JUNIT

Presented and Authored by : Asfiya Khan

Cybage®
Delivering Value Scientifically

Agenda

- Why Unit Testing?
- Whose responsible for it?
- Old Way Vs New Way
- Terminologies
- Assert Methods
- Why Mocking
- Mocking Frameworks
- Summary
- TakeAway

Why UNIT TESTING?

- Unit test is a test that examines the behavior of a single unit of work
- In Java, single distinct unit of work is a method
- A unit test confirms that a method accepts the expected range of input and that the method returns the expected value of each test input.

Cybage®
Delivering Value Scientifically

Whose responsible for performing it???

Whose responsible for performing it???

- Tests done on particular functions and modules.
- Require knowledge of internal program and design code.
- Have to be done by **PROGRAMMERS** and NOT by the **TESTERS** .

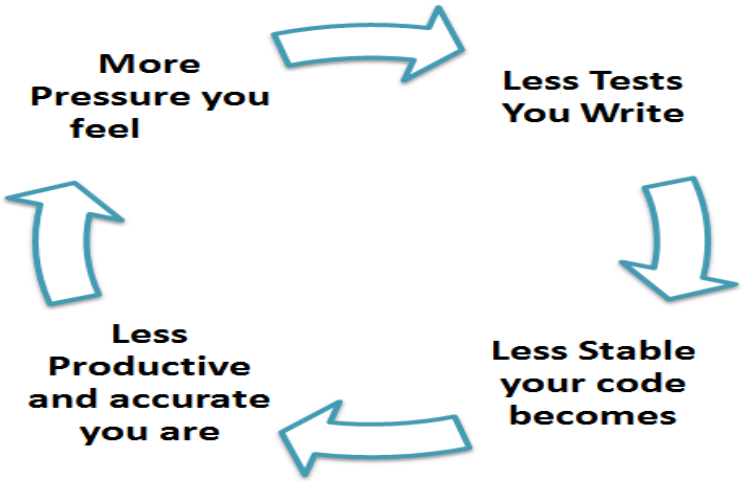
Cybage®
Delivering Value Scientifically

Unit Testing Myth

Myth: It requires time and I am always overscheduled

My code is rock solid! I do not need unit tests.

Myths by their very nature are false assumptions. These assumptions lead to a vicious cycle as follows –



Truth is Unit testing increase the speed of development.

JUNIT as a UNIT Test Framework

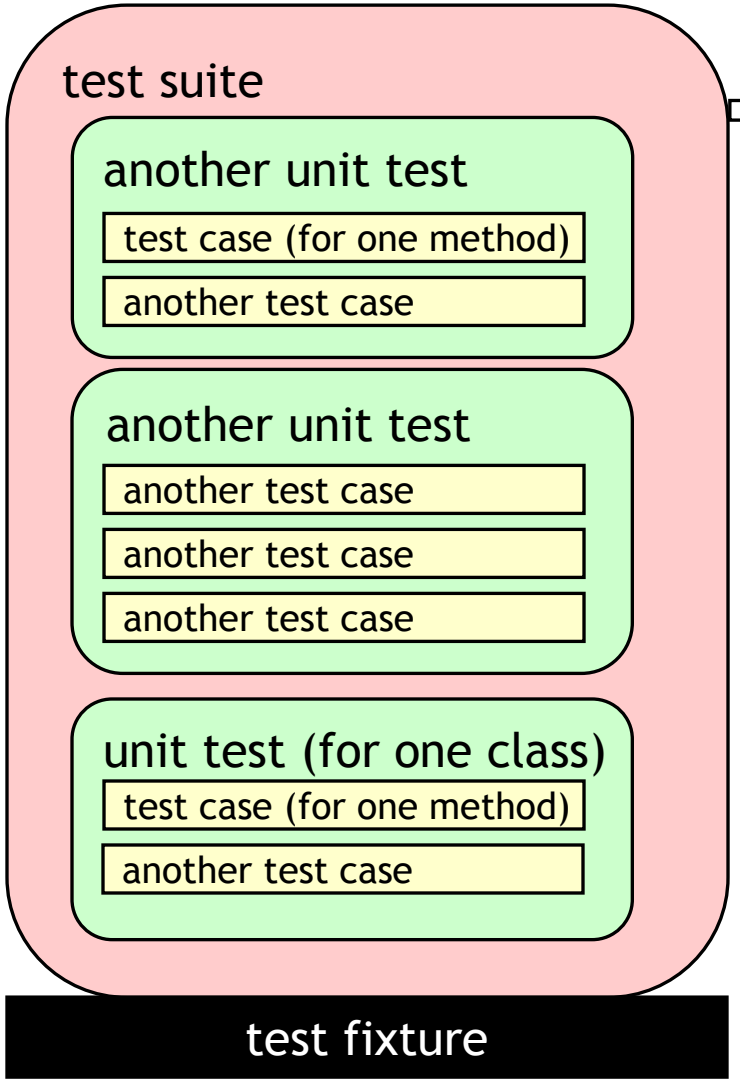
Framework

- is a semi complete application, which provides a reusable, common structure
- Developers incorporate a framework into their application and extend it to meet their needs

Unit testing framework must follow following rules

- Each unit test must run independently
- Errors must be detected and reported by each test
- It must be easy to define which unit tests will run

Unit Testing Framework



- A unit test tests the methods in a single class
- A test case tests (insofar as possible) a single method
  - You can have multiple test cases for a single method
- A test suite combines unit tests
- The test fixture provides software support for all this
- The test runner runs unit tests or an entire test suite
- Integration testing (testing that it all works together) is not well supported by JUnit

8

Terminology

A test fixture sets up the data (both objects and primitives) that are needed to run tests

- Example: If you are testing code that updates an employee record, you need an employee record to test it on

A unit test is a test of a *single* class

A test case tests the response of a single method to a particular set of inputs

A test suite is a collection of test cases

A test runner is software that runs tests and reports results

An integration test is a test of how well classes work together

- JUnit provides some limited support for integration tests

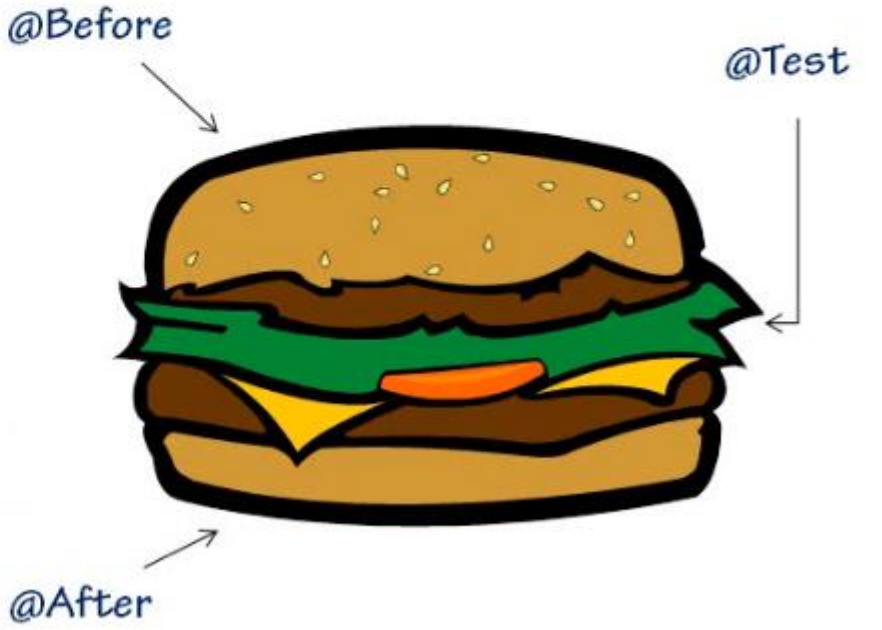
Assert Methods

Method	Description
void assertEquals(boolean expected, boolean actual)	It checks whether two values are equals similar to equals method of Object class
void assertFalse(boolean condition)	functionality is to check that a condition is false.
void assertNotNull(Object object)	"assertNotNull" functionality is to check that an object is not null.
void assertNull(Object object)	"assertNull" functionality is to check that an object is null.
void assertTrue(boolean condition)	"assertTrue" functionality is to check that a condition is true.
void fail()	If you want to throw any assertion error, you have fail() that always results in a fail verdict.
void assertSame([String message]	"assertSame" functionality is to check that the two objects refer to the same object.
void assertNotSame([String message]	"assertNotSame" functionality is to check that the two objects do not refer to the same object.

Junit Annotations

JUnit Annotations (Basic)

- @Test
- @Before
- @After
- @BeforeClass
- @AfterClass
- @Ignore
- @Test(expected = Exception.class)
- @Test(timeout = 100)



Junit Annotations

Annotation	Usage	Use

@Test	@Test
	public void testMethod() {
	// testing code
	}
		Imply that a method is a test case

@BeforeClass	@BeforeClass
	public static void beforeTestClass() {
	//init code
	}
		Provides way to initialize data/variables once before the entire test class is executed

@AfterClass	@AfterClass
	public static void afterTestClass () {
	//final cleanup code
	}
		Provides way to clean up data/variables once after the entire test class is executed

@Before	@Before
	public void beforeEveryMethod(){
	//init before every test method
	}
		Provides way to initialize variables before each method marked with @Test

@After	@After
	public void afterEveryMethod(){
	//cleanup after every test method
	}
		Provides way to cleanup variables after each method marked with @Test

Cybage®
Delivering Value Scientifically

Junit Annotations

Annotation	Usage	Use

@Ignore	@Ignore
	@Test
	public void testMethod() {
	// test method whose result will be ignored
	}
		Allows for some test methods to be ignored .

@RunWith	@RunWith(OtherRunner.class)
	public class TestingClass {
	// other
	}
		Invoke the class it references to run the tests in that class instead of the runner built into JUnit.

@Suite	@Suite.SuiteClasses({TestClass1.class, TestClass2.class … })
	public class TestingSuiteClass {
	}
		A runner which allows you to run multiple Test Classes via a single invocation

@FixMethodOrder

JUnit MethodSorters

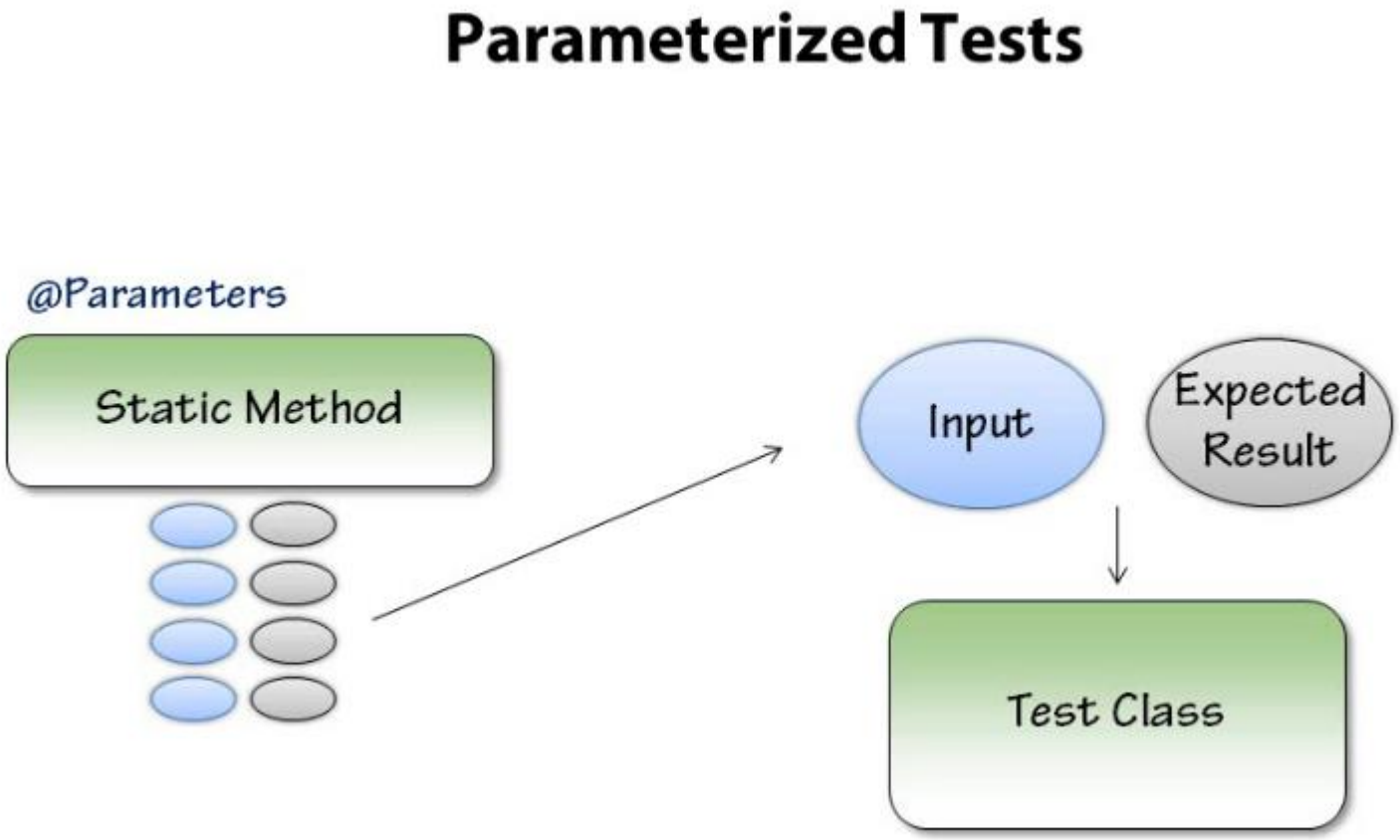
MethodSorters was introduced since JUnit 4.11 release. This class declared three types of execution order, which can be used in your test cases while executing them.

MethodSorters. DEFAULT – Sorts the test methods in a deterministic, but not predictable, order.

MethodSorters.JVM – Leaves the test methods in the order returned by the JVM.

MethodSorters.NAME_ASCENDING – Sorts the test methods by the method name, in lexicographic order, with Method.toString() used as a tiebreaker.

Parameterized Tests



Mocking

- Unit testing relies on mock objects being created to test sections of code that are not yet part of a complete application.
- Mock objects fill in for the missing parts of the program.
- For example, you might have a function that needs variables or objects that are not created yet.
- In unit testing, those will be accounted for in the form of mock objects created solely for the purpose of the unit testing done on that section of code.

Mocking Frameworks

Below are the following mocking frameworks available:

- JUNIT
- Powermock
- JWalk
- Mockito

Mockito

- Sometimes it is not possible to replicate exact production environment.
- At times database is not available.
- Network access is not allowed.

There can be many more such restrictions.

To deal with such limitations, we have to create mock for these unavailable resources.

Mockito Annotations

@Mock is used for mock creation. It makes the test class more readable.

@Spy is used to create a spy instance. We can use it instead spy(Object) method.

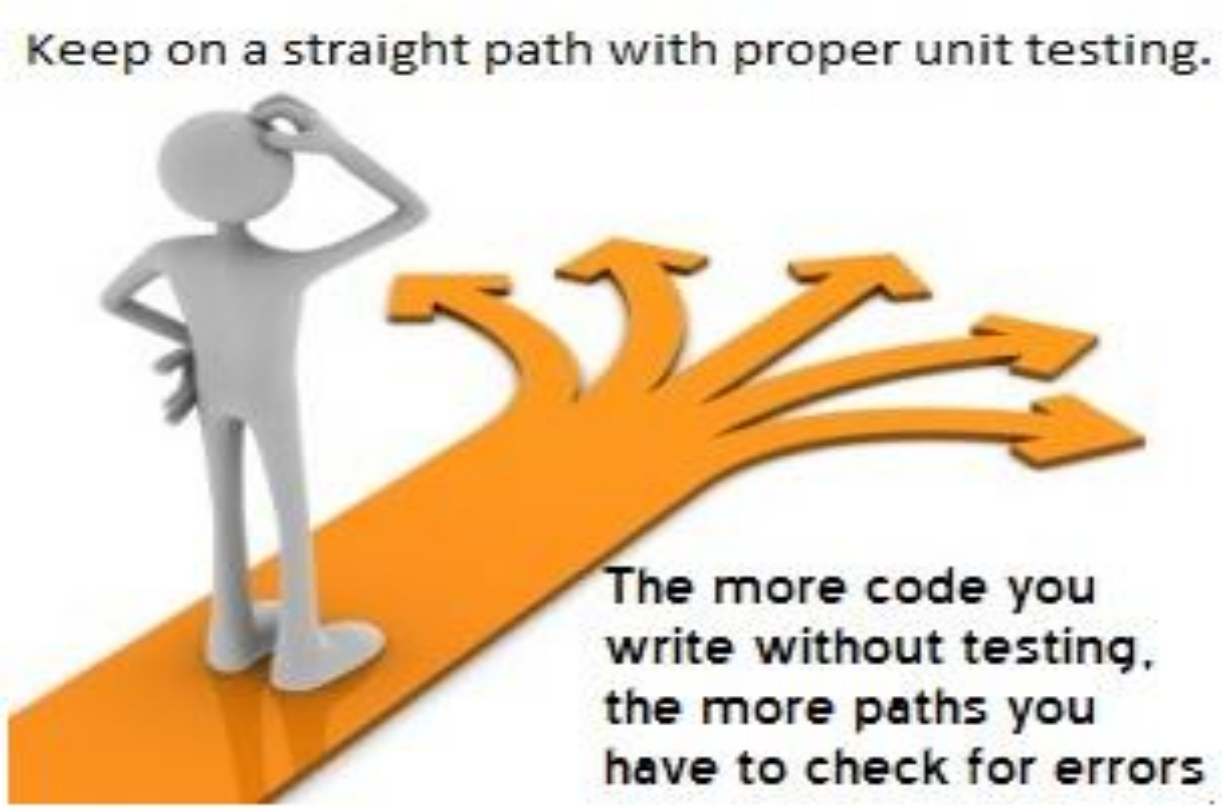
@InjectMocks is used to instantiate the tested object automatically and inject all the *@Mock* or *@Spy* annotated field dependencies into it (if applicable).

@Captor is used to create an argument captor

Summary

- Try to catch every failure situation and to evaluate every execution path so you make your code robust.
- By definition we only test the functionality of the units and should be done in conjunction with other software testing activities.

Take Away!!!



Any Questions?

Thank you!