CHAPTER 4

Organizing Your Tests

Prior chapters have given you enough unit-testing fodder to hit the ground running. The problem with hitting the ground running, however, is that you're leaping into a new, unfamiliar environment, and you're liable to make a few wrong turns or even hurt yourself in the process.

Better that you take a few minutes to pick up a few pointers on what you're getting into. This chapter introduces a few JUnit features as part of showing you how to best organize and structure your tests.

Some of the topics you'll read about include:

- How to make your tests visually consistent using arrange-act-assert
- Keeping tests maintainable by testing behavior, not methods
- The importance of test naming
- Using `@Before` and `@After` for common initialization and cleanup needs
- How to safely ignore tests getting in your way

Keeping Tests Consistent with AAA

When we wrote tests for the first iloveyouboss example on page 3, we visually organized our tests into three chunks: arrange, act, and assert, also known as triple-A (AAA).

```
@Test
public void answersArithmeticMeanOfTwoNumbers() {
   ScoreCollection collection = new ScoreCollection();
   collection.add(() -> 5);
   collection.add(() -> 7);

   int actualResult = collection.arithmeticMean();

   assertThat(actualResult, equalTo(6));
}
```


Back then, we added comments to identify each of the chunks explicitly, but these comments add no value once you understand the AAA idiom.

AAA is a part of just about every test you'll write. With AAA, you:

- *Arrange*. Before we execute the code we're trying to test, ensure that the system is in a proper state by creating objects, interacting with them, calling other APIs, and so on. In some rare cases, we won't arrange anything, because the system is already in the state we need.
- *Act*. Exercise the code we want to test, usually by calling a single method.
- *Assert*. Verify that the exercised code behaved as expected. This can involve inspecting the return value of the exercised code or the new state of any objects involved. It can also involve verifying that interactions between the tested code and other objects took place.

The blank lines that separate each portion of a test are indispensable visual reinforcement to help you understand a test even more quickly.

You might need a fourth step:

- *After*. If running the test results in any resources being allocated, ensure that they get cleaned up.

Testing Behavior Versus Testing Methods

When you write tests, focus on the behaviors of your class, not on testing the individual methods.

To understand what that means, think about the tedious but time-tested example of an `ATM` class for a banking system. Its methods include `deposit()`, `withdraw()`, and `getBalance()`. We might start with the following tests:

- `makeSingleDeposit`
- `makeMultipleDeposits`

To verify the results of each of those tests, you need to call `getBalance()`. Yet you probably don't want a test that focuses on verifying the `getBalance()` method. Its behavior is probably uninteresting (it likely just returns a field). Any interesting behavior requires other operations to occur first—namely, deposits and withdrawals. So let's look at the `withdraw()` method:

- `makeSingleWithdrawal`
- `makeMultipleWithdrawals`
- `attemptToWithdrawTooMuch`