Using JUnit 4/5 in Maven Project

# Maven Project

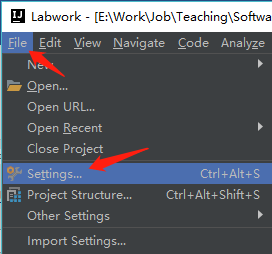
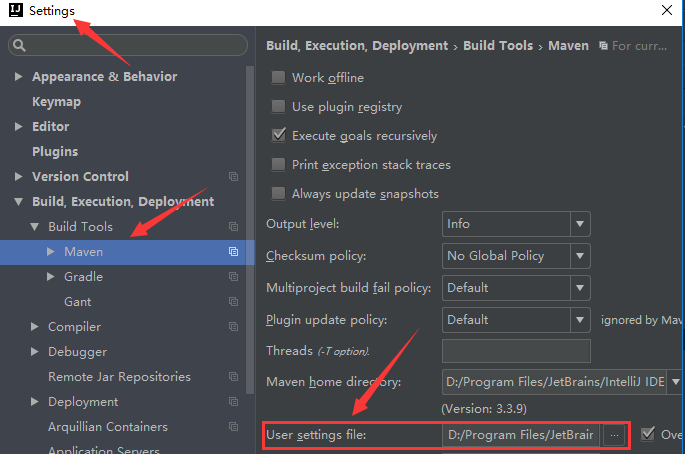
## What is Maven? Why do we use Maven?

Please read: <https://maven.apache.org/what-is-maven.html>

## Mirror Configuration for Maven

Please read: <https://maven.apache.org/guides/mini/guide-mirror-settings.html>

1. Find the location of settings.xml: Open Project Settings by clicking “File🡪Settings…”, and then you can find the location of the settings.xml as follows:

1. You might need to restart your IDE after revising settings.xml.
2. Repositories can be [declared inside a project](https://maven.apache.org/guides/mini/guide-multiple-repositories.html), which means that if you have your own custom repositories, those sharing your project easily get the right settings out of the box. However, you may want to use **an alternative mirror for a particular repository** **without changing the project files**. To configure a mirror of a given repository, you provide it in your settings.xml: giving the new repository its own id and url, and specify the mirrorOf setting that is the ID of the repository you are using a mirror of. In China, we usually choose **Aliyun** as our repository mirror site. You can configure it in settings.xml as follows:

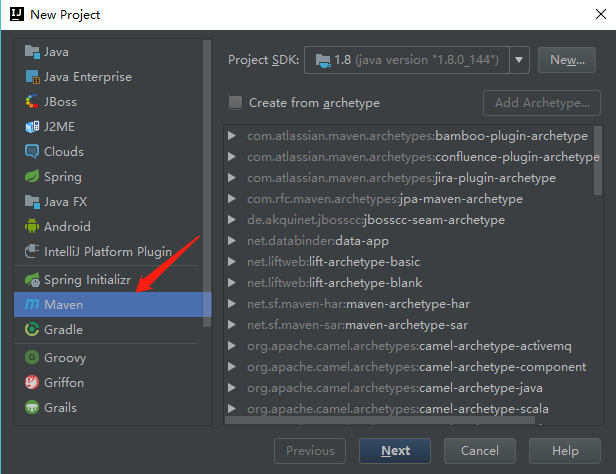
<mirrors>  
<mirror>  
 <id>nexus-aliyun</id>  
 <mirrorOf>\*</mirrorOf>  
 <name>Nexus aliyun</name>  
 <url>http://maven.aliyun.com/nexus/content/groups/public</url>  
</mirror>

</mirrors>

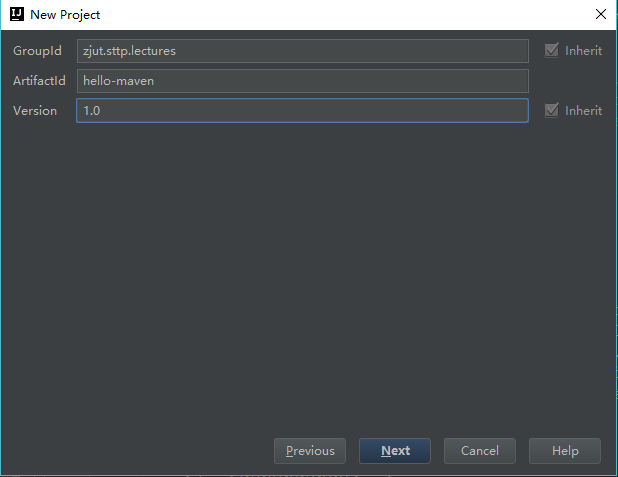
1. As an example, I also give you my own settings.xml. See Lectures/Lec03-JUnit/settings.xml.

## Create Your Own Maven Project

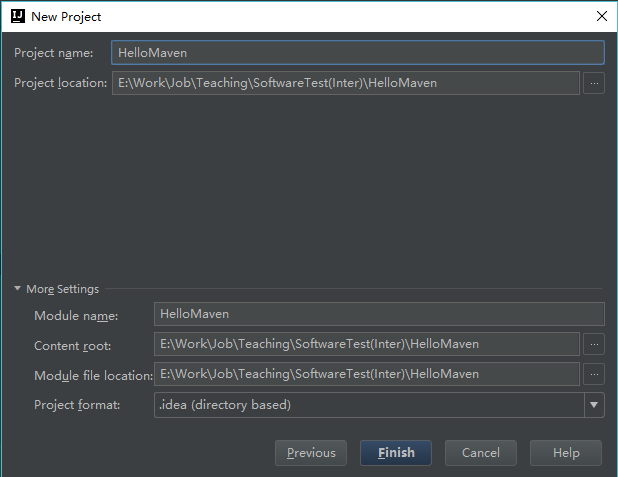
1. Click “File🡪New🡪Project…🡪” and choose “Maven” as follows

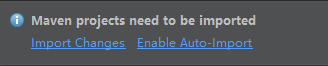


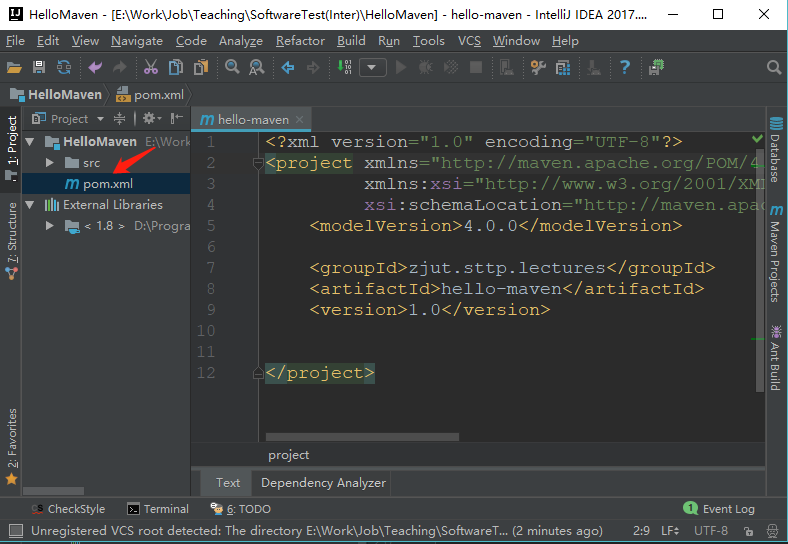
1. Click “Next” button, and then name your GroupId, ArtifactId and Version. Refer to [this](https://maven.apache.org/guides/mini/guide-naming-conventions.html) for more details about naming conventions on GroupId, ArtifactId and Version. Here, we name them as zjut.sttp.lectures, hello-maven, and 1.0, as shown in the following



1. Click “Next” button and create our first Maven project: “HelloMaven”.



1. The newly created project is as follows. Please *first* note that there is a floating window at the bottom right corner, like . You can choose “Enable Auto-Import”. *Second*, we note that there is a “pom.xml”. POM stands for "Project Object Model". It is an XML representation of a Maven project held in a file named pom.xml. Please refer to [this](https://maven.apache.org/pom.html) for more details about POM. In short, pom.xml is mainly used for dependency management.



# Using JUnit 4 in Maven Project

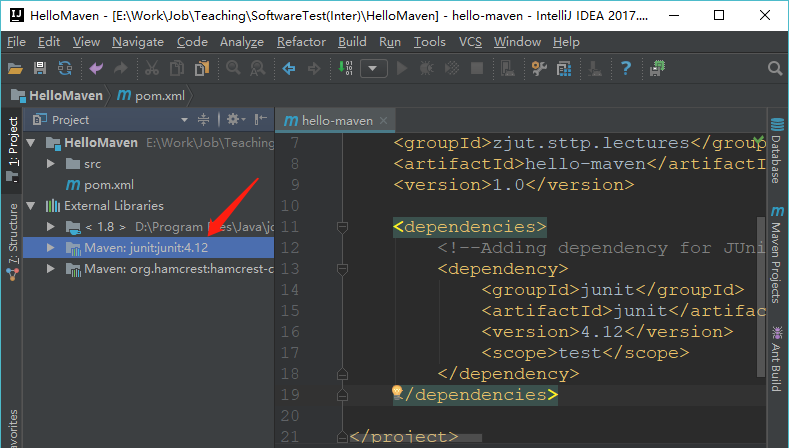
## Configure JUnit 4 in Maven Project

1. Adding dependency for JUnit4 is very simple! We only need to add a single dependency into the pom.xml.

<dependencies>  
 <!--Adding dependency for JUnit4 is very simple!-->  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>4.12</version>  
 <scope>test</scope>  
 </dependency>  
</dependencies>

Please Note: Dependency scope is used to limit the transitivity /trænsəˈtɪvəti/ of a dependency and to determine when a dependency is included in a classpath. Please refer to [here](https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#dependency-scope) for more details about Dependency Scope.

1. Then, you’ll find the “junit 4.12” is added to your classpath. Wow, amazing, right?



## Start Using JUnit 4

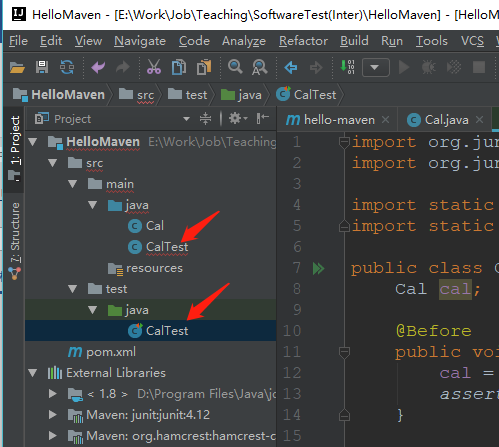
### Create a Java Class for Testing: Cal.java

public class Cal {  
 public int add(int i, int j) {  
 return i + j;  
 }  
  
 public void squareRoot(int n) {  
 for (; ; ) ; // Bug : 死循环  
 }  
  
 public float divide(int i, int j) {  
 return i / j;  
 }  
}

### Create a Testing Class: CalTest.java

import org.junit.Before;  
import org.junit.Test;  
  
import static org.junit.Assert.*assertEquals*;  
import static org.junit.Assert.*assertNotNull*;  
  
public class CalTest {  
 Cal cal;  
  
 @Before  
 public void start() {  
 cal = new Cal();  
 *assertNotNull*(cal);  
 }  
  
 @Test  
 public void testAdd() {  
 int add = cal.add(1, 2);  
 *assertEquals*(3, add);  
  
 *assertEquals*(4, cal.add(2, 2));  
 }  
}

1. Then, put Cal.java and CalTest.java into the right places. By putting CalTest.java into different places, /src/main/java/ and /src/test/java/, you can understand the role of dependency scope. If the dependency scope of JUnit 4.12 is set to “test”, CalTest.java in /src/main/java/ cannot be complied successfully. Please refer to [here](https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#dependency-scope) for more details about Dependency Scope.



## Perform Testing in JUnit 4: Timeout Test & Exception Test

* Annotations: @Test
* Assert Functions: assertEquals, assertNotNull
* Timeout test
* Exception test
* Parameterized Test

***Timeout Testing***. In two scenarios, we need to use Timeout Testing. ***First***, it can be used for ***performance test*** to ensure the method is returned within a reasonable time. ***Second***, it is advisable to use Timeout Testing when we write test cases with external dependencies, which never give a 100% certainty that they will be available while executing the test cases. If the given task inside a test case takes more than specified duration, then test will fail.

**E.g.**: The *squareRoot* method in Cal.java (see Section 2.2.1) is expected to be returned within 1 second. However, due to having an endless loop, *squareRoot* will never finish. Please choose a right assertion to make a Timeout Testing for *squareRoot*.

***Exception Test***. Some test cases are used to test the execution of the supplied Executable throws an exception of the *expectedType* and returns the exception.

**E.g.**: Dividing by zero will lead to an *ArithmeticException*. It means the *divide* method in task02.Calculator has a bug if it does not throw an *ArithmeticException* in this scenario. Please write a test method to check if the *divide* method in Cal.java (see Section 2.2.1) can throw an *ArithmeticException* when the *divisor* parameter is zero.

import org.junit.Before;  
import org.junit.Test;  
  
import static org.junit.Assert.*assertEquals*;  
import static org.junit.Assert.*assertNotNull*;  
  
public class CalTest {  
 Cal cal;  
  
 @Before  
 public void start() {  
 cal = new Cal();  
 *assertNotNull*(cal);  
 }

//This test can't run more than 1 second, else failed  
 @Test(timeout = 1000)  
 public void testTimeout() {  
 cal.squareRoot(3);  
 }

@Test(expected = ArithmeticException.class)  
 public void testException() {  
 cal.divide(3, 0);  
 }  
}

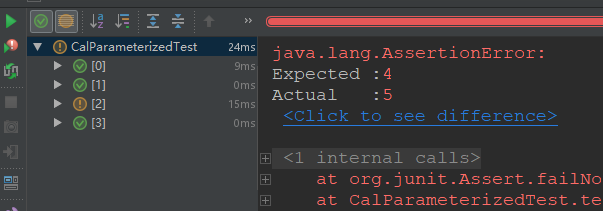
## Parameterized Test in JUnit 4

Parameterized tests make it possible to run a test multiple times with different arguments.

1. Testing code

import org.junit.Test;  
import org.junit.runner.RunWith;  
import org.junit.runners.Parameterized;  
  
import java.util.Arrays;  
import java.util.Collection;  
  
import static org.junit.Assert.*assertEquals*;  
  
@RunWith(Parameterized.class)  
public class CalParameterizedTest {  
 @Parameterized.Parameters  
 public static Collection<Object[]> data() {  
 return Arrays.*asList*(new Object[][]{  
 {0, 0, 0}, {1, 1, 2}, {2, 3, 4}, {3, 5, 8}  
 });  
 }  
  
 private int in1, in2;  
 private int expected;  
  
 public CalParameterizedTest(int in1, int in2, int expected) {  
 this.in1 = in1;  
 this.in2 = in2;  
 this.expected = expected;  
 }  
  
 @Test  
 public void test() {  
 *assertEquals*(expected, new Cal().add(in1, in2));  
 }  
}

1. Running results



# Using JUnit 5 in Maven Project

Please note: JUnit 5 requires Java 8 at runtime.

## What is JUnit 5?

Unlike previous versions of JUnit, JUnit 5’s architecture supports **running multiple test engines** simultaneously: you can run the JUnit Vintage test engine with virtually any other test engine that is compatible with JUnit 5. JUnit 5 is composed of several different modules from three different sub-projects.

**JUnit 5 = *JUnit Platform* + *JUnit Jupiter* + *JUnit Vintage***

* The **JUnit Platform** serves as a foundation for [launching testing frameworks](https://junit.org/junit5/docs/current/user-guide/#launcher-api) on the JVM. It also defines the [TestEngine](https://junit.org/junit5/docs/current/api/org.junit.platform.engine/org/junit/platform/engine/TestEngine.html) API for developing a testing framework that runs on the platform. Furthermore, the platform provides a [Console Launcher](https://junit.org/junit5/docs/current/user-guide/#running-tests-console-launcher) to launch the platform from the command line and a [JUnit 4 based Runner](https://junit.org/junit5/docs/current/user-guide/#running-tests-junit-platform-runner) for running any TestEngine on the platform in a JUnit 4 based environment. First-class support for the JUnit Platform also exists in popular IDEs (see [IntelliJ IDEA](https://junit.org/junit5/docs/current/user-guide/#running-tests-ide-intellij-idea), [Eclipse](https://junit.org/junit5/docs/current/user-guide/#running-tests-ide-eclipse), [NetBeans](https://junit.org/junit5/docs/current/user-guide/#running-tests-ide-netbeans), and [Visual Studio Code](https://junit.org/junit5/docs/current/user-guide/#running-tests-ide-vscode)) and build tools (see [Gradle](https://junit.org/junit5/docs/current/user-guide/#running-tests-build-gradle), [Maven](https://junit.org/junit5/docs/current/user-guide/#running-tests-build-maven), and [Ant](https://junit.org/junit5/docs/current/user-guide/#running-tests-build-ant)).
* **JUnit Jupiter** is the combination of the new [programming model](https://junit.org/junit5/docs/current/user-guide/#writing-tests) and [extension model](https://junit.org/junit5/docs/current/user-guide/#extensions) for writing tests and extensions in JUnit 5. The Jupiter sub-project provides a TestEngine for running Jupiter based tests on the platform.
* **JUnit Vintage** provides a TestEngine for running JUnit 3 and JUnit 4 based tests on the platform. JUnit Vintage itself is comprised of two modules:
* **junit: junit** is the API for JUnit 3 and JUnit 4.
* **junit-vintage-engine** is the test engine for running JUnit 3 and JUnit 4 tests on the JUnit Platform.

IntelliJ IDEA supports running tests on the JUnit Platform since version 2016.2. For details please see the [post on the IntelliJ IDEA blog](https://blog.jetbrains.com/idea/2016/08/using-junit-5-in-intellij-idea/). Note, however, that it is recommended to use IDEA 2017.3 or newer since these newer versions of IDEA will download the following JARs automatically based on the API version used in the project: junit-platform-launcher, junit-jupiter-engine, and junit-vintage-engine.

## Configure JUnit 5 in Maven Project

### Adding Dependency

Please read: [here](https://junit.org/junit5/docs/current/user-guide/#running-tests-ide-intellij-idea) or [this webpage](https://howtodoinjava.com/junit5/junit5-maven-dependency/).

<dependencies>  
 <!-- 1. JUnit 4 dependency can be removed, if using JUnit Vintage. -->  
 <!--<dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>4.13</version>  
 <scope>test</scope>  
 </dependency>-->

<dependency>  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter-engine</artifactId>  
 <version>5.7.1</version>  
 <scope>test</scope>  
 </dependency>

<!-- 1. If adding JUnit 4 dependency, JUnit Vintage must be added;

2. JUnit Vintage supports JUnit 4. That is, all JUnit 4 tests can be run smoothly with JUnit Vintage.

-->  
 <dependency>  
 <groupId>org.junit.vintage</groupId>  
 <artifactId>junit-vintage-engine</artifactId>  
 <version>5.7.1</version>  
 <scope>test</scope>  
 </dependency>  
  
</dependencies>

### Q & A

1. Maven conflict: Using Maven Helper to Resolve Maven dependency conflict, refer to [*here*](https://developpaper.com/using-maven-helper-to-resolve-maven-plug-in-conflict/) and [*here (in Chinese)*](https://blog.csdn.net/wxb141001yxx/article/details/104618917) for more details.
2. If fail to add dependencies, we need to clear the cache by clicking "File🡪Invalidate Caches/Restart..." to reset the environment.

## Perform Testing in JUnit 5: Timeout Test & Exception Test

* Timeout test: JUnit 5 provides us with two ***Timeout Assertions*** (see [here](https://junit.org/junit5/docs/current/user-guide/#writing-tests-declarative-timeouts)), which are used to test long running tasks.
* Exception test:

import lec03.Cal;  
import org.junit.jupiter.api.Test;  
  
import static java.time.Duration.*ofMillis*;  
import static org.junit.jupiter.api.Assertions.\*;  
  
public class CalTest {  
 @Test  
 public void timeoutTesting() {  
 *assertTimeoutPreemptively*(*ofMillis*(1000),  
 () -> new Cal().squareRoot(3));  
 }  
  
 @Test  
 public void exceptionTesting() {  
 Exception e = *assertThrows*(ArithmeticException.class,  
 () -> new Cal().divide(3, 0));  
 *assertEquals*("/ by zero", e.getMessage());  
 }  
}

## Parameterized Test in JUnit 5

Please refer to [this](https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests).

### Adding Dependency

<dependency>  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter-params</artifactId>  
 <version>5.7.1</version>  
 <scope>test</scope>  
</dependency>

### Writing Test Code

import lec03.Cal;  
import org.junit.jupiter.api.Test;  
import org.junit.jupiter.params.ParameterizedTest;  
import org.junit.jupiter.params.provider.CsvSource;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
public class CalTest {

@ParameterizedTest  
 @CsvSource({"0,0,0", "1,1,2", "2, 3, 4", "3, 5, 8"})  
 void parameterizedTest(int in1, int in2, int expected) {  
 *assertEquals*(expected, new Cal().add(in1, in2));  
 }  
}

### Running Results

