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.

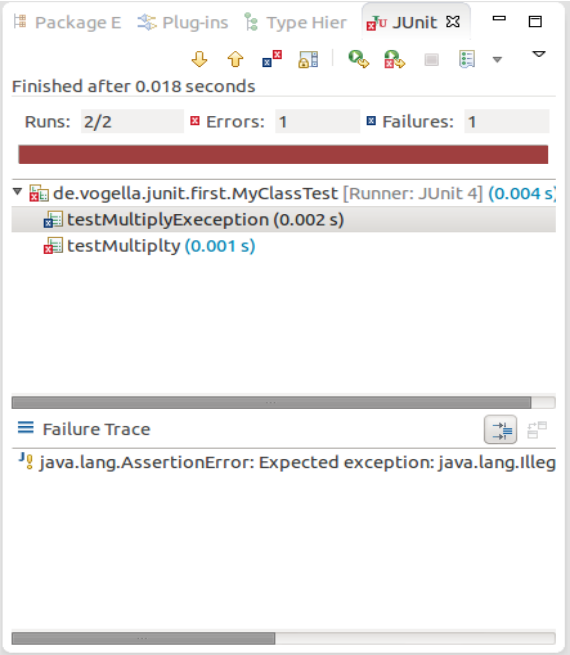
To run a test, select the test class, right-click on it and select Run-as ▸ JUnit Test.

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

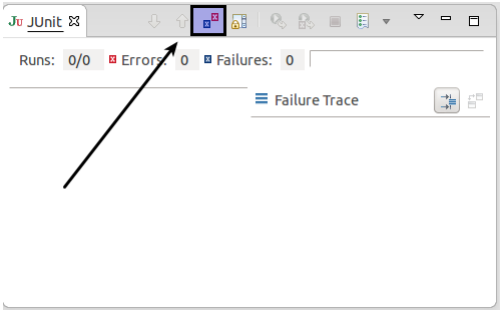
Eclipse provides the Alt + Shift + X, T shortcut to run the test in the selected class.

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

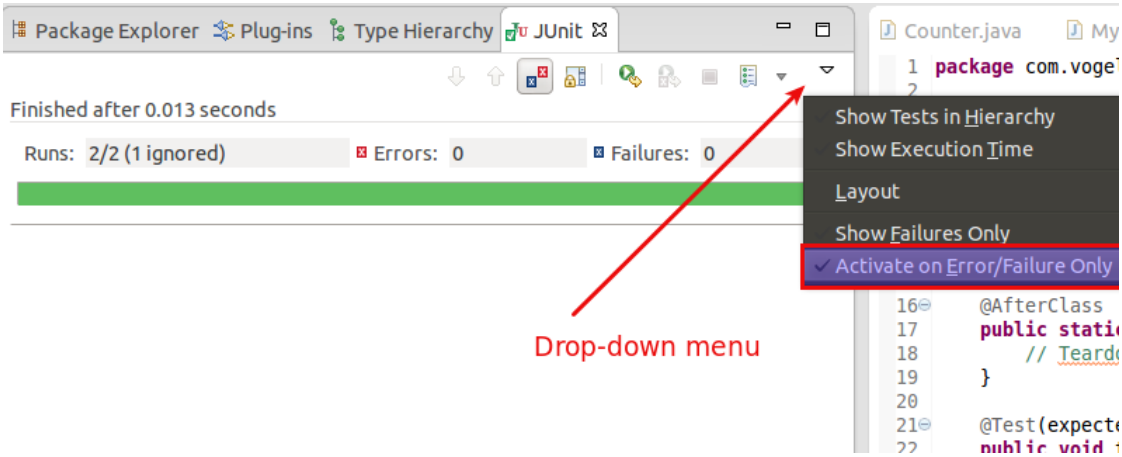
To see the result of a JUnit test, Eclipse uses the *JUnit* view which shows the results of the tests. You can also select individual unit tests in this view, right-click on them and select *Run* to execute them again.



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



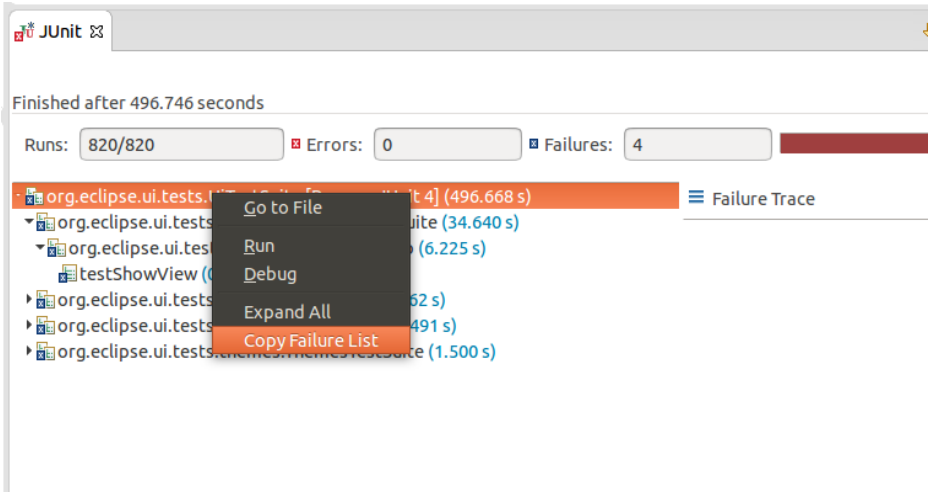
You can also define that the view is only activated if you have a failing test.



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

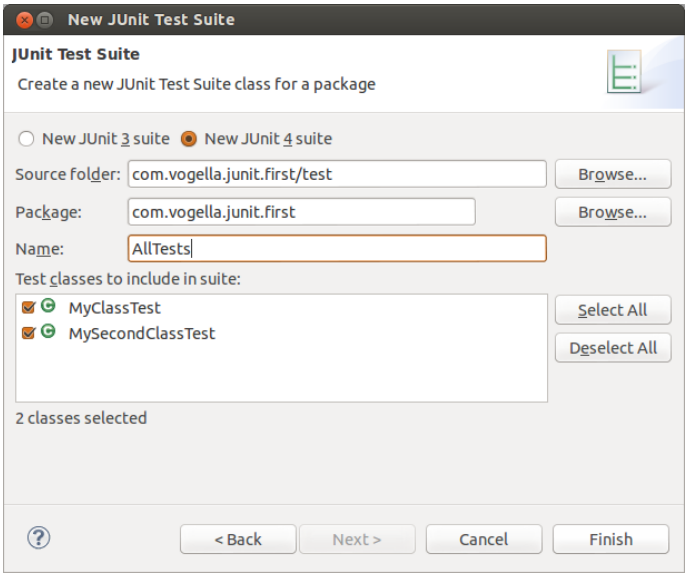
Static import is a feature that allows fields and methods defined in a class as public static to be used without specifying the class in which the field is defined.

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.



1.6. Testing exception

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

```
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

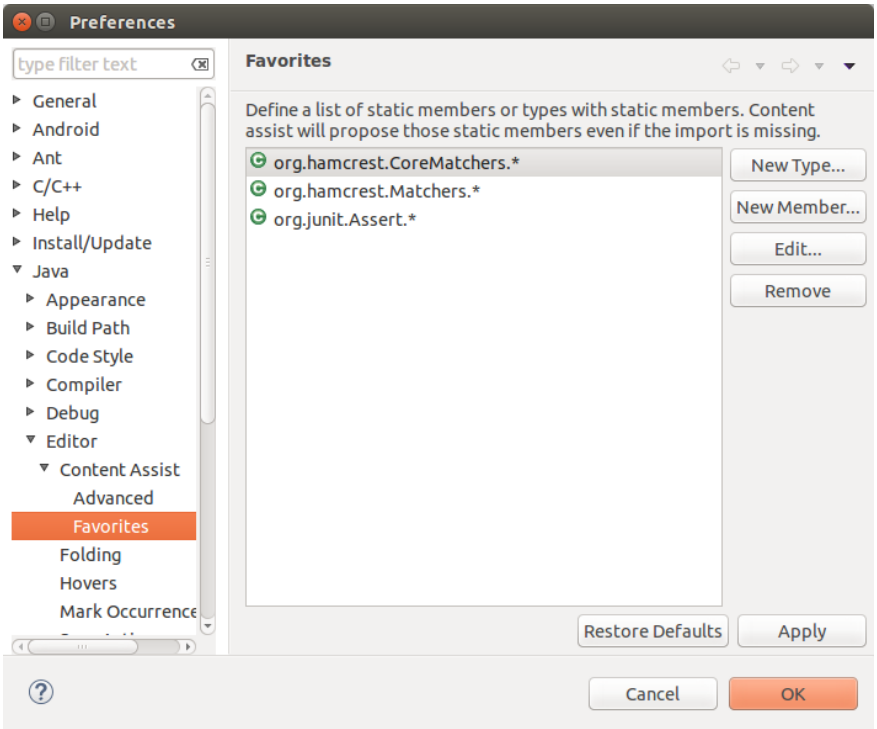
The Eclipse IDE cannot always create the corresponding static import statements automatically.

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*.

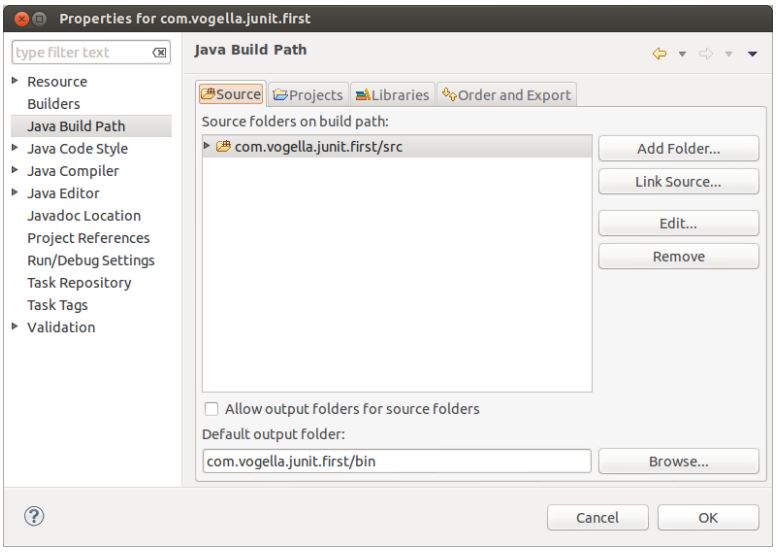


You can now use *Content Assists* (shortcut: Ctrl + Space) to add the method and the import.

3. Exercise: Using JUnit

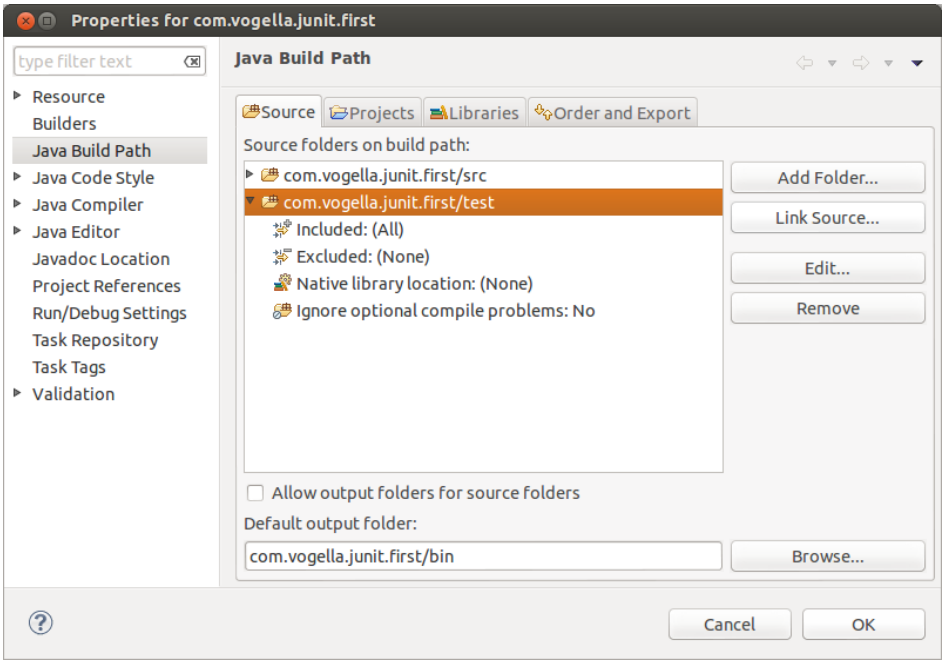
3.1. Project preparation

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.



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.



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

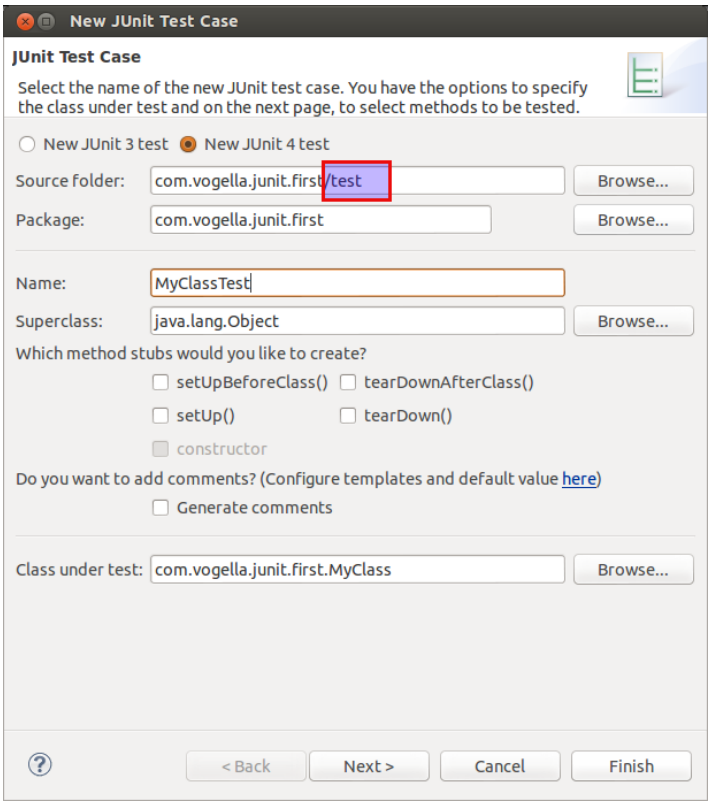
In the *src* folder, create the com.vogella.junit.first package and the following class.

```
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;
  }
}
```

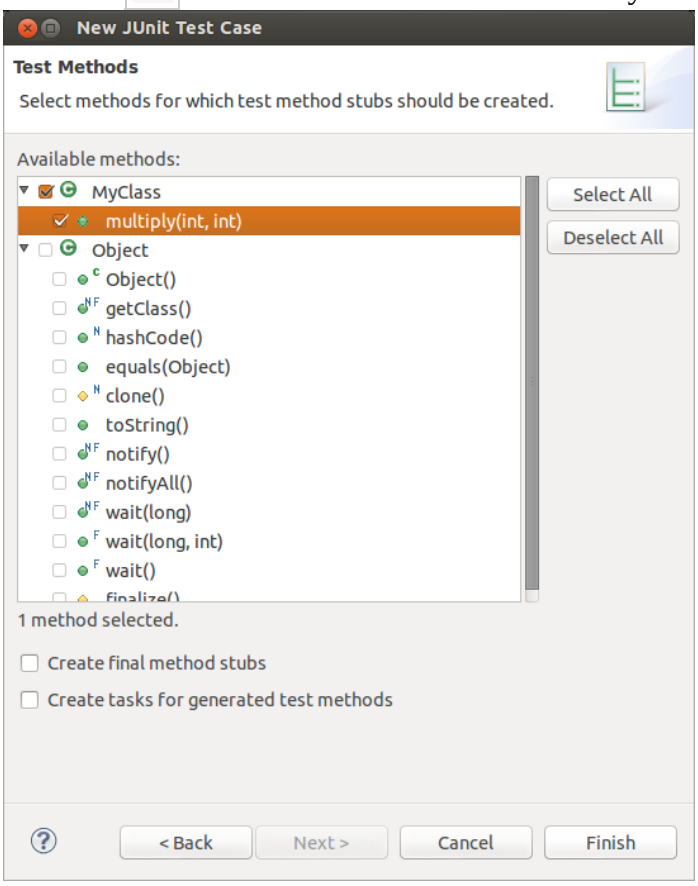
3.3. Create a JUnit test

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

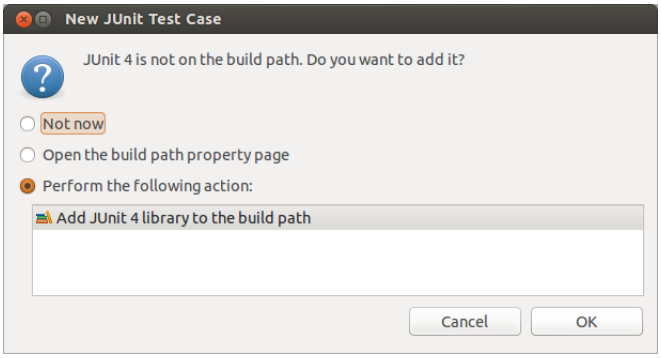
In the following wizard ensure that the *New JUnit 4 test* flag is selected and set the source folder to *test*, so that your test class gets created in this folder.



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



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.



Create a test with the following code.

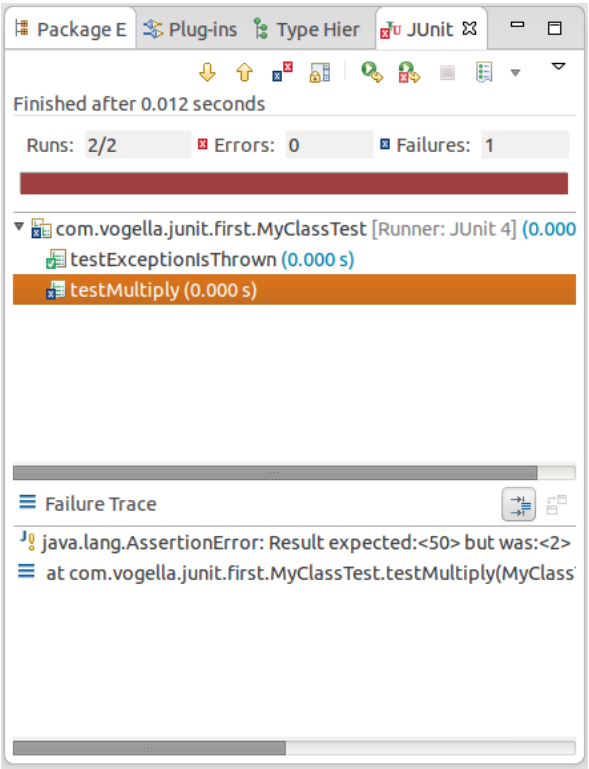
```
package com.vogella.junit.first;

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.



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")
```

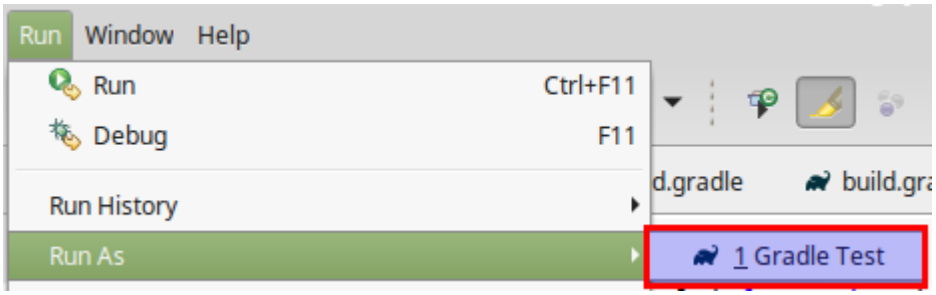
```
    testRuntime("org.junit.jupiter:junit-jupiter-engine:1.0.0-M4")
    // to run JUnit 3/4 tests:
    testCompile("junit:junit:4.12")
    testRuntime("org.junit.vintage:junit-vintage-engine:4.12.0-M4")
}
```

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;
        }
    }

}
```