Tutorial: Test-driven development

Last modified: 20 August 2024

Whether you like to write your tests before writing production code, or like to create the tests afterwards, IntelliJ IDEA makes it easy to create and run unit tests. In this tutorial we're going to show how to use IntelliJ IDEA to write tests first (Test Driven Development or TDD ↗).

Create a project

Create a new project

1. Launch IntelliJ IDEA.

   If the Welcome screen opens, click **New Project**. Otherwise, go to **File | New | Project** in the main menu.

2. From the list on the left, select **Java**.

3. Name the new project, for example: `MoodAnalyser` and change its location if necessary.

4. Select **Gradle** as a build tool and **Groovy** as a DSL.

5. From the **JDK** list, select the JDK that you want to use in your project.

If the JDK is installed on your computer, but not defined in the IDE, select **Add JDK** and specify the path to the JDK home directory.

If you don't have the necessary JDK on your computer, select **Download JDK**.

6. Click **Create**.

IntelliJ IDEA creates a project with pre-configured structure and essential libraries. JUnit 5 will be added as a dependency to the `build.gradle` file.

Create a new package

1. Right-click the **main | java** folder in the **Project** tool window and select **New | Package**.
2. Name the new package `com.example.demo` and press Enter.

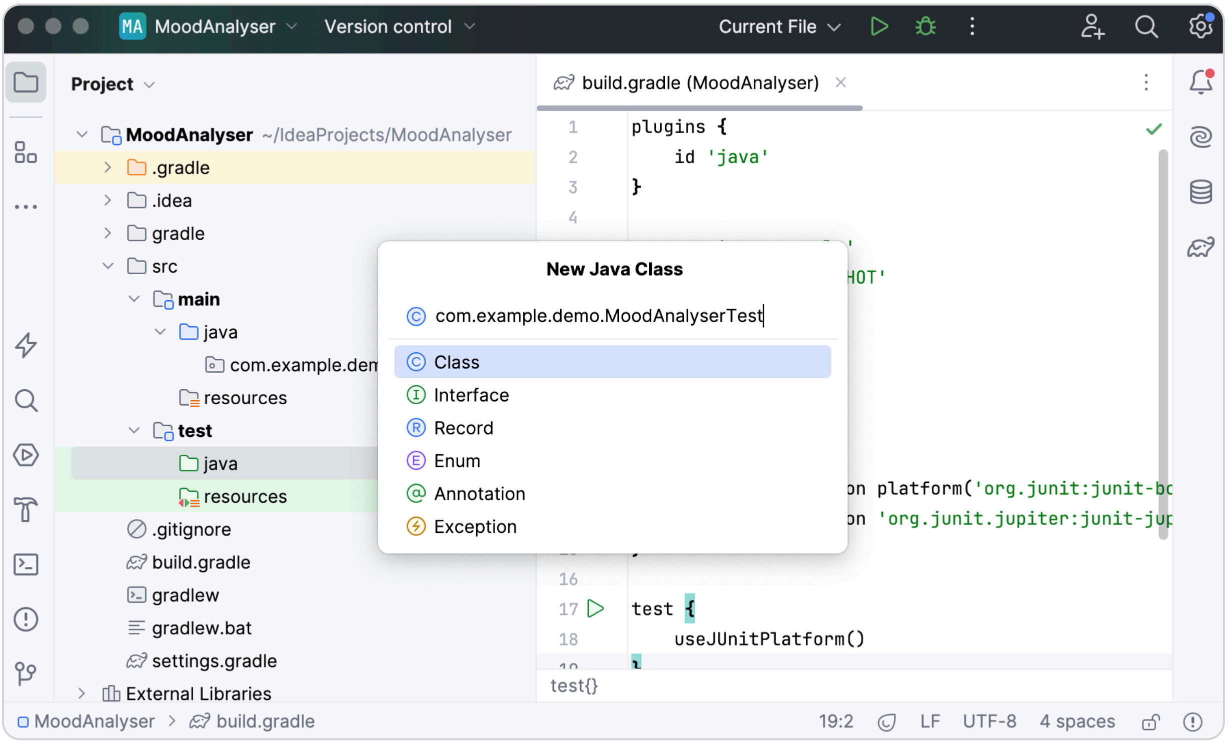
Write the test body

Create your first test

Given that we're writing our tests first without necessarily having the code we're testing available to us yet, we'll create our first test via the project panel and place it in a package.

1. Right-click the test root folder and select **New | Java Class**.

   In the popup that opens, name the new package and test class: `com.example.demo.MoodAnalyserTest`.

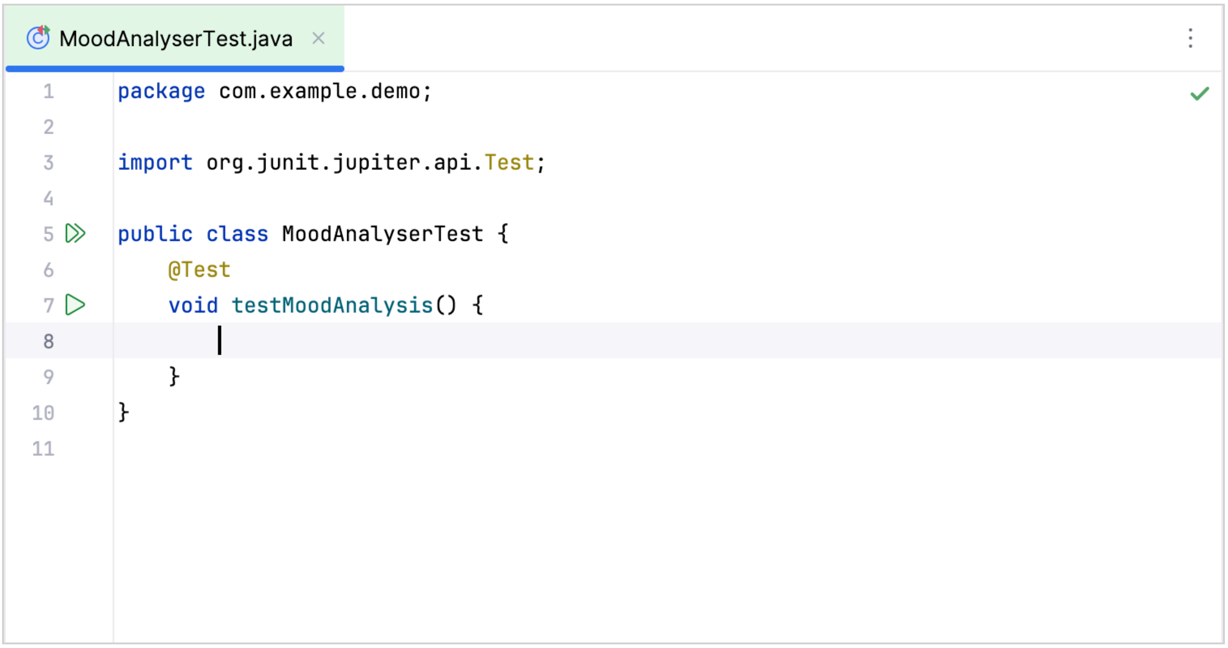


com.example.demo.MoodAnalyserTest

```
plugins {
    id 'java'
}
```

2. Place the caret inside the curly braces in the class, press Alt Insert.
3. Select **Test Method** from the menu to create a test method from the default template.

   Name the method `testMoodAnalysis`, press Enter, and the caret will end up in the method body.



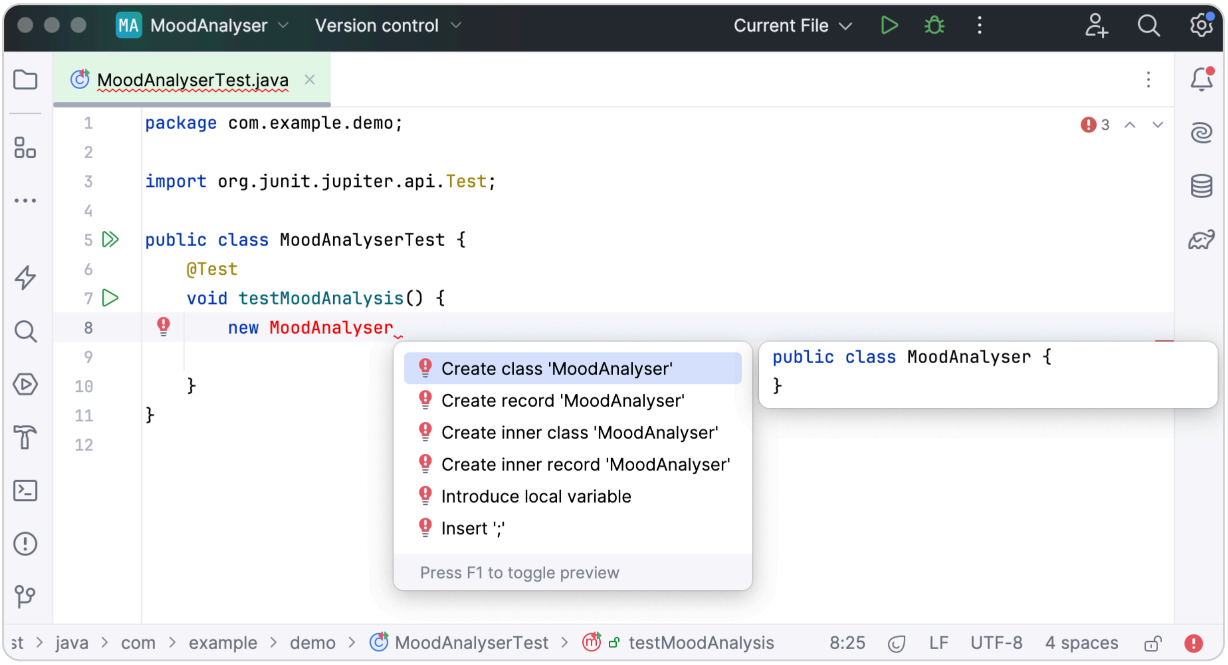
```java
package com.example.demo;

import org.junit.jupiter.api.Test;

public class MoodAnalyserTest {
    @Test
    void testMoodAnalysis() {
        
    }
}
```

   You can alter the default test method template - for example, if you wish to change the start of the method name from `test` to `should`.

Create a new class from the test

It may seem counter-intuitive to write test code for classes and methods that don't exist, but IntelliJ IDEA makes this straightforward while keeping the compiler happy. IntelliJ IDEA can create classes and methods for you if they don't already exist.

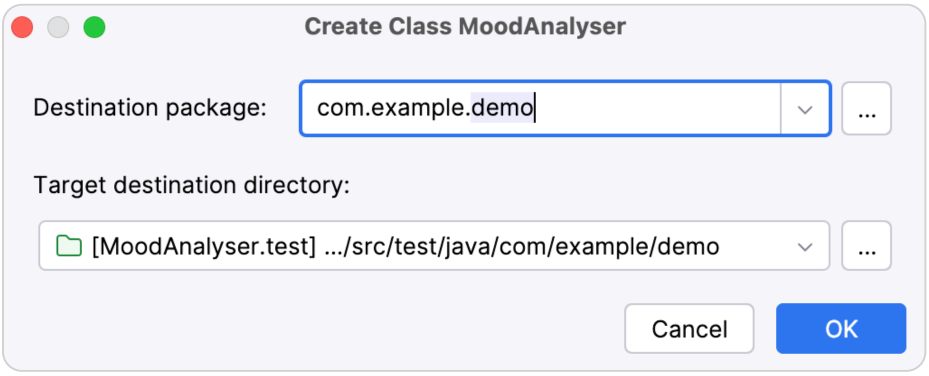
1. Type `new MoodAnalyser`, press Alt Enter, and select **Create class 'MoodAnalyser'**.



```java
package com.example.demo;

import org.junit.jupiter.api.Test;

public class MoodAnalyserTest {
    @Test
    void testMoodAnalysis() {
        new MoodAnalyser
    }
}
```

MoodAnalyser
Version control
Current File
MoodAnalyserTest.java
Create class 'MoodAnalyser'
Create record 'MoodAnalyser'
Create inner class 'MoodAnalyser'
Create inner record 'MoodAnalyser'
Introduce local variable
Insert ';'
Press F1 to toggle preview
public class MoodAnalyser {
}
st > java > com > example > demo > MoodAnalyserTest > testMoodAnalysis
8:25 LF UTF-8 4 spaces

2. In the dialog that opens, select the `com.example.demo` package in the **`main | java`** folder and click **OK**.



Create Class MoodAnalyser

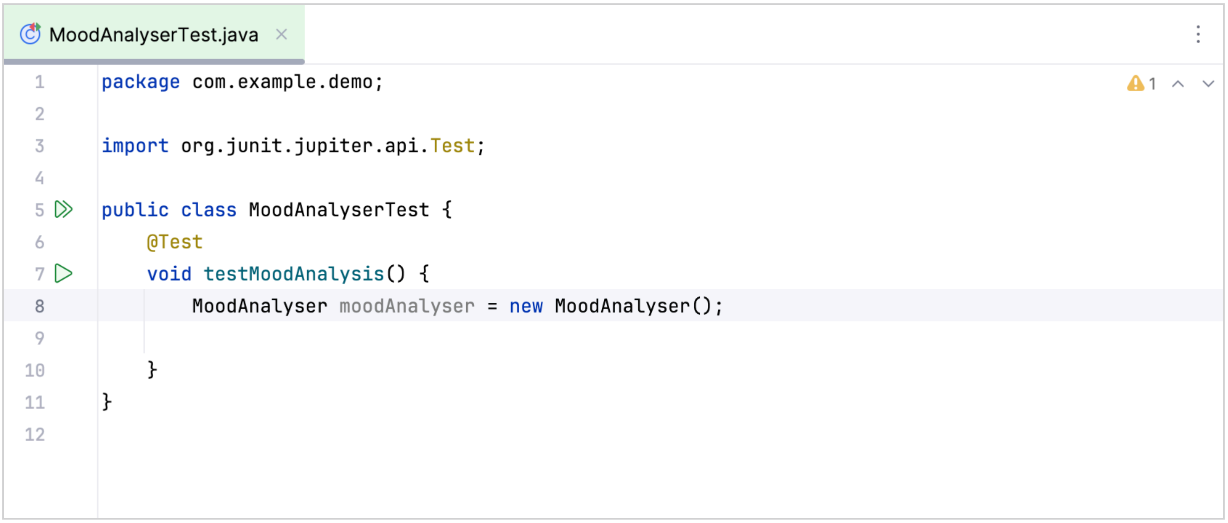
Destination package: com.example.demo

Target destination directory: [MoodAnalyser.test] .../src/test/java/com/example/demo

Create variables

As always, you can use IntelliJ IDEA's refactoring tools to create variables to store results in, and IntelliJ IDEA will import the most appropriate classes for you if the correct libraries are on the classpath.

1. Switch back to the test class, place your cursor after `new MoodAnalyser`, type `()` and press Ctrl Alt V to invoke the Extract/Introduce Variable refactoring.
2. Name the new variable `moodAnalyser`.



```
package com.example.demo;

import org.junit.jupiter.api.Test;

public class MoodAnalyserTest {
    @Test
    void testMoodAnalysis() {
        MoodAnalyser moodAnalyser = new MoodAnalyser();

    }
}
```

Place the caret at the test class or at the test subject in the source code and press Ctrl Shift T to quickly navigate between them. Alternatively, use the Split Screen mode to see both files at the same time.

Complete the test body

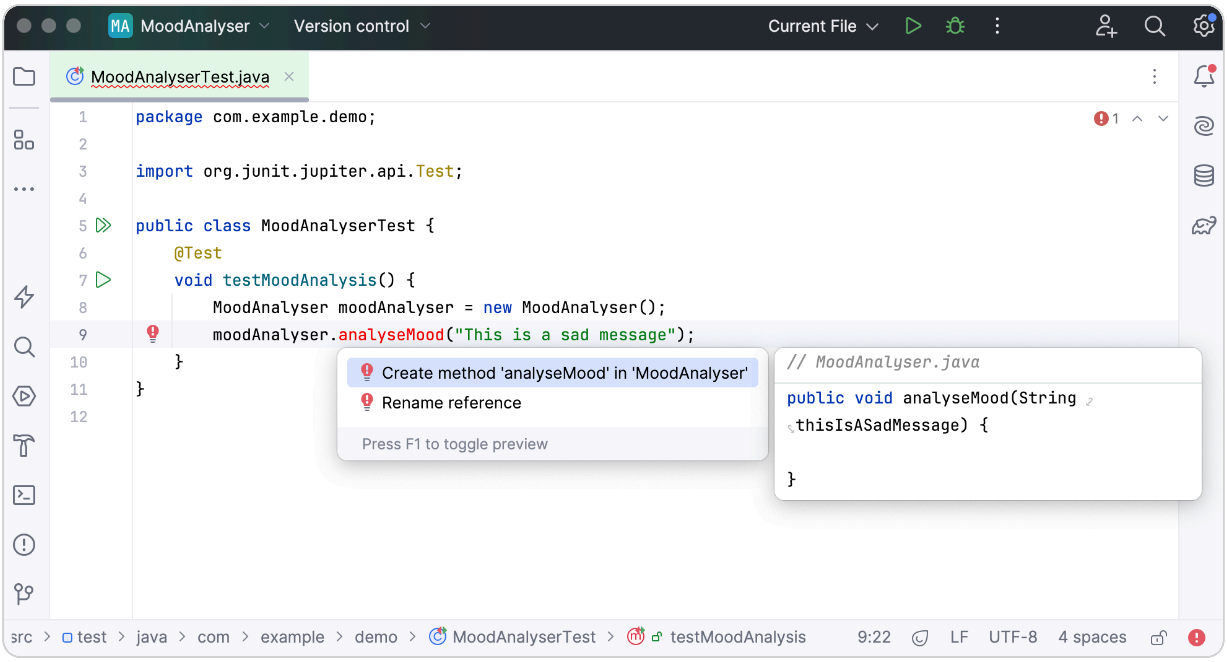
Continue writing the test body, including names of methods that you need that don't exist.

1. In the test class, type the following statement:

```
moodAnalyser.analyseMood("This is a sad message");
```

`analyseMood` will be marked as an unresolved reference.

2. Place the caret at `analyseMood`, press Alt Enter, and click **Create method 'analyseMood' in 'MoodAnalyser'**.



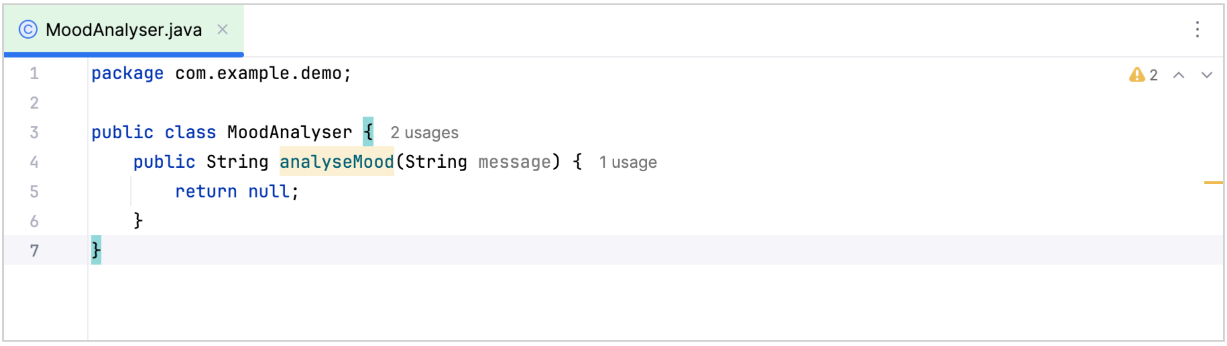
```
package com.example.demo;

import org.junit.jupiter.api.Test;

public class MoodAnalyserTest {
    @Test
    void testMoodAnalysis() {
        MoodAnalyser moodAnalyser = new MoodAnalyser();
        moodAnalyser.analyseMood("This is a sad message");
    }
}
```

MoodAnalyser
Version control
Current File
MoodAnalyserTest.java
Create method 'analyseMood' in 'MoodAnalyser'
Rename reference
Press F1 to toggle preview
src > test > java > com > example > demo > MoodAnalyserTest > testMoodAnalysis
9:22 LF UTF-8 4 spaces

```
// MoodAnalyser.java
public void analyseMood(String thisIsASadMessage) {

}
```

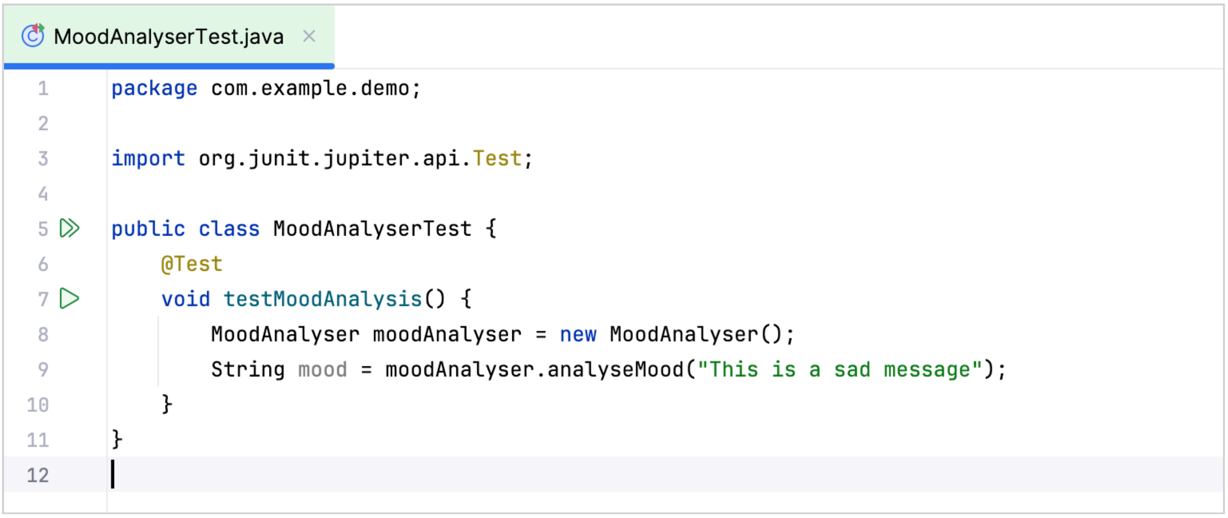
3. Make sure the `MoodAnalyser` class looks as follows:

```java
public class MoodAnalyser {
    public String analyseMood(String message) {
        return null;
    }
}
```



```
package com.example.demo;

public class MoodAnalyser {  2 usages
    public String analyseMood(String message) {  1 usage
        return null;
    }
}
```

4. In the test class, place the caret at `analyseMood`, press Ctrl Alt V, and type `mood`.


MoodAnalyserTest.java

```java
package com.example.demo;

import org.junit.jupiter.api.Test;

public class MoodAnalyserTest {
    @Test
    void testMoodAnalysis() {
        MoodAnalyser moodAnalyser = new MoodAnalyser();
        String mood = moodAnalyser.analyseMood("This is a sad message");
    }
}

```

Add an assertion statement

1. Open the **`build.gradle`** file, add the following dependency and click to import the changes:

```
dependencies {
    testImplementation(
            'org.hamcrest:hamcrest-library:2.2'
    )
}
```

2. In **`MoodAnalyserTest`**, add the following statement:

```java
assertThat(mood, CoreMatchers.is("SAD"));
```

Import the missing methods and classes by pressing Alt Enter.

Code

Run the tests

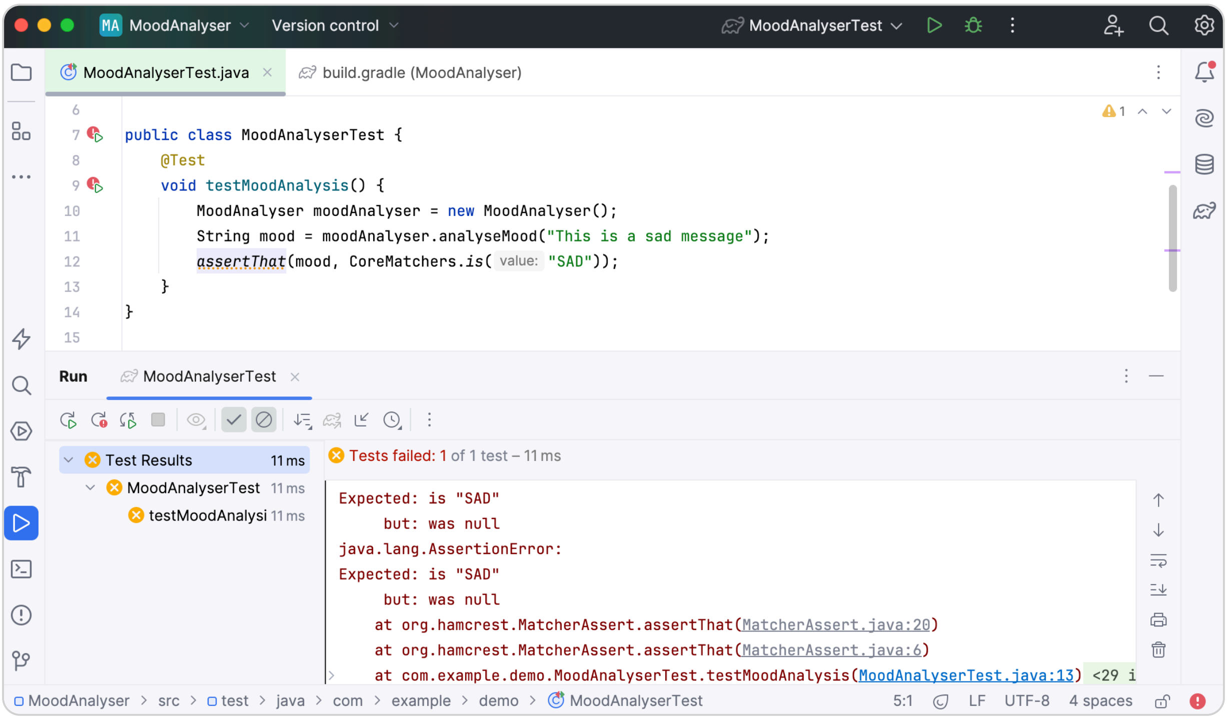
When following a TDD approach, typically you go through a cycle of Red-Green-Refactor ↗. You'll run a test, see it fail (go red), implement the simplest code to make the test pass (go green), and then refactor the code so your test stays green and your code is sufficiently clean.

The first step in this cycle is to run the test and see it fail.

Given that we've used IntelliJ IDEA features to create the simplest empty implementation of the method we're testing, we do not expect our test to pass.

- From inside the test, press Ctrl Shift F10 to run this individual test.

  The results will be shown in the Run tool window. The test name will have an icon next to it — either red for an exception or yellow for an assertion that fails. For either type of failure, a message stating what went wrong is also shown.



```java
public class MoodAnalyserTest {
    @Test
    void testMoodAnalysis() {
        MoodAnalyser moodAnalyser = new MoodAnalyser();
        String mood = moodAnalyser.analyseMood("This is a sad message");
        assertThat(mood, CoreMatchers.is( value: "SAD"));
    }
}
```

```
Test Results          11 ms
  MoodAnalyserTest    11 ms
    testMoodAnalysi   11 ms

Tests failed: 1 of 1 test – 11 ms

Expected: is "SAD"
     but: was null
java.lang.AssertionError:
Expected: is "SAD"
     but: was null
    at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:20)
    at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:6)
    at com.example.demo.MoodAnalyserTest.testMoodAnalysis(MoodAnalyserTest.java:13) <29 i
```

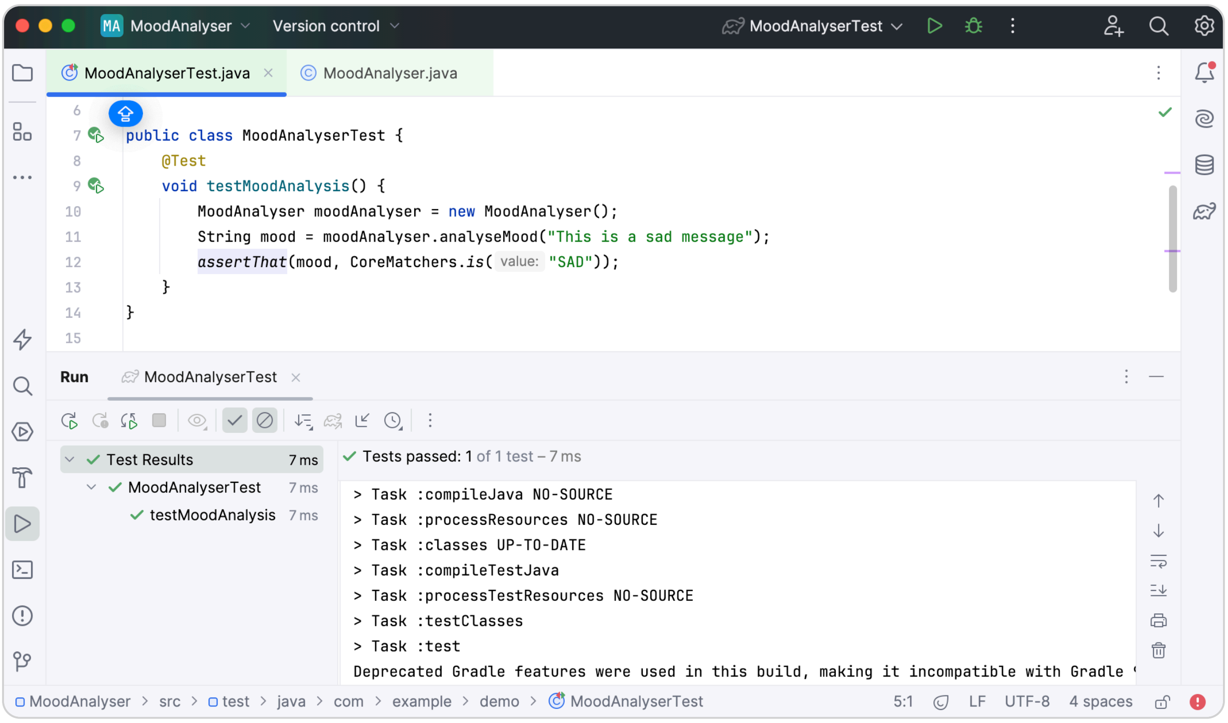
Implement the code

The next step is to make the tests pass, which means implementing the simplest thing that works. Often with TDD, the simplest thing that works might be hard-coding your expected value. We will see later how iterating over this process will lead to more realistic production code.

Fix the test

1. In **`MoodAnalyser`**, replace `null` with the `SAD` return value: `return "SAD";`.
2. Re-run the test, using `Shift` `F10` to re-run the last test.

   See the test pass - the icon next to the test method should go green.



```
public class MoodAnalyserTest {
    @Test
    void testMoodAnalysis() {
        MoodAnalyser moodAnalyser = new MoodAnalyser();
        String mood = moodAnalyser.analyseMood("This is a sad message");
        assertThat(mood, CoreMatchers.is( value: "SAD"));
    }
}
```

```
✓ Test Results 7 ms
  ✓ MoodAnalyserTest 7 ms
    ✓ testMoodAnalysis 7 ms
```

```
✓ Tests passed: 1 of 1 test – 7 ms

> Task :compileJava NO-SOURCE
> Task :processResources NO-SOURCE
> Task :classes UP-TO-DATE
> Task :compileTestJava
> Task :processTestResources NO-SOURCE
> Task :testClasses
> Task :test
Deprecated Gradle features were used in this build, making it incompatible with Gradle
```

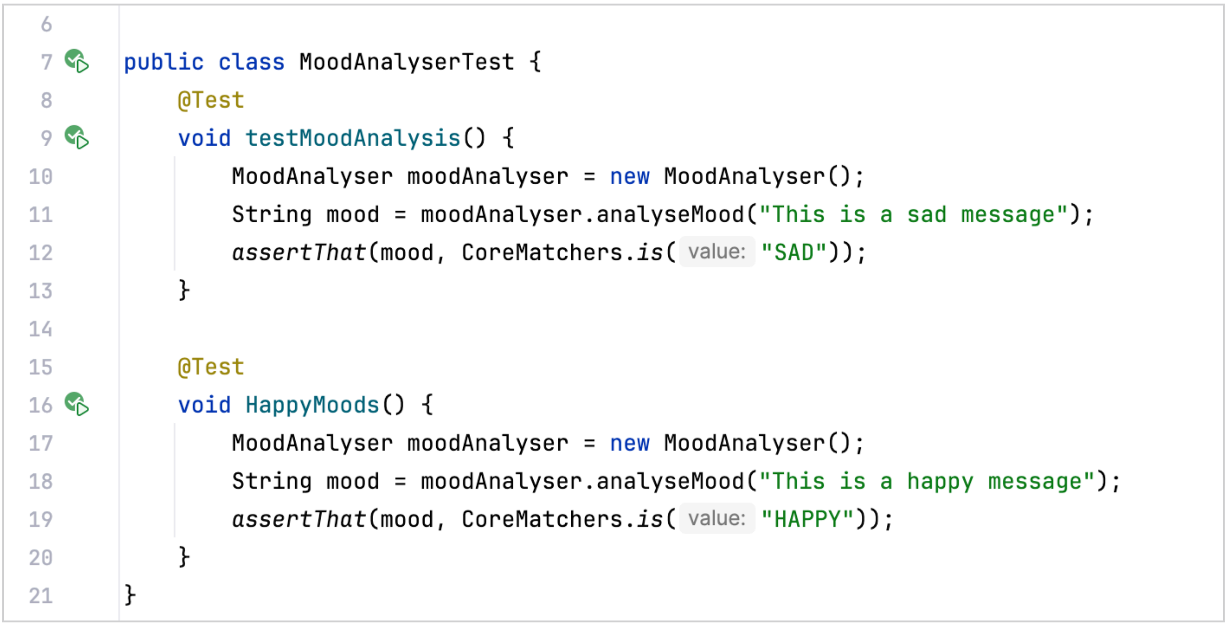
Iterate

Developing code is an iterative process. When following a TDD-style approach, this is even more true. In order to drive out more complex behaviour, we add tests for other cases.

Add the second test case

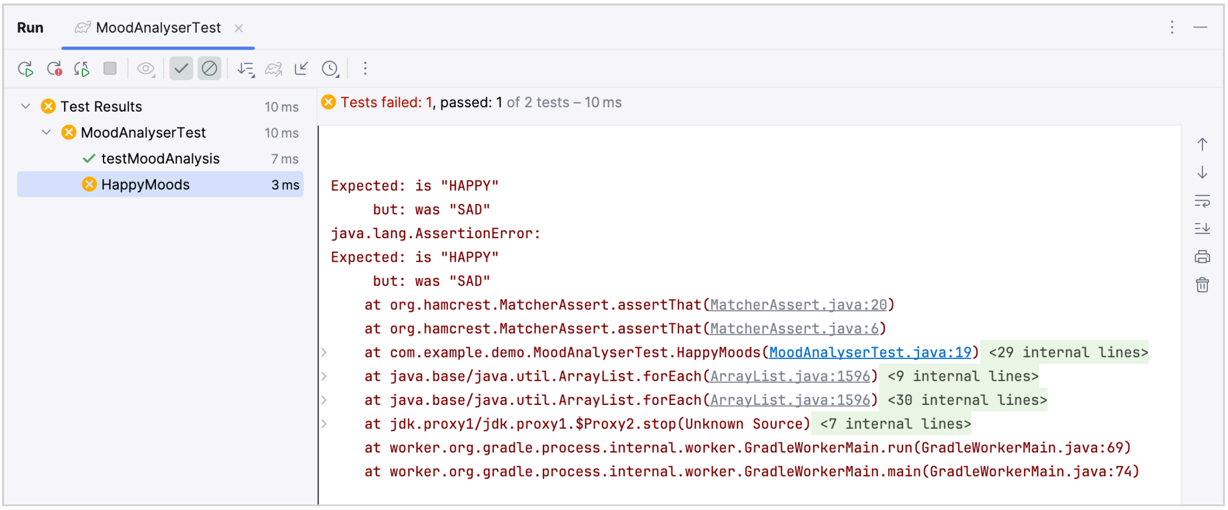
1. In your test class, use Alt Insert again to create a new test method. Name it `HappyMoods`.
2. Add the following code to your class.

```
@Test
void HappyMoods() {
    MoodAnalyser moodAnalyser = new MoodAnalyser();
    String mood = moodAnalyser.analyseMood("This is a happy message")
    assertThat(mood, CoreMatchers.is("HAPPY"));
}
```



```
public class MoodAnalyserTest {
    @Test
    void testMoodAnalysis() {
        MoodAnalyser moodAnalyser = new MoodAnalyser();
        String mood = moodAnalyser.analyseMood("This is a sad message");
        assertThat(mood, CoreMatchers.is( value: "SAD"));
    }

    @Test
    void HappyMoods() {
        MoodAnalyser moodAnalyser = new MoodAnalyser();
        String mood = moodAnalyser.analyseMood("This is a happy message");
        assertThat(mood, CoreMatchers.is( value: "HAPPY"));
    }
}
```

3. Run this second test case by pressing Alt Shift R, you will see that it fails.



Tests failed: 1, passed: 1 of 2 tests – 10 ms

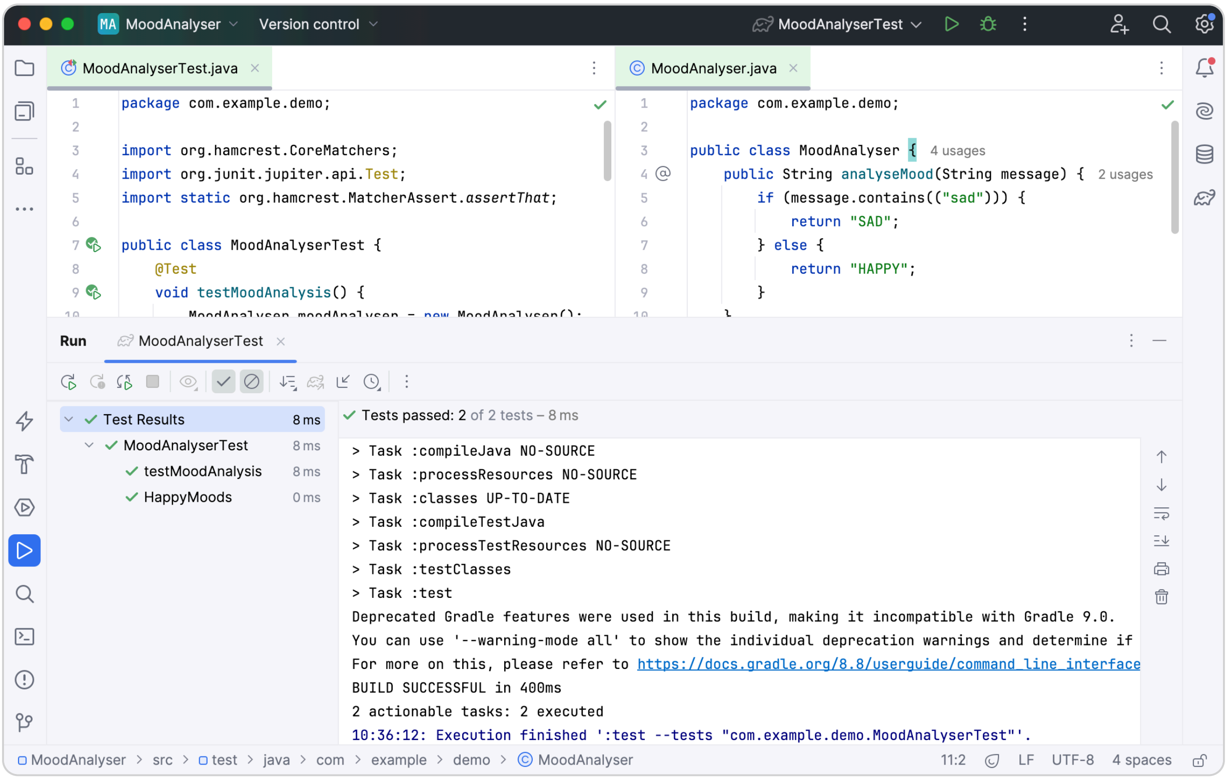
```
Expected: is "HAPPY"
     but: was "SAD"
java.lang.AssertionError:
Expected: is "HAPPY"
     but: was "SAD"
    at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:20)
    at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:6)
    at com.example.demo.MoodAnalyserTest.HappyMoods(MoodAnalyserTest.java:19) <29 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1596) <9 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1596) <30 internal lines>
    at jdk.proxy1/jdk.proxy1.$Proxy2.stop(Unknown Source) <7 internal lines>
    at worker.org.gradle.process.internal.worker.GradleWorkerMain.run(GradleWorkerMain.java:69)
    at worker.org.gradle.process.internal.worker.GradleWorkerMain.main(GradleWorkerMain.java:74)
```

Fix the second test

1. Change the code in the method being tested to make this test pass:

```java
package com.example.demo;

public class MoodAnalyser {
    public String analyseMood(String message) {
        if (message.contains(("sad"))) {
            return "SAD";
        } else {
            return "HAPPY";
        }
    }
}
```

2. Re-run both the tests by pressing Ctrl Shift F10 inside the test class, not inside a single method, and see that both tests now pass.



```
package com.example.demo;

import org.hamcrest.CoreMatchers;
import org.junit.jupiter.api.Test;
import static org.hamcrest.MatcherAssert.assertThat;

public class MoodAnalyserTest {
    @Test
    void testMoodAnalysis() {
        MoodAnalyser moodAnalyser = new MoodAnalyser();
```

```
package com.example.demo;

public class MoodAnalyser {  4 usages
    public String analyseMood(String message) {  2 usages
        if (message.contains(("sad"))) {
            return "SAD";
        } else {
            return "HAPPY";
        }
    }
```

```
Test Results            8 ms
  MoodAnalyserTest      8 ms
    testMoodAnalysis    8 ms
    HappyMoods          0 ms
```

Tests passed: 2 of 2 tests – 8 ms

```
> Task :compileJava NO-SOURCE
> Task :processResources NO-SOURCE
> Task :classes UP-TO-DATE
> Task :compileTestJava
> Task :processTestResources NO-SOURCE
> Task :testClasses
> Task :test
Deprecated Gradle features were used in this build, making it incompatible with Gradle 9.0.
You can use '--warning-mode all' to show the individual deprecation warnings and determine if
For more on this, please refer to https://docs.gradle.org/8.8/userguide/command_line_interface
BUILD SUCCESSFUL in 400ms
2 actionable tasks: 2 executed
10:36:12: Execution finished ':test --tests "com.example.demo.MoodAnalyserTest"'.
```

MoodAnalyser > src > test > java > com > example > demo > MoodAnalyser   11:2 LF UTF-8 4 spaces

Summary

Writing your first test in a test-first style takes a small amount of setup – creating the test class, creating the test methods, and then creating empty implementations of the code that will eventually become production code. IntelliJ IDEA automates a lot of this initial setup.

As you iterate through the process, creating tests and then making the changes required to get those tests to pass, you build up a comprehensive suite of tests for your required functionality, and the simplest solution that will meet these requirements.