



JUNIT

Assert methods and Annotations



Agenda (Contd.).



Assert methods and Annotations

Objectives

At the end of this module, you will be able to:

- Work with Assert methods
- Work with Annotations related to JUNIT

Assert methods and Annotations



Assert methods with JUnit

- **assertArrayEquals()**

- Used to test if two arrays are equal to each other

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

Assert methods with JUnit

- **assertArrayEquals()**

Used to test if two arrays are equal to each other. 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**

The `assertEquals()` method can compare any two objects to each other. If the two objects compared are not same, then an `AssertionError` will be thrown.

Assert methods with JUnit

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;  
    }  
}
```

Assert methods with JUnit

```
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:<5> but was:<5>
```

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 (Contd.).

- **assertTrue() , assertFalse()**

- Used to test whether a method returns true or false

```
assertTrue (testClass.isSafe());
```

```
assertFalse (testClass.isSafe());
```

- **assertNull(),assertNotNull()**

- Used to test a variable to see if it is null or not null

```
assertNull (testClass.getObject());
```

```
assertNotNull (testClass.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
```

Assert methods with JUnit (Contd.).

- **assertTrue() , assertFalse()**
 - If the `isSafe()` method returns true, the `assertTrue()` method will return normally. Else an exception will be thrown, and the test will stop there.
 - If the `isSafe()` method returns false, the `assertFalse()` method will return normally. Else an exception will be thrown, and the test will stop there.
- **assertNull(),assertNotNull()**
 - If the `testClass.getObject()` returns null, the `assertNull()` method will return normally, else the `assertNull()` method will throw an exception, and the test will be stopped.
 - The `assertNotNull()` method works oppositely of the `assertNull()` method. It throws an exception if a null value is passed to it, and returns normally if a non-null value is passed to it.
- **assertSame(),assertNotSame()**
 - Used to check if two object references point to the same object or not.

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();
    }

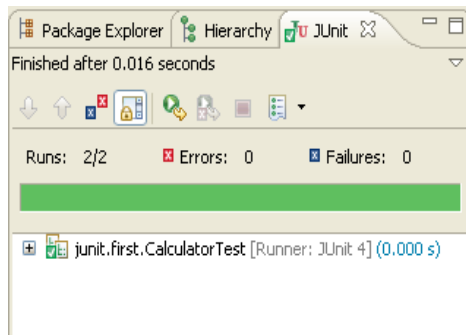
    @After
    public void after()
    {
        System.out.println("After Test");
    }

    @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



Annotations

- 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 (Contd.).

- **@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

O/P :
Before Test
Add function
Sub function
After Test

```
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));
    }
}
```

Annotations (Contd.).

- **@Ignore**

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

```
@Ignore("Not Ready to Run")
```

```
@Test
```

```
public void testComuteTax() { }
```

- **@Test**

- Used to identify that a method is a test method

Annotations (Contd.).

Two optional parameters are supported by Test Annotation.

The first optional parameter 'expected' is used to declare 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 checkOutOfBounds()  
{  
    new ArrayList<String>().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=1000)  
public void infinityCheck()  
{  
    while(true);  
}
```

Annotations (Contd.).

Two optional parameters are supported by Test Annotation.

The first optional parameter 'expected' is used to declare 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 checkOutOfBounds()  
{  
    new ArrayList<String>().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=1000)
```

```
    public void infinityCheck()  
{  
    while(true);  
}
```


Annotations (Contd.).

- **Timeout**

- It defines a timeout period in milliseconds with “timeout” parameter
- The test fails when the timeout period exceeds.

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

Quiz

- **From tester point of view, What is the use of @Ignore annotation?**
- **From tester point of view, What is the use of @Test (timeout = 1000)**



Summary

- Assert methods
- Annotations



Thank You

