JUnit Programming Cookbook

Hot Recipes for the JUnit Framework

JAVA CODE GEEKS

Java Code Geeks
Java 2 Java Developers Resource Center

JUnit Programming Cookbook

Contents

Preface

JUnit is a unit testing framework to write repeatable tests. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks which is collectively known as xUnit that originated with SUnit. A research survey performed in 2013 across 10,000 Java projects hosted on GitHub found that JUnit, (in a tie with slf4j-api), was the most commonly included external library. (Source: https://en.wikipedia.org/wiki/JUnit)

In this ebook, we provide a compilation of JUnit tutorials that will help you kick-start your own programming projects. We cover a wide range of topics, from basic usage and configuration, to multithreaded tests and integration with other testing frameworks. With our straightforward tutorials, you will be able to get your own projects up and running in minimum time.

About the Author

JCGs (Java Code Geeks) is an independent online community focused on creating the ultimate Java to Java developers resource center; targeted at the technical architect, technical team lead (senior developer), project manager and junior developers alike.

JCGs serve the Java, SOA, Agile and Telecom communities with daily news written by domain experts, articles, tutorials, reviews, announcements, code snippets and open source projects.

You can find them online at https://www.javacodegeeks.com/

Chapter 1

Hello World Example

In this example we shall show you how to start with JUnit hello world. JUnit is an open-source testing framework used by Java programmers. It contains various methods to include in class to make your test cases run smoothly.

Currently latest stable version is 4.x and 5.x is coming most probably in Q1 of 2017. JUnit contains many annotations that are used while creating test cases.

- **@BeforeClass**: It is used to write code that we want to run before all test cases.
- **@Before**: It will run before every test case.
- **@Test**: This is actual test case.
- **@After**: It will run after every test case.
- **@AfterClass**: It is used to write code that we want to run after all test cases.

For the sake of simplicity of the example we are using the Maven so that you don't need to include the jar yourself. Maven is dependency management tool for Java. The jar and its dependencies would be automatically pulled by Maven.

Tools/technologies needed: Eclipse Maven Java JUnit 4.12 (pulled by Maven automatically)

With this example we will try to show the basic usage of the JUnit. Let's start with the creation of project in Eclipse.

1.1 Setup JUnit Hello World Project

First you need to Select File → New → Maven Project

You will see the below screen. Select the top most checkbox as we need simple maven project.

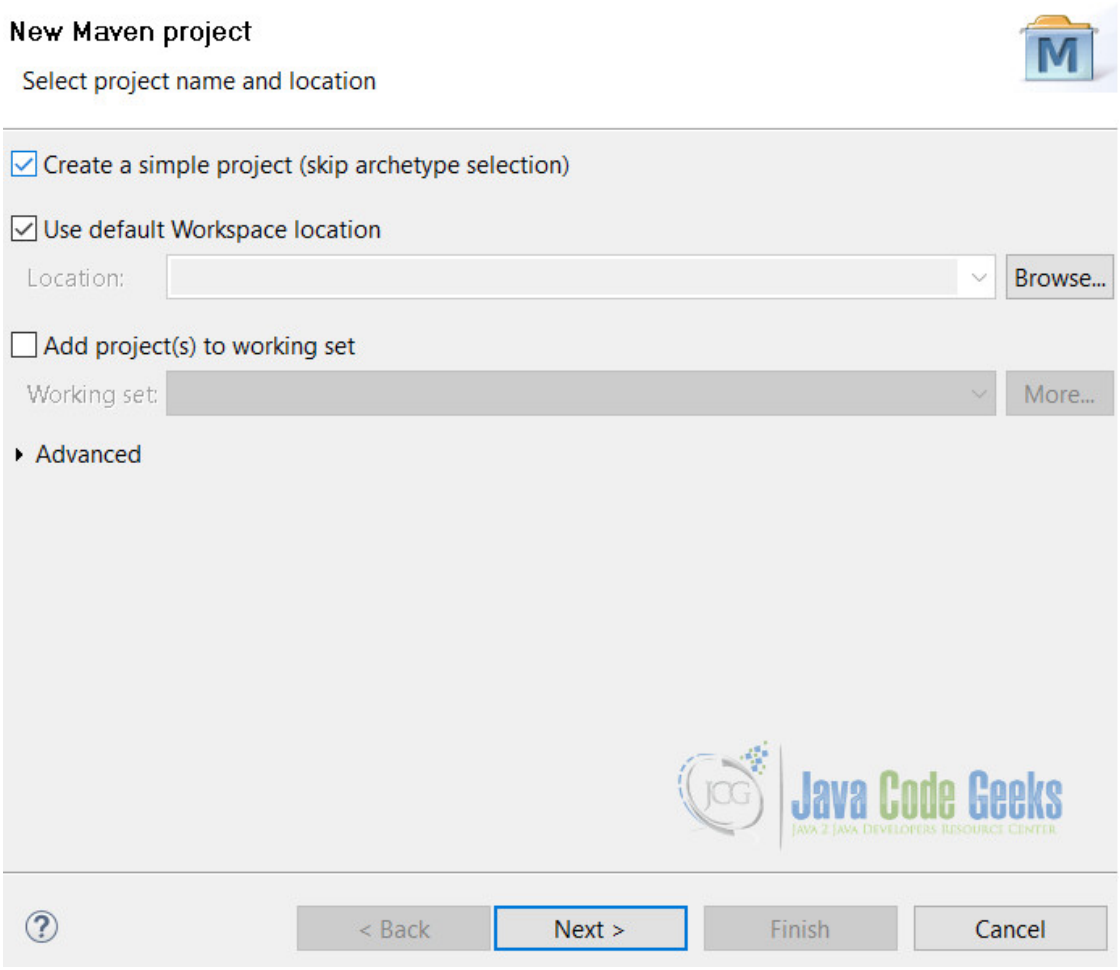

Figure 1.1: First page for maven project

Click on Next button which will take you to second screen. Fill the required details as described in below:

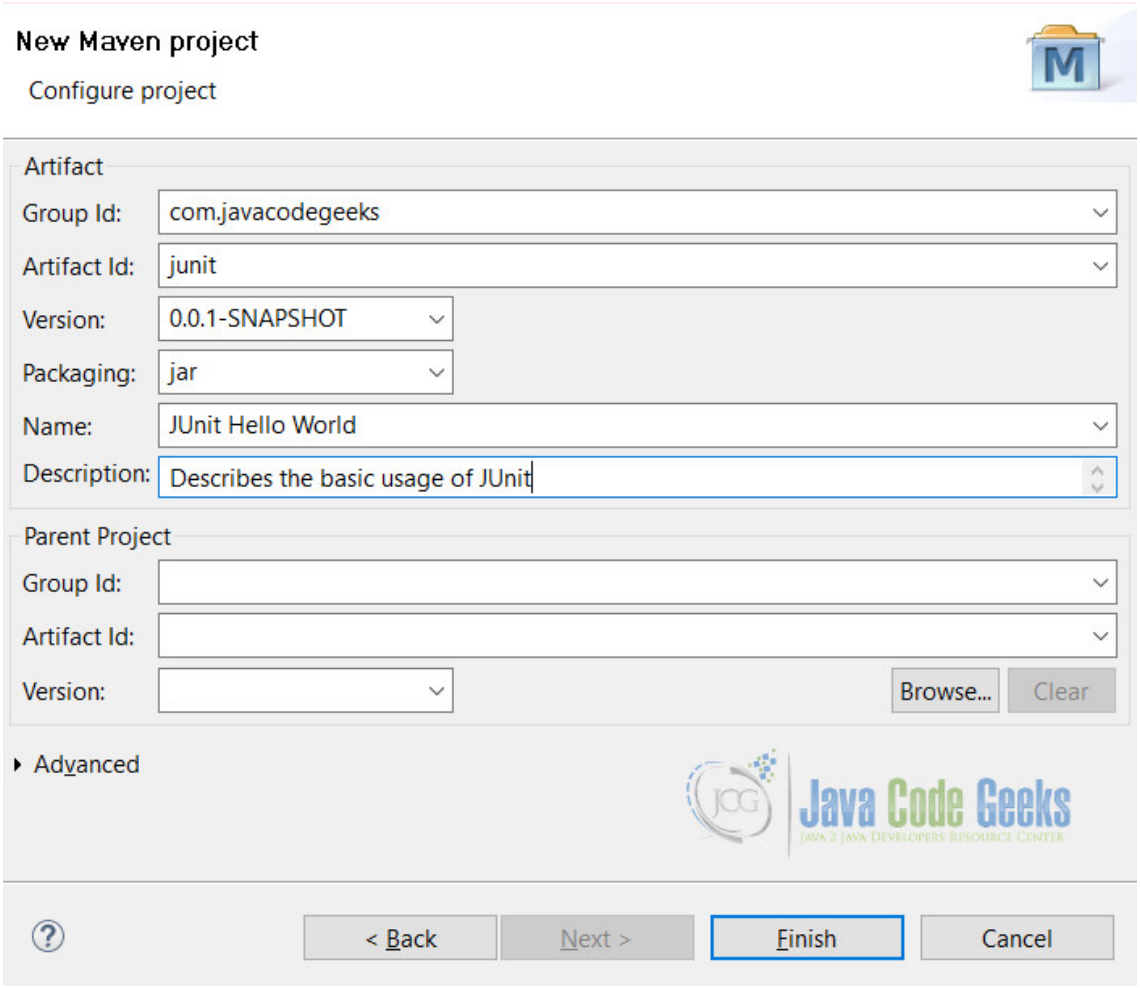

Figure 1.2: Maven project configurations

Click on finish. Now you are ready for your project. Open `pom.xml` and copy dependencies to it.

pom.xml

```
<dependencies>
 <dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
 </dependency>
</dependencies>
```

1.2 Java Classes

Let's create class which contains one method.

JUnitHelloWorld.java

```
package junit;

public class JUnitHelloWorld {
```

```
        public boolean isGreater(int num1, int num2){
                return num1 > num2;
        }
}
```

In this class we have a method named `isGreater()` which tells us that if first number is greater than second number or not. It will return `true` or `false` depending on the parameters passed.

JUnitHelloWorldTest.java

```
package junit;

import static org.junit.Assert.assertTrue;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class JUnitHelloWorldTest {

        @BeforeClass
        public static void beforeClass() {
                System.out.println("Before Class");
        }

        @Before
        public void before() {
                System.out.println("Before Test Case");
        }

        @Test
        public void isGreaterTest() {
                System.out.println("Test");
                JUnitHelloWorld helloWorld = new JUnitHelloWorld();
                assertTrue("Num 1 is greater than Num 2", helloWorld.isGreater(4, 3));
        }

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

        @AfterClass
        public static void afterClass() {
                System.out.println("After Class");
        }
}
```

In this class we can see there are five methods. Most important is the `@Test` method, which is our main test case. Other methods are optional and may or may not be used.

1.3 Run JUnit Project

Right click on `JUnitHelloWorldTest` and Run As → JUnit Test. We will see the following output:

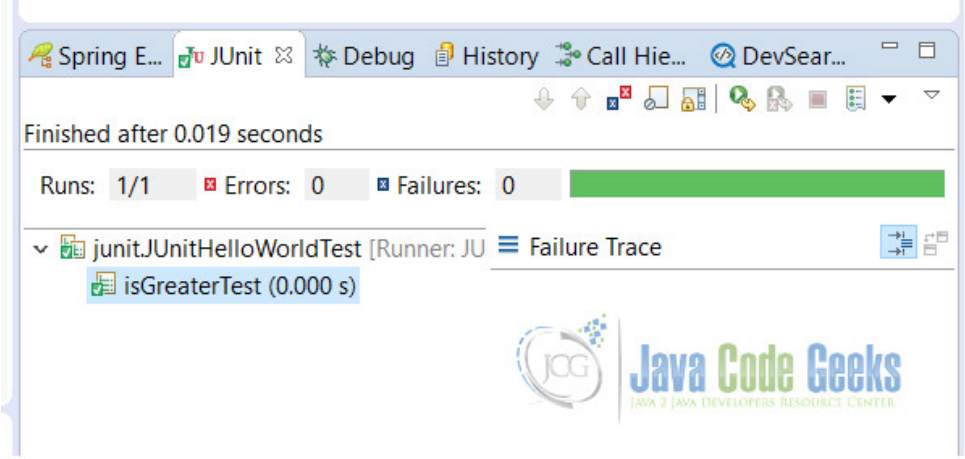

Figure 1.3: JUnit Test Case Passed

And also in the output window here's what we should see

Figure 1.4: Output of JUnit

The result shown in image JUnit Test Case Passed is due to the test case passed. We can see the `line number 27` from `JUnitHelloWorldTest` class that 4 is greater than 3.

```
assertTrue("Num 1 is greater than Num 2", helloWorld.isGreater(4, 3));
```

We can also notice the output in console which shows us the method calls. We can see how all annotations work and how the priority of methods is called.

Now change the parameters to 2 and 3.

```
assertTrue("Num 1 is greater than Num 2", helloWorld.isGreater(2, 3));
```

When you run the above code it will generate the below error

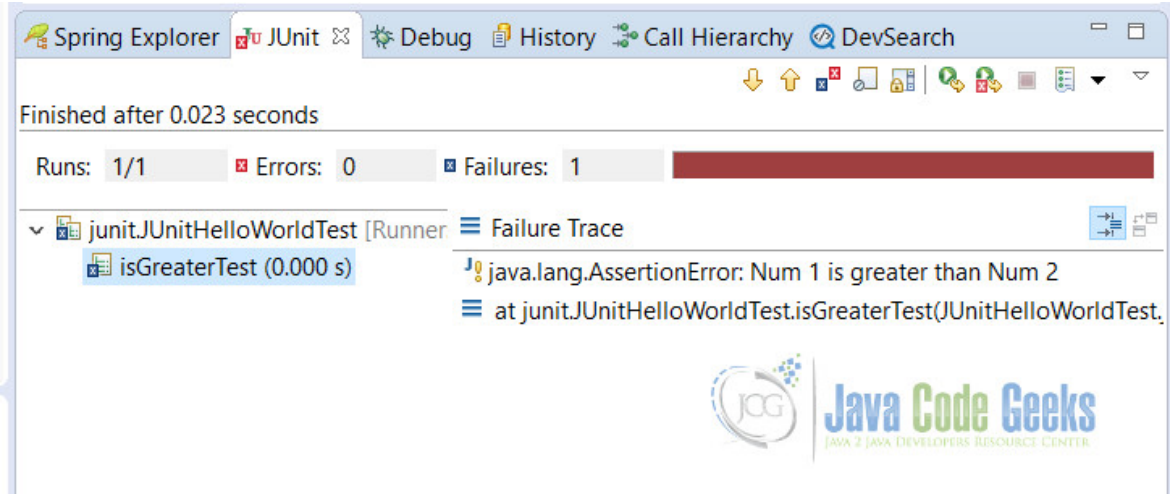

Figure 1.5: JUnit Test Case Failed

From above it shows that our test case fails because we are expecting the result to evaluate to `true` but we are getting `false`

1.4 Download the Eclipse Project

This was an example of JUnit Hello World.

Download

You can download the full source code of this example here: **JUnitHelloWorld**

Chapter 2

JUnit Tutorial for Beginners

2.1 Introduction

In this post, we will discus the basics of setting up your JUnit Test cases. We'll go step by step in creating test cases as we go along with creating our application. Before we dive into it though, why do we even need to create test cases? Isn't it enough to just create the implementation since it's what we are delivering anyway?

Although the actual implementation is part of the package, the JUnit Test case is a **bullet proof evidence** that what we wrote is what the actual requirements or functions will do. It is the concrete basis of a specific unit/function which does what it needs to do.

Knowing the impact in the stability of the application. The JUnit Test cases defines the stability of an application even after several extensions to it. If done correctly, it guarantees that the extension made to the system will not break the entire system as a whole. How does it prevent it? If the developers write clean unit and integration tests, it will report any side effects via the reporting plugins that the application uses.

Regression and Integration Testing. The effort of testing is relative to the applications size and changes done. By creating JUnit Test cases, regression and integration tests can be automated and can definitely save time and effort.

Overall, creating JUnit Test cases are definitely a must do by all developers, sadly there are still who don't uses it's power to it's full extent and some just doesn't do it. It's sometimes a shame to think that one of the purest way of developing bullet proof code is not done by the developers. It can be because of the lack of training, experience or just pressure of not delivering the actual value (which is a problem within itself since although not part of the implementation, it's a most valuable component of your code) but thats not an excuse especially that software are now globally taking over most of the major systems (medical, auto, planes, buildings) in the world. The stability of these systems rely on the stability of the unit test cases.

So as a precursor to being a skilled full blown developer that loves to do unit test, let's dive into some of the beginners guide into doing it.

2.2 Tools

For this example, I'll be using Java as the platform, Eclipse as the IDE, and Maven as the project management tool. If you're not yet familiar with these tools, please visit the Java, Eclipse IDE and Maven site.

2.3 Step by Step Guide

2.3.1 Create your project

Let's create a project first.

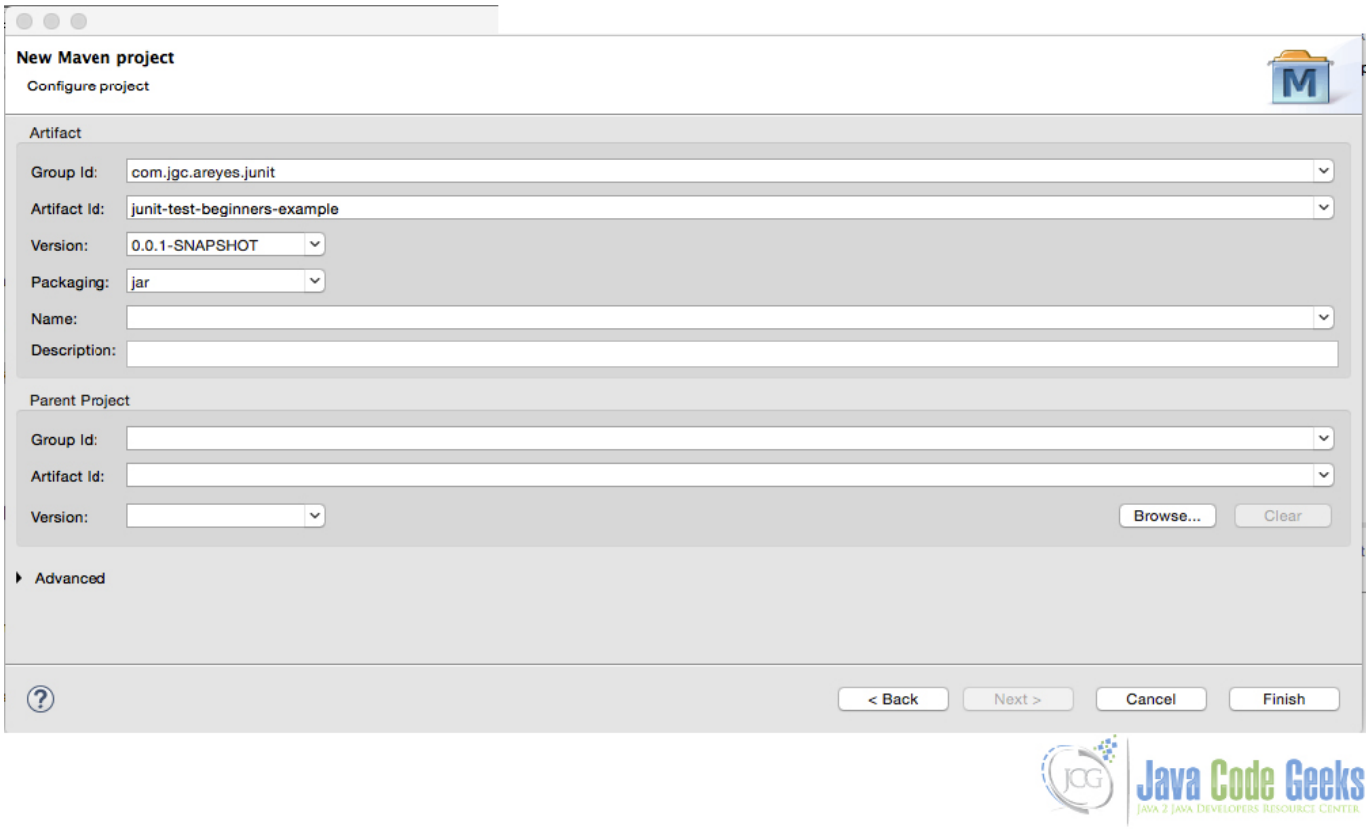

Figure 2.1: New Maven Project

After creating the project, you'll be shown a project like the one below:

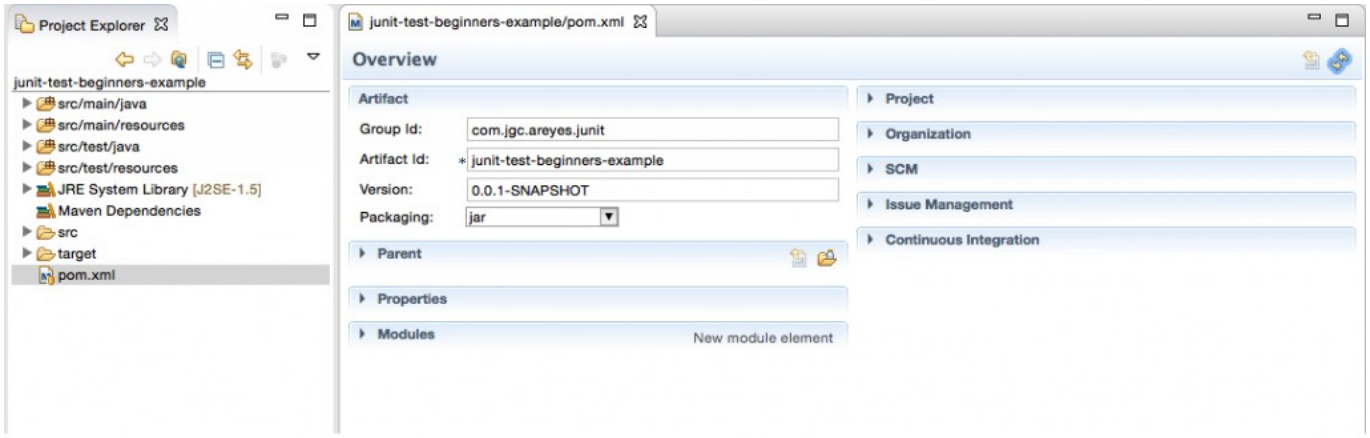

Figure 2.2: New Maven Project .xml

Make sure you include the Junit Library to your dependency list.

pom.xml

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↩
    XMLSchema-instance" xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven ↩
    .apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.jgc.areyes.junit</groupId>
  <artifactId>junit-test-beginners-example</artifactId>
  <version>0.0.1-SNAPSHOT</version>


  <dependencies>
                <dependency>
                        <groupId>junit</groupId>
                        <artifactId>junit</artifactId>
                        <version>4.12</version>
                        <scope>test</scope>
                </dependency>

        </dependencies>


</project>
```

2.3.2 Create the Service Class

Majority of the developers I know starts first with creating the implementation rather than the JUnit Test case, it isn't a bad practice at all but wouldn't it be more concrete if we create the JUnit Test cases first based on the design then create the implementation to pass all JUnit Test case? This is the case in the TDD, one of the most successful schemes of actual software development.

Assuming we are creating a service to manage an account. We need to introduce the following service methods:

- Create a new Account
- Update an Account
- Remove an Account
- List All Account Transactions

We have an OOP design that will handle this service and so we introduce the following classes (Class Diagram).

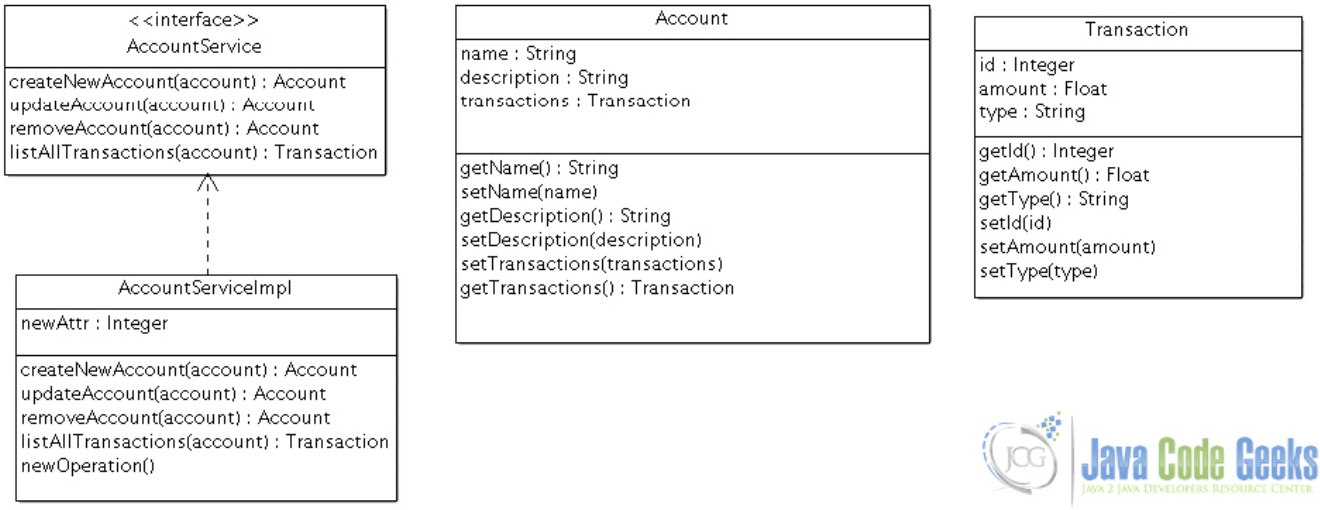

Figure 2.3: Class Diagram

Here is the actual class that doesn't have any implementation yet. We will create the implementation after creating the test cases for this class.

AccountServiceImpl.java

```
package com.areyes1.jgc.svc;

import java.util.List;

import com.areyes1.jgc.intf.AccountService;
import com.areyes1.jgc.obj.Account;
import com.areyes1.jgc.obj.Transaction;

public class AccountServiceImpl implements AccountService {
        public Account createNewAccount(Account account) {
                // TODO Auto-generated method stub
                return null;
        }

        public Account updateAccount(Account account) {
                // TODO Auto-generated method stub
                return null;
        }

        public Account removeAccount(Account account) {
                // TODO Auto-generated method stub
                return null;
        }

        public List listAllTransactions(Account account) {
                // TODO Auto-generated method stub
                return null;
        }
}
```

2.3.3 Create JUnit Test Cases

Now that we have the service placeholder, let's create the Junit Test case for the AccountServiceImpl class. The Test cases will be the basis of your class functional aspect so you as a developer should write a solid and good test case (and not just fake it to pass).

When you create a test case, it will initially look like this:

AccountServiceImplTest.java

```
/**
 *
 */
package com.areyes1.jgc.svc;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

/**
 * @author alvinreyes
 *
 */
public class AccountServiceImplTest {

        /**
         * Test method for {@link com.areyes1.jgc.svc.AccountServiceImpl#createNewAccount( ↩
            com.areyes1.jgc.obj.Account)}.
         */
        @Test
        public void testCreateNewAccount() {
                fail("Not yet implemented");
        }

        /**
         * Test method for {@link com.areyes1.jgc.svc.AccountServiceImpl#updateAccount(com. ↩
            areyes1.jgc.obj.Account)}.
         */
        @Test
        public void testUpdateAccount() {
                fail("Not yet implemented");
        }

        /**
         * Test method for {@link com.areyes1.jgc.svc.AccountServiceImpl#removeAccount(com. ↩
            areyes1.jgc.obj.Account)}.
         */
        @Test
        public void testRemoveAccount() {
                fail("Not yet implemented");
        }

        /**
         * Test method for {@link com.areyes1.jgc.svc.AccountServiceImpl# ↩
            listAllTransactions(com.areyes1.jgc.obj.Account)}.
         */
        @Test
        public void testListAllTransactions() {
                fail("Not yet implemented");
        }
```

```
}
```

As it can be seen above, it will fail the test case if it's not yet implemented. From here on now, it's a matter of discipline from the developer to create concrete and solid test cases.

Observing the JUnit Test cases.

- Using @Test to define a test method
- Put them all on the Test package (src/test/main/)
- Class always has a suffix of Test (AccountServiceImplTest)
- Methods always starts with "test".

Here's is the modified version of the JUnit Test cases. This will now be the basis of our implementation code.

AccountServiceImplTest.java

```
/**
 *
 */
package com.areyes1.jgc.svc;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.hamcrest.CoreMatchers.*;
import com.areyes1.jgc.obj.Account;
/**
 * @author alvinreyes
 *
 */
public class AccountServiceImplTest {

        AccountServiceImpl accountService = new AccountServiceImpl();

        /**
         * Test method for {@link com.areyes1.jgc.svc.AccountServiceImpl#createNewAccount( ↩
             com.areyes1.jgc.obj.Account)}.
         */
        @Test
        public void testCreateNewAccount() {
                Account newAccount = new Account();
                newAccount.setName("Alvin Reyes");
                newAccount.setDescription("This is the description");

                Account newAcccountInserted = accountService.createNewAccount(newAccount);

                //      Check if the account has the same composition.
                assertThat(newAccount, isA(Account.class));
                assertEquals(newAccount.getName(), newAcccountInserted.getName());


        }

        /**
         * Test method for {@link com.areyes1.jgc.svc.AccountServiceImpl#updateAccount(com. ↩
             areyes1.jgc.obj.Account)}.
         */
        @Test
```

```
        public void testUpdateAccount() {


                //      The old account (assumed that this came from a database or mock)
                Account oldAccount = new Account();
                oldAccount.setName("Alvin Reyes");
                oldAccount.setDescription("This is the description");

                String name = oldAccount.getName();
                //      Check if the account is still the same. it is expected to be ↩
                    different since we updated it.
                Account expectedAccountObj =  new Account();
                expectedAccountObj = accountService.updateAccount(oldAccount);
                assertThat(expectedAccountObj, isA(Account.class));
                assertNotEquals(name, expectedAccountObj.getName());

        }

        /**
         * Test method for {@link com.areyes1.jgc.svc.AccountServiceImpl#removeAccount(com. ↩
            areyes1.jgc.obj.Account)}.
         */
        @Test
        public void testRemoveAccount() {

                //      Set up the account to be removed.
                Account toBeRemovedAccount = new Account();
                toBeRemovedAccount.setName("Alvin Reyes");
                toBeRemovedAccount.setDescription("This is the description");

                //      Removed the account.
                assertTrue(accountService.removeAccount(toBeRemovedAccount));

        }

        /**
         * Test method for {@link com.areyes1.jgc.svc.AccountServiceImpl# ↩
            listAllTransactions(com.areyes1.jgc.obj.Account)}.
         */
        @Test
        public void testListAllTransactions() {

                //       Dummy Transactions (can be mocked via mockito)
                Account account = new Account();
                account.setName("Alvin Reyes");

                //      Service gets all transaction
                accountService.listAllTransactions(account);

                //      Check if there are transactions.
                assertTrue(accountService.listAllTransactions(account).size() > 1);


        }

}
```

Running the test case will show the following result.

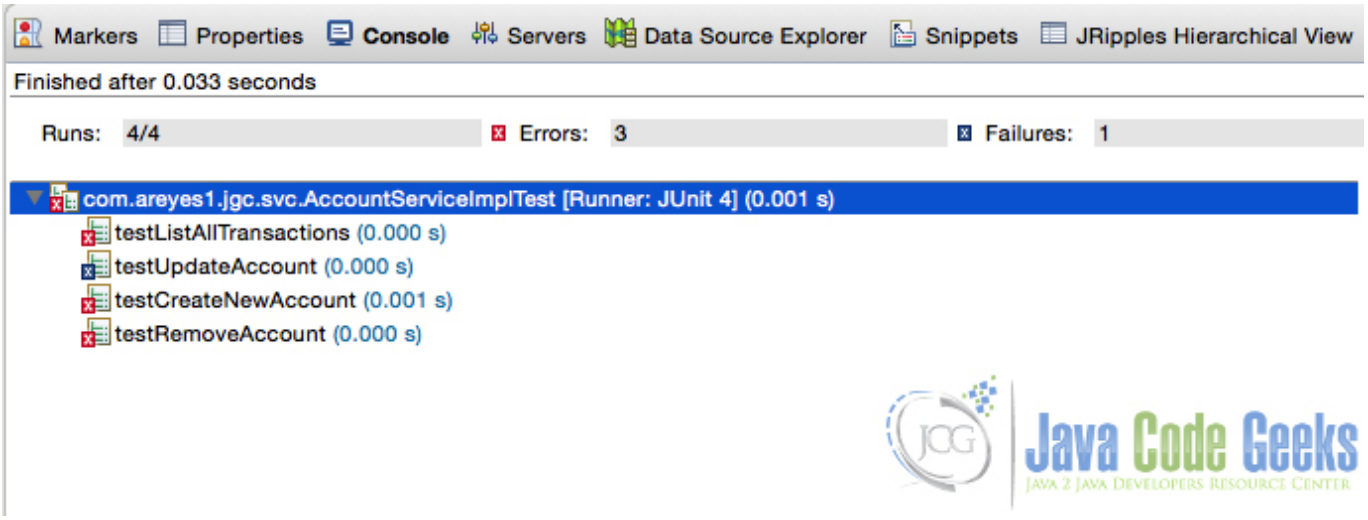

Figure 2.4: Failed Test cases

It failed cause we still have to code our implementation. Now in the implementation of our logic, our goal is make sure that these test cases succeeds!

2.3.4 Code the implementation

Now that the test cases are setup, we can now code our implementation. The test cases we created above will be the basis of how we will create the implementation. The goal is to pass the test cases!

```
package com.areyes1.jgc.svc;

import java.util.ArrayList;
import java.util.List;

import com.areyes1.jgc.intf.AccountService;
import com.areyes1.jgc.obj.Account;
import com.areyes1.jgc.obj.Transaction;

public class AccountServiceImpl implements AccountService {
        public Account createNewAccount(Account account) {
                //      Dummy Dao! Database insert here.
                //      accountDao.insert(account);
                //      Ultimately return the account with the modification.
                return account;

        }

        public Account updateAccount(Account account) {
                //      Dummy Dao! Database insert here.
                //      accountDao.update(account);
                //      Ultimately return the account with the modification.
                account.setName("Alvin Reyes: New Name");
                return account;
        }

        public boolean removeAccount(Account account) {
                //      Dummy Dao! Database insert here.
                //      accountDao.delete(account);
                //      Ultimately return the account with the modification.
```

```
                //      if exception occurs, return false.
                return true;
        }

        public List listAllTransactions(Account account) {
                // accountDao.loadAllTransactions(account);
                List listOfAllTransactions = new ArrayList();
                listOfAllTransactions.add(new Transaction());
                listOfAllTransactions.add(new Transaction());
                listOfAllTransactions.add(new Transaction());
                account.setTransactions(listOfAllTransactions);

                return listOfAllTransactions;
        }
}
```

Running the test case will show the following result.

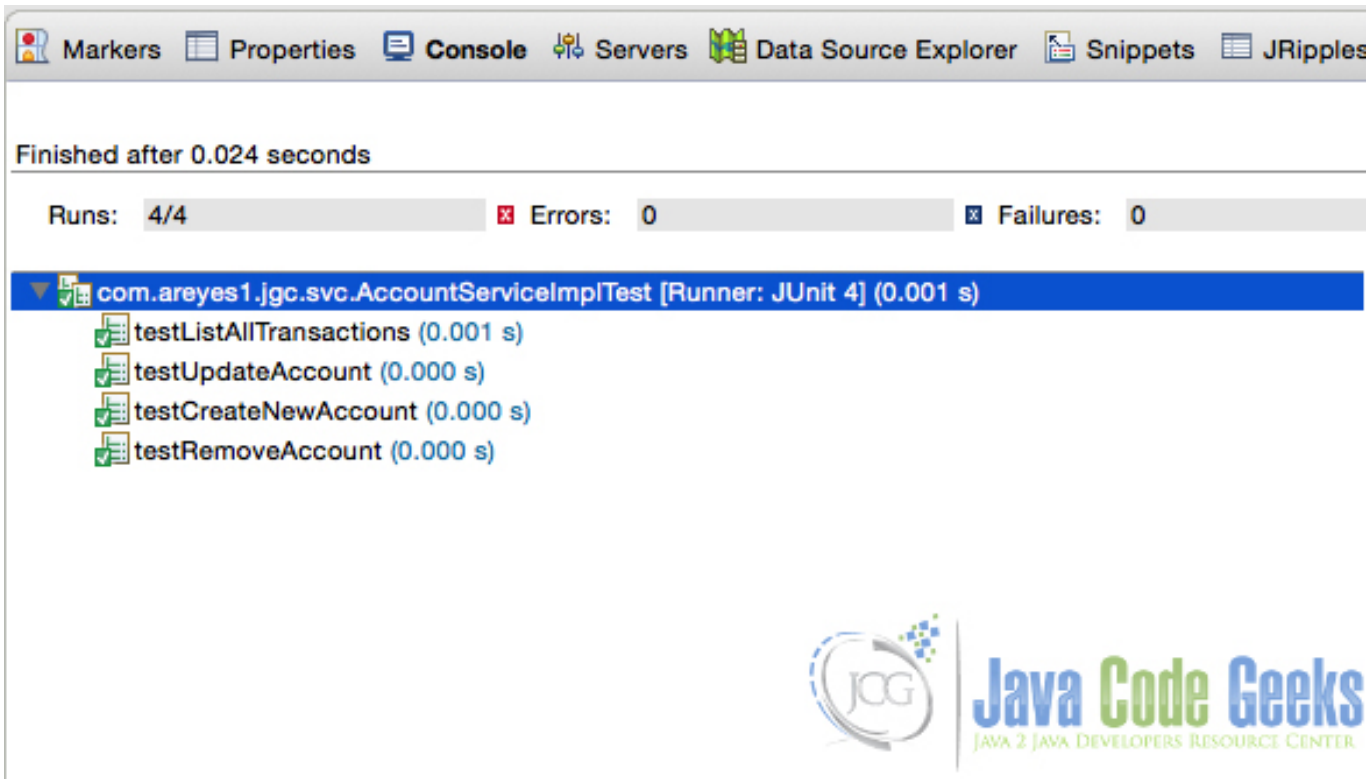

Figure 2.5: Passed Test case

2.3.5 Run your maven

Run your maven to see results.

```
 T E S T S
Running com.areyes1.jgc.svc.AccountServiceImplTest
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.051 sec

Results :

Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
```

```
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ junit-test-beginners-example ---
[INFO] Building jar: /Users/alvinreyes/EclipseProjects/Java/junit-test-beginners-example/ ↩
    target/junit-test-beginners-example-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ junit-test-beginners- ↩
    example ---
[INFO] Installing /Users/alvinreyes/EclipseProjects/Java/junit-test-beginners-example/ ↩
    target/junit-test-beginners-example-0.0.1-SNAPSHOT.jar to /Users/alvinreyes/.m2/ ↩
    repository/com/jgc/areyes/junit/junit-test-beginners-example/0.0.1-SNAPSHOT/junit-test- ↩
    beginners-example-0.0.1-SNAPSHOT.jar
[INFO] Installing /Users/alvinreyes/EclipseProjects/Java/junit-test-beginners-example/pom. ↩
    xml to /Users/alvinreyes/.m2/repository/com/jgc/areyes/junit/junit-test-beginners- ↩
    example/0.0.1-SNAPSHOT/junit-test-beginners-example-0.0.1-SNAPSHOT.pom
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ junit-test-beginners- ↩
    example ---
[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build  ↩
    is platform dependent!
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:compile (default-compile) @ junit-test-beginners- ↩
    example ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ junit-test- ↩
    beginners-example ---
[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build  ↩
    is platform dependent!
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:testCompile (default-testCompile) @ junit-test- ↩
    beginners-example ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ junit-test-beginners-example  ↩
    ---
[INFO] Skipping execution of surefire because it has already been run for this  ↩
    configuration
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 9.456 s
[INFO] Finished at: 2015-10-12T16:28:01-05:00
[INFO] Final Memory: 11M/28M
[INFO] ------------------------------------------------------------------------
```

2.4 Download the Eclipse project

This was an example of JUnit Test Beginners Tutorial

Download

You can download the full source code of this example here: **junit-test-beginners-example**

Chapter 3

Categories Example

3.1 Introduction

JUnit has an awesome feature of organizing group of test cases called Categorizing. It can help developers differentiate test cases from one another. In this post, I'll showcase how easy it is to categorize unit tests by `@Category`.

3.2 Maven Project and Configuration

pom.xml

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↩
    XMLSchema-instance"
        xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd ↩
            /maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>com.jgc.areyes.junit</groupId>
        <artifactId>junit-categories-example</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <build>
                <plugins>
                        <plugin>
                                <groupId>org.apache.maven.plugins</groupId>
                                <artifactId>maven-surefire-plugin</artifactId>
                                <version>2.17</version>
                                <dependencies>
                                        <dependency>
                                                <groupId>org.apache.maven.surefire</groupId ↩
                                                    >
                                                <artifactId>surefire-junit47</artifactId>
                                                <version>2.17</version>
                                        </dependency>
                                </dependencies>
                        </plugin>
                </plugins>
        </build>
        <dependencies>
                <dependency>
                        <groupId>junit</groupId>
                        <artifactId>junit</artifactId>
                        <version>4.12</version>
                        <scope>test</scope>
                </dependency>
```

```
        </dependencies>
</project>
```

3.3 Source code sample

Define the interfaces first. In order for us to group test cases, we need to create a unifier/union on them. We use `interface` class to tag a specific class or method to a group. Here are the interfaces that we will use in this example.

Interfaces

```
public interface FunctionalGroupTests1 {}
public interface FunctionalGroupTests2 {}
public interface IntegrationTests {}
public interface SanityTests {}
```

We then use those interfaces on our test cases. This will differentiate the test case for our own test purposes. In the example below, we tag tests method by category using the `@Category` annotation

JUnitTestCategoryExample.java

```
package com.areyes.junit.svc;

import static org.hamcrest.CoreMatchers.isA;
import static org.junit.Assert.*;

import org.junit.Assert;
import org.junit.Test;
import org.junit.experimental.categories.Category;

import com.areyes.junit.cat.intf.FunctionalGroupTests1;
import com.areyes.junit.cat.intf.FunctionalGroupTests2;
import com.areyes.junit.cat.intf.IntegrationTests;
import com.areyes.junit.cat.intf.SanityTests;

public class JUnitTestCategoryExample {

        @Test @Category(FunctionalGroupTests1.class)
        public void testFunctionalTests1Test1() {
                //      You're test case here: Below is just an example.
                int numberInLoop = 0;
                for (int i=0;i<1000;i++) {
                        numberInLoop++;

                }
                System.out.println("FunctionalGroupTests1: testFunctionalTests1Test1");
                Assert.assertThat(numberInLoop,isA(Integer.class));
        }

        @Test @Category(FunctionalGroupTests1.class)
        public void testFunctionalTests1Test2() {
                //      You're test case here: Below is just an example.
                int numberInLoop = 0;
                for (int i=1000;i<4000;i++) {
                        numberInLoop++;

                }
                System.out.println("FunctionalGroupTests1: testFunctionalTests1Test2");
                Assert.assertThat(numberInLoop,isA(Integer.class));
        }
```

```
        @Test @Category(FunctionalGroupTests2.class)
        public void testFunctionalTests2Test1() {
                //      You're test case here: Below is just an example.
                int numberInLoop = 0;
                do{
                        numberInLoop++;
                }while(numberInLoop != 1000);
                System.out.println("FunctionalGroupTests2: testFunctionalTests2Test1");
                Assert.assertThat(numberInLoop,isA(Integer.class));
        }

        @Test @Category(FunctionalGroupTests2.class)
        public void testFunctionalTests2Test2() {
                System.out.println("FunctionalGroupTests2: testFunctionalTests2Test2");
        }

        @Test @Category({IntegrationTests.class,FunctionalGroupTests1.class})
        public void testIntegrationTestsTest1() {
                System.out.println("IntegrationTests: testIntegrationTestsTest1");
        }

        @Test @Category(SanityTests.class)
        public void testSanityTestsTest1() {
                System.out.println("SanityTests: testSanityTestsTest1");
        }
}
```

3.4 Running our example

3.4.1 Running Tests by Category

We can run specific test case category by running the commands in Maven below. `mvn test -Dgroups="com.areyes.junit.cat.intf.FunctionalGroupTests1, com.areyes.junit.cat.intf.FunctionalGroupTests2"` `mvn test -Dgroups="com.areyes.junit.cat.intf.IntegrationTests, com.areyes.junit.cat.intf.SanityTests"`

3.4.2 Running tests by Category Profile

Alternatively, we can run tests by profile. We need to update our pom.xml and add a new profiles. We will then use these profiles and tag the categories we created to each as shown below.

pom.xml

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↩
    XMLSchema-instance"
        xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd ↩
            /maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>com.jgc.areyes.junit</groupId>
        <artifactId>junit-categories-example</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <build>
                <plugins>
                        <plugin>
                                <groupId>org.apache.maven.plugins</groupId>
                                <artifactId>maven-surefire-plugin</artifactId>
                                <version>2.17</version>
                                <dependencies>
```

```
                                        <dependency>
                                                <groupId>org.apache.maven.surefire</groupId ↩
                                                    >
                                                <artifactId>surefire-junit47</artifactId>
                                                <version>2.17</version>
                                        </dependency>
                                </dependencies>
                                <configuration>
                                        <groups>${testcase.groups}</groups>
                                        <excludes>
                                                <exclude>${exclude.tests}</exclude>
                                        </excludes>
                                        <includes>
                                                <include>${include.tests}</include>
                                        </includes>
                                </configuration>
                        </plugin>
                </plugins>
        </build>
        <profiles>
            <profile>
                <id>sanityTests</id>
                <properties>
                    <testcase.groups>com.areyes.junit.svc.SanityTests</testcase.groups>
                </properties>
            </profile>
            <profile>
                <id>functionalGroupTests1</id>
                <properties>
                    <testcase.groups>com.areyes.junit.svc.FunctionalGroupTests1</testcase. ↩
                        groups>
                </properties>
            </profile>
            <profile>
                <id>functionalGroupTests2</id>
                <properties>
                    <testcase.groups>com.areyes.junit.svc.FunctionalGroupTests2</testcase. ↩
                        groups>
                </properties>
            </profile>
            <profile>
                <id>integrationTests</id>
                <properties>
                    <testcase.groups>com.areyes.junit.svc.IntegrationTests</testcase.groups ↩
                        >
                </properties>
            </profile>
        </profiles>

        <dependencies>
                <dependency>
                        <groupId>junit</groupId>
                        <artifactId>junit</artifactId>
                        <version>4.12</version>
                        <scope>test</scope>
                </dependency>
        </dependencies>
</project>
```

Run them by using the following maven command: `mvn test -pfunctionalGroupTests1`

3.5 Download the Eclipse project

This was an example of JUnit Category.

Download

You can download the full source code of this example here: **junit-categories-example**

Chapter 4

Disable Test Example

4.1 Introduction

There are instances that our test cases aren't ready yet and it would almost be certain that when we run our builds with these, there is a high probability that our test executions will fail. This can be avoided using the `@Ignore` annotation.

Developers sometimes just pass the test case to avoid build failures (assert(true)). This is a bad practice and should be avoided. The `@Ignore` annotation is a better alternative as it will allow the runner to just ignore the test case ensuring that it won't be part of the build run.

This annotation can be used by developers to tag a specific test case as disabled. This means that even if we run our builds, the JUnit test runner won't bother running them since it assumes that it's not yet ready or intentionally disabled.

4.2 Source

Let's start with creating an implementation class. The following code will be our implementation class that we will create a test case from.

MessageService.java

```
package com.areyes.junit.svc;

/**
 * The Class MessageService.
 */
public class MessageService {



        /** The message. */
        private String message;


        /**
         * Instantiates a new message service.
         *
         * @param message the message
         */
        public MessageService(String message) {
                this.message = message;
        }

        /**
```

```
         * Prints the message.
         *
         * @return the string
         */
        public String printMessage() {
                return message;
        }

        /**
         * Salutation message.
         *
         * @return the string
         */
        public String salutationMessage() {
                message = "Hi!" + message;
                return message;
        }

        /**
         * This will be the method to get the salutation messages specifically for executives.
         * @return
         */
        public String salutationMessageForExecutives() {
                return "this is not yet implemented";
        }

}
```

As it can be seen on the class above, we have a method that has not been implemented yet. We don't want our test case to run this don't we? So in order for our test case to ignore this, we use the `@Ignore` on the specific test. See the test case below.

MessageServiceTest.java

```
package com.areyes.junit.svc;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Ignore;
import org.junit.Test;
import static org.hamcrest.CoreMatchers.isA;

public class MessageServiceTest {

        private String CONST_MSG = "Message is me";
        private MessageService msgService = new MessageService(CONST_MSG);

        @Test
        public void testPrintMessage() {
                //      Check type return
                assertThat(msgService.printMessage(), isA(String.class));
                assertEquals(CONST_MSG, msgService.printMessage());

        }

        @Test
        public void testSalutationMessage() {
                String messageSal = msgService.salutationMessage();
                assertThat(messageSal, isA(String.class));
                assertEquals("Hi!" + CONST_MSG,messageSal);
```

```
        }


        @Ignore
        @Test
        public void testSalutationMessageForExecutives() {
                assertThat(msgService.salutationMessageForExecutives(), isA(String.class));
                assertEquals(CONST_MSG, msgService.salutationMessage());
        }
}
```

As seen on the `testSalutationMessageForExecutives()` method, it's as simple as putting an `@Ignore`annotation on your test case. This is our way (developers) to tell the JUnit Runner that we don't want to run this specific case since we haven't finished the implementation yet.

4.3 Result

Result of running the test in Eclipse

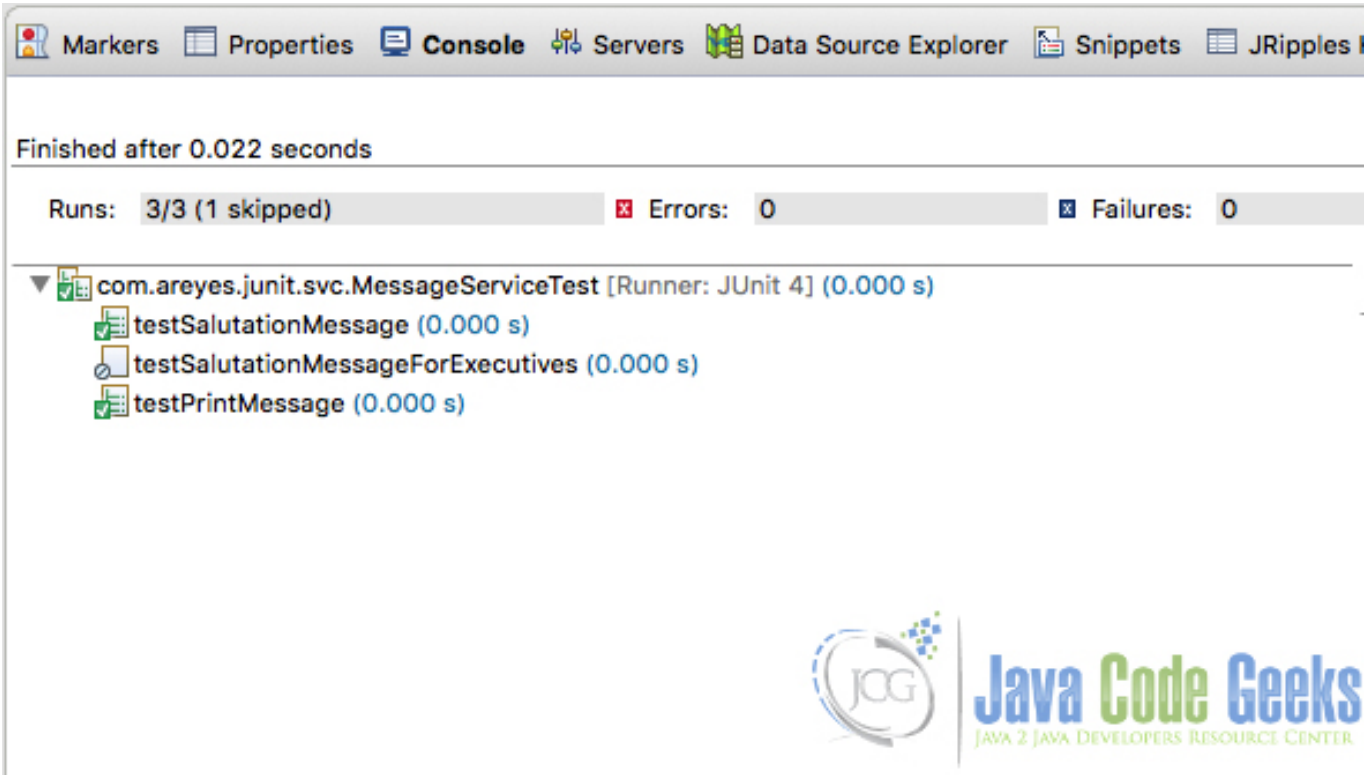

Figure 4.1: JUnit Disable Ignore example

4.4 Download the Eclipse project

This was an example of JUnit Disable Test

Download

You can download the full source code of this example here: **junit-disable-example**

Chapter 5

MultiThreaded Test Example

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.

5.1 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.

5.2 Technology Stack

Technologies used in this example are

- Java
- JUnit 4.12
- Eclipse (users can use any IDE of their choice)
- Maven (for dependency management)

5.3 Project Setup

Start creating a Maven project.

Select `File -> New -> Maven Project`

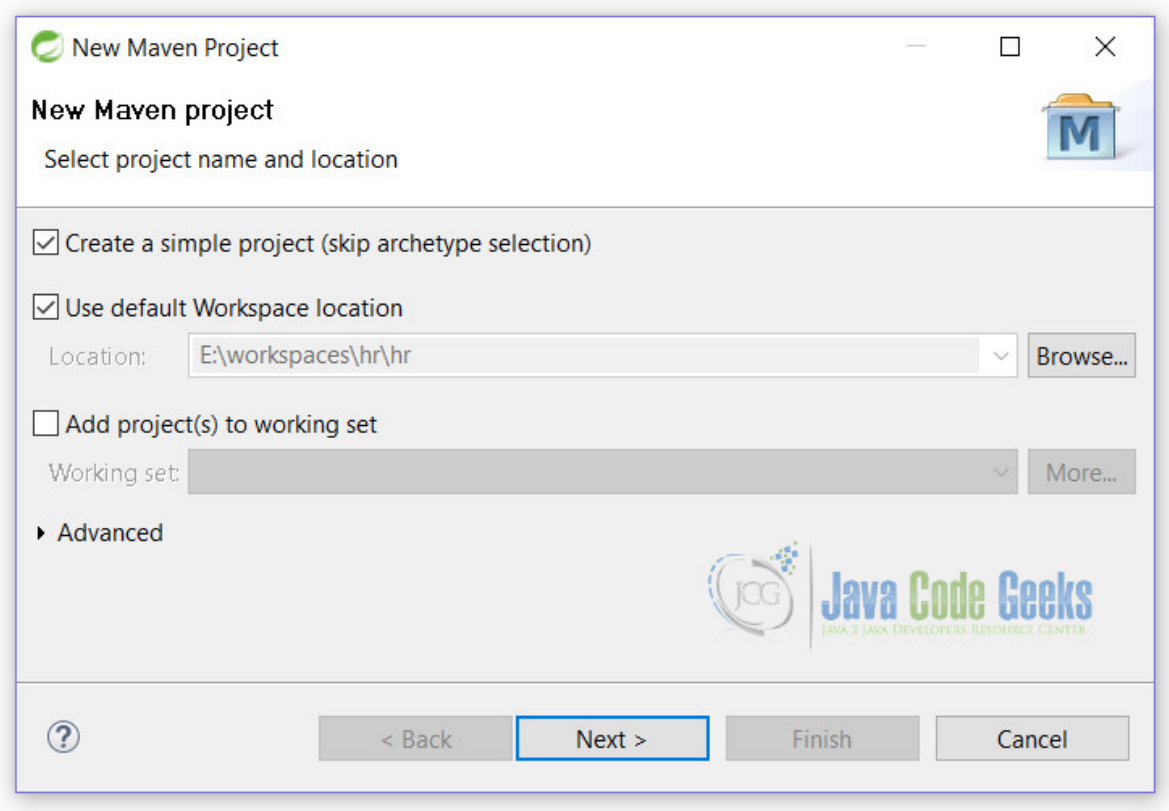

Figure 5.1: JUnit MultiThread Example Step 1

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

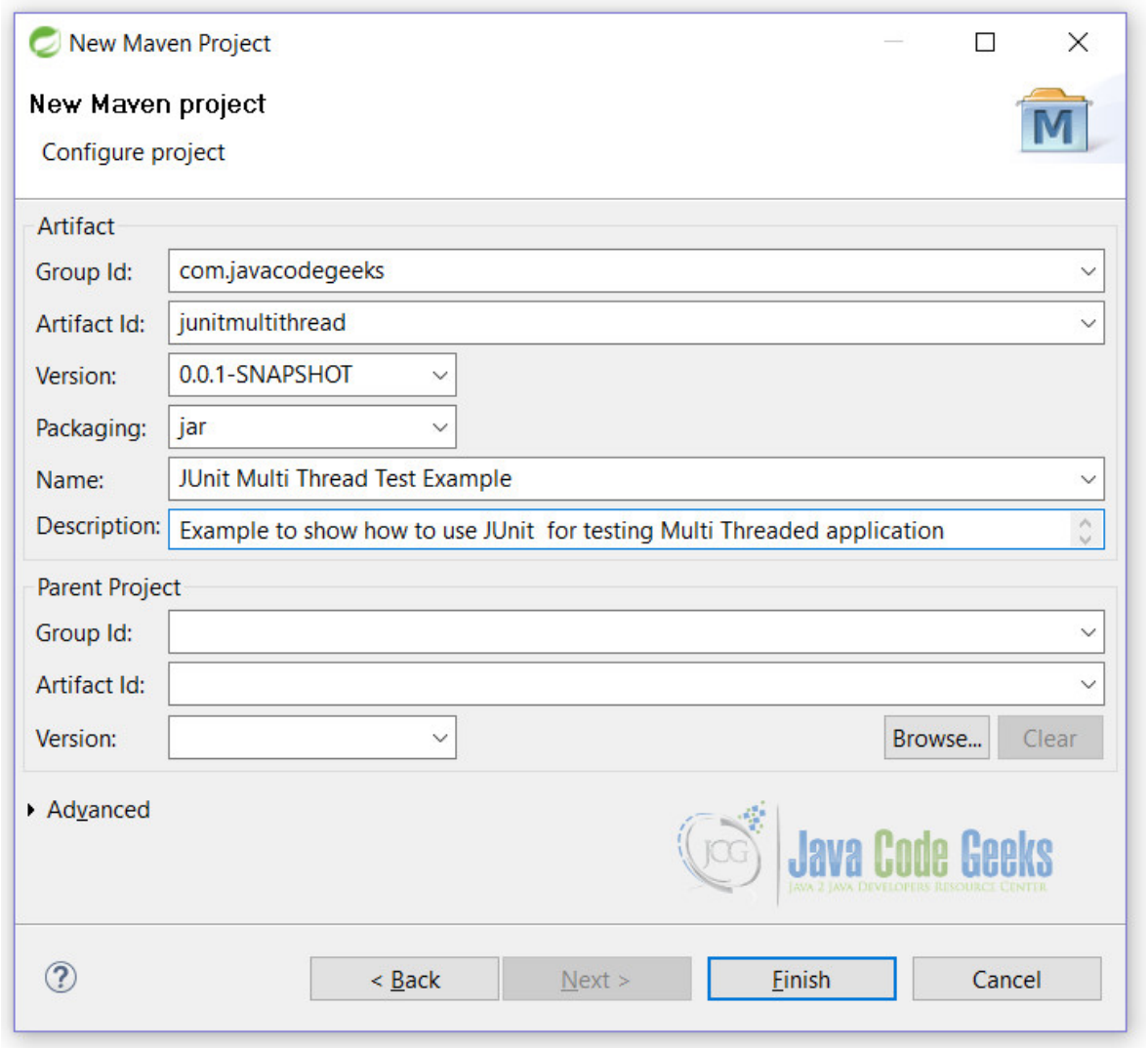

Figure 5.2: JUnit MultiThread Example Step 2

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

5.4 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.

5.4.1 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 `@ThreadCount` annotation later in the example.

5.4.2 Classes

CountCheck.java

```
package junitmultithread;

import java.util.concurrent.atomic.AtomicInteger;

public class CountCheck {

        private final AtomicInteger count = new AtomicInteger();

        public void initialize(int number){
                count.set(number);
        }

        public void addOne() {
                count.incrementAndGet();
        }

        public int getCount() {
                return count.get();
        }
}
```

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

```
package junitmultithread;

import static org.junit.Assert.assertEquals;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
```

```
import com.anarsoft.vmlens.concurrent.junit.ConcurrentTestRunner;
import com.anarsoft.vmlens.concurrent.junit.ThreadCount;

@RunWith(ConcurrentTestRunner.class)
public class CountCheckTest {

        private CountCheck counter = new CountCheck();

        @Before
        public void initialCount(){
                counter.initialize(2);
        }

        @Test
        public void addOne() {
                counter.addOne();
        }

        @After
        public void testCount() {
                assertEquals("Value should be 6", 6, counter.getCount());
        }
}
```

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 `addOne()` 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.

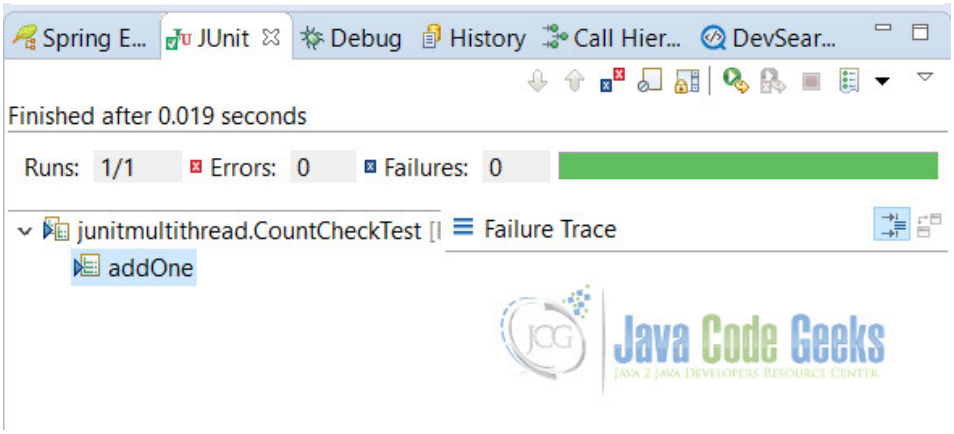

Figure 5.3: JUnit Multi Thread Example Test Result

As a result, test case is passed, because we are testing with the `assertEquals()`, which tests for equality.

5.4.2.1 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.

```
package junitmultithread;

import static org.junit.Assert.assertEquals;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

import com.anarsoft.vmlens.concurrent.junit.ConcurrentTestRunner;
import com.anarsoft.vmlens.concurrent.junit.ThreadCount;

@RunWith(ConcurrentTestRunner.class)
public class CountCheckThreadCountTest {

        private CountCheck counter = new CountCheck();
        private final static int THREAD_COUNT = 5;

        @Before
        public void initialCount() {
                counter.initialize(2);
        }

        @Test
        @ThreadCount(THREAD_COUNT)
        public void addOne() {
                counter.addOne();
        }

        @After
        public void testCount() {
                assertEquals("Value should be 7", 7, counter.getCount());
        }
}
```

We have taken thread count to **5**. Remember, by default there are **4** threads.

5.5 Conclusion

In conclusion, through this example, we learnt to test the multi threaded applications using JUnit. Also, you got to know about the use of the concurrent-junit library to test the multi threading application.

5.6 Download the Eclipse project

This is an example of JUnit MultiThread.

Download

You can download the full source code of this example here: **JUnitMultiThread.zip**

Chapter 6

Keyboard Input Example

In this post we shall show users the usage of JUnit keyboard input working. This example is very useful in case users want to enter data from keyboard for testing of their methods. Do not worry, we will the same in the post.

Users are advised to see the JUnit Hello World example where they can see the basic usage of JUnit. In addition, if users want to run their test cases in the order of their choice, they are recommended to visit JUnit FixMethodOrder example.

First of all, let's start with the small introduction of JUnit.

6.1 JUnit Introduction

JUnit is a testing framework for Java programmers. This is the testing framework where users can unit test their methods for working. Almost all Java programmers used this framework for basic testing. Basic example of the JUnit can be seen in JUnit Hello World example.

6.2 Tools Required

Users have to have a basic understanding of the Java. These are the tools that are used in this example.

- Eclipse (users can use any tool of their choice. We are using STS, which is built on the top of the Eclipse)
- Java
- Maven (optional, users can also use this example without Maven, but we recommend to use it)
- JUnit 4.12 (this is the latest version we are using for this example)

6.3 Project Setup

In this post we are using the Maven for initial setup of our application. Let's start with the creating of the Maven project. Open eclipse, `File -> New -> Maven Project` On the following screen, fill in the details as shown and click on Next button.

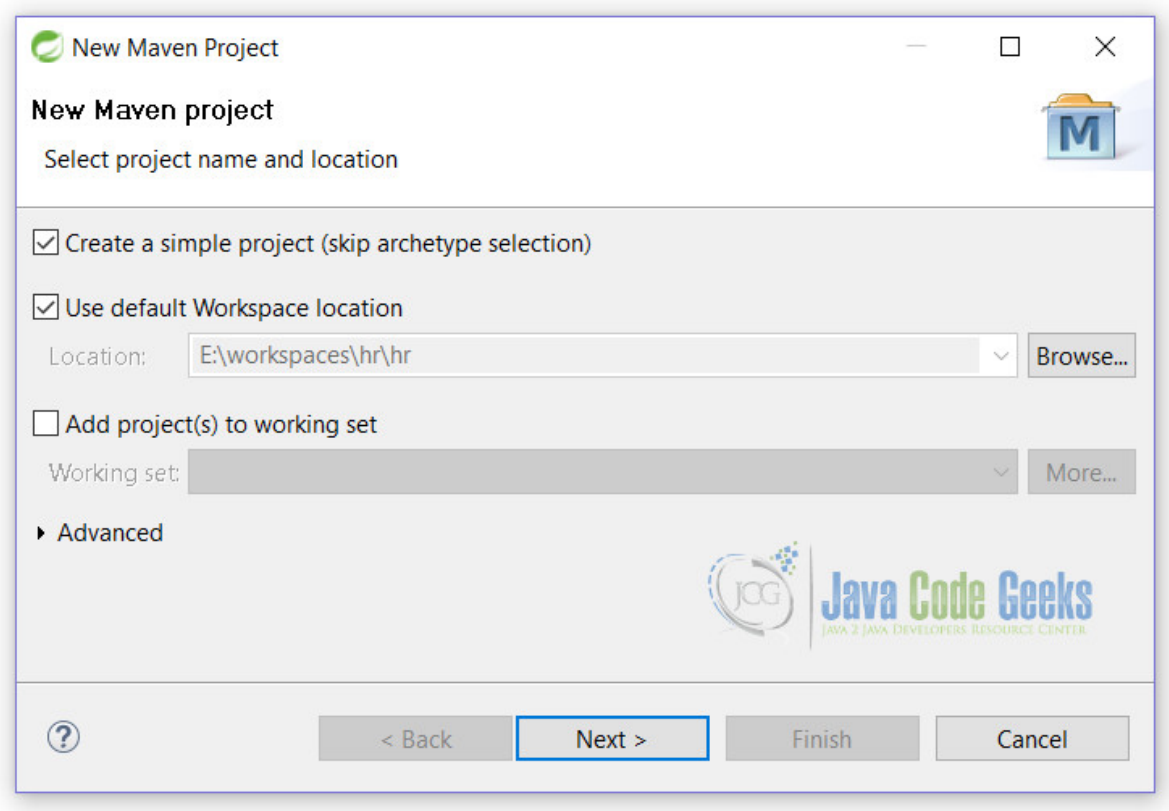

Figure 6.1: Project Setup Screen 1

Clicking on Next button users will be taken to the below screen. Fill in the details as shown and click on finish.

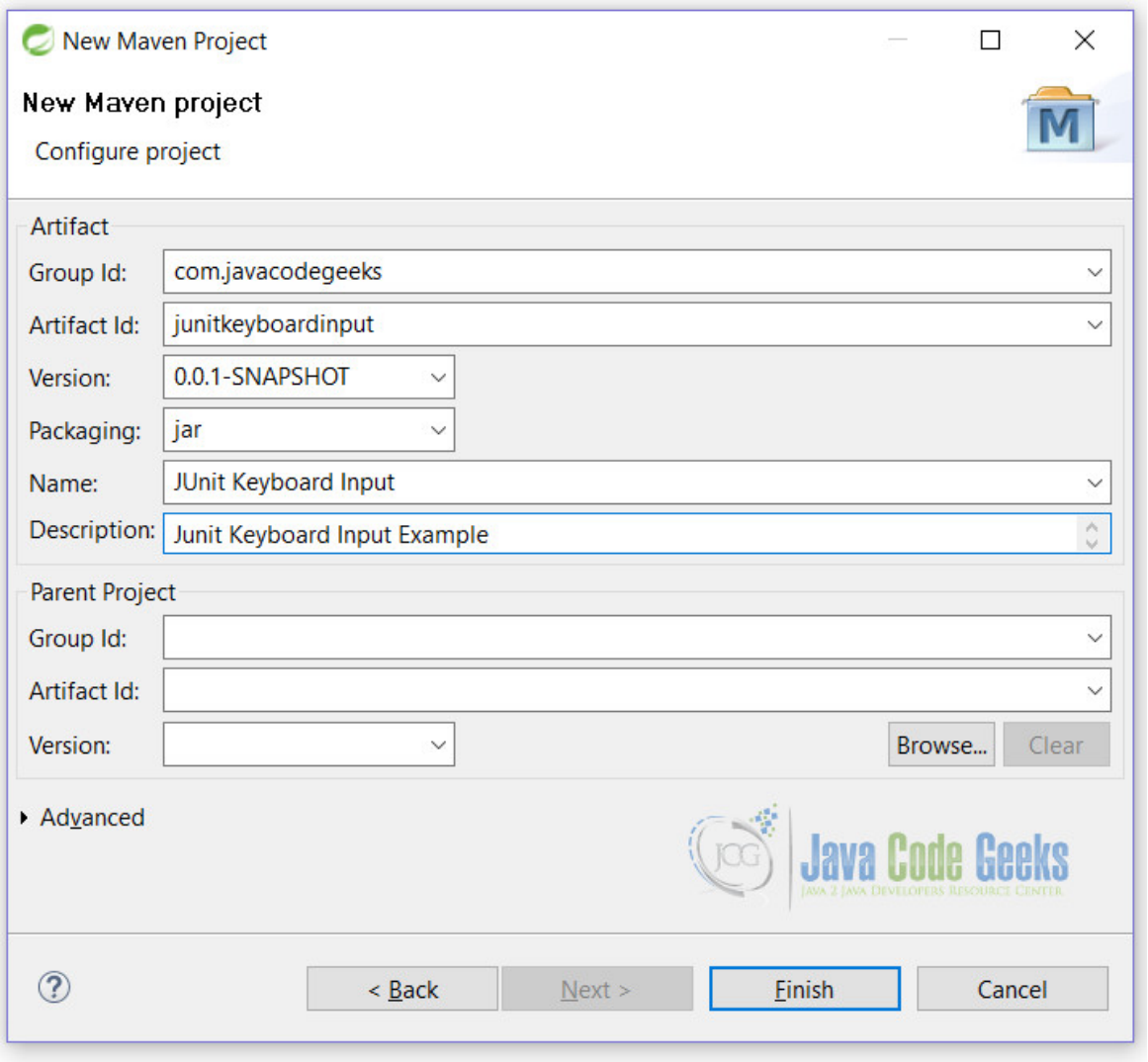

Figure 6.2: Project Setup Screen 2

Now we are ready to start writing code for our application.

6.4 JUnit Keyboard Input

Open `pom.xml` and add the following lines to it. This is the main file which is used by the Maven to download dependencies on users local machine.

pom.xml

```
<dependencies>
     <dependency>
          <groupId>junit</groupId>
          <artifactId>junit</artifactId>
          <version>4.12</version>
     </dependency>
</dependencies>
```

Create a class with the following code. This class will test the parameter passed for leap year. If the passed parameter is Leap year, it will return `true` otherwise it will return `false`.

LeapYear.java

```
package junitkeyboardinput;

public class LeapYear {

        public boolean isLeapYear(int year) {
                return ((year % 400 == 0) || ((year % 4 == 0) && (year % 100 != 0)));
        }
}
```

Finally, we create a `LeapYearTest` class which is a JUnit keyboard input test class, that test our `isLeapYear()` method. We are using, Scanner class of Java.

LeapYearTest.java

```
package junitkeyboardinput;

import static org.junit.Assert.assertTrue;

import java.io.IOException;
import java.util.Scanner;

import org.junit.Test;

public class LeapYearTest {

        @Test
        public void isLeapYearTest() throws IOException {
                LeapYear check = new LeapYear();
                assertTrue("Leap Year", check.isLeapYear(2015));
        }

        @Test
        public void isLeapYearKeboardTest() throws IOException {
                LeapYear leapYear = new LeapYear();

                Scanner sc = new Scanner(System.in);
                System.out.print("Enter year(yyyy):");
                int year = sc.nextInt();
                assertTrue("Leap Year", leapYear.isLeapYear(year));
                sc.close();
        }
}
```

Let's see what is happening in this class. **Line no 22**: We are creating object of Scanner class. **Line no 24**: We are taking the input as Integer **Line no 26**: closing the Scanner.

In the console window, users will be prompted for entering year. Enter year and depend upon the input, test case will fail or pass.

```
Enter year(yyyy):
```

If user passes, 2016 as the year, result will be shown in JUnit window.

```
Enter year(yyyy): 2016
```

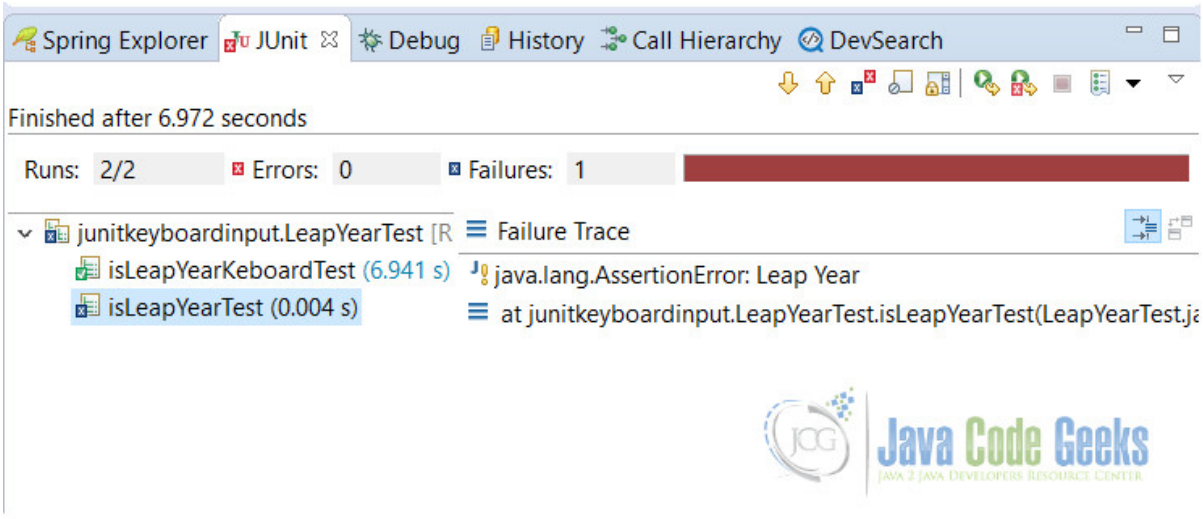

Figure 6.3: JUnit Result Window

6.5 Examine the output

Users can see that there are 2 test cases, out of which one is passed and other is failed. Let's examine the output. 1. One of the test case is failed i.e. `isLeapYearTest()` , as the result of the parameter passed which is not a leap year.(See LeapYearTest.java, line no 15) 2. Other test is passed i.e. `isLeapYearKeboardTest()`, as we have passed the leap year.

We advised users to experiment with the example to see the real output.

6.6 Conclusion

In conclusion, this is an example of JUnit Keyboard input, where we have learnt how they can use the keyboard to test their methods in JUnit.

6.7 Download the Eclipse Project

This was an example of JUnit Keyboard Input.

Download

You can download the full source code of this example here: **JUnitKeyboardInput.zip**

Chapter 7

Group Tests Example

In this example we shall show users, how they can group and run their JUnit test cases. JUnit group tests example, will try to resolve issue of running multiple group tests all together. This is not a big deal in JUnit.

This can be achieved in different ways in JUnit. It's wide API, helps developers all around the world to achieve the flexibility to test their methods. Let's start with the introduction of JUnit and what JUnit group tests example is all about.

7.1 Introduction

JUnit is very popular library among Java developers for testing the programs at unit level. JUnit provides many resources to test each and every type of method. You can test simple methods, in the order of the test cases, through keyboard input or multithreaded applications

This example will show the usage of different JUnit annotations that provides users with the ease of running the group test cases. We will be using the Maven as a dependency and build tool for this example.

7.2 Technologies Used

Following set of technologies are used for this example to work.

- Java
- Maven - It is used as a dependency and build tool.
- Eclipse - Users can use any IDE of their choice.
- JUnit 4.12

7.3 Project Setup

IDE used for this example is Eclipse. Start by creating a new maven project. Select `File -> New -> Maven Project`. You will see following screen. Fill in the details as shown and click on Next button.

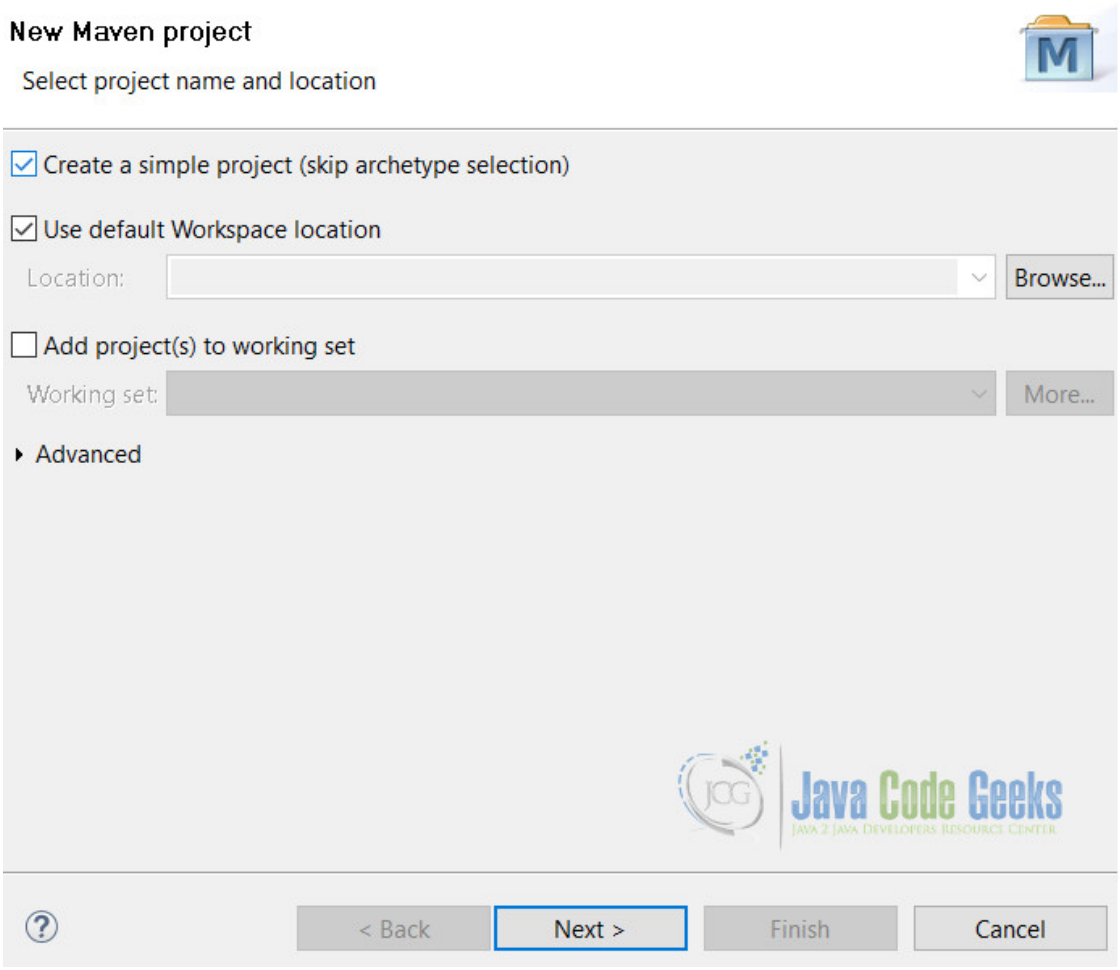

Figure 7.1: JUnit Group Tests Example Setup 1

On this screen, you will be asked to enter about the project. Fill all details as shown and click on Finish button.

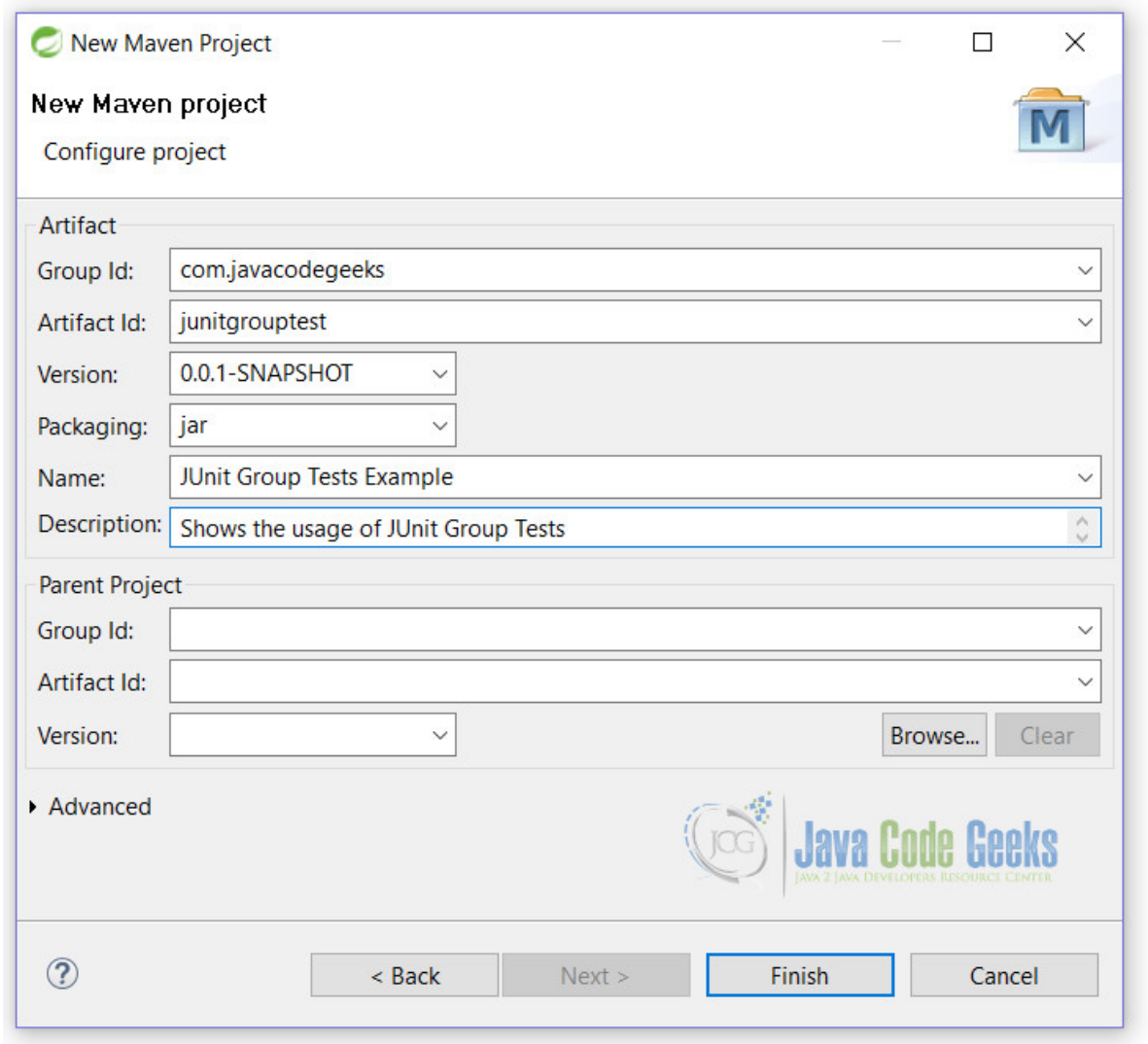

Figure 7.2: JUnit Group Tests Example Setup 2

That's it. You are done with the project creation. We will start coding the example right away after this.

7.4 JUnit Group Tests Example

Now open `pom.xml` and add the following lines to it.

pom.xml

```
<dependencies>
     <dependency>
          <groupId>junit</groupId>
          <artifactId>junit</artifactId>
          <version>4.12</version>
    </dependency>
</dependencies>
```

There are 2 approaches in JUnit to group test the methods. We will start with basic and then go with the more complicated one.

- @RunWith(Suite.class)

- @RunWith(Categories.class)

7.4.1 @RunWith(Suite.class)

This annotation is helpful whenever we want to test multiple classes at once. In this case we do not need to run each individual class for testing. Simply run the class with `@RunWith(Suite.class)` annotation and it will take care of running all your test cases one by one. See the below example for more clarity.

ClassATest.java

```
package junitgrouptest;

import org.junit.Test;

public class ClassATest {

        @Test
        public void classA_Test1(){
                System.out.println("classA_Test1");
        }

        @Test
        public void classA_Test2(){
                System.out.println("classA_Test2");
        }

}
```

ClassBTest.java

```
package junitgrouptest;

import org.junit.Test;

public class ClassBTest {

        @Test
        public void classB_Test1() {
                System.out.println("classB_Test1");
        }

        @Test
        public void classB_Test2() {
                System.out.println("classB_Test2");
        }
}
```

ClassCTest.java

```
package junitgrouptest;

import org.junit.Test;

public class ClassCTest {

        @Test
        public void classC_Test1() {
                System.out.println("classC_Test1");
        }

        @Test
```

```
        public void classC_Test2() {
                System.out.println("classC_Test2");
        }
}
```

7.4.1.1 Test Suite

Now, we will create a class that will helps in running all our testcases at once.

ClassTestSuite.java

```
package junitgrouptest;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({ ClassATest.class, ClassBTest.class, ClassCTest.class })
public class ClassTestSuite {

}
```

When you run `ClassTestSuite` class, you will get the following output:

```
classA_Test1
classA_Test2
classB_Test1
classB_Test2
classC_Test1
classC_Test2
```

Since, we have included all our classes in `@SuiteClasses()` annotation, all test cases of every class runs. We can modify it to run our specific classes also.

It is very convenient to run in these types of scenarios. But when you want more complicated test cases like some specific test cases to run from a class, or you want to run some type of test cases all together, then you need a more control over your test cases.

In JUnit 4.8, the concept of `categories` is introduced. We will see the usage of `categories` in below example.

7.4.2 @RunWith(Categories.class)

Another way of running test suite is with `@RunWith(Categories.class)` annotation. This is more organized way of running your test cases. By this way, users have more control over test cases. `@Category` interface is used for this purpose. It works more like a marker interface, where we mark the test cases with it.

For this to work, first of all we need to create interfaces according to our choices like slowTests. You can take any type of name of your choice. To understand how it works, let's start by writing our example.

Simple interface without any methods in it.

SlowTests.java

```
package junitgrouptest;

public interface SlowTests {

}
```

PerfomanceTests.java

```
package junitgrouptest;

public interface PerfomanceTests {

}
```

And now make some changes in our previous classes by assigning them the category. `@Category` annotation can be used at both `method level` as well as at `class level`. Both cases are taken care in this example. For the sake of simplicity, we are going to show only changed code here. See the highlighted lines for changes.

ClassATest.java

```
...
@Test
@Category(PerformanceTests.class)
public void classA_Test1() {
     System.out.println("classA_Test1");
}
...
```

ClassBTest.java

```
...
@Test
@Category(PerformanceTests.class)
public void classB_Test1() {
     System.out.println("classB_Test1");
}

@Test
@Category(SlowTests.class)
public void classB_Test2() {
     System.out.println("classB_Test2");
}
...
```

ClassCTest.java

```
...
@Category(SlowTests.class)
public class ClassCTest {
...
```

7.4.2.1 Test Suite

Finally, we will create Test Suites to run these test cases.

PerformanceTestsSuite.java

```
package junitgrouptest;

import org.junit.experimental.categories.Categories;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Categories.class)
@Categories.IncludeCategory(PerformanceTests.class)
@Suite.SuiteClasses({ClassATest.class, ClassBTest.class, ClassCTest.class})
public class PerformanceTestsSuite {

}
```

When we run this test suite, we will get the following output:

```
classA_Test1
classB_Test1
```

This output is self explainatory. It shows us that the test cases marked with `@Category(PerformanceTests.class)` annotations runs and others don't.

SlowTestsSuite.java

```
package junitgrouptest;

import org.junit.experimental.categories.Categories;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Categories.class)
@Categories.IncludeCategory(SlowTests.class)
@Suite.SuiteClasses({ClassATest.class, ClassBTest.class, ClassCTest.class})
public class SlowTestsSuite {

}
```

When we run this test suite, we will get the following output:

```
classB_Test2
classC_Test1
classC_Test2
```

All test cases marked with `@Category(SlowTests.class)` annotation runs.

Similarly, like `@Categories.IncludeCategory()` annotation, we can also exclude some test cases from running. For this we have to use `@Categories.ExcludeCategory()` annotation.

7.5 Conclusion

JUnit Group Tests example provides, the way to test the JUnit test cases in more organized way. Users have learnt how they can achieve this by using 2 scenarios.

Firstly, by using the `@RunWith(Suite.class)` annotation

Secondly, with the use of `@RunWith(Categories.class)` annotation

7.6 Download the Eclipse Project

This is an example of JUnit Group Tests.

Download

You can download the full source code of JUnit Group Tests Example here: **JunitGroupTests.zip**

Chapter 8

RunListener Example

In JUnit RunListener Example, we shall show users how they can add `RunListener` to the test cases. There are cases when we want to respond to the events during a test case run. Here we can extend the `RunListener` class and override the methods according to our implementation. The JUnit `RunListener` can listen to the events of the JUnit lifecycle.

If somehow, listener throws an error, then it will be removed from remainder of the test case run. Let's see the example.

Furthermore, if user wants to learn about the basic of JUnit, they are advised to visit the below examples from Java Code Geeks.

- JUnit Hello World Example
- JUnit FixMethodOrder Example
- JUnit Keyboard Input Example
- JUnit MultiThreaded Test Example
- JUnit Group Tests Example

8.1 Technology Stack

We have used following technologies for this example to work.

- Java
- JUnit 4.12
- Maven - build and dependency tool
- Eclipse - IDE for writing code

8.2 Project Setup

Create a new maven project

Select `File -> New -> Maven Project`

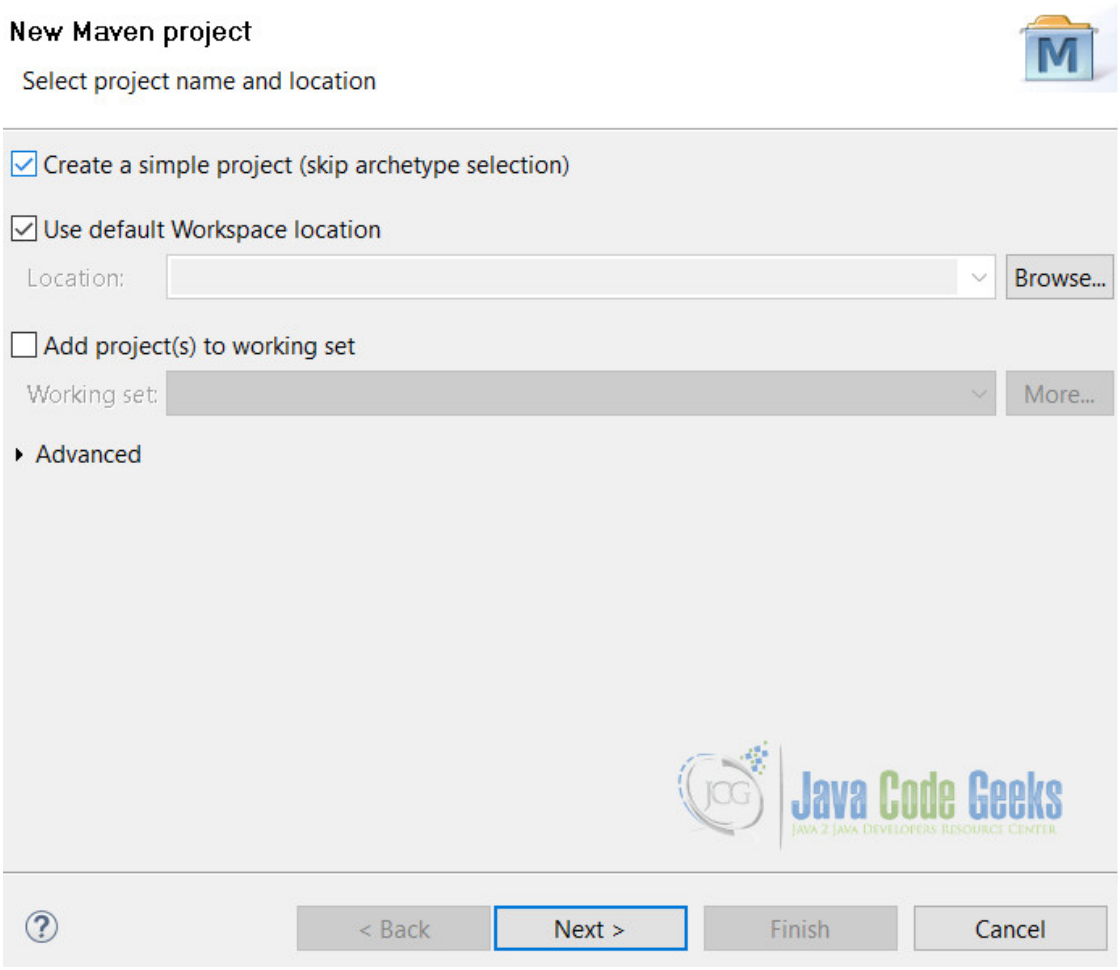

Figure 8.1: JUnit RunListener Example setup 1

Click on Next button. Fill in the details as detailed below:

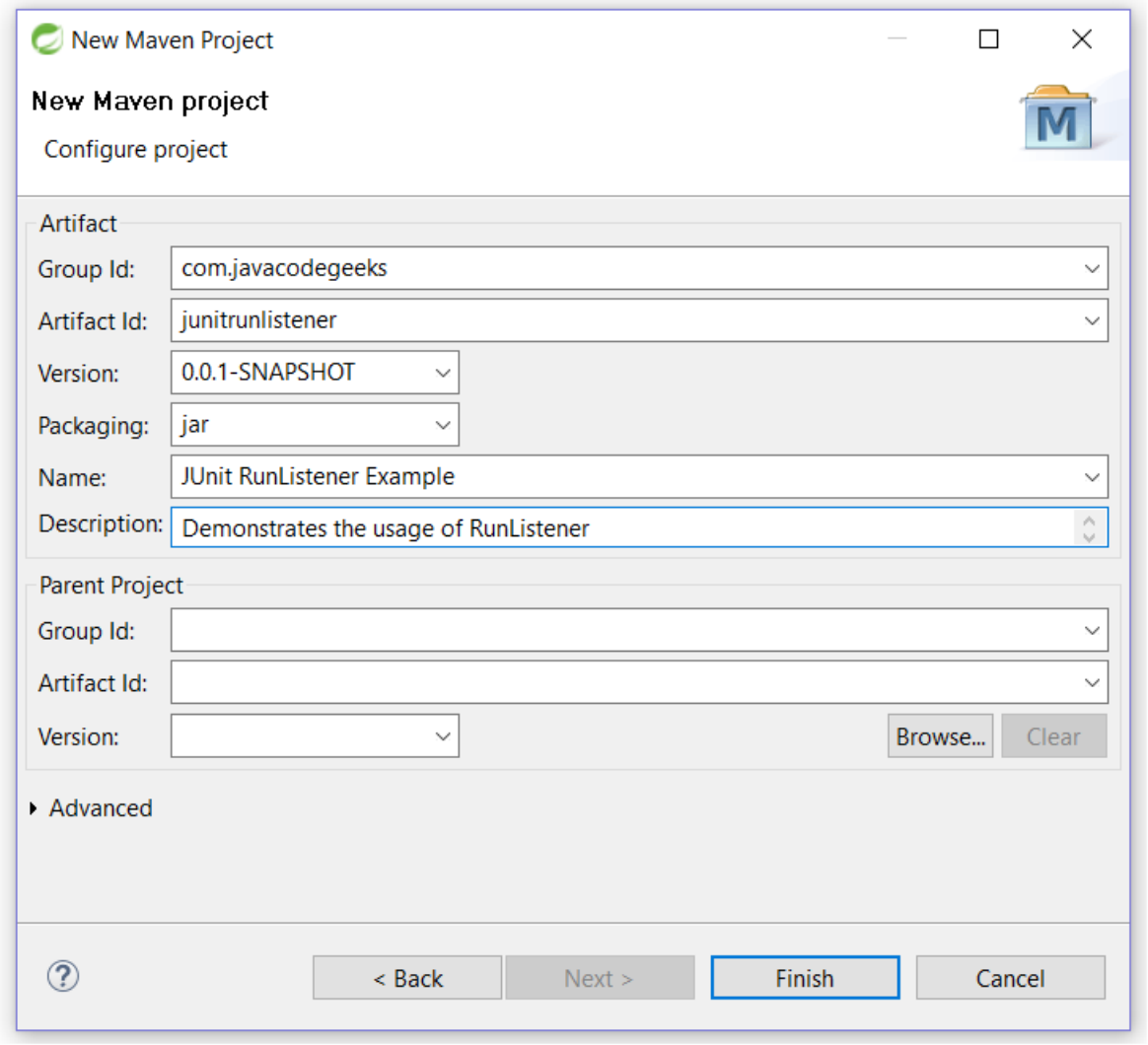

Figure 8.2: JUnit RunListener Example setup 2

With the click on Finish button, we are ready to start coding for this example.

8.3 JUnit RunListener Example

First of all we need to provide the JUnit jar to the project. For this we add following lines to the `pom.xml` pom.xml

```
<dependencies>
     <dependency>
          <groupId>junit</groupId>
          <artifactId>junit</artifactId>
          <version>4.12</version>
     </dependency>
</dependencies>
```

Now, we will create a class which will extends the `RunListener` class. This class has many methods that we can override. It is our wish which methods to implement and which to ignore. For the sake of knowledge of users, we have taken all methods here and write about them. You can skip some of them. Code of this class is self explanatory.

OurListener.java

```
package junitrunlistener;

import org.junit.runner.Description;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;
import org.junit.runner.notification.RunListener;

public class OurListener extends RunListener {

        // Called before any tests have been run.
        public void testRunStarted(Description description) throws java.lang.Exception {
                System.out.println("Test cases to execute : " + description.testCount());
        }

        // Called when all tests have finished
        public void testRunFinished(Result result) throws java.lang.Exception {
                System.out.println("Test cases executed : " + result.getRunCount());
        }

        // Called when an atomic test is about to be started.
        public void testStarted(Description description) throws java.lang.Exception {
                System.out.println("Execution Started : " + description.getMethodName());
        }

        // Called when an atomic test has finished, whether the test succeeds or
        // fails.
        public void testFinished(Description description) throws java.lang.Exception {
                System.out.println("Execution Finished : " + description.getMethodName());
        }

        // Called when an atomic test fails.
        public void testFailure(Failure failure) throws java.lang.Exception {
                System.out.println("Execution Failure : " + failure.getException());
        }

        // Called when a test will not be run, generally because a test method is
        // annotated with Ignore.
        public void testIgnored(Description description) throws java.lang.Exception {
                System.out.println("Execution Ignored : " + description.getMethodName());
        }

        // Called when an atomic test flags that it assumes a condition that is false
        public void testAssumptionFailure(Failure failure){
                System.out.println("Assumption Failure : " + failure.getMessage());
        }
}
```

8.3.1 Test Classes

We will create 2 test classes for this example.

TestClassA.java

It has 2 test methods, `test_A_1()` and `test_A_2()`.

```
package junitrunlistener;

import static org.junit.Assert.assertTrue;

import org.junit.Test;
```

```
public class TestClassA {

        @Test
        public void test_A_1(){
                assertTrue(1==2);
        }

        @Test
        public void test_A_2(){
                assertTrue(true);
        }
}
```

Due to line no 11(highlighted), `test_A_1()` method fails and throws `java.lang.AssertionError`.

TestClassB.java

This class also have 2 methods, `test_B_1()` and `test_B_2()`.

```
package junitrunlistener;

import static org.junit.Assert.assertTrue;

import org.junit.Ignore;
import org.junit.Test;

public class TestClassB {

        @Test
        public void test_B_1(){
                assertTrue(true);
        }

        @Ignore
        @Test
        public void test_B_2(){
                assertTrue(2==5);
        }
}
```

As you can see that `test_B_2()` is marked with `@Ignore` annotation. This annotation will simply ignore this test case from running.

8.3.2 Main Class

TestClassRun.java

Now, we are ready to run our tests. Create a class with the following code.

```
package junitrunlistener;

import org.junit.runner.JUnitCore;

public class TestClassRun {

        public static void main(String[] args) {
                JUnitCore runner = new JUnitCore();
                runner.addListener(new OurListener());
                runner.run(TestClassA.class, TestClassB.class);
        }
}
```

Here, we have used `JUnitCore` class of JUnit to run the test cases. This is required as we need to add our custom listener to the test cases. See the highlighted line in above class.

8.3.2.1 Output

```
Test cases to execute : 4
Execution Started : test_A_1
Execution Failure : java.lang.AssertionError
Execution Finished : test_A_1
Execution Started : test_A_2
Execution Finished : test_A_2
Execution Started : test_B_1
Execution Finished : test_B_1
Execution Ignored : test_B_2
Test cases executed : 3
```

It is cleared from the output that with each test case, `testStarted()` and `testFinished()` methods are called.

One of the test is failed due to the condition which we have passed in `TestClassA` class.

8.4 Conclusion

We have learnt in this example, that by using a custom listener in JUnit we can log and do tasks accordingly on the basis of methods executed. Like, if you want to call or notify the user that a particular test case is failed, you can simply write that piece of code in `testFailure()` method.

It is also very useful in logging purpose.

8.5 Download the Eclipse Project

This is JUnit RunListener Example.

Download

You can download the full source code of this example here: **JUnitRunListenerExample.zip**

Chapter 9

Hamcrest Example

In this example we shall show users the usage of Hamcrest. Through JUnit Hamcrest Example we will show users what is hamcrest, where it is used, why it is used, when it is used and how to use it on your applications. If you are a regular user of my JUnit series then you are already familiar with the JUnit.

If you want to see more example of JUnit, please visit my series page.

We will start by getting a little bit information about the hamcrest. It has a very good integration with JUnit and both provides a good framework for testing.

9.1 Introduction

Hamcrest is a open source framework matcher library used in various language to match expression for your test cases. You can visit github page if you want to explore the code of it.

Hamcrest has a very rich library of methods to fulfill our needs. It is used with different testing frameworks like JUnit and jMock. Hamcrest is typically viewed as a third generation matcher framework.

- **First Generation**: It typically uses `assert(some statement)`. In this case tests are not easily readable.
- **Second Generation**: It uses the special methods such as `assertEquals()` for test. But this approach creates a lots of assert methods.
- **Third Generation**: It uses `assertThat()` method for test. It is more flexible and covers most of the scenarios. The benefit is that you still get fluent error messages when the assertion fails, but now you have greater extensibility.

In our example we will use `assertThat()` for all our tests.

9.2 Technologies Used

- Java
- JUnit 4.12 - Testing framework
- Hamcrest 1.3 - Used for matchers
- Eclipse - IDE for code
- Maven - dependency management tool

9.3 Project SetUp

Open Eclipse. Select `File -> New -> Maven Project`. Fill in the following details.

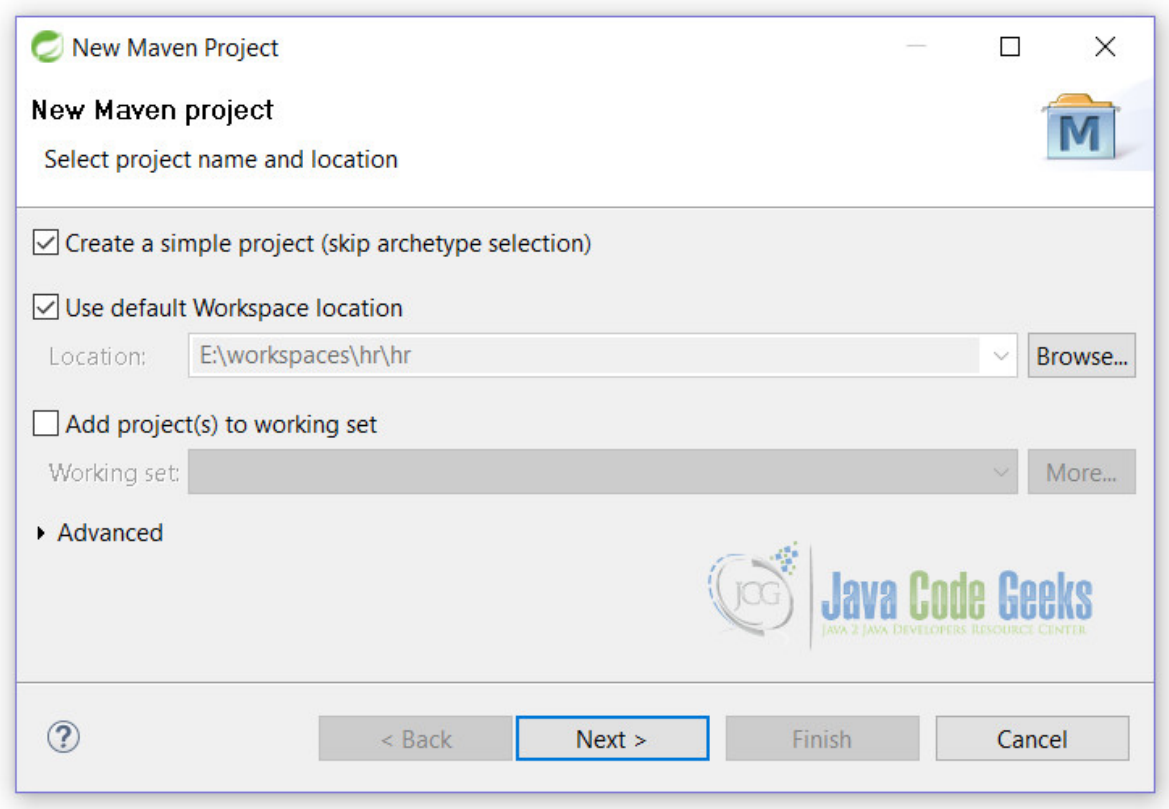

Figure 9.1: JUnit Hamcrest Example Setup 1

After clicking Next button, you will be taken to following screen. Simply fill out all the necessary details and click on Finish button.

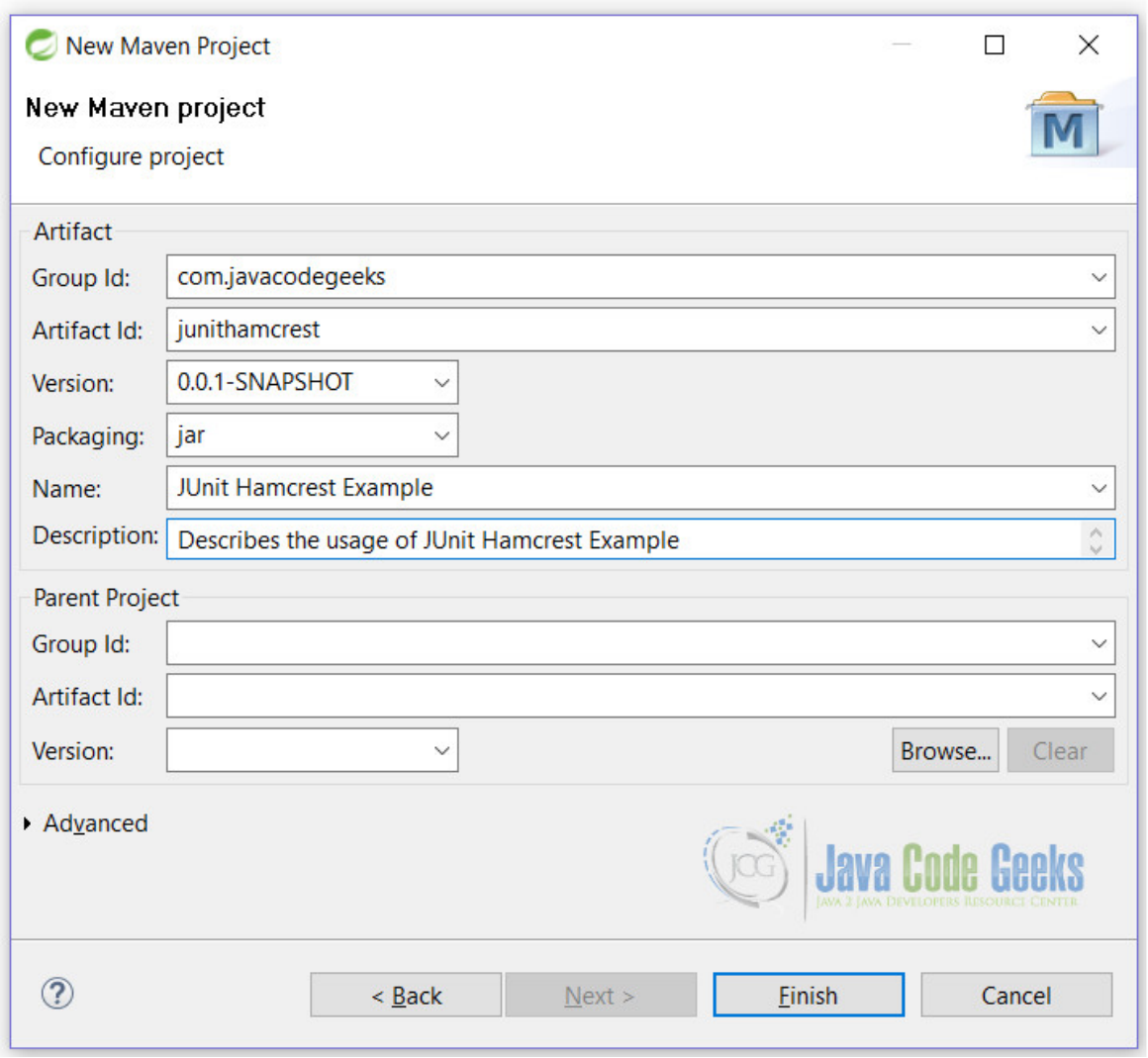

Figure 9.2: JUnit Hamcrest Example Setup 2

This will create an empty maven project.

9.4 JUnit Hamcrest Example

Start by writing following lines to the `pom.xml`

pom.xml

```
<dependencies>
        <!-- https://mvnrepository.com/artifact/junit/junit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.hamcrest/hamcrest-library -->
        <dependency>
            <groupId>org.hamcrest</groupId>
```

```
            <artifactId>hamcrest-library</artifactId>
            <version>1.3</version>
        </dependency>
    </dependencies>
```

If you simply write JUnit in `pom.xml`, it will fetch `hamcrest-core` jar with it. But we have also included `hamcrest-library` jar as we need to use different matchers that are not provided by default by JUnit.

We will discuss details about each and every part of the example. First of all, let's create a model class. This is a simple class that will help us to run our test cases.

Employee.java

```
package junithamcrest;

import java.util.List;

public class Employee {

        private Long empId;
        private String empName;
        private String gender;
        private List awards;

        public Employee(Long empId, String empName, String gender, List awards) {
                super();
                this.empId = empId;
                this.empName = empName;
                this.gender = gender;
                this.awards = awards;
        }

        public Long getEmpId() {
                return empId;
        }

        public void setEmpId(Long empId) {
                this.empId = empId;
        }

        public String getEmpName() {
                return empName;
        }

        public void setEmpName(String empName) {
                this.empName = empName;
        }

        public String getGender() {
                return gender;
        }

        public void setGender(String gender) {
                this.gender = gender;
        }

        public List getAwards() {
                return awards;
        }

        public void setAwards(List awards) {
                this.awards = awards;
        }
```

```
}
```

Now create a test class, so that we can test above class. This class will test all hamcrest matchers. We have tried to cover most common but users are advised to see other matchers if they want to dig more deeper.

JUnitHamcrestTestClass.java

```
package junithamcrest;

import static org.hamcrest.CoreMatchers.anyOf;
import static org.hamcrest.CoreMatchers.equalTo;
import static org.hamcrest.CoreMatchers.everyItem;
import static org.hamcrest.Matchers.allOf;
import static org.hamcrest.Matchers.containsString;
import static org.hamcrest.Matchers.endsWith;
import static org.hamcrest.Matchers.equalToIgnoringCase;
import static org.hamcrest.Matchers.hasProperty;
import static org.hamcrest.Matchers.hasSize;
import static org.hamcrest.Matchers.instanceOf;
import static org.hamcrest.Matchers.is;
import static org.hamcrest.Matchers.isA;
import static org.hamcrest.Matchers.isIn;
import static org.hamcrest.Matchers.notNullValue;
import static org.hamcrest.Matchers.startsWith;
import static org.hamcrest.Matchers.stringContainsInOrder;
import static org.hamcrest.Matchers.emptyCollectionOf;
import static org.junit.Assert.assertThat;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import org.junit.BeforeClass;
import org.junit.Test;

public class JUnitHamcrestTestClass {

        // Creating instances
        private static Employee empA;
        private static Employee empB;
        private static List strList = Arrays.asList("Apple", "Apricot", "August");

        @BeforeClass
        public static void init() {

                // creating objects
                empA = new Employee(1001L, "Vinod Kumar Kashyap", "Male", Arrays.asList(" ↩
                    Best Team", "Star Employee"));
                empB = new Employee(1002L, "Asmi Kashyap", "Female", Arrays.asList("Star  ↩
                    Employee"));
        }

        /**
         * This method will test functionality of `is` matcher.
         */
        @Test
        public void isTest() {
                // checks that empA is an object of Employee class
                assertThat(empA, isA(Employee.class));

                // below are 3 versions of "is" method. All are same
                assertThat(2, equalTo(2));
```

```
                assertThat(2, is(equalTo(2)));
                assertThat(2, is(2));
        }

        /**
         * This method will test functionality of 'beans' matcher.
         */
        @Test
        public void beansTest() {
                // checks that object contains the property
                assertThat(empA, hasProperty("empName"));

                // checks that the object contains the property with a value
                assertThat(empA, hasProperty("empName", equalTo("Vinod Kumar Kashyap")));
        }

        /**
         * This method will test functionality of 'collections' matcher.
         */
        @Test
        public void collectionsTest() {
                // checks that object is of checked size
                assertThat(empA.getAwards(), hasSize(2));

                // checks a collection for the element present
                assertThat("Best Team", isIn(empA.getAwards()));

                // checks for empty collection
                assertThat(new ArrayList(), emptyCollectionOf(String.class));
        }

        /**
         * This method will test functionality of 'String' matcher.
         */
        @Test
        public void stringTest() {
                assertThat(empA.getEmpName(), containsString("Kumar"));
                assertThat(empA.getEmpName(), endsWith("Kashyap"));
                assertThat(empB.getEmpName(), startsWith("Asmi"));

                // checks by ignoring case
                assertThat(empA.getEmpName(), equalToIgnoringCase("vinod KUMAR Kashyap"));

                // checks that the elements are occurring in the same order
                assertThat(empA.getEmpName(), stringContainsInOrder(Arrays.asList("Vinod",  ↩
                    "Kashyap")));
        }

        /**
         * Other common matchers
         */
        @Test
        public void otherCommonTest() {
                // all of the conditions should be met to pass the case
                assertThat(empB.getGender(), allOf(startsWith("F"), containsString("ale"))) ↩
                    ;

                // any of the conditions should be met to pass the case
                assertThat(empB.getEmpName(), anyOf(startsWith("Dhwani"), endsWith("yap"))) ↩
                    ;

                // checks that value is not null
```

```
                assertThat(empA, is(notNullValue()));

                // checks that object id instance of a Class
                assertThat(empA.getEmpId(), instanceOf(Long.class));

                // checks every item in  list
                assertThat(strList, everyItem(startsWith("A")));
        }
}
```

Now we will start with explanation part by part of the example. Most of the matchers are self explanatory.

9.4.1 Is Matcher

This is one of the most common matcher used.

```
@Test
        public void isTest() {
                // checks that empA is an object of Employee class
                assertThat(empA, isA(Employee.class));

                // below are 3 versions of "is" method. All are same
                assertThat(2, equalTo(2));
                assertThat(2, is(equalTo(2)));
                assertThat(2, is(2));
        }
```

Line no 7,8,9 works exactly same. The last form is allowed since `is(T value)` is overloaded to return `is(equalTo(value))`.

9.4.2 Beans Matchers

These matchers are used to check out beans.

```
@Test
        public void beansTest() {
                // checks that object contains the property
                assertThat(empA, hasProperty("empName"));

                // checks that the object contains the property with a value
                assertThat(empA, hasProperty("empName", equalTo("Vinod Kumar Kashyap")));
        }
```

If you see here, we are testing that our class has a property associated with it or not.

9.4.3 Collections Matchers

These matchers works with the collections. JUnit and Hamcrest provides various ways to test collections.

```
@Test
        public void collectionsTest() {
                // checks that object is of checked size
                assertThat(empA.getAwards(), hasSize(2));

                // checks a collection for the element present
                assertThat("Best Team", isIn(empA.getAwards()));

                // checks for empty collection
```

```
                assertThat(new ArrayList(), emptyCollectionOf(String.class));
        }
```

9.4.4 String Matchers

These matchers helps to work with Strings.

```
@Test
        public void stringTest() {
                assertThat(empA.getEmpName(), containsString("Kumar"));
                assertThat(empA.getEmpName(), endsWith("Kashyap"));
                assertThat(empB.getEmpName(), startsWith("Asmi"));

                // checks by ignoring case
                assertThat(empA.getEmpName(), equalToIgnoringCase("vinod KUMAR Kashyap"));

                // checks that the elements are occurring in the same order
                assertThat(empA.getEmpName(), stringContainsInOrder(Arrays.asList("Vinod",  ↩
                    "Kashyap")));
        }
```

9.4.5 Other Common Matchers

There are many different matchers used. Here we are using some of the common matchers. It is also possible to chain matchers, via the `anyOf()` of `allOf()` method. See below for details.

```
@Test
        public void otherCommonTest() {
                // all of the conditions should be met to pass the case
                assertThat(empB.getGender(), allOf(startsWith("F"), containsString("ale"))) ↩
                    ;

                // any of the conditions should be met to pass the case
                assertThat(empB.getEmpName(), anyOf(startsWith("Dhwani"), endsWith("yap"))) ↩
                    ;

                // checks that value is not null
                assertThat(empA, is(notNullValue()));

                // checks that object id instance of a Class
                assertThat(empA.getEmpId(), instanceOf(Long.class));

                // checks every item in  list
                assertThat(strList, everyItem(startsWith("A")));
        }
```

Line no 4 uses `allOf()` matcher. It defines that all of the condition inside should match to pass the test. **Line no 7** uses `anyOf()` matcher, which tells us that if any of the condition is matched, test case will pass. As you see that in our scenario first condition is not matched but second one does. This case passes with flying colors. Because any one of the conditions is true. **Line no 16** scans every item in a list and matches the condition. If condition is matched it will pass the test.

9.5 Output

For running the example simply right click on class and `Run As -> Junit Test`. You will see the following output in JUnit console.

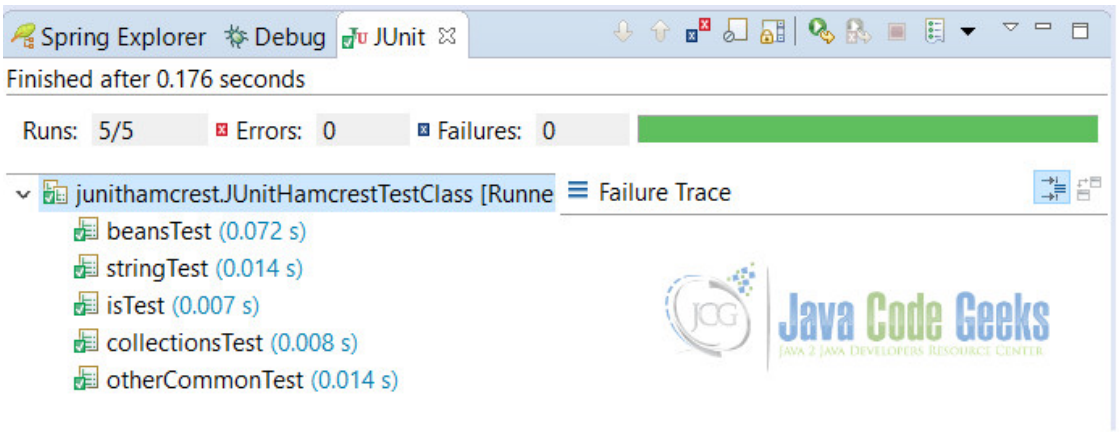

Figure 9.3: JUnit Hamcrest Example Output

9.6 Conclusion

JUnit Hamcrest Example focus mostly on the usage of the JUnit Hamcrest matchers. You have learned what is hamcrest, why we should use it, where it should be used, how to use it. This example shows the usage of a simple example which covers most if the matchers.

9.7 Download the Eclipse project

This is JUnit Hamcrest Example.

Download

You can download the full source code of this example here: **JUnitHamcrest.zip**

Chapter 10

Report Generation Example

In this example we shall show users how we can generate reports using the Maven and JUnit. JUnit Report Generation example demonstrates the basic usage of the reporting functionality of JUnit tests.

As you already know, JUnit is the basic unit test framework for the Java programmers. This example focuses more on generating the report. Let's start by analyzing the ways through which we can generate the HTML reports of our test cases.

10.1 Introduction

JUnit helps us in validation our methods for functionality. But in some cases we have to see the report also for the test cases. In the process of developing reports, Maven plays an important role as it will make a text, XML and also HTML reports. All JUnit test cases with the details are printed in the reports. We will see in this example how this can be achieved.

However, reports can be generated in many different ways like with Ant, TestNG and other independent libraries. But we will focus on very simple case i.e. with the help of Maven.

We will be using the **surefire** plugin of maven to generate the reports for our example.

10.2 Technologies Used

We will be using following technologies to work n this example

- **Java** - primary language for coding
- **Eclipse** - IDE for coding
- **Maven** - dependency management tool and also to generate reports for our test cases.
- **JUnit 4.12** - testing framework

10.3 Project Setup

Open Eclipse. Select `File -> New -> Maven Project`. Following screen will appear. Fill in the values displayed below and then click on Next.

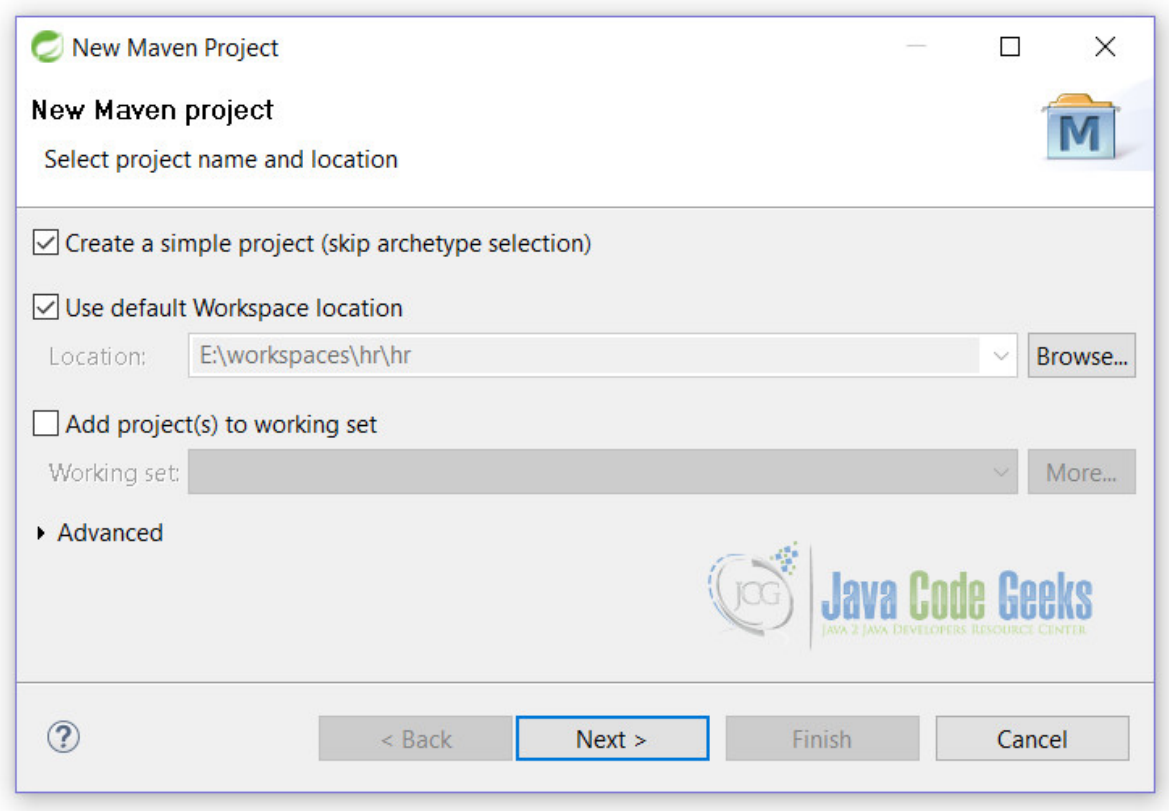

Figure 10.1: Junit Report Generation Example Setup 1

Fill in all the details as shown and click on Finish button.

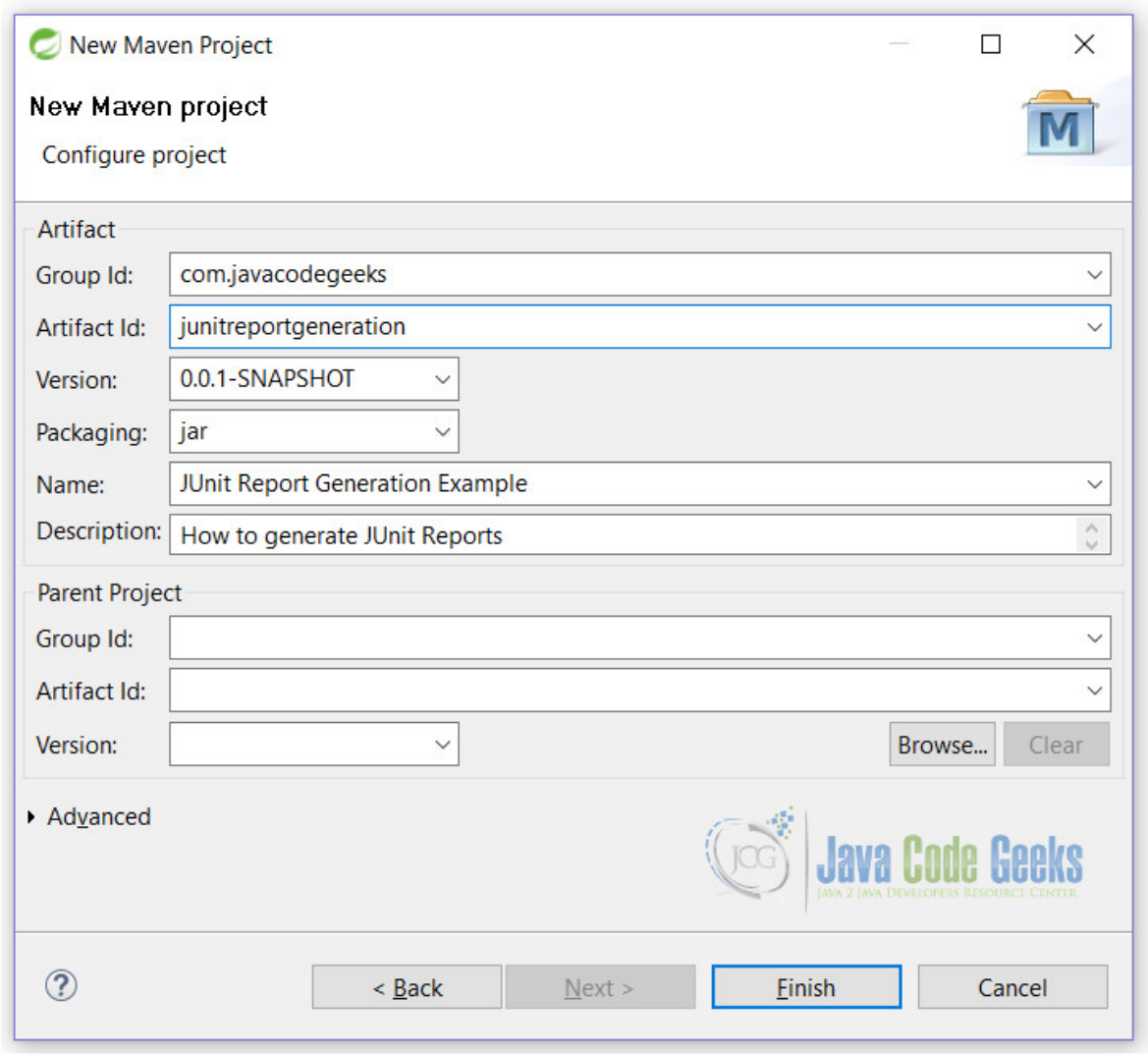

Figure 10.2: Junit Report Generation Example Setup 2

Clicking on Finish button will create a blank Maven project that will be a starting point of our example.

10.4 JUnit Report Generation Example

First of all we need to put the dependencies for our project. Simply put the following line in the `pom.xml` file.

pom.xml

```
<dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
    </dependencies>

    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
```

```
    </properties>

    <reporting>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-report-plugin</artifactId>
                <version>2.19.1</version>
            </plugin>
        </plugins>
    </reporting>
```

Lines 1-7 will download JUnit jar file. Lines 9-12 will ask Maven to use Java 1.8 for this example. Lines 14-22 are used to fetch the surefire plugin that helps us to generate the report for our test cases.

This will ready our project. Let's start creating a unit test case.

10.4.1 JUnit Report Generation Test Class

We will be creating a small test class with only 4 test cases to be tested. By default all test cases will be passed so that our report will be generated.

JUnitReportGenerationTest.java

```
package junitreportgeneration;

import static org.hamcrest.CoreMatchers.instanceOf;
import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertThat;
import static org.junit.Assert.assertTrue;

import java.util.ArrayList;
import java.util.List;

import org.junit.Test;

public class JUnitReportGenerationTest {

        private String developer = "Vinod";

        @Test
        public void instanceOfTest() {
                assertThat(new ArrayList(), instanceOf(List.class));
        }

        @Test
        public void assertTrueTest() {
                assertTrue(true);
        }

        @Test
        public void equalToTest() {
                assertThat(developer, is("Vinod"));
        }

        @Test
        public void failTest() {
                assertThat(developer, is("Any"));
        }
}
```

This is a simple test class.

10.5 Generate Reports

To generate a report you need to simple run the Maven command:

```
mvn clean install test surefire-report:report
```

To run from eclipse you need to follow some steps.

- Right click on project
- Run As → Run Configurations...
- You will be prompted with the screen

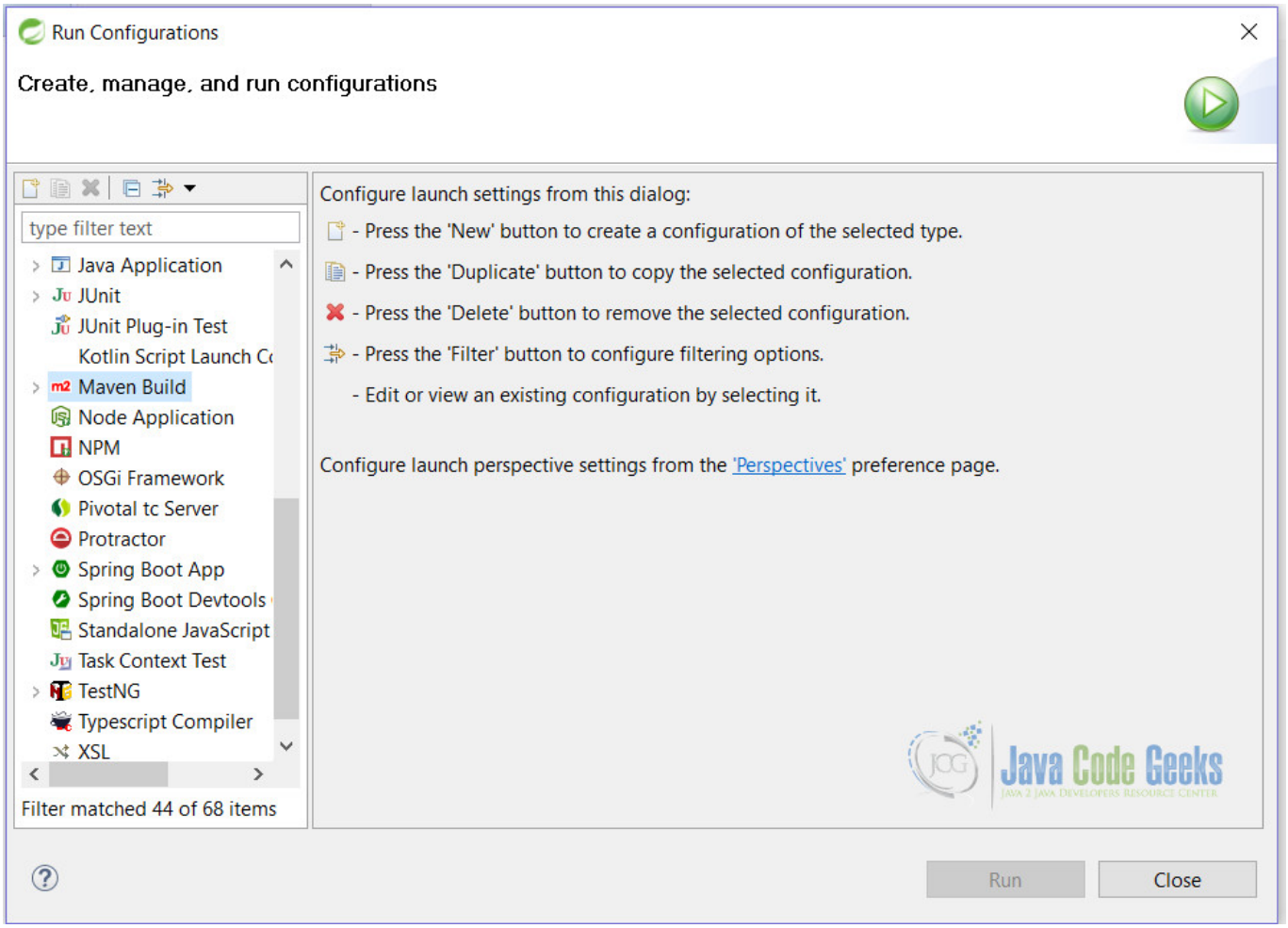

Figure 10.3: Junit Report Generation Example Run 1

- Double Click on Maven Build
- You will be prompted with following screen

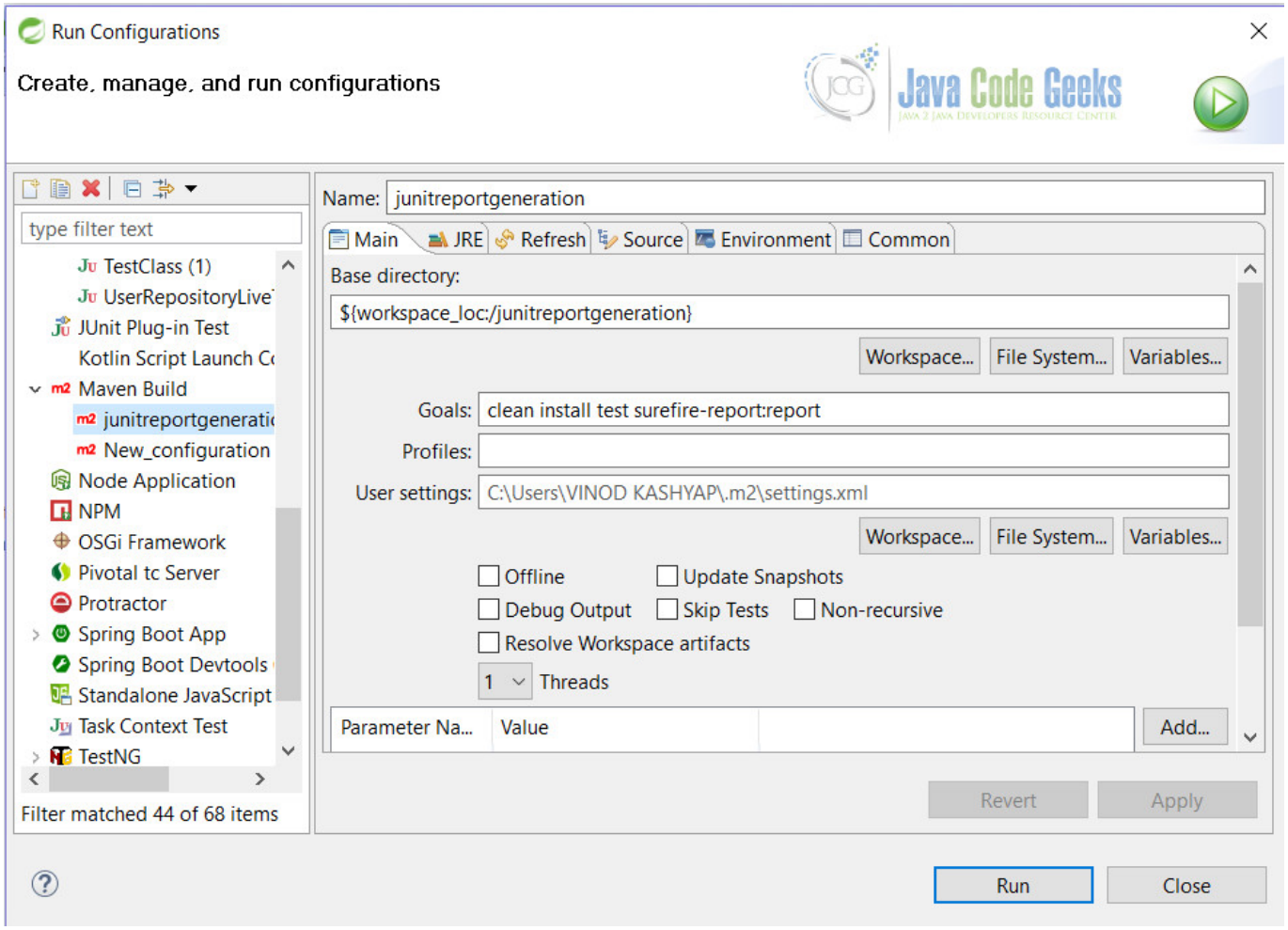

Figure 10.4: Junit Report Generation Example Run 2

- For Base Directory field, Select **Workspace…** button and select your project
- Fill in the details as shown above and click on **Apply** button.
- Now click on **Run** button on same window

You will see the output generated in the target folder.

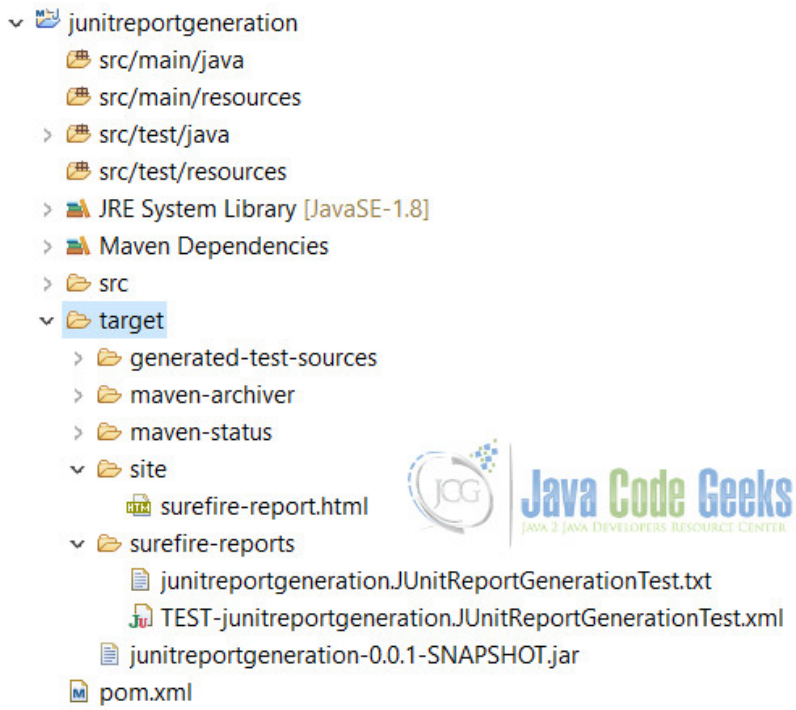

Figure 10.5: Junit Report Generation Example Project Structure

Open `sure-fire.html` file from `target -> site` folder in any browser. You will see the following output.

Last Published: 2017-03-20 | Version: 0.0.1-SNAPSHOT

Built by Maven

Surefire Report

Summary

[Summary] [Package List] [Test Cases]

Tests	Errors	Failures	Skipped	Success Rate	Time
3	0	0	0	100%	0.02

Note: failures are anticipated and checked for with assertions while errors are unanticipated.

Package List

[Summary] [Package List] [Test Cases]

Package	Tests	Errors	Failures	Skipped	Success Rate	Time
junitreportgeneration	3	0	0	0	100%	0.02

Note: package statistics are not computed recursively, they only sum up all of its testsuites numbers.

junitreportgeneration

	Class	Tests	Errors	Failures	Skipped	Success Rate	Time
	TestClass	3	0	0	0	100%	0.02

Test Cases

[Summary] [Package List] [Test Cases]

TestClass

	equalToTest	0.012
	instanceOfTest	0.007
	assertTrueTest	0.001

Figure 10.6: Junit Report Generation Example Output

10.6 Conclusion

Through this example we have learnt that generation of a simple HTML report of JUnit test cases is very simple. We have generated reports with the help of the Maven plugin `surefire`.

10.7 Download the Eclipse Project

This is a JUnit Report Generation Example.

Download

You can download the full source code of this example here: **JUnitReportGeneration.zip**