JUnit Cheatsheet

Learning a new concept can be overwhelming so we've compiled a cheatsheet you can reference while you write your own JUnit tests!

Creating a JUnit Test Class and JUnit Test Case

To create a JUnit test class, make sure you import `org.junit.jupiter.api.*` and `static org.junit.jupiter.api.Assertions.*`. These will give you access to method annotations like `@Test` and `@BeforeEach` and assertion methods like `assertTrue()` and `assertFalse()`.

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

public class ExampleTestClass {
    @Test
    public void yourTestCase() {
        // Assertion methods called here
    }
}
```

JUnit Method Annotations

Method annotations are used by JUnit so that JUnit knows how to treat your methods. Before you write a method, you must attach a method annotation. This special syntax is then interpreted by JUnit to know how to execute your method.

`@Test`: Turns a public method into a JUnit test case.

```java
@Test
public void test() {
    ...
}
```

`@Timeout(time)`: Times the test such that the test will fail after `time` milliseconds. Thus, the code must finish execution before `time`. Note that you still need the `@Test`.

```java
// Test will fail after 1000 ms
@Test
@Timeout(1000)
public void test() {
    ...
}
```

`@BeforeEach`: The method will be executed before each `@Test`

```
private int num; // This is a field

// This method will execute before each @Test
@BeforeEach
public void setUp() {
    num = 0;
}

@Test
public void test() {
    assertSame(0, num);
    num++;
    assertSame(1, num);
}
```

JUnit Assertion Methods

Assertion methods are the building blocks of JUnit and how you will write testing code inside your test methods. When you use an assertion method, the result needs to match up with what the assertion method expects, otherwise, your test will fail. Below are the most common types of assertion methods that you will use:

`assertTrue(test)`: Fails if the `test` is `false`

```
@Test
public void test() {
    int x = 2;
    String s = "Hello World";
    assertTrue(true);
    assertTrue(x == 2);
    assertTrue(s.equalsIgnoreCase("Hello World"));
}
```

`assertFalse(test)`: Fails if the `test` is `true`

```
@Test
public void test() {
    int x = 2;
    String s = "Hello World";
    assertFalse(false);
    assertFalse(x != 2);
    assertFalse(s.contains("a"));
}
```

`assertEquals(expected, actual)`: Fails if the `expected` and `actual` are not equal

```
@Test
public void test() {
```

```java
    String s1 = "Hello World";
    String s2 = "Hello World";
    String s3 = "Hello World";
    assertEquals(s1, s2);
    assertEquals(s2, s3);
    assertEquals(s3, s1);

    List<Integer> list1 = new ArrayList<>();
    List<Integer> list2 = new ArrayList<>();
    for (int i = 1; i <= 5; i++) {
        list1.add(i);
        list2.add(i);
    }
    assertEquals(list1, list2);
}
```

`assertSame(expected, actual)`: Fails if the `expected` and `actual` are not equal using reference semantics (==)

```java
@Test
public void test() {
    int x = 2;
    assertSame(2, x);

    List<Integer> list1 = new ArrayList<>();
    List<Integer> list2 = list1;
    assertSame(list1, list2);
}
```

`assertNotSame(expected, actual)`: Fails if `expected` and `actual` are equal using reference semantics (==)

```java
@Test
public void test() {
    int x = 2;
    assertNotSame(3, x);

    List<Integer> list1 = new ArrayList<>();
    List<Integer> list2 = new ArrayList<>();
    assertNotSame(list1, list2);
}
```

`assertNull(value)`: Fails if `value` is non-null

```java
@Test
public void test() {
    Map<String, Integer> map = new HashMap<>();
    map.put("cse122", 1);
    assertNull(map.get("Hello World"));

    String s = null;
    assertNull(s);
}
```

`assertNotNull(value)`: Fails if `value` is null

```java
@Test
public void test() {
    Map<String, Integer> map = new HashMap<>();
    map.put("cse122", 1);
    assertNotNull(map.get("cse122"));

    String s = "Hello World";
    assertNotNull(s);
}
```

`assertArrayEquals(Any[] expectedValues, Any[] actualValues)`: Fails if `expectedValues` and `actualValues` do not have the same elements, in the same order.

```java
@Test
public void test() {
    int[] a = new int[] {1, 2, 3};
    int[] b = new int[] {1, 2, 3};
    assertArrayEquals(a, b);
}
```

`assertThrows(exception.class, () -> {code})`: Fails if `code` does not throw `exception`

```java
@Test
public void test() {
    List<Integer> list = new ArrayList<>();
    assertThrows(IndexOutOfBoundsException.class, () -> {
        list.get(2); // List is currently: []
    });

    assertThrows(IndexOutOfBoundsException.class, () -> {
        list.add(1); // List is currently: [1]
        list.add(2); // List is currently: [1, 2]
        list.add(3); // List is currently: [1, 2, 3]
        list.remove(3); // Index 3
    });

}
```

Using JUnit to test Java's ArrayList Implementation:

Below is an example of a JUnit testing class that tests Java's ArrayList implementation:

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

public class ArrayListTest {
    private static final int TIMEOUT = 2000;
    private List<String> list;

    @BeforeEach
    public void setUp() {
        list = new ArrayList<>();
    }

    @Test
    @Timeout(TIMEOUT)
    public void testAddingElements() {
        assertTrue(list.isEmpty());
        list.add("Hunter Schafer");
        list.add("Miya Natsuhara");
        list.add("CSE 122");

        assertEquals("Hunter Schafer", list.get(0));
        assertEquals("Miya Natsuhara", list.get(1));
        assertEquals("CSE 122", list.get(2));

        assertTrue(list.size() == 3);
    }

    @Test
    public void testContains() {
        assertTrue(list.isEmpty());
        list.add("CSE 122");

        assertTrue(list.contains("CSE 122"));
        assertFalse(list.contains("Hello World"));
    }

    @Test
    public void testNegativeIndexGet() {
        assertTrue(list.isEmpty());
        assertThrows(IndexOutOfBoundsException.class, () -> list.get(-1));
    }
}
```