JUnit5.x

Unit Test Framework for Java

Testing – an important aspect of development

- Unit testing : Testing/verification of smaller Units like functions, classes.(white box testing)
- Integration Testing : test, verify on how the different component/units of application interact and exchange information.(mainly white box testing)
- Acceptance Testing : Application testing from the user perspective to verify that the application matches the requirements.(black box testing)

The unit-test process

The structure of unit-test case

- Create the object under test
- Initialize the properties of the object under test
- Invoke the methods/functions by specifying different combinations of input values
- Verify/Assert the test results
- If the test results are not matching ,modify the object code and again run and verify the test method with results till it matches.
- Clean/destroy the object
- Proceed with more test case methods on the object under test.

Unit-Test Framework

The unit-testing framework automates the unit testing process.

- Allows to write and recognize the test case methods as test cases
- Configure the test methods
- Executes the test cases
- Generate/show the test results
- Display the failures if any for each individual test case.
- Combine multiple test cases in a test suite.
- Generate the test results.

xUnit test-standard

- The framework based on a design by Kent Beck.
- The **xUnit** frameworks allow testing of different elements (units) of software, such as functions and classes.
- Provide an automated solution to execute the same tests many times, and no need to remember what should be the result of each test.

xUnit implementations

- JUnit for java
- TestNG for Java
- CppUnit for C++
- Microsoft Unit Testing Framework for C++
- NUnit for .Net code
- HTMLUnit for html
- DbUnit for database testing

JUnit as xUnit Framework

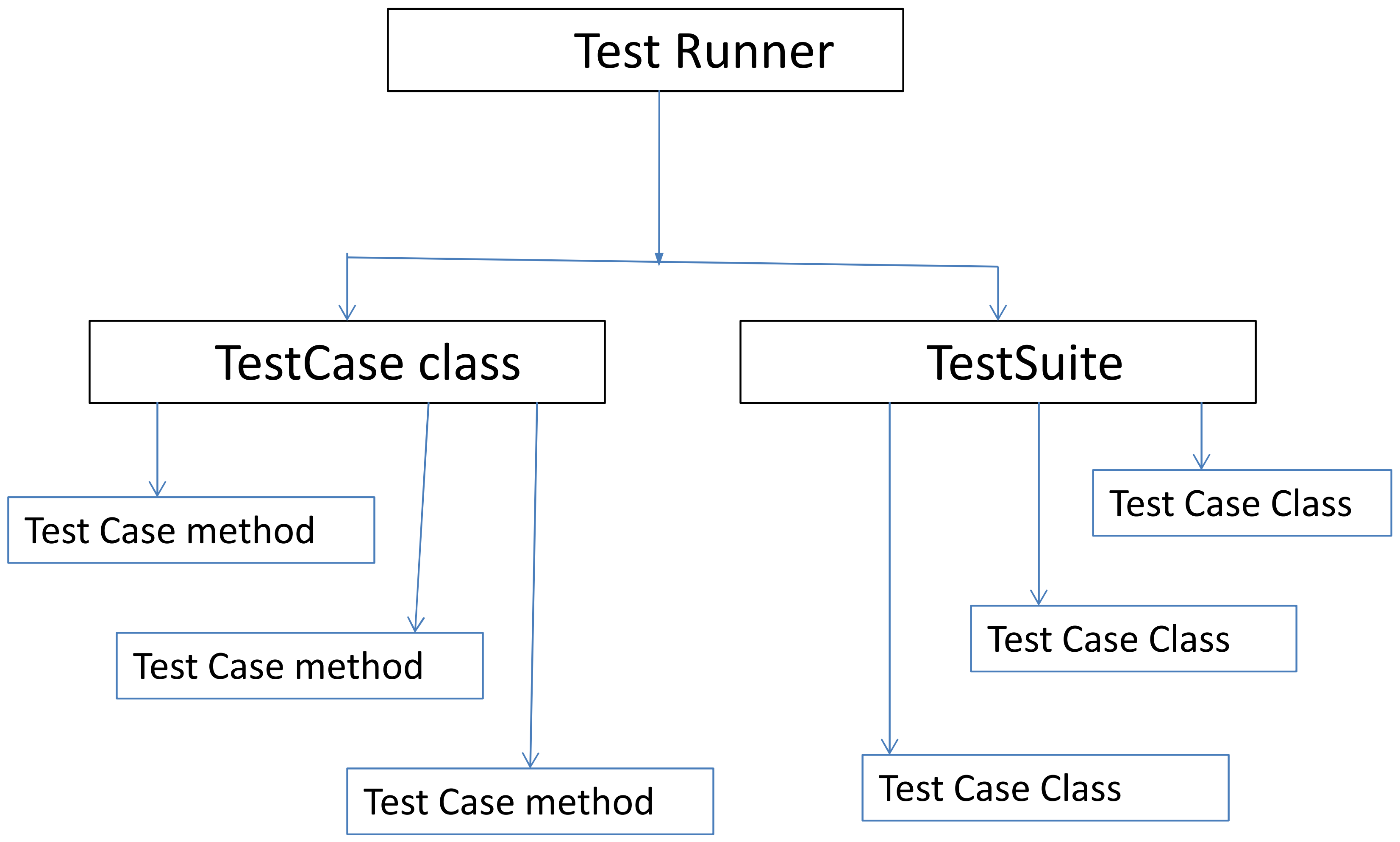
- JUnit originally written by Erich Gamma and Kent Beck.(junit.org)
- JUnit is an open source unit-testing framework for java used to write and run repeatable tests.
- It follows the xUnit architecture for unit testing frameworks.

JUnit Features

The JUnit features include

- Provide test fixture to execute the test cases
- The Test fixtures for sharing common test data
- Supports the assertions for testing/verifying expected results
- Test suites for easily organizing and running tests
- Graphical and textual **Test Runners**

Test implementations



JUnit 3.x

- Every the test case class (Test Fixture) should be extending the base class as TestCase class from JUnit library and must be public.
- Every test method should begin with ‘test’ i.e testData, testUser, testDB etc.
- Every test method should return void and must be public.
- The test case life cycle is controlled by overriding base class methods.
- Built-in assertions to verify/compare results.
- Built-in test runner to execute the test cases.
- A test suite to group multiple test case classes for execution

Test Case in JUnit4.0

- The JUnit4 recognizes any public test method decorated with ‘@Test’ annotation as test case in any class.
- Multiple test methods in single test class can be written.
- Inside each test method the test case result can be verified with assertions supported by JUnit.
- Multiple test case classes can be combined in Test Suite class configured with annotations.

Test case in JUnit 5.x

- Any test method decorated with ‘@Test’ annotation as test case in any class.
- The test method and test class need not be public.
- More improved annotations for parameterized test cases, parallel test cases, ordering of test cases.
- Test class life cycle controlled with methods marked annotations

Test Success or Failure

- The success or failure of a test method is defined by the outcome of a comparison method that compares(asserts) the results.
- If the comparison (assertion) returns false , an **AssertionFailedError** exception is thrown which indicates the failure of the test.
- If no exception occurs in test method it is treated as success.

A set of assertions

- The comparison of test results is done by a set of assert methods that returns the result
- The class **Assert** provides **a** set of static methods to assert the values of different types.
- Messages are displayed only when an assertion fails.

Assert methods

- **assertEquals**(boolean expected, boolean actual)
- **assertEquals**(java.lang.String message, byte expected, byte actual)
- **assertNotNull**(java.lang.Object object)
- **assertSame**(java.lang.Object expected, java.lang.Object actual)
- **assertNotSame**(java.lang.Object expected, java.lang.Object actual)

More Assertions

- Assert methods include:
  - *assertEquals(expected,actual*)
  - *assertTrue*(boolean)
  - *assertFalse* (boolean)
  - *assertNull*(object)
  - *assertNotNull*(object)
  - assertSame(firstObject, secondObject)
  - assertNotSame(firstObject, secondObject)
  - assertArrayEquals(expected, actual)

TestCase class life-cycle

- It defines the test fixture to run multiple test cases.
- Supports test fixture initialization and tear down methods marked with annotations for initialization and destroying of test data for every test method.
- Static methods with annotations at class level define the class level initializations and tear down.
- The new test case class object is created, initialized and tear down for every test case method before and after each test case.
- The test-runner runs a collection of test case methods Or collection of test case classes as Test suite

TestSuite class

- The TestSuite is a container/enclosure of multiple Test case classes and other TestSuite instances.
- The test-suite run is invoked by the framework

Annotated test-case

```
class AccountTest {

  @Test
  public void testAccountDeposit()
  {
    Account ac= new Account(12000);
    ac.deposit(1000);
    Assert.assertEquals(11000, ac.getBalance());
 }
```

If the assert method throws exception it is treated as test failure otherwise success.

Junit 4.x Test Configuration

- To specify the test timeout specify with the annotation as

  @Test(**timeout=1000**)
  public void runLongTest() {....}

- To expect an exception in the test method

  @Test(**expected=ArrayIndexOutOfBoundsException.class**)
  public void testBounds() {
  new ArrayList<Object>().get(1);
  }

- If the method doesn't throw an exception of this type or if it throws a different exception than the one declared, the test is treated as failure.

JUnit 4.x: Ignore/skip the tests

- To temporarily disable a test or a group of tests use ignore annotation.
- Methods annotated with Test and @Ignore will not be executed as tests.
- A class containing test methods can also be annotate d with @Ignore and none of the containing tests will be executed.
- @Ignore @Test public void notYetRun() { ...}
- @Ignore public class TryNotMe {..}

Junit4.X : Test Fixture life

```
public class AccountTest {
  @Test
  public void testAccountDeposit()
  {  …..//some code to test   }
    @BeforeClass
    public static void initClass()
     {  //class level setup code  }
    @AfterClass
  public static void closeTestClass()
    {   //class level close     }
    @Before
   public void initTestSetUp()
    {    //set up for test}
    @After
   public void tearDown()
   { //close test setup }
```

Test Fixtures life in the test case

- Test Setup:
  - Use the `@Before` annotation on a method containing code to run before each test case.
- Test Teardown:
  - Use the `@After` annotation on a method containing code to run after each test case.
  - These methods will run even if exceptions are thrown in the test case or an assertion fails.
- It is allowed to have any number of these annotations.
  - All methods annotated with `@Before` is invoked before each test case, but they may be run in any order.

JUnit4.x: Static Fixtures at class level

- Test Class Setup:
  - Use the **@BeforeClass** annotation on a method containing code to run once before when the test class starts running the tests.
- Test Class Teardown:
  - Use the **@ AfterClass** annotation on a method containing code to run after all the test cases have been finished.

Junit4.x: Test Suite annotation

```
@RunWith(Suite.class)
//Add multiple test implementations here to run
@SuiteClasses({com.data.test.TestBackupFirst.class,com.data.test.UserTest.class,c
    om.data.test.TestBackupSecond.class})

public class RunAllTests {

public static Test suite() {
 TestSuite suite = new TestSuite("Test for com.data.test AllTests");
 return suite;
}
```

Junit4.x : Explicitly fail the test

- Static method : class org.junit.Assert.fail(): Fails a test with no message.
- Static method : class org.junit.Assert.fail(String errorMessage): Fails a test with given message.

JUnit5 architecture

- JUnit 5 is composed of several different modules from three different sub-projects.
- **JUnit Platform**: foundation for defining and launching test cases on the JVM. Supports a console test runner and UI test runner.
- **JUnit Jupiter**: provides a combination of the new programming model and extension model for writing tests and extensions in JUnit 5, provides a TestEngine for running Jupiter based tests on the platform.
- **JUnit Vintage**: provides a TestEngine for running JUnit 3 and JUnit 4 based tests on the platform.

JUnit 5 annotations

- **@Test:** mark the method as test case method
- **@ParameterizedTest:** allows to re-use the same test case for multiple different combinations of test data.
- **@RepeatedTest:** execute the same test case no of times.
- **@DisplayName:** customize the name of the test case to be displayed.
- **@BeforeEach:** common setup for every test case in test class.
- **@AfterEach:** common tear down for every test case in test class
- **@BeforeAll:** common setup at test class level.
- **@AfterAll:** common tear doen at test class level.
- **@Tag:** to declare tags for filtering tests, either at the class or method level.
- **@Disabled:** to disable or skip tests at class or method level.

More JUnit 5 annotations

- **@TestFactory**: denotes that a method is a test factory for dynamic tests.
- **@TestTemplate**: denotes that a method is a template for test cases designed to be invoked multiple times.
- **@TestMethodOrder**: configure the test method execution order for the test cases.
- **@TestInstance**: configure the test instance lifecycle for the annotated test class.
- **@Nested**: configures the test class as a non-static nested test class.

More Junit5 annotations..

- **@Timeout**: to fail a test, test factory, test template, or lifecycle method if its execution time exceeds a given time duration.
- **@ExtendWith**: to register extensions declaratively.
- **@RegisterExtension**: to register extensions programmatically via fields.
- **@TempDir**: to supply a temporary directory via field injection or parameter injection in a lifecycle method or test method.

Parameterized test

- Parameterized tests make it possible to run a test multiple times with different arguments.
- They are declared with @ParameterizedTest annotation.
- In Parameterized tests , you must declare at least one *source* that will provide the arguments for each invocation and then *consume* the arguments in the test method.

Source of Parameters

- The @ValueSource annotation to specify a single array as the source of and type of arguments. It is a single argument per parameterized test.

Value Source Types

- The types of literal values are supported by @ValueSource.
- Short : shorts
- Byte : bytes
- Int : ints
- Long : longs
- Float: floats
- Double : doubles
- Char: chars
- Boolean: booleans
- java.lang.String : strings
- java.lang.Class : classes

Types of sources

- The following types are supported as source parameters :
- java.lang.String
- java.util.List
- java.util.Set,
- java.util.Map
- Primitive arrays (e.g., int[], char[][], etc.),
- Object arrays (e.g.,String[], Integer[][], etc.).
- Subtypes of the supported types are not supported.

Null and Empty to parameterized tests

- To have null and empty values supplied to the parameterized tests, sources of null and empty values for parameterized tests that accept a single argument.
- @**NullSource**: provides a single null argument to the annotated @ParameterizedTest method.
- @NullSource cannot be used for a parameter that has a primitive type.
- @**EmptySource**: provides a single empty argument to the annotated @ParameterizedTest method.
- Combine @NullSource, @EmptySource, and @ValueSource to test a wider range of null, empty, and blank input.
- The composed @NullAndEmptySource annotation simplifies the same.

Enum source

- The @**EnumSource** provides a way to pass the Enum constants as parameters.
- An optional mode attribute that enables fine-grained control over which constants are passed to the test method.
- For example, exclude names from the enum constant pool or specify regular expressions.

Method Source

- The @MethodSource allows to refer to one or more static factory methods of the test class or external classes.
- Each factory method must generate a *stream* of *arguments*, and each set of arguments within the stream will be provided as the physical arguments for individual invocations of the annotated @ParameterizedTest method.

Comma separated values

- The @CsvSource allows to express argument lists as comma-separated values (i.e., String literals).
- @ParameterizedTest
- @CsvSource({ "apple, 1", "banana, 2", "'lemon, lime', 0xF1" })
- The default delimiter is a comma (,), but you can use another character by setting the delimiter attribute.
- Uses single quote ‘ as quote character .

Test data from external files

- @CsvFileSource lets to use CSV files from the classpath.
- Each line from a CSV file results in one invocation of the parameterized test.
- The default delimiter is a comma (,), but you can use another character by setting the delimiter attribute.
- Alternatively, the delimiterString attribute allows to use a String delimiter instead of a single character.
- Any line beginning with a # symbol is interpreted as a comment and is ignored.
- Uses a double quote " as the quote character.
- An *unquoted* empty value is always be converted to a null

Custom ArgumentsProvider

- @ArgumentsSource used to specify a custom, reusable ArgumentsProvider.
- The implementation of ArgumentsProvider must be declared as either a top-level class or as a static nested class.