



# SWE 4604

## Software Testing and Quality Assurance

### Lab 3

Prepared By Maliha Noushin Raida, Lecturer, CSE  
Islamic University of Technology

# Unit Testing

A **Unit Test** is a method that instantiates a small portion of our application and **verifies its behavior independently from other parts**.

It focuses on the functionality, correctness and accuracy of the unit. Generally carried out by the developers of the project. Can be both black-box and white-box testing.

Steps of Unit Test:

- Initializes a small piece of an application it wants to test (also known as the system under test, or SUT), Example: class, procedure or functions.
- Applies some stimulus to the system under test (usually by calling a method on it)
- Observes the resulting behavior (Passes/Fails)

# Unit Best Practice

Unit tests differ from other kinds of tests in that they must be F.I.R.S.T.

**Fast:** You may have thousands of these and you will run them constantly, so each test should take milliseconds at the most. No network connections! No database connections! Stay in memory. Mock external components.

**Isolatable/Independent:** Tests should never depend on each other. You must be able to run them in any order whatsoever.

**Repeatable:** Tests should run the same in any environment. They must not depend on external factors.

**Self-Describing:** Each test just returns a boolean result for pass or fail. You must never have to sift through output to find out whether it worked or not.

**Timely:** Write them before the code. No one likes to write tests for code that is already written

# Tools for Unit Testing

Language	Some Testing Frameworks
Java	JUnit, TestNG
JavaScript	QUnit, NodeUnit, Mocha, should.js, espresso, Jasmine
Python	unittest2, nose, py.test
Ruby	minitest, test-unit, riot, RSpec

# JUnit

- A unit testing framework for Java.
- The basic idea:  
For a given class Foo, create another class "FooTest" to test it, containing various "test case" methods to run. Each method looks for particular results and passes / fails.
- Unit provides "assert" commands to help us write tests.
  - The idea:  
Put assertion calls in your test methods to check things you expect to be true. If they aren't, the test will fail.

# Testing Via Junit

Test Suite: A java class which contains many test cases or java classes to test together in a single run is called a test suite.

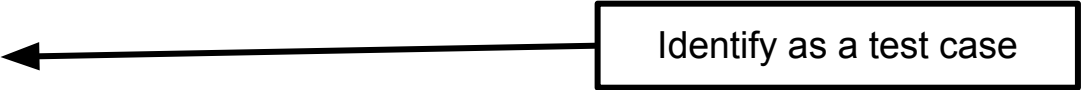
Test Case: A java script written to test a single unit.

## **Features:**

- Assertions
- Test Fixture for sharing common test data
- Test runners for running test case
- Aggregating test cases in test suite

# JUnit test Example

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
class name {
...
    @Test
    public void name()
    {
        // a test case method
        ...
    }
}
```



Identify as a test case

# Junit Annotations

@Test	This annotation is a replacement of org.junit.TestCase which indicates that public void method to which it is attached can be executed as a test Case.
@Before/ @BeforeEach	This annotation is used if you want to execute some statement such as preconditions before each test case.
@BeforeClass/ @BeforeAll	This annotation is used if you want to execute some statements before all the test cases for e.g. test connection must be executed before all the test cases.
@Test(timeout=500)/ @Timeout(5)	This annotation can be used if you want to set some timeout during test execution for e.g. if you are working under some SLA (Service level agreement), and tests need to be completed within some specified time.



# JUnit Annotations

<code>@After/ @AfterEach</code>	This annotation can be used if you want to execute some statements after each Test Case for e.g resetting variables, deleting temporary files ,variables, etc.
<code>@AfterClass/ @AfterAll</code>	This annotation can be used if you want to execute some statements after all test cases for e.g. Releasing resources after executing all test cases.
<code>@Ignore/ @Disabled</code>	This annotation can be used if you want to ignore some statements during test execution for e.g. disabling some test cases during test execution.
<code>@Test(expected= Exception.class)</code>	We expect an exception to be thrown anywhere in the annotated test method

#### JUNIT 4 – LifeCycle Call-back Annotation

@BeforeClass

@Before

@Test ---  
Testcase 1

@After

@Before

@Test ---  
Testcase 2

@After

@AfterClass

#### JUNIT 5 – LifeCycle Call-back Annotation

@BeforeAll

@BeforeEach

@Test ---  
Testcase 1

@AfterEach

@BeforeEach

@Test ---  
Testcase 2

@AfterEach

@AfterAll

# JUnit Assertion

**void assertEquals(boolean expected, boolean actual):** Checks that two primitives/objects are equal.

**void assertTrue(boolean condition):** Checks that a condition is true.

**void assertFalse(boolean condition):** Checks that a condition is false.

**void assertNotNull(Object object):** Checks that an object isn't null.

**void assertNull(Object object):** Checks that an object is null.

**void assertSame(object1, object2):** This method tests if two object references point to the same object.

**void assertNotSame(object1, object2):** This method tests if two object references do not point to the same object.

**void assertEquals(expectedArray, resultArray):** This method will test whether two arrays are equal to each other.

Link: [Assert \(JUnit API\)](#), [Assertions \(JUnit 5.9.2 API\)](#)

# Task 1

## For Individual

Suppose we were asked to write a function to determine whether a value was a prime number. We want it to return true or false if the value is an integer in the range 2..1000000000000 inclusive, and throw an error message for all other inputs. Our test suite would make the following assertions:

- 2 should be prime
- 20 should not be prime
- 47 should be prime
- 933 should not be prime
- 1000000000000 should not be prime
- Should throw on non-integer (e.g., 2.5)
- Should throw on integers less than 2 (e.g., 1)
- Should throw on negatives (e.g., -14)
- Should throw if too large (e.g., 1000000000001)