and correctness. Save your performance-optimization ideas for the next problem.
 - **Simplest possible functionality.** In particular, the spec for `find()` is underdetermined. Just do the minimum required for `SmallLibrary`. Save your ideas for making it better for the last problem.
-

Problem 4: BigLibrary

Now that we've implemented a simple version of the library, let's make it better by implementing `BigLibrary`. You will choose your own representation for `BigLibrary`. You can borrow ideas and code from your `SmallLibrary`, but your goal should be to make the operations of `BigLibrary` work efficiently even when there are millions of books in the library.

1. Choose a rep for `BigLibrary` and write down its rep invariant and abstraction function. You may want to use `Map` or `SortedSet` or other data structures in your rep. You may want to store information redundantly to save time or space when you're implementing the operations.
2. Implement the rep invariant of `BigLibrary` by writing assertions in the `checkRep()` method.
3. Implement the operations of `BigLibrary`, including calls to `checkRep()` at appropriate points. As usual, make sure to implement `equals()` and `hashCode()` as appropriate for a mutable type.
4. Write down an argument that your type is safe from rep exposure.
5. Finally, run your `LibraryTest` tests, and revise until your `BigLibrary` implementation passes your `LibraryTest` tests.

Notes:

- **Aim for constant-time or logarithmic-time performance.** Since `BigLibrary` is designed for big book collections, try to make its operations run in $O(1)$ or $O(\log N)$ time for a library with N books. Don't count the cost of `checkRep()`.
 - **Simplest possible functionality.** Again, start out by implementing only minimum required behavior for `find()`. Save your ideas for making it better for the final problem on this problem set.
-

Problem 5: Improving find()

Now improve your `BigLibrary` so that `find()` behaves more like a user would expect a library catalog interface to behave. Examples of reasonable behavior that is allowed by the `find()` spec include:

- Matching words in the `keywords` argument to words in title or author names.
- Ranking the resulting list of books so that books that match more keywords appear earlier in the list.
- Ranking books that match multiple contiguous keywords higher in the list.
- Ranking older books or checked-out books lower in the list.
- Supporting quotation marks in the `keywords` argument, so that (for example) `"\"David Foster Wallace\" \"Infinite Jest\""` finds books whose title or author contains David Foster Wallace or Infinite Jest as contiguous words.

When you add this new behavior to `BigLibrary.find()`, you should strengthen its spec accordingly, so that clients of `BigLibrary` can expect the behavior. Don't change `Library`'s spec, however, and leave your `LibraryTest` tests and `SmallLibrary` implementation unchanged. Instead, to test your new stronger behavior, you should put the new tests in `BigLibraryTest.java`.

1. Write down the spec for your new behavior for `BigLibrary.find()` as a Javadoc comment above the method, including preconditions and postconditions as appropriate.
 2. Devise, document, and implement test cases for your stronger `find()` spec in `BigLibraryTest`.
 3. Change the rep of `BigLibrary` as needed to handle your new behavior, update the rep invariant and abstraction function comments, and update the `checkRep()` method.
 4. Implement your new `BigLibrary.find()`.
 5. Finally, run all your tests, including the original `LibraryTest` tests and your new `BigLibraryTest` tests, and revise until your `BigLibrary` implementation passes your tests.
-

Problem Set 3 Final

III openlearninglibrary.mit.edu/courses/course-v1:MITx+6.005.1x+3T2016/courseware/Week_11/ps3-final

1. [Course](#), [current location](#)
2. [Progress](#)

Download the final grading tests for Problem Set 3 here:

| [ps3-final-grader.jar](#)

Drag the file into the root folder of your Eclipse ps3 project, and choose Copy File from the dialog.

To run the autograder with the new tests, right-click on `grader.xml` in Eclipse and choose Run As / Ant Build. The grader should automatically find both the beta and final tests, and its output in the Console tab should include the line:

| Your code has been compiled and run against the **BETA + FINAL** grading tests.

If you are still seeing just **BETA**, then make sure you dragged `ps3-final-grader.jar` to the right place. It should appear in Eclipse right after `ps3-beta-grader.jar`.