## Links for reference

|  |  |
| --- | --- |
| Junit user guide | <https://junit.org/junit5/docs/current/user-guide/#writing-tests> |
| Mockito ref links | <https://site.mockito.org/>  <https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html> |
|  |  |
|  |  |
| Spring framework guru repos for reference | <https://github.com/springframeworkguru/testing-junit5-mockito/tree/mvn-deps> |
| Misc lib |  |

Mockito

## Mockito jars

|  |  |
| --- | --- |
| Mockito core  <dependency>  <groupId>org.mockito</groupId>  <artifactId>mockito-core</artifactId>  <version>${mockito.version}</version>  <scope>test</scope>  </dependency> | Mockito junit 5 bridge jar  <dependency>  <groupId>org.mockito</groupId>  <artifactId>mockito-junit-jupiter</artifactId>  <version>${mockito.version}</version>  <scope>test</scope>  </dependency> |
|  | Mockito-inline (this is to mock static and final methods) |

About Mockito

Dummy - Object used just to get the code to compile

• Fake - An object that has an implementation, but not production ready

• Stub - An object with pre-defined answers to method calls

• Mock - An object with pre-defined answers to method calls, and has expectations of executions. Can throw an exception if an unexpected invocation is detected

• Spy - In Mockito Spies are Mock like wrappers around the actual object.

• Verify - Used to verify number of times a mocked method has been called

• Argument Matcher - Matches arguments passed to Mocked Method & will allow or disallow

• Argument Captor - Captures arguments passed to a Mocked Method • Allows you to perform assertions of what was passed in to method

### Annotations

|  |  |
| --- | --- |
| @Mock | It is just to mark the field as mock it won’t create the mock object |
| @ExtendWith(MockitoExtension.class) | This is used to initialize the mocks (the variables which are annotated with @Mock)  Employee e= Mockito.createMock(Employee.class) // this also can create the mock object  Note:- Before every test method execution again all new mocks will be created and all those new mocks will be again injected into test class, why ? bec to clear previous stubbed behaviours |
| @Spy | Used to create spy |
| @InjectMocks | To inject the current class mocks into class which is under test  *VetRepository* **vetRepository**=Mockito.*mock*(*VetRepository*.class); // Mock creation using mock method  @Mock  VetController **vetController**; // mock creation using @Mock +@ExtendWith  @InjectMocks //This anno is capable of injecting mocks (which are created in any of above 2 ways) present in current class to class under test  VetSDJpaService **vetSDJpaService**; |
| @Captor | Captured the arguments to mock |

3 ways to create mock

|  |  |
| --- | --- |
| @ExtendWith(MockitoExtension.class) // from junit 5, in junit 4 we used runners to initialize the mocks  @RunWith(MockitoJUnitRunner.class) // old way junit 4  This is the best and easiest way | @ExtendWith(MockitoExtension.class)  @Mock VetService vetService; |
| 1. Employee e= Mockito.createMock(Employee.class) simple method called inline mocks |  |
| 1. @Mock + use MockitoAnnotations.initMocks(this); | @BeforeEach void setUp() {  MockitoAnnotations.*initMocks*(this);// old way  MockitoAnnotations.openMocks(testClass);// new way  }  @Mock VetService vetService; |
| Use junit Rule |  |

Sample program

@RunWith(MockitoJUnitRunner.class)

public class AddingMachineAnnotationTest {

@Mock

private List<Integer> mockList;

@InjectMocks

private AddingMachine machine;

// @Rule

// public MockitoRule rule = MockitoJUnit.rule().strictness(Strictness.STRICT\_STUBS);

// @Before

// public void setUp() {

// MockitoAnnotations.openMocks(this);

// }

### **Argument Matcher**

Matching the arguments is called argument matcher

Note: while writing Mockito.when() instead of passing argument at runtime we can pass this ArgumentMatcher

*when*(**vetRepository**.save(ArgumentMatchers.*any*(Vet.class))).thenReturn(**vet**);

Here for save method instead of passing argument we are passing argument matcher

|  |  |
| --- | --- |
| ArgumentMatchers.any() – anything includes null values …  ArgumentMatchers.any(String.class)  ArgumentMatchers.anyObject() //Deprecated in new versions  ArgumentMatchers.anyLong() | ArgumentMatchers.eq(“2”)  ArgumentMatchers.*same*(2l) |

Mockito.when().ThenReturn()

*when*(mockedList.get(0)).thenReturn("first");  
*when*(mockedList.get(1)).thenThrow(new RuntimeException());

Note: while writing Mockito.when() instead of passing argument at runtime we can pass this ArgumentMatcher

*when*(**vetRepository**.save(ArgumentMatchers.*any*(Vet.class))).thenReturn(**vet**);

Argument matchers

Mockito verifies argument values in natural java style: by using an equals() method. Sometimes, when extra flexibility is required then you might use argument matchers:

//stubbing using built-in anyInt() argument matcher

when(mockedList.get(anyInt())).thenReturn("element");

|  |  |
| --- | --- |
| Original  public *Set*<Vet> findAll() {  *Set*<Vet> vets = new HashSet<>();  **vetRepository**.findAll().forEach(vets::add);  return vets; } | Source code  @Test void findById() {  Long id=2l;  *when*(**vetRepository**.findById(ArgumentMatchers.*anyLong*())).thenReturn(Optional.*of*(**vet**));  Vet result = **vetSDJpaService**.findById(id);  Assertions.*assertThat*(result).isNotNull(); } |
|  |  |

when(mock.someMethod()).thenReturn(10);

//you can use flexible argument matchers, e.g:

when(mock.someMethod(anyString())).thenReturn(10);

//setting exception to be thrown:

when(mock.someMethod("some arg")).thenThrow(new RuntimeException());

//you can set different behavior for consecutive method calls.

//Last stubbing (e.g: thenReturn("foo")) determines the behavior of further consecutive calls.

when(mock.someMethod("some arg"))

.thenThrow(new RuntimeException())

.thenReturn("foo");

//Alternative, shorter version for consecutive stubbing:

when(mock.someMethod("some arg"))

.thenReturn("one", "two");

//is the same as:

when(mock.someMethod("some arg"))

.thenReturn("one")

.thenReturn("two");

//shorter version for consecutive method calls throwing exceptions:

when(mock.someMethod("some arg"))

.thenThrow(new RuntimeException(), new NullPointerException();

Custom Argument matcher- rarely used

Instead of using this prefer – when().thenAnswer(), or BDD’s given().willAnswer() because this supports only 1 stubbing,

Writing when.ThenReturn() based on data

|  |  |
| --- | --- |
| Generic argument matcher using –any(Person.class)  Here it will just compare the types if types are matched then output will be returned | Custom argument matcher – stub based on input – cfg behaviour based on inputs  Here when input matched then only return type will be returned,  If input didn’t match it will throw error saying stubbed is quite diff from what has been passed in real time |
| Mockito.*when*(**vetController**.listPersonVets(ArgumentMatchers.***any***(Person.class)))  .thenReturn(result); | Mockito.*when*(**vetController**.  listPersonVets(ArgumentMatchers.***argThat***(*ag*->*ag*.getFirstName().equals("Manis"))))  .thenReturn(result); |
| *Mockito.when(ownerService*  *.save(ArgumentMatchers.****any****(Owner.class)))  .thenReturn(owner);* | Mockito.*when*(**ownerService**  .save(ArgumentMatchers.***argThat***(*arg*->*arg*.getLastName().equals("vv"))))  .thenReturn(**owner**); |

Case study 1:-

Here my requirement is based on input I should return the value, then

Stubbing 2 times is not possible for 2 diff inputs, if u do it will throw errors (as we should stub only once), Hence prefer thenAnswer();

//If last name is bv then return ownerSai

Mockito.*when*(**ownerService**.save(ArgumentMatchers.  
 *argThat*(*argu*->*argu*.getLastName().equals("bv"))))  
 .thenReturn(**ownerSai**);

//If last name is vv then return owner

Mockito.*when*(**ownerService**.save(ArgumentMatchers.*argThat*(*arg*->*arg*.getLastName().equals("vv"))))  
 .thenReturn(**owner**);

* Argument based stubbing /thenAnswer()

Stub or define behaviour Based upon arguments

Based on inputs return the value, below will be the code if we use plain ArgumentMatchers

Mockito.*when*(**bindingResult**.hasErrors()).thenReturn(false);  
Mockito.*when*(**ownerService**.save(ArgumentMatchers.*any*(Owner.class)))  
 .thenReturn(**owner**); //Generic stubbing ~~ what ever the input same output  
Assertions.*assertThat*(**ownerController**.processCreationForm(**ownerSai**,**bindingResult**))  
 .endsWith("1");

Below will be the code if we want to define behaviour based on input- depending upon input return value changes

|  |  |
| --- | --- |
| Original method | Junit code |
| public String processCreationForm(@Valid Owner *owner*, *BindingResult result*) {  if (*result*.hasErrors()) {  return ***VIEWS\_OWNER\_CREATE\_OR\_UPDATE\_FORM***;  } else {  System.***out***.println(*owner*.getLastName());  Owner savedOwner = **ownerService**.save(*owner*);  return "redirect:/owners/" + savedOwner.getId();  } } | @ParameterizedTest @ValueSource(strings = {"false"}) @DisplayName("willAnswer") void processCreationFormMockitoStyleWillAnswer(boolean *hasErrors*) {  System.***out***.println("test method invoked with value -->" + *hasErrors*);  System.***out***.println("stubbing with thenAnswer");  Mockito.*when*(**ownerService**.save(ArgumentMatchers.  *any*(Owner.class)))  .thenAnswer(*invocation* -> {  Owner argument = (Owner) *invocation*.getArgument(0); //Here based on arguments we are returning some value, if this is not required u can use ArgumentMatchers.any(Owner.class) also  if ("bv".equals(argument.getLastName())){  return **owner**;  }else {  return **ownerSai**;  }  });   System.***out***.println(**ownerController**.processCreationForm(**ownerSai**, **bindingResult**)); } |

when(mock.someMethod(anyString())).thenAnswer(

new Answer() {

public Object answer(InvocationOnMock invocation) {

Object[] args = invocation.getArguments();

Object mock = invocation.getMock();

return "called with arguments: " + Arrays.toString(args);

}

});

BDD style when-then

Instead of when().thenReturn, verify() we can use BDD style methods

Given () – means pre condition – means here we will write all when then statements/stubbing’s ~~ setting predefined answers for real method calls

When () – states action – real method call

Then () – means result – here we will verify the behavior like how many time it called

|  |  |
| --- | --- |
| Mockito style – when.ThenReturn() | BDD style |
| Mockito.*when*(**ownerService**.findById(ArgumentMatchers.*anyLong*()))  .**then**Return(**owner**); **ownerController**.showOwner(2L); Mockito.*verify*(**ownerService**,Mockito.*times*(1))  .findById(Mockito.*anyLong*()); | *//Given -- pre condition – we are setting predefined answers* BDDMockito.***given***(  **ownerService**.findById(ArgumentMatchers.*anyLong*())  ).**will**Return(**owner**);  *//****When*** *- action -- the real method calls* **ownerController**.showOwner(2L);  *//Then - result* BDDMockito.***then***(**ownerService**).should(Mockito.*atLeastOnce*()).  findById(ArgumentMatchers.*anyLong*()); |
| For non-void method use method called 🡪 thenThrow() | *willThrow()* |

//Here instead of Mockito when().ThenReturn() methods we used BDDMockito.given(),WillReturn() methods

@ParameterizedTest  
@ValueSource(strings = {"false", "true"})  
void processCreationFormBddStyle(boolean *hasErrors*) {  
 System.***out***.println("test method invoked wiht value -->" + *hasErrors*);  
 if (!*hasErrors*) {  
 *//Given* BDDMockito.**given**(**ownerService**.save(ArgumentMatchers.*argThat*(*arg* -> *arg*.getLastName().endsWith("v")))).**will**Return(**owner**);  
 *//When* **ownerController**.processCreationForm(**owner**, **bindingResult**);  
 *//Then* BDDMockito.**then**(**ownerService**).should().save(ArgumentMatchers.*any*(Owner.class));  
  
 } else {  
 *//Given* BDDMockito.**given**(**bindingResult**.hasErrors()).**will**Return(true);  
 *//When* **ownerController**.processCreationForm(**owner**, **bindingResult**);  
 *//Then* BDDMockito.**then**(**bindingResult**).should(Mockito.*times*(1)).hasErrors();  
 }  
}

Throwing exceptions

|  |  |
| --- | --- |
| Mockito style – method names | BDD style – method names |
| When(),ThenReturn()  ThenThrow() for non-void methods & doThrow() for void methods | Given().willReturn  WillThrow() – for both void and non-void methods |

|  |  |
| --- | --- |
| Non-void methods, Mockito style | *when*(**vetRepository**.findAll()).thenThrow(RuntimeException.class);  when(mockedList.get(1)).thenThrow(new RuntimeException());  consecutive stubbing  when(mock.someMethod("some arg"))  .thenThrow(new RuntimeException(), new NullPointerException(); |
| Void methods , mockito style | Mockito.*doThrow*(RuntimeException.class).when(**vetRepository**).deleteById(1L);  // u can send either class or u can create object and throw  doThrow(new RuntimeException()).when(mockedList).clear();  doThrow(RuntimeException.class).when(mock).someVoidMethod(); |
| Non-void methods, BDD style | BDDMockito.*given*(**vetRepository**.findAll()).  willThrow(RuntimeException.class); |
| Void methods, BDD style | BDDMockito.*willThrow*(RuntimeException.class)  .given(**vetRepository**).delete(ArgumentMatchers.*any*(Vet.class)); |
|  |  |

Non-void BDD style

@DisplayName("Throwing exceptions for non-void methods in BDD style")  
@Test  
void findAllWithExceptionsForNonVoidMethods() {  
*//Given -pre condition* BDDMockito.*given*(**vetRepository**.findAll()).  
 willThrow(RuntimeException.class);  
*//When - action - real* org.junit.jupiter.api.Assertions.  
 *assertThrows*(RuntimeException.class,()->**vetSDJpaService**.findAll());  
 *//Then* BDDMockito.*then*(**vetRepository**).should().findAll();  
}

Void methods BDD style

@DisplayName("Throw exceptions for void methods in BDD style")  
@Test  
void deleteTestExceptionsVoidMethodsBddStyle() {  
 *//Given -pre condition* BDDMockito.*willThrow*(RuntimeException.class)  
 .given(**vetRepository**).delete(ArgumentMatchers.*any*(Vet.class));  
 *//When - action- real method call* org.junit.jupiter.api.Assertions.  
 *assertThrows*(RuntimeException.class,()->**vetSDJpaService**.delete(**vet**));  
 *//Then - result* BDDMockito.*then*(**vetRepository**).should(BDDMockito.*times*(1)).delete(*any*(Vet.class));  
}

Non-Void methods Mockito style

@DisplayName("Throw exceptions for non-void methods")  
@Test  
void findAllWithExceptionNon\_VoidMethod() {  
 *when*(**vetRepository**.findAll()).thenThrow(RuntimeException.class);  
 org.junit.jupiter.api.Assertions.  
 *assertThrows*(RuntimeException.class,()->**vetSDJpaService**.findAll());  
 *verify*(**vetRepository**,*times*(1)).findAll();  
}

Void methods Mockito style

@DisplayName("Throw exceptions for void methods ")  
@Test  
void deleteExceptionsForVoidMethods() {  
 Mockito.*doThrow*(RuntimeException.class).when(**vetRepository**).deleteById(1L);  
  
 org.junit.jupiter.api.Assertions.  
 *assertThrows*(RuntimeException.class,()->**vetSDJpaService**.deleteById(1L));  
   
 *verify*(**vetRepository**,*times*(1)).deleteById(1L);  
}

Argument captor

With Argument captor, we can capture the arguments that are passed to the mocked method which are not returnable

**We should captor while defining behavior itself** like 🡪 writing when().ThenReturn() at that time itself, we can also write at verify() stage too

But better to write while stubbing

Code sample:-

@Captor  
ArgumentCaptor<Long> **ac**;  
  
@Test  
void initUpdateOwnerFormMockitoStyle() {  
 Long ownerId = 499L;

// here we will capture the value that is being passed to mocked object’s method  
 Mockito.*when*(**ownerService**.findById(**ac**.capture())).thenReturn(**owner**);  
 **ownerController**.initUpdateOwnerForm(ownerId, **model**);  
 System.***out***.println("captured value is " + **ac**.getValue());  
}

Mockito.verify()

We should write this verify only after actual call happenned

Mockito.*verify*(vetService,Mockito.*times*(1)).findAll();  
Mockito.*verify*(vetService,Mockito.*atLeastOnce*()).findAll();  
Mockito.*verify*(vetService,Mockito.*atMost*(2)).findAll();  
Mockito.*verify*(vetService,Mockito.*never*()).deleteById(2L);

Mockito.*verify*(vetService,Mockito.*atLeastOnce*()).deleteById(Mockito.*eq*(2L));// while verifying we can check even the same value or not

Mockito.*verify*(vetService,Mockito.*atLeastOnce*()).deleteById(Mockito.*anyLong*());

Sample

|  |  |
| --- | --- |
| Sample source  public void deleteById(Long aLong) {  vetRepository.deleteById(aLong); } | Sample junit  void deleteById() {  vetSDJpaService.deleteById(2L);  Mockito.*verify*(vetRepository,Mockito.*times*(1)).deleteById(2L); } |

//This is interesting

// will print a custom message on verification failure

verify(mock, description("This will print on failure")).someMethod();

// will work with any verification mode

verify(mock, times(2).description("someMethod should be called twice")).someMethod();

Diff between verify() and when()

Verify() only accepts single argument object 🡪 verify(mock, times(2).description("a”)

When() accepts entire method🡪 *when*(mockedList.get(0)).thenReturn("first");

#### **Verify order**

InOrder order= inOrder(mock1,mock2) // here order declaration doesn’t matter, it matters only in verify()

|  |  |
| --- | --- |
|  |  |
| public String listVetsAndLoad(Person *person*) {  System.***out***.println("In employee controller");  **visitController**.loadPetWithVisit(*person*.getId(),new HashMap<>());  return **vetController**.listPersonVets(*person*); } | @DisplayName("OrderOfMocksDemo") @Test public void orderMocksDemo() {  Visit visit=new Visit();  *//Here order declaration is not important*  ***InOrder* inOrder= *Mockito.inOrder*(vetController,visitController);**  Mockito.*when*(**visitController**.loadPetWithVisit(ArgumentMatchers.*anyLong*(),ArgumentMatchers.*any*(HashMap.class)))  .thenReturn(visit);  Mockito.*when*(**vetController**.listPersonVets(ArgumentMatchers.*any*(Person.class)))  .thenReturn("abcd");  **employeeController**.listVetsAndLoad(new Person(1L,"Mani","deep"));  *//Here only order declaration is important* **inOrder.verify**(**visitController**,*times*(1))  .loadPetWithVisit(ArgumentMatchers.*anyLong*(),ArgumentMatchers.*any*(HashMap.class));  inOrder.verify(**vetController**,*times*(1)).listPersonVets(Mockito.*any*(Person.class)); } |

Mockito.spy()

Spy= original+mock

Spy is the real object & we can stub too (we can define the behaviour)

If we don’t define the behaviour then real method will be called

@Test  
void setAllowedFields() {  
 **petService**.save(**pet**);*// Since this is spy & as we didnt stubbed original method will be called* System.***out***.println("ps record-->"+**petService**.findById(1L));*// now also original method will be called* Mockito.*when*(**petService**.findById(1L)).thenReturn(null);  
 *// as we have stubbed findby method in above line,below will gives null* System.***out***.println("ps record after stubbing-->"+**petService**.findById(1L));  
}

Mockito.reset()

@AfterEach  
void tearDown() {  
 Mockito.*reset*(**clinicService**);  
}

This is the best way to clear the stubs, every time, no need to re-create and re-initialise the mock to the same variable, this is d best way

## Run Gradle Test from command prompt intellij

* In windows use backslash only
* Dot( .) means current directory
* **Start** command in git bash Is to open the file

In intellij type below commands

1. Run “**test** “command with **gradlew** file

Command is “**.\gradlew test”**

1. To see all the test reports use the below command

**“Start** **./build/reports/tests/test/index.html**”

Note: - in git hub terminology start means open whereas dot means current directory

Hamcrest

## Run gradle tests

## **Hamcrest**

Hamcrest-library

### Documentation –

Home page: <http://hamcrest.org>

In junit 4 hamcrest is already added as a transitive dependency

From junit 5 you have to manually add Hamcrest, but in junit 4 it was automatically added

Main jars are ()

// https://mvnrepository.com/artifact/org.hamcrest/hamcrest

testImplementation 'org.hamcrest:hamcrest:2.2'

Hamcrest 2.2 is stable for several years

Java implementation: <http://hamcrest.org/JavaHamcrest/>

JavaDocs: <http://hamcrest.org/JavaHamcrest/javadoc/>

Tutorial: <https://code.google.com/archive/p/hamcrest/wikis/Tutorial.wiki>

Mailing list: <https://groups.google.com/forum/?fromgroups#!forum/hamcrest-java>

Source code: <https://github.com/hamcrest/JavaHamcrest>

## Main classes in Hamcrest

Org.hamcrest.Matcher is an interface

Org.hamcrest.Matchers is a class – all methods is , null, notnull, all these utility methods are present in this class

Org.hamcrest.MatcherAssert.

In this class we have 2 important methods 🡪 assertThat() it is a class contain only assertThat methods

#### Methods in Matchers Class

The below are the methods present in org.hamcrest.Matchers class

Famous one is

**assertThat(“abc”, is(“abc”));**

For all below assertions example refer **simpleAssertions.java,StringTests.java**

* anyOf(matcher1,matcher2); true if any matchers match
* allOf → true if all matchers match
* ContainsString(String str)

Ex:- assertThat(“abcd”,containsString(“abcdef”));

* anyOf → not → flips boolean
* isA,
* instanceOf → type test
* not
* nullValue, ex:- assertThat(null,nullValue());// to assert left hand side var is null
* notNullValue
* sameInstance
* greaterThan,
* greaterThanOrEqualTo
* lessThan,

for time comparisions use this

assertThat(LocalDate.now(),is(lessThan(localDate.now().minusDays(2))));

* lessThanOrEqualTo
* closeTo → used for floating point and BigDecimals is,
* equalTo
* equalToIgnoreCase,
* equalToCompressingWhiteSpace white space trimmed, then internal collapsed to single space
* containsString,
* endsWith,
* startsWith
* emptyString()
* emptyOrNullString()

In collections we can use these methods

hasSize(int size)

contains

containsInAnyOrder

hasItem()

In Arrays we can use these methods

## Suggestions in junit-Mockito

1. Avoid using Mockito.anyString() use Mockito.eq() or direct value which internally will use equals() method
2. Try using short hand operator Mockito.thenReturn(“abcs”, “def”); instead of using thenReturn 2 times
3. Verify() is mandatory
4. Assertion is mandatory- Use captor while writing assert statements
5. Try reading the documentation for each and every method and class