SWE 4604

Software Testing and Quality Assurance

Lab 3

Prepared By Maliha Noushin Raida, Lecturer, CSE
Islamic University of Technology

Advanced JUnit Topics

In last lab we understood the basic usage of the Junit framework, now we will moving on to some of the advance topics.

Covered Topics:

- Introduction to Junit
- Junit annotations
- Junit execution sequence
- Junit assertions

Test Case Execution Order

For Junit 4,

@FixMethodOrder and MethodSorters.class supporting the facility of setting an order for a test’s execution.

@FixMethodOrder:

@FixMethodOrder(MethodSorters.DEFAULT): FIxed order of the test cases

Or, @FixMethodOrder(MethodSorters.JVM): No fixed order, every time changes its order

Or, @FixMethodOrder(MethodSorters.NAME_ASCENDING): Fixed order, according to the the test method name.

Test Case Execution Order

For Junit 5,

The annotations @TestMethodOrder and @Order supporting the facility of setting an order for a test’s execution.

@TestMethodOrder:

- @TestMethodOrder(Alphanumeric.class): Sorted according to test method name. Have fixed order in every execution.
- @TestMethodOrder(OrderAnnotation.class): Allows the @Order annotation to be place before the test method and explicitly mention the order.
- @TestMethodOrder(Random.class): Random order execution, order changes in each run.

Test Case Execution Order

@TestMethodOrder(Alphanumeric.class)

```
@TestMethodOrder(Alphanumeric.class)
public class JUnit5TestOrder {
@Test
        public void Testcase_3() {
                System.out.println("Testcase_3 executes");
        }
    @Test
    public void Testcase_1() {
                System.out.println("Testcase_1 executes");
        }
    @Test
     public void Testcase_2() {
                System.out.println("Testcase_2 executes ");
} }
```

Test Case Execution Order

@TestMethodOrder(Random.class)

```
@TestMethodOrder (Random.class)
public class JUnit5TestOrder {
@Test
        public void Testcase_3() {
                System.out.println("Testcase_3 executes");
        }
     @Test
     public void Testcase_1() {
                System.out.println("Testcase_1 executes");
        }
     @Test
      public void Testcase_2() {
                System.out.println("Testcase_2 executes ");
} }
```

Test Case Execution Order

@TestMethodOrder(OrderAnnotation.class)

```
@TestMethodOrder(OrderAnnotation.class)
public class JUnit5TestOrder {

@Test
                @Order(1)
        public void Testcase_3() {
                System.out.println("Testcase_3 executes");
        }

     @Test
                @Order(2)
                public void Testcase_1() {
                System.out.println("Testcase_1 executes");
        }
     @Test
@Order(3)
      public void Testcase_2() {
                System.out.println("Testcase_2 executes ");
}
}
```

Parameterized Junit Testing

What's parameterized test cases? A parameterized test executes the same test multiple times with different arguments.

Five steps to follow to create a parameterized test for **JUnit 4**:

- Annotate test class with @RunWith(Parameterized.class).
- Create a public static method annotated with @Parameterized.Parameters that returns a Collection of Objects (as Array) as test data set.
- Create a public constructor that takes in what is equivalent to one "row" of test data.
- Create an instance variable for each "column" of test data.
- Create your test case(s) using the instance variables as the source of the test data.

For example, see the attached PrimeNumberCheckerTestJUNIT4.java file

Parameterized Junit Testing

Steps to create the parameterized tests for **Junit 5**,

- Annotate test cases with @ParameterizedTest
- Next annotation indicated the source of the test data, these can be, @ValueSource, @NullSource, @EmptySource, @MethodSource, @CSVFileSource, @CSVSource and so on.

```
@ParameterizedTest
@ValueSource(strings = { "X", "Y" })
@EnumSource(value = TimeUnit.class, names = {"HOURS", "DAYS"})
@MethodSource("stringProvider")
@CsvSource({ "A, 1", "'B, C', 2" })
@CsvFileSource(resources = "/two-column.csv")
@ArgumentsSource(MyArgumentsProvider.class)
void parameterizedTest(Object argument) {
    assertNotNull(argument);
}
```

For example, see the attached PrimeNumberCheckerTestJUNIT5.java file

Mocking

Mocking is the act of removing external dependencies from a unit test in order to create a controlled environment around it. Typically, we mock all other classes that interact with the class that we want to test.

Common targets for mocking are:

- Database connections,
- Web services,
- Classes that are slow,
- Classes with side effects, and
- Classes with non-deterministic behavior.

Mockito

Mockito is the most popular framework used for Mocking in JUnit and Java.

The benefits or advantages of Mockito are explained in the following points:

- No handwriting - The developers do not need to write their Mock codes.
- Return values - Mockito supports the return values.
- Safe Refactoring - Even if an interface method is renamed or the parameters are reordered, the test codes created as Mocks will not break.
- Exception support - Mockito enables exceptions.
- Annotation support - Mockito supports the creation of Mocks with annotation.

Mockito

To add the mockito to your java project, use maven package mockito-core 4.11.0 and include the following code in pom.xml file,

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>4.11.0</version>
    <scope>test</scope>
</dependency>
```

Mockito

Annotations:

- @Mock is used for mock creation. It makes the test class more readable. Or you can use mock(<class>) method.
- @Spy is used to create a spy instance. We can use it instead of the spy(Object) method.
- @InjectMocks is used to instantiate the tested object automatically and inject all the @Mock or @Spy annotated field dependencies into it (if applicable).

Mockito

@Mock or mock():

```
@Mock
List<String> mockedList;

@Test
public void whenUseMockAnnotation_thenMockIsInjected() {
    mockedList.add("one");
    Mockito.verify(mockedList).add("one");
    assertEquals(0, mockedList.size());

    Mockito.when(mockedList.size()).thenReturn(100);
    assertEquals(100, mockedList.size());
}
```

```
@Test
public void whenNotUseMockAnnotation_thenCorrect() {
    List mockList = Mockito.mock(ArrayList.class);

    mockList.add("one");
    Mockito.verify(mockList).add("one");
    assertEquals(0, mockList.size());

    Mockito.when(mockList.size()).thenReturn(100);
    assertEquals(100, mockList.size());
}
```

Mockito

@Spy

```java
@Test
public void whenNotUseSpyAnnotation_thenCorrect() {
    List<String> spyList = Mockito.spy(new ArrayList<String>());

    spyList.add("one");
    spyList.add("two");

    Mockito.verify(spyList).add("one");
    Mockito.verify(spyList).add("two");

    assertEquals(2, spyList.size());

    Mockito.doReturn(100).when(spyList).size();
    assertEquals(100, spyList.size());
}
```

```java
@Spy
List<String> spiedList = new ArrayList<String>();

@Test
public void whenUseSpyAnnotation_thenSpyIsInjectedCorrectly() {
    spiedList.add("one");
    spiedList.add("two");

    Mockito.verify(spiedList).add("one");
    Mockito.verify(spiedList).add("two");

    assertEquals(2, spiedList.size());

    Mockito.doReturn(100).when(spiedList).size();
    assertEquals(100, spiedList.size());
}
```

Mockito

@InjectMock

```java
@Mock
Map<String, String> wordMap;

@InjectMocks
MyDictionary dic = new MyDictionary();

@Test
public void whenUseInjectMocksAnnotation_thenCorrect() {
    Mockito.when(wordMap.get("aWord")).thenReturn("aMeaning");

    assertEquals("aMeaning", dic.getMeaning("aWord"));
}
```

```java
public class MyDictionary {
    Map<String, String> wordMap;

    public MyDictionary() {
        wordMap = new HashMap<String, String>();
    }
    public void add(final String word, final String meaning) {
        wordMap.put(word, meaning);
    }
    public String getMeaning(final String word) {
        return wordMap.get(word);
    }
}
```

Mockito

Most Commonly use function of Mockito,

- when(…).thenReturns(…): Mocks can return different values depending on arguments passed into a method. The when(….).thenReturn(….) method chain is used to specify a return value for a method call with pre-defined parameters. Example:

```
    @Mock
    Database databaseMock;

    @Test
    void ensureMockitoReturnsTheConfiguredValue() {

        // define return value for method getUniqueId()
        when(databaseMock.getUniqueId()).thenReturn(42);

        Service service = new Service(databaseMock);
        // use mock in test....
        assertEquals(service.toString(), "Using database with
id: 42");
    }
```

Mockito

- Verify the calls on the mock objects: Mockito keeps track of all the method calls and their parameters to the mock object. **verify()** method can be used on the mock object to verify that the specified conditions are met.

For example, see the ToDoBusinessMock2.java
For extra material, use this link,
https://www.baeldung.com/mockito-verify

Mockito

- doReturn(…).when(…): This method configuration can be used to configure the reply of a mocked method call. This is similar to when(….).thenReturn(….). doReturn can be useful if you are using spy object.

```
@Test
void ensureSpyForListWorks() {
    var list = new ArrayList<String>();
    var spiedList = spy(list);

    doReturn("42").when(spiedList).get(99);
    String value = (String) spiedList.get(99);

    assertEquals("42", value);
}
```