#### **Annotations for Junit testing**

The Junit 4.x framework is annotation based, so let's see the annotations that can be used while writing the test cases.

**@Test** annotation specifies that method is the test method.

**@Test(timeout=1000)** annotation specifies that method will be failed if it takes longer than 1000 milliseconds (1 second).

**@BeforeClass** annotation specifies that method will be invoked only once, before starting all the tests.

**@Before** annotation specifies that method will be invoked before each test.

**@After** annotation specifies that method will be invoked after each test.

**@AfterClass** annotation specifies that method will be invoked only once, after finishing all the tests.

## Assert class

The org.junit.Assert class provides methods to assert the program logic.

#### Methods of Assert class

The common methods of Assert class are as follows:

1. **void assertEquals(boolean expected,boolean actual)**: checks that two primitives/objects are equal. It is overloaded.
2. **void assertTrue(boolean condition)**: checks that a condition is true.
3. **void assertFalse(boolean condition)**: checks that a condition is false.
4. **void assertNull(Object obj)**: checks that object is null.
5. **void assertNotNull(Object obj)**: checks that object is not null.

#### **Required jar files**

You need to load **junit4.jar** and **hamcrest-core.jar** files.

MOCKITO

Mockito is a mocking framework. It is a Java-based library used to create simple and basic test APIs for performing unit testing of Java applications. It can also be used with other frameworks such as **JUnit** and **TestNG**.

What is Mocking?

Mocking is a process of developing the objects that act as the **mock** or **clone** of the real objects. In other words, mocking is a testing technique where mock objects are used instead of real objects for testing purposes. Mock objects provide a specific (dummy) output for a particular (dummy) input passed to it.

To mock objects, you need to understand the three key concepts of mocking, i.e., stub, fake, and mock. Some of the unit tests involve only stubs, whereas some involve fake and mocks.

The brief description of the mocking concepts is given below:

1. **Stub:** Stub objects hold predefined data and provide it to answer the calls during testing. They are referred to as a dummy object with a minimum number of methods required for a test. It also provides methods to verify other methods used to access the internal state of a stub, when necessary. Stub object is generally used for **state verification**.
2. **Fake:** Fake are the objects that contain working implementations but are different from the production one. Mostly it takes shortcuts and also contains the simplified version of the production code.
3. **Mock:** Mock objects act as a dummy or clone of the real object in testing. They are generally created by an open-source library or a mocking framework like Mockito, EasyMock, etc. Mock objects are typically used for **behavior verification**.

## What is Mockito?

Mockito is a Java-based mocking framework used for unit testing of Java application. Mockito plays a crucial role in developing testable applications. It internally uses the Java Reflection API to generate mock objects for a specific interface. Mock objects are referred to as the dummy or proxy objects used for actual implementations.

# **Methods of Mockito**

The Mockito framework provides a variety of methods such as mock(), verify(), when(), etc., used to test Java applications. Using these predefined methods makes testing very easy.

|  |  |
| --- | --- |
| mock() | t is used to create mock objects of a given class or interface. Mockito contains five **mock()** methods with different arguments. When we didn't assign anything to mocks, they will return default values. All five methods perform the same function of mocking the objects. |
| when() | It enables stubbing methods. It should be used when we want to mock to return specific values when particular methods are called. |
| verify() | The **verify()** method is used to check whether some specified methods are called or not. In simple terms, it validates the certain behavior that happened once in a test. It is used at the bottom of the testing code to assure that the defined methods are called. |
| spy() | Mockito provides a method to partially mock an object, which is known as the **spy** method. When using the spy method, there exists a real object, and spies or stubs are created of that real object. If we don't stub a method using spy, it will call the real method behavior. The main function of the spy() method is that it overrides the specific methods of the real object. One of the functions of the spy() method is it verifies the invocation of a certain method. |
| doThrow() | It is used when to stub a void method to throw an exception. It creates a new exception instance for each method invocation. |
| doNothing() | It is used for setting void methods to do nothing. The doNothing() method is used in rare situations. By default, the void methods on mock instances do nothing, i.e., no task is performed. |
|  |  |
|  |  |

## Mocking Example

A screenshot of a computer program

Description automatically generated

Now, create an implementation class named **ToDoBusiness** for **ToDoService** interface.

|  |
| --- |
| 1. **public** **class** ToDoBusiness { 3. **public** ToDoService doService; 5. **public** ToDoBusiness(ToDoService doService) { 6. **this**.doService = doService; 7. } 9. **public** List<String> getTodosforHibernate(String user) 10. {   List<String> hibernatelist = **new** ArrayList<String>(); 11. List<String> Combinedlist = doService.getTodos(user); 13. **for**(String todo: Combinedlist) { 14. **if**(todo.contains("Hibernate")) { 15. hibernatelist.add(todo); 16. } 17. } 19. **return** hibernatelist; 20. } 21. } |
| 1. Create a JUnit test case named **ToDoBusinessMock** for unit testing. 2. **public** **class** ToDoBusinessMock { 4. @Test 5. **public** **void** testusing\_Mocks() { 7. ToDoService doService = mock(ToDoService.**class**); 9. List<String> combinedlist = Arrays.asList(" Use Core Java ", " Use Spring Core ", " Use w3eHibernate ", " Use Spring MVC "); 10. when(doService.getTodos("dummy")).thenReturn(combinedlist); 12. ToDoBusiness business = **new** ToDoBusiness(doService); 14. List<String> alltd = business.getTodosforHibernate("dummy"); 16. System.out.println(alltd); 17. assertEquals(1, alltd.size()); 18. } 19. } |

## Example of mocking a List class

Create a mock test class named **TestList** for testing the List class.

|  |
| --- |
| 1. **public** **class** TestList { 3. @Test 4. **public** **void** testList\_ReturnsSingle\_value() { 6. List mocklist = mock(List.**class**); 7. when(mocklist.size()).thenReturn(1); 9. assertEquals(1, mocklist.size()); 10. assertEquals(1, mocklist.size()); 12. System.out.println( mocklist.size()); 13. System.out.println(mocklist); 14. } 15. } |
| **Mocking a list.get method**   1. @Test 2. **public** **void** testList\_get() { 4. List mocklist = mock(List.**class**); 6. when(mocklist.get(0)).thenReturn("Mockito"); 8. assertEquals("Mockito", mocklist.get(0)); 9. System.out.println(mocklist.get(0)); |
|  |

**USING VERFIY**

A screenshot of a computer program

Description automatically generatedImplementation class

|  |
| --- |
| 1. **public** **class** ToDoBusiness { 2. **public** ToDoService doService; 3. **public** ToDoBusiness(ToDoService doService) { 4. **this**.doService = doService; 5. } 6. **public** **void** deleteTodosNotRelatedToHibernate(String user) { 7. List<String> Combinedlist = doService.getTodos(user); 8. **for**(String todos:Combinedlist) { 9. **if**(!todos.contains("Hibernate")) { 10. doService.deleteTodos(todos); |

Now make a mock class and verify if method delete tododisinvoked or not

|  |
| --- |
| 1. **public** **class** ToDoBusinessMock { 2. @Test 3. **public** **void** deleteTodosusing\_BDD() { 4. ToDoService todoService = mock(ToDoService.**class**); 5. List<String> combinedlist = Arrays.asList("Use Hibernate Java", "Use Hibernate Core", "Use Hibernate", "Use Spring MVC"); 6. given(todoService.getTodos("dummy")).willReturn(combinedlist); 7. ToDoBusiness business = **new** ToDoBusiness(todoService); 8. business.deleteTodosNotRelatedToHibernate("dummy"); 9. verify(todoService).deleteTodos("Use Spring MVC"); 10. } 11. } |

Using spy

|  |
| --- |
| 1. **public** **class** SpyTest { 2. @Test 3. **public** **void** test() { 4. List spyArrayList = spy(ArrayList.**class**); 5. assertEquals(0, spyArrayList.size()); 6. spyArrayList.add("Mockito"); 7. assertEquals(1, spyArrayList.size()); 8. } 9. } |

# **Argument Matchers**

Argument matchers are mainly used for performing flexible verification and stubbing in Mockito. It extends **ArgumentMatchers** class to access all the matcher functions. Mockito uses **equal()** as a legacy method for verification and matching of argument values. In some cases, we need more flexibility during the verification of argument values, so we should use argument matchers instead of **equal()** method. The ArgumentMatchers class is available in **org.mockito** package.

The ArgumentMatchers class contains a variety of methods; some of them are listed below:

|  |  |
| --- | --- |
| **Method type and method name** | **Description** |
| <T> any() | It matches all values (anything), including null values and varargs. |
| boolean anyBoolean() | It matches to any boolean or not-null boolean values. |
| byte anyByte() | It matches to any byte or not-null byte values. |
| char anyChar() | It matches to any char or not-null character values. |
| Collection <T> anyCollection | It matches any not-null collection in the application. |
| double anyDouble() | It matches to any double or not-null double values. |
| float anyFloat() | It matches to any float or not-null float values. |
| int anyInt() | It matches to any int or not-null integer values. |
| Iterable<T> anyIterable() | It matches to any int or not-null integer values. |
| Iterable<T> anyIterable() | It matches to any not-null iterable values. |
| List<T> anyList() | It matches any not-null list. |
| long anyLong() | It matches to any long or not-null long values. |
| Set<T> anySet() | It matches any not-null set. |
| short anyShort() | It matches to any short or not-null short values. |
| String anyString() | It matches any not-null String. |
| <T> argThat(ArgumentMatcher<T> matcher) | It allows the creation of custom argument matchers. |
| boolean booleanThat(ArgumentMatcher<Boolean> matcher) | It allows the creation of the custom boolean argument matchers. |
| byte byteThat(ArgumentMatcher<Byte> matcher) | It allows the creation of custom byte argument matchers. |
| char charThat(ArgumentMatcher<Character> matcher) | It allows the creation of custom char argument matchers. |
| String contains(String substring) | It matches the String argument that contains the substring. |
| double doubleThat(ArgumentMatcher<Double> matcher) | It allows the creation of custom double argument matchers. |
| String endsWith(String suffix) | It matches the String argument that ends with the given suffix. |
| boolean eq(boolean value) | It matches with the boolean argument that is equal to the given value. |
| double eq(double value) | It matches with the double argument that is equal to the given value. |
| long eq(long value) | It matches the long argument that is equal to the given value. |
| <T> isNotNull() | It matches with the not null argument. |
| <T> is Null() | It matches the null argument. |
| <T> same(T value) | It checks whether the object argument is the same as the given value. |
|  |  |
|  |  |

**Example of using a list an dmocking its get method**

**public** **class** TestList {

1. @Test
2. **public** **void** testList\_Argument\_Matchers() {
4. List<String> mocklist = mock(List.**class**);
6. when(mocklist.get(Mockito.anyInt())).thenReturn("Mockito");
8. assertEquals("Mockito", mocklist.get(0));
9. assertEquals("Mockito", mocklist.get(1));
10. assertEquals("Mockito", mocklist.get(2));
11. }

### **Example of throwing an exception using Argument matcher**

1. **public** **class** TestList {
3. @Test(expected = RuntimeException.**class**)
4. **public** **void** testList\_ThrowsException() {
6. List<String> mocklist = mock(List.**class**);
8. when(mocklist.get(Mockito.anyInt())).thenThrow(**new** RuntimeException("Error.."));
9. mocklist.get(0);
10. }
11. }

# **Hamcrest Matchers**

**Hamcrest** is a popular framework that help us to create the matcher objects. It is used for writing software tests and also performs unit testing in Java programming language. Hamcrest is mainly used with other unit testing frameworks like **JUnit, jMockit, Mockito,** etc.

1. **public** **class** HamcrestMockito {
3. @Test
4. **public** **void** test() {
6. List<Integer> num = Arrays.asList(99,123,45,148,33);
8. assertThat( num, hasSize(5));
9. assertThat( num, hasItems(99,45));
11. assertThat( num, everyItem(greaterThan(20)));
12. assertThat( num, everyItem(greaterThanOrEqualTo(33)));
13. assertThat( num, everyItem(lessThan(150)));
14. }

# **Mockito Annotations**

The Mockito framework provides a variety of annotations to make the code simple and easy to understand. Also, it reduces the lines of code that helps in focusing on the business logic. In Mockito, annotations are useful when we want to use the mocked object at different places to avoid calling the same methods multiple times.

The Mockito annotations are given below:

* **@Mock:** It is used to mock the objects that helps in minimizing the repetitive mock objects. It makes the test code and verification error easier to read as parameter names (field names) are used to identify the mocks. The @Mock annotation is available in the **org.mockito** package.  
  Following code snippet shows how to use the @mock annotation:

1. @Mock
2. ToDoService servicemock;

#### **Note: The @Mock annotation is always used with the @RunWith annotation.**

* **@RunWith:** It is a class-level annotation. It is used to keep the test clean and improves debugging. It also detects the unused stubs available in the test and initialize mocks annotated with @Mock annotation. The @RunWith annotation is available in the **org.mockito.junit** package.  
  Following code snippet shows how to use the @RunWith annotation:

1. @RunWith(MockitoJUnitRunner.**class**)
2. **public** **class** ToDoBusinessMock {
3. .....
4. }

n the above code snippet, the **MockitoJUnitRunner** class is used to check that all the mocks are created and autowired when needed.

* **@InjectMocks:** It marks a field or parameter on which the injection should be performed. It allows shorthand mock and spy injections and minimizes the repetitive mocks and spy injection. In Mockito, the mocks are injected either by setter injection, constructor injection, and property injection. The @InjectMocks annotation is available in the **org.mockito** package.  
  Following code snippet shows how to use the @InjectMocks annotation:

1. @InjectMocks
2. ToDoBusiness business;

* **@Spy -** It allows the creation of partially mock objects. In other words, it allows shorthand wrapping of the field instances in a spy object. Like other annotations, @Spy annotation is also available in the **org.mockito** package.  
  Following code snippet shows how to use the @Spy annotation:

1. @Spy
2. ArrayList<String> arraylistSpy;

### **Example of Mockito annotations (@Mock, @RunWith, @InjectMocks, @Captor, @Spy)**

**Step 1:** Create an interface named **ToDoService** that contains two unimplemented methods.

|  |  |  |
| --- | --- | --- |
| **ToDoService.java**   1. **import** java.util.List; 2. **public** **interface** ToDoService { 3. **public** List<String> getTodos(String user); 4. **public** **void** deleteTodos(String doString); 5. } | **Step 2:** Create an implementation class named **ToDoBusiness.**  **ToDoBusiness.java**  **public** **class** ToDoBusiness {  **public** ToDoService doService;  **public** ToDoBusiness(ToDoService doService) {  **this**.doService = doService;      }  **public** List<String> getTodosforHibernate(String user) {          List<String> hibernatelist = **new** ArrayList<String>();        List<String> Combinedlist = doService.getTodos(user);  **for**(String todo:Combinedlist) {  **if**(todo.contains("Hibernate"))            {   hibernatelist.add(todo);     }            } **return** hibernatelist;  }    **public** **void** deleteTodosNotRelatedToHibernate(String user) {   List<String> Combinedlist = doService.getTodos(user);  **for**(String todos:Combinedlist) {  **if**(!todos.contains("Hibernate"))              {   doService.deleteTodos(todos);              }   }  } | **Step 3:** Create a mock test named **ToDoBusinessMock** for testing purposes. In the following code, we will use all the annotations that we have discussed above.  **ToDoBusinessMock.java**  @RunWith(MockitoJUnitRunner.**class**)  **public** **class** ToDoBusinessMock {      @Mock      ToDoService servicemock;      @InjectMocks      ToDoBusiness business;       @Captor  ArgumentCaptor<String> argumentCaptor;      @Spy      ArrayList<String> arraylistSpy;       @Test  **public** **void** deleteTodosusing\_BDD() {        List<String> combinedlist = Arrays.asList("Use Hibernate Java",        "Use Hibernate Core", "Use Hibernate", "Use Spring MVC");         given(servicemock.getTodos("dummy")).willReturn(combinedlist);          arraylistSpy.add("Mockito1");        arraylistSpy.add("Mockito2");         // When        business.deleteTodosNotRelatedToHibernate("dummy");          // Then        verify(arraylistSpy).add("Mockito1");        verify(arraylistSpy).add("Mockito2");        verify(servicemock, times(1)).deleteTodos("Use Spring MVC");        verify(servicemock, never()).deleteTodos("Use Hibernate Java");        verify(servicemock, never()).deleteTodos("Use Hibernate");         assertEquals(2, arraylistSpy.size());        System.out.println("test is working..");       }          @Test  **public** **void** deleteTodosusing\_BDD\_usingArgumentCaptor() {          //Given        List<String> combinedlist = Arrays.asList("Use Hibernate Java",        "Use Hibernate Core", "Use Hibernate", "Use Spring MVC");          given(servicemock.getTodos("dummy")).willReturn(combinedlist);          //When        business.deleteTodosNotRelatedToHibernate("dummy");          //Then        then(servicemock).should().deleteTodos(argumentCaptor.capture());        assertThat(argumentCaptor.getValue(),is("Use Spring MVC"));        System.out.println("test is working..");       }   } |

# **PowerMock**

 The PowerMock framework uses a custom classloader and bytecode manipulation techniques to enable the mocking of static methods, final classes, final methods, private methods, constructor, and removal of static initializers. The main aim of PowerMock is to extend the existing APIs with some methods and annotations to provide extra features that make unit testing quite easy.

The PowerMock framework provides a class called **PowerMockito** used to create mock objects and initiates verification and expectation. The PowerMockito provides the functionality to work with the Java reflection API.

Using the powermock

1.We need to add power mock dependencies

**Step 2:** Apply the PowerMock annotations

To use PowerMock with Mockito, we need to apply the following **two** annotations in the test:

**@RunWith(PowerMockRunner.class):** It is the same as we have used in our previous examples. The only difference is that in the previous example we have used MockitoUnitRunner.class, now we will use PowerMockRunner.class for enabling the PowerMockito APIs in the test.

**@PrepareForTest:** It tells PowerMock to prepare some classes for testing. It can be applied to both the test classes and the individual test methods. It includes classes with final, static, private, or native methods that can be mocked.

1. @RunWith(PowerMockRunner.**class**)
2. @PrepareForTest(Utility.**class**)
3. **public** **class** Powermock\_test {
4. }

There are following examples of PowerMock with Mockito and JUnit frameworks.

### **1. Mocking static methods**

**Step 1:** Create a class that contains a static method. We have created a class with the name **Utility.**

**Utility.java**

1. **public** **class** Utility {
2. **public** **static** String staticMethod(String call) {
3. **return** call;
4. }
5. }
6. @RunWith(PowerMockRunner.**class**)
7. @PrepareForTest(Utility.**class**)
8. **public** **class** Powermock\_test {
10. @Test
11. **public** **void** TestStaticMethod\_WithPowerMockito() {
13. String call = " Hi there, I'm using PowerMock with Mockito ";
14. String callexpectation = " Call Expectation for you. ";
16. PowerMockito.mockStatic(Utility.**class**);
17. PowerMockito.when(Utility.staticMethod(call)).thenReturn(callexpectation);
19. String actualcall = Utility.staticMethod(call);
20. assertEquals(callexpectation, actualcall);
21. }
22. }

### **Mocking private methods**

In the following example, we will create a mock of private methods.

**Step 1:** Create a class that contains a private method. We have created class with the name **Utility** and defined a private method and a public method (that returns the object of private method).

**Utility.java**

1. **public** **class** Utility {
3. **private** String privateMethod(String message) {
4. **return** message;
5. }
7. **public** String callPrivateMethod(String message) {
8. **return** privateMethod(message);
9. }
10. }
11. @RunWith(PowerMockRunner.**class**)
12. @PrepareForTest(Utility.**class**)
13. **public** **class** Powermock\_test {
15. @Test
16. **public** **void** TestPrivateMethod\_WithPowerMock() **throws** Exception {
18. String message = " PowerMock with Mockito and JUnit ";
19. String expectedmessage = " Using with EasyMock ";
21. Utility mock =PowerMockito.spy(**new** Utility());
22. PowerMockito.doReturn(expectedmessage).when(mock, "privateMethod", message);
24. String actualmessage = mock.callPrivateMethod(message);
25. assertEquals(expectedmessage, actualmessage);
27. System.out.println(PowerMockito.verifyPrivate(getClass()));
28. }
29. }

### **Mocking final methods**

In the following example, we are going to mock final methods.

**Step 1:** Create a class that contains a final method. We have created a class with the name Utility and defined a final method named **finalMethod.**

**Utility.java**

1. **public** **class** Utility {
3. **public** **final** String finalMethod(String message) {
4. **return** message;
5. }
6. }
7. @RunWith(PowerMockRunner.**class**)
8. @PrepareForTest(Utility.**class**)
9. **public** **class** Powermock\_test {
11. @Test
12. **public** **void** TestFinalMethod\_WithPowerMock() **throws** Exception {
14. String message = " PowerMock with Mockito and JUnit ";
15. Utility uti = PowerMockito.mock(Utility.**class**);
16. PowerMockito.whenNew(Utility.**class**).withNoArguments().thenReturn(uti);
18. Utility uti2 =  **new** Utility();
19. PowerMockito.verifyNew(Utility.**class**).withNoArguments();
21. PowerMockito.when(uti2.finalMethod(message)).thenReturn(message);
23. String message2 = uti2.finalMethod(message);
24. Mockito.verify(uti2).finalMethod(message);
25. assertEquals(message, message2);
26. }
27. }