1. **what is the use of framework in automation testing**

     Ans:  Framework is nothing but it is separate project contains packages/classes/methods.

              generally framework contains reusable components/code

              Ex: in framework i created classes/methods to handle exceptions/custom reports, reading data from database, excel file

             It will be a jar file in java

             It will be DLL in C#

              in automation code we add this jar/dll’s (using buildpath->add external jar files option) and call appropriate classes/method Whenever it is required

     Ex: selenium jar file is framework and we add this file jar file in our project

           We use classes/methods of selenium framework (WebDriver, Webelement classes, get method)

1. **what are things u consider/keep in framework**

     Ans: Create classes/methods to handle exceptions/custom reports, reading data from database, excel file

1. **different types of frameworks we have in selenium/qtp**

     Ans: Data driven, hybrid, keyword driven, BDD

#### The Data-Driven Testing Framework.

In this Framework , while Test case logic resides in Test Scripts, the Test Data is separated and kept outside the Test Scripts.Test Data is read from the external files (Excel Files, Text Files, CSV Files, ODBC Sources, DAO Objects, ADO Objects) and are loaded into the variables inside the Test Script. Variables are used both for Input values and for Verification values. Test Scripts themselves are prepared either using Linear Scripting or Test Library Framework.

**The Hybrid Test Automation Framework.**

As the name suggests this framework is the combination of one or more frameworks discussed above pulling from their strengths and trying to mitigate their weaknesses. This hybrid test automation framework is what most frameworks evolve into over time and multiple projects. Maximum industry uses Keyword Framework in combination of Function decomposition method.

**The Keyword-Driven** or Table-Driven framework requires the development of data tables and keywords, **independent of the test automation tool**used to execute them . Tests can be designed with or without the Application. In a keyword-driven test, the functionality of the application-under-test is documented in a table as well as in step-by-step instructions for each test.

There are 3 basis components of a Keyword Driven Framework

* Keyword
* Application Map ,
* Component Function.

1. **what is data driven frame work and when do we use it?**

     To test same functionality more than once with different set of data/input data

         We store the test data in external files (excel/csv) or table.

             Data Driven Testing Framework is a framework which is driven by test data, which means that test data is the important factor here. **The basic expectation with this kind of test automation framework is that your test scripts should be built in such a way that it should work for different sets of data without any changes to the test script**.

1. **what is POM and modular framework**

## Framework Methodologies

There are four popular test automation methodologies :

1. Modular  
2. Keyword  
3. Datadriven  
4. Hybrid.

## Modular Framework:

In this methodology, is one of the most basic type of automation framework. In this framework, a test scripts are written to match a functionality that represent modules of the application-under-test. These modules in turn are used in a hierarchical fashion to build large test cases.

**Advantages:**

Modular division of scripts leads to easier maintenance and also the scalability of the automated test Scripts are independent to write.

**Disadvantages:**

The main problem with modular frameworks is that the test script have test data embedded in them, which will become problem when updating the code /script.

It is also difficult when ever a test step fails which has to be find out by debugging where the test case failed.

1) Create a ‘[**New Package**](http://toolsqa.wpengine.com/selenium-webdriver/configure-eclipse-with-selenium-webdriver/#package)‘ file and name it as ‘appModule**’**, by right click on the Project and select New > Package. We will be creating different packages for Page Objects, Utilities, Test Data, Test Cases and Modular actions. It is always recommended to use this structure, as it is easy to understand, easy to use and easy to maintain.

2) Create ‘[New Class](http://toolsqa.wpengine.com/selenium-webdriver/configure-eclipse-with-selenium-webdriver/#class)‘ and name it as SignIn\_Action by right click on package ‘appModule‘ and select New > Class**.** It will add new class ‘SignIn\_Action’ under package ‘appModule’.

3) Now create a Public Static VoidMethod and name it as Execute and club the following steps in to it:

* Click on the My Account link.
* Enter Username
* Enter Password
* Click on the Submit button

This method will not have any Argumen**t** (driver) and Return value as it is a voidmethod.

**what is testng and junit?**

TestNG is a testing framework that overcomes the limitations of another popular testing framework called JUnit. The "NG" means "Next Generation"

## Benefits of TestNG

There are number of benefits but from Selenium perspective, major advantages of TestNG are :

1. It gives the ability to produce ***HTML Reports*** of execution
2. ***Annotations*** made testers life easy
3. Test cases can be ***Grouped & Prioritized*** more easily
4. ***Parallel***testing is possible
5. Generates ***Logs***
6. Data ***Parameterization***is possible

## Test Case Writing

Writing a test in TestNG is quite simple and basically involves following steps:

**Step 1** – Write the business logic of the test

**Step 2** – Insert TestNG annotations in the code

**Step 3** – Add the information about your test (e.g. the class names, methods names, groups names etc…) in a testng.xml file

**Step 4** – Run TestNG

## Annotations in TestNG

**@BeforeSuite**: The annotated method will be run before all tests in this suite have run.

**@AfterSuite**: The annotated method will be run after all tests in this suite have run.

**@BeforeTest**: The annotated method will be run before any test method belonging to the classes inside the tag is run.

**@AfterTest**: The annotated method will be run after all the test methods belonging to the classes inside the tag have run.

**@BeforeGroups**: The list of groups that this configuration method will run before. This method is guaranteed to run shortly before the first test method that belongs to any of these groups is invoked.

**@AfterGroups**: The list of groups that this configuration method will run after. This method is guaranteed to run shortly after the last test method that belongs to any of these groups is invoked.

**@BeforeClass**: The annotated method will be run before the first test method in the current class is invoked.

**@AfterClass**: The annotated method will be run after all the test methods in the current class have been run.

**@BeforeMethod**: The annotated method will be run before each test method.

**@AfterMethod**: The annotated method will be run after each test method.

**@Test**: The annotated method is a part of a test case.  
 

## Benefits of using Annotations

1. It identifies the methods it is interested in by looking up annotations. Hence method names are not restricted to any pattern or format.
2. We can pass additional parameters to annotations.
3. Annotations are strongly typed, so the compiler will flag any mistakes right away.
4. Test classes no longer need to extend anything (such as Test Case, for JUnit 3).
5. **writing unit test cases with testing**

TestNG is a testing framework for unit test development. jmockit is a framework for mock objects that provides mock object functionality using the java.lang.instrument package of jdk 1.5. Together, these frameworks can provide the tools to create very robust test cases without design limitations of other testing frameworks currently available

* To begin, create a new java project and call it *TestNGTutorial*.
* Right click on your new project and select*New --> Folder*. Name it *lib* and click Finish.
* Usually you don't want to package your test code with your regular code, so let's make an additional source directory, *test*. To do that, right click on your new project and select *Source Folder*.
* Next we need to add TestNG to our build path. To make our life easier and since we are serious about unit testing ;) we will use the TestNG plugin for Eclipse. It includes a test runner and other functionality that will make things easier for test development. To install the plugin, in Eclipse, go to *Help --> Software Updates --> Find and Install...*   
  From there, select "Search for new feature to install" and click "Next>". In the upper right hand corner, click on "New Remote Site". Enter "*TestNG Plugin*" as the Name and *http://beust.com/eclipse* as the URL and click OK. Make sure TestNG is the only thing with a check next to it and click Finish. In the new window that is displayed, check the box next to testng and click Next>. Click Next> and then Click Finish.
* Once the download is complete, Eclipse will ask you what to install. Click Install All. Once the install is complete, restart Eclipse.
* Once Eclipse is back, right click on your project and click Properties. Select Java Build Path from the left and click on the Libraries tab. On the right, click Add Variable and select TESTNG\_HOME. Click the Extend... button to the right and select /lib/testng-jdk15.jar and click Ok. Click OK to exit the Properties window.

We now need to get jmockit. This is the mock framework we will be using , extract the jmockit.jar file from the archive and place it in the lib folder we created previously.

unit testing on the following Java program, which uses static methods to perform arithmetic operations on two integers. Take note that divide throws an IllegalArgumentException for divisor of zero.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | /\*\*  \* The Calculator class provides static methods for  \* arithmetic operations on two integers.  \*/  public class Calculator {  public static int add(int number1, int number2) {  return number1 + number2;  }    public static int sub(int number1, int number2) {  return number1 - number2;  }    public static int mul(int number1, int number2) {  return number1 \* number2;  }    // Integer divide. Return a truncated int.  public static int divInt(int number1, int number2) {  if (number2 == 0) {  throw new IllegalArgumentException("Cannot divide by 0!");  }  return number1 / number2;  }    // Real number divide. Return a double.  public static double divReal(int number1, int number2) {  if (number2 == 0) {  throw new IllegalArgumentException("Cannot divide by 0!");  }  return (double) number1 / number2;  }  } |

##### First Test Case

Let's do it under Eclipse.

1. Create a new Eclipse Java project called "JUnitTest".
2. Create a new class called "Calculator" under "src" folder, with the above program code.
3. Create a new folder called "test" for storing test scripts ⇒ Right-click on the project ⇒ New ⇒ Folder ⇒ In folder name, enter "test". Make "test" a source folder by right-click on "test" ⇒ Build Path ⇒ Use as source folder.
4. Create the first test case called "AddSubTest" ⇒ Right-click on folder "test" ⇒ New ⇒ Other ⇒ Java ⇒ JUnit ⇒ JUnit Test Case ⇒ New JUnit 4 test ⇒ In Name, enter "AddSubTest". Enter the following codes:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | import static org.junit.Assert.\*;  import org.junit.Test;    public class AddSubTest {  @Test  public void testAddPass() {  // assertEquals(String message, long expected, long actual)  assertEquals("error in add()", 3, Calculator.add(1, 2));  assertEquals("error in add()", -3, Calculator.add(-1, -2));  assertEquals("error in add()", 9, Calculator.add(9, 0));  }    @Test  public void testAddFail() {  // assertNotEquals(String message, long expected, long actual)  assertNotEquals("error in add()", 0, Calculator.add(1, 2));  }    @Test  public void testSubPass() {  assertEquals("error in sub()", 1, Calculator.sub(2, 1));  assertEquals("error in sub()", -1, Calculator.sub(-2, -1));  assertEquals("error in sub()", 0, Calculator.sub(2, 2));  }    @Test  public void testSubFail() {  assertNotEquals("error in sub()", 0, Calculator.sub(2, 1));  }  } |

1. To run the test case, right-click on the file ⇒ Run as ⇒ JUnit Test. The test result is shown in the JUnit panel. 4 tests were run and all succeeded. Study the test results.
2. Try modify one of the test to force a test failure and observe the test result, e.g.,
3. @Test
4. public void testAddPass() {
5. assertEquals("error in add()", 0, Calculator.add(1, 2));
6. .....

}

1. **what are important elements in testng.xml**
2. what are different annotations in testng and junit
3. **what is group and suite and parallel execution in testing**

Group test is a new innovative feature in TestNG, which doesn’t exist in JUnit framework. It permits you to dispatch methods into proper portions and perform sophisticated groupings of test methods. Not only can you declare those methods that belong to groups, but you can also specify groups that contain other groups. Then, TestNG can be invoked and asked to include a certain set of groups (or regular expressions), while excluding another set. Group tests provide maximum flexibility in how you partition your tests and doesn't require you to recompile anything if you want to run two different sets of tests back to back.

1. **how to decide which test cases needs to be automated**

Generally, we automate to avoid repeated manual work, get faster feedback, save time on running tests over and over again, and ensure we are always executing tests consistently with the same preconditions and expectations

1. **what are the steps in automation testing or when do we automation testing?**

      Ans: write manual test cases (if they are not there)

               out of all manual test cases we will pick test cases that are of regression testing

                       (test cases that needs to be executed)

               put all positive scenarios in regression

               we dont put any UI related changes in regression testing

**15) what is the use of dataprovider annotation**

In TestNG, the **DataProvider feature** helps to write data-driven tests, which essentially means that **same test method can be run multiple times with different data-sets**.  It helps in providing complex parameters to the test methods as it is not possible to do this from XML. To use DataProvider feature in tests we have to declare a method annotated by @DataProvider and then use the said method in the test method using the ‘**dataProvider**‘ attribute in the Test annotation.

**16) Difference between junit and testing**

JUnit and TestNG are both very popular unit test framework in Java. However, TestNG's goal is much wider, which includes not only unit testing, but also support of integration and acceptance testing, etc

**Junit** is a unit testing framework while **TestNG** addresses testing at a higher level.

Test prioritization, Parallel testing is possible in TestNG. It is not supported by JUnit.

**Annotations:**  
In JUnit, the @BeforeClass and @AfterClass methods have to be declared as static. TestNG does not have this constraint.

**TestNG** has provided four additional setup/teardown pairs for the suite, test and groups, i.e. @BeforeSuite, @AfterSuite, @BeforeTest, @AfterTest, @BeforeGroup and @AfterGroup, @BeforeMethod and @AfterMethod.

**Parameterized test:**  
This feature is implemented in both tools, however in quite different ways.

TestNG has basically two ways for providing varying parameter values to a test method: by setting the testng.xml, and by defining a @DataProvider method.

In **JUnit**, @RunWith and @Parameters are used together to facilitate parameterized tests, while the @Parameters method has to return List[] with all the actual values, which will be fed into a dedicated class constructor as an argument.

     17) how to generate reports using testng

**1.1- STEPS TO GENERATE REPORTS USING TESTNG.**

Step-1**)** TestNG supplies two interfaces to implement reporting in Selenium Webdriver. You can apply any of these two in your projects.

1. Testers commonly use the <*[ITestListener](http://testng.org/javadocs/org/testng/ITestListener.html" \t "_blank)*> Interface.
2. Another is the <*IReporter*> Interface which is least popular.

In this blog post, we’ll cover the application of the <***ITestListener***> Interface. You’ll have to create a TestNG project in Eclipse. You can refer our below post that teaches to create a basic TestNG project.

Step-2) Generate Reports Using the <ITestListener> Interface**.**

GenerateReport.Java - Implementation of the ITestListener Interface.

Java

|  |  |
| --- | --- |
| 1 | import org.testng.ITestContext;  import org.testng.ITestListener;  import org.testng.ITestResult;    public class GenerateReport implements ITestListener {    @Override  public void onStart(ITestContext arg0) {    System.out.println("+Begin test: " + arg0.getName());    }    @Override  public void onTestStart(ITestResult arg0) {    System.out.println(" Starting test: " + arg0.getName());    }    @Override  public void onTestSuccess(ITestResult arg0) {    System.out.println(" Test passed: " + arg0.getName());    }    @Override  public void onTestFailure(ITestResult arg0) {    System.out.println(" Test failed: " + arg0.getName());    }    @Override  public void onTestSkipped(ITestResult arg0) {    System.out.println(" Test ignored: " + arg0.getName());    }    @Override  public void onFinish(ITestContext arg0) {    System.out.println("-End test: " + arg0.getName());    }    @Override  public void onTestFailedButWithinSuccessPercentage(ITestResult arg0) {    // TODO Auto-generated method stub    }    } |

**Step-3) Verify the Report Generation Process.**

VerifyReportTest.Java - Verify if the test code is producing the report.

Java

|  |  |
| --- | --- |
|  | import org.testng.Assert;  import org.testng.annotations.Listeners;  import org.testng.annotations.Test;    @Listeners(GenerateReport.class)  public class VerifyReportTest {    @Test  public void testSimulation0() {    Assert.assertTrue(true);    }    @Test  public void testSimulation00() {    Assert.assertTrue(false);    }    // Test case <testSimulation000> depends on the intentionally  // failed test case <testSimulation00>    @Test(dependsOnMethods = "testSimulation00")  public void testSimulation000() {    }    } |