JUnit Testing Cheatsheet

A concise reference for writing effective unit tests in Java using JUnit. Covers annotations, assertions, test fixtures, and best practices for robust testing.

JUnit Fundamentals

Core Annotations

`@Test`	Marks a method as a test case. JUnit will execute this method when running tests.
`@BeforeEach` (JUnit 5) / `@Before` (JUnit 4)	Specifies a method to be executed before each test method in the class. Used for setting up test fixtures.
`@AfterEach` (JUnit 5) / `@After` (JUnit 4)	Specifies a method to be executed after each test method in the class. Used for tearing down test fixtures.
`@BeforeAll` (JUnit 5) / `@BeforeClass` (JUnit 4)	Specifies a method to be executed once before any of the test methods in the class are executed. Must be static.
`@AfterAll` (JUnit 5) / `@AfterClass` (JUnit 4)	Specifies a method to be executed once after all of the test methods in the class have been executed. Must be static.
`@Disabled` (JUnit 5) / `@Ignore` (JUnit 4)	Marks a test method as disabled/ignored. The test will not be executed.

Basic Assertions

`assertEquals(expected, actual)`	Asserts that two values are equal. Can be used with various data types.
`assertTrue(condition)`	Asserts that a condition is true.
`assertFalse(condition)`	Asserts that a condition is false.
`assertNull(object)`	Asserts that an object is null.
`assertNotNull(object)`	Asserts that an object is not null.
`assertSame(expected, actual)`	Asserts that two objects refer to the same object.
`assertNotSame(expected, actual)`	Asserts that two objects do not refer to the same object.

Exception Testing

`assertThrows(expectedType, executable)` - Asserts that the execution of the supplied executable throws an exception of the expected type.

```java
@Test
void testException() {
    IllegalArgumentException exception = assertThrows(IllegalArgumentException.class, () -> {
        throw new IllegalArgumentException("Invalid argument");
    });
    assertEquals("Invalid argument", exception.getMessage());
}
```

Advanced Assertions & Features

Advanced Assertions (JUnit 5)

`assertAll(executables...)`

Asserts that all supplied executables do not throw exceptions. Useful for grouping multiple assertions.

```
@Test
void
testMultipleAssertions() {
    assertAll(
        () ->
assertEquals(2, 1 + 1),
        () -> assertTrue(5
> 3)
    );
}
```

`assertTimeout(duration, executable)`

Asserts that the execution of the supplied executable completes before the given timeout.

```
@Test
void testTimeout() {

assertTimeout(Duration.ofS
econds(1), () -> {
        Thread.sleep(500);
    });
}
```

`assertTimeoutPreemptively(duration, executable)`

Similar to `assertTimeout` but terminates the execution preemptively if the timeout is exceeded.

```
@Test
void
testTimeoutPreemptively()
{

assertTimeoutPreemptively(
Duration.ofSeconds(1), ()
-> {

Thread.sleep(2000); //
This will likely fail
    });
}
```

Assumptions

Assumptions are conditions that must be true for a test to be meaningful. If an assumption fails, the test is aborted.

- `assumeTrue(condition)` - Assumes that the condition is true.
- `assumeFalse(condition)` - Assumes that the condition is false.
- `assumingThat(assumption, executable)` - Executes the executable only if the assumption is met.

```
@Test
void testWithAssumption() {

assumeTrue(System.getProperty("os.name")
.startsWith("Windows"));
    // This test will only run on
Windows
    assertEquals("C:\\",
System.getProperty("user.home"));
}
```

Parameterized Tests (JUnit 5)

Parameterized tests allow you to run the same test multiple times with different input values.

- `@ParameterizedTest` - Marks a method as a parameterized test.
- `@ValueSource` - Provides a simple array of literal values as the source of arguments.
- `@CsvSource` - Allows you to specify multiple arguments as comma-separated values.

```
@ParameterizedTest
@ValueSource(ints = { 2, 4, 6 })
void testNumberIsEven(int number) {
    assertTrue(number % 2 == 0);
}

@ParameterizedTest
@CsvSource({"1,one", "2,two",
"3,three"})
void testNumberName(int number, String
name) {
    assertEquals(name,
numberToName(number));
}
```

Test Fixtures and Suites

Test Fixtures

Test fixtures provide a fixed baseline for running tests. They ensure that the tests are executed in a consistent and repeatable environment.

- Use `@BeforeEach` (JUnit 5) / `@Before` (JUnit 4) to set up the fixture before each test.
- Use `@AfterEach` (JUnit 5) / `@After` (JUnit 4) to tear down the fixture after each test.
- Use `@BeforeAll` (JUnit 5) / `@BeforeClass` (JUnit 4) to set up the fixture once before all tests.
- Use `@AfterAll` (JUnit 5) / `@AfterClass` (JUnit 4) to tear down the fixture once after all tests.

```
class MyTest {

    private MyObject obj;

    @BeforeEach
    void setUp() {
        obj = new MyObject();
        obj.initialize();
    }

    @AfterEach
    void tearDown() {
        obj.cleanup();
        obj = null;
    }

    @Test
    void testSomething() {
        // Test using obj
    }
}
```

Test Suites

Test suites allow you to group multiple test classes into a single execution unit.

- JUnit 4: Use `@RunWith(Suite.class)` and `@Suite.SuiteClasses({TestClass1.class, TestClass2.class})`.
- JUnit 5: Use `@Suite` and `@SelectClasses({TestClass1.class, TestClass2.class})`.

```
@RunWith(Suite.class)
@Suite.SuiteClasses({TestClass1.class, TestClass2.class})
public class MyTestSuite {
    // Empty class, acts as a holder for the suite
}

@Suite
@SelectClasses({TestClass1.class, TestClass2.class})
public class MyTestSuite {}
```

Best Practices

Writing Effective Tests

- **Test one thing at a time:** Each test method should focus on verifying a single aspect of the code.
- **Write clear and descriptive test names:** Test names should clearly indicate what is being tested.
- **Follow the Arrange-Act-Assert pattern:** Arrange the test data, act by invoking the method under test, and assert the expected outcome.
- **Keep tests independent:** Tests should not rely on the state of other tests.
- **Test edge cases and boundary conditions:** Ensure that the code handles unusual or extreme inputs correctly.
- **Write tests that are repeatable and reliable:** Tests should produce the same results every time they are run.
- **Cover all code paths:** Ensure your tests provide sufficient coverage of your code.
- **Use meaningful assertion messages:** Provide clear messages when assertions fail to help identify the root cause.

Mocking

Mocking is a technique used to isolate the code under test from its dependencies. Mock objects simulate the behavior of real objects, allowing you to verify interactions and control the test environment.

- **Mockito:** A popular Java mocking framework that provides a simple and intuitive API.
- **EasyMock:** Another Java mocking framework with similar capabilities.

```java
import org.mockito.Mockito;
import static org.mockito.Mockito.*;
import org.junit.jupiter.api.Test;

class MyServiceTest {

    @Test
    void testDoSomething() {
        MyDependency dependency = mock(MyDependency.class);
        MyService service = new MyService(dependency);

        when(dependency.getValue()).thenReturn(10);

        service.doSomething();

        verify(dependency).getValue();
    }
}
```