# Creating Generics and Testsuites

In the previous module, we started off with the representation of the [sample project hierarchy and various framework components](http://www.softwaretestinghelp.com/selenium-framework-design-selenium-tutorial-21/). We also discussed about the data source – “excels” used to store the test data and their excel manipulations. We also discussed about the new strategies and resources to mature our framework.

Now we are moving ahead with advanced topics in this [Selenium Training](http://www.softwaretestinghelp.com/selenium-tutorial-1/) Series. In this session, we would take the opportunity to discuss two important concepts that plays an important role to mature the framework. We would discuss the **concept of Generics and reusability aspects**. We would also discuss about **creation and significance of Test suite**.

For better understanding we would accompany the concepts with adequate examples and sample code.

**Generics**

**By the literal notion, a generic is something that can act as a descriptive of an entire group or classes.**

While automating applications, we come across various end to end scenarios. An end to end scenario may consist of several trivial functionality. Thus, many of these functionality can act as common functionality to more than one test script with slight or almost no modifications.

Hence, it is advisable to create a generic class consisting of methods that can be claimed as common and can be shared among multiple test scripts instead of implementing the same code again and again for multiple test scripts.

Take a note that generics also introduce the power of reusability in our framework. Reusability reduces time taken for code, errors, bugs, maintenance etc. exceptionally.

**Type of Generics**

**#1) Application Specific**

The meager functionality belonging to application under test can become a part of the Application Specific generics. Take the Login Functionality for instance. Login is one such functionality that can be a fragment of almost all the test scripts. Thus, instead of writing the login code all over again in the test scripts, we would create a common method in the generic class and call it wherever needed.

**#2) Framework Specific**

Aside from Application specific generics, we may have common methods which do not directly relate with the application under test but are part of the chosen framework. Consider an Excel reading functionality when we have employed Test Data Driven Framework. It would make no sense if we would write the code for reading excels again and again in all the test scripts. Therefore, we induce the code once in the generic class and make a call to it whenever required.

**Creation of Generic Class**

User is leveraged to create as many generic classes as he/she desires based on the modularity infused.

Let us understand the concept of generic by creating one.

**Step 1:** Create a new java class “CommonMethods.java” that would act as a generic class consisting of common methods preferably in the package other than where test scripts reside.

**Step 2:** The next step is to copy and paste the below code in the “CommonMethods.java” generic class. Number of common methods can be implemented inside the periphery of this class. Below is the code snippet for login functionality.

|  |  |
| --- | --- |
| 1 | /\*\* |
| 2 | \* Login the Test application | |

|  |  |
| --- | --- |
| 3 | \* |
| 4 | \* @param username | |

|  |  |  |
| --- | --- | --- |
| 5 | \* @param password | |
| 6 | \*/ |

|  |  |  |
| --- | --- | --- |
| 7 | public void login(String username, String password) { | |
| 8 | try { |

|  |  |  |
| --- | --- | --- |
| 9 | // Enter User Name | |
| 10 | | WebElement userName = driver.findElement(By.id("loginID")); | |

|  |  |
| --- | --- |
| 11 | userName.clear(); |
| 12 | userName.sendKeys(username); | |

|  |  |
| --- | --- |
| 13 | // Enter Password |
| 14 | WebElement passWord = driver.findElement(By.id("Password")); | |

|  |  |
| --- | --- |
| 15 | passWord.clear(); |
| 16 | passWord.sendKeys(password); | |

|  |  |
| --- | --- |
| 17 | // Click on the Sign In Button |
| 18 | WebElement signin = driver.findElement(By.id("SignIn\_button")); | |

|  |  |
| --- | --- |
| 19 | signin.click(); |
| 20 | driver.manage().window().maximize(); | |

|  |  |  |
| --- | --- | --- |
| 21 | } catch (Exception e) { | |
| 22 | e.printStackTrace(); |

|  |  |
| --- | --- |
| 23 | } |
| 24 | } |

Take a note that the aforementioned method is a parameterized method. Thus, the same method can be used to test the login functionality with different sets of test data.

**Step 3:** The next step is to call the common method within the test script. The process is a two step process. First we create the instance of the generic class within the test class and then we call the common method on the created instance by passing the required arguments. In the code fragment below, we created an instance of “TestScript1.java” class and called the login () method to login the application.

|  |  |  |
| --- | --- | --- |
| 1 | // Create Object of the generic class | |
| 2 | toolsObj = new Tools(); |

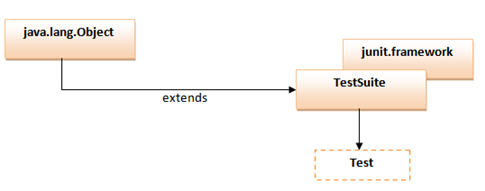
|  |  |  |
| --- | --- | --- |
| 3 | // Login the test application by calling the common method | |
| 4 | preTestObj.login(“username”, “password”); |

Take a mention that the above code can be placed anywhere inside the test class. User can place the code in the setup () method or in the test () method.

**Testsuite**

Test suite is an assortment of more than one test script grouped together for execution purpose. Thus, test suite executes the number of specified test scripts unattended. Test suite has the capability to cite the test scripts to be executed automatically; all that is required from the user is to mark an entry each for the individual test script within the test suite. The entry is supposed to be the “class name” of the test script with “.class” extension or simply the compiled form of our java class.

**Below is the sample Test suite created in java.** Take a note that Test suite is a java based class that belongs to the family of JUnit. Thus, you may encounter several JUnit annotations in the code.

[](http://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2014/11/Selenium-test-suite.jpg)

------------

**Code Snippet**

|  |  |  |
| --- | --- | --- |
| 1 | package com.axway.webliv.tests; | |
| 2 | import org.junit.AfterClass; |

|  |  |
| --- | --- |
| 3 | import org.junit.BeforeClass; |
| 4 | import org.junit.runner.JUnitCore; | |

|  |  |
| --- | --- |
| 5 | import org.junit.runner.Result; |
| 6 | import org.junit.runner.RunWith; | |

|  |  |  |
| --- | --- | --- |
| 7 | import org.junit.runner.notification.Failure; | |
| 8 | import org.junit.runners.Suite; |

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | import com.axway.webliv.tests.MetaData.\*; | | |
| 10 | |  |

|  |  |
| --- | --- |
| 11 | @RunWith(Suite.class) |
| 12 | @Suite.SuiteClasses({ |

|  |  |
| --- | --- |
| 13 |  |
| 14 | ChangeStatusBinarySphereTest.class, | |

|  |  |  |
| --- | --- | --- |
| 15 | ChangeStatusRestrictionTest.class, | |
| 16 | ChangeStatusDocSphereTest.class, |

|  |  |  |
| --- | --- | --- |
| 17 | }) | |
| 18 |  |

|  |  |  |
| --- | --- | --- |
| 19 | public class TestSuite { | |
| 20 | /\*\* |

|  |  |  |
| --- | --- | --- |
| 21 | \* Setup method to set system properties | |
| 22 | \*/ |

|  |  |
| --- | --- |
| 23 | @BeforeClass |
| 24 | public static void Setup() { | |

|  |  |
| --- | --- |
| 25 | } |
| 26 | /\*\* | |

|  |  |  |
| --- | --- | --- |
| 27 | \* @param args | |
| 28 | \*/ |

|  |  |
| --- | --- |
| 29 | public static void main(String[] args) { |
| 30 | Result result = JUnitCore.runClasses(TestSuite.class); | |

|  |  |
| --- | --- |
| 31 | System.out.println("TEST CASES RUN: " + result.getRunCount()); |
| 32 | System.out.println("TEST CASES FAILED: " + result.getFailureCount()); |

|  |  |
| --- | --- |
| 33 | for (Failure failure : result.getFailures()) { |
| 34 | System.out.println("\nTEST NAME: " + failure.getTestHeader()); | |

|  |  |  |
| --- | --- | --- |
| 35 | System.out.println("\nERROR: " + failure.getMessage() + "\n"); | |
| 36 | System.out.println(failure.getTrace()); |

|  |  |  |
| --- | --- | --- |
| 37 | System.exit(1); | |
| 38 | } |

|  |  |
| --- | --- |
| 39 | } |
| 40 | /\*\* | |

|  |  |  |
| --- | --- | --- |
| 41 | \* Report test results | |
| 42 | \*/ |

|  |  |
| --- | --- |
| 43 | @AfterClass |
| 44 | public static void TearDown() { | |

|  |  |  |
| --- | --- | --- |
| 45 | } | |
| 46 | } |

**Code Walk-Through**

A test suite is none other than a simple JUnit class having *setup()*and *teardown()* methods; the one’s we discussed at length in our preceding modules. The only remarkable difference lies in its competence to execute more than one test script in a single go.

**Import Statements**

|  |  |
| --- | --- |
| 1 | import org.junit.AfterClass; |
| 2 | import org.junit.BeforeClass; | |

|  |  |  |
| --- | --- | --- |
| 3 | import org.junit.runner.JUnitCore; | |
| 4 | import org.junit.runner.Result; |

|  |  |
| --- | --- |
| 5 | import org.junit.runner.RunWith; |
| 6 | import org.junit.runner.notification.Failure; | |

|  |  |
| --- | --- |
| 7 | import org.junit.runners.Suite; |

The above import statements are embedded in the class to be able to use various annotations provided by the JUnit.

*import org.junit.runner.JUnitCore;*  
*import org.junit.runner.RunWith;*  
*import org.junit.runners.Suite;*

The above statements possess the underlying architecture to execute the test suite consisting of multiple test classes.

*import org.junit.runner.Result;*

The import statement allows the user to store the test execution statuses and their manipulations.

*import org.junit.AfterClass;*  
*import org.junit.BeforeClass;*

These import statements are used to identify and annotate *setup()*and *teardown()* methods. The *setup()* method annotated with BeforeClass instructs the program control to execute the method before each of the test script execution. Like *setup()*, *teardown()*method annotated with AfterClass tells the program control to execute the method after each of the test script execution.

**Class Entry**

|  |  |  |
| --- | --- | --- |
| 1 | @RunWith(Suite.<strong>class</strong>) | |
| 2 | @Suite.SuiteClasses({ |

|  |  |  |
| --- | --- | --- |
| 3 | ChangeStatusBinarySphereTest.<strong>class</strong>, | |
| 4 | ChangeStatusRestrictionTest.<strong>class</strong>, |

|  |  |  |
| --- | --- | --- |
| 5 | ChangeStatusDocSphereTest.<strong>class</strong>, | |
| 6 | }) |

This section of the class allows the user to mark the entries of the test script to be executed in the next run. Remember the entries are marked with “.class” extension i.e. in their compiled formats.

**Execution – main ()**

|  |  |
| --- | --- |
| 1 | public static void main(String[] args) { |
| 2 | Result result = JUnitCore.runClasses(TestSuite.class); | |

|  |  |
| --- | --- |
| 3 | System.out.println("TEST CASES RUN: " + result.getRunCount()); |
| 4 | System.out.println("TEST CASES FAILED: " + result.getFailureCount()); | |

|  |  |
| --- | --- |
| 5 | for (Failure failure : result.getFailures()) { |
| 6 | System.out.println("\nTEST NAME: " + failure.getTestHeader()); | |

|  |  |  |
| --- | --- | --- |
| 7 | System.out.println("\nERROR: " + failure.getMessage() +"\n"); | |
| 8 | System.out.println(failure.getTrace()); |

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | System.exit(1); | | |
| 10 | | } |

|  |  |
| --- | --- |
| 11 | } |

This part of the code deals with the execution. The program execution always initiates from the main().

The runClasses method of JUnitCore class is used to execute the test suite. The “Result class” and its methods are used to determine the execution status in terms of Passed and Failed test cases.

Thus, user is leveraged to play around with the test suite class to be able to suffice his/her requirements.

**Conclusion**

In this module, we tried to make you acquainted with the concept of generics and common methods. We also discussed the benefits we get out generics like reusability. We also shared the practical approaches towards creation of generics and their accessibility.

**Here are the cruxes of this article:**

* Generic is something that can act as a descriptive of an entire group or classes. Generic in our framework is a class that is solely consists of methods those can be shared across multiple test classes.
* Generics can be classified into two categories:
  + Application Specific
  + Framework Specific
* A simple java class can be created to act as a Generic. Number of common methods can be implemented inside the generic class. These methods can be parameterized methods.
* The common methods can be accessed by calling them on the instance of generic class within the test scripts.
* Test suite is an assortment of more than one test script grouped together for execution purpose. Thus, test suite executes the number of specified test scripts unattended.
* A test suite is none other than a simple JUnit class having*setup()* and *teardown()* methods; the one’s we discussed at length in our preceding modules. The only remarkable difference lies in its competence to execute more than one test script in a single go.