Pragmatic Unit Testing
in Java with JUnit
Third Edition

Jeff Langr

edited by Kelly Talbot

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Using the Core Assertion Forms

The bulk of your assertions will use either assertTrue or assertEquals. Let's review and refine your knowledge of these two assertion workhorses. Let's also see how to keep your tests streamlined by eliminating things that don't add value.

The Most Basic Assertion Form: assertTrue

The most basic assert form accepts a boolean expression or reference as an argument, and fails the test if that argument evaluates to false.

```
org.junit.jupiter.api.Assertions.assertTrue(someBooleanExpression);
```

Here's an example demonstrating the use of assertTrue:

```
utj3-junit/01/src/test/java/scratch/AnAccount.java
@Test
void hasPositiveBalanceAfterInitialDeposit() {
    var account = new Account("an account name");

    account.deposit(50);

    Assertions.assertTrue(account.hasPositiveBalance());
}
// ...
```

Technically you could use assertTrue for every assertion you had to write. But an assertTrue failure tells you only that the assertion failed and nothing more. Seek more precise assertions such as assertEquals, which reports what was expected vs. what was actually received when it fails. You'll find test failures easier to understand and resolve as a result.

Eliminating Clutter

As documents that you'll spend time reading and re-reading, you'll want to streamline your tests. You learned in the first chapter (see Chapter 1, Building Your First JUnit Test, on page ?) that the public keyword is unnecessary when declaring both JUnit test classes and test methods. Such additional keywords and other unnecessary elements represent *clutter*.

> Streamline your tests by eliminating unnecessary clutter.

You'll be scanning lots of tests to gain rapid understanding of what your system does and where your changes must go. Getting rid of clutter makes it easier to understand tests at a glance.

Asserts pervade JUnit tests. Rather than explicitly scope each assert call with the class name (Assertion), use a static import:

```java
import static org.junit.jupiter.api.Assertions.assertTrue;
```

The result is a de-cluttered, more-concise assertion statement:

utj3-junit/01/src/test/java/scratch/AnAccount.java
```java
@Test
void hasPositiveBalanceAfterInitialDeposit() {
    var account = new Account("an account name");

    account.deposit(50);

    assertTrue(account.hasPositiveBalance());
}
```

Generalized Assertions

Here's another example of assertTrue which explains how a result relates to some expected outcome:

utj3-junit/01/src/test/java/scratch/AnAccount.java
```java
@Test
void depositIncreasesBalance() {
    var account = new Account("an account name");
    var initialBalance = account.getBalance();

    account.deposit(100);

    assertTrue(account.getBalance() > initialBalance);
}
```

A test name—depositIncreasesBalance—is a general statement about the behavior you want the test to demonstrate. Its assertion—assertTrue(balance > initialBalance)—corresponds to the test name, ensuring that the balance has increased as an outcome of the deposit operation. The test does not *explicitly* verify by how much the balance increased. As a result, you might describe its assert statement as a *generalized* assertion.

Eliminating More Clutter

The preceding examples depend on the existence of an initialized Account instance. You can create an Account in a @BeforeEach method (see Initializing with @BeforeEach and @BeforeAll, on page ? for more information) and store a reference to it as a field on the test class:

utj3-junit/02/src/test/java/scratch/AnAccount.java
```java
class AnAccount {
    Account account;

    @BeforeEach
```

```java
   void createAccount() {
      account = new Account("an account name");
   }

   @Test
   void hasPositiveBalanceAfterInitialDeposit() {
      account.deposit(50);

      assertTrue(account.hasPositiveBalance());
   }
   // ...
}
```

JUnit creates a new instance of the test class for each test (see Observing the JUnit Lifecycle, on page ? for further explanation>). That means you can also safely initialize fields at their point of declaration:

utj3-junit/03/src/test/java/scratch/AnAccount.java
```java
class AnAccount {
   Account account = new Account("an account name");

   @Test
   void hasPositiveBalanceAfterInitialDeposit() {
      // ...
   }
   // ...
}
```

Use assertEquals for Explicit Comparisons

Your test names should be generalizations of behavior, but each test should present a specific example with a specific result. If the test makes a deposit, you know what the new balance amount should be. In most cases, you should be explicit with your assertion and verify the actual new balance.

The assertion assertEquals compares an expected answer to the actual answer, allowing you to explicitly verify an outcome's value. It's overloaded so that you can appropriately compare all primitive types, wrapper types, and object references. (To compare two arrays, use assertArrayEquals instead.) Most of your assertions should probably be assertEquals.

Here's the deposit example again, asserting by how much the balance increased:

utj3-junit/01/src/test/java/scratch/AnAccount.java
```java
@Test
void depositIncreasesBalanceByAmountDeposited() {
   account.deposit(50);

   account.deposit(100);

   assertEquals(150, account.getBalance());
```

```
}
```

You design the example for each test and you know the expected outcome. Encode it in the test with `assertEquals`.