How to Make Good JUnit Tests

Using JUnit 5 tools

What are Unit Tests?

Wikipedia: Unit tests are typically automated tests written and run by software developers to ensure that a section of an application (known as the "unit") meets its design and behaves as intended. … To isolate issues that may arise, each test case should be tested independently.

What is JUnit?

Wikipedia: JUnit is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development. … JUnit is linked as a JAR at compile-time. … A research survey performed in 2013 across 10,000 Java projects hosted on GitHub found that JUnit was the most commonly included external library [at 30.7%].

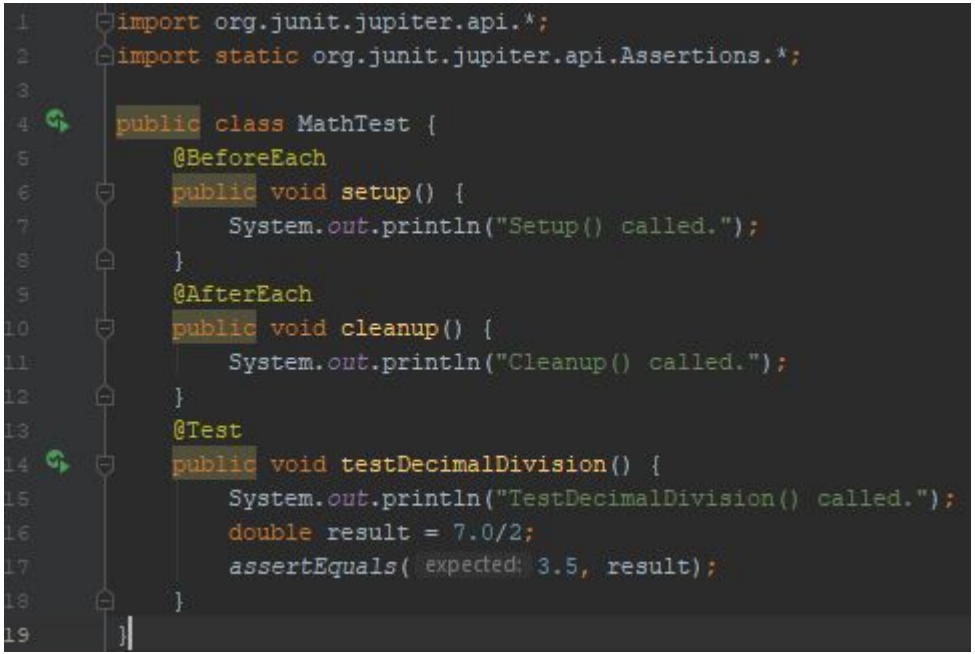
Basic Structure of JUnit Test

Setup: Labeled with a `@BeforeEach` tag, this function does any setting up of variables and test state so that each test can run without having to worry about initializing the its own state. It is called before every test.

Test: Labeled with `@Test tag` (and optionally `@DisplayName` to give it a name), this function tests one aspect of one function. Generally, it will set up parameters to the function, call the function, and then ensure that the function executed correctly (using assertions).

Cleanup: Labeled with a `@AfterEach` tag, this function does any finalizing or resetting of variables and test state, so that the next test case can be set up without crashing (for example, closing a file that was opened). It is called after every test, regardless of whether it passes or fails.

Simple Example

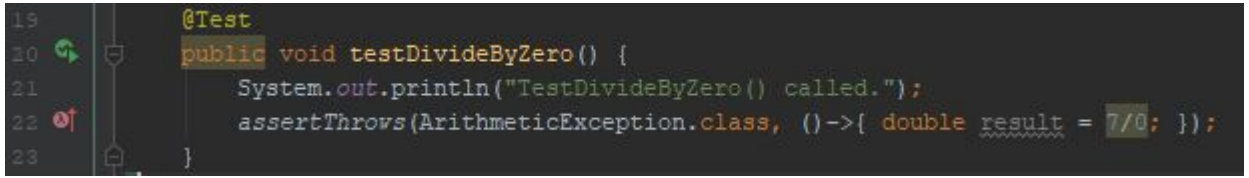


```java
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

public class MathTest {
    @BeforeEach
    public void setup() {
        System.out.println("Setup() called.");
    }
    @AfterEach
    public void cleanup() {
        System.out.println("Cleanup() called.");
    }
    @Test
    public void testDecimalDivision() {
        System.out.println("TestDecimalDivision() called.");
        double result = 7.0/2;
        assertEquals( expected: 3.5, result);
    }
}
```

When you run this test you should have the following printed in the console:

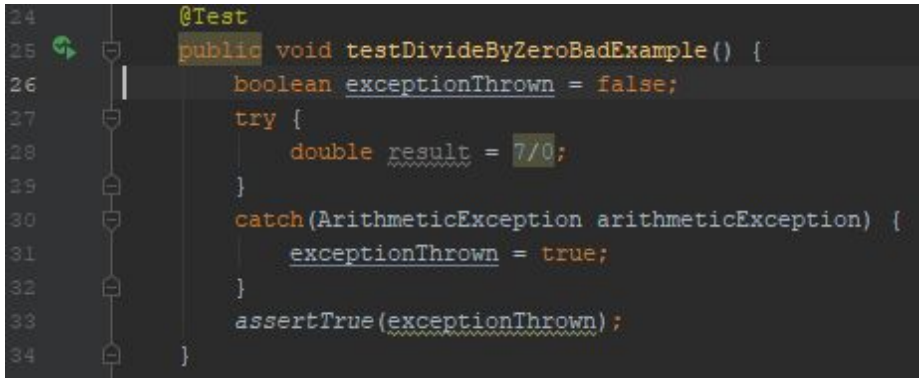
```
Setup() called.
TestDecimalDivision() called.
Cleanup() called.

Process finished with exit code 0
```

How do you Check if a Function Throws an Exception?



`assertThrows()` takes 2 parameters. The first parameter is the class of what exception we expect the function that we are testing to throw. The second parameter is a lambda function that contains the function call that we are testing. Lambda functions are out of the scope of this class, but all you need to know is that this assertion runs the code in the lambda function and fails if the class in the first parameter is not thrown.

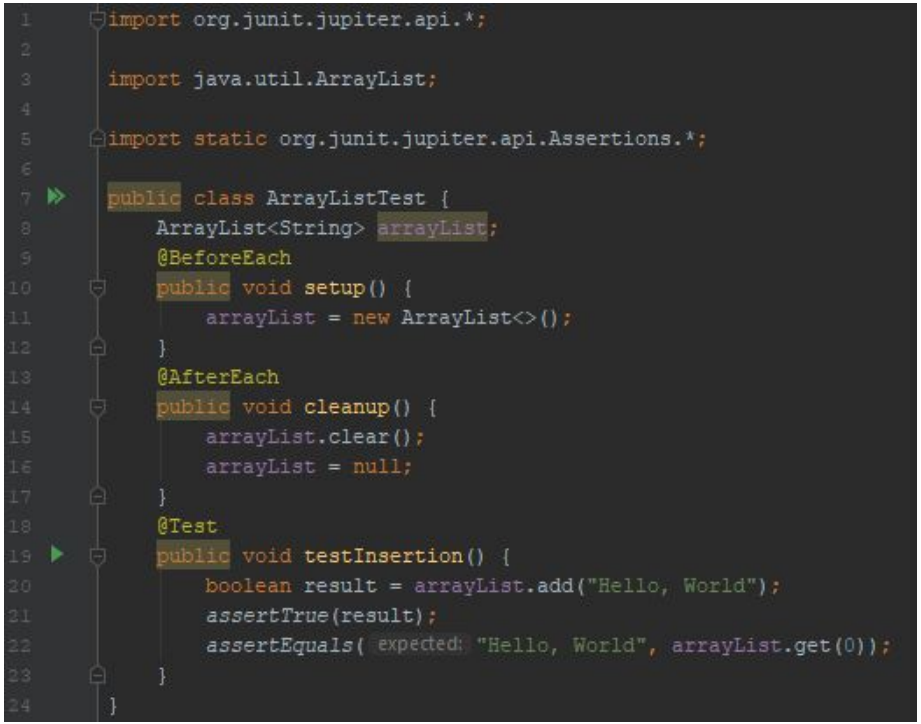
Bad Example of Checking for Thrown Exception



```
24      @Test
25      public void testDivideByZeroBadExample() {
26          boolean exceptionThrown = false;
27          try {
28              double result = 7/0;
29          }
30          catch(ArithmeticException arithmeticException) {
31              exceptionThrown = true;
32          }
33          assertTrue(exceptionThrown);
34      }
```

Note how difficult it is to understand what is being tested in this example. While it is functionally the same as the previous example, it takes 7 more lines of code to express and it is much less clear.

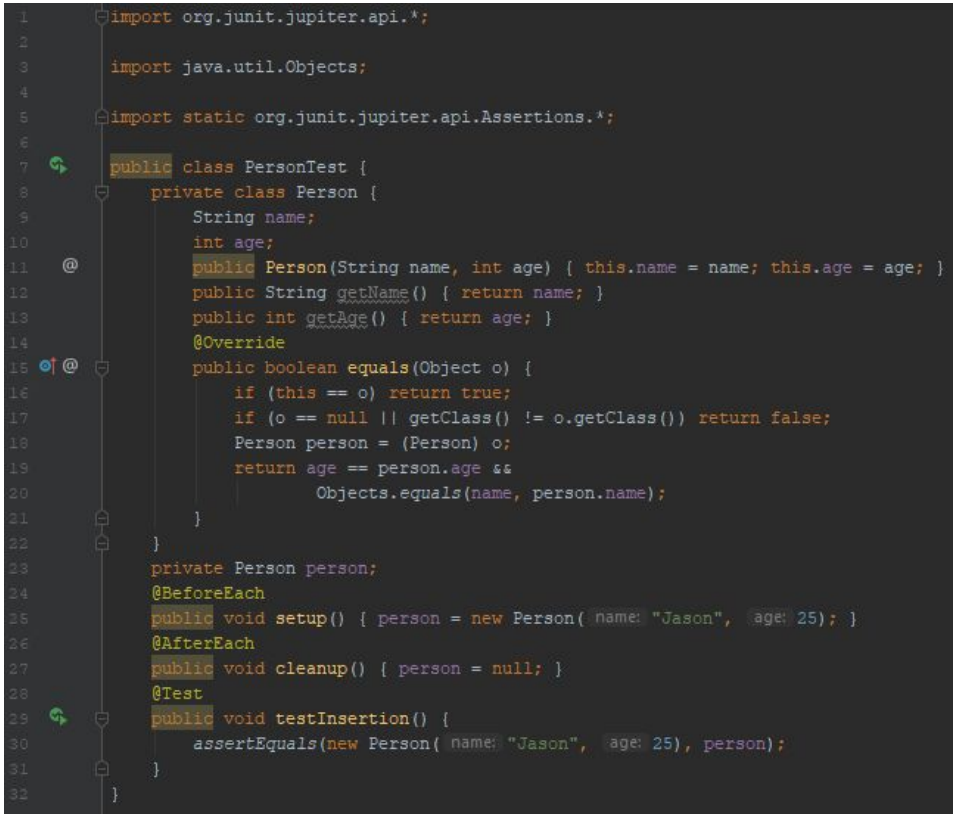
How do you Check for Correct Function Execution?



```
1  import org.junit.jupiter.api.*;
2
3  import java.util.ArrayList;
4
5  import static org.junit.jupiter.api.Assertions.*;
6
7  public class ArrayListTest {
8      ArrayList<String> arrayList;
9      @BeforeEach
10     public void setup() {
11         arrayList = new ArrayList<>();
12     }
13     @AfterEach
14     public void cleanup() {
15         arrayList.clear();
16         arrayList = null;
17     }
18     @Test
19     public void testInsertion() {
20         boolean result = arrayList.add("Hello, World");
21         assertTrue(result);
22         assertEquals( expected: "Hello, World", arrayList.get(0));
23     }
24 }
```

It is often not acceptable to just make sure that the function returned the right value (see line 21). Whenever possible, you should verify with another function call or by accessing the state of objects acted upon (see line 22).

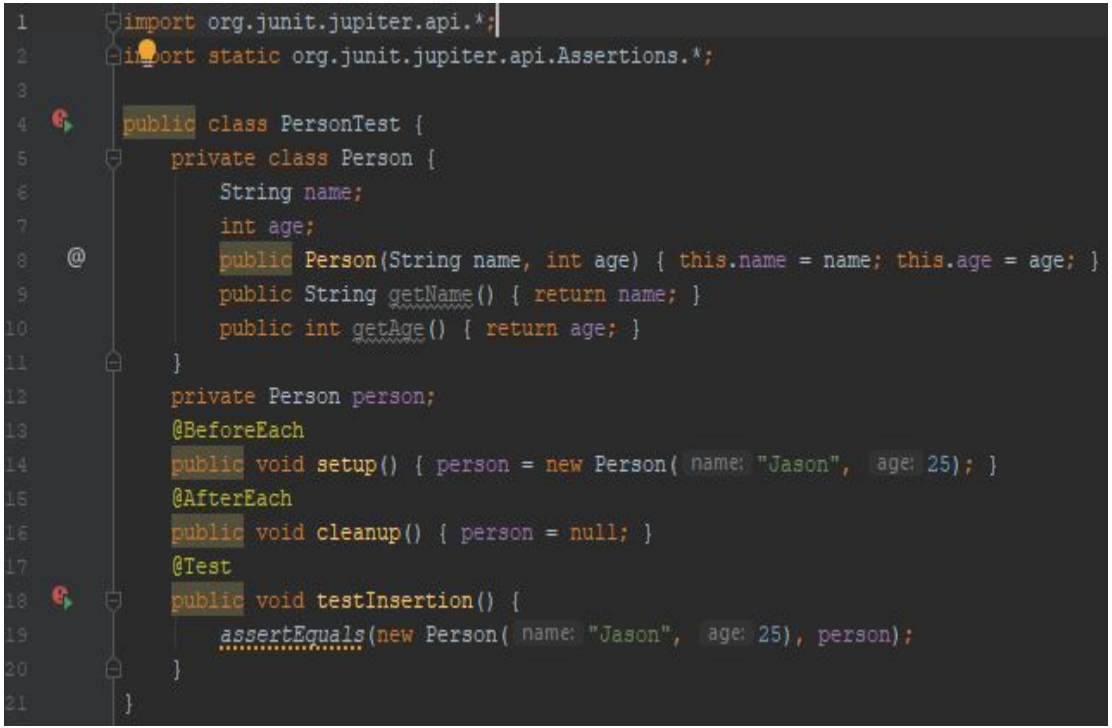
How do you Use AssertEquals?



```java
import org.junit.jupiter.api.*;

import java.util.Objects;

import static org.junit.jupiter.api.Assertions.*;

public class PersonTest {
    private class Person {
        String name;
        int age;
        public Person(String name, int age) { this.name = name; this.age = age; }
        public String getName() { return name; }
        public int getAge() { return age; }
        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
            if (o == null || getClass() != o.getClass()) return false;
            Person person = (Person) o;
            return age == person.age &&
                    Objects.equals(name, person.name);
        }
    }
    private Person person;
    @BeforeEach
    public void setup() { person = new Person( name: "Jason",  age: 25); }
    @AfterEach
    public void cleanup() { person = null; }
    @Test
    public void testInsertion() {
        assertEquals(new Person( name: "Jason",  age: 25), person);
    }
}
```

The `assertEquals()` function takes two parameters: the expected value and the actual value. If the expected value has an `equals()` function defined, it passes the actual value to that and checks that it returns true. If the expected value does not have an `equals()` function defined, it will simply compare the pointer values (very common source of bugs).

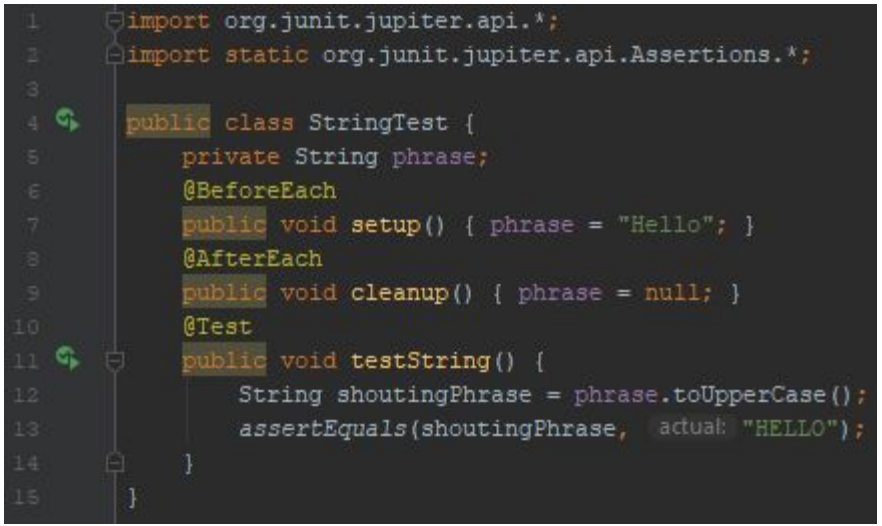
Bad Examples of AssertEquals



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

public class PersonTest {
    private class Person {
        String name;
        int age;
        public Person(String name, int age) { this.name = name; this.age = age; }
        public String getName() { return name; }
        public int getAge() { return age; }
    }
    private Person person;
    @BeforeEach
    public void setup() { person = new Person( name: "Jason",  age: 25); }
    @AfterEach
    public void cleanup() { person = null; }
    @Test
    public void testInsertion() {
        assertEquals(new Person( name: "Jason",  age: 25), person);
    }
}
```

Notice that the Person class does not implement the `equals()` method. The test case will not work as expected because it will compare the two objects' memory addresses and fail because the objects are equal, but distinct, objects in memory.

Bad Examples of AssertEquals



```
1  import org.junit.jupiter.api.*;
2  import static org.junit.jupiter.api.Assertions.*;
3
4  public class StringTest {
5      private String phrase;
6      @BeforeEach
7      public void setup() { phrase = "Hello"; }
8      @AfterEach
9      public void cleanup() { phrase = null; }
10     @Test
11     public void testString() {
12         String shoutingPhrase = phrase.toUpperCase();
13         assertEquals(shoutingPhrase,  actual: "HELLO");
14     }
15 }
```

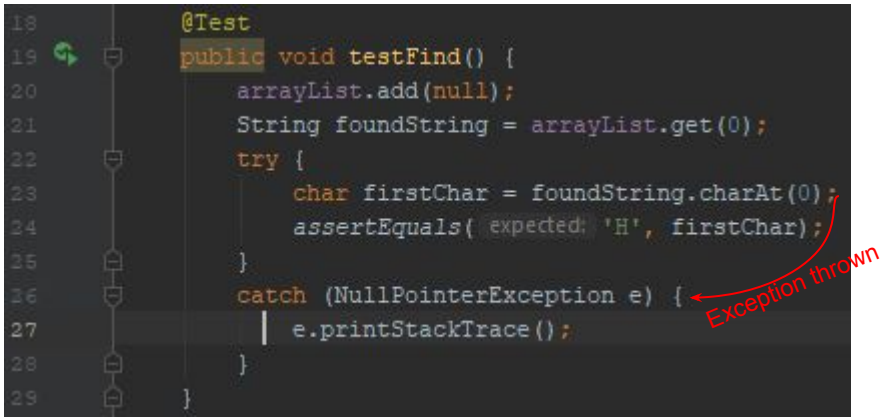
This one is tricky because it passes the test and seems to be okay. Alas, the problem with this example is that the intention is not clearly communicated. Note that on line 13 the label `"actual:"` is referring to the string literal `"HELLO"`. This tells the reader that the actual value is the string literal the value that we are testing is the expected, when the reverse is true. These two parameters should be switched.

How do you Ensure a Test Passes only when Correct?

```
18      @Test
19      public void testFind() {
20          arrayList.add(null);
21          String foundString = arrayList.get(0);
22          try {
23              char firstChar = foundString.charAt(0);
24              assertEquals( expected: 'H', firstChar);
25          }
26          catch (NullPointerException e) {
27              fail(e.getMessage());
28          }
29      }
```

Make sure that all possible branches are covered in your tests. Note the call to `fail()` on line 27. A common mistake is to just print the exceptions stack trace and continue execution. The problem in this case is that if execution continued, the test case would never fail and assertion and would thus pass the test even though it executed incorrectly.

Bad Examples of Branch Coverage



This time, we only printed the stack trace instead of calling `fail()`. You can see that now the test case passes, even though it failed to execute the way we wanted it to! This is because a `NullPointerException` is thrown at line 23, and caught at line 26. Line 24 is never executed, so it does not fail any asserts. Make sure to cover all of your test's branches!

How do you use the Right Assert?

Common Asserts:

- `assertEquals(expected, actual)` //Asserts that expected and actual are equal
- `assertFalse(condition)` //Asserts that the supplied condition is not true
- `assertNotEquals(unexpected, actual)` //Asserts that expected and actual are not equal
- `assertNotNull(actual)` //Asserts that actual is not null.
- `assertNull(actual)` //Asserts that actual is null.
- `assertThrows(expectedType, executable)` //Asserts that execution of the supplied executable throws an exception of the expectedType and returns the exception
- `assertTrue(condition)` //Asserts that the supplied condition is true
- `fail()` //Fails a test with the given failure message
- `assertDoesNotThrow(executable)` //Assert that execution of the supplied executable does not throw any kind of exception.

Note: Links to JUnit API for each function provided

Note: Each function above also has a signature with a String message parameter. It is a good idea to use this version of each function to give a description about what would cause this assertion to fail.

Bad Example of Assert Selection

```
10      @Test
11      public void testString() {
12          String shoutingPhrase = phrase.toUpperCase();
13          assertTrue( condition: shoutingPhrase.length() == 5);
14      }
```

```
10      @Test
11      public void testString() {
12          String shoutingPhrase = phrase.toUpperCase();
13          assertEquals( expected: false, shoutingPhrase.contains("he"));
14      }
```

This first example should use `assertEquals()` instead of `assertTrue()`. The second example should use `assertFalse()` instead of `assertEquals()`.

Remember: Always ask yourself if there is another `assert` function that is more **clear** or more **concise** than what you're using!