Junit Test

- Unit Testing or Component Testing is a software testing technique in which single component of a software is tested

- The objective here is to pick each unit (code) and verify it.

What is JUnit?

- Junit is a Unit Testing framework for Java
- The framework is used by Java Developers to write and execute tests
- Every time a new code is added, all the test cases have to be re-executed

UNIT TESTING

- Unit Testing refers to the testing of small chunks of codes
- It helps in early identification of defects
- The developers are tend to spend more time on reading the code
- Successful code increases the confidence of the developers

Manual Testing

- If the test cases are tested manually without any automated tool, it is manual testing.
- It is less reliable and time-consuming

Automated Testing

- If the test cases are tested by tool support, it is automated testing.
- It is more reliable and faster

Two types of Unit Testing

JUnit is an open source network used to write and run tests

Junit provides annotations to identify test methods

JUnit provides assertions to test expected results

JUnit Annotations

- JUnit Annotations refer to the syntactic meta-data added to the Java source code for better structure and readability
- The biggest difference between JUnit4 and JUnit3 is the introduction of Junit Annotations

@Test	Tells JUnit which public void method can be run as a test case
@Before	To execute some statement before each test case
@After	To execute some statement after each test case
@Ignores	Used to ignore some statements during test execution

@BeforeClass	Used to execute a statement before all the test cases
@AfterClass	Used to execute a statement after all the test cases
@Test (time out =500)	Used to set some timeout while executing the test
@Test (expected=Illegal ArgumentException.class)	Used to handle some exception during test execution

Sr.No.	Annotations	Description
1	@Test	The *Test* annotation tells JUnit that the public void method can be run as a test case.
2	@Before	Annotating a public void method with @Before causes that method to be run before each *Test* method.
3	@After	If you allocate external resources in a Before method, you need to release them after the test runs. Annotating the method with *@After* causes that method to be run after the Test method.
4	@BeforeClass	Annotating a public static void method with @BeforeClass causes it to be run once before any of the test methods in the class.
5	@AfterClass	This will perform the method after all tests are run. This can be used to perform clean-up activities.
6	@Ignore	The Ignore annotation is used to ignore the test and that test will not be executed.

https://reflectoring.io/junit5/

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

public class TestJunit {

   String message = "Worldline java";

   @Test
   public void testPrintMessage() {
      System.out.println("Inside testPrintMessage()");
      assertEquals(message, "Worldline java");
   }
   public static void main(String[] args) {
      TestJunit result = new TestJunit();
      result.testPrintMessage();
   }
}
```

```
javac -cp junit-4.13.1.jar; . TestJunit.java
java -cp junit-4.13.1.jar;. TestJunit
```

Assert statements

Assert is a method used in determining pass or fail status of a test case. In JUnit, all the assertions are in the Assert class.

Sr.No.	Method	Description
1	**void assertEquals(boolean expected, boolean actual)**	Checks that two primitives/objects are equal.
2	**void assertTrue(boolean condition)**	Checks whether the condition is true.
3	**void assertFalse(boolean condition)**	Checks whether the condition is false.
4	**void assertNotNull(Object object)**	Checks whether an object isn't null.
5	**void assertNull(Object object)**	Checks whether an object is null.
6	**void assertSame(object1, object2)**	The assertSame() method tests if two object references point to the same object.
7	**void assertNotSame(object1, object2)**	The assertNotSame() method tests if two object references do not point to the same object.
8	**void assertArrayEquals(expectedArray, resultArray)**	The assertArrayEquals() method will test whether two arrays are equal to each other.

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

import org.junit.Test;

public class JunitClass {
@Test
public void setup()
{
    String str = "This is my first JUnit program";
    assertEquals("This is my first JUnit program",str);
}
}
```

```java
package io.javabrains;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

class MathUtilsTest {

    @Test
    void testAdd() {
        MathUtils mathUtils = new MathUtils();
        int expected = 1;
        int actual = mathUtils.add(1, 1);
        assertEquals(expected, actual, "The add method should add two numbers");
    }
```

```java
package io.javabrains;

public class MathUtils {

    public int add(int a, int b) {
        return a + b;
    }

    public double computeCircleArea(double radius) {
        return 0;
    }
```

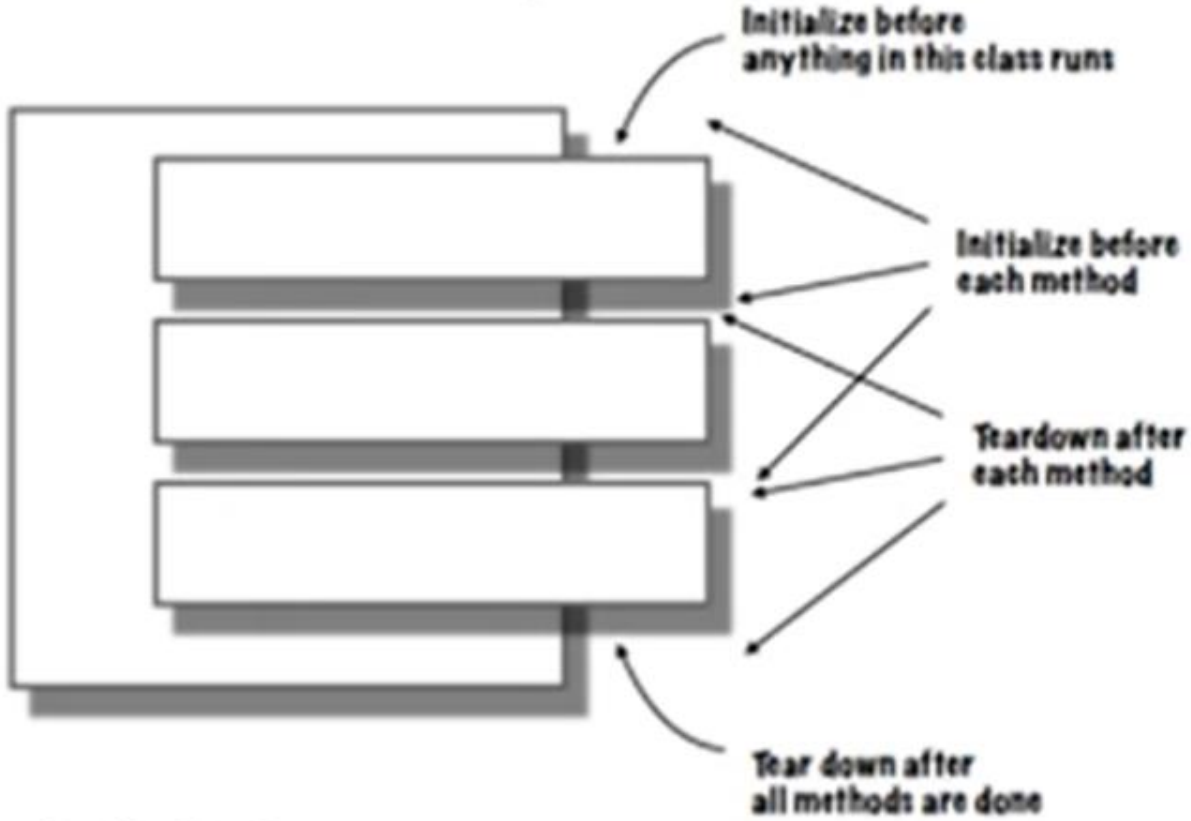
```java
public class Assert {
    @Test
    public void testAssertions()
    {
        String str = new String ("edu");
        String str1 = new String ("edu");
            String str2 = null;
            String str3= "edu";
            String str4 ="edu";
            int val = 5;
            int val1 = 6;
            String[] exceptedArray = {"one", "two", "three"};
            String[] resultArray= {"one", "two", "three"};
            assertEquals(str, str1);
            //check for true
            assertTrue(val<val1);
            //check for false
            assertFalse(val<val1);
            //check for null
            assertNotNull(str);
            //check if it is null
            assertNull(str2)
    }
```

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

public class DemoClass {

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

}
```

```java
@BeforeAll
void beforeAllInit() {
    System.out.print("This needs to run before all");
}

@BeforeEach
void init() {
    mathUtils = new MathUtils();
}

@AfterEach
void cleanup() {
    System.out.print("Cleaning up...");
}
```

```java
@Test
void testDivide() {
    MathUtils mathUtils = new MathUtils();
    assertThrows(NullPointerException.class, () -> mathUtils.divide(1, 0), "Divide by zero should throw");
}
```

Test life cycle



Lifecycle hooks

- @BeforeAll
- @AfterAll
- @BeforeEach
- @AfterEach

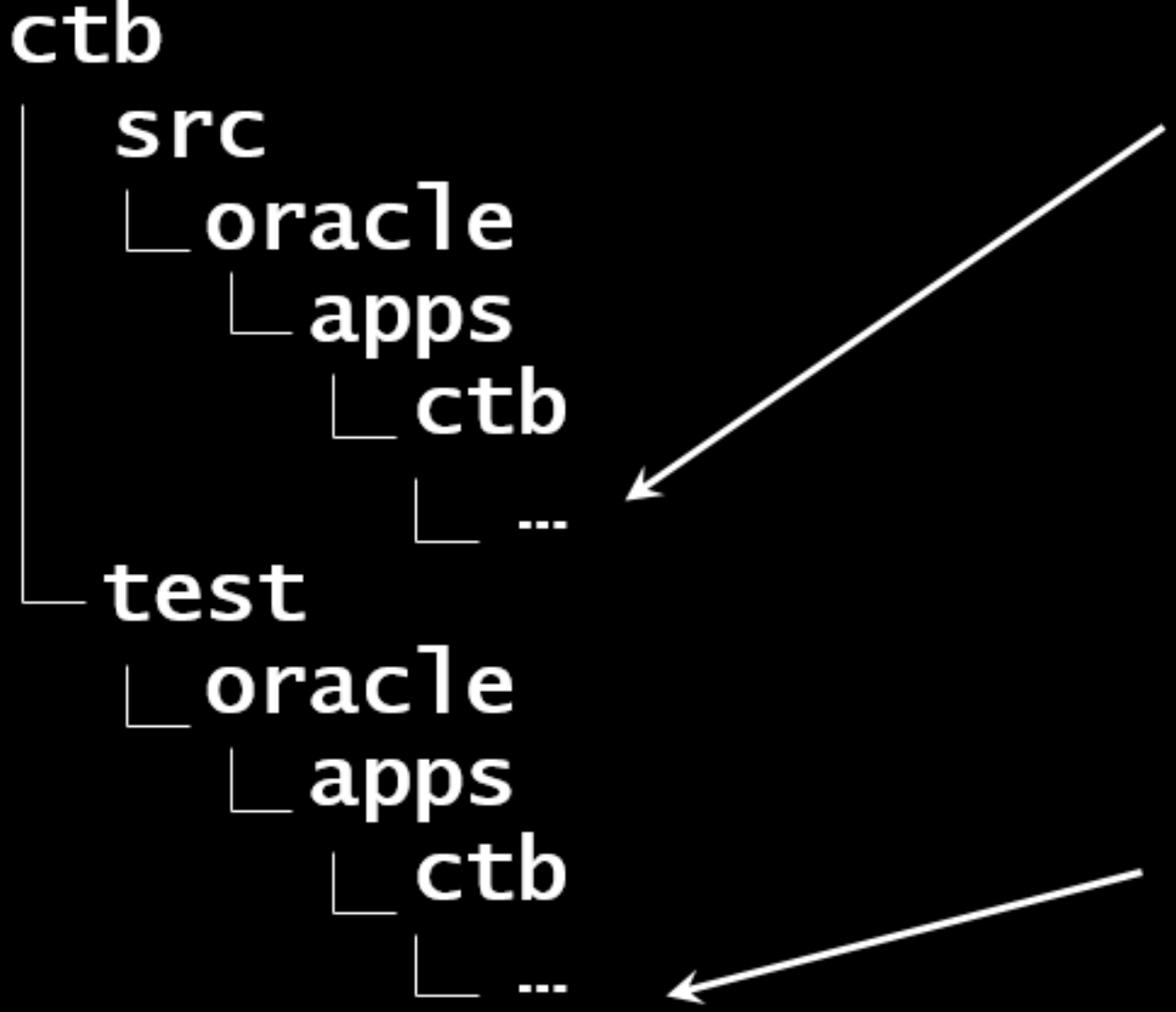
```java
@Test
@DisplayName("multiply method")
void testMultiply() {
    // assertEquals(4, mathUtils.multiply(2, 2), "should return the right product");
    assertAll(
            () -> assertEquals(4, mathUtils.multiply(2, 2)),
            () -> assertEquals(0, mathUtils.multiply(2, 0)),
            () -> assertEquals(-2, mathUtils.multiply(2, -1))
            );
}
```

```java
@Nested
class AddTest {

    @Test
    @DisplayName("Testing add method for +")
    void testAddPositive() {
        assertEquals(2, mathUtils.add(1, 1), "The add method should add two numbers");
    }

    @Test
    @DisplayName("Testing add method for -")
    void testAddNegative() {
        assertEquals(-2, mathUtils.add(-1, -1), "The add method should add two numbers");
    }

}
```

Writing a Unit Test

1. Create a class to hold the unit tests
2. Initialise objects (setUp() method)
3. (State assertions – preconditions)*
4. Call operations on the objects that are being unit tested
5. State assertions/failures expected
6. Clean up (tearDown() method)
7. Execute the unit test

Setting up Unit Tests




```
public class SomeClass
{
   ..

   public void someMethod()
   {
      ..
   }

   ..
}
```

```
public class SomeClassTest
{
   public void testSomeMethod()
   {
      ..
   }
}
```

A first test case

```java
import static org.junit.jupiter.api.Assertions.assertEquals;

import example.util.Calculator;

import org.junit.jupiter.api.Test;

class MyFirstJUnitJupiterTests {

    private final Calculator calculator = new Calculator();

    @Test
    void addition() {
        assertEquals(2, calculator.add(1, 1));
    }

}
```

```java
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import com.cc.airline.ticketing.SeatingPlan;

public class SeatingPlanTest {
	@Before
	public void setUp() throws Exception {
		System.out.println(
			"Starting test of the SeatingPlan default constructor");
	}
	@After
	public void tearDown() throws Exception {
		System.out.println(
			"Test of the SeatingPlan default constructor complete");
	}
	@Test
	public void testSeatingPlan() {
		SeatingPlan sp = new SeatingPlan();
		assertNotNull(sp);
		assertEquals(sp.getSeats().size(), 14);
		assertNotNull(sp.getSeatReserver());

	}
```

```java
public class Student {
    public Student(String studentName, int studentId,
                   String courseName, double gpa)
    {
        super();
        this.studentName = studentName;
        this.studentId = studentId;
        this.courseName = courseName;
        this.gpa = gpa;
    }
    public Student()
    {
        // via setter methods, rest fields are done
    }
    String studentName;
    int studentId;
    String courseName;
    double gpa;
    public String getStudentName() { return studentName; }
    public void setStudentName(String studentName)
    {
        this.studentName = studentName;
    }
    public int getStudentId() { return studentId; }
    public void setStudentId(int studentId)
    {
        this.studentId = studentId;
    }
    public String getCourseName() { return courseName; }
    public void setCourseName(String courseName)
    {
        this.courseName = courseName;
    }
    public double getGpa() { return gpa; }
    public void setGpa(double gpa) { this.gpa = gpa; }
}
```

https://www.geeksforgeeks.org/junit-writing-sample-test-cases-for-studentservice-in-java/

```
assertEquals(true, studentServicesJavaObject.getStudents(

// creating a student object
Student student = new Student();
student.setStudentId(1);
student.setStudentName("Rachel");
student.setCourseName("Java");
student.setGpa(9.2);
studentServicesJavaObject.appendStudent(student, studentL

// After appending the data
assertEquals(true, studentServicesJavaObject.getStudents(
Student monica = new Student("Monica", 2, "Java", 8.5);
studentServicesJavaObject.insertStudent(monica, studentL:

// After inserting the data
assertEquals(true, studentServicesJavaObject.getStudentNa
assertEquals(true, studentServicesJavaObject.getStudents(
Student phoebe = new Student("Phoebe", 3, "Python", 8.5)
studentServicesJavaObject.appendStudent(phoebe, studentL:
assertEquals(true, studentServicesJavaObject.getStudents(
assertEquals(true, studentServicesJavaObject.getStudentNa
assertEquals(true, studentServicesJavaObject.getStudents(
```

What is Mocking?

- Mocking refers to the development of objects which are a mock or clone of real objects.
- In the technique mock objects are used instead of real objects for testing
- Mock objects give a particular output for each particular input.
- Mockito is the most popular framework used for Mocking

- Mockito is a Java based framework used for unit testing of Java applications
- This mocking framework helps in development of testable applications

 A correct set of unit tests should expose a bug in the Circle class. This means that your unit tests must include at least one test that would fail on the provided code, but will pass after you identify and correct the ‘bug’. So, when you discover the bug, correct it and leave your test in place.

Create a JUnit test suite and test runner for the DVD Collection project.

- The test suite should include separate JUnit test classes for the DVD, DVDCollection, and DVDConsoleUI classes.
- Each JUnit test class should test the methods and properties for a single class.
- Each JUnit test should test a single operation.
- Tests should be named to describe the property being tested.
- The test suite should cover all of the major operations for the tested classes.
- All tests should be contained in a single package named testing.
- The DVD project should be contained in a separate package.
- A test runner should be provided to run the test suite

Consider a software program that is artificially seeded with 100 faults. While testing this program, 159 faults are detected, out of which 75 faults are from those artificially seeded faults. Assuming that both real and seeded faults are of same nature and have same distribution, the estimated number of undetected real faults is _____.