Parameterisation, DataProvider and Factory

This lesson explains in detail how to parameterize a test and use @Parameter, @DataProvider and @Factory annotations effectively for testing purposes.

`@Parameters`

Tests can be parameterized using `@Parameter`. They can be passed from ***testng.xml***. Parameters can be set at `suite` level or `test` level.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="Sample Test Suite">
    <parameter name="browser" value="chrome" />
    <test name="Sample Test">
        <parameter name="browser" value="chrome" />
    </test>
</suite>
```

Here in the above ***testng.xml***, we set the parameter **browser** to **chrome** at the test suite level. The same parameter can be overridden at the test level as well, as shown.

To use parameter in the test method, follow the following approach.

```
@Parameters("browser")
@Test
public void testMethod(String browser) {
```

```
public void testMethod(String browser) {

}
```

For configuration methods like `@BeforeXXX` and `@AfterXXX`, follow the following approach.

```
@Parameters("browser")
@BeforeMethod
public void initMethod(String browser) {

}
```

`@DataProvider`

`@DataProvider` is for running a test case multiple times by passing different sets of parameters.

Creating `@DataProvider` method

`@DataProvider` can be created like either of two ways given below:

```
@DataProvider( name = "createData" )
public Object[][] createTestData() {

    Object[][] data = new Object[2][2];
    data[0] = new Object[] { "Sam", 21 };
    data[1] = new Object[] { "Smith", 22 };

    return data;
}
```

```
@DataProvider( name = "createData" )
public Iterator<Object[]> createTestData() {

    List<Object[]> data = new ArrayList<Object[]>();
    data.add(new Object[] { "Sam", 21 });
    data.add(new Object[] { "Smith", 22 });

    return data.iterator();
}
```

In the above code snippet, we are creating a data set for 2 test runs. In the first run, the test method with 2 parameters - `String` and `int`, will be injected with

"Sam" and *21* respectively. Likewise, during the second run, values *"Smith"* and *22* will be passed.

When a `name` attribute is not provided for a `@DataProvider` annotated method, by default, the name of the method is taken.

In addition, the return type of a `@DataProvider` can be custom data types also like `CustomClass[]` or `Iterator<CustomClass>`

By default, the `@DataProvider` will be looked upon in the same class as the test method. If the `@DataProvider` is present elsewhere, the method needs to be declared `public static`. The following code snippet explains this.

```
public class DataProviderFactory {

    @DataProvider( name = "createData" )
    public static Iterator<Object[]> createTestData() {

        List<Object[]> list = new ArrayList<Object[]>();
        list.add(new Object[] { "Sam", 21 });
        list.add(new Object[] { "Smith", 22 });

        return list.iterator();
    }

}
```

```
public class TestClass {

    @Test(dataProvider = "createData", dataProviderClass = DataProviderFac
tory.class)
    public void testMethod(String name, int age) {

    }
}
```

Using `@DataProvider` in test method

When using `@DataProvider`, we should ensure the order and type of the parameters list that is given as an argument to the test method.

If the test method and data provider are in the same class.

```
@Test(dataProvider = "createData")
```

```
public void testMethod(String name, int age) {
}
```

If the test method and data provider are in different classes, use:

```
@Test(dataProvider = "createData", dataProviderClass = MyDataProviderClass
.class)
public void testMethod(String name, int age) {
}
```

`@DataProvider` tests can be made to run in parallel by setting `@DataProvider(parallel = true)` and thread count can be controlled from ***testng.xml*** using property `data-provider-thread-count="10"`.

```
<suite name="Sample Test Suite" parallel="tests" thread-count="5" data-pro
vider-thread-count="10">
```

`@Factory`

`@Factory` is for creating test classes at runtime.

```
public class TestClass {

    private String browser;

    public TestClass(String browser) {
        this.browser = browser;
    }

    @Test
    public void testMethod() {
    }

    @Factory
    public Object[] createTestClasses() {

        Object[] data = new Object[2];
        data[0] = new TestClass("chrome");
        data[1] = new TestClass("firefox");
        return data;
    }

}
```

In the above code snippet, we are creating a `@Factory` method that creates two instances of the test class with parameters - for the first run with *“chrome”* and for the second run with *“firefox”*

The default constructor is required when the data provider method is not static.

```
public class TestClass {

    private String browser;

    public TestClass() {
    }

    @Factory(dataProvider = "data")
    public TestClass(String browser) {
        this.browser = browser;
    }

    @Test
    public void testMethod() {
    }

    @DataProvider
    public Iterator<Object[]> data() {

        List<Object[]> arr = new ArrayList<>();
        arr.add(new Object[] { "firefox" });
        arr.add(new Object[] { "chrome" });
        return arr.iterator();
    }

}
```

As we know already, if the `@DataProvider` annotated method is not present in the same class, we need to provide `@Factory(dataProviderClass = DataProviderFactory.class, dataProvider = "createData")`, and the `@DataProvider` annotated method needs to be `public static`.

Now that you are familiar with the details related to the major annotations, in the next lesson, you will learn about the use of different listeners.