Unit testing

The purpose of this exercise is to provide you with opportunity to practice unit testing a codebase that doesn't include any automated tests.

Learning objectives

After completing this exercise, you'll understand:

- How to write unit tests in a "legacy" codebase.
- How unit testing can show that 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 and 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 doesn't have any unit tests. Your job is 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 that the application code functions as expected.
- Good test method names are provided that clearly state what is being tested.

Getting started

1. Import the unit testing exercises project into IntelliJ.
2. Create a test class for the class you'll test. For instance, if you're testing the `StringBits` class, create a class in the `test/java/com/techelevator` directory called `StringBitsTests`.
3. Write test methods in the test class to verify the class under test works as expected.

Tips and tricks

Unit testing reduces regressions

Real applications grow over time, and it can be difficult, if not impossible, to foresee all the consequences of changing code. New features may make assumptions regarding the behavior of existing components which inadvertently alters the application's behavior. Bug fixes can fix one thing, but break something else. Unit tests can help detect these **regressions**.

> Note: Regression has several meanings in software engineering, but in general, it means what the word implies: the *opposite of progress*. Whatever change occurred, it made things worse.

Ideally, as you build your application, you're scaffolding it with unit tests. For every new block of code you write, or change you make, you need to write and run unit tests. Provided you run all the tests, both new and old, together, any changes you made that inadvertently broke something should be revealed. *Unit testing can't prevent regressions, but it can alert you to them.*

Unit testing can make you a faster developer

Consider the command line applications you worked on over the past few exercises. Although you can test these applications manually, you'd have to do several things:

1. Make a code change
2. Start the application
3. Click through the menus
4. Manually review results
5. Verify that code works as expected

A unit test automates this effort, is repeatable, quicker, and is therefore more reliable.

Writing good unit tests helps you ship code faster and more reliably than developers who don't.

Naming matters

Test methods must clearly state what's being tested in the method name.

For instance, if you want to verify that an Add method returns 4 when it's 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`. This is verbose, but verbose methods are preferred in unit tests, as they clearly articulate what's being tested.

There are some other best practices to follow when writing unit tests, too.
