SENG 275

SOFTWARE TESTING

DR. NAVNEET KAUR POPLI

DEPT. OF ELECTRICAL AND COMPUTER ENGINEERING

University of Victoria

JUNIT TESTING

University of Victoria

Unit testing frameworks in different languages

- Java: junit

https://junit.org/junit5/docs/current/user-guide/#writing-tests

- Python: unittest

https://docs.pytest.org/en/7.2.x/reference/reference.html#api-reference

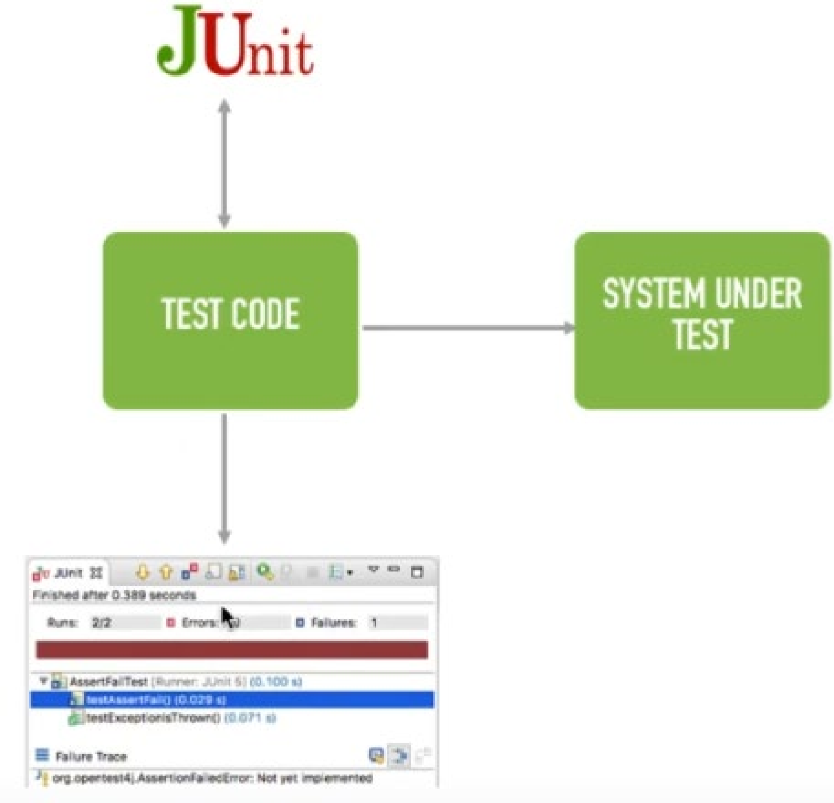
- Javascript: jestjs
- Go: testify

Junit

- JUnit is an open-source Unit Testing Framework for JAVA.
- It is useful for Java Developers to write and run repeatable tests.
- **Erich Gamma and Kent Beck** initially developed it.
- https://www.youtube.com/watch?v=1zaCvLVU70o
- It is an instance of xUnit architecture.
- It is used for Unit Testing of a small chunk of code.

How Junit works?

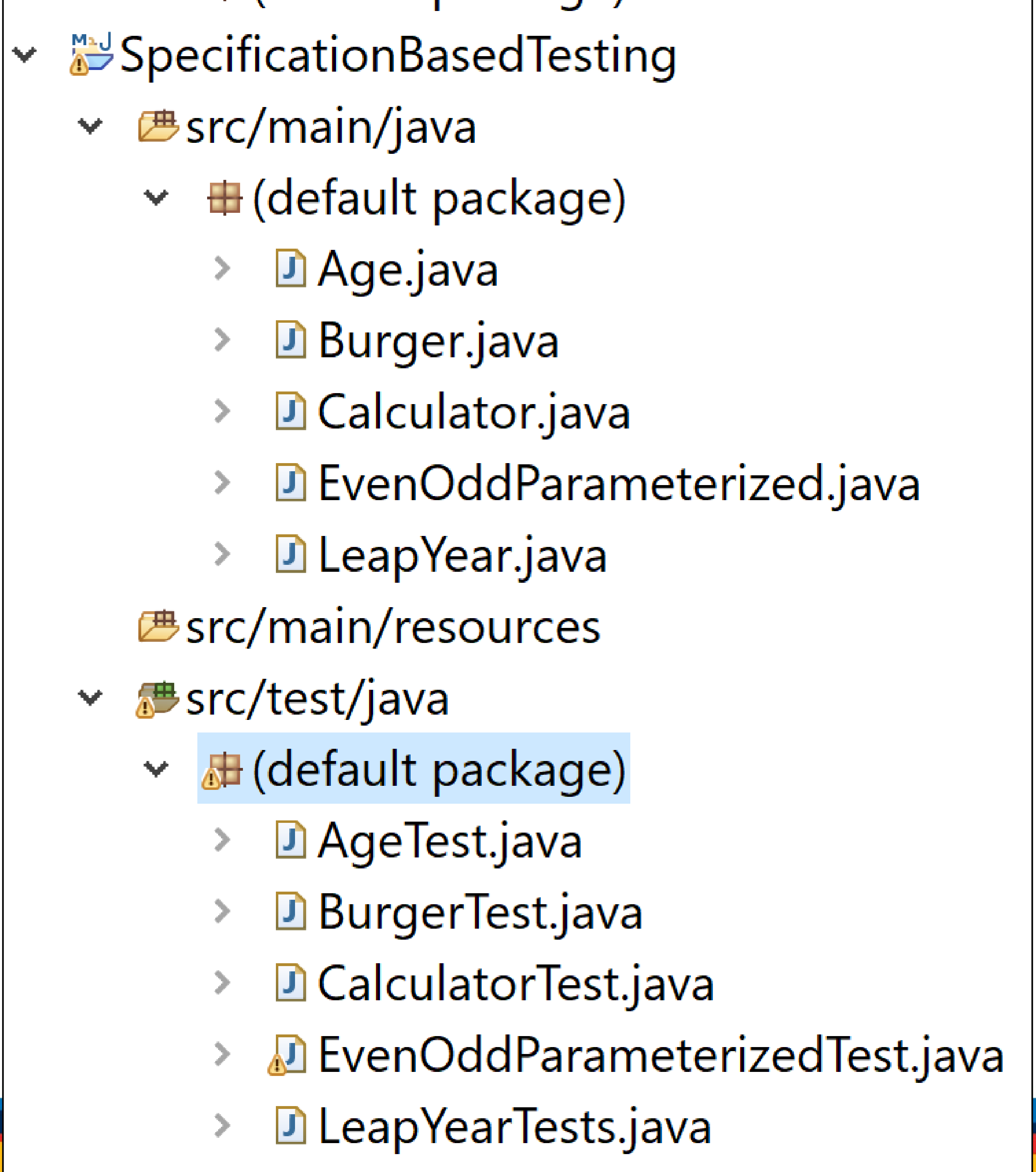
- Junit is a Java testing framework used to test a piece of java code.
- It has a **Runner** which determines the tests, executes those tests, validates these tests using **assertions** and **reports the results to the developers.**



```
Finished after 0.389 seconds
Runs: 2/2    Errors: 0    Failures: 1

AssertFailTest [Runner: JUnit 5] (0.100 s)
    testAssertFail() (0.029 s)
    testExceptionIsThrown() (0.071 s)

Failure Trace
org.opentest4j.AssertionFailedError: Not yet implemented
```

Where should the test be located?

- Typically, unit tests are created in a **separate source folder** to keep the test code separate from the real code.
- The standard convention from the Maven and Gradle build tools is to use:
- **src/main/java - for Java classes**
- **src/test/java - for test classes**



```
SpecificationBasedTesting
    src/main/java
        (default package)
            Age.java
            Burger.java
            Calculator.java
            EvenOddParameterized.java
            LeapYear.java
    src/main/resources
    src/test/java
        (default package)
            AgeTest.java
            BurgerTest.java
            CalculatorTest.java
            EvenOddParameterizedTest.java
            LeapYearTests.java
```

JUNIT ASSERTIONS AND ANNOTATIONS

University of Victoria

Assertions and Annotations

- The simplest form of **self-testing** is the *assertion*.
- An assertion is a **Boolean expression** at a specific point in a program which will be **true unless there is a bug** in the program.
- In other words, an **assertion states that a certain condition must be true at the time the assertion is executed**.
- JUnit **Annotations** is a special form of syntactic meta-data that can be added to Java source code for better code readability and structure.

Annotations for Junit testing

@Test annotation is applied over methods to mark them as test methods.

Visibility of @Test annotated methods can be made public, default and protected in Junit 5 but in Junit 4 they can only be public.

@Test(timeout=1000) annotation specifies that method will be failed if it takes longer than 1000 milliseconds (1 second).

@BeforeClass/@BeforeAll annotation specifies that method will be invoked only once, before starting all the tests. JUnit 5 supports @BeforeAll instead of @BeforeClass. Sometimes several tests need to share computationally expensive setup (like logging into a database). While this can compromise the independence of tests, sometimes it is a necessary optimization.

@Before/@BeforeEach annotation specifies that method will be invoked before each test case. This annotation is commonly used to develop necessary preconditions for each @Test method. JUnit 5 supports @BeforeEach instead of @Before. E.g., clearing the changes made by a test to a **list** before the next test.

```
public class MyTestClass {

   @BeforeClass
   public void initGlobalResources() {

         /* This method will be called only once per test class.  */
   }

   @Before
   public void initializeResources() {
         / * This method will be called before calling every test. */
   }

   @Test
   public void myTestMethod1() {
       /* initializeResources() method will be called before calling this method */
   }
```

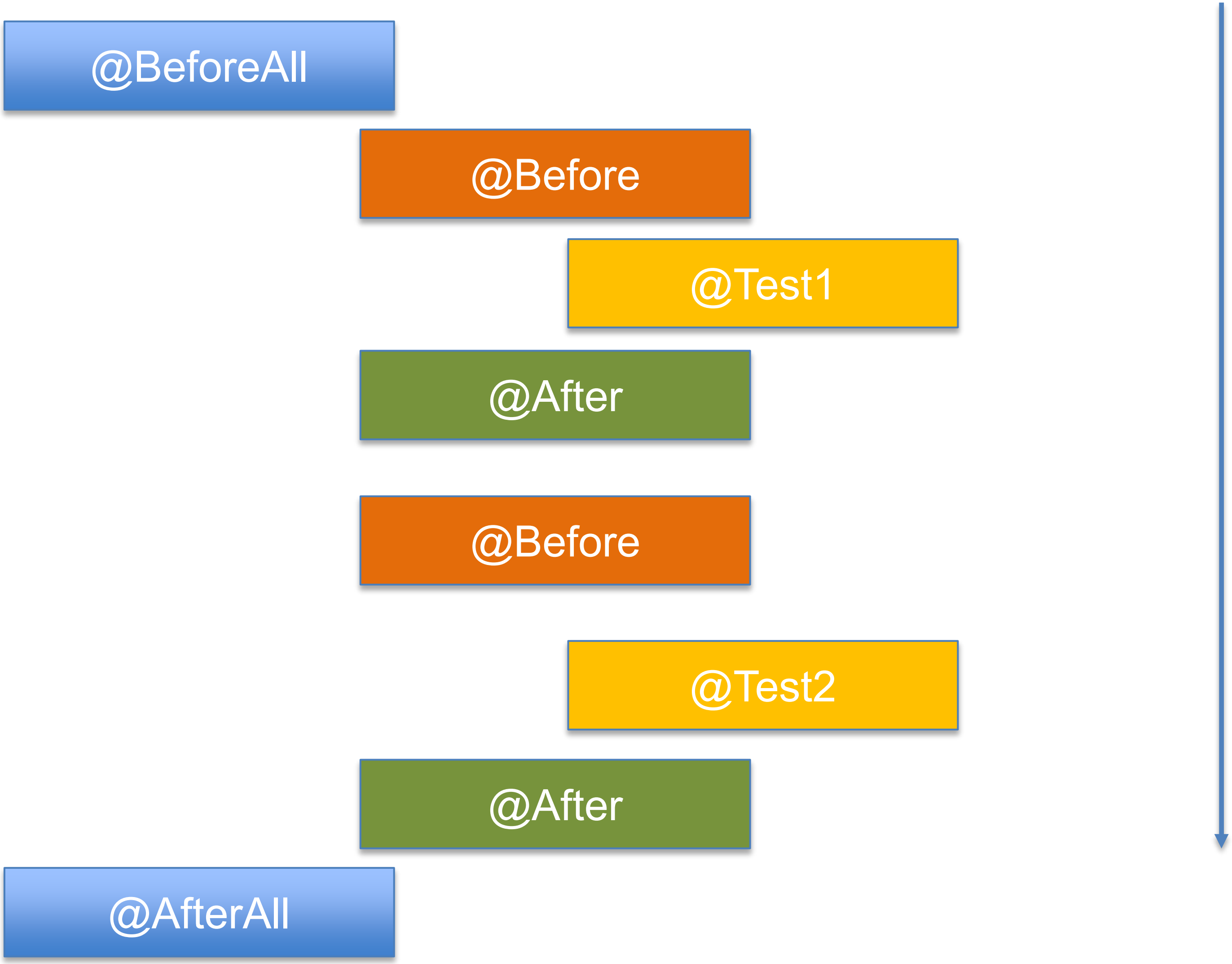
Annotations for Junit 4 testing

@AfterClass/@AfterAll annotation specifies that method will be invoked only once, after finishing all the tests. If you allocate expensive external resources in a @BeforeClass method, you need to release them after all the tests in the class have run. Annotating a public static void method with @AfterClass causes that method to be run after all the tests in the class have been run.

@After/@AferEach annotation specifies that method will be invoked after each test case. The annotations @AfterClass and @After are same in functionality. The only difference is the method annotated with @AfterClass will be called once per test class based, and the method annotated with @After will be called once per test based.

```
public class MyTestClass {

    @Test
    public void myTestMethod1() {
        // write your test code here...
    }
     @After
    public void initResources() {
        /**
         * This method will be called after every test method.
         */
    }
    @AfterClass
    public void closeGlobalResources() {
        /**
         * This method will be called only once per test class. It will be called
         * after executing all tests.
         */
    }
```

Order of execution



@BeforeAll

@Before

@Test1

@After

@Before

@Test2

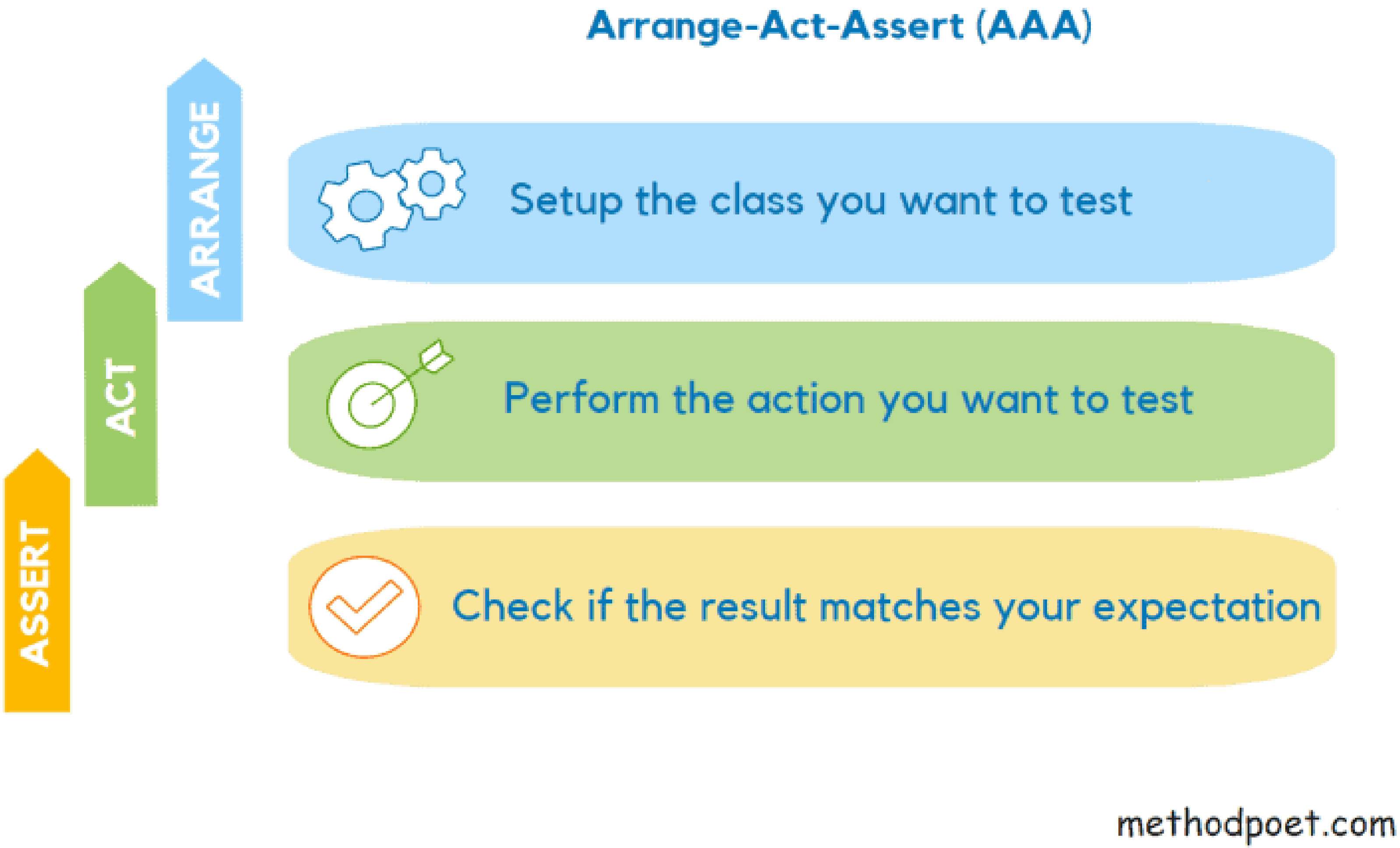
@After

@AfterAll

THE AAA STRATEGY

University of Victoria

AAA Strategy



Arrange-Act-Assert (AAA)

ARRANGE

Setup the class you want to test

ACT

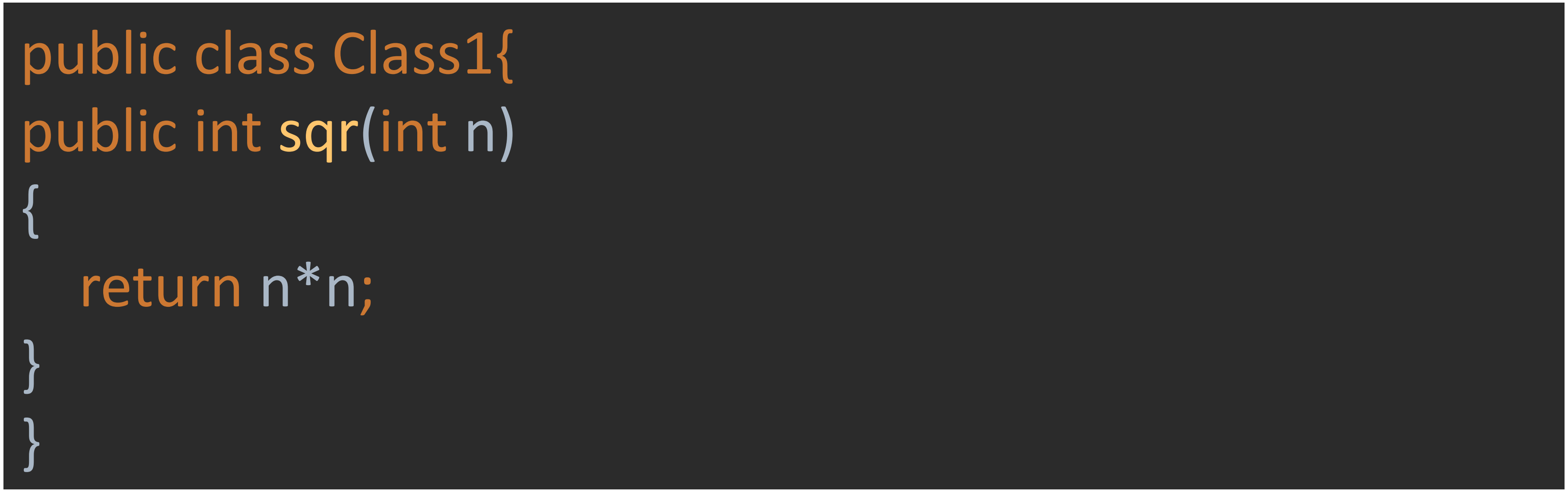
Perform the action you want to test

ASSERT

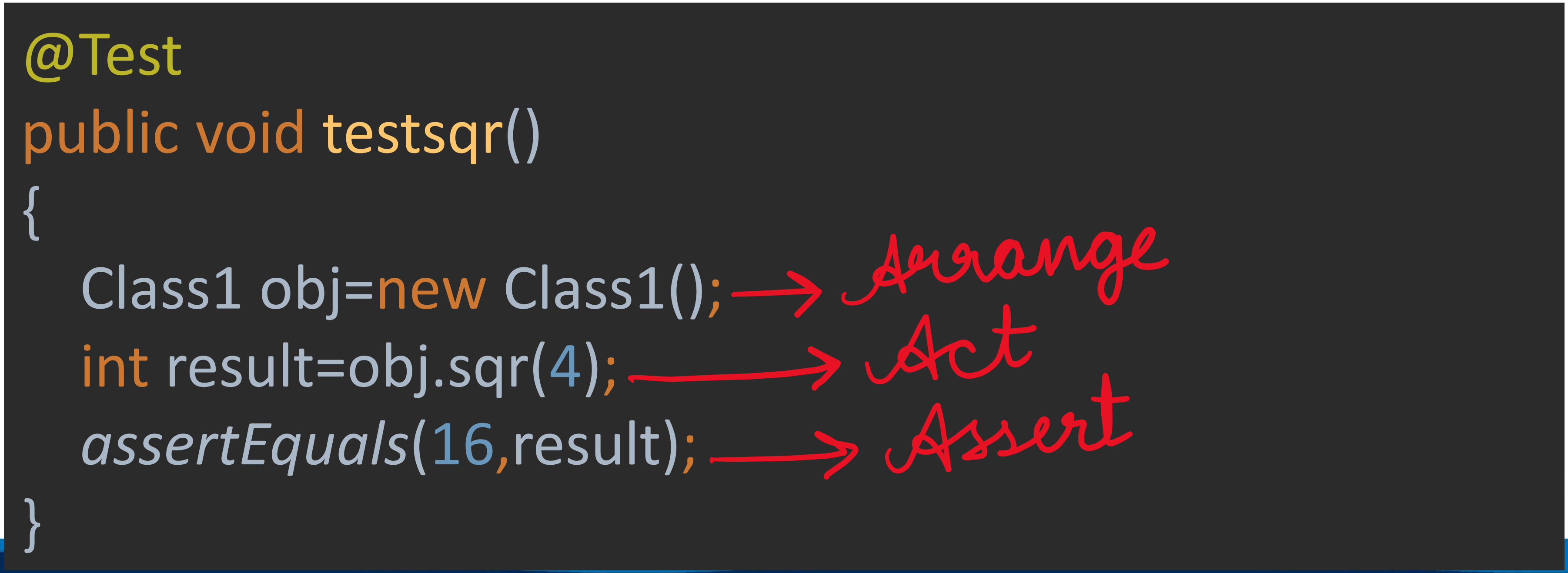
Check if the result matches your expectation

methodpoet.com

Program to find square of an integer



```
public class Class1{
public int sqr(int n)
{
   return n*n;
}
}
```



```
@Test
public void testsqr()
{
   Class1 obj=new Class1(); → Arrange
   int result=obj.sqr(4); → Act
   assertEquals(16,result); → Assert
}
```

LET’S TRY SOME ASSERTIONS AND ANNOTATIONS

Assert (the e-a order)

- **assertEquals**(expected, actual)
- **assertTrue**(condition)
- **assertFalse**(condition)
- **assertArrayEquals**(expectedArray, actualArray);
- **assertThat**(some_object_or_value).has_some_relation_to(some_other_object_or_value)

assertThat

- The assertThat assertion is the only one in JUnit 4 that has a **reverse order (a-e)** of the parameters compared to the other assertions.

assertThat([value], [matcher statement]);

- assertThat(some_number).**isNotEqualTo**(some_other_number);
- assertThat(some_object_reference).**isNotNull**();
- assertThat(some_object).**isSameAs**(some_other_object_reference);
- assertThat(some_condition).**isTrue**();
- assertThat(some_condition).**isFalse**();

The Benefit of Using AssertThat Over Other Assert Methods

Readability:

“**Assert that the actual value is equal to the expected value 100**.”

- assertThat(actual, equalTo(100));

//OR

- assertThat(actual, is(equalTo(100)));

//OR

assertThat(actual, is(100));

Create a Calculator class and test it

- Create a ‘Calculator’ class
- In it create functions
  - –int doSum(int a, int b) which calculates sum of two integers
  - –int doProduct(int a, int b) which calculates product of two integers
  - –Boolean compareTwoNums(int a, int b) which compares two integers for equality.
- Now create tests for testing these functions
  - –void testSum()
  - –void testProduct()
  - –void testCompare()
- Now create annotations
  - –to be performed before every test @Before
  - –before the entire class @BeforeClass
  - –which must be performed after each test @After
  - –to be performed after the entire class @AfterClass

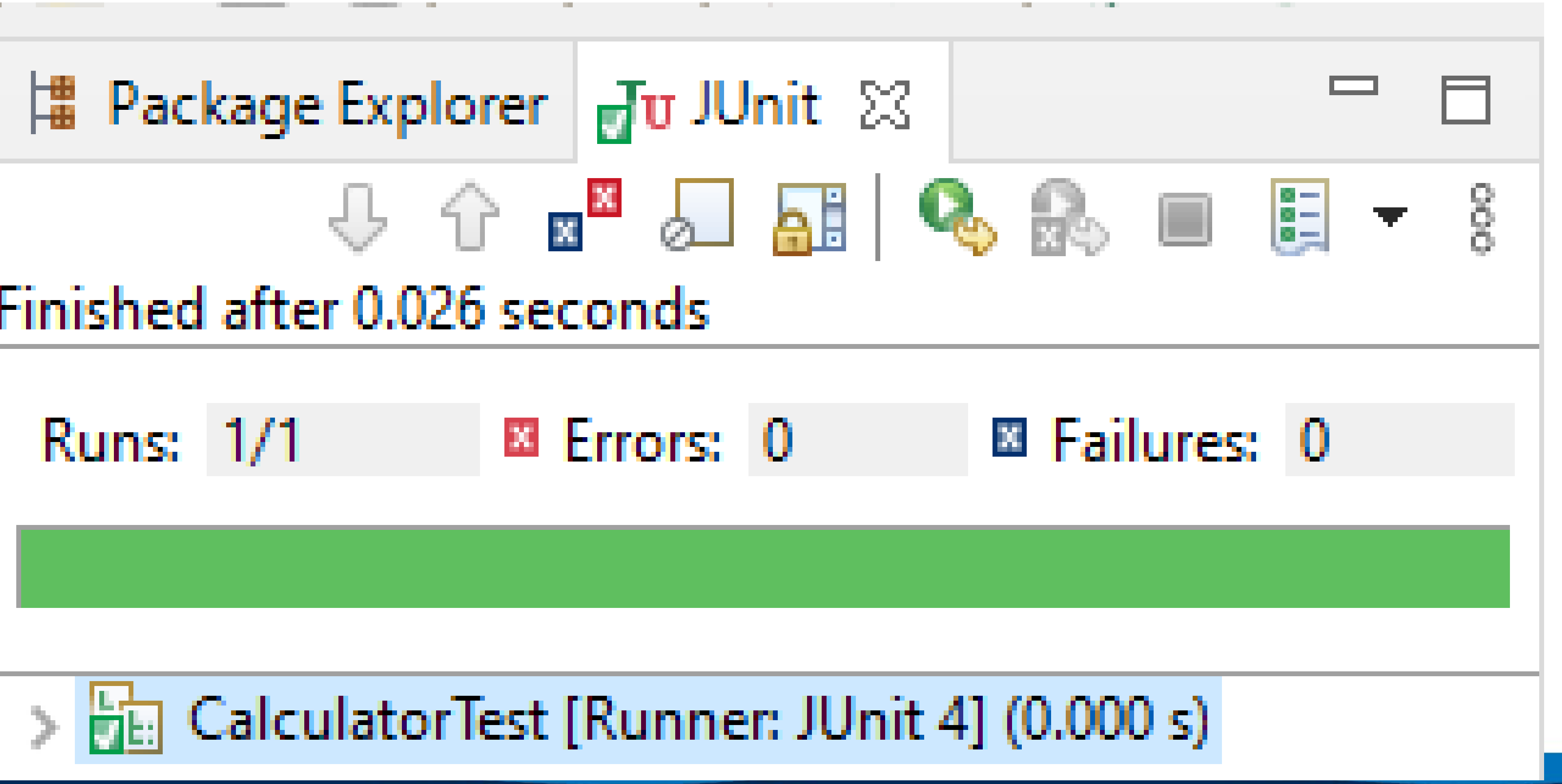
Calculator class

```
public class Calculator {
    //sum
    public int doSum(int a, int b) {
        return a + b;
    }

    //product
    public int doProduct(int a, int b) {
        return a * b;
    }

    //compare
    public Boolean compareTwoNums(int a, int b) {
        return a == b;
    }
}
```

Test doSum() function using `CalculatorTest` junit class

```java
public class CalculatorTest{
Calculator c=new Calculator();
@Test
public void testSum()
{
int expected=17;
int actual=c.doSum(12, 3);
assertEquals(expected, actual);
/*note: sometimes Assert.assertEquals() might work if
assertEquals() is deprecated*/
}
}
```

Test passes



Finished after 0.026 seconds

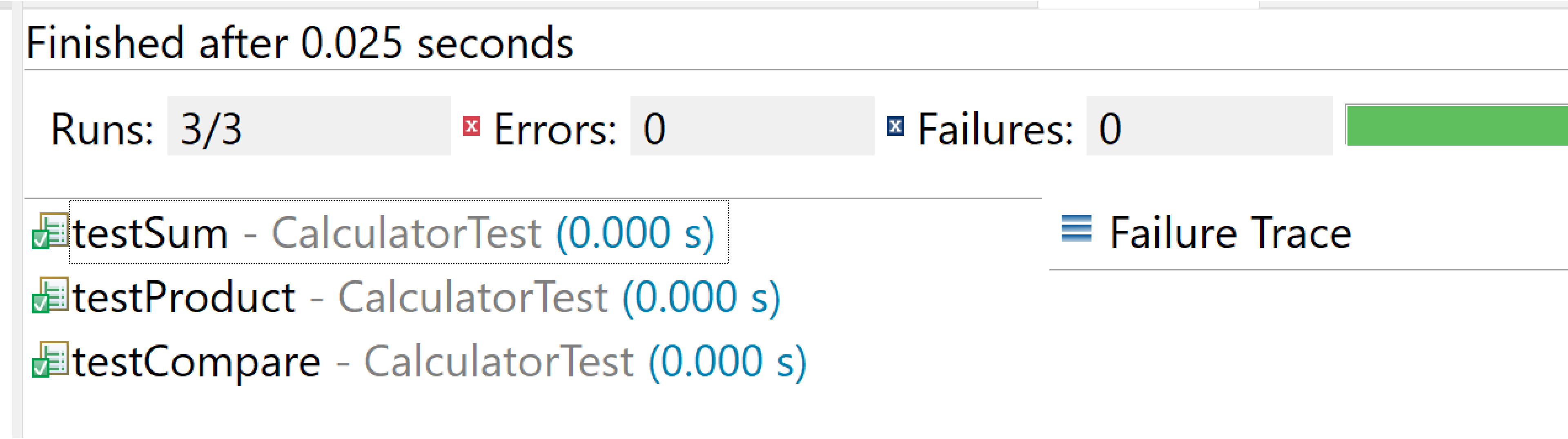
Runs: 1/1 Errors: 0 Failures: 0

CalculatorTest [Runner: JUnit 4] (0.000 s)

Write the entire test class

```
public class CalculatorTest {
Calculator c=new Calculator();
@Test
public void testSum()
{
int expected=70;
int actual=C.doSum(30,40);
assertEquals(expected,actual);
System.out.println("The Sum is:
"+actual);
}
@Test
public void testProduct()
{
int expected=35;
int actual=C.doProduct(5,7);
assertEquals(expected, actual);
System.out.println("The Product is:
"+actual);
}
@Test
public void testCompareTrue()
{
boolean
actual=C.compareTwoNums(12,12);
assertTrue(actual);
System.out.println("The Comparison
is: "+actual);
}
@Test
public void testCompareFalse()
{
boolean
actual=C.compareTwoNums(12,1);
assertTrue(actual);
System.out.println("The Comparison
is: "+actual);
}
```

Running all 3 tests



Finished after 0.025 seconds

Runs: 3/3 Errors: 0 Failures: 0

testSum - CalculatorTest (0.000 s)

testProduct - CalculatorTest (0.000 s)

testCompare - CalculatorTest (0.000 s)

Complete CalculatorTest Class

- test functions
  - void testSum()
  - void testProduct()
  - void testCompare()
- Before and after annotations
  - to be performed before every test @Before
  - before the entire class @BeforeClass
  - which must be performed after each test @After
  - to be performed after the entire class @AfterClass

```
public class CalculatorTest {
@BeforeClass
/*This will be executed before the entire
class*/
public static void beforeClassMethod()
{System.out.println("Establishing
Connection to the database");}
Calculator C;
@Before
/*This will be executed before each test
method*/
public void init()
{System.out.println("Initializing the
calculator instance");
C=new Calculator();}
@Test
public void testSum()
{int expected=70;
int actual=C.doSum(30,40);
assertEquals(expected,actual);
System.out.println("The Sum is:
"+actual);}
@Test
public void testProduct()
{int expected=35;
int actual=C.doProduct(5,7);
assertEquals(expected, actual);
System.out.println("The Product is:
"+actual);}
@Test
public void testCompareTrue()
{boolean actual=C.compareTwoNums(12,12);
assertTrue(actual);
System.out.println("The Comparison is:
"+actual);}
@Test
public void testCompareFalse()
{boolean actual=C.compareTwoNums(12,1);
assertTrue(actual);
System.out.println("The Comparison is:
"+actual);}
@After
public void tearDown()
{System.out.println("Test method executed
successfully");}
@AfterClass
/*This will be executed after the entire
class. There are no rules for ordering the
functions*/
public static void afterClassMethod()
{System.out.println("Tearing down
Connection to the database");}
}
```

Output

```
•Establishing Connection to the database

  –Initializing the calculator instance
    •The Sum is: 70
  –Test method executed successfully

  –Initializing the calculator instance
    •The Comparison is: true
  –Test method executed successfully

  –Initializing the calculator instance
    •The Product is: 35
  –Test method executed successfully

•Tearing down Connection to the database
```

For Intellij

- Create a Gradle project.
- In src.main.java folder, create a Calculator class and in the src.test.java folder create a CalculatorTest class.
- Rest remains the same.

TRY YOURSELF

For testing doSum() function having 3 integers, write 4 tests:

1. Test case which adds **three positive** integers (testSum1) :[30,40,50]=120
2. Test case which adds **three negative** (testSum2) :[-30,-40,-50]= -120
3. Test case which adds **three same** integers (testSum3) :[30,30,30]= 90
4. Test case which adds two **positive integers with 0** (testSum4) :[30,30,0]=60

```
public class CalculatorTest {
Calculator c;

@Before
public void init()
{
c=new Calculator();
}

@Test
public void testSum1()
{
int expected=120;
int actual=c.doSum(30,40,50);
assertEquals(expected, actual);
}
@Test
public void testSum2()
{
int expected=-120;
int actual=c.doSum(-30,-40,-50);
assertEquals(expected, actual);
}
@Test
public void testSum3()
{
int expected=90;
int actual=c.doSum(30,30,30);
assertEquals(expected, actual);
}
@Test
public void testSum4()
{
int expected=60;
int actual=c.doSum(30,30,0);
assertEquals(expected, actual);
}
```

Create a program to Calculate SI,CI and test it

$$SI = p \cdot r \cdot t$$

Where:

SI is the Simple **interest** paid

P is the **principal**—the original amount of money borrowed

R is the **interest rate**, a per-year rate, written as a decimal

T is the **time** of the loan, expressed in years or portions of a year

Write a program to calculate SI and CI and test the program using the values given in the solved exercises in the following slides.

Solved Example of SI and CI

Treasury Notes (T-notes) are bonds issued by the federal government to cover its expenses. Suppose you obtain a $1,000 T-note with a 4% annual rate, with a maturity in 2 years. How much interest will you earn?

Solution:

Identify the information given in the problem.

Simple Interest (SI): unknown

Principal (p): $1000

Rate (r): 4%=0.04

Time (t): 2 years

Put the information in the simple interest equation.

$$SI=1000\cdot 0.04\cdot 2$$

Multiply.

$$SI=80$$

Answer

You would earn $80 in interest.

Compound Interest

$$CI=P(1+r/n)^{nt} -P$$

Where:

- P is the principal amount
- r is the rate of interest(decimal)
- n is frequency or no. of times the interest is compounded annually. We will consider frequency to be =1. then

$$CI=P(1+r)^{t} -P$$

- t is the overall tenure.

Solved question

- If we invest \$50,000 in an investment account paying 10% interest compounded annually, how much will the CI be in 5 years?
- Because we are starting with \$50,000, P=50,000. Our interest rate is 10%, so r=0.1. We want to know the CI in 5 years, so

$$CI=P(1+r)^{t}-P$$

Use the compound interest formula.

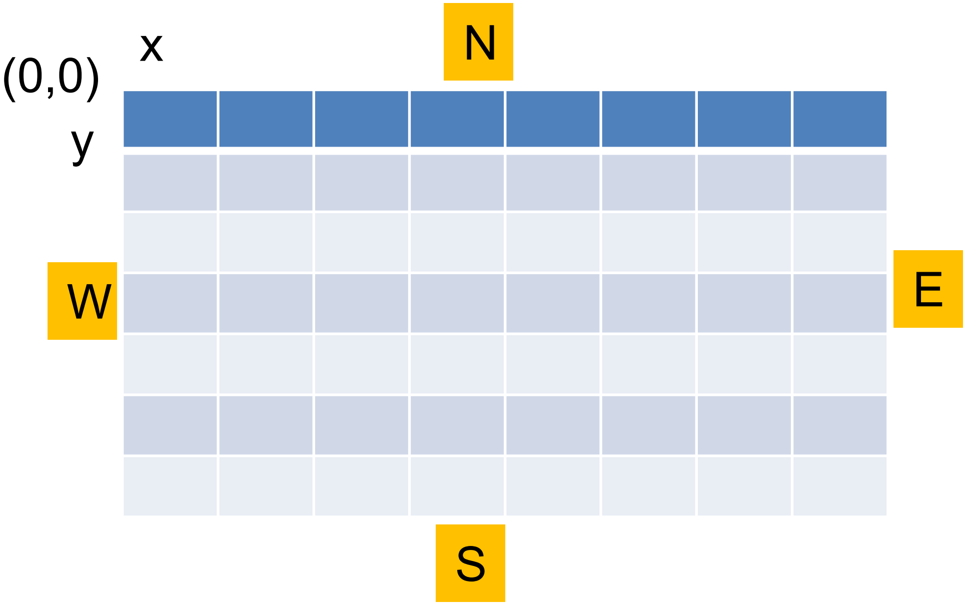
$$CI=50000*(1+0.1)^{10}*5= 30525.50000000003$$

Note: use Math.pow() function. E.g., Math.pow(2,4) is 2 raised to power 4 which is 16.

Jpacman again – Direction test !!!

1. Complete the unit tests in the board.DirectionTest class in JPacman.

- Open the board.DirectionTest.
- Create additional test methods in DirectionTest for e.g., the south, east, and west directions. Test for ‘north’ is done for you.
- Run the tests, and ensure they pass.



Fail a test

- In your IDE, modify one or more of your test cases so that they fail.
- Repair the tests so that they pass again.