TDD

Origin

Test Driven Development is a core part of agile process formalized by Kent Beck called eXtreme Programming(XP)

XP originally had the rule to test everything that could possibly break. Now, however, the practice of testing in XP has evolved into Test-Driven Development.

What is TDD?

Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards.

What can be tested?

- Valid Input.
- In-valid Input.
- Exceptions.
- Boundary conditions.
- Everything that should be a possible break.

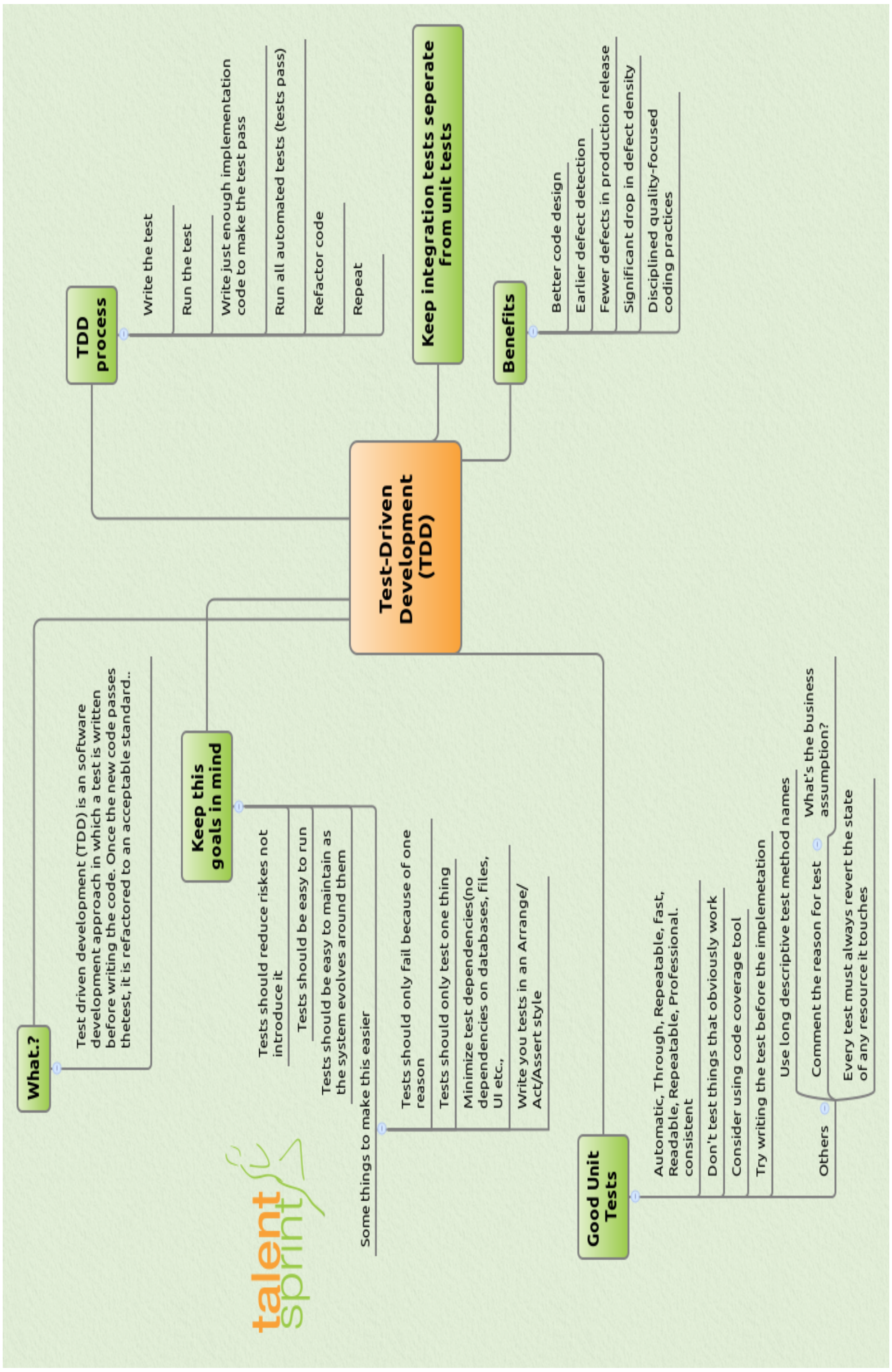
Aspects of TDD

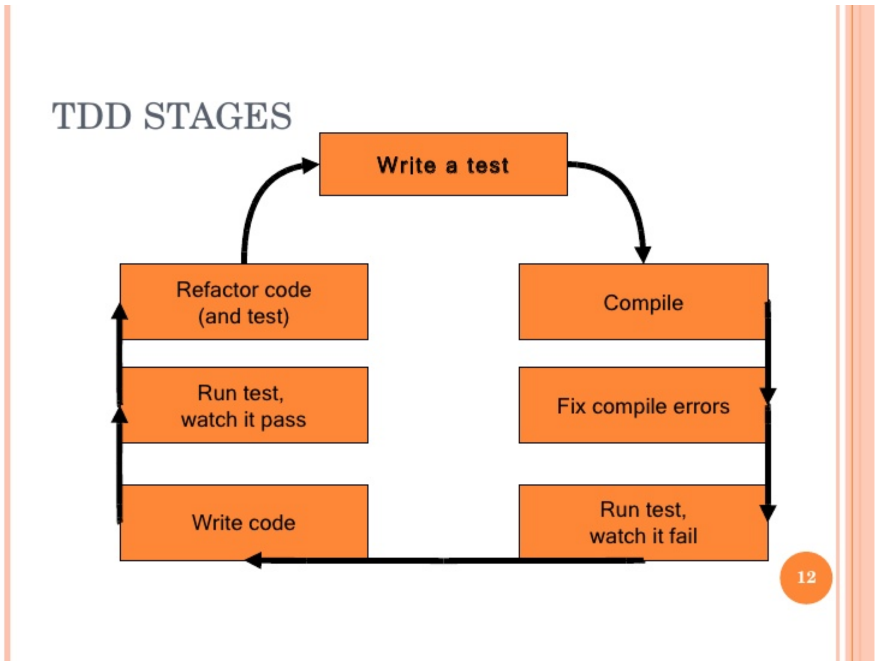
- Features.
  - High level user requirements.
  - User story.
- Customer Tests.
  - Customer Identified user acceptance tests.
- Developer Tests.
  - Tests developed during software construction.

TDD life cycle

The different steps involved in a test driven development cycle are:

- Write a single test.
- Compile it. It shouldn't compile because you've not written the implementation code.
- Implement just enough code to get the test to compile.
- Implement just enough code to get the test to pass.
- Run the test and see it pass.
- Refactor for clarity.
- Repeat.


Test-Driven Development (TDD)
TDD process
Write the test
Run the test
Write just enough implementation code to make the test pass
Run all automated tests (tests pass)
Refactor code
Repeat
Keep integration tests seperate from unit tests
Benefits
Better code design
Earlier defect detection
Fewer defects in production release
Significant drop in defect density
Disciplined quality-focused coding practices
What.?
Test driven development (TDD) is an software development approach in which a test is written before writing the code. Once the new code passes thetest, it is refactored to an acceptable standard..
Keep this goals in mind
Tests should reduce riskes not introduce it
Tests should be easy to run
Tests should be easy to maintain as the system evolves around them
Some things to make this easier
Tests should only fail because of one reason
Tests should only test one thing
Minimize test dependencies(no dependencies on databases, files, UI etc.,
Write you tests in an Arrange/ Act/Assert style
Good Unit Tests
Automatic, Through, Repeatable, Fast, Readable, Repeatable, Professional, consistent
Don't test things that obviously work
Consider using code coverage tool
Try writing the test before the implemetation
Use long descriptive test method names
Others
Comment the reason for test
What's the business assumption?
Every test must always revert the state of any resource it touches
talent sprint



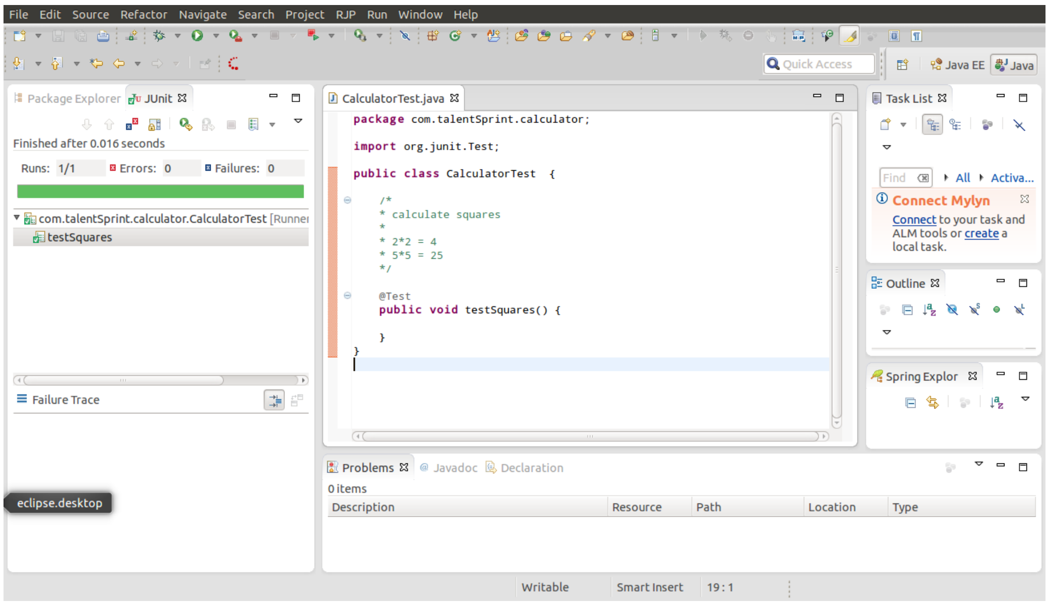
Sample calculator example using TDD

Requirement: Calculator.square() should correctly calculate square value of the integer passed to it.

CalculatorTest class calls the **Calculator.square()** to check its implemention

```
package com.talentSprint.calculator;
import static org.junit.Assert.assertEquals;
import org.junit.Test;
public class CalculatorTest {
    /*
    * calculate squares
    *
    * 2*2 = 4
    * 5*5 = 25
    */
    @Test
    public void testSquares() {
    }
}
```

We have written the test without any implementation, now let's run the code.



Finished after 0.016 seconds

Runs: 1/1 Errors: 0 Failures: 0

com.talentSprint.calculator.CalculatorTest [Runner

testSquares

```
package com.talentSprint.calculator;

import org.junit.Test;

public class CalculatorTest  {

    /*
    * calculate squares
    *
    * 2*2 = 4
    * 5*5 = 25
    */

    @Test
    public void testSquares() {

    }
}
```

Problems: 0 items

Description	Resource	Path	Location	Type

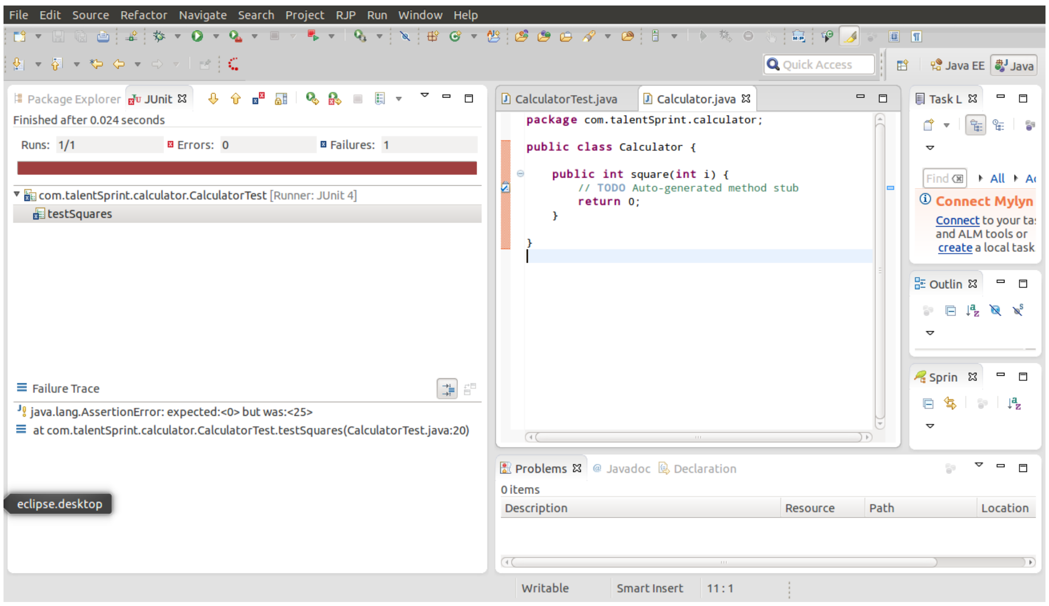
The code successfully executed. Now let's change the implementation for **CalculatorTest** class

```
package com.talentSprint. calculator ;
import static org . junit . Assert . assertEquals ;
import org. junit . Test;

public class CalculatorTest {

    /*
    * calculate squares
    *
    * 2*2 = 4
    * 5*5 = 25
    */

    @Test

    public void testSquares () {
        Calculator calculator = new Calculator();
        assertEquals( calculator .square(5), 25);
    }
}
```

Code for **Calulator.java**, before logic is implemented to calculate square of an integer

```
package com.talentSprint. calculator ;

public class Calculator {
    /*
     * Calculates and returns the square value
     * of integer parameter sent to this method
     *
     */
    public int square(int i) {
        return 0;
    }
}
```

Let's run the unit test again and check the result. The below snapshot shows that the test has failed.



```
Finished after 0.024 seconds
Runs: 1/1    Errors: 0    Failures: 1

com.talentSprint.calculator.CalculatorTest [Runner: JUnit 4]
    testSquares

Failure Trace
java.lang.AssertionError: expected:<0> but was:<25>
at com.talentSprint.calculator.CalculatorTest.testSquares(CalculatorTest.java:20)
```

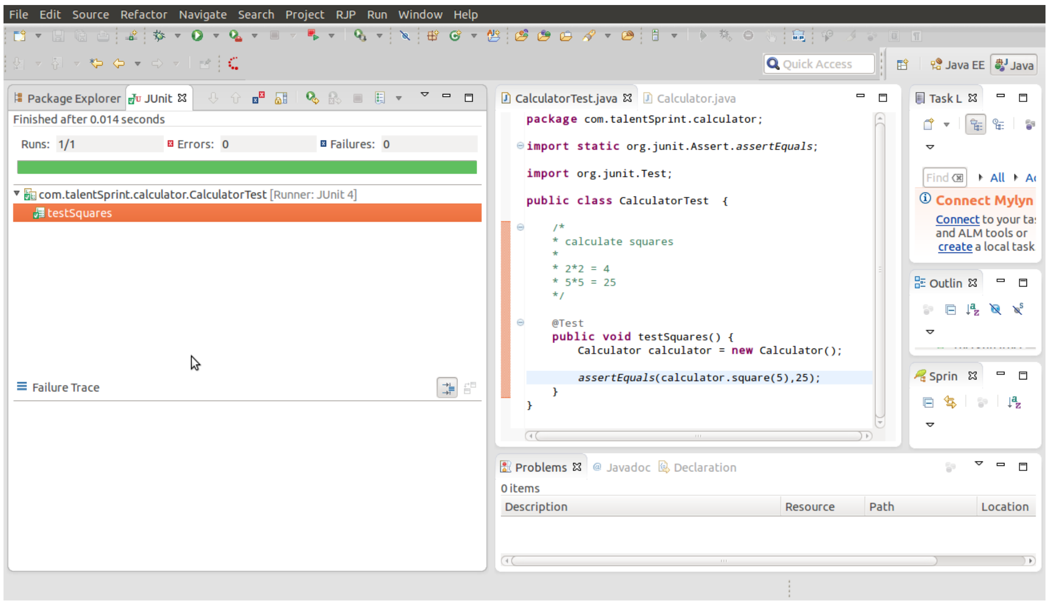
```
package com.talentSprint.calculator;

public class Calculator {

    public int square(int i) {
        // TODO Auto-generated method stub
        return 0;
    }

}
```

```
0 items
Description    Resource    Path    Location

Writable    Smart Insert    11 : 1
```

Now let's add the implementation to **Calculator** class.

```
package com.talentSprint.calculator;

public class Calculator {

        /*
         * Calculates and returns the square value
         * of integer parameter sent to this method
         *
         */
        public int square(int i) {
                // TODO Auto-generated method stub
                int squareVal = i * i;

                return squareVal;
        }
}
```

Let's rerun the unit test to check the result, the test passed because the expected value is 25 and the returned value is 25



Finished after 0.014 seconds

Runs: 1/1 Errors: 0 Failures: 0

com.talentSprint.calculator.CalculatorTest [Runner: JUnit 4]
testSquares

```
package com.talentSprint.calculator;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class CalculatorTest  {

    /*
     * calculate squares
     *
     * 2*2 = 4
     * 5*5 = 25
     */

    @Test
    public void testSquares() {
        Calculator calculator = new Calculator();

        assertEquals(calculator.square(5),25);
    }
}
```

0 items

Merits of TDD

The following are generally the Merits of TDD:

- TDD promotes the development of high quality code.
- TDD shortens the programming feedback loop.
- User requirements more easily understood.
- Reduced software defect rates.
- Less debug time.
- TDD provides concrete evidence that your software works.

Challenges of TDD

The following are generally the drawbacks of Traditional testing:

- Programmers like to code, not to test.
- Test writing is time consuming.
- Lot of test code is created and has to be maintained at a significant cost
- Implement just enough code to get the test to pass.

JUnit

JUnit promotes the idea of "first testing then coding", which emphasis on setting up the test data fora piece of code which can be tested first and then can be implemented. This approach is like "test a little, code a little, test a little, code a little..." which increases programmer productivity and stability of program code that reduces programmer stress and the time spent on debugging.

Features

- JUnit is an open source framework, used for writing and running tests.
- Provides Annotation to identify the test methods.
- JUnit tests allow you to write code faster which increasing quality.

- JUnit is elegantly simple. It is less complex and takes less time.
- JUnit tests can be run automatically and they check their own results and provide immediate feedback. There's no need to manually comb through a report of test results.
- JUnit tests can be organized into test suites containing test cases and even other test suites.
- Junit shows test progress in a bar that is green if test is going fine and it turns red when a test fails.

What is a Unit Test Case ?

A Unit Test Case is a part of code which ensures that the another part of code (method) works as expected. To achieve those desired results quickly, test framework is required. JUnit is perfect unit test framework for java programming language.

A formal written test-case is characterized by a known input and by an expected output, which is worked out before the test is executed. The known input should test a precondition and the expected output should test a post condition. There must be at least two test cases for each requirement: one positive test and one negative test. If a requirement has sub-requirements, each sub-requirement must have at least two test cases as positive and negative.

JUnit Test Framework

JUnit is a Regression Testing Framework used by developers to implement unit testing in Java and accelerate programming speed and increase the quality of code. JUnit Framework can be easily integrated with either of the followings:

- Eclipse
- Ant
- Maven

Features of JUnit Test Framework

JUnit test framework provides following important features

- Fixtures

- Test suites
- Test runners
- JUnit classes

Fixtures

Fixtures is a fixed state of a set of objects used as a baseline for running tests. The purpose of a test fixture is to ensure that there is a well known and fixed environment in which tests are run so that results are repeatable. It includes

- setUp() method which runs before every test invocation.
- tearDown() method which runs after every test method.

Example:

```
import junit.framework.*;

public class JavaTest extends TestCase {
    protected int value1, value2;
    // assigning the values
    protected void setUp(){
        value1=3;
        value2=3;
    }
    // test method to add two values
    public void testAdd(){
        double result= value1 + value2;
        assertTrue(result == 6);
    }
}
```

Test suite

Test suite means bundle a few unit test cases and run it together.

Example:

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

//JUnit Suite Test
@RunWith(Suite.class)
@Suite.SuiteClasses({
TestJunit1.class ,TestJunit2.class
})
public class JunitTestSuite {
}
```

```
import org.junit.Test;
import org.junit.Ignore;
import static org.junit.Assert.assertEquals;

public class TestJunit1 {
    String message = ''Robert'';
    MessageUtil messageUtil = new MessageUtil(message);

    @Test
    public void testPrintMessage() {
        System.out.println('' Inside testPrintMessage()'');
        assertEquals(message, messageUtil.printMessage());
    }
}
```

```
import org.junit.Test;
import org.junit.Ignore;
import static org.junit.Assert.assertEquals;

public class TestJunit2 {
    String message = ''Robert'';
    MessageUtil messageUtil = new MessageUtil(message);

    @Test
    public void testSalutationMessage() {
        System.out.println('' Inside testSalutationMessage()'');
    }
}
```

Test runner

Test runner is used for executing the test cases. Here is an example which assumes TestJunit test class already exists.

```java
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {

    public static void main(String[] args) {
        Result result = JUnitCore.runClasses(TestJunit.class);

        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }

        System.out.println(result.wasSuccessful());
    }
}
```

JUnit classes

JUnit classes are important classes which is used in writing and testing JUnits. Some of the important classes are

- **Assert** which contain a set of assert methods.
- **TestCase** which contain a test case defines the fixture to run multiple tests.
- **TestResult** which contain methods to collect the results of executing a test case.
- **TestSuite** A TestSuite is a Composite of Tests.

Annotation

Annotations are like meta-tags that you can add to you code and apply them to methods or in class. These annotation in JUnit gives us information about test methods, which methods are going to run before and after test methods, which methods run before and after all the methods, which methods or class will be ignore during execution. List of annotations and their meaning in JUnit

- **@Test** - The Test annotation tells JUnit that the public void method to which it is attached can be run as a test case.
- **@Before** - Annotating a public void method with @Before causes that method to be run before each Test method.
- **@After** - Annotating a public void method with @After causes that method to be run after the Test method.
- **@BeforeClass** - Annotating a public static void method with @BeforeClass causes it to be run once before any of the test methods in the class.
- **@AfterClass** - This will perform the method after all tests have finished. This can be used to perform clean-up activities.
- **@Ignore** - The Ignore annotation is used to ignore the test and that test will not be executed.

JUnit Sample example

Now let's go through a step by step process to get a kick start in JUnit using a basic example.

Create a Class **MessageUtil.java**

```
public class MessageUtil {
    private String message;
    //Constructor
    //@param message to be printed

    public MessageUtil(String message){
        this.message = message;
    }

    // prints the message
    public String printMessage(){
        System.out.println(message);
        return message;
    }
}
```

Create Test Case Class

- Create a java test class say TestJunit.java.
- Add a test method testPrintMessage() to your test class.
- Add an Annotaion @Test to method testPrintMessage().
- Implement the test condition and check the condition using assertEquals API of Junit.

Create a java class file name **TestJunit.java**

```java
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class TestJunit {
    String message = "Hello World";
    MessageUtil messageUtil = new MessageUtil(message);

    @Test
    public void testPrintMessage() {
        assertEquals(message, messageUtil.printMessage());
    }
}
```

Create Test Runner Class

- Create a TestRunner java class.
- Use runClasses method of JUnitCore class of JUnit to run test case of above created test class.
- Get the result of test cases run in Result Object.
- Get failure(s) using getFailures() methods of Result object.
- Get Success result using wasSuccessful() methods of Result object.

Create a java class file name TestRunner.java

```java
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {

    public static void main(String[] args) {
        Result result = JUnitCore.runClasses(TestJunit.class);

        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }

        System.out.println(result.wasSuccessful());
    }
}
```

Compile the MessageUtil, Test case and Test Runner classes using javac Now run the Test Runner which will run test case defined in provided Test Case class.

Verify the output. It should return

```
Hello World
true
```

Now update TestJunit in so that test fails. Change the message string.

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class TestJunit {
    String message = "Hello World";
    MessageUtil messageUtil = new MessageUtil(message);

    @Test
    public void testPrintMessage() {
        message = "New Word";
        assertEquals(message,messageUtil.printMessage());
    }
}
```

Now run the Test Runner which will run test case defined in provided Test Case class. Verify the output

```
Hello World
testPrintMessage(TestJunit): expected: <[New Wor]d> but was: <[Hello Worl]d>
false
```

JUnit Exception Test

Junit provides a handy option of Timeout. If a test case takes more time than specified numberof milliseconds then Junit will automatically mark it as failed. The timeout parameter is used alongwith @Test annotation. Now let's see @Test(expected) in action.

Create a Class

- Create a java class to be tested say **MessageUtil.java.**
- Add a error condition inside printMessage() method.
- Add expected exception ArithmeticException to testPrintMessage() test case.

```
/*
* This class prints the given message on console.
*/
public class MessageUtil {

    private String message;
    //Constructor
    //@param message to be printed
    public MessageUtil(String message){
        this.message = message;
    }

    // prints the message
    public void printMessage(){
        System.out.println(message);
        int a = 0;
        int b = 1 / a;
    }

   // add "Hi!" to the message
   public String salutationMessage(){
        message = "Hi!" + message;
        System.out.println(message);
        return message;
   }
}
```

Create Test Case Class

- Create a java test class say **TestJunit.java.**

```
import org.junit.Test;
import org.junit.Ignore;
import static org.junit.Assert.assertEquals;

public class TestJunit {
    String message = "Robert";
    MessageUtil messageUtil = new MessageUtil(message);

    @Test(expected = ArithmeticException.class)
    public void testPrintMessage() {
        System.out.println("Inside testPrintMessage()");
        messageUtil.printMessage();
    }

    @Test
    public void testSalutationMessage() {
        System.out.println("Inside testSalutationMessage()");
        message = "Hi!" + "Robert";
        assertEquals(message,messageUtil.salutationMessage());
    }
}
```

Create Test Runner Class

- Create a java class file name **TestRunner.java.**

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {

    public static void main(String[] args) {
        Result result = JUnitCore.runClasses(TestJunit.class);

        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }

        System.out. println ( result . wasSuccessful ( ) ) ;
    }
}
```

Compile the MessageUtil, Test case and Test Runner classes using `javac`
Now run the Test Runner which will run test cases defined in provided Test Case class.
Verify the output. testPrintMessage() test case will be passed.

```
Inside  testPrintMessage()
Robert
Inside  testSalutationMessage()
Hi!Robert
true
```

Now replace expected exception as IndexOutOfBoundsException in MessageUtil.testPrintMessage() test case and rerun the unit test and verify the output

```
Inside  testPrintMessage()
Robert
Inside  testSalutationMessage()
Hi!Robert
testPrintMessage(TestJunit): Unexpected exception, expected<java.lang.
    IndexOutOfBoundsException> but was<java.lang.ArithmeticException>
false\emph{}
```

JUnit Parameterized Test

JUnit 4 has introduced a new feature Parameterized tests. Parameterized tests allow developer to run the same test over and over again using different values. There are five steps,that you need to follow to create Parameterized tests

- Annotate test class with @RunWith(Parameterized.class).
- Create a public static method annotated with @Parameters that returns a Collection of Objects (as Array) as test data set.
- Create a public constructor that takes in what is equivalent to one "row" of test data.
- Create an instance variable for each "column" of test data.

- Create your tests case(s) using the instance variables as the source of the test data. The test case will be invoked once per each row of data.

Create a Class

- Create a java class to be tested say **PrimeNumberChecker.java.**

Create Parameterized Test Case Class

- Create a java test class say **PrimeNumberCheckerTest.java.**

```java
import java.util.Arrays;
import java.util.Collection;
import org.junit.Test;
import org.junit.Before;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import org.junit.runner.RunWith;
import static org.junit.Assert.assertEquals;

@RunWith(Parameterized.class)
public class PrimeNumberCheckerTest {
    private Integer inputNumber;
    private Boolean expectedResult;
    private PrimeNumberChecker primeNumberChecker;

    @Before
    public void initialize () {
        primeNumberChecker = new PrimeNumberChecker();
    }

    // Each parameter should be placed as an argument here
    // Every time runner triggers, it will pass the arguments
    // from parameters we defined in primeNumbers() method
    public PrimeNumberCheckerTest(Integer inputNumber,Boolean expectedResult) {
        this.inputNumber = inputNumber;
        this.expectedResult = expectedResult;
    }

    @Parameterized.Parameters
    public static Collection primeNumbers() {
        return Arrays.asList(new Object[][] {
        { 2, true },
        { 6, false },
        { 19, true },
        { 22, false },
        { 23, true }
        });
}

// This test will run 4 times since we have 5 parameters defined
    @Test
    public void testPrimeNumberChecker() {
        System.out.println("Parameterized Number is : " + inputNumber);
        assertEquals(expectedResult,
        primeNumberChecker.validate(inputNumber));
    }
}
```

Create Test Runner Class

- Create a java class file name **TestRunner.java**

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {

    public static void main(String[] args) {
        Result result = JUnitCore.runClasses(PrimeNumberCheckerTest.class);
        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }
        System.out.println(result.wasSuccessful());
    }
}
```

Compile the PrimeNumberChecker, PrimeNumberCheckerTest and Test Runner classes using javac

Now run the Test Runner which will run test cases defined in provided Test Case class.

Verify the output.

```
Parameterized Number is : 2
Parameterized Number is : 6
Parameterized Number is : 19
Parameterized Number is : 22
Parameterized Number is : 23
true
```