

Unit Testing - Individual Exercise

The purpose of this exercise is to provide you with opportunity to practice the art of [unit testing](#) a codebase that does not include any automated tests.

Learning Objectives

After completing this exercise, students will understand:

- How to write unit tests in a "legacy" codebase.
- How unit testing can be used to show the code is functioning correctly.
- How to structure unit tests in an organized, readable format.
- Why unit tests are important.
- How to write readable unit tests.

Evaluation Criteria & Functional Requirements

Code without tests is **bad code**. It doesn't matter how well written it is; it doesn't matter how pretty or object-oriented or well-encapsulated it is. With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse." ~ Michael Feathers, *Working Effectively with Legacy Code*

You've been hired as a new developer at Acme Inc. As such, you have inherited some [legacy code](#) that does not currently have any unit tests. Your job will be to:

- Create unit tests for all classes to ensure that the code is tested.

Your code will be evaluated based on the following criteria:

- The project must not have any build errors.
- Unit tests pass as expected.
- There is appropriate code coverage to verify the application code is functioning as expected.
- Good test method names are provided that clearly state what is being tested.

Getting Started

- Import the unit-testing-exercises project into Eclipse.
- Create a test class for the class you are going to be testing. For instance, if you are testing the StringBits class, you will create a class in the test/java/com/techelevator directory called StringBitsTests.
- Write test methods in the test class to verify the class under test is working as expected.

Tips and Tricks

- On your journey as a software developer, you will often find that you have inherited code that is untested, brittle, and riddled with ~~bugs~~ opportunities. One of the best things you can do for your benefit and for the benefit of the teams you work with is to write unit tests for any code you are modifying. This will help to remove [broken windows](#), which ultimately makes for better software, which yields happier customers, which yields greater revenues for your company.

- While it might seem counter intuitive at first, unit testing will actually make you a faster developer. For instance, consider the command line applications you have worked on over the past few exercises. While it is possible to test these applications manually, it takes quite a bit of time to make a code change, start the application, click through the menus, manually review the results, and verify your code worked as expected. A unit test automates this effort, is repeatable, and is therefore more reliable. It is also faster. Get into the habit of writing good unit tests, and you will ship code faster and more reliably than developers who don't.
 - Test methods should clearly state what is being tested in the method name. For instance, if you were to verify that an Add method returns 4 when it is passed 2 and 2, then the name of the test method should be something like `add_should_return_4_when_2_and_2_are_passed`. Yes, this is verbose, but verbose methods are preferred in unit tests, as they clearly articulate what is being tested.
 - There are some [best practices you should follow when writing unit tests](#).
-