Contents

Introduction

JUnit is a framework for writing unit tests in Java.

When writing a JUnit, it is necessary to import the desired methods and annotations that will be used for testing.

A common set of imports is provided below:

```
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.assertNull;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertSame;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertArrayEquals;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.assertFalse;
```

Breakdown of Annotations and Methods

Annotations

@Before

The `@Before` annotation is added to any method that should be run before a test is executed. This is most often used to initialize instance variables that are used in the tests. A method annotated with the `@Before` annotation is run before *each* individual test method is run.

```
import org.junit.Before;

public class MyDataStructureTests {
    private MyDataStructure testStructure;

    @Before
    public void setUp() {
        testStructure = new MyDataStructure();
    }

}
```

@Test

The `@Test` annotation is added to any method that should be run as a unit test. These are the methods that give feedback by giving information about if and how the test code failed. An important thing to remember is that there is no predetermined order in which these test methods run. This is because each of the test methods are meant to be independent of each other. This annotation has two optional arguments: `timeout` and `expected`.

`timeout` is used to set the maximum running time (in milliseconds) that a test will be allowed to run. If a test runs longer than this value, it is flagged as failed with a `TimeoutException`.

`expected` is used to check for thrown exceptions. If a test expects an exception but one of the desired type is not thrown or no exception is thrown, it is flagged as failed with an `ExpectedException`.

```
import org.junit.Before;
```

```
2  import org.junit.Test;
3
4  public class MyDataStructureTests {
5      private MyDataStructure testStructure;
6
7      @Before
8      public void setUp() {
9          testStructure = new MyDataStructure();
10     }
11
12     @Test
13     public void aNormalTest() {
14     }
15
16     @Test(timeout = 100)
17     public void testWithTimeout() {
18         while (true) {
19             // Fails after 100 ms because of the timeout
20         }
21     }
22
23     @Test(expected = IllegalArgumentException.class)
24     public void expectIllegalArgument() {
25         // Test passes only if the following exception is thrown at
26         // some point in the test and is propagated up.
27         throw new IllegalArgumentException("Missing args");
28     }
29
30     @Test(timeout = 100, expected = IllegalArgumentException.class)
31     public void expectIllegalArgumentWithTimeout() {
32     }
33
34 }
```

Test Methods

assertNull()

The `assertNull` method checks whether the `Object` passed as the parameter is `null`. If the parameter is `null`, the test passes; otherwise the test will fail and the line number of the failed assertion will be provided in the logs.

```
1  import org.junit.Before;
2  import org.junit.Test;
3  import static org.junit.Assert.assertNull;
4
5  public class MyDataStructureTests {
6      private MyDataStructure testStructure;
7
8      @Before
9      public void setUp() {
10         testStructure = new MyDataStructure();
11     }
12
```

```
13      @Test
14      public void testFirstItemIsNull() {
15          assertNull(testStructure.getFirst());
16      }
17
18  }
```

assertNotNull()

The `assertNotNull` method checks whether the `Object` passed as the parameter is not `null`. If the parameter is not `null`, the test passes; otherwise the test will fail and the line number of the failed assertion will be provided in the logs.

```
1  import org.junit.Before;
2  import org.junit.Test;
3  import static org.junit.Assert.assertNotNull;
4
5  public class MyDataStructureTests {
6      private MyDataStructure testStructure;
7
8      @Before
9      public void setUp() {
10         testStructure = new MyDataStructure();
11     }
12
13     @Test
14     public void testFirstItemIsNotNull() {
15         assertNotNull(testStructure.getFirst());
16     }
17
18 }
```

assertSame()

The `assertSame` method checks whether the two `Objects` passed as parameters are the exact same `Object` (in other words, using ==). If the parameters are not references to the same object, the test will fail and the line number of the failed assertion will be provided in the logs.

```
1  import org.junit.Before;
2  import org.junit.Test;
3  import static org.junit.Assert.assertSame;
4
5  public class MyDataStructureTests {
6      private MyDataStructure testStructure;
7
8      @Before
9      public void setUp() {
10         testStructure = new MyDataStructure();
11     }
12
13     @Test
14     public void testAssertFirstItemSame() {
15         assertSame(testStructure.getFirst(), testStructure.getFirst());
16     }
17
```

```
18      @Test
19      public void testAssertDataStructureSame() {
20          // Guaranteed to fail
21          assertSame(new MyDataStructure(), testStructure.getFirst());
22      }
23
24  }
```

assertEquals()

The `assertEquals` method checks whether the two `Objects` passed as parameters are equal using the `equals` method of the first parameter. If the parameters are not equal, the test will fail and the line number of the failed assertion will be provided in the logs.

```
1  import org.junit.Before;
2  import org.junit.Test;
3  import static org.junit.Assert.assertEquals;
4
5  public class MyDataStructureTests {
6      private MyDataStructure testStructure;
7
8      @Before
9      public void setUp() {
10         testStructure = new MyDataStructure();
11     }
12
13     @Test
14     public void testAssertFirstItemEquals() {
15         assertEquals(new MyDataStructure(), testStructure);
16     }
17
18 }
```

assertArrayEquals()

The `assertArrayEquals` method checks whether the two arrays passed in are equal in length and that, for each index i, `expected[i].equals(actual[i])`. If either of these conditions are not true, the test will fail and the line number of the failed assertion will be provided in the logs.

```
1  import org.junit.Before;
2  import org.junit.Test;
3  import static org.junit.Assert.assertArrayEquals;
4
5  public class MyDataStructureTests {
6      private MyDataStructure testStructure;
7
8      @Before
9      public void setUp() {
10         testStructure = new MyDataStructure();
11     }
12
13     @Test
14     public void testAssertArrayIsEmpty() {
15         assertArrayEquals(new Object[0], testStructure.toArray());
16     }
```

```
17
18 }
```

assertTrue()

The `assertTrue` method checks whether the boolean passed as a parameter is `true`. If the parameters is not `true`, the test will fail and the line number of the failed assertion will be provided in the logs.

```
1  import org.junit.Before;
2  import org.junit.Test;
3  import static org.junit.Assert.assertTrue;
4
5  public class MyDataStructureTests {
6      private MyDataStructure testStructure;
7
8      @Before
9      public void setUp() {
10         testStructure = new MyDataStructure();
11     }
12
13     @Test
14     public void testAssertFirstItemExists() {
15         assertTrue(testStructure.hasFirst());
16     }
17
18 }
```

assertFalse()

The `assertFalse` method checks whether the boolean passed as a parameter is `false`. If the parameters is not `false`, the test will fail and the line number of the failed assertion will be provided in the logs.

```
1  import org.junit.Before;
2  import org.junit.Test;
3  import static org.junit.Assert.assertFalse;
4
5  public class MyDataStructureTests {
6      private MyDataStructure testStructure;
7
8      @Before
9      public void setUp() {
10         testStructure = new MyDataStructure();
11     }
12
13     @Test
14     public void testAssertFirstItemDoesNotExist() {
15         assertFalse(testStructure.hasFirst());
16     }
17
18 }
```

Best Practices for Writing JUnits

For the JUnit methods that take in two parameters, the first parameter is the "expected", or correct value, while the second parameter is the "actual", or tested value. Be sure to follow this convention so

that test logs are easy to read.

```
    assertEquals(expectedObject, actualObject);
    assertSame(expectedObject, actualObject);
```

It is more effective to write many, short, specific tests rather than a few, large, general tests. This will allow mistakes made in the classes that are being tested to be easily spotted.

```
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.assertFalse;

public class MyDataStructureTests {
    private MyDataStructure testStructure;

    @Before
    public void setUp() {
        testStructure = new MyDataStructure();
    }

    @Test
    public void doNotDoThis() {
        testStructure.addFirst(1);
        testStructure.addFirst(2);
        testStructure.addFirst(3);
        testStructure.addFirst(4);
        assertTrue(testStructure.hasFirst());
        assertTrue(testStructure.hasLast());
        testStructure.clear();
        assertFalse(testStructure.hasFirst());
        assertFalse(testStructure.hasLast());
        testStructure.addFirst(1);
        testStructure.addFirst(4);
        testStructure.addFirst(2);
        testStructure.addFirst(3);
        assertTrue(testStructure.hasFirst());
        assertTrue(testStructure.hasLast());
    }

    @Test
    public void doDoThis() {
        testStructure.addFirst(1);
        testStructure.addFirst(2);
        testStructure.addFirst(3);
        testStructure.addFirst(4);
        assertTrue(testStructure.hasFirst());
        assertTrue(testStructure.hasLast());
    }

}
```

In order to save time and headaches, make the names of the test methods as descriptive as possible. You may want to prefix the name with something like `test` so that you can easily jump to that test when that test fails.

```
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.assertTrue;

public class MyDataStructureTests {
    private MyDataStructure testStructure;

    @Before
    public void setUp() {
       testStructure = new MyDataStructure();
    }

    @Test
    public void testHasFirstByAddingElement() {
       testStructure.addFirst(1);
       assertTrue(testStructure.hasFirst());
    }

    @Test
    public void testHasFirstByAddingManyElements() {
       testStructure.addFirst(1);
       testStructure.addFirst(2);
       testStructure.addFirst(3);
       testStructure.addFirst(4);
       assertTrue(testStructure.hasFirst());
    }

}
```

In the case of an assertion failure for any of the test methods covered above, you can have a message be printed that may be helpful to the user. This message should be the first argument of the method.

```
    assertEquals("Root nodes aren't equal", expectedObject,
       actualObject);
    assertSame("Root nodes aren't the same", expectedObject,
       actualObject);
```

When writing tests that deal with data structures, be sure to test the contents of the entire data structures. Sometimes, unwanted changes can occur in parts of the data structures that should have been left untouched.

```
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.assertArrayEquals;

public class MyDataStructureTests {
    private MyDataStructure testStructure;

    @Before
    public void setUp() {
       testStructure = new MyDataStructure();
    }

    @Test
    public void testBackingArrayHasFirstByAddingManyElement() {
```

```
15          testStructure.addFirst(1);
16          testStructure.addFirst(2);
17          testStructure.addFirst(3);
18
19          int[] backingArray = testStructure.getBackingArray();
20          int[] expectedArray = new int[3];
21
22          expectedArray[0] = 1;
23          expectedArray[1] = 2;
24          expectedArray[2] = 3;
25
26          assertArrayEquals(backingArray, expectedArray);
27
28      }
29
30  }
```

Make sure that complete branch coverage is met. For each method of the class being tested, determine which order of method calls or which parameters are needed to be passed for each branch of the method to have occurred. Once these things are determined, write a single test for each branching situation.

Make them deterministic. Avoid relying on randomly generated values, or if randomly generated values are required, be sure to use a seed that will allow the same sequence of numbers to be generated for each run of the tests.

Avoid using static variables. Since the order in which the tests are run is not predetermined (and in fact the tests may be run in parallel), it is unwise to create assertions based on static variables contained within the test class.

Limit the use of `assertTrue` and `assertFalse` to only testing whether a method or variable is `true` or `false`. In the case of an assertion failure, the error messages generated are not helpful.

Know what methods are already provided to you, and use those methods. For example, all implementations of the `java.util.List` interface have a standard definition of `equals()`. Use this instead of checking the size and object equality yourself.

Avoid looking at your code or "testing" your tests by running your code. The best unit tests are written by figuring out the expected results for a given input on paper and then creating assert statements comparing an expected result to the actual result.

Make sure that all edge cases are covered. Try running methods in an uncommon or strange order and seeing if anything breaks.

Most Importantly