5: @Test annotation can be used above test method or above Test Class

6: If a test class contains multiple test method to execute all test method we can write @Test annotation

above Test Class or for every Test Method within that class.

Script:

```
package testNG;

import org.testng.annotations.Test;

    public class Script_2
    {
        @Test
        public void CreateAccount()
        {
                System.out.println("Create Account");
        }
        @Test
        public void EditAccount()
        {
                System.out.println("Edit Account");
        }
        @Test
        public void DeleteAccount()
        {
                System.out.println("Delete Account");
        }

}
```

Output:

```
Create Account
Delete Account
Edit Account
```

7: If a test class contain multiple test method that method execution order will be alphabethical

8: To change method execution order we use Priority keyword

9: priority can be:

  A: Default value is 0

  B: Either +ve or -ve integer

  C: Can be duplicate

10: Priority cannot be

  1: Decimal values

  2: Variables

Script:

```
package testNG;

import org.testng.annotations.Test;

public class Script_3
{
	@Test(priority=0)
	public void CreateAccount()
	{
		System.out.println("Create Account");
	}
	@Test(priority=1)
	public void EditAccount()
	{
		System.out.println("Edit Account");
	}
	@Test(priority=2)
	public void DeleteAccount()
	{
		System.out.println("Delete Account");
	}

}
```

11: To execute any test method multiple times, we use invocationCount keyword

12: Default value of invocation count is 1.

Script:

```
@Test(invocationCount=3)
	public void CreateAccount()
	{
		System.out.println("CreateAccout");
	}
```

13: To skip any test method execution we use enabled=false

Script:

```
package testNG;

import org.testng.annotations.Test;

public class Script_2
{
	@Test
	public void CreateAccount()
	{
		System.out.println("Create Account");
	}
	@Test(enabled=false)
	public void EditAccount()
	{
		System.out.println("Edit Account");
	}
	@Test
	public void DeleteAccount()
	{
		System.out.println("Delete Account");
	}

}
```

14: dependsOnMethods: This keyword is used to make test method execution depend on other test method execution

Script:

```
package testNG;

import org.testng.annotations.Test;

public class Script_5
{
	@Test
	public void Contact()
	{
		System.out.println("Contact Added");
	}
	@Test(dependsOnMethods="Contact")
	public void Chat()
	{
		System.out.println("Chat");
	}

}
```

15: dependsOnGroups(): This keyword is used to make a method to execute only if group is passed

Script:

```
package testNG;

import org.testng.annotations.Test;

public class Script_6
{
	@Test(dependsOnGroups="Sample")
	public void test1()
	{
		System.out.println("Test 1");
	}
	@Test(groups="Sample")
	public void test2()
	{
		System.out.println("Test 2");
	}
	@Test(groups="Sample")
	public void test3()
	{
		System.out.println("Test 3");
	}
	@Test(groups="Sample")
	public void test4()
	{
		System.out.println("Test 4");
	}
	@Test(groups="Sample")
	public void test5()
	{
		System.out.println("Test 5");
	}

}
```

16: alwaysRun=true: This method is used to execute a method irrespective of dependent method status

Script:

```
package testNG;

import org.testng.Assert;
import org.testng.annotations.Test;

public class Script_7
{
	@Test
	public void Contact()
	{
		Assert.fail();
		System.out.println("Contact Added");
	}
	@Test(dependsOnMethods="Contact",alwaysRun=true)
	public void Chat()
	{
		System.out.println("Chat");
	}



}
```

17: Batch Execution/ Test Suite:

-->It is an xml file which contains all the test classes that need to be executed

**Procedure to create Test Suite

-->Select TestNG Package--->Right Click-->TestNG-->Convert To testNG--->Click on finish

Result: testNG.xml file will be created

**Procedure to execute Test Suite

-->Select TestNG.xml-->Right Click-->Run as-->testNG Suite

```
package testNG;

import org.testng.annotations.Test;

public class Script_8
{
	@Test
	public void test1()
	{
		System.out.println("Test 1");
	}
	@Test
	public void test2()
	{
		System.out.println("Test 2");
	}
	@Test
	public void test3()
	{
		System.out.println("Test 3");
	}
	@Test
	public void test4()
	{
		System.out.println("Test 4");
	}
	@Test
	public void test5()
	{
		System.out.println("Test 5");
	}


}
```

-->Convert above Test Class to Test Suite file

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">
<suite name="Suite">
  <test thread-count="5" name="Test">
    <classes>
      <class name="testNG.Script_8">
      <methods>
      <exclude name="test2"></exclude>
      </methods>
      </class>
    </classes>
  </test> <!-- Test -->
</suite> <!-- Suite -->
```

-->Above modify test suite to skip test2 method execution

--->Belowe modify test suite to include test1 and test2

```
<suite name="Suite">
  <test thread-count="5" name="Test">
    <classes>
      <class name="testNG.Script_8">
      <methods>
      <include name="test1"></include>
      <include name="test2"></include>
      </methods>
      </class>
    </classes>
  </test> <!-- Test -->
</suite> <!-- Suite -->
```

***TestNG Annotations:

1: @Test: This method with annotation will be considered as test method

2: @BeforeMethod: This annotation is used to execute a method BEFORE every test method execution

3: @AfterMethod: This annotation is used to execute a method AFTER every test method execution

4: @BeforeClass: This annotation is used to execute a method BEFORE test class execution

5: @AfterClass: This annotation is used to execute a method AFTER test class execution

6: @BeforeTest: This annotation is used to execute a method BEFORE suite level test tag execution

7: @AfterTest: This annotation is used to execute a method AFTER suite level test tag execution

8: @BeforeSuit: This annotation is used to execute a method BEFORE suite execution

9: @AfterSuit: This annotation is used to execute a method AFTER suite execution

@BeforeSuite

@BeforeTest

@BeforeClass

@BeforeMethod

@Test

@AfterMethod

@AfterClass

@AfterTest

@AfterSuite

Script:

```
package testNG;

import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;



public class Annotation
{
	@BeforeMethod
	public void Login()
	{
		System.out.println("Login to app");
	}

	@Test
	public void AddUser()
	{
```

```
                System.out.println("User Added");
        }

        @Test(priority=1)
        public void ModifyUser()
        {
                System.out.println("User Details Modified");
        }

        @Test(priority=2)
        public void DeleteUser()
        {
                System.out.println("User Deleted");
        }

        @AfterMethod
        public void Logout()
        {
                System.out.println("Logout from App");
        }


}
```

output:

```
Login to app
User Added
Logout from App
Login to app
User Deleted
Logout from App
```

Login to app

User Details Modified

Logout from App

**Inheritance in TestNG

->Every test class will contain common navigation

-->All the common navigation will be automated under Base Class or

SuperTestNG which need to be inherited to Sub Classes

-->Super class methods needs to be executed either before sub class method execution or after sub class

method execution

-->So we use @BeforeClass and @AfterClass annotation for base class methods.

Script:

```
package testNG;

import org.testng.annotations.AfterClass;

import org.testng.annotations.BeforeClass;

public class BaseClass

{

        @BeforeClass

        public void Preconditions()

        {

                System.out.println("Open Browser");

        }

        @AfterClass

        public void Postconditions()

        {
```

```
        System.out.println("Close browser");
    }


}
package testNG;

import org.testng.annotations.Test;

public class BaseClass1 extends BaseClass
{
    @Test
    public void Signup()
    {
        System.out.println("Application Signup");
    }


}
```

Parametrization Using TestNG*

1: Executing test method by using multiple inputs.

2: In TestNG, to achieve parametrization that is to pass values for method arguments we use following annotations:

1: @DataProvider

2: @parameters

A: @DataProvider: This annotation is used to pass data for test method from test class level

-->Procedure to perform parametrization using data provider:

1: create method with return type two dimensional array object @DataProvider annotation

2: Within that method create two dimensional array object & store data

Syntax: Object[][] rv= new Object[m][n];

-->m represents no. of times test method need to be executed

-->n represents no. of inputs that need to be pass for each time method execution

-->Test Method will receive data from data provider and executes

Script:

```
    @DataProvider
        public Object[][] data()
        {
                Object[][] rv = new Object[3][2];
                rv[0][0]="admin1";
                rv[0][1]="manager1";
                rv[1][0]="admin2";
                rv[1][1]="manager2";
                rv[2][0]="admin3";
                rv[2][1]="manager3";
                return rv;

        }
        @Test(dataProvider="data")
        public void Login(String user,String pass)
        {
                System.out.println(user);
                System.out.println(pass);
        }
```

Limitations:

1: We cant handle huge data using data provider

2: one data provider method can support one test method

3: Data Provider two dimensional array object size is fixed.

2: @parameters: This annotation is used to execute test method by passing value from suite level.

-->We can execute test method only one time

-->Values should be written in @Parameters annotations

Ques: WATS to login facebook app by fetching data from suite file

Script:

```
    public class Script_10
     {
        @Parameters({"email","pass"})
        @Test
        public void testLogin(String email,String pass)
        {
                WebDriver driver = new FirefoxDriver();
                driver.get("https://www.facebook.com/");
                driver.findElement(By.id("email")).sendKeys(email);
                driver.findElement(By.id("pass")).sendKeys(pass);
        }
```

Suite File:

```
<suite name="Suite">
  <test thread-count="5" name="Test">
  <parameter name="email" value="abc@gmail.com"></parameter>
  <parameter name="pass" value="1234"></parameter>
  <classes>
  <class name="testNG.Script_10"></class>
  </classes>
  </test> <!-- Test -->
</suite> <!-- Suite -->
```

***TestNG Verification:

-->To do verification we use if else condition which will increase test script length

-->In TestNG, we use Assert/Hard Assert class static methods for verification.

1:assertEquals(): This method is used to verify expected and actual result.

-->If both results are same, verification is passed and test method will be pass else fail.

Script:

```
	@Test
	public void test()
	{
		String str1="hii";
		String str2="hello";
		Assert.assertEquals(str1, str2);
```

2: assertNotEquals(): This method is used to verify expected and actual result.

-->If both results are not same, verification is passed and test method will be pass else fail.

Script:

```
@Test
	public void test()
	{
		String str1="hii";
		String str2="hello";
		Assert.assertNotEquals(str1, str2);
	}
```

3: assertTrue(): This method is used to verify condition is true or false

--->if it is true then verification is pass

4: assertFalse(): This method is used to verify condition is true or false

--->if it is false then verification is pass

5: assertNull(): This method is used to verify element is empty or not

-->if it is empty then verification is pass

6: assertNotNull(): This method is used to verify element is empty or not

-->if it is not empty then verification is pass

7: fail(): This method is used to fail test method execution

Script:

```
package testNG;

import org.testng.Assert;
import org.testng.annotations.Test;

public class Script_11
{
	@Test
	public void test1()
	{
		String str1="hii";
		String str2="hello";
		Assert.assertEquals(str1, str2);
	}
	@Test(dependsOnMethods="test1")
	public void test2()
	{
		System.out.println("hii");
	}
	@Test
	public void test3()
	{
		System.out.println("hiiiiii");
	}

}
```

**Limitations of Hard Assert:

-->In a test method, multiple varifications are existing.

-->If one of the verification is failed then other verifications execution in that method will be skipped

-->To overcome above limitation we use Soft Assert non static methods for verification

**SoftAssert

-->It is a class which contain nonstatic methods for verification

-->SoftAssert will execute complete test method irrespective of verification status.

-->To notify fail verification in test method last cmd should be assertAll()

Script:

```
      @Test
	public void test()
	{
		WebDriver driver = new FirefoxDriver();
		driver.get("https://www.saucedemo.com/");
		SoftAssert soft = new SoftAssert();
		soft.assertTrue(driver.getTitle().equals("xyz"));
		driver.findElement(By.id("user-name")).sendKeys("standard_user");
		driver.findElement(By.id("password")).sendKeys("secret_sauce");
		driver.findElement(By.id("login-button")).click();
		soft.assertAll();
```

Ques: Diff. between Hard Assert & Soft Assert

HardAssert	SoftAssert
1: It is static in nature	1: It is non static in nature
2: If verifications fails it will stop the execution	2: If verification fails it will continue the execution
3: assertAll() is not mandatory.	3: assertAll() is mandatory.

AssignQues: WATS to login facebook application using testNG