CHAPTER 1

Building Your First JUnit Test

In this chapter, we'll write a unit test by working through a small example. You'll set up your project, add a test class, and see what a test method looks like. Most importantly, you'll get JUnit to run your new, passing test.

Reasons to Write a Unit Test

Joe has just completed work on a small feature change, adding several dozen lines to the system. He's fairly confident in his change, but it's been a while since he's tried things out in the deployed system. Joe runs the build script, which packages and deploys the change to the local web server. He pulls up the application in his browser, navigates to the appropriate screen, enters a bit of data, clicks submit, and…stack trace!

Joe stares at the screen for a moment, then the code. Aha! Joe notes that he forgot to initialize a field. He makes the fix, runs the build script again, cranks up the application, enters data, clicks submit, and…hmm, that's not the right amount. Oops. This time, it takes a bit longer to decipher the problem. Joe fires up his debugger and after a few minutes discovers an off-by-one error in indexing an array. He once again repeats the cycle of fix, deploy, navigate the GUI, enter data, and verify results.

Happily, Joe's third fix attempt has been the charm. But he spent about fifteen minutes working through the three cycles of code/manual test/fix.

Lucia works differently. Each time she writes a small bit of code, she adds a *unit test* that verifies the small change she added to the system. She then runs all her unit tests, including the new one just written. They run in seconds, so she doesn't wait long to find out whether or not she can move on.

Because Lucia runs her tests with each small change, she only moves on when all the tests pass. If her tests fail, she knows she's created a problem

and stops immediately to fix it. The problems she creates are a lot easier to fix since she's added only a few lines of code since she last saw all the tests pass. She avoids piling lots of new code atop her mistakes before discovering a problem.

Lucia's tests are part of the system and included in the project's GitHub repository. They continue to pay off each time she or anyone else changes code, alerting the team when someone breaks existing behavior.

Lucia's tests also save Joe and everyone else on the team significant amounts of comprehension time on their system. "How does the system handle the case where the end date isn't provided?" asks Madhu, the product owner. Joe's response, more often than not, is, "I don't know; let me take a look at the code." Sometimes, Joe can answer the question in a minute or two, but frequently, he ends up digging about for a half hour or more.

Lucia looks at her unit tests and finds one that matches Madhu's case. She has an answer within a minute or so.

You'll follow in Lucia's footsteps and learn how to write small, focused unit tests. You'll start by learning basic JUnit concepts.

Learning JUnit Basics: Your First Testing Challenge

For your first example, you'll work with a small class named `CreditHistory`. Its goal is to return the mean (average) for a number of credit rating objects.

In this book, you'll probe the many reasons for choosing to write unit tests. For now, you'll start with a simple but critical reason: you want to continue adding behaviors to `CreditHistory` and want to know the moment you break any previously coded behaviors.

Initially, you will see screenshots to help guide you through getting started with JUnit. After this chapter, you will see very few screenshots, and you won't need them.

The screenshots demonstrate using JUnit in IntelliJ IDEA. If you're using another integrated development environment (IDE), the good news is that your JUnit test code will look the same whether you use IDEA, Eclipse, VSCode, or something else. How you set up your project to use JUnit *will* differ. The way the JUnit looks and feels will differ from IDE to IDE, though it will, in general, operate the same and produce the same information.

Here's the code you need to test:

utj3-credit-history/01/src/main/java/credit/CreditHistory.java

```
import java.time.LocalDate;
import java.time.Month;
import java.util.*;

public class CreditHistory {
   private final List<CreditRating> ratings = new ArrayList<>();

   public void add(CreditRating rating) {
      ratings.add(rating);
   }

   public int arithmeticMean() {
      var total = ratings.stream().mapToInt(CreditRating::rating).sum();
      return total / ratings.size();
   }
}
```

The CreditHistory class collects CreditRating objects through its add method. Its current primary goal is to provide you with an average (arithmeticMean) of the scores contained in the credit rating objects.

You implement CreditRating with a Java record declaring a single rating field.

utj3-credit-history/01/src/main/java/credit/CreditRating.java

```
public record CreditRating(int rating) {}
```

Your first exercise is small, and you could easily enter it from scratch. Typing in the code yourself should help you grow your coding skills faster. Still, you can also choose to download the source for this and all other exercises from https://pragprog.com/titles/utj3/pragmatic-unit-testing-in-java-with-junit-third-edition/.

Where to Put the Tests

Your project is laid out per the Apache Software Foundation's standard directory layout:[1]

```
utj3-credit-history
  src/
    main/
      java/
        credit/
          CreditHistory.java
          CreditRating.java
    test/
      java/
        credit/
```

1. https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

Your two production source files for this project are stored in the directory src/main/java in the package named credit. (IntelliJ IDEA refers to the directory src/main/java as a *Sources Root*.)

You're ready to write a test that describes the behavior in CreditHistory. You'll be putting the test in the same package as the production source—credit—but in the *Test Sources Root* directory src/test/java.

Your IDE probably provides you with many ways to create a new test class. In IDEA, you'll create it by following these steps in the Project explorer:

1. Select the package src/test/java/credit from the Project or Packages explorer.
2. Right-click to bring up the context menu.
3. Select New ▶ Java Class. You will see the New Java Class popup, which defaults its selection to creating a new class.
4. Type the classname ACreditHistory ("a credit history"); press enter. IDEA's inspections may be unhappy about your test naming convention. You can reconfigure the inspection,[2] or you can go with the old-school name CreditHistoryTest.

Running Tests: Testing Nothing at All

When you press enter from the New ▶ Java Class menu item, IDEA provides you with an empty class declaration for ACreditHistory. Your first job is to squeeze a test method into it:

utj3-credit-history/01/src/test/java/credit/ACreditHistory.java

```
class ACreditHistory {
    @org.junit.jupiter.api.Test
    void whatever() {
    }
}
```

To be a bit more specific: Within the body of ACreditHistory, type in the three lines that start with the @org.junit.jupiter.api.Test annotation.

Lines marked with arrows in code listings represent added lines, changed lines, or otherwise interesting bits of code.

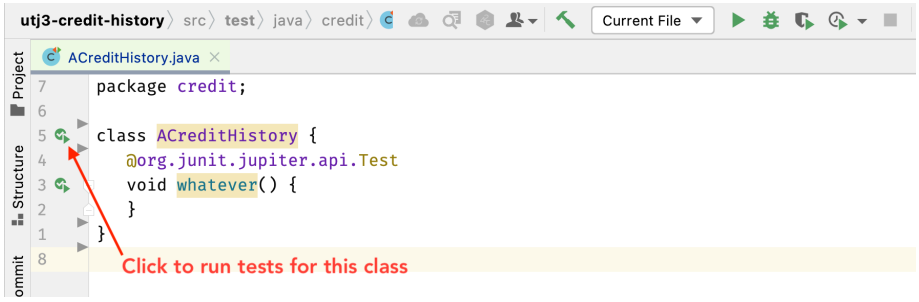
Type? Yes. It's better to type code and tests in yourself while learning, rather than copy/paste them, unless typing isn't at all your thing. It'll feel more like

2. https://langrsoft.com/2024/04/28/your-new-test-naming-convention/

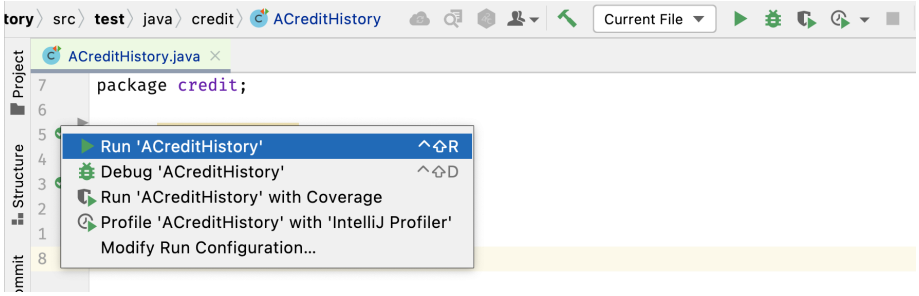
real development, which should help you learn more. It also won't take as long as you think. Your IDE offers numerous time-saving shortcuts, such as intellisense, live templates, and context-sensitive "quick fix."

Your test is an empty method annotated with the type @org.junit.jupiter.api.Test. When you tell JUnit to run one or more tests, it will locate all methods annotated with @Test and run them. It'll ignore all other methods.

You can run your empty test, which, for now, you've given a placeholder name of whatever. As usual, you have many options for executing tests. You'll start by being mousey. Click the little green arrow that appears to the left of the class declaration, as shown in the following figure. (Chances are good your IDE has a similar icon.)



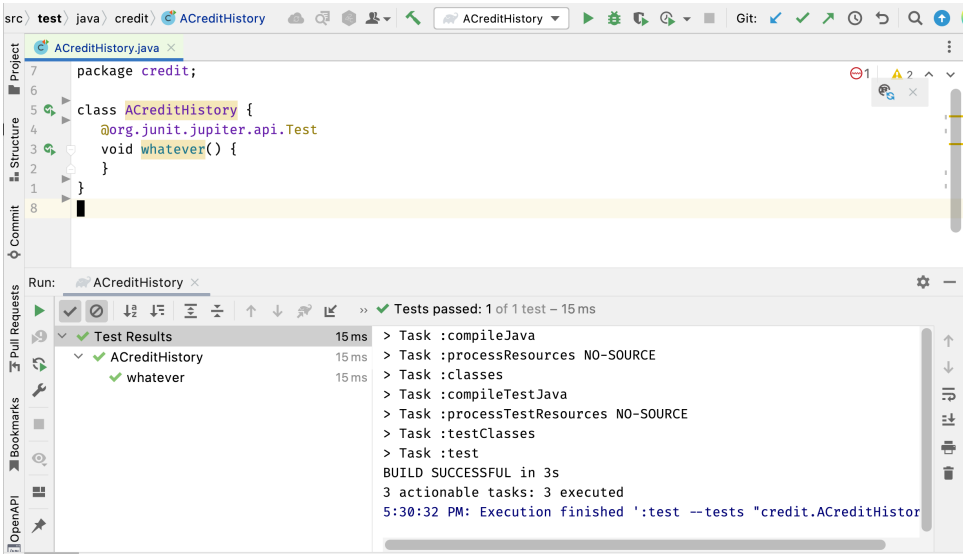
Clicking the green arrow pops up a context menu where you can select the option to run all tests in ACreditHistory, as shown in this figure:



Clicking Run 'ACreditHistory' runs the whatever test. It's passing, as the figure on page 8 reveals.

If your test isn't getting executed, make sure it follows these three guidelines:

- it is annotated with @org.junit.jupiter.api.Test
- it has a void return
- it has no parameters



The built-in JUnit test runner appears at the bottom of the IDE. Its left-hand panel shows a summary of all the tests executed. Your summary shows that you ran the `whatever` test within `ACreditHistory`, that it succeeded (because it has a green check mark), and that it took 12 milliseconds to execute.

The test runner's right-hand panel shows different information depending on what's selected in the left-hand panel. By default, it tells you how many tests passed out of the number that were executed (yours: "1 of 1"). It also provides you with information captured as part of the JUnit process execution. (In this screenshot, the IDE is configured to use Gradle to execute the test via the `build` task, which also executes the tests.)

You now know something fundamental about how JUnit behaves: an empty test passes. More specifically and more usefully, a test whose method execution completes—without having encountered any failure points or throwing any exceptions—is a passing test.

Writing a First Real Test

An empty test isn't of much use. Let's devise a good first test.

You could start with a meaty test that adds a few credit scores, asks for the average, and then ascertains whether or not you got the right answer. This *happy path* test case—in contrast with negative or error-based tests—is not the only test you'd want to write, though. You have some other cases to consider for verifying `arithmeticMean`:

- What happens if you add only one credit rating?
- What happens if you don't add any credit ratings?
- Are there any *exceptional cases*—conditions under which a problem could occur? How does the code behave under these conditions?

Starting with a happy path test is one choice; you have other options. One is to start with the simplest test possible, move on to incrementally more complex tests, and finally to exceptional cases. Another option is to start with the exceptional cases first, then cover happy path cases in complexity order. Other ordering schemes are, of course, possible.

When writing unit tests for code you've already written, ultimately, the order really doesn't matter. But if you follow a consistent approach, you'll be less likely to miss something. Throughout this book, the progression you'll prefer will be to start with the simplest case, then move on to incrementally more complex happy path cases, and then to exception-based tests.

The Simplest Possible Case

The simplest case often involves *zero* or some concept of *nothing*. Calculating the arithmetic mean involves creating a credit history with nothing added to it. You think that an empty credit history should return an average of zero.

Update ACreditHistory with the following code, which replaces the whatever test with a new one:

utj3-credit-history/02/src/test/java/credit/ACreditHistory.java

```
Line 1  import org.junit.jupiter.api.Test;
    -   import static org.junit.jupiter.api.Assertions.assertEquals;
    -
    -   class ACreditHistory {
    5       @Test
    -       void withNoCreditRatingsHas0Mean() {
    -           var creditHistory = new CreditHistory();
    -           var result = creditHistory.arithmeticMean();
    -           assertEquals(0, result);
   10       }
    -   }
```

Let's step through the updated lines in ACreditHistory.java.

Each of your tests will call one or more *assertion* methods to verify your assumptions about the system. That'll add up to piles of lines of assertions. Since these assertions are static methods, add a static import at 2 so that you don't have to constantly qualify your assertion calls.

Line 2: You simplify your test declaration by introducing an import statement for the @Test annotation.

The test name whatever wasn't much of a winner, so supply a new one at line 6. As with all tests you write, strive for a test name that summarizes what the test verifies. Here's a wacky idea: have the test name complete a sentence about the behavior it describes.

A Credit History...with no credit ratings...has a 0 mean

Your test describes a credit history object in a certain context—it has no credit ratings. You expect something to hold true about that credit history in that context: it has a zero mean. Your test name is a concise representation of that context and expected outcome:

```
ACreditHistory ... withNoCreditRatingsHas0Mean() { }
```

Was that a snort? No, you don't have to follow this test class naming convention, but it's as valid as any other. You'll read about alternative naming schemes at Documenting Your Tests with Consistent Names, on page 190.

On to the body of the code—where the work gets done. Your test first creates a CreditHistory instance (line 7). This new object allows your test to run from a clean slate, keeping it isolated from the effects of other tests. JUnit helps reinforce such isolation by creating a new instance of the test class—ACreditHistory—for each test it executes.

Your test next (at line 8) interacts with the CreditHistory test instance to exercise the behavior that you want to verify. Here, you call its arithmeticMean method and capture the return value in result.

Your test finally (at line 9) asserts that the expected (desired) result of 0 matches the actual result captured.

Your call to assertEquals uses JUnit's bread-and-butter assertion method, which compares a result with what you expect. The majority of your tests will use assertEquals. The rest will use one of many other assert forms that you'll learn in Chapter 5, Examining Outcomes with Assertions, on page 99.

An assertEquals method call *passes* if its two arguments match each other. It *fails* if the two arguments do not match. The test method as a whole fails if it encounters any assertion failures.

The hard part about learning assertEquals is remembering the correct order of its arguments. The value your test *expects* comes first; the *actual* value

returned by the system you're testing second. The signature for assertEquals makes the order clear. If you ever forget, use your IDE to show it to you:

```
public static void assertEquals(int expected, int actual)
```

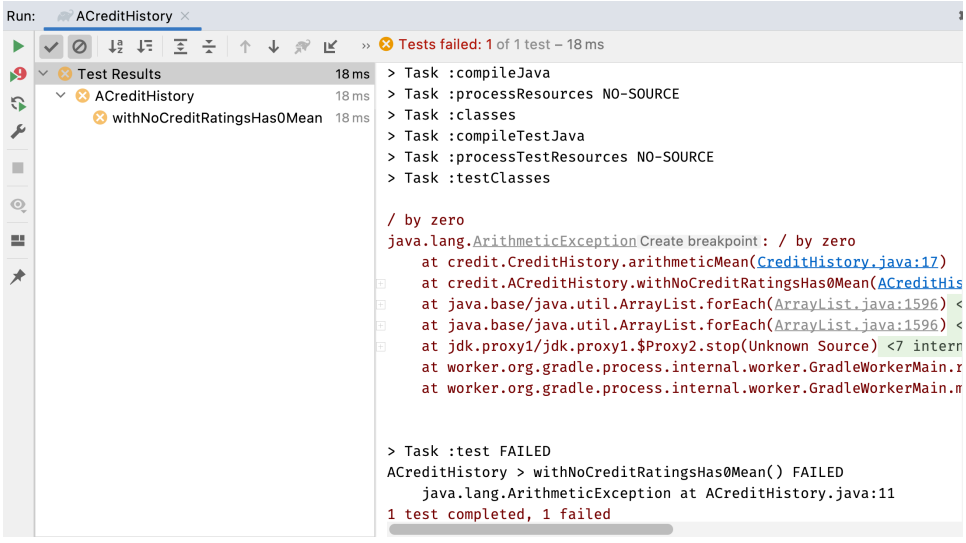
When you run your test, you'll see why the order for *expected* and *actual* arguments matters. You'll do that in the forthcoming section, Making It Fail, on page 14. Stick around!

Dealing with Failure

You previously learned to click on the little JUnit run icon next to the class declaration to execute all its tests. But you're going to be running tests quite often—potentially hundreds of times per day, and mousing about is a much slower, labor-intensive process. It behooves you to be more efficient. Repetitive stress injuries are real and unpleasant.

Any good IDE will show you the appropriate keyboard shortcut when you hover over a button. Hovering over the JUnit "run" button reveals Ctrl-Shift-R as the appropriate shortcut in my IDE. Hover over yours. Write down the shortcut it provides. Press it and run your tests. Press it again. And again. And remember it. And from here on out, for the thousands of times you will ultimately need to run your tests, use the keyboard. You'll go faster, and your tendons will thank you.

Your test is failing. Your JUnit execution should look similar to the following figure.



This information-rich view contains several pieces of information about your test's failing execution:

1. The JUnit panel to the left, which gives you a hierarchical listing of the tests executed, marks both the test class name ACreditHistory and the test method name withNoCreditRatingsHas0Mean with a yellow x. You can click on that test method name to focus on its execution details.

2. The JUnit panel to the right gets to the point with a statistical summary: Tests failed: 1 of 1 test That is, JUnit executed one test, and that sole test failed.

3. Below that redundantly phrased summary, JUnit shows the gory execution details for the test. The failure left behind an exception stack trace that tells you the test barfed before even reaching its assertion statement.

4. The stack trace screams at you in red text—the favored color of items designed to alert, like errors, stop signs, and poisoned lipstick. You have a divide-by-zero problem. The stack trace is linked to appropriate lines in the source, which allows you to quickly navigate to the offending code:

```
public int arithmeticMean() {
   var total = ratings.stream().mapToInt(CreditRating::rating).sum();
   return total / ratings.size(); // oops!
}
```

Your test added no credit ratings to the CreditHistory. As a result, ratings.size() returns a 0, and Java throws an ArithmeticException as its way of telling you it wants nothing to do with that sort of division. Oops!

Your exception-throwing test reveals another useful JUnit nugget: if code executed in a test run throws an exception that's not caught, that counts as a failing test.

Fixing the Problem

The unit test did its job: it notified you of a problem. Earlier, you decided that it's possible someone could call arithmeticMean before any credit ratings are added. You also decided that you don't want the code to throw an exception in that case; you instead want it to return a 0. The unit test captures and documents your choice.

Your unit test will continue to protect you from future *regressions*, letting you know anytime the behavior of arithmeticMean changes.

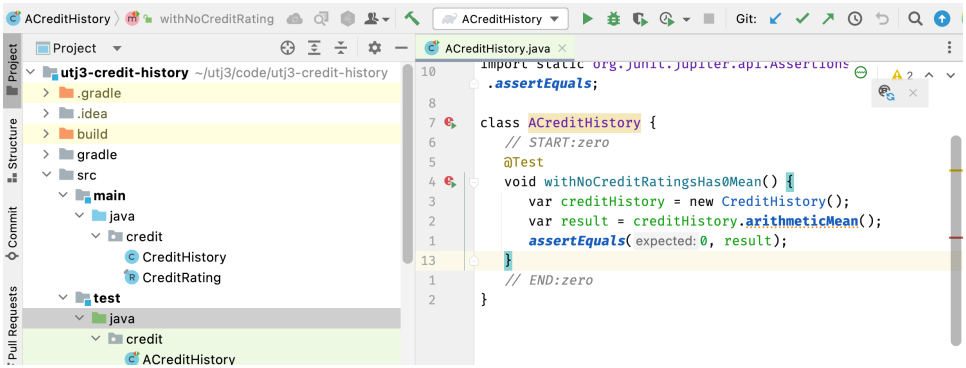
To get the failing test to pass—to fix your problem—add a *guard clause* to the arithmeticMean method in CreditHistory:

utj3-credit-history/03/src/main/java/credit/CreditHistory.java

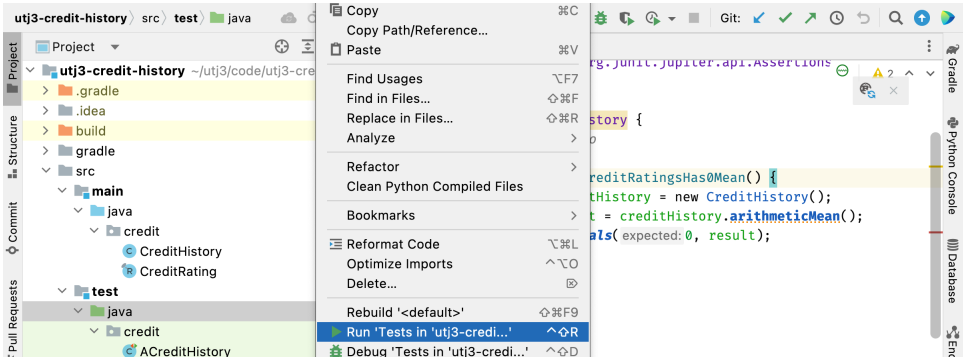
```
public int arithmeticMean() {
    if (ratings.isEmpty()) return 0;

    var total = ratings.stream().mapToInt(CreditRating::rating).sum();
    return total / ratings.size();
}
```

Run the tests again to see if your change did the trick. This time, kick them off by using the Project view (usually the upper-left-most tool window in IDEA and other IDEs). Drill down from the project at its top level until you can select the test/java directory, as shown in this figure:

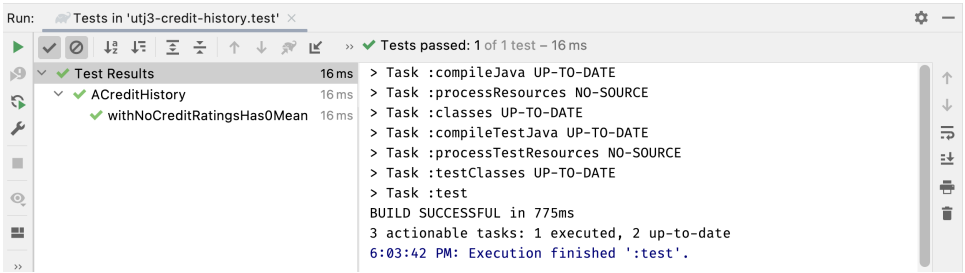


A right-click brings up a near-freakishly large context menu:



Select the option Run 'All Tests'. JUnit will execute all the tests within src/test/java. Success! Here's the passing test (as shown in the figure on page 14), where everything is a glorious green and devoid of stack trace statements.

Looks good, right? Feels good, right? Go ahead and hit that Ctrl-Shift-R keystroke (or its equivalent on your machine) to run the test again. Bask in the glory.



Moving On to a One-Based Test: Something's Happening!

Your zero-based test saved your bacon. Maybe a one-based test can do the same? Write a test that adds one and only one credit score:

utj3-credit-history/04/src/test/java/credit/ACreditHistory.java

```
@Test
void withOneRatingHasEquivalentMean() {
   var creditHistory = new CreditHistory();
   creditHistory.add(new CreditRating(780));
   var result = creditHistory.arithmeticMean();
   assertEquals(780, result);
}
```

You might have quickly put that test in place by duplicating the zero-based test, adding a line to call creditHistory.add(), and changing the assertion.

Your new test passes. Are you done with it? No. Two critical steps remain:

1. Ensure you've seen it fail.
2. Clean it up.

Making It Fail

If you've never seen a test fail for the right reason, don't trust it.

The test you just wrote contains an assertion that expects arithmeticMean to return a specific value. "Failing for right reason" for this example would mean that arithmeticMean returns some value other than 780 (the expected value). Perhaps the calculation is incorrect, or perhaps the code never makes the calculation and returns some initial value.

You want to break your code so that the test fails. When it fails, ensure that the failure message JUnit provides makes sense. Let's try that.