Using JUnit 4 for Eclipse

1.1. Creating JUnit tests

You can write the JUnit tests manually, but Eclipse supports the creation of JUnit tests via wizards.

For example, to create a JUnit test or a test class for an existing class. Right-click on your new class, select this class in the Package Explorer_ view, right-click on it and select New ▸ JUnit Test Case.

Alternatively you can also use the JUnit wizards available under File ▸ New ▸ Other… ▸ Java ▸ JUnit.

1.2. Running JUnit tests

The Eclipse IDE also provides support for executing your tests interactively.

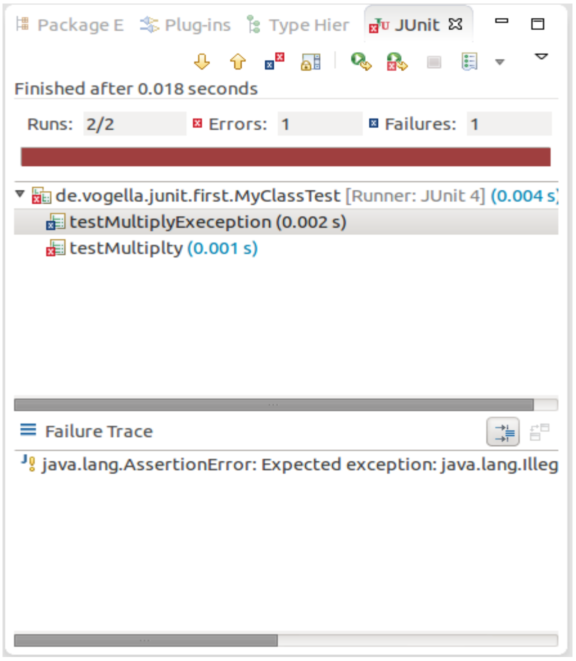
To run a test, select the test class, right-click on it and select Run-as ▸ JUnit Test.

This starts JUnit and executes all test methods in this class.

Eclipse provides the Alt + Shift + X , T shortcut to run the test in the selected class.

To run only the selected test, position the cursor on the test method name and use the shortcut.

To see the result of a JUnit test, Eclipse uses the *JUnit* view which shows the results of the tests. You can also select individual unit tests in this view, right-click on them and select *Run* to execute them again.



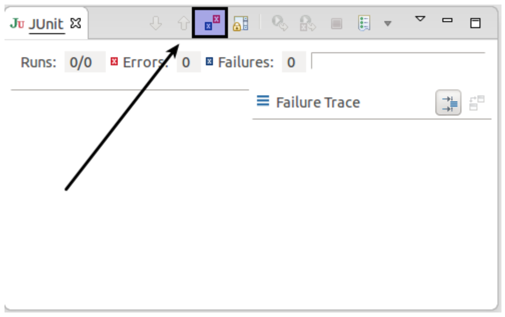
Finished after 0.018 seconds

Runs: 2/2 Errors: 1 Failures: 1

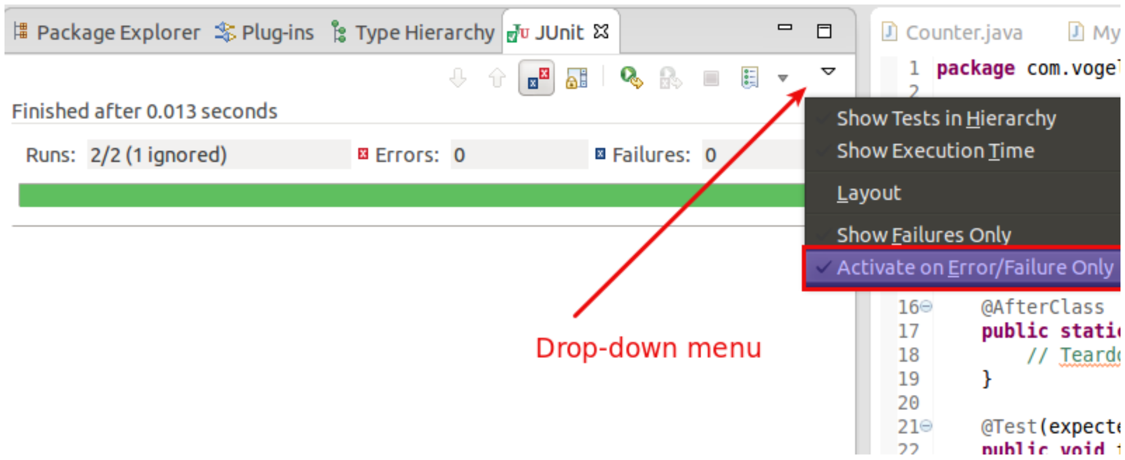
de.vogella.junit.first.MyClassTest [Runner: JUnit 4] (0.004 s)
testMultiplyExeception (0.002 s)
testMultiplty (0.001 s)

java.lang.AssertionError: Expected exception: java.lang.Illeg

By default, this view shows all tests. You can also configure, that it only shows failing tests.



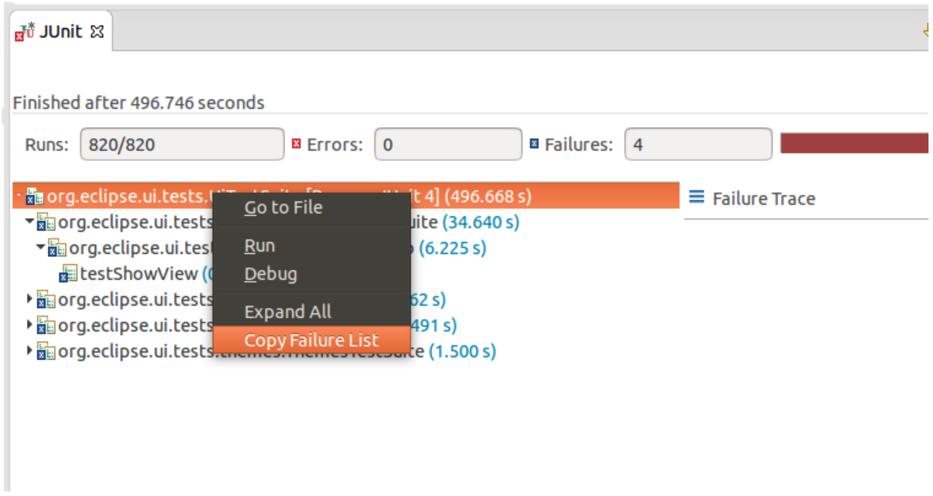
You can also define that the view is only activated if you have a failing test.



NOTE: Eclipse creates run configurations for tests. You can see and modify these via the Run ▸ Run Configurations… menu.

1.3. Extracting the failed test and stacktraces

To get the list of failed test, right click on the test result and select *Copy Failure List*. This copies the failed tests and there stack traces into the clipboard.



1.4. JUnit static imports

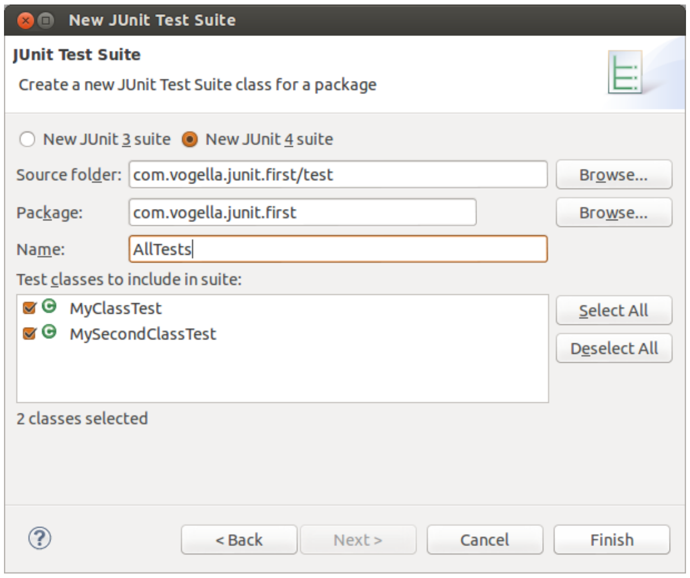
Static import is a feature that allows fields and methods defined in a class as public static to be used without specifying the class in which the field is defined.

JUnit assert statements are typically defined as public static to allow the developer to write short test statements. The following snippet demonstrates an assert statement with and without static imports.

```
// without static imports you have to write the following statement
Assert.assertEquals("10 x 5 must be 50", 50, tester.multiply(10, 5));


// alternatively define assertEquals as static import
import static org.junit.Assert.assertEquals;

// more code

// use assertEquals directly because of the static import
assertEquals("10 x 5 must be 50", 50, tester.multiply(10, 5));
```

1.1. Wizard for creating test suites

You can create a test suite via Eclipse. For this, select the test classes which should be included in suite in the *Package Explorer* view, right-click on them and select New ▸ Other… ▸ JUnit ▸ JUnit Test Suite.


New JUnit Test Suite

JUnit Test Suite

Create a new JUnit Test Suite class for a package

New JUnit 3 suite [✓ New JUnit 4 suite]

Source folder: com.vogella.junit.first/test

Package: com.vogella.junit.first

Name: AllTests

Test classes to include in suite:

✓ MyClassTest
✓ MySecondClassTest

2 classes selected

1.6. Testing exception

The @Test (expected = Exception.class) annotation is limited as it can only test for one exception. To testexceptions, you can use the following testpattern.

```
try {
  mustThrowException();
  fail();
} catch (Exception e) {
  // expected
  // could also check for message of exception, etc.
}
```

1.7. JUnit Plug-in Test

JUnit Plug-in tests are used to write unit tests for your plug-ins. These tests are executed by a special test runner that launches another Eclipse instance in a separate VM. The test methods are executed within that instance.

2. Setting Eclipse up for using JUnits static imports

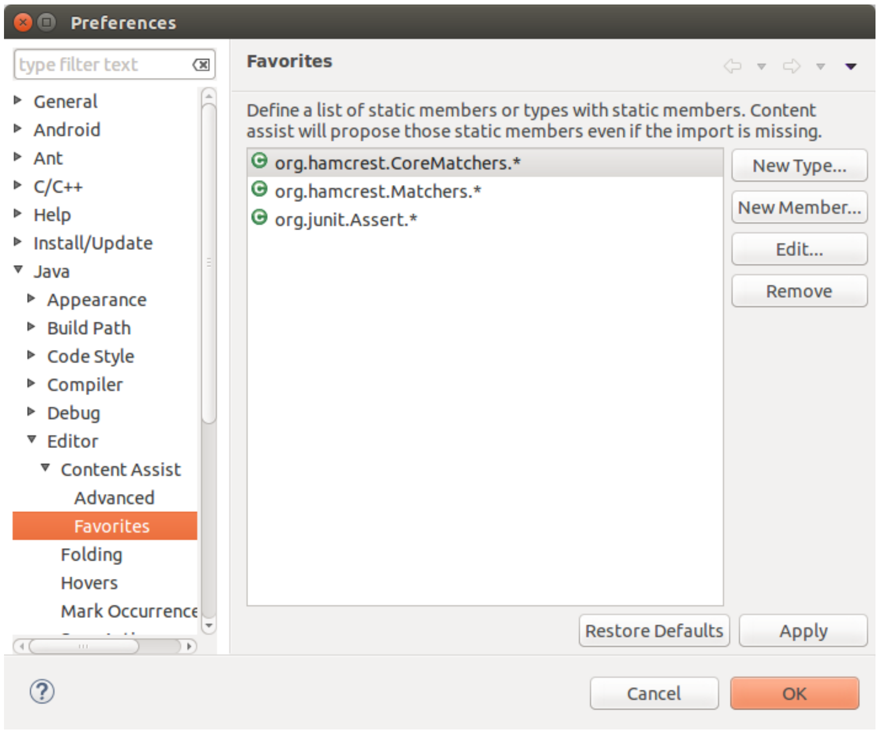
The Eclipse IDE cannot always create the corresponding static import statements automatically.

You can configure the Eclipse IDE to use code completion to insert typical JUnit method calls and to add the static import automatically. For this open the Preferences via Window ▸ Preferences and select Java ▸ Editor ▸ Content Assist ▸ Favorites.

Use the New Type button to add the following entries to it:

- org.junit.Assert
- org.hamcrest.CoreMatchers
- org.hamcrest.Matchers

This makes, for example, the assertTrue, assertFalse and assertEquals methods directly available in the *Content Assists*.


Favorites

Define a list of static members or types with static members. Content assist will propose those static members even if the import is missing.

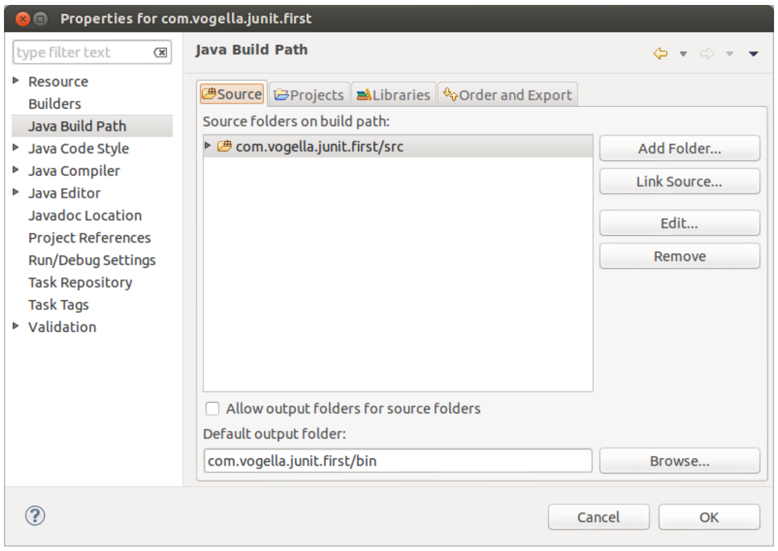
org.hamcrest.CoreMatchers.*
org.hamcrest.Matchers.*
org.junit.Assert.*

You can now use *Content Assists* (shortcut: Ctrl + Space) to add the method and the import.

3. Exercise: Using JUnit

3.1. Project preparation

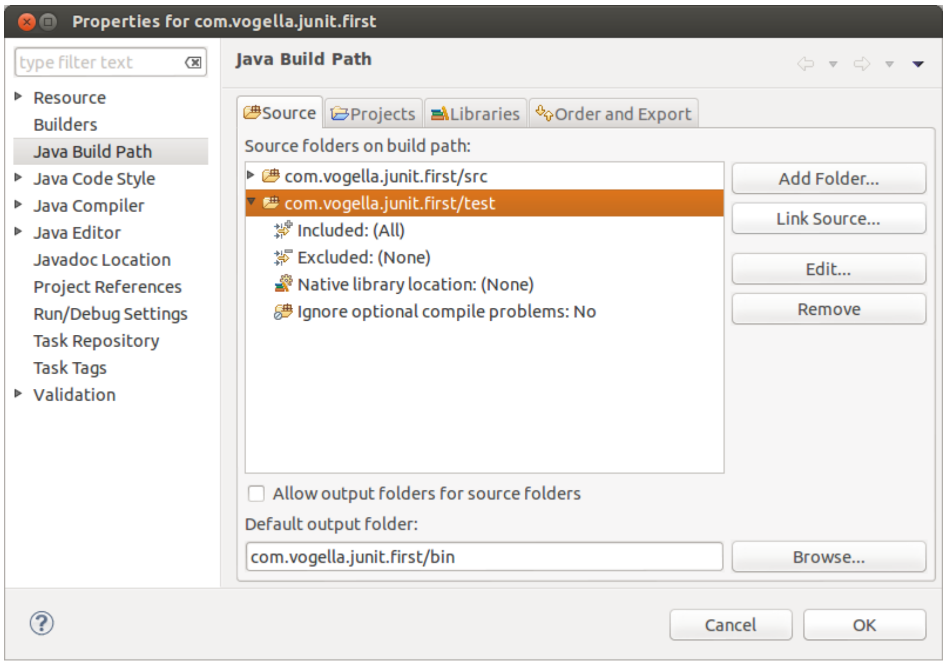
Create a new project called *com.vogella.junit.first*. Create a new source folder *test*. For this right-click on your project, select *Properties* and choose Java ▸ Build Path. Select the *Source* tab.



Source folders on build path:
com.vogella.junit.first/src
Allow output folders for source folders
Default output folder:
com.vogella.junit.first/bin

Press the Add Folder button. Afterwards, press the Create New Folder button. Enter *test* as folder name.

The result is depicted in the following screenshot.



Source folders on build path:
com.vogella.junit.first/src
com.vogella.junit.first/test
Included: (All)
Excluded: (None)
Native library location: (None)
Ignore optional compile problems: No
Allow output folders for source folders
Default output folder:
com.vogella.junit.first/bin

NOTE:You can also add a new source folder by right-clicking on a project and selecting New ▸ Source Folder.

3.2. Create a Java class

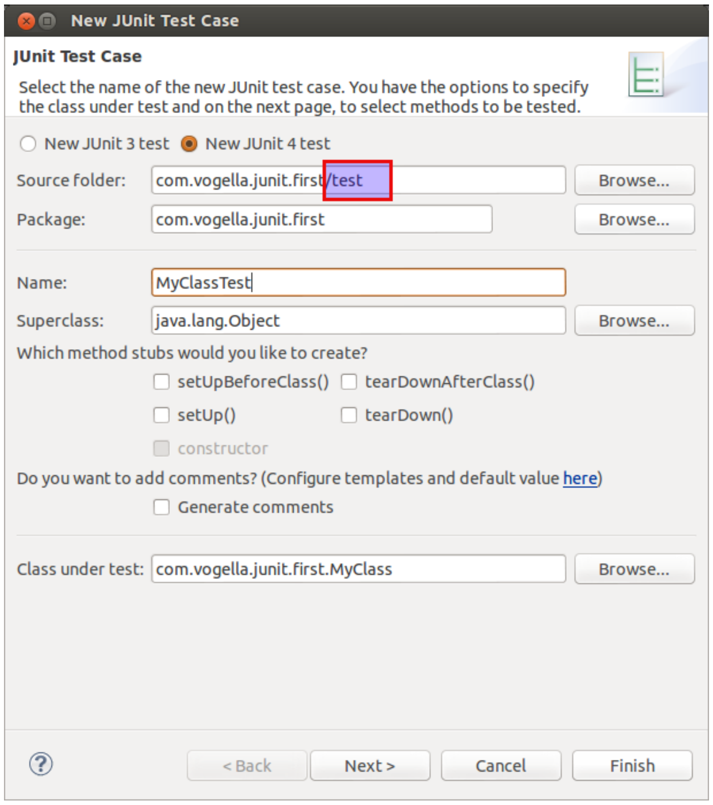
In the *src* folder, create the com.vogella.junit.first package and the following class.

```
package com.vogella.junit.first;

public class MyClass {
  public int multiply(int x, int y) {
    // the following is just an example
    if (x > 999) {
      throw new IllegalArgumentException("X should be less than 1000");
    }
    return x / y;
  }
}
```

3.3. Create a JUnit test

Right-click on your new class in the *Package Explorer* view and select New ▸ JUnit Test Case.

In the following wizard ensure that the *New JUnit 4 test* flag is selected and set the source folder to *test*, so that your test class gets created in this folder.


New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

New JUnit 3 test [✓ New JUnit 4 test]

Source folder: com.vogella.junit.first/test

Package: com.vogella.junit.first

Name: MyClassTest

Superclass: java.lang.Object

Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()

☐ setUp() ☐ tearDown()

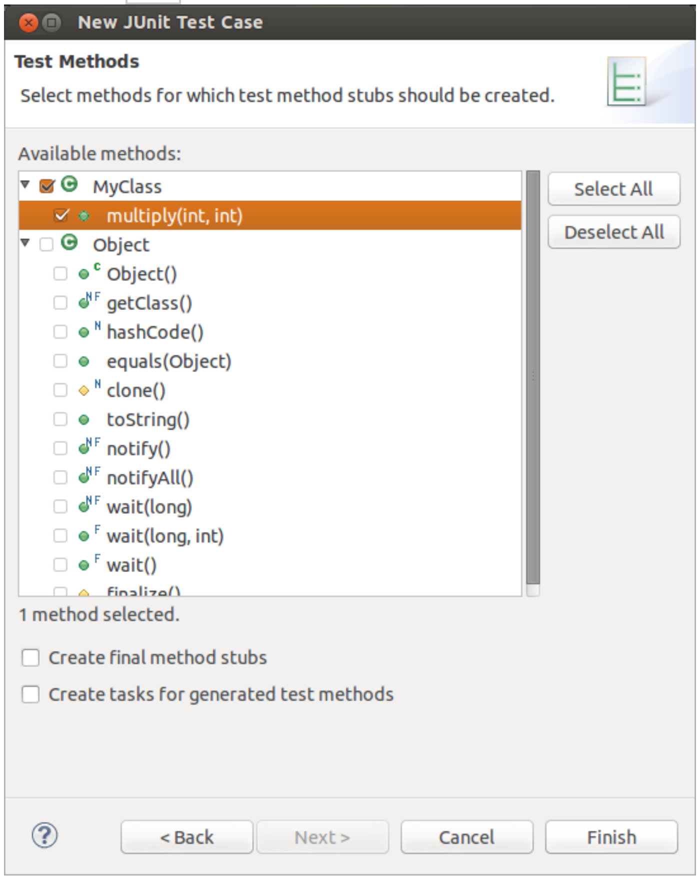
☐ constructor

Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

Class under test: com.vogella.junit.first.MyClass

Press the Next button and select the methods that you want to test.



New JUnit Test Case

Test Methods

Select methods for which test method stubs should be created.

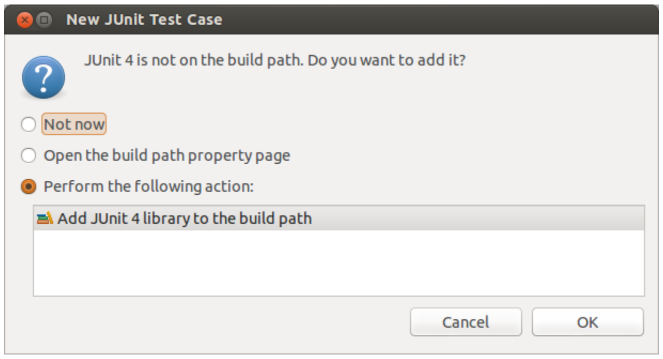
Available methods:

- [✓] MyClass
  - [✓] multiply(int, int)
- [ ] Object
  - [ ] Object()
  - [ ] getClass()
  - [ ] hashCode()
  - [ ] equals(Object)
  - [ ] clone()
  - [ ] toString()
  - [ ] notify()
  - [ ] notifyAll()
  - [ ] wait(long)
  - [ ] wait(long, int)
  - [ ] wait()
  - [ ] finalize()

1 method selected.

- [ ] Create final method stubs
- [ ] Create tasks for generated test methods

If the JUnit library is not part of the classpath of your project, Eclipse will prompt you to add it. Use this to add JUnit to your project.



New JUnit Test Case

JUnit 4 is not on the build path. Do you want to add it?

- Not now
- Open the build path property page
- [✓ Perform the following action:]

Add JUnit 4 library to the build path

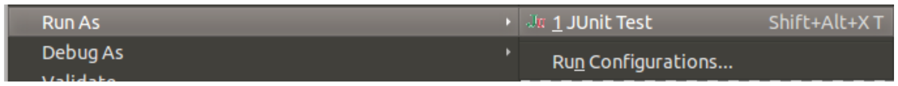
Create a test with the following code.

```
package com.vogella.junit.first;

import static org.junit.Assert.assertEquals;

import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
```

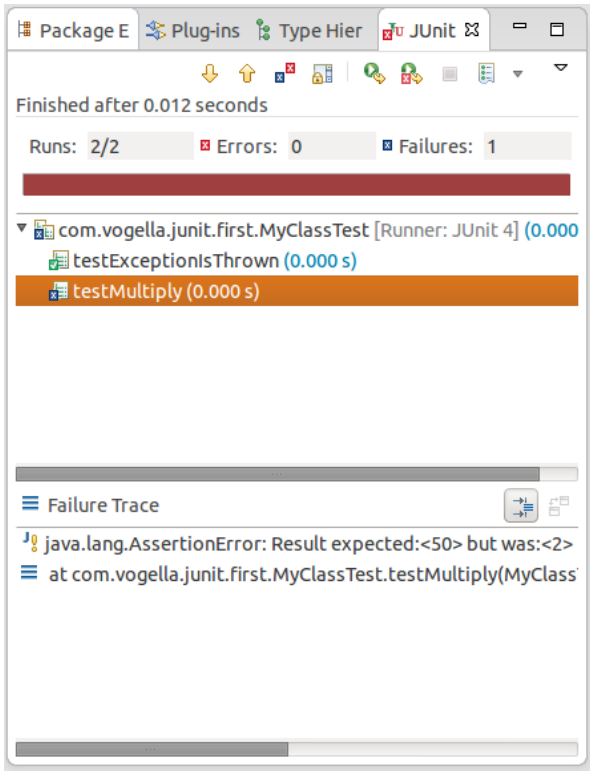
```
public class MyClassTest {

  @Test(expected = IllegalArgumentException.class)
  public void testExceptionIsThrown() {
    MyClass tester = new MyClass();
    tester.multiply(1000, 5);
  }

  @Test
  public void testMultiply() {
    MyClass tester = new MyClass();
    assertEquals("10 x 5 must be 50", 50, tester.multiply(10, 5));
  }
}
```

3.4. Run your test in Eclipse

Right-click on your new test class and select Run-As ▸ JUnit Test.



The result of the tests are displayed in the JUnit view. In our example one test should be successful and one test should show an error. This error is indicated by a red bar.



Finished after 0.012 seconds

Runs: 2/2   Errors: 0   Failures: 1

com.vogella.junit.first.MyClassTest [Runner: JUnit 4] (0.000
testExceptionIsThrown (0.000 s)
testMultiply (0.000 s)

Failure Trace
```
java.lang.AssertionError: Result expected:<50> but was:<2>
at com.vogella.junit.first.MyClassTest.testMultiply(MyClass
```

The test is failing, because our multiplier class is currently not working correctly. It does a division instead of multiplication. Fix the bug and re-run the test to get a green bar.

4. Mocking

Unit testing also makes use of object mocking. In this case the real object is exchanged by a replacement which has a predefined behavior for the test.

There are several frameworks available for mocking. To learn more about mock frameworks please see the Mockito tutorial.

10. Overview of JUnit 5

JUnit 5 is the next major release of JUnit and is still under development. JUnit 5 consists of a number of discrete components:

- JUnit Platform - foundation layer which enables different testing frameworks to be launched on the JVM
- Junit Jupiter - is the JUnit 5 test framework which is launched by JUnit Platform
- JUnit Vintage - legacy TestEngine which runs older tests

10.1. Usage of JUnit 5 with Gradle

```
buildscript {
   repositories {
      mavenCentral()
      // The following is only necessary if you want to use SNAPSHOT releases.
      // maven { url 'https://oss.sonatype.org/content/repositories/snapshots' }
   }
   dependencies {
      classpath 'org.junit.platform:junit-platform-gradle-plugin:1.0.0-M4'
   }
}

repositories {
   mavenCentral()
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'org.junit.platform.gradle.plugin'

dependencies {
   testCompile("org.junit.jupiter:junit-jupiter-api:1.0.0-M4")
```

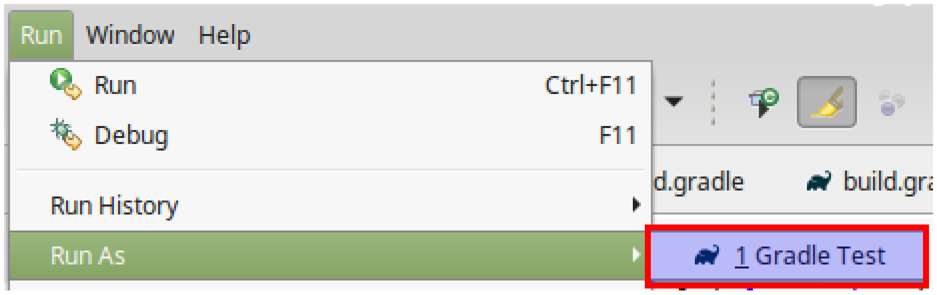
```
    testRuntime("org.junit.jupiter:junit-jupiter-engine:1.0.0-M4")
    // to run JUnit 3/4 tests:
    testCompile("junit:junit:4.12")
    testRuntime("org.junit.vintage:junit-vintage-engine:4.12.0-M4")
}
```

You can find the official gradle.build example here: https://github.com/junit-team/junit5-samples/blob/master/junit5-gradle-consumer/build.gradle

After letting gradle set up your project can then execute your JUnit 5 tests through the terminal:

```
gradle junitPlatformTest
```

If you are using Eclipse it is best to install the Buildship tooling. Then you can start your tests via Run as ▸ Gradle Test. The result of the test execution will be displayed in the Console view.



10.2. Usage of JUnit 5 with Maven

This example shows how to import all components of JUnit 5 into your project.

We need to register the individual components with Maven surefire:

```
<build>
    <plugins>
        <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.1</version>
            <configuration>
                <source>${java.version}</source>
                <target>${java.version}</target>
            </configuration>
        </plugin>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.19.1</version>
            <configuration>
                <includes>
                    <include>**/Test*.java</include>
                    <include>**/*Test.java</include>
                    <include>**/*Tests.java</include>
                    <include>**/*TestCase.java</include>
                </includes>
                <properties>
                    <!-- <includeTags>fast</includeTags> -->
                    <excludeTags>slow</excludeTags>
                </properties>
            </configuration>
            <dependencies>
                <dependency>
                    <groupId>org.junit.platform</groupId>
                    <artifactId>junit-platform-surefire-provider</artifactId>
                    <version>${junit.platform.version}</version>
                </dependency>
                <dependency>
                    <groupId>org.junit.jupiter</groupId>
                    <artifactId>junit-jupiter-engine</artifactId>
                    <version>${junit.jupiter.version}</version>
                </dependency>
                <dependency>
                    <groupId>org.junit.vintage</groupId>
                    <artifactId>junit-vintage-engine</artifactId>
                    <version>${junit.vintage.version}</version>
                </dependency>
            </dependencies>
        </plugin>
    </plugins>
</build>
```

And add the dependencies:

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>${junit.jupiter.version}</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>${junit.version}</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

You can find a complete example of a working maven configuration here: https://github.com/junit-team/junit5-samples/blob/r1.0.0-M4/junit5-maven-consumer/pom.xml

The above works for Java projects but not yet for Android projects.

10.3. Defining test methods

JUnit uses annotations to mark methods as test methods and to configure them. The following table gives an overview of the most important annotations in JUnit for the 4.x and 1.x versions. All these annotations can be used on methods.

Table 3. Annotations

import org.junit.jupiter.api.*	**Import statement for using the following annotations.**
@Test	Identifies a method as a test method.
@RepeatedTest(<Number>)	Repeats the test a <Number> of times
@TestFactory	Method is a Factory for dynamic tests
@BeforeEach	Executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class).
@AfterEach	Executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures.
@BeforeAll	Executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as static to work with JUnit.
@AfterAll	Executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as static to work with JUnit.

Table 3. Annotations

import org.junit.jupiter.api.*	**Import statement for using the following annotations.**
@Nested	Lets you nest inner test classes to force a certain execution order
@Tag("<TagName>")	Tests in JUnit 5 can be filtered by tag. Eg., run only tests tagged with "fast".
@ExtendWith	Lets you register an Extension class that integrates with one or more extension points
@Disabled or @Disabled("Why disabled")	Marks that the test should be disabled. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included. It is best practice to provide the optional description, why the test is disabled.
@DisplayName("<Name>")	<Name> that will be displayed by the test runner. In contrast to method names the DisplayName can contain spaces.

10.4. Disabling tests

The @Disable annotation allow to statically ignore a test.

Alternatively you can use Assume.assumeFalse or Assume.assumeTrue to define a condition for the test. Assume.assumeFalse marks the test as invalid, if its condition evaluates to true. Assume.assumeTrue evaluates the test as invalid if its condition evaluates to false. For example, the following disables a test on Linux:

Assume.assumeFalse(System.getProperty("os.name").contains("Linux"));

10.1. Test Suites

To run multiple tests together, you can use test suites. They allow to aggregate multiple test classes. JUnit 5 provides two annotations:

- @SelectPackages - used to specify the names of packages for the test suite

- @SelectClasses - used to specify the classes for the test suite. They can be located in different packages.

```
@RunWith(JUnitPlatform.class)
@SelectPackages("com.vogella.junit1.examples")
public class AllTests {}
```

```
@RunWith(JUnitPlatform.class)
@SelectClasses({AssertionTest.class, AssumptionTest.class, ExceptionTest.class})
public class AllTests {}
```

10.6. Expecting Exceptions

Exception is handling with org.junit.jupiter.api.Assertions.expectThrows(). You define the expected Exception class and provide code that should throw the exception.

```
import static org.junit.jupiter.api.Assertions.expectThrows;

@Test
void exceptionTesting() {
   // set up user
   Throwable exception = expectThrows(IllegalArgumentException.class, () -> user.setAge("23"));
   assertEquals("Age must be an Integer.", exception.getMessage());
}
```

This lets you define which part of the test should throw the exception. The test will still fail if an exception is thrown outside of this scope.

10.7. Grouped assertions

```
@Test
void groupedAssertions() {
   Address address = new Address();
   // In a grouped assertion all assertions are executed, even after a failure.
   // The error messages get grouped together.
   assertAll("address name",
      () -> assertEquals("John", address.getFirstName()),
      () -> assertEquals("User", address.getLastName())
   );
}
   => org.opentest4j.MultipleFailuresError: address name (2 failures)
   expected: <John> but was: <null>
   expected: <User> but was: <null>
```

10.3. Timeout tests

If you want to ensure that a test fails if it isn't done in a certain amount of time you can use the assertTimeout() method. This method will wait until

```
import static org.junit.jupiter.api.Assertions.assertTimeout;
import static java.time.Duration.ofSeconds;
import static java.time.Duration.ofMinutes;
```

```
@Test
void timeoutNotExceeded() {
    assertTimeout(ofMinutes(1), () -> service.doBackup());
}

// if you have to check a return value
@Test
void timeoutNotExceededWithResult() {
    String actualResult = assertTimeout(ofSeconds(1), () -> {
        return restService.request(request);
    });
    assertEquals(200, request.getStatus());
}
=> org.opentest4j.AssertionFailedError: execution exceeded timeout of 1000 ms by 212 ms
```

If you want your tests to cancel after the timeout period is passed you can use the assertTimeoutPreemptively() method.

```
@Test
void timeoutNotExceededWithResult() {
    String actualResult = assertTimeoutPreemptively(ofSeconds(1), () -> {
        return restService.request(request);
    });
    assertEquals(200, request.getStatus());
}
=> org.opentest4j.AssertionFailedError: execution timed out after 1000 ms
```

10.9. Running the same test repeatedly on a data set

Sometimes we want to be able to run the same test on a data set. Holding the data set in a Collection and iterating over it with the assertion in the loop body has the problem that the first assertion failure will stop the test execution. In JUnit 4 this was done with parameterized tests, we will reuse the example used there:

```
package testing;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;

import java.util.Arrays;
import java.util.Collection;

import static org.junit.Assert.assertEquals;
import static org.junit.runners.Parameterized.*;

@RunWith(Parameterized.class)
public class ParameterizedTestFields {
    // fields used together with @Parameter must be public
    @Parameter(0)
    public int m1;
    @Parameter(1)
    public int m2;
    @Parameter(2)
    public int result;


    // creates the test data
    @Parameters
    public static Collection<Object[]> data() {
        Object[][] data = new Object[][] { { 1 , 2, 2 }, { 5, 3, 15 }, { 121, 4, 484 } };
        return Arrays.asList(data);
    }


    @Test
    public void testMultiplyException() {
        MyClass tester = new MyClass();
        assertEquals("Result", result, tester.multiply(m1, m2));
    }


    // class to be tested
    class MyClass {
        public int multiply(int i, int j) {
            return i *j;
        }
    }

}
```

APPENDIX

Unit tests with Mockito - Tutorial

Lars Vogel, Fabian Pfaff (c) 2012, 2016 vogella GmbH version 1.9, 10.04.2017

Table of Contents

- 1. Prerequisites
- 2. Testing with mock objects
- 3. Adding Mockito as dependencies to a project
- 4. Using the Mockito API

- 5. Exercise: Write an instrumented unit test using Mockito
- 6. Exercise: Creating mock objects using Mockito
- 7. Using PowerMock with Mockito
- 8. Using a wrapper instead of Powermock
- 9. Mockito resources

This tutorial explains testing with the Mockito framework for writing software tests.

1. Prerequisites

The following tutorial is based on an understanding of unit testing with the JUnit framework.

In case your are not familiar with JUnit please check the following JUnit Tutorial.

2. Testing with mock objects

2.1. Target and challenge of unit testing

A unit test should test functionality in isolation. Side effects from other classes or the system should be eliminated for a unit test, if possible.

This can be done via using test replacements (*test doubles*) for the real dependencies. Test doubles can be classified like the following:

- A *dummy object* is passed around but never used, i.e., its methods are never called. Such an object can for example be used to fill the parameter list of a method.
- *Fake* objects have working implementations, but are usually simplified. For example, they use an in memory database and not a real database.
- A *stub* class is an partial implementation for an interface or class with the purpose of using an instance of this stub class during testing. Stubs usually don't respond to anything outside what's programmed in for the test. Stubs may also record information about calls.
- A *mock object* is a dummy implementation for an interface or a class in which you define the output of certain method calls. Mock objects are configured to perform a certain behavior during a test. They typical record the interaction with the system and test can validate that.

Test doubles can be passed to other objects which are tested. Your tests can validate that the class reacts correctly during tests. For example, you can validate if certain methods on the mock object were called. This helps to ensure that you only test the class while running tests and that your tests are not affected by any side effects.

Mock objects typically require less code to configure and should therefore be preferred.

2.2. Mock object generation

You can create mock objects manually (via code) or use a mock framework to simulate these classes. Mock frameworks allow you to create mock objects at runtime and define their behavior.

The classical example for a mock object is a data provider. In production an implementation to connect to the real data source is used. But for testing a mock object simulates the data source and ensures that the test conditions are always the same.

These mock objects can be provided to the class which is tested. Therefore, the class to be tested should avoid any hard dependency on external data.

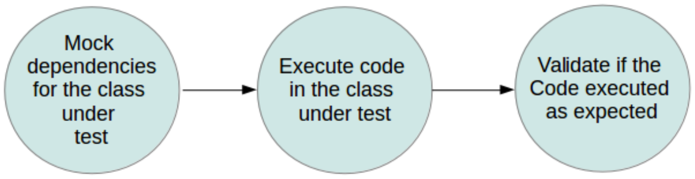
Mocking or mock frameworks allows testing the expected interaction with the mock object. You can, for example, validate that only certain methods have been called on the mock object.

2.3. Using Mockito for mocking objects

Mockito is a popular mock framework which can be used in conjunction with JUnit. Mockito allows you to create and configure mock objects. Using Mockito simplifies the development of tests for classes with external dependencies significantly.

If you use Mockito in tests you typically:

- Mock away external dependencies and insert the mocks into the code under test
- Execute the code under test
- Validate that the code executed correctly


Mock dependencies for the class under test
Execute code in the class under test
Validate if the Code executed as expected

3. Adding Mockito as dependencies to a project

3.1. Using Gradle for a Java project

If you use Gradle in a Java project, add the following dependency to the Gradle build file.

```
repositories { jcenter() }
dependencies { testCompile 'org.mockito:mockito-core:2.7.22' }
```

3.2. Using Gradle for an Android project

Add the following dependency to the Gradle build file:

```
dependencies {
  // ... more entries
  testCompile 'junit:junit:4.12'

  // required if you want to use Mockito for unit tests
  testCompile 'org.mockito:mockito-core:2.7.22'
  // required if you want to use Mockito for Android tests
  androidTestCompile 'org.mockito:mockito-android:2.7.22'
}
```

3.3. Using Maven

Maven users can declare a dependency. Search for g:"org.mockito", a:"mockito-core" via the http://search.maven.org website to find the correct pom entry.

3.4. Using the Eclipse IDE

The Eclipse IDE supports the Gradle as well as the Maven build system. These build system allow to manage your software dependencies. Therefore, you are advised to use either the Gradle or Maven tooling in Eclipse.

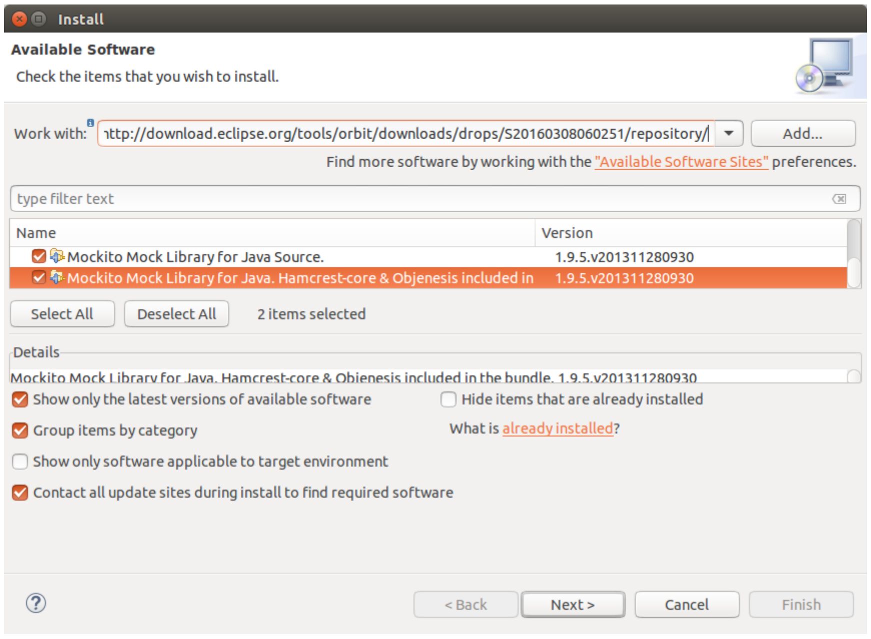
3.5. Using IntelliJ

If you are using IntelliJ, you should use either Gradle or Maven to manage your dependencies to Mockito.

3.6. OSGi or Eclipse plug-in development

In Eclipse RCP applications dependencies are usually obtained from p2 update sites. The Orbit repositories are a good source for third party libraries, which can be used in Eclipse based applications or plug-ins.

The Orbit repositories can be found here http://download.eclipse.org/tools/orbit/downloads



Work with: ıttp://download.eclipse.org/tools/orbit/downloads/drops/S20160308060251/repository/

Find more software by working with the "Available Software Sites" preferences.

Name	Version
✓ Mockito Mock Library for Java Source.	1.9.5.v201311280930
✓ Mockito Mock Library for Java. Hamcrest-core & Objenesis included in	1.9.5.v201311280930

2 items selected

Details

Mockito Mock Library for Java. Hamcrest-core & Objenesis included in the bundle. 1.9.5.v201311280930

✓ Show only the latest versions of available software

☐ Hide items that are already installed

✓ Group items by category

What is already installed?

☐ Show only software applicable to target environment

✓ Contact all update sites during install to find required software

4. Using the Mockito API

4.1. Creating mock objects with Mockito

Mockito provides several methods to create mock objects:

- Using the static mock() method.
- Using the @Mock annotation.

If you use the @Mock annotation, you must trigger the creation of annotated objects. The MockitoRule allows this. It invokes the static method MockitoAnnotations.initMocks(this) to populate the annotated fields. Alternatively you can use @RunWith(MockitoJUnitRunner.class).

The usage of the @Mock annotation and the MockitoRule rule is demonstrated by the following example.

```
import static org.mockito.Mockito.*;

public class MockitoTest {

  @Mock
  MyDatabase databaseMock;

  @Rule public MockitoRule mockitoRule = MockitoJUnit.rule();

  @Test
  public void testQuery() {
    ClassToTest t  = new ClassToTest(databaseMock);
    boolean check = t.query("* from t");
    assertTrue(check);
    verify(databaseMock).query("* from t");
  }
}
```

Tells Mockito to mock the databaseMock instance

Tells Mockito to create the mocks based on the @Mock annotation

Instantiates the class under test using the created mock

Executes some code of the class under test

Asserts that the method call returned true

Verify that the query method was called on the MyDatabase mock

Static imports

By adding the org.mockito.Mockito.*; static import, you can use methods like mock() directly in your tests. Static imports allow you to call static members, i.e., methods and fields of a class directly without specifying the class.

Using static imports greatly improves the readability of your test code, you should use it.

4.2. Configuring mocks

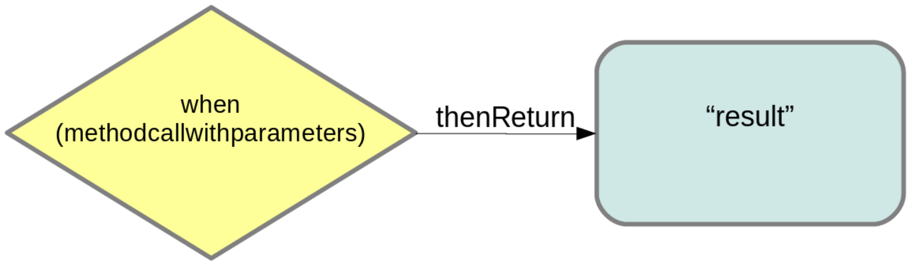
Mockito allows to configure the return values of its mocks via a fluent API. Unspecified method calls return "empty" values:

- null for objects
- 0 for numbers
- false for boolean
- empty collections for collections
- …

The following assert statements are only for demonstration purposes, a real test would use the mocks to unit test another functionality.

4.2.1. "when thenReturn" and "when thenThrow

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



You also can use methods like anyString or anyInt to define that dependent on the input type a certain value should be returned.

If you specify more than one value, they are returned in the order of specification, until the last one is used. Afterwards the last specified value is returned.

The following demonstrates the usage of when(….).thenReturn(….).

```
import static org.mockito.Mockito.*;
import static org.junit.Assert.*;

@Test
public void test1() {
    //  create mock
    MyClass test = mock(MyClass.class);

    // define return value for method getUniqueId()
    when(test.getUniqueId()).thenReturn(43);
    // use mock in test....
    assertEquals(test.getUniqueId(), 43);
}


// demonstrates the return of multiple values
@Test
public void testMoreThanOneReturnValue() {
    Iterator<String> i= mock(Iterator.class);
    when(i.next()).thenReturn("Mockito").thenReturn("rocks");
    String result= i.next()+"  "+i.next();
    //assert
    assertEquals("Mockito rocks", result);
}

// this test demonstrates how to return values based on the input
@Test
public void testReturnValueDependentOnMethodParameter() {
    Comparable<String> c= mock(Comparable.class);
    when(c.compareTo("Mockito")).thenReturn(1);
    when(c.compareTo("Eclipse")).thenReturn(2);
    //assert
    assertEquals(1, c.compareTo("Mockito"));
}

// this test demonstrates how to return values independent of the
input value

@Test
public void testReturnValueInDependentOnMethodParameter() {
    Comparable<Integer> c= mock(Comparable.class);
    when(c.compareTo(anyInt())).thenReturn(-1);
    //assert
    assertEquals(-1, c.compareTo(9));
}

// return a value based on the type of the provide parameter

@Test
public void testReturnValueInDependentOnMethodParameter2() {
    Comparable<Todo> c= mock(Comparable.class);
    when(c.compareTo(isA(Todo.class))).thenReturn(0);
    //assert
    assertEquals(0, c.compareTo(new Todo(1)));
}
```

The when(….).thenReturn(….) method chain can be used to throw an exception.

```
Properties properties = mock(Properties.class);

when(properties.get(”Anddroid”)).thenThrow(new IllegalArgumentException(...));

try {
   properties.get(”Anddroid”);
   fail(”Anddroid is misspelled”);
} catch (IllegalArgumentException ex) {
  // good!
}
```

4.2.2. "doReturn when" and "doThrow when"

The doReturn(…).when(…).methodCall call chain works similar to when(….).thenReturn(….). It is useful for mocking methods which give an exception during a call, e.g., if you use use functionality like Wrapping Java objects with Spy.

doReturnWhen.java

The doThrow variant can be used for methods which return void to throw an exception. This usage is demonstrated by the following code snippet.

```
Properties properties = new Properties();

Properties spyProperties = spy(properties);

doReturn(“42”).when(spyProperties).get(”shoeSize”);

String value = spyProperties.get(”shoeSize”);

assertEquals(”42”, value);
```

4.3. Wrapping Java objects with Spy

@Spy or the spy() method can be used to wrap a real object. Every call, unless specified otherwise, is delegated to the object.

```
import static org.mockito.Mockito.*;

@Test
public void testLinkedListSpyWrong() {
  // Lets mock a LinkedList
  List<String> list = new LinkedList<>();
  List<String> spy = spy(list);

  // this does not work
  // real method is called so spy.get(0)
  // throws IndexOutOfBoundsException (list is still empty)
  when(spy.get(0)).thenReturn("foo");

  assertEquals("foo", spy.get(0));
}

@Test
public void testLinkedListSpyCorrect() {
  // Lets mock a LinkedList
  List<String> list = new LinkedList<>();
  List<String> spy = spy(list);

  // You have to use doReturn() for stubbing
  doReturn("foo").when(spy).get(0);

  assertEquals("foo", spy.get(0));
}
```

4.4. Verify the calls on the mock objects

Mockito keeps track of all the method calls and their parameters to the mock object. You can use the verify() method on the mock object to verify that the specified conditions are met. For example, you can verify that a method has been called with certain parameters. This kind of testing is sometimes called *behavior testing*. Behavior testing does not check the result of a method call, but it checks that a method is called with the right parameters.

```
import static org.mockito.Mockito.*;

@Test
public void testVerify()  {
  // create and configure mock
  MyClass test = Mockito.mock(MyClass.class);
  when(test.getUniqueId()).thenReturn(43);


  // call method testing on the mock with parameter 12
  test.testing(12);
  test.getUniqueId();
  test.getUniqueId();


  // now check if method testing was called with the parameter 12
  verify(test).testing(ArgumentMatchers.eq(12));

  // was the method called twice?
  verify(test, times(2)).getUniqueId();
  // other alternatives for verifiying the number of method calls for a
method
  verify(test, never()).someMethod("never called");
  verify(test, atLeastOnce()).someMethod("called at least once");
  verify(test, atLeast(2)).someMethod("called at least twice");
  verify(test, times(5)).someMethod("called five times");
  verify(test, atMost(3)).someMethod("called at most 3 times");
  // This let's you check that no other methods where called on this
object.
  // You call it after you have verified the expected method calls.
  verifyNoMoreInteractions(test);
}
```

In case you do not care about the value, use the anyX, e.g., anyInt, anyString(), or any(YourClass.class) methods.

4.5. Using @InjectMocks for dependency injection via Mockito

You also have the @InjectMocks annotation which tries to do constructor, method or field dependency injection based on the type. For example, assume that you have the following class.

```
public class ArticleManager {
  private User user;
  private ArticleDatabase database;

  public ArticleManager(User user, ArticleDatabase database) {
    super();
    this.user = user;
    this.database = database;
  }

  public void initialize() {
    database.addListener(new ArticleListener());
  }
}
```

This class can be constructed via Mockito and its dependencies can be fulfilled with mock objects as demonstrated by the following code snippet.

```
@RunWith(MockitoJUnitRunner.class)
public class ArticleManagerTest {

     @Mock ArticleCalculator calculator;
     @Mock ArticleDatabase database;
     @Mock User user;

     @Spy private UserProvider userProvider = new ConsumerUserProvider();
```

```
    @InjectMocks private ArticleManager manager;

    @Test public void shouldDoSomething() {
      // calls addListener with an instance of ArticleListener
      manager.initialize();

      // validate that addListener was called
      verify(database).addListener(any(ArticleListener.class));
    }
}
```

creates an instance of ArticleManager and injects the mocks into it

Mockito can inject mocks either via constructor injection, setter injection, or property injection and in this order. So if ArticleManager would have a constructor that would only take User and setters for both fields, only the mock for User would be injected.

4.6. Capturing the arguments

The ArgumentCaptor class allows to access the arguments of method calls during the verification. This allows to capture these arguments of method calls and to use them for tests.

To run this example you need to add hamcrest-library to your project.

```
import static org.hamcrest.Matchers.hasItem;
import static org.junit.Assert.assertThat;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.verify;

import java.util.Arrays;
import java.util.List;

import org.junit.Rule;
import org.junit.Test;
import org.mockito.ArgumentCaptor;
import org.mockito.Captor;
import org.mockito.junit.MockitoJUnit;
import org.mockito.junit.MockitoRule;


public class MockitoTests {

  @Rule
  public MockitoRule rule = MockitoJUnit.rule();

  @Captor
  private ArgumentCaptor<List<String>> captor;


  @Test
  public final void shouldContainCertainListItem() {
    List<String> asList = Arrays.asList("someElement_test", "someElement");
    final List<String> mockedList = mock(List.class);
    mockedList.addAll(asList);

    verify(mockedList).addAll(captor.capture());
    final List<String> capturedArgument = captor.getValue();
    assertThat(capturedArgument, hasItem("someElement"));
  }
}
```

4.7. Using Answers for complex mocks

It is possible to define a Answer object for complex results. While thenReturn returns a predefined value every time, with answers you can calculate a response based on the arguments given to your stubbed method. This can be useful if your stubbed method is supposed to call a function on one of the arguments or if your method is supposed to return the first argument to allow method chaining. There exists a static method for the latter. Also note that there a different ways to configure an answer:

```
import static org.mockito.AdditionalAnswers.returnsFirstArg;

@Test
public final void answerTest() {
  // with doAnswer():
  doAnswer(returnsFirstArg()).when(list).add(anyString());
  // with thenAnswer():
  when(list.add(anyString())).thenAnswer(returnsFirstArg());
  // with then() alias:
  when(list.add(anyString())).then(returnsFirstArg());
}
```

Or if you need to do a callback on your argument:

```
@Test
public final void callbackTest() {
  ApiService service = mock(ApiService.class);
  when(service.login(any(Callback.class))).thenAnswer(i -> {
    Callback callback = i.getArgument(0);
    callback.notify("Success");
    return null;
  });
}
```

It is even possible to mock a persistence service like an DAO, but you should consider creating a fake class instead of a mock if your Answers become too complex.

```
List<User> userMap = new ArrayList<>();
UserDao dao = mock(UserDao.class);
when(dao.save(any(User.class))).thenAnswer(i -> {
   User user = i.getArgument(0);
   userMap.add(user.getId(), user);
   return null;
});
when(dao.find(any(Integer.class))).thenAnswer(i -> {
   int id = i.getArgument(0);
   return userMap.get(id);
});
```

4.8. Mocking final classes

Since Mockito v2 it is possible to mock final classes. This feature is incubating and is deactivated by default. To activate the mocking of final classes create the file org.mockito.plugins.MockMaker in either src/test/resources/mockito-extensions/ or src/mockito-extensions/. Add this line to the file: *mock-maker-inline*. With this modification we now can mock a final class.

```
final class FinalClass {
  public final String finalMethod() { return "something"; }
}

@Test
public final void mockFinalClassTest() {
    FinalClass instance = new FinalClass();

    FinalClass mock = mock(FinalClass.class);
    when(mock.finalMethod()).thenReturn("that other thing");

    assertNotEquals(mock.finalMethod(), instance.finalMethod());
}
```

4.9. Clean test code with the help of the strict stubs rule

The strict stubs rule helps you to keep your test code clean and checks for common oversights. It adds the following:

- test fails early when a stubbed method gets called with different arguments than what it was configured for (with PotentialStubbingProblem exception).
- test fails when a stubbed method isn't called (with UnnecessaryStubbingException exception).
- org.mockito.Mockito.verifyNoMoreInteractions(Object) also verifies that all stubbed methods have been called during the test

```
@Test
public void withoutStrictStubsTest() throws Exception {
  DeepThought deepThought = mock(DeepThought.class);

  when(deepThought.getAnswerFor("Ultimate Question of Life, The Universe, and
Everything")).thenReturn(42);
  when(deepThought.otherMethod("some mundane thing")).thenReturn(null);

  System.out.println(deepThought.getAnswerFor("Six by nine"));

  assertEquals(42, deepThought.getAnswerFor("Ultimate Question of Life, The
Universe, and Everything"));
  verify(deepThought, times(1)).getAnswerFor("Ultimate Question of Life, The
Universe, and Everything");
}
// activate the strict subs rule
@Rule public MockitoRule rule = MockitoJUnit.rule().strictness(Strictness.STRICT_STUBS);

@Test
public void withStrictStubsTest() throws Exception {
  DeepThought deepThought = mock(DeepThought.class);

  when(deepThought.getAnswerFor("Ultimate Question of Life, The Universe, and
Everything")).thenReturn(42);
  // this fails now with an UnnecessaryStubbingException since it is
never called in the test
  when(deepThought.otherMethod("some mundane thing")).thenReturn(null);

  // this will now throw a PotentialStubbingProblem Exception since we
usually don't want to call methods on mocks without configured
behavior
  deepThought.someMethod();

  assertEquals(42, deepThought.getAnswerFor("Ultimate Question of Life, The
Universe, and Everything"));
  // verifyNoMoreInteractions now automatically verifies that all
stubbed methods have been called as well
  verifyNoMoreInteractions(deepThought);
}
```

4.10. Limitations

Mockito has certain limitations. For example, you cannot mock static methods and private methods.

See FAQ for Mockito limitations for the details

4.11. Behavior testing vs. state testing

Mockito puts a focus on behavior testing, vs. result testing. This is not always correct, for example, if you are testing a sort algorithm, you should test the result not the internal behavior.

```
// state testing
testSort() {
  testList = [1, 7, 3, 8, 2]
  MySorter.sort(testList)

  assert testList equals [1, 2, 3, 7, 8]
}


// incorrect would be behavior testing
// the following tests internal of the implementation
testSort() {
  testList = [1, 7, 3, 8, 2]
  MySorter.sort(testList)

  assert that compare(1, 2) was called once
  assert that compare(1, 3) was not called
  assert that compare(2, 3) was called once
  ....
}
```

5. Exercise: Write an instrumented unit test using Mockito

5.1. Create Application under tests on Android

Create an Android application with the package name com.vogella.android.testing.mockito.contextmock. Add a Util class with a static method which allows to create an intent with certain parameters as in the following example.

```
package com.vogella.android.testing.mockito.contextmock;

import android.content.Context;
import android.content.Intent;

public class Util {

  public static Intent createQuery(Context context, String query, String value) {
    // Reuse MainActivity for simplification
    Intent i = new Intent(context, MainActivity.class);
    i.putExtra("QUERY", query);
    i.putExtra("VALUE", value);
    return i;
  }
}
```

5.2. Add the Mockito dependency to the app/build.gradle file

```
dependencies {
  // ... more entries
  testCompile 'junit:junit:4.12'

  // required if you want to use Mockito for unit tests
  testCompile 'org.mockito:mockito-core:2.7.22'
  // required if you want to use Mockito for Android tests
  androidTestCompile 'org.mockito:mockito-android:2.7.22'
}
```

5.3. Create test

Create a new unit test running on Android using Mockito in the androidTest folder. This test should check if the intent contains the correct extras. For this you mock the Context object with Mockito.

```
package com.vogella.android.testing.mockito.contextmock;

import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.support.test.runner.AndroidJUnit4;

import org.junit.Test;
import org.junit.runner.RunWith;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import static org.mockito.Mockito.mock;

@RunWith(AndroidJUnit4.class)
public class UtilTest2 {

  @Test
  public void shouldContainTheCorrectExtras() throws Exception {
    Context context = mock(Context.class);
    Intent intent = Util.createQuery(context, "query", "value");
    assertNotNull(intent);
    Bundle extras = intent.getExtras();
    assertNotNull(extras);
    assertEquals("query", extras.getString("QUERY"));
    assertEquals("value", extras.getString("VALUE"));
  }
}
```

6. Exercise: Creating mock objects using Mockito

6.1. Target

Create an API, which can be mocked and use Mockito to do the job.

6.2. Create a sample Twitter API

Implement a TwitterClient, which works with ITweet instances. But imagine these ITweet instances are pretty cumbersome to get, e.g., by using a complex service, which would have to be started.

```
public interface ITweet {

  String getMessage();
}
public class TwitterClient {

  public void sendTweet(ITweet tweet) {
    String message = tweet.getMessage();

    // send the message to Twitter
  }
}
```

6.3. Mocking ITweet instances

In order to avoid starting up a complex service to get ITweet instances, they can also be mocked by Mockito.

```
@Test
public void testSendingTweet() {
  TwitterClient twitterClient = new TwitterClient();

  ITweet iTweet = mock(ITweet.class);

  when(iTweet.getMessage()).thenReturn("Using mockito is great");

  twitterClient.sendTweet(iTweet);
}
```

Now the TwitterClient can make use of a mocked ITweet instance and will get "Using Mockito is great" as message when calling getMessage() on the mocked ITweet.

6.4. Verify method invocation

Ensure that getMessage() is at least called once.

```
@Test
public void testSendingTweet() {
  TwitterClient twitterClient = new TwitterClient();

  ITweet iTweet = mock(ITweet.class);

  when(iTweet.getMessage()).thenReturn("Using mockito is great");
```

```
   twitterClient.sendTweet(iTweet);

   verify(iTweet, atLeastOnce()).getMessage();
}
```

6.5. Validate

Run the test and validate that it is successful.

7. Using PowerMock with Mockito

7.1. Powermock for mocking static methods

Mockito cannot mock static methods. For this you can use Powermock. PowerMock provides a class called "PowerMockito" for creating mock/object/class and initiating verification, and expectations, everything else you can still use Mockito to setup and verify expectation (e.g. times(), anyInt()).

```
import java.net.InetAddress;
import java.net.UnknownHostException;

public final class NetworkReader {
  public static String getLocalHostname() {
    String hostname = "";
    try {
      InetAddress addr = InetAddress.getLocalHost();
      // Get hostname
      hostname = addr.getHostName();
    } catch ( UnknownHostException e ) {
    }
    return hostname;
  }
}
```

To write a test which mocks away the NetworkReader as dependency you can use the following snippet.

```
import org.junit.runner.RunWith;
import org.powermock.core.classloader.annotations.PrepareForTest;

@RunWith( PowerMockRunner.class )
@PrepareForTest( NetworkReader.class )
public class MyTest {

// Find the tests here

 @Test
public void testSomething() {
    mockStatic( NetworkUtil.class );
    when( NetworkReader.getLocalHostname() ).andReturn( "localhost" );

    // now test the class which uses NetworkReader
}
```

8. Using a wrapper instead of Powermock

Sometimes you can also use a wrapper around a static method, which can be mocked with Mockito.

```
class FooWraper {
  void someMethod() {
    Foo.someStaticMethod()
  }
}
```

8.1. Learn more about Powermock

See Using PowerMock with Mockito for more information

Get the source code

9. Mockito resources

Mockito home page