JUnit (4.X)

Coding (or) Implementation of module/task is done by developer. After Coding is done it will be submitted to Testing Team (QA=Quality Analysis Team). QA Team may do different types of testing. But before submitting code to QA, if DEV performs Individual tasks/modules tested then Error(BUG) rate will be reduced in Application.

Programmer cannot test complete project he can test only his module or task that is called as Unit Testing. Here Unit Indicates a class, method, one layer, one module etc..

Programmer can do testing in two ways

I. **Manual Approach**: This approach may not given accurate result. On testing every time it is a time consuming approach.
II. **Programming Approach**: It will give more accurate result. One time if we write code for Unit Testing, 2nd time onwards time saving approach.

Checking module, by entering input manually, writing sysouts in program and observing, de-bug for complete step by step check manually comes under Manual Testing.

JUnit is a Framework (given by apache) to test one module/task(Unit),using programming approach. It provides accurate result in testing. This JUnit always provides Test -- PASS or FAIL only.

JUnit Supports two types of Programs for Unit Testing those are:

1. **Test Case (one module/part check)** : It is a class used to write test methods which confirms the code working functionality. It returns test PASS/FAIL. On running Test case, result will be shown on JUnit Console.
2. **Test Suite (One or multiple Test cases)** : It is also a class, It is a collection of multiple Test Cases together to run all test cases at a time.

To Write JUnit Test case we need to know two important concepts. Those are Annotations in JUnit and Assert (C) API which has all test methods (static), those are also called as assert methods.

Annotations in JUnit:

a. **@Test** : It must be applied over test methods in Test Case. It will be executed by JUnit. It returns Test Pass/FAIL
b. **@Test(timeout=200)** : To avoid long time testing, or dead locks in Testing timeout option is used. After given time is over test will be considered as FAIL.
c. **@Before** : To provide basic initialization or to pre-process any logic like object creations, data setting, connections openings, Files loading etc will be done in this method. It will be executed once for every test method in Test Case (class).
d. **@After** : After Test method is executed, to execute post-process logic of a test method this annotated method is used. It will be executed once for every test method in Test Case (class).
e. **@BeforeClass** : To initialize any static data or a logic which will be executed only once will be written in this annotated method in Test Case. It will be executed only once before all test begun in Test Case. It must be applied to a static method.
f. **@AfterClass** : It will be executed after all test methods are executed in a Test Case. It is used to execute static post-process logic. That means one time post process logic of test methods. This annotated method must be static.
g. **@RunWith**(__.class) : These are used to add extra capabilities to JUnit Test classes
h. **@SuitClasses**({ .class,..}) : This annotation is used to create Test Suite.

Creating a JUnit Test case :- One test case is used to test one module or task mainly. Naming Rule is mainly followed in development bit it is optional. class name must look like "Test**<Module name>**".

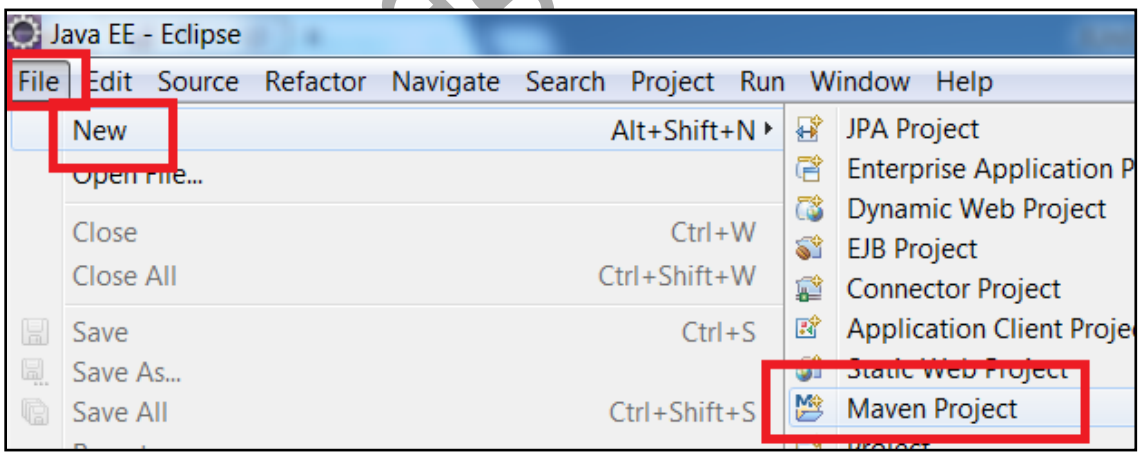
ex: for Employee module Test case is :TestEmployee. Like other examples are TestAdmin, TestLocation.

Steps to create JUnit Example Test case with eclipse screens:

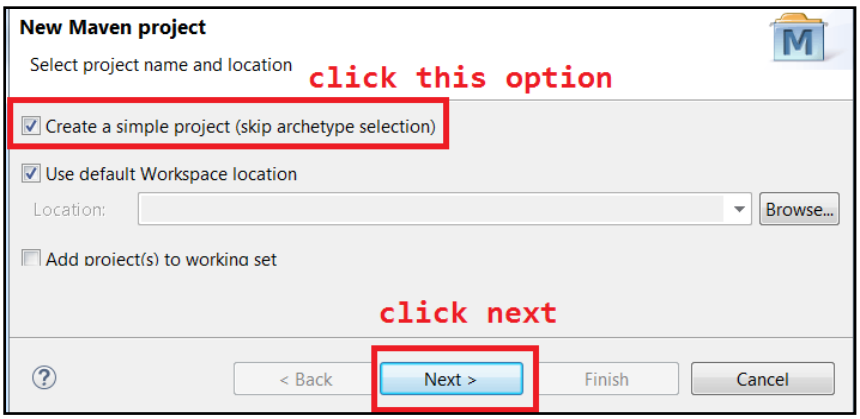
In this Example we are using a Maven project here. Maven supports in-built design of JUnit Test Programming.

a. Create one maven project & set Updated JDK/JRE.
b. Right click on "**src/test/java**" folder.
c. Choose new option and go to choose "other .."
d. Search with "JUnit" word
e. Then select "JUnit Test Case"
f. Click on next
g. Enter Name and package
h. Click on finish button.

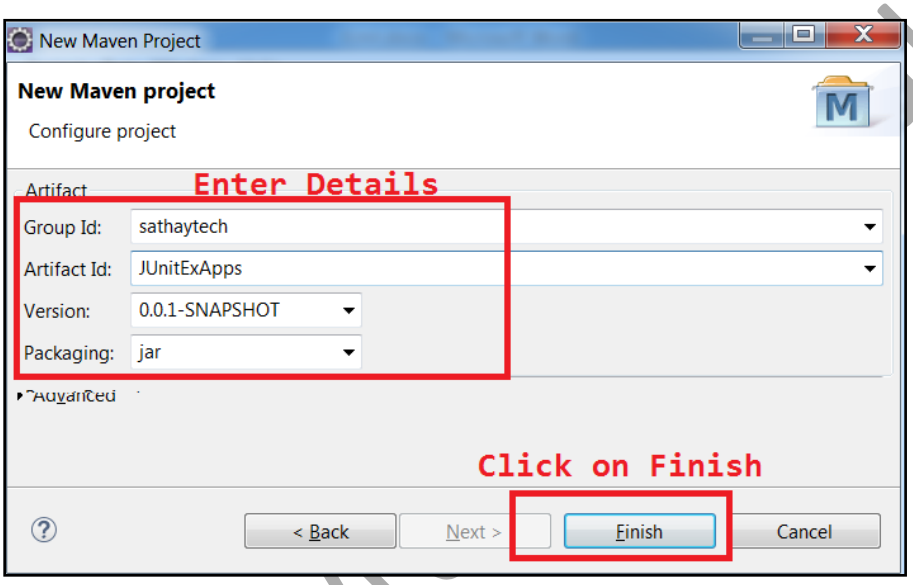
Screen : 1



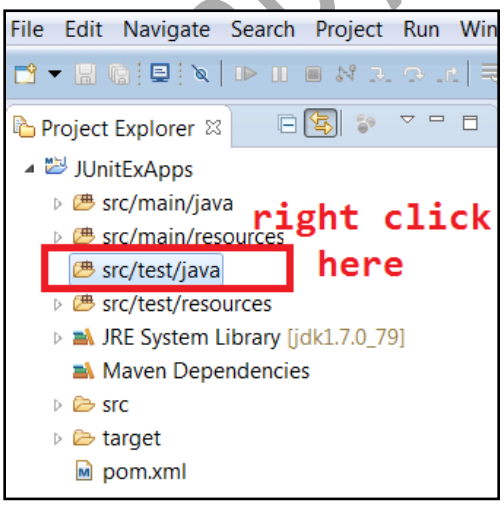
Screen : 2



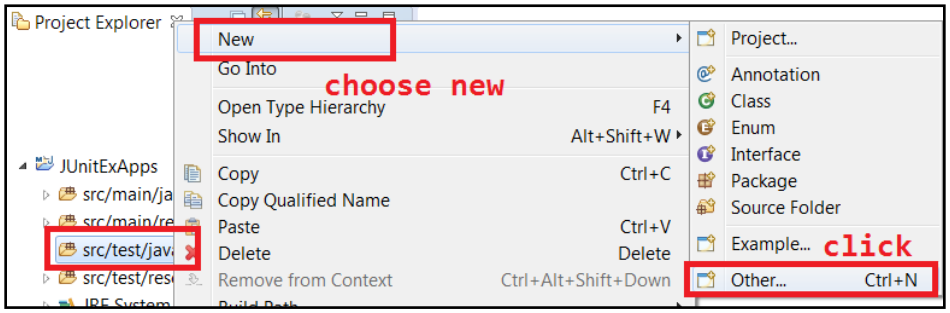
Screen : 3



Screen : 4

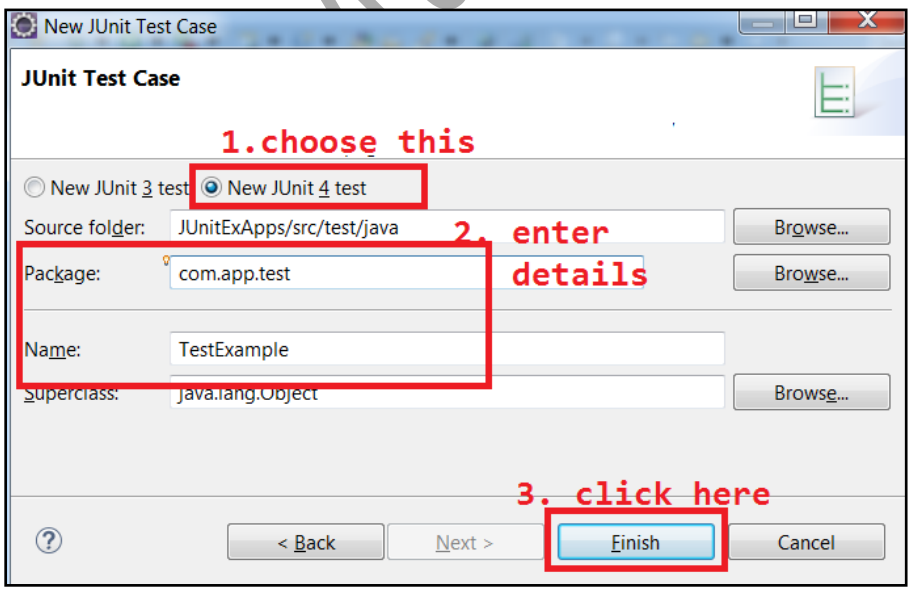


Screen : 5

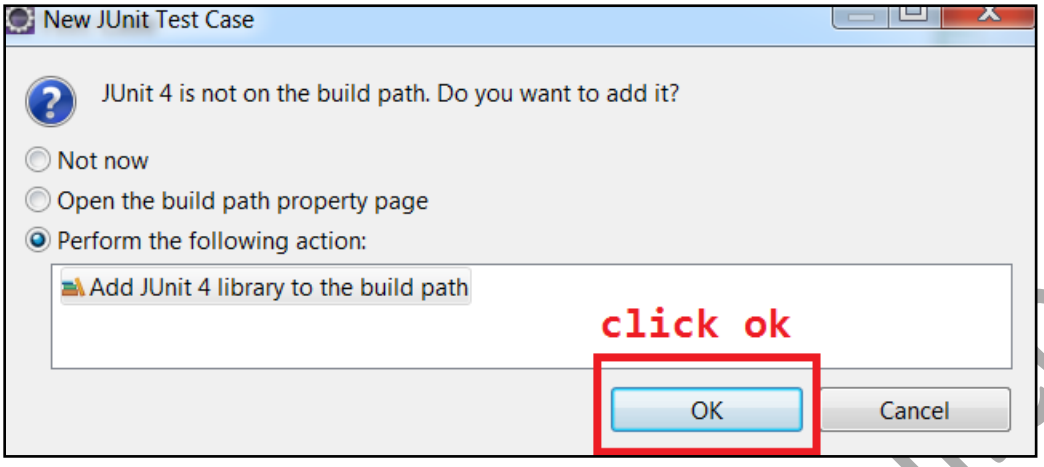


Screen : 6

Screen : 7



Screen : 8



Note:

1. Test case is a class. It contains test methods.
2. Every Test method should be public and void type. It must be annotated with @Test(org.junit package). If no annotation is provided over method then it is called as simple method.
3. We can write supportive methods for test methods like Before, After etc...
4. @Before is used to design one method which will be executed before every test method.
5. @After is used to write one method which executes after every test method.
6. To execute any logic only one time before or after all tests then use @BeforeClass and @AfterClass over methods but those methods must be static type.

Example : Test Case:-

```
package com.app.test;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class TestExample {
```

```
	@BeforeClass
	public static void onetimeBefore(){
	 System.out.println("Before All..");
	}
	@Before
	public void preWork(){
	 System.out.println("before..");
	}
	@Test
	public void testA() {
	 System.out.println("Test-1");
	}
	@Test
	public void testB() {
	 System.out.println("Test-2");
	}
	@Test
	public void testC() {
	 System.out.println("Test-3");
	}
	@After
	public void postWork(){
	 System.out.println("After..");
	}
	@AfterClass
	public static void ontimeAfter(){
	 System.out.println("After All..");
	}
}
```


static import in java: To use one class in another class, if both are in different packages, then we must write import statement. import syntax:

import package.className;

If we use import statement , it will import all members (static and non-static) Sometimes we need only static data then if we use normal import it will load all members and waste then memory hence app performance will be reduced. To

load only static members (static variables and static methods) use static import concept.

static import syntax:

```
import static  pack.class.*;
--or--
import static  pack.class.member;
```

Here member means variable or method.

Example:

```
package com.app;
public class A{
  int id;
  String name;
  static int pid =3;
  void  m1() { ... }
  static void m2(){ ... }
}
```

From above class if we want load only static propeties into JVM then use " import static com.app.A.* "

```
package com.one;
import static com.app.A.*;
public class B{
 void show() {
  sysout(pid);
  m2(); //method call
 }
}
```


Assert (org.junit) JUnit API

To validate any application Logic we use Assert API methods, which are also called as UnitTesting methods.

All these methods are static and void methods. These throws Assertion Error in fail These methods will tell "Is Test PASS or FAIL?" only.

After executing logic it will return some output(actual result) it can compared with expected result. If valid then PASS else FAIL, in this way JUnit Testing will be done

Every Assert method is overloaded to define user friendly Error Messages. All assert methods are given as,

- assertEquals("exp", "original");
- assertEquals("MSG","exp", "org");
- assertNotEquals("exp", "org");
- assertNotEquals("MSG","exp", "org");
- assertTrue(5>(8+9)/5)
- (boolean condition)
- assertTrue("Seems False..",5>(8+9)/5);
- assertFalse(5>(8+9)/5);
- assertFalse("Might be True",5>(8+9)/5);
- fail();fail("Falied method test");
- assertSame(ob1, ob2);
- assertSame("not same..",ob1, ob2);
- assertNotSame(ob1, ob2);
- assertNotSame("same..",ob1, ob2);

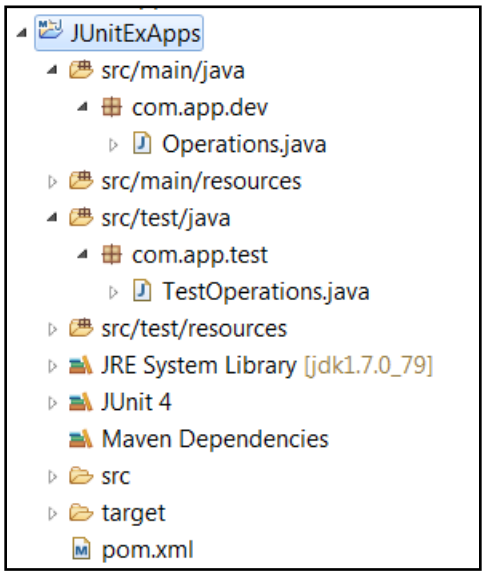
Here assert=Expected [(or) Expected Result Is]. assertSame checks hashCodes, not data. assertEquals checks data.

Example#1:

Requirement : Develop one class with method that performs sum operation.

Test : Test Above method using JUnitTest case with some random data.

Folder Structure:



Code : Requirement

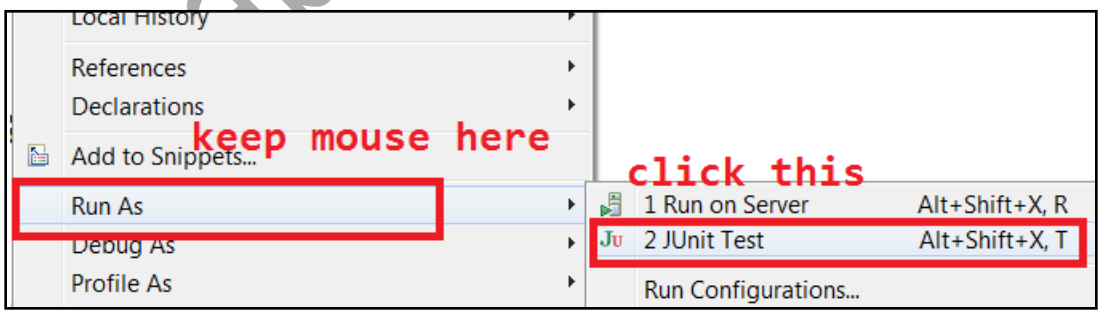
```
package com.app.dev;

public class Operations {

	public int doSum(int x,int y){
		return x+y;
	}
}
```

Create one JUnit Test case to verify that above code written properly or not.

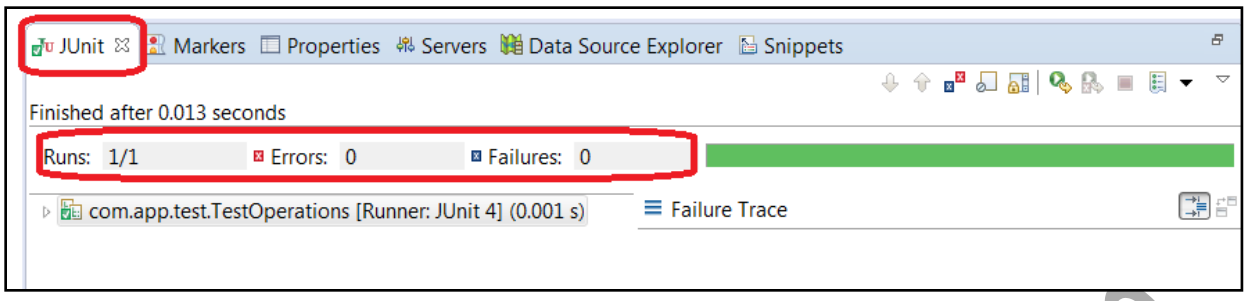
```
package com.app.test;

import static org.junit.Assert.assertEquals;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import com.app.dev.Operations;
```

```
public class TestOperations {
	Operations opr=null;
	int val1,val2,exp;

	@Before
	public void preSetup(){
		opr=new Operations();
		val1=val2=1;
		exp=2;
	}

	@Test
	public void test() {
		int res=opr.doSum(val1, val2);
		assertEquals(
				"Dosum is not working",exp,res);
	}

	@After
	public void postTest(){
		opr=null; val1=val2=exp=0;
	}
}
```


Run As JUnit Test: Right click on Editor then



* Result will be printed on JUnit Console: Observe no.of Test PASS/FAIL.



Method Execution Order:

JUnit By Default takes care of executing methods in its own order (DEFAULT) it can be deterministic, but not predictable(no 100% confirmation) JUnit also supports JVM Method Sorting (MethodSorters.JVM) or we can go for method name sorting as (MethodSorters.NAME_ASCENDING)

Example:-

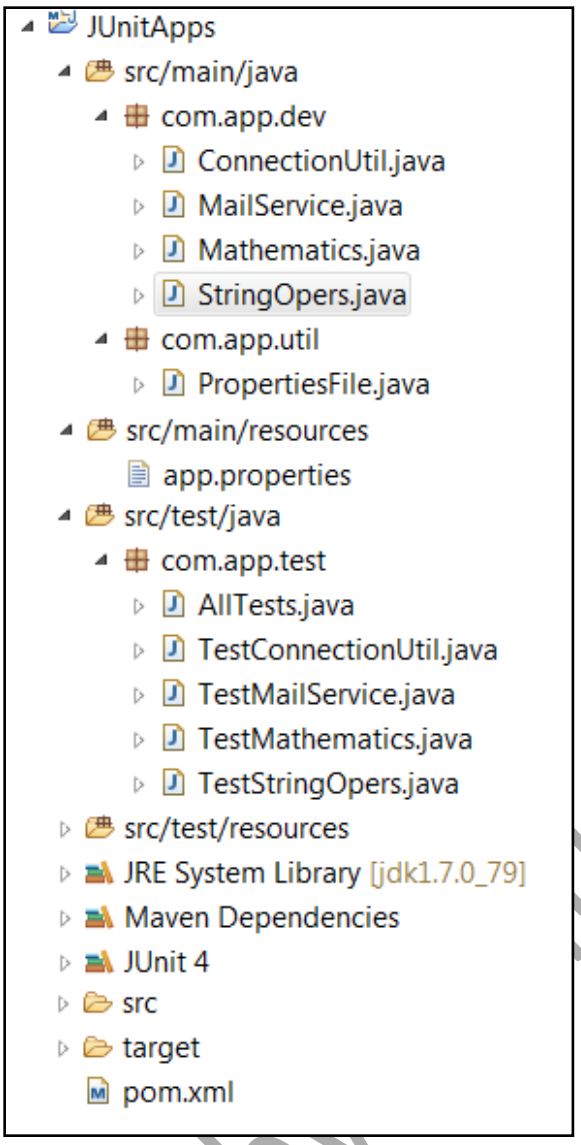
```
@FixMethodOrder(MethodSorters.NAME_ASCENDING)

public class TestSample {

  ....// @Test

}
```

Example JUnit Application with Requirement and JUnit Test cases

Requirements : Write An **Application** and Test Case using JUnit for

1. Singleton Database connection check
2. Sending Email using Java Mail API
3. Find input Number is Armstrong number or not.
4. Input String following ABC Cycle or not

Application code followed by JUnit Test case code:-
Folder structure:



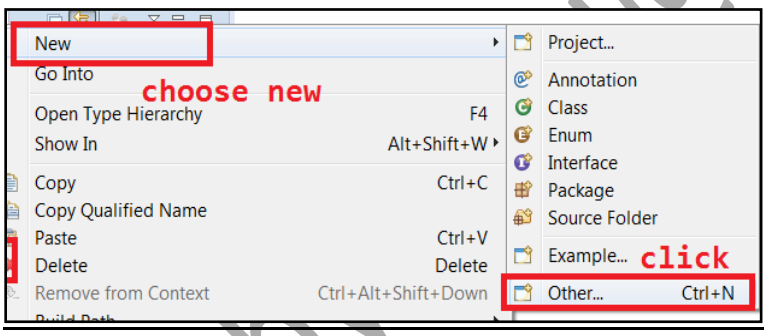
pom.xml code:-

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
     <modelVersion>4.0.0</modelVersion>
     <groupId>sathatech</groupId>
     <artifactId>JUnitApps</artifactId>
     <version>1.0</version>
```

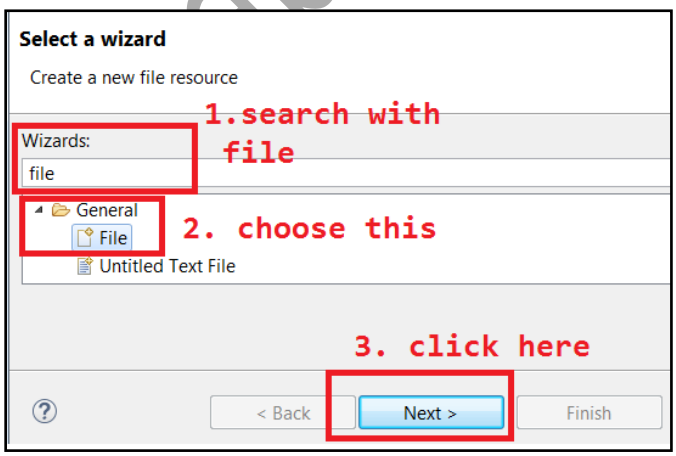
```
    <dependencies>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.6</version>
        </dependency>

        <dependency>
            <groupId>javax.mail</groupId>
            <artifactId>mail</artifactId>
            <version>1.4</version>
        </dependency>

    </dependencies>
</project>
```

Create one Properties for all key=value, for email and Database connection.

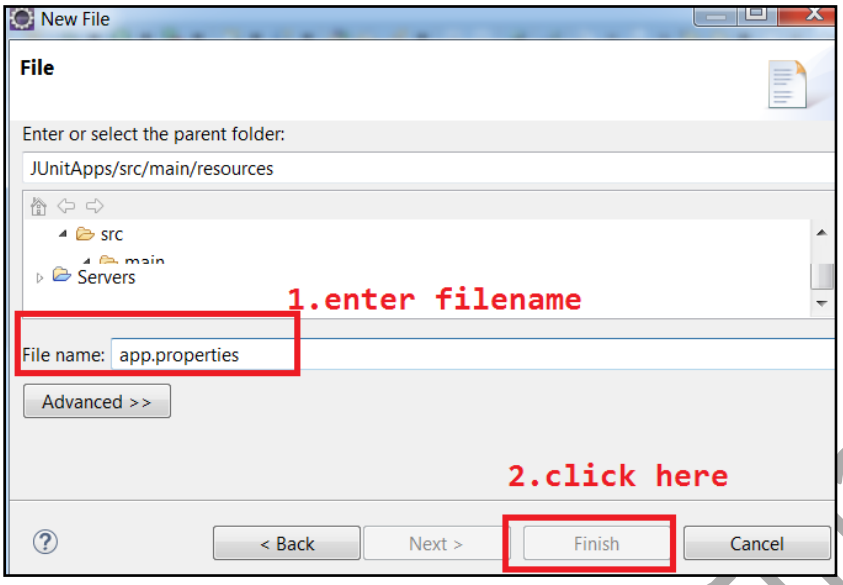
Right click on "src/main/resources" folder , then goto new and select other



Search with "file" option and select file , click next button



Enter File name ex: app.properties and click finish button



app.properties (code)

```
#DB Properties
dc=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/test
un=root
pwd=root

#mail
username=raghusirjava@gmail.com
password=Asdfghjkl1@
AuthKey=mail.smtp.auth
TLSKey=mail.smtp.starttls.enable
HostKey=mail.smtp.host
PortKey=mail.smtp.port
```

1. Code for : Singleton Database connection check

```
package com.app.dev;

import static com.app.util.PropertiesFile.*;

import java.sql.Connection;
```

```
import java.sql.DriverManager;


public class ConnectionUtil {

     private static Connection con=null;
     static{
          try {

     Class.forName(getProperties().getProperty("dc"));

     con=DriverManager.getConnection(getProperties().getPrope
rty("url"),
                         getProperties().getProperty("un"),
getProperties().getProperty("pwd"));
          } catch (Exception e) {
               e.printStackTrace();
          }

     }
     public static Connection getSingeltonConnection(){
          return con;
     }
}
```

JUnitTest Case code:

```
package com.app.test;


import static org.junit.Assert.*;
import java.sql.Connection;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import com.app.dev.ConnectionUtil;
```

```
public class TestConnectionUtil {

	Connection con1,con2;

	@Before
	public void setUp(){

		con1=ConnectionUtil.getSingeltonConnection();
		con2=ConnectionUtil.getSingeltonConnection();
	}

	@Test
	public void test() {
		assertNotNull(con1);
		assertNotNull(con2);
		assertSame("Not a singleton connection",con1,
con2);
	}
	@After
	public void clear(){
		con1=con2=null;
	}
}
```

2. Sending Email using Java Mail API:

```
package com.app.dev;

import static com.app.util.PropertiesFile.*;
import java.util.Properties;

import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
```

```
public class MailService {

     public static boolean sendEmail(String toAddr,String
subject,String text) {
          boolean isMailSent=true;
          final String username =
getProperties().getProperty("username");
          final String password =
getProperties().getProperty("password");

Properties props = new Properties();
props.put(getProperties().getProperty("AuthKey"), "true");
props.put(getProperties().getProperty("TLSKey"), "true");
props.put(getProperties().getProperty("HostKey"),
"smtp.gmail.com");
props.put(getProperties().getProperty("PortKey"), "587");

Session session = Session.getInstance(props,
     new javax.mail.Authenticator() {
     protected PasswordAuthentication
               getPasswordAuthentication() {
     return new PasswordAuthentication(username, password);
                }
          });

try {

 Message message = new MimeMessage(session);
 message.setFrom(new InternetAddress(
                    getProperties().getProperty("username")));
message.setRecipients(Message.RecipientType.TO,InternetAddre
ss.parse(toAddr));
message.setSubject(subject);//subject
message.setText(text);//message

Transport.send(message);
System.out.println("Done");

} catch (MessagingException e) {
```

```
                isMailSent=false;
                e.printStackTrace();
            }
            return isMailSent;
        }
}
```

JUnit Test case Code:-

```
package com.app.test;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import com.app.dev.MailService;

public class TestMailService {

    String toAddr,sub,text;

    @Before
    public void setUp(){
        toAddr="javabyraghu@gmail.com";
        sub="TESTAAA";
        text="HEllo..";
    }
    @Test(timeout=5*1000)
    public void test() {
    boolean flag=MailService.sendEmail(toAddr, sub, text);
        assertTrue(flag);
    }
    @After
    public void clean(){
        toAddr="javabyraghu@gmail.com";
        sub="TEST";
        text="HEllo..";
```

```
	}
}
```

3. Find input Number is Armstrong number or not:

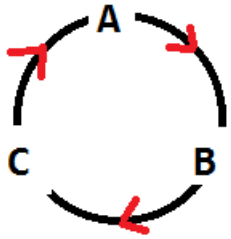
```
package com.app.dev;

public class Mathematics {

	public static boolean isArmStrong(int num){
		int temp=num,c;
		int armNum=0;
		int len=Integer.toString(num).length();
		while(num>0){
			c=num%10;
			armNum= armNum+ (int)Math.pow(c, len);
			num=num/10;
		}
		return temp==armNum;
	}

}
```

JUnit Test Case code:

```
package com.app.test;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import com.app.dev.Mathematics;

public class TestMathematics {
```

```
	int num;
	@Before
	public void preSet(){
		num=548834;
	//http://mathworld.wolfram.com/NarcissisticNumber.html
	}
	@Test
	public void test() {
		boolean flag=Mathematics.isArmStrong(num);
		assertTrue(flag);
	}
	@After
	public void postModify(){
		num=0;
	}

}
```

4. Input String following ABC Cycle or not: String should follow Cycle




```
package com.app.dev;

public class StringOpers {

	public static boolean isFollwingABCCycle(String str){
		char[] arr=str.toCharArray();
		boolean flag=true;
		for (int i=0;i<arr.length-1;i++) {
			if(!(getNext(arr[i]).equals(arr[i+1]))) {
				flag=false;
				break;
			}
```

```
            }

            return flag;
        }
        private static Character getNext(char c){
            if(c=='A') c= 'B';
            else if(c=='B') c= 'C';
            else if(c=='C') c= 'A';
            return c;
        }

}
```

JUnit Test Case Code:-

```
package com.app.test;

import static org.junit.Assert.assertTrue;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import com.app.dev.StringOpers;

public class TestStringOpers {

    String str;
    @Before
    public void setStr(){
        str="CABCA";
    }
    @Test
    public void test() {
        assertTrue(StringOpers.isFollwingABCCycle(str));
    }
    @After
    public void clean(){
```

```
            str=null;
      }

}
```

Test Suite :

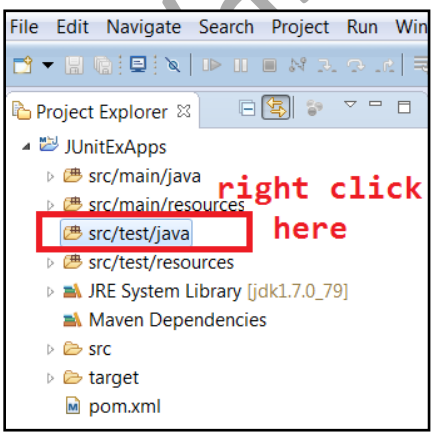
In JUnit, only one Test cases can be run at a time. If we want to run multiple Test cases at a time , then we need to create a JUnit Test suite. Every Suite must contain minimum one Test Case to Run.

Suite Also Follows naming convention which also optional. Test suite also a class but it has no logical method to execute. Test Suite class name must be suffixed with Tests word. Example All**Tests**.

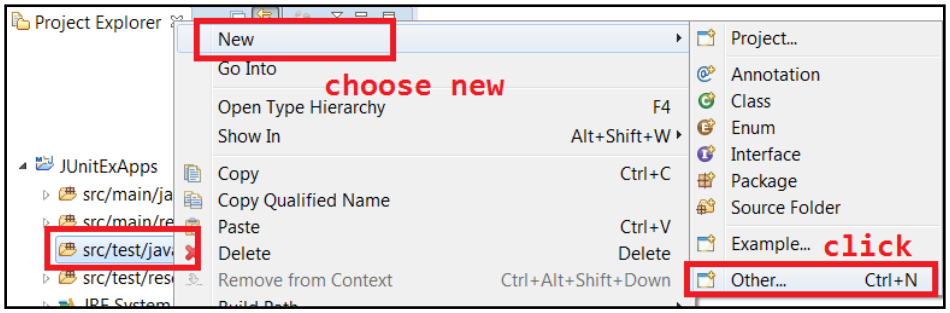
Creating JUnit Test Suite:

a. Right click on "src/test/java"
b. Choose new option
c. Click on other
d. Search using JUnit Test
e. Select JUnit Test Suite
f. Enter package and name
g. Select JUnit Test Cases to be in Test Suite
h. Click on finish

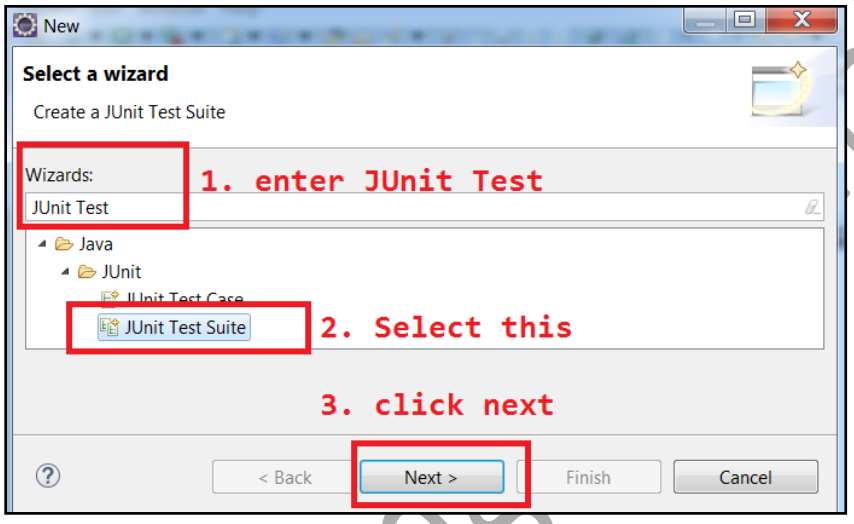
Screen : 1



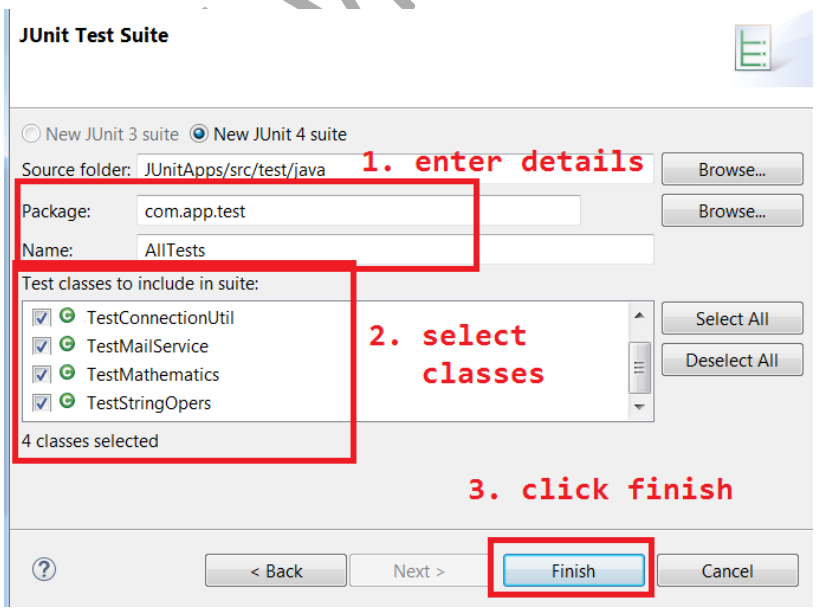
Screen : 2



Screen : 3



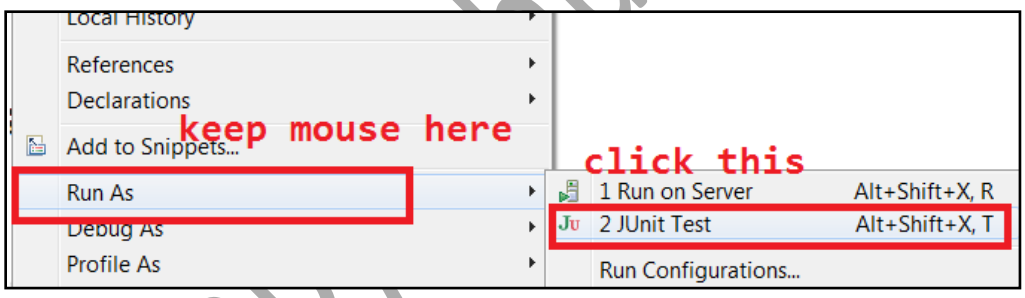
Screen : 4



It can written with two simple annotations. Example shown below.

```
package com.app.test;

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

@RunWith(Suite.class)
@SuiteClasses({ TestConnectionUtil.class,
TestMailService.class, TestMathematics.class,
TestStringOpers.class })
public class AllTests {

}
```

To Run Suite : Right click on code (Editor) , choose Run As, JUnit Test.



output on JUnit Console: