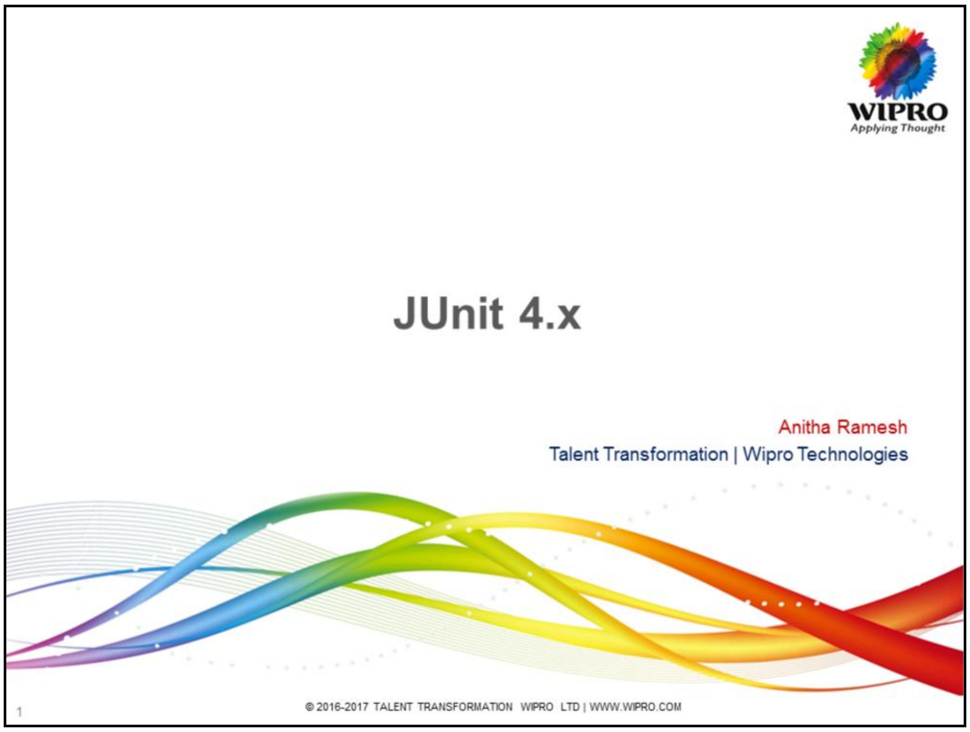
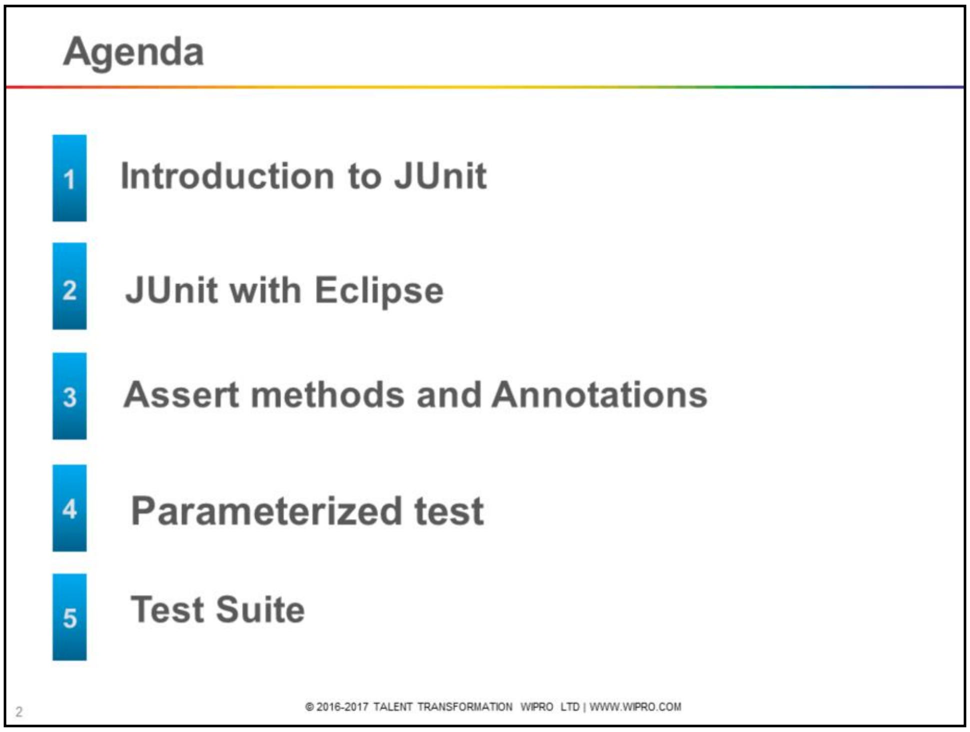


JUnit 4.x

Anitha Ramesh
Talent Transformation | Wipro Technologies

1



Agenda

1 Introduction to JUnit

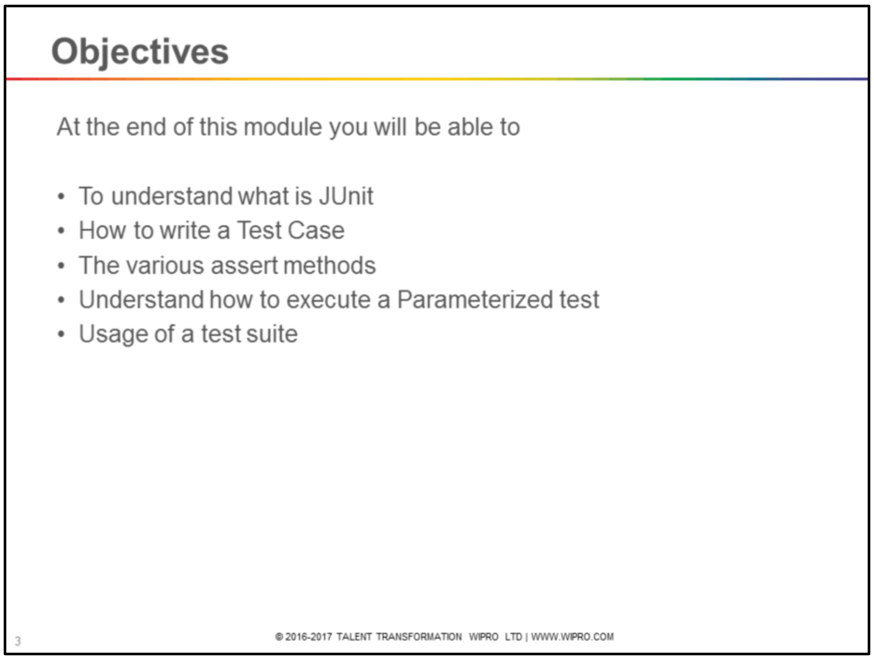
2 JUnit with Eclipse

3 Assert methods and Annotations

4 Parameterized test

5 Test Suite

2

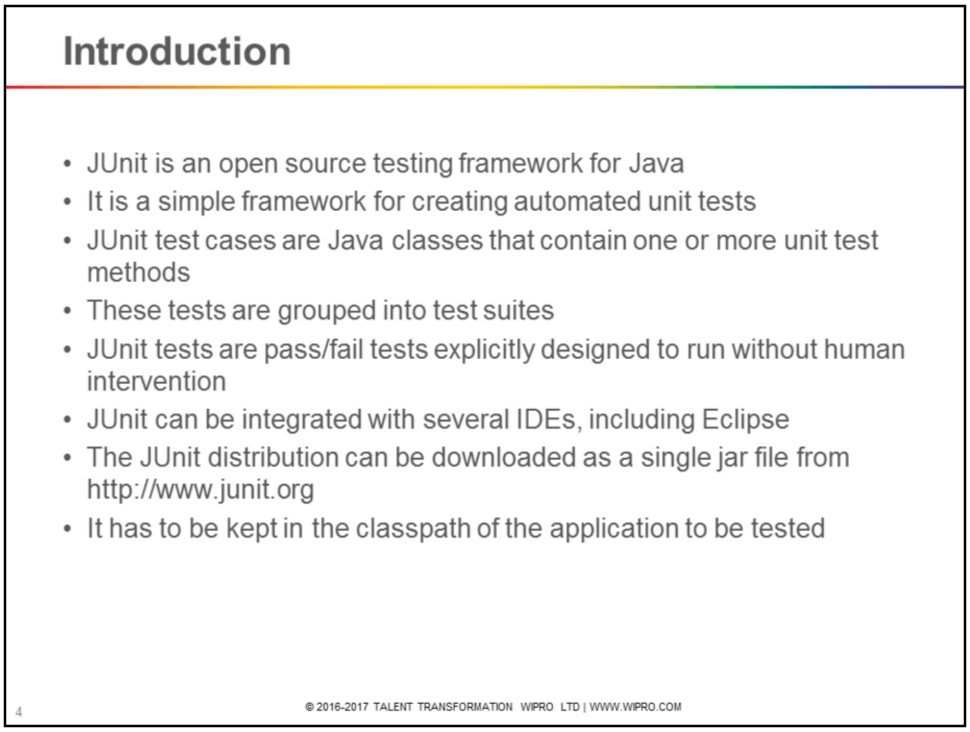


Objectives

At the end of this module you will be able to

- To understand what is JUnit
- How to write a Test Case
- The various assert methods
- Understand how to execute a Parameterized test
- Usage of a test suite

3


Introduction

- JUnit is an open source testing framework for Java
- It is a simple framework for creating automated unit tests
- JUnit test cases are Java classes that contain one or more unit test methods
- These tests are grouped into test suites
- JUnit tests are pass/fail tests explicitly designed to run without human intervention
- JUnit can be integrated with several IDEs, including Eclipse
- The JUnit distribution can be downloaded as a single jar file from http://www.junit.org
- It has to be kept in the classpath of the application to be tested

4

Unit Testing

A unit test is a piece of code written by a developer that executes a specific functionality in the code under test. Unit tests ensure that code is working as intended and validate that this is still the case after code changes.

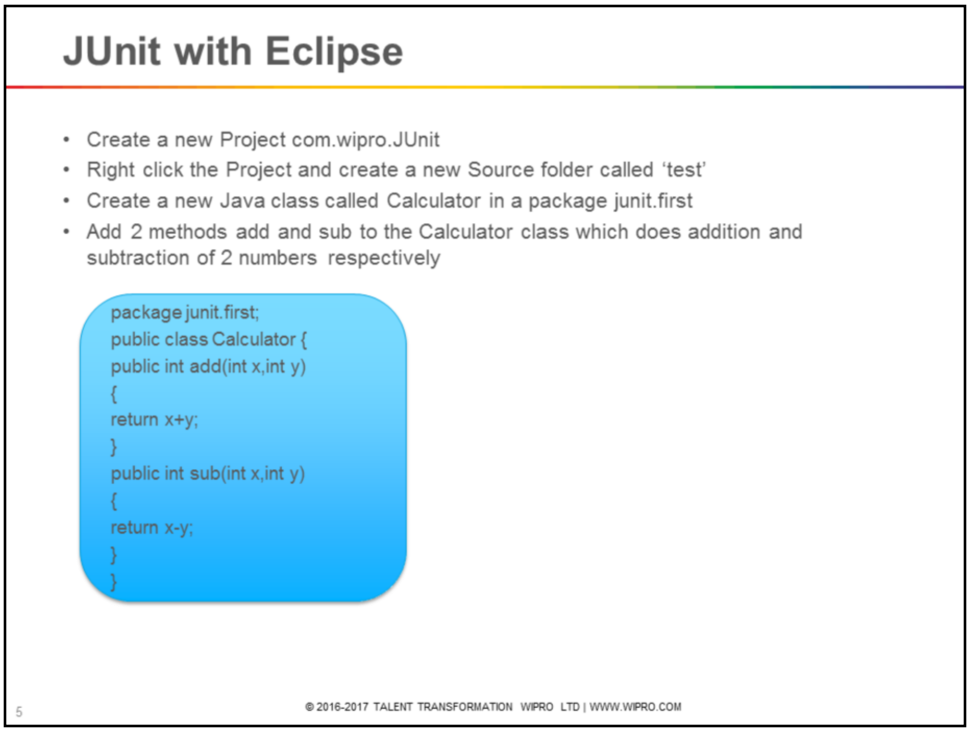
Unit Testing with JUnit

JUnit 4.x is a test framework which uses annotations to identify methods that are tests. JUnit assumes that all test methods can be executed in an arbitrary order. Therefore tests should not depend on other tests.

Installation of JUnit

If you use Eclipse you can use the integrated JUnit in Eclipse for your testing.

If you want to control the used JUnit library explicitly, download JUnit4.x.jar from the JUnit website at http://www.junit.org/ . The download contains the "junit-4.*.jar" which is the JUnit library. Add this library to your Java project and add it to the classpath.



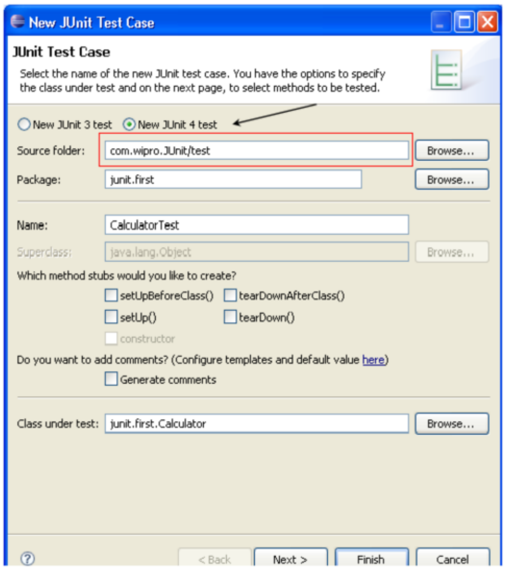
JUnit with Eclipse

- Create a new Project com.wipro.JUnit
- Right click the Project and create a new Source folder called 'test'
- Create a new Java class called Calculator in a package junit.first
- Add 2 methods add and sub to the Calculator class which does addition and subtraction of 2 numbers respectively

```
package junit.first;
public class Calculator {
public int add(int x,int y)
{
return x+y;
}
public int sub(int x,int y)
{
return x-y;
}
}
```

5

JUnit with Eclipse


New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

New JUnit 3 test [✓ New JUnit 4 test]

Source folder: com.wipro.JUnit/test

Package: junit.first

Name: CalculatorTest

Superclass: java.lang.Object

Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()

☐ setUp() ☐ tearDown()

☐ constructor

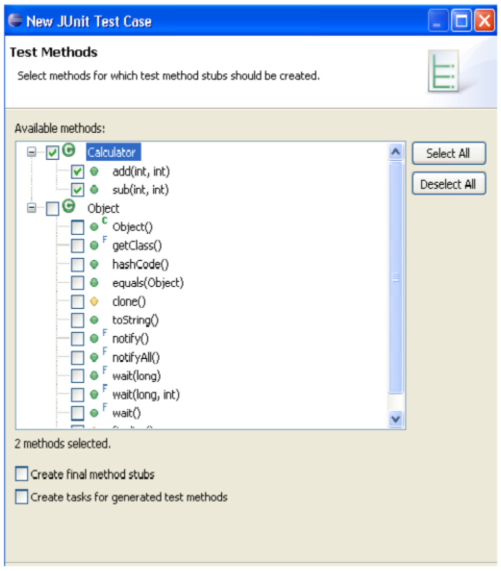
Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

Class under test: junit.first.Calculator

- Right click on the Calculator class in the Package Explorer and select New->JUnitTestCase
- select "New JUnit4 test"
- set the source folder to "test" – the test class gets created here

JUnit with Eclipse



New JUnit Test Case

Test Methods

Select methods for which test method stubs should be created.

Available methods:

- [✓] Calculator
  - [✓] add(int, int)
  - [✓] sub(int, int)
- [ ] Object
  - [ ] Object()
  - [ ] getClass()
  - [ ] hashCode()
  - [ ] equals(Object)
  - [ ] clone()
  - [ ] toString()
  - [ ] notify()
  - [ ] notifyAll()
  - [ ] wait(long)
  - [ ] wait(long, int)
  - [ ] wait()

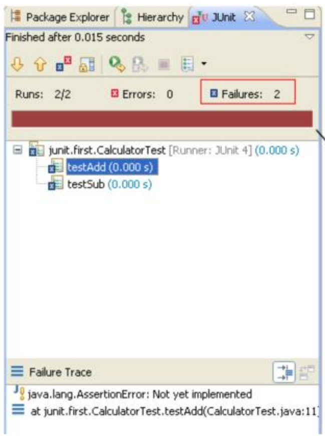
2 methods selected.

[ ] Create final method stubs

[ ] Create tasks for generated test methods

- Press "Next" and select the methods you want to test

JUnit with Eclipse

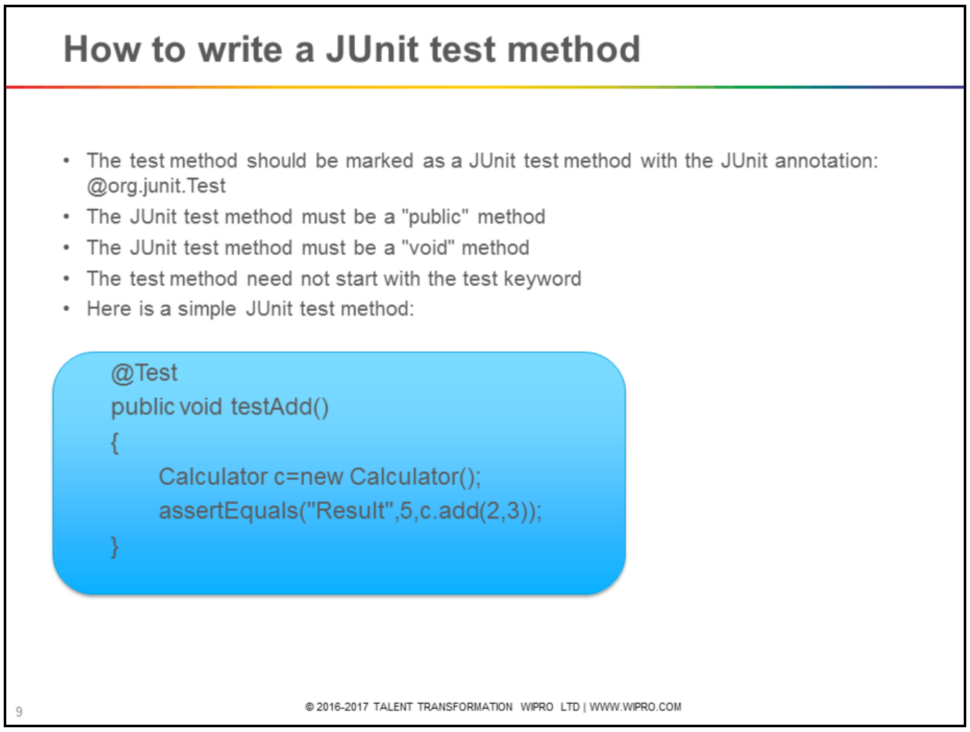


```
Finished after 0.015 seconds
Runs: 2/2    Errors: 0    Failures: 2

junit.first.CalculatorTest [Runner: JUnit 4] (0.000 s)
    testAdd (0.000 s)
    testSub (0.000 s)

java.lang.AssertionError: Not yet implemented
at junit.first.CalculatorTest.testAdd(CalculatorTest.java:11)
```

- Right click on CalculatorTest class and select

  Run-As → JUnit Test
- The results of the test will be displayed in JUnit view
- This is because the testAdd and testSub are not implemented correctly

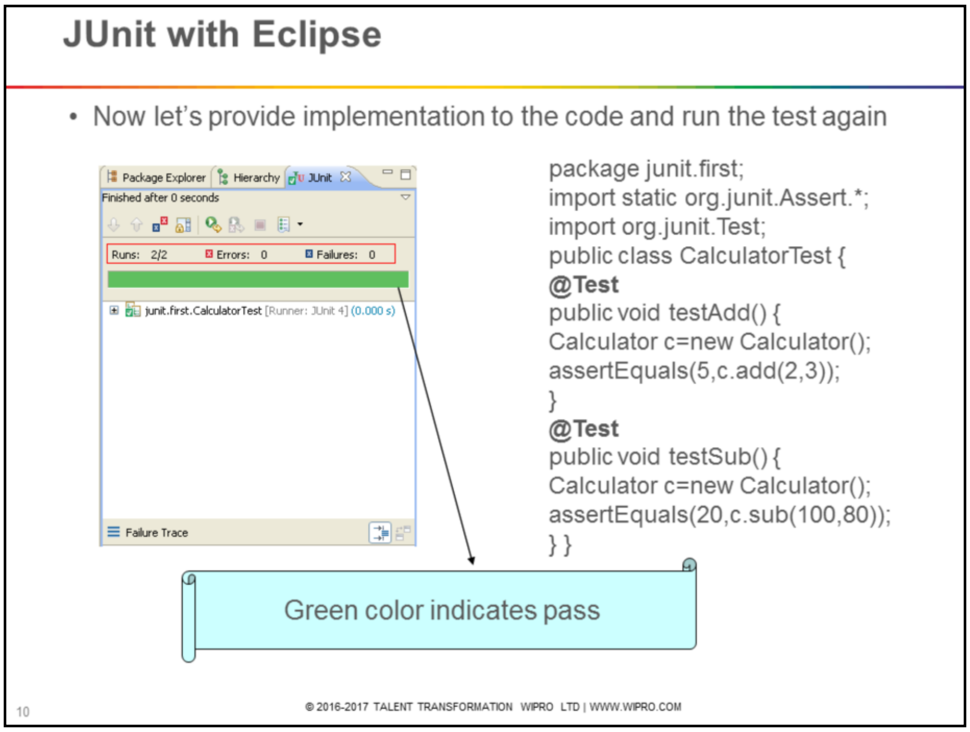
Brown color indicates failure



How to write a JUnit test method

- The test method should be marked as a JUnit test method with the JUnit annotation: @org.junit.Test
- The JUnit test method must be a "public" method
- The JUnit test method must be a "void" method
- The test method need not start with the test keyword
- Here is a simple JUnit test method:

```
@Test
public void testAdd()
{
      Calculator c=new Calculator();
      assertEquals("Result",5,c.add(2,3));
}
```



JUnit with Eclipse

- Now let's provide implementation to the code and run the test again

Package Explorer
Hierarchy
JUnit
Failure Trace

```
Finished after 0 seconds
Runs: 2/2    Errors: 0    Failures: 0
junit.first.CalculatorTest [Runner: JUnit 4] (0.000 s)
```

```
package junit.first;
import static org.junit.Assert.*;
import org.junit.Test;
public class CalculatorTest {
@Test
public void testAdd() {
Calculator c=new Calculator();
assertEquals(5,c.add(2,3));
}
@Test
public void testSub() {
Calculator c=new Calculator();
assertEquals(20,c.sub(100,80));
} }
```

Green color indicates pass

10

© 2016-2017 TALENT TRANSFORMATION WIPRO LTD | WWW.WIPRO.COM

Unit tests are implemented as classes with test methods. Each test method usually tests a single method of the target class. Sometimes, a test method can test more than one method in the target class, and sometimes, if the method to test is big, you split the test into multiple test methods.

The unit test class is an ordinary class, with two methods, tesAdd() and testSub. Notice how this method is annotated with the JUnit annotation @Test. This is done to signal to the unit test runner, that this is method represents a unit test, that should be executed. Methods that are not annotated with @Test are not executed by the test runner.

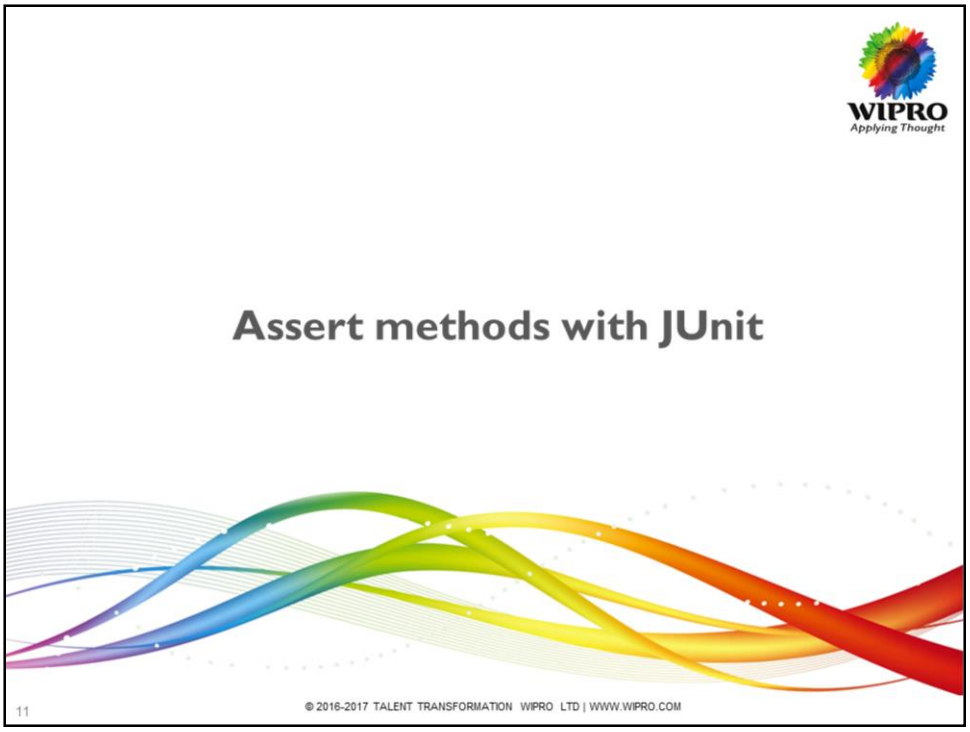
Inside the testAdd() method an instance of Calculator is created. Then it's add() method is called with two integer values.

Finally, the assertEquals() method is called. It is this method that does the actual testing. In this method we compare the output of the called method (add()) with the expected output.

If the two values are equal, nothing happens. The assertEquals() method returns normally. If the two values are not equal, an exception is thrown, and the test method stops executing here.

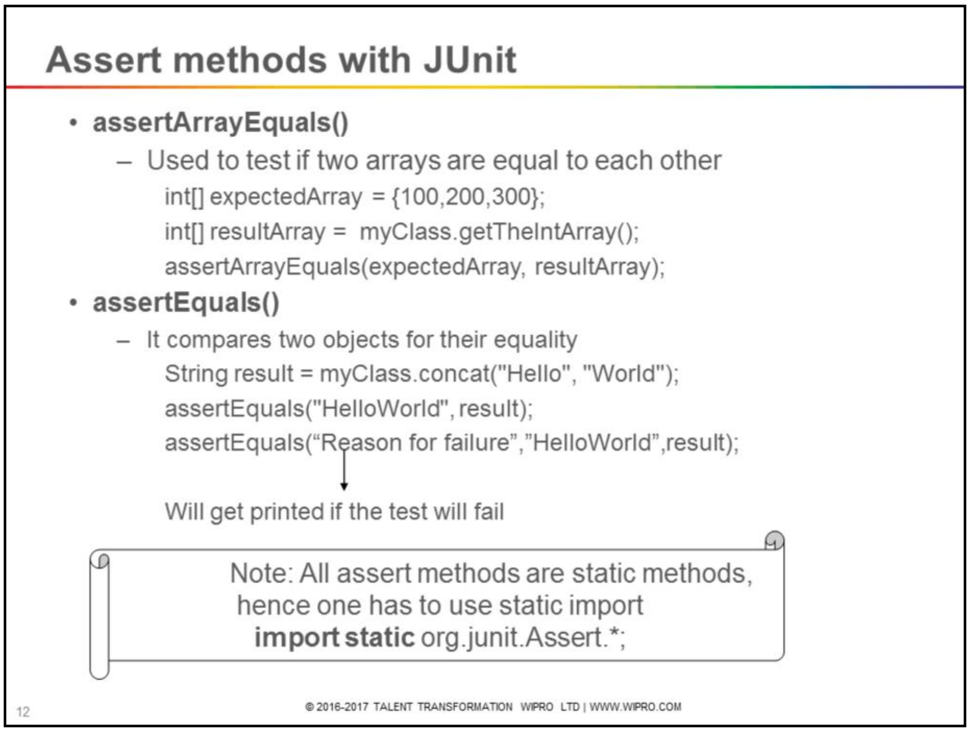
The assertEquals() method is a statically imported method, which normally resides in the org.junit.Assert class. Notice the static import of this class at the top of MyUnitTest. Using the static import of the method is shorter than writing Assert.assertEquals().

You can have as many test methods in a unit test class as you want. The test runners will find them all, and execute each of them.



Assert methods with JUnit

11


Assert methods with JUnit

- **assertArrayEquals()**
  - Used to test if two arrays are equal to each other

```
int[] expectedArray = {100,200,300};
int[] resultArray =  myClass.getTheIntArray();
assertArrayEquals(expectedArray, resultArray);
```

- **assertEquals()**
  - It compares two objects for their equality

```
String result = myClass.concat("Hello", "World");
assertEquals("HelloWorld", result);
assertEquals("Reason for failure","HelloWorld",result);
```

Will get printed if the test will fail

Note: All assert methods are static methods, hence one has to use static import **import static** org.junit.Assert.*;

12

assertArrayEquals()

The assertArrayEquals() method will test whether two arrays are equal to each other. In other words, if the two arrays contain the same number of elements, and if all the elements in the array are equal to each other.

To check for element equality, the elements in the array are compared using their equals() method. More specifically, the elements of each array are compared one by one using their equals() method. Which means, that it is not enough that the two arrays contain the same elements. They must also be present in the same order.

If the arrays are equal, the assertArrayEquals() will proceed without errors. If the arrays are not equal, an exception will be thrown, and the test aborted. Any test code after the assertArrayEquals() will not be executed.

assertEquals

First the myClass.concat() method is called, and the result is stored in the variable result. Second, the result value is compared to the expected value "HelloWorld", using the assertEquals() method.

If the two objects are equal according to their implementation of their equals() method, the assertEquals() method will return normally. Otherwise the assertEquals() method will throw an exception, and the test will stop there.

This example compared to String objects, but the assertEquals() method can compare any two objects to each other. The assertEquals() method also come in versions which compare primitive types like int and float to each other.

The new assertEquals methods use Autoboxing, and hence all the assertEquals(primitive, primitive) methods will be tested as assertEquals(Object, Object). This may lead to some interesting results. For example autoboxing will convert all numbers to the Integer class, so an Integer(10) may not be equal to Long(10). This has to be considered when writing tests for arithmetic methods. For example, the following Calc class and it's corresponding test CalcTest will give you an error.

```
public class Calc {
 public long add(int a, int b) {
  return a+b;
 }
}
```

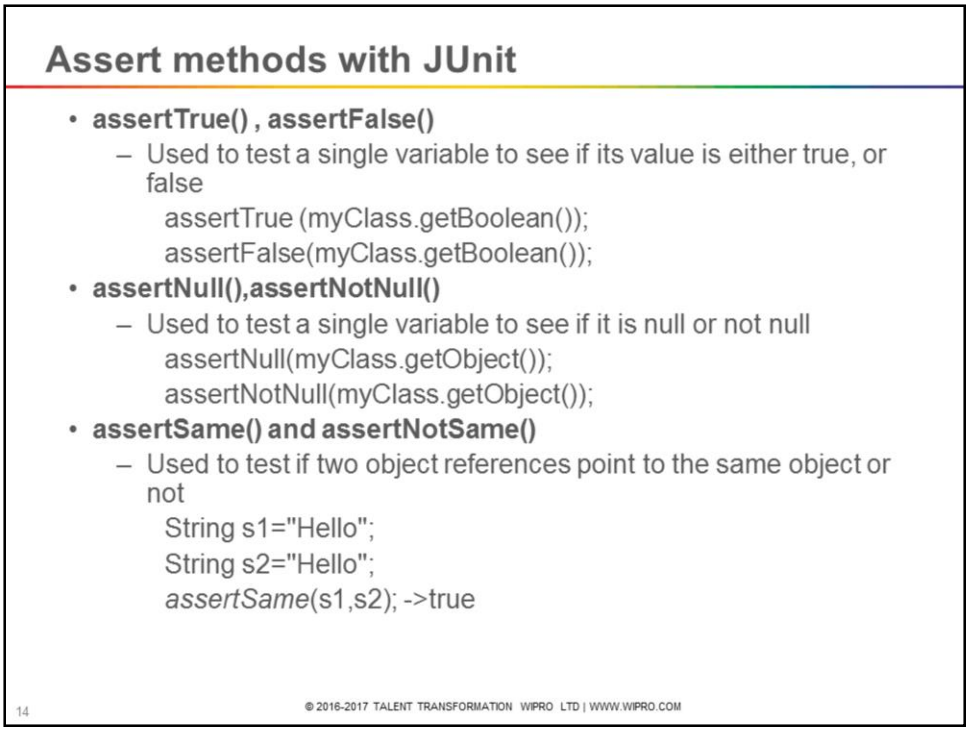
```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class CalcTest {
 @Test
 public void testAdd() {
  assertEquals(5, new Calc().add(2, 3));
 }
}
```

You will end up with the following error.

```
 java.lang.AssertionError: expected:&lt;5&gt; but was:&lt;5&gt;
```

This is due to autoboxing. By default all the integers are cast to Integer, but we were expecting long here. Hence the error. In order to overcome this problem, it is better if you type cast the first parameter in the assertEquals to the appropriate return type for the tested method as follows

```
 assertEquals((long)5, new Calc().add(2, 3));
```


Assert methods with JUnit

- **assertTrue() , assertFalse()**
  - Used to test a single variable to see if its value is either true, or false

```
assertTrue (myClass.getBoolean());
assertFalse(myClass.getBoolean());
```

- **assertNull(),assertNotNull()**
  - Used to test a single variable to see if it is null or not null

```
assertNull(myClass.getObject());
assertNotNull(myClass.getObject());
```

- **assertSame() and assertNotSame()**
  - Used to test if two object references point to the same object or not

```
String s1="Hello";
String s2="Hello";
assertSame(s1,s2); ->true
```

14

assertTrue() , assertFalse()

If the getBoolean() method returns true, the assertTrue() method will return normally. Otherwise an exception will be thrown, and the test will stop there.

If the getBoolean() method returns false, the assertFalse() method will return normally. Otherwise an exception will be thrown, and the test will stop there.

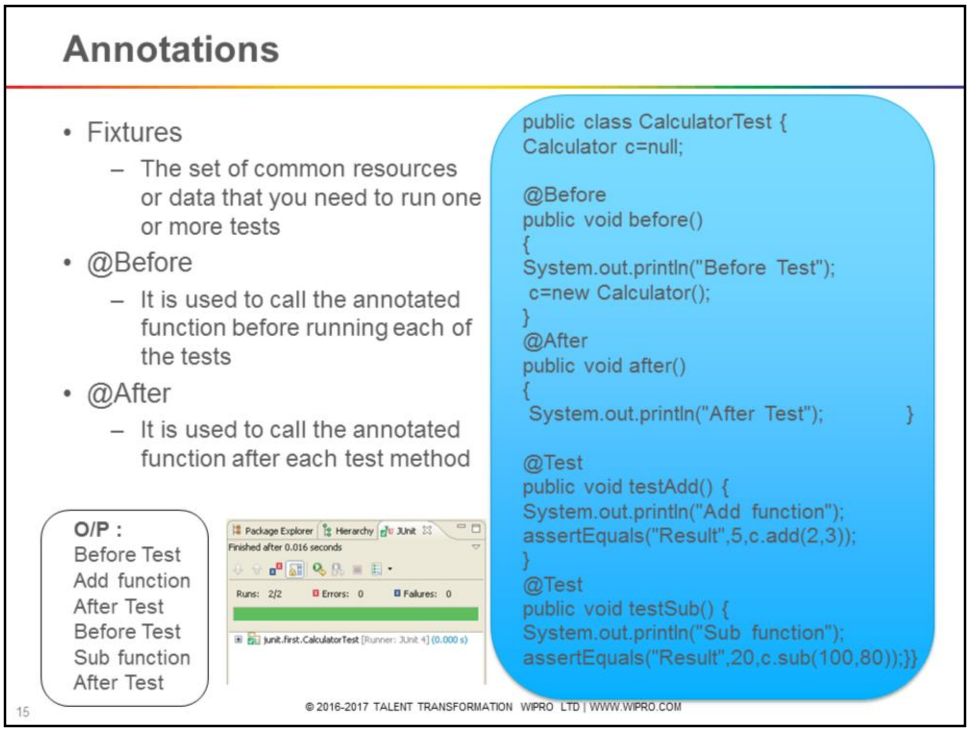
assertNull(),assertNotNull()

If the myClass.getObject() returns null, the assertNull() method will return normally. If a non-null value is returned, the assertNull() method will throw an exception, and the test will be aborted here.

The assertNotNull() method works oppositely of the assertNull() method, throwing an exception if a null value is passed to it, and returning normally if a non-null value is passed to it.

assertSame(),assertNotSame()

The assertSame() and assertNotSame() methods tests if two object references point to the same object or not. It is not enough that the two objects pointed to are equals according to their equals() methods. It must be exactly the same object pointed to.



Annotations

- Fixtures
  - The set of common resources or data that you need to run one or more tests
- @Before
  - It is used to call the annotated function before running each of the tests
- @After
  - It is used to call the annotated function after each test method

```
public class CalculatorTest {
Calculator c=null;

@Before
public void before()
{
System.out.println("Before Test");
 c=new Calculator();
}
@After
public void after()
{
 System.out.println("After Test");           }

@Test
public void testAdd() {
System.out.println("Add function");
assertEquals("Result",5,c.add(2,3));
}
@Test
public void testSub() {
System.out.println("Sub function");
assertEquals("Result",20,c.sub(100,80));}}
```

O/P :

```
Before Test
Add function
After Test
Before Test
Sub function
After Test
```

```
Finished after 0.016 seconds
Runs: 2/2    Errors: 0    Failures: 0
junit.first.CalculatorTest [Runner: JUnit 4] (0.000 s)
```

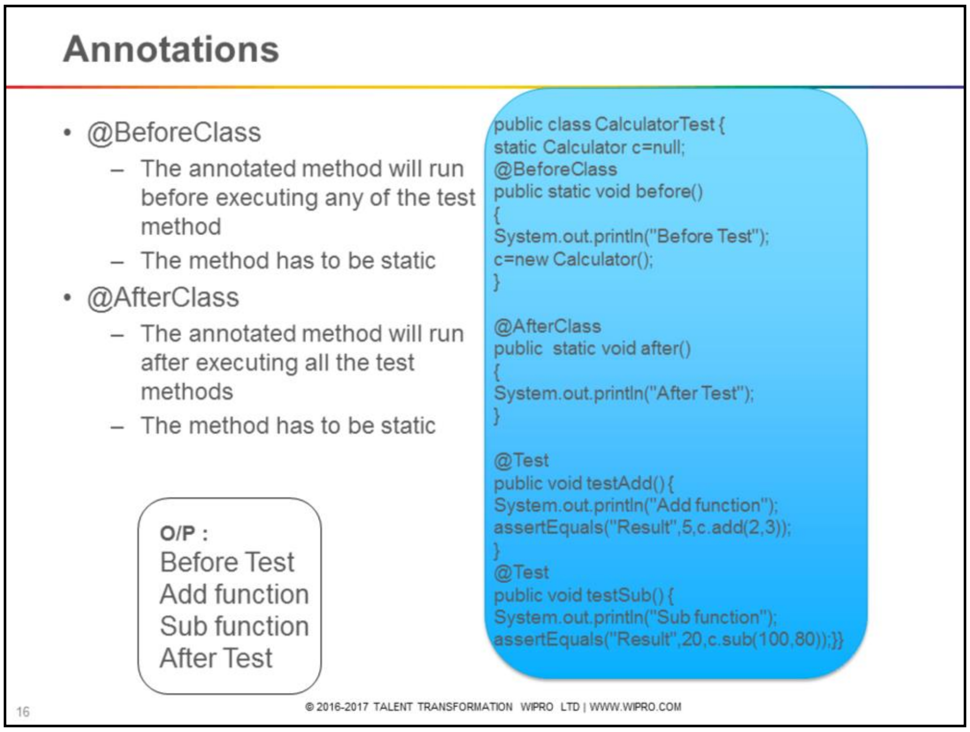
15

© 2016-2017 TALENT TRANSFORMATION WIPRO LTD | WWW.WIPRO.COM

Let's consider the case in which each of the tests that you design needs a common set of objects. One approach can be to create those objects in each of the methods. Alternatively, the JUnit framework provides two special methods, setUp() and tearDown(), to initialize and clean up any common objects. This avoids duplicating the test code necessary to do the common setup and cleanup tasks. These are together referred to as *fixtures*.

The framework calls the setup() before and tearDown() after each test method—thereby ensuring that there are no side effects from one test run to the next.

In Junit 4.x the @Before annotation does the role of the setUp() method and the @After annotation performs the role of the tearDown() method of JUnit 3.x



Annotations

- @BeforeClass
  - The annotated method will run before executing any of the test method
  - The method has to be static
- @AfterClass
  - The annotated method will run after executing all the test methods
  - The method has to be static

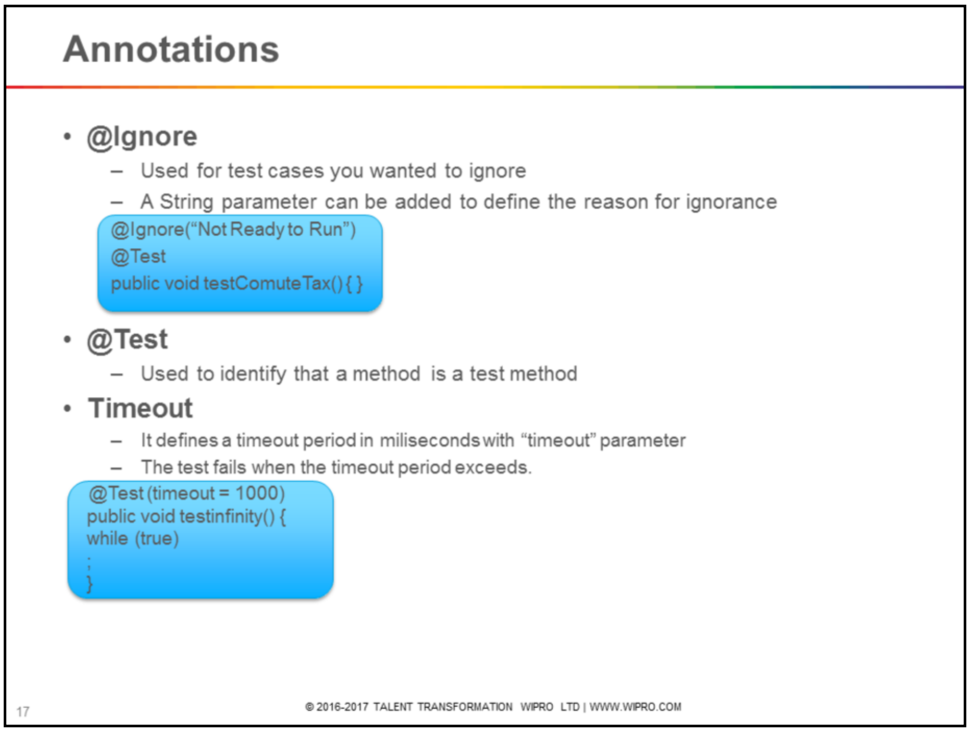
```
public class CalculatorTest {
static Calculator c=null;
@BeforeClass
public static void before()
{
System.out.println("Before Test");
c=new Calculator();
}

@AfterClass
public  static void after()
{
System.out.println("After Test");
}

@Test
public void testAdd() {
System.out.println("Add function");
assertEquals("Result",5,c.add(2,3));
}
@Test
public void testSub() {
System.out.println("Sub function");
assertEquals("Result",20,c.sub(100,80));}}
```

O/P :

```
Before Test
Add function
Sub function
After Test
```

16

@BeforeClass and @AfterClass
Use @BeforeClass and @AfterClass annotations for class wide “setup” and “tearDown” respectively. Think them as one time setup and tearDown. They run for one time before and after all test cases.



Annotations

- @Ignore
  - Used for test cases you wanted to ignore
  - A String parameter can be added to define the reason for ignorance

```
@Ignore("Not Ready to Run")
@Test
public void testComuteTax() { }
```

- @Test
  - Used to identify that a method is a test method
- Timeout
  - It defines a timeout period in miliseconds with "timeout" parameter
  - The test fails when the timeout period exceeds.

```
@Test (timeout = 1000)
public void testinfinity() {
while (true)
;
}
```

17 © 2016-2017 TALENT TRANSFORMATION WIPRO LTD | WWW.WIPRO.COM

The Test annotation supports two optional parameters. The first, expected, declares that a test method should throw an exception. If it doesn't throw an exception or if it throws a different exception than the one declared, the test fails. For example, the following test succeeds:

@Test(**expected=IndexOutOfBoundsException.class**)

public void outOfBounds()

{

new ArrayList<Object>().get(1);

}

The second optional parameter, timeout, causes a test to fail if it takes longer than a specified amount of clock time (measured in milliseconds). The following test fails:

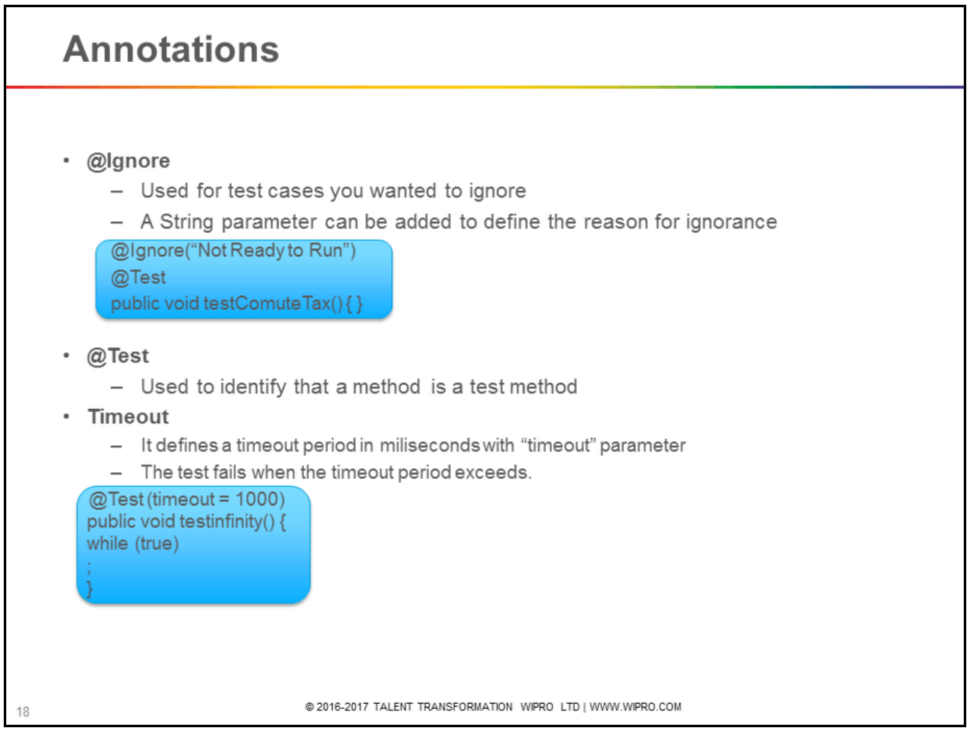
@Test(**timeout=100**)

public void infinity()

{

while(true);

}



Annotations

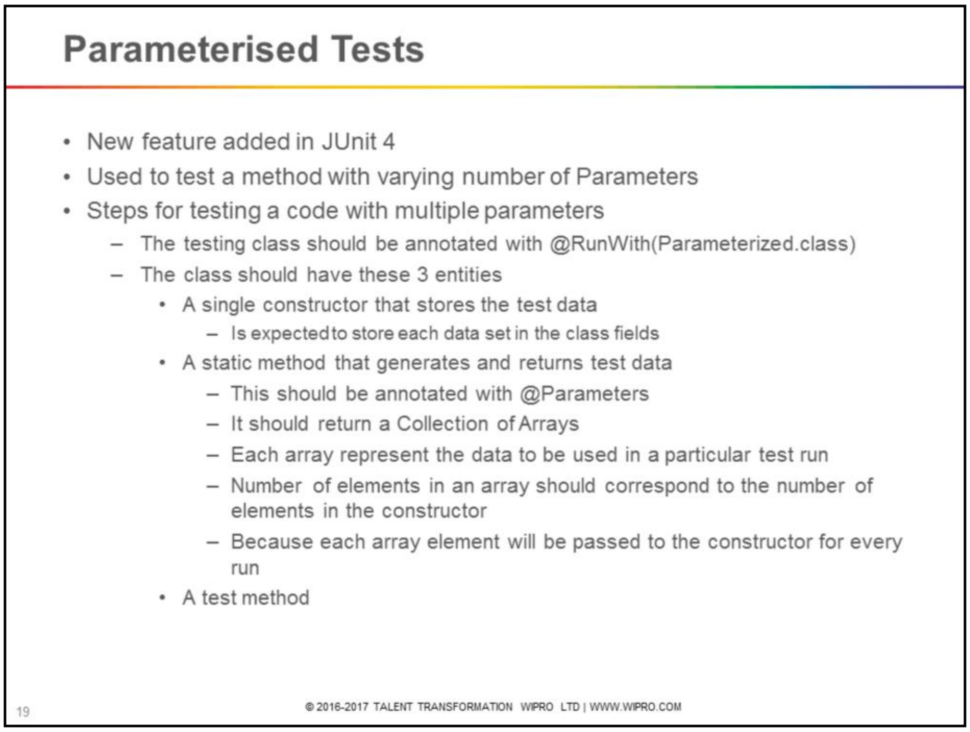
- **@Ignore**
  - Used for test cases you wanted to ignore
  - A String parameter can be added to define the reason for ignorance

```
@Ignore("Not Ready to Run")
@Test
public void testComuteTax() { }
```

- **@Test**
  - Used to identify that a method is a test method
- **Timeout**
  - It defines a timeout period in miliseconds with "timeout" parameter
  - The test fails when the timeout period exceeds.

```
@Test (timeout = 1000)
public void testinfinity() {
while (true)
;
}
```

18



Parameterised Tests

- New feature added in JUnit 4
- Used to test a method with varying number of Parameters
- Steps for testing a code with multiple parameters
  - The testing class should be annotated with @RunWith(Parameterized.class)
  - The class should have these 3 entities
    - A single constructor that stores the test data
      - Is expected to store each data set in the class fields
    - A static method that generates and returns test data
      - This should be annotated with @Parameters
      - It should return a Collection of Arrays
      - Each array represent the data to be used in a particular test run
      - Number of elements in an array should correspond to the number of elements in the constructor
      - Because each array element will be passed to the constructor for every run
    - A test method

19

© 2016-2017 TALENT TRANSFORMATION WIPRO LTD | WWW.WIPRO.COM

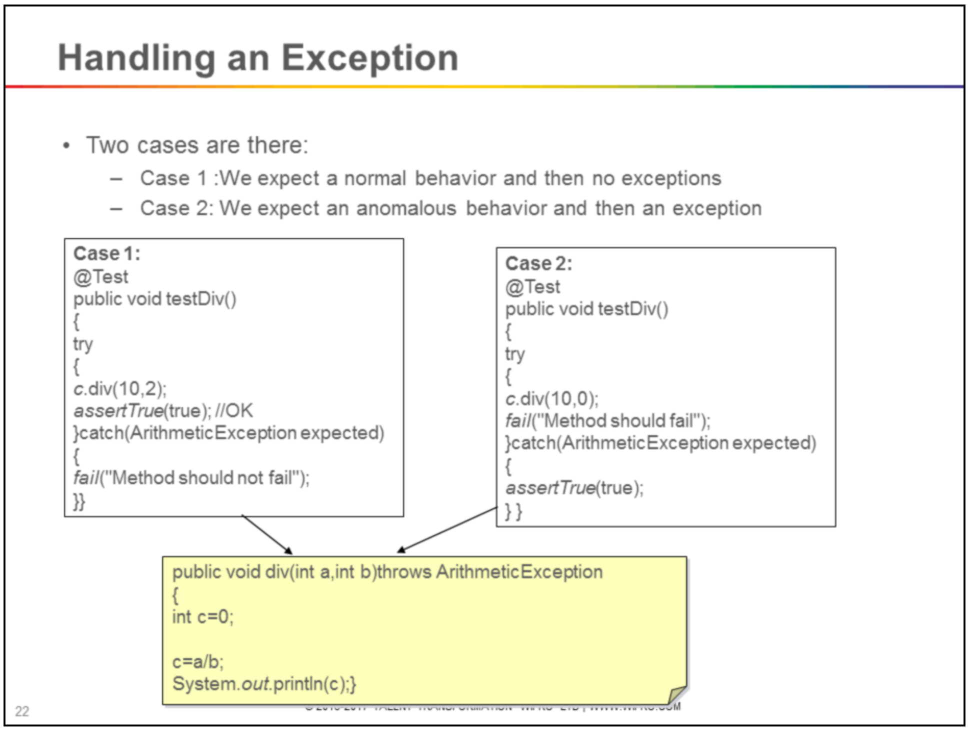
Structure of a parameterized test class

The method that generates test data must be annotated with @Parameters, and it must return a Collection of Arrays. Each array represents the data to be used in a particular test run. The number of elements in each array must correspond to the number of parameters in the class's constructor, because each array element will be passed to the constructor, one at a time as the class is instantiated over and over.

The constructor is simply expected to store each data set in the class's fields, where they can be accessed by the test methods. Note that only a single constructor may be provided. This means that each array provided by the data-generating method must be the same size, and you might have to pad your data sets with nulls if you don't always need a particular value.

Let's put this together. When the test runner is invoked, the data-generating method will be executed, and it will return a Collection of Arrays, where each array is a set of test data. The test runner will then instantiate the class and pass the first set of test data to the constructor. The constructor will store the data in its fields. Then each test method will be executed, and each test method will have access to that first set of test data. After each test method has executed, the object will be instantiated again, this time using the second element in the Collection of Arrays, and so on.

```
package junit.first;
import junit.first.Stringmanip.*;
import java.util.Arrays;
import java.util.Collection;
import org.junit.Test;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import org.junit.runner.RunWith;
import static org.junit.Assert.*;



 @RunWith(Parameterized.class)
 public class StringmanipTest2
 {
   // Fields
  private String datum;
  private String expected;

   /*
   Constructor.
    The JUnit test runner will instantiate this class once for every
    element in the Collection returned by the method annotated with
     @Parameters.
   */
  public StringmanipTest2(String datum, String expected)
  {
    this.datum = datum;
     this.expected = expected;
   }

   /*
     Test data generator.
    This method is called the the JUnit parameterized test runner and
    returns a Collection of Arrays.  For each Array in the Collection,
    each array element corresponds to a parameter in the constructor.
   */
@Parameters
      public static Collection<Object[]> generateData()
     {
        // In this example, the parameter generator returns a List of
        // arrays.  Each array has two elements: { datum, expected }.
      // These data are hard-coded into the class, but they could be
      // generated or loaded in any way you like.
      Object[][] data = new  Object[][]{
               { "Anitha", "ANITHA" },
               { "anitha", "ANITHA" },
               { "ANITHA", "ANITHA" }
            };
       return Arrays.asList(data);
     }


     /*
       The test.
       This test method is run once for each element in the Collection returned
       by the test data generator -- that is, every time this class is
      instantiated. Each time this class is instantiated, it will have a
       different data set, which is available to the test method through the
        instance's fields.
       */
      @Test
      public void testUpperCase()
      {
         Stringmanip s = new Stringmanip(this.datum);
       String actualResult = s.upperCase();
       assertEquals(actualResult, this.expected);
     }
   }
```



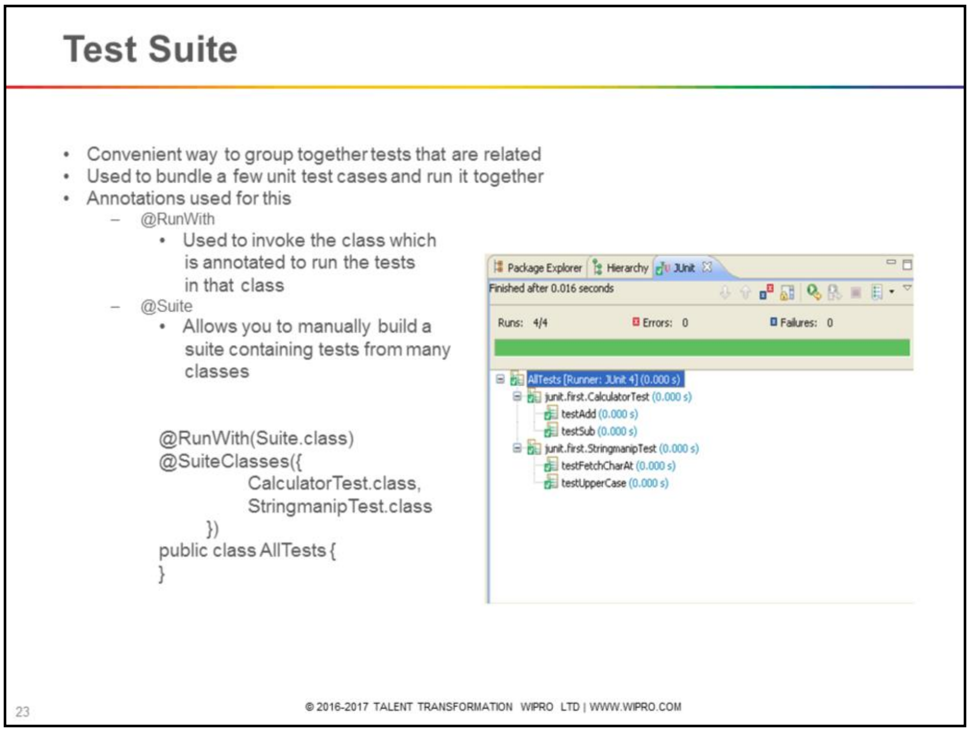
Handling an Exception

- Two cases are there:
  - Case 1 :We expect a normal behavior and then no exceptions
  - Case 2: We expect an anomalous behavior and then an exception

Case 1:
```
@Test
public void testDiv()
{
try
{
c.div(10,2);
assertTrue(true); //OK
}catch(ArithmeticException expected)
{
fail("Method should not fail");
}}
```

Case 2:
```
@Test
public void testDiv()
{
try
{
c.div(10,0);
fail("Method should fail");
}catch(ArithmeticException expected)
{
assertTrue(true);
} }
```

```
public void div(int a,int b)throws ArithmeticException
{
int c=0;

c=a/b;
System.out.println(c);}
```



Test Suite

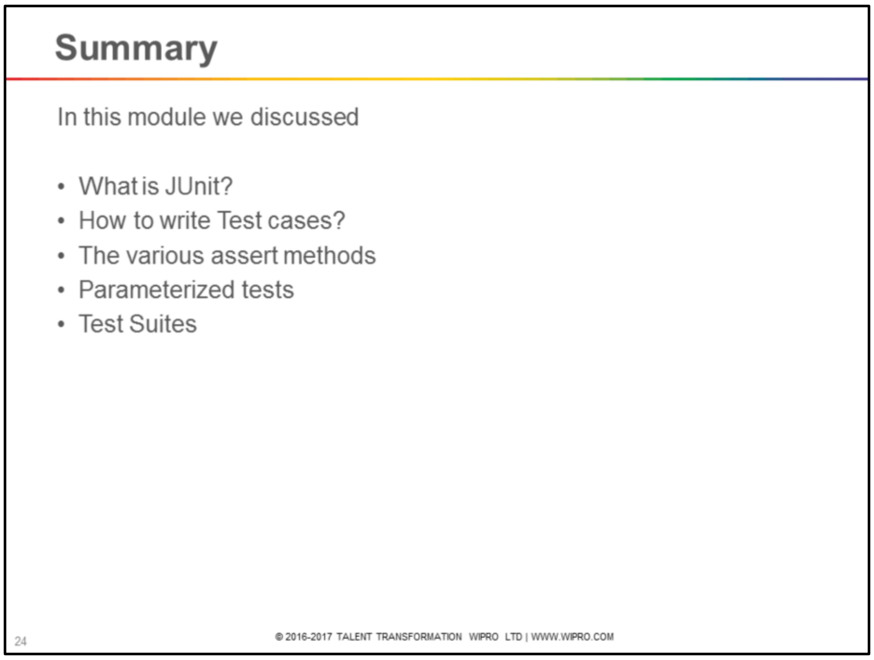
- Convenient way to group together tests that are related
- Used to bundle a few unit test cases and run it together
- Annotations used for this
  - @RunWith
    - Used to invoke the class which is annotated to run the tests in that class
  - @Suite
    - Allows you to manually build a suite containing tests from many classes

```
@RunWith(Suite.class)
@SuiteClasses({
            CalculatorTest.class,
            StringmanipTest.class
    })
public class AllTests {
}
```

```
Finished after 0.016 seconds
Runs: 4/4      Errors: 0      Failures: 0

AllTests [Runner: JUnit 4] (0.000 s)
  junit.first.CalculatorTest (0.000 s)
    testAdd (0.000 s)
    testSub (0.000 s)
  junit.first.StringmanipTest (0.000 s)
    testFetchCharAt (0.000 s)
    testUpperCase (0.000 s)
```

When a class is annotated with @RunWith,JUnit will invoke the class it references to run the tests in that class.
Using Suite as a runner allows you to manually build a suite containing tests from many classes.
Example:-
Calculator.java

```
package junit.first;
public class Calculator {
public int add(int x,int y)
{
return x+y;
}
public int sub(int x,int y)
{
return x-y;
}
}
```

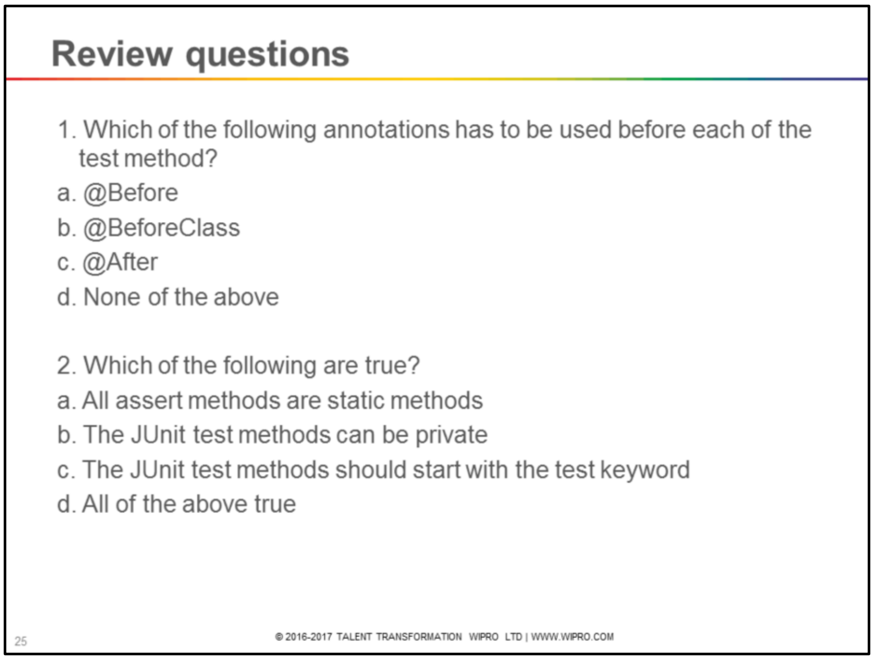


Summary

In this module we discussed

- What is JUnit?
- How to write Test cases?
- The various assert methods
- Parameterized tests
- Test Suites

24



Review questions

1. Which of the following annotations has to be used before each of the test method?

a. @Before

b. @BeforeClass

c. @After

d. None of the above

2. Which of the following are true?

a. All assert methods are static methods

b. The JUnit test methods can be private

c. The JUnit test methods should start with the test keyword

d. All of the above true

Thank you