ABOUT VINOD KUMAR KASHYAP

Vinod is Sun Certified and love to work in Java and related technologies. Having more than 10 years of experience, he had developed software's including technologies like Java, Hibernate, Struts, Spring, HTML 5, jQuery, CSS, Web Services, MongoDB, AngularJS. He is also a JUG Leader of Chandigarh Java User Group.

JUnit MultiThreaded Test Example

Posted by: Vinod Kumar Kashyap ☐ in junit ☐ February 16th, 2017

In this post we shall show users how to test the multi threaded java application with the help of JUnit. JUnit MultiThread example clears users mind to understand the basic usage of testing the multi threading application.

Users are advised to visit the JUnit Hello World example for basic understanding of the JUnit. For testing the methods by passing value through keyboard, visit JUnit Keyboard Input example.

We will cover the details in the following example.

1. Introduction

Testing multi threaded applications using JUnit is not so difficult as thought by some developers. We will try to understand the way of testing such applications. This is an example, where a

```
counter
```

is to be accessed and updated by many threads simultaneously. JUnit MultiThread example shows very basic usage.

2. Technology Stack

Technologies used in this example are

- Java

- Maven (for dependency management)

3. Project Setup

Tip
You may skip project creation and jump directly to the **beginning of the example** below.
Start creating a Maven project.
Select

```
File -> New -> Maven Project
```

Figure 1: JUnit MultiThread Example Step 1

Clicking on Next button, users are taken to next screen as shown below. Fill in the details as follows.

Figure 2: JUnit MultiThread Example Step 2

We are done with the creation of the Maven project, with the click of Finish button.

4. JUnit MultiThread Example

Now, let's start with the coding part. Start by adding the following lines to the

```
pom.xml
```

.

pom.xml

```
<dependencies>
     <!-- JUnit -->
     <dependency>
          <groupId>junit</groupId>
          <artifactId>junit</artifactId>
          <version>4.12</version>
     </dependency>

     <!-- Concurrent JUnit -->
     <dependency>
          <groupId>com.vmlens</groupId>
          <artifactId>concurrent-junit</artifactId>
          <version>1.0.0</version>
     </dependency>
</dependencies>
```

As users can see, we are using JUnit 4.12 and a library concurrent-junit for testing the JUnit multi thread applications.

4.1 concurrent-junit

Concurrent-junit library helps the users to test the methods for multi threading. It will create threads for testing methods. By default, number of threads created by this library is **4**, but we can set the desired number of threads. This can be achieved by

```
@ThreadCount
```

annotation of concurrent-junit. We will see the use of

annotation later in the example.

4.2 Classes

CountCheck.java

```
01 package junitmultithread;
02
03 import java.util.concurrent.atomic.AtomicInteger;
04
05 public class CountCheck {
06
07     private final AtomicInteger count = new AtomicInteger();
08
09     public void initialize(int number){
10         count.set(number);
11     }
12
13     public void addOne() {
14         count.incrementAndGet();
15     }
16
17     public int getCount() {
18         return count.get();
19     }
20 }
```

If users closely examine, we are using the AtomicInteger for our variable. Since taking variable as an Integer will not help. Increment an Integer is a multi step process which will create race condition.
Methods used in example are explained below.

- `initialize()`

  method initializes the variable.

- `addOne()`

  method increments the variable.

- `getCount()`

  method returns the value of variable.

Next, create a class with JUnit test.

CountCheckTest.java

```
01 package junitmultithread;
02
03 import static org.junit.Assert.assertEquals;
04
05 import org.junit.After;
06 import org.junit.Before;
07 import org.junit.Test;
08 import org.junit.runner.RunWith;
09
10 import com.anarsoft.vmlens.concurrent.junit.ConcurrentTestRunner;
11 import com.anarsoft.vmlens.concurrent.junit.ThreadCount;
12
13 @RunWith(ConcurrentTestRunner.class)
14 public class CountCheckTest {
15
16     private CountCheck counter = new CountCheck();
17
18     @Before
19     public void initialCount(){
20         counter.initialize(2);
21     }
22
23     @Test
24     public void addOne() {
25         counter.addOne();
26     }
27
28     @After
29     public void testCount() {
30         assertEquals("Value should be 6", 6, counter.getCount());
31     }
32 }
```

First of all, let's analyze the code line by line.
Line 13 is using

```
@RunWith(ConcurrentTestRunner.class)
```

annotation, that helps to run the application with threads. As we have previously explained, by default it will create **4** threads.
Line 19 is using method, that will run before all test methods and initialize the counter variable. This example creates **4** threads which calls the

method of

`CheckCount.java`

class.
Line 24 is our main test case.
Line 29 will run after all threads stop executing the threads.

After running the

`CheckCountTest.java`

class, the output will be shown in JUnit window.

Figure 3: JUnit Multi Thread Example Test Result

As a result, test case is passed, because we are testing with the

`assertEquals()`

, which tests for equality.

4.2.1 Use of @ThreadCount

Finally, we will show the usage of

`@ThreadCount`

annotation.

CountCheckThreadCountTest.java

See the highlighted code, which is different from the above code.

```
01 package junitmultithread;
02
03 import static org.junit.Assert.assertEquals;
04
05 import org.junit.After;
06 import org.junit.Before;
07 import org.junit.Test;
08 import org.junit.runner.RunWith;
09
10 import com.anarsoft.vmlens.concurrent.junit.ConcurrentTestRunner;
11 import com.anarsoft.vmlens.concurrent.junit.ThreadCount;
12
13 @RunWith(ConcurrentTestRunner.class)
14 public class CountCheckThreadCountTest {
15
```