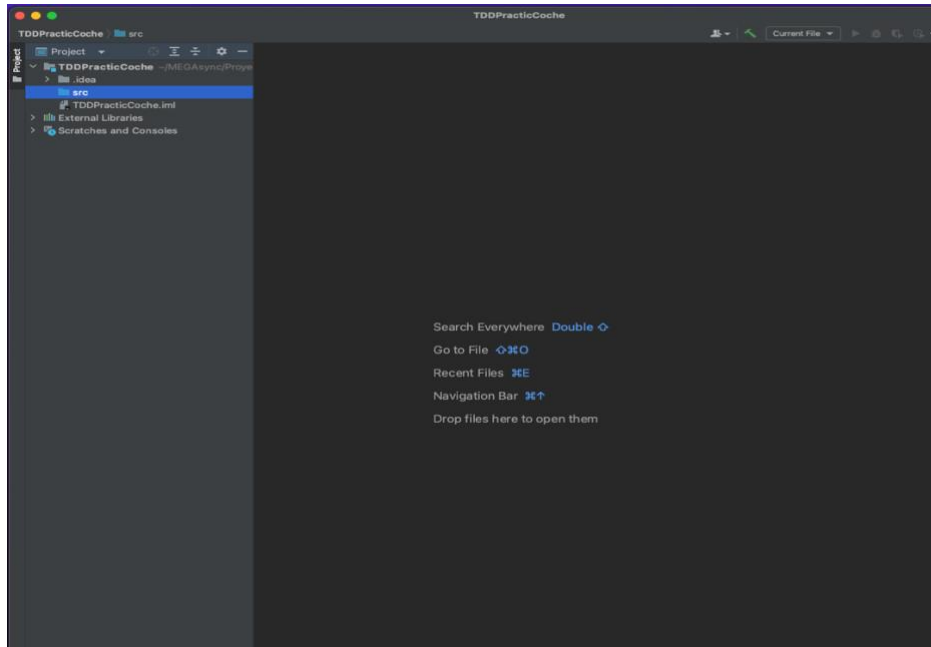
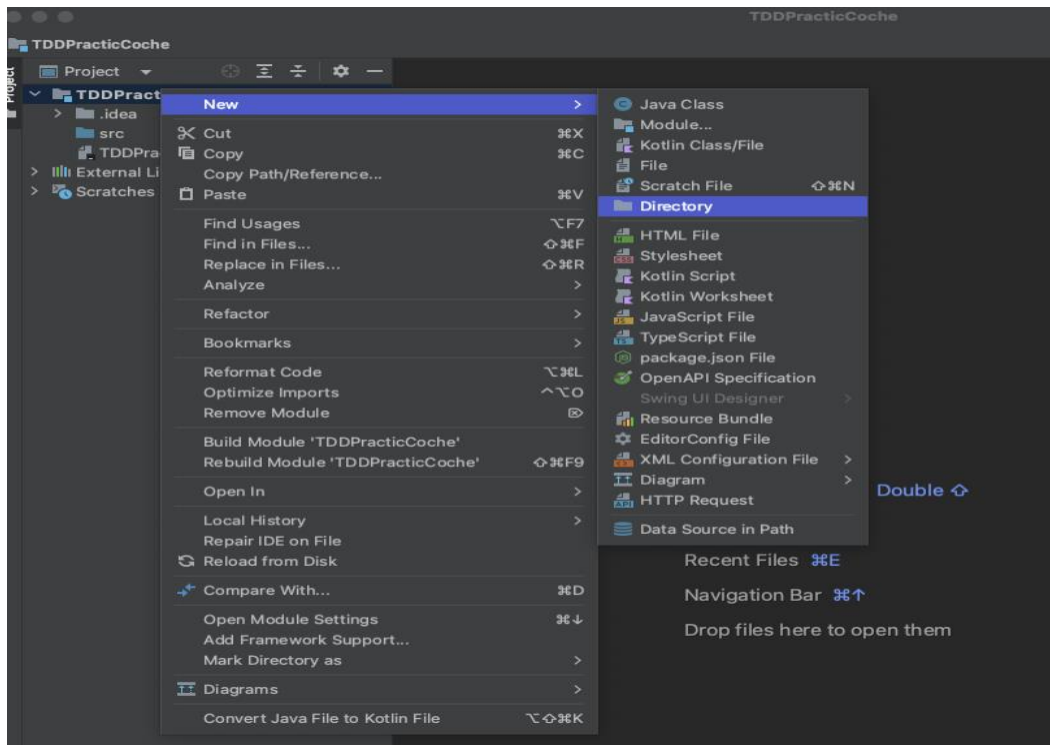


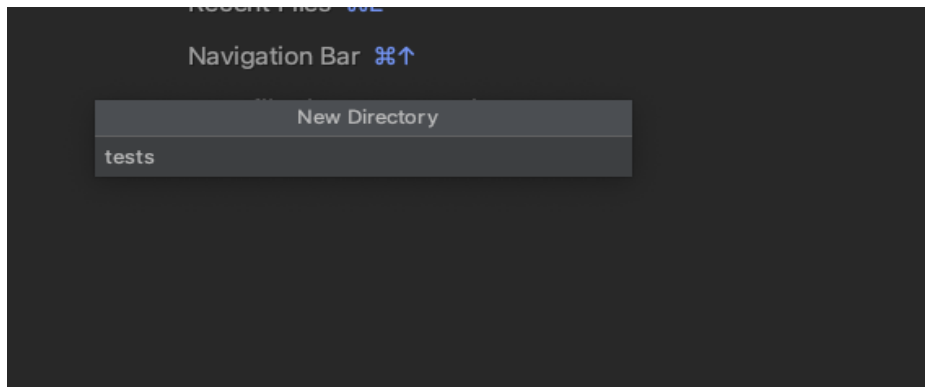
Mi primer TDD

En primer lugar, nos dirigimos a IntelliJ y creamos un nuevo proyecto llamado TDDPracticaCoche.

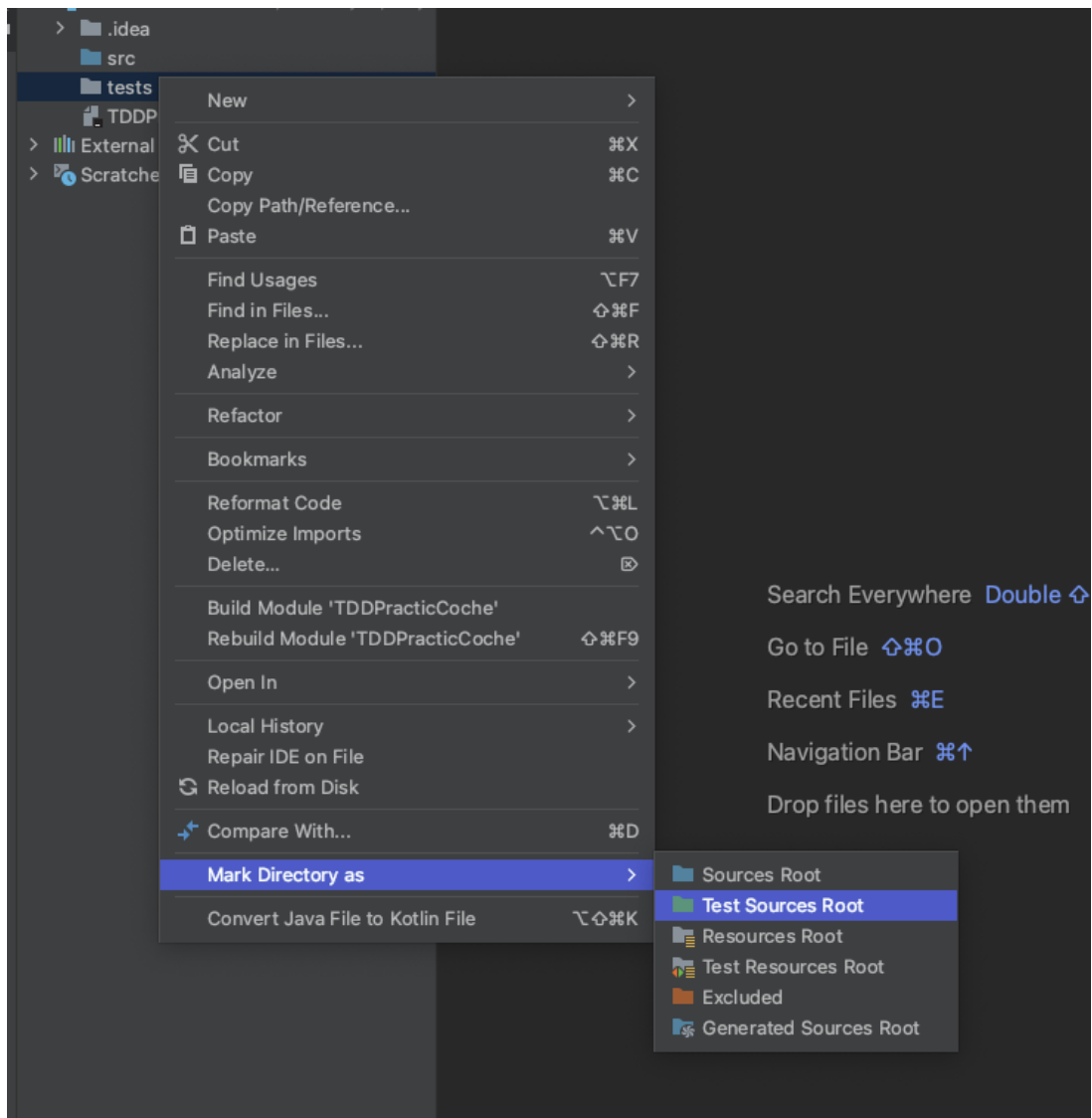


Ahora, sobre el nombre del proyecto hacemos click derecho, "new" y seleccionamos un nuevo directorio y le damos como nombre "tests", donde realizaremos nuestras pruebas.





Creado el directorio “tests” hacemos click derecho sobre el y nos dirigimos hasta la opción “Mark Directory as” para marcar que se trata de un directorio de test.



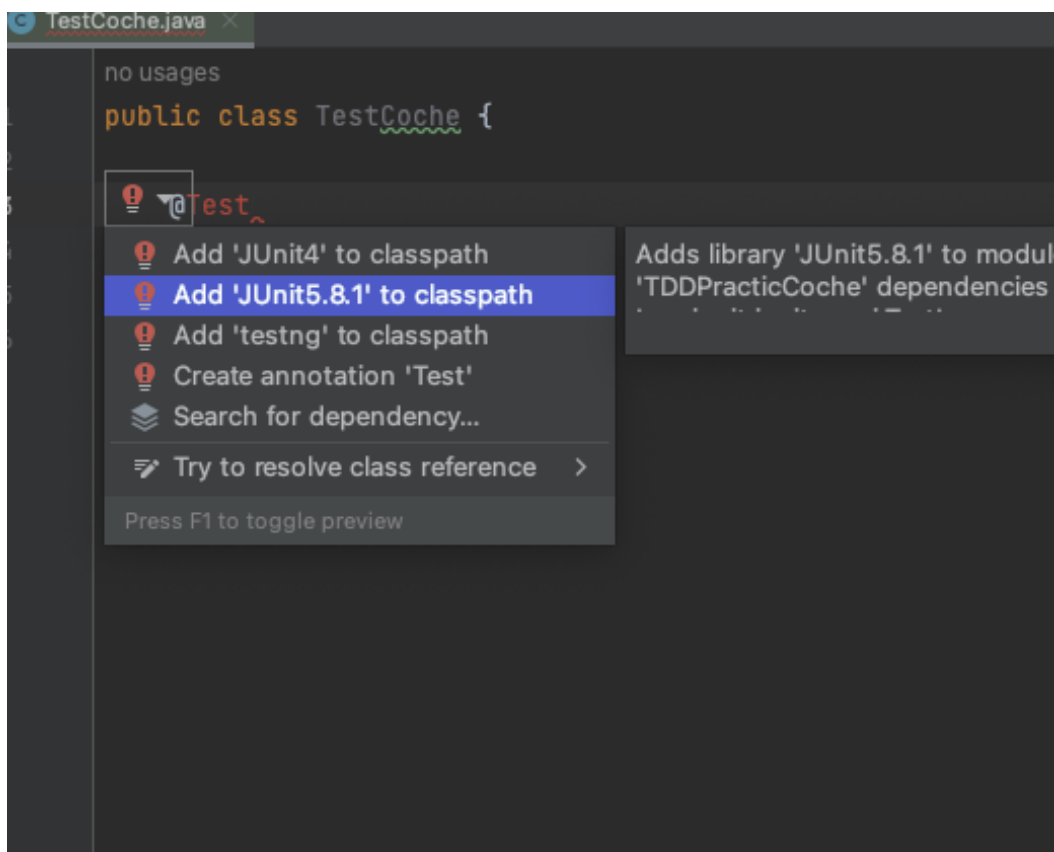
Hecho esto, volvemos a hacer click derecho sobre el directorio, “new” y seleccionamos “Java-class” indicando “TestCoche” como el nombre de la clase.

Creada la clase TestCoche indicamos que se trata de un test con @Test.



```
TestCoche.java x
no usages
1 public class TestCoche {
2
3     @Test_
4
5 }
6
```

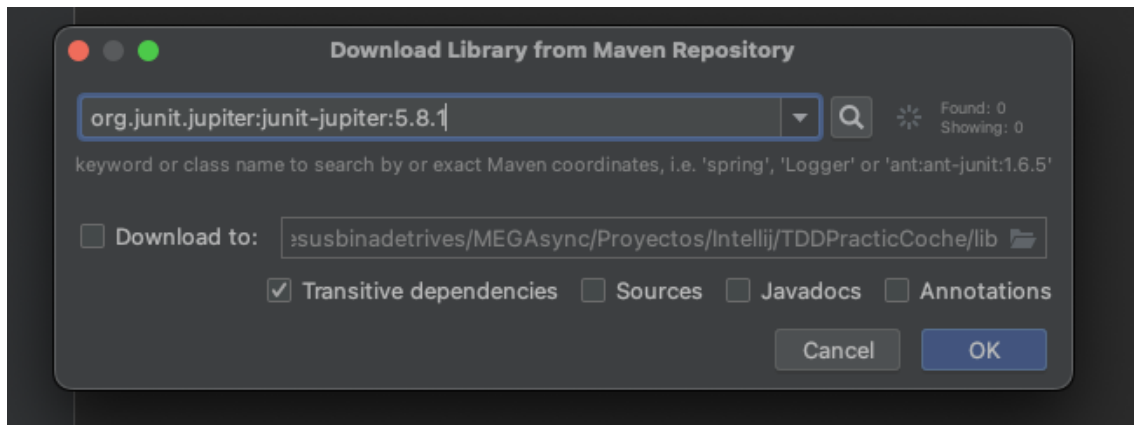
Como podemos ver se muestra en rojo y da error. Para solucionarlo debemos posicionarnos encima y nos aparecerá una bombilla de color rojo. Estando sobre ella, indicamos “add Junit 5”.



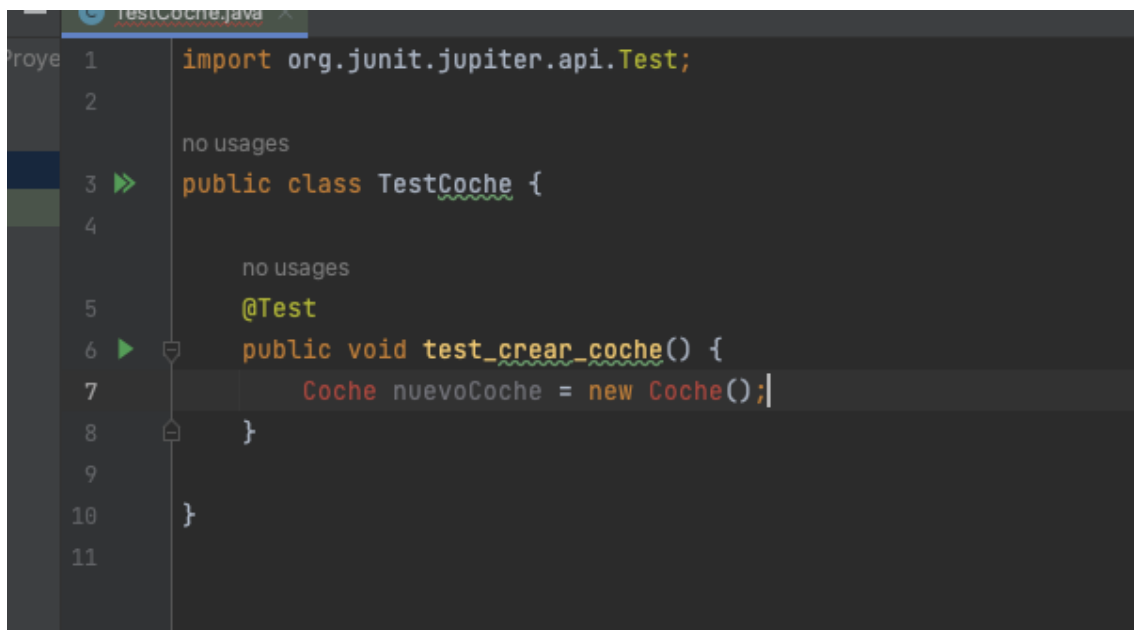
```
TestCoche.java x
no usages
public class TestCoche {
    @Test_
}

Add 'JUnit4' to classpath
Add 'JUnit5.8.1' to classpath
Add 'testng' to classpath
Create annotation 'Test'
Search for dependency...
Try to resolve class reference >
Press F1 to toggle preview

Adds library 'JUnit5.8.1' to module 'TDDPracticCoche' dependencies
```



Realizado esto, porcedemos a crear nuestro primer método de test. En el, creamos un objeto de la clase coche.



Como se observa, la clase Coche aparece en rojo y es debido a que todavía no existe y debemos crearla. Para ello, nos posicionamos sobre el nombre de la clase y se nos mostrará una opción para crear la clase.

```
1 import org.junit.jupiter.api.Test;
2
3 no usages
4 public class TestCoche {
5
6     no usages
7     @Test
8     public void test_crear_coche() {
9         Coche nuevoCoche = new Coche();
10    }
11 }
```

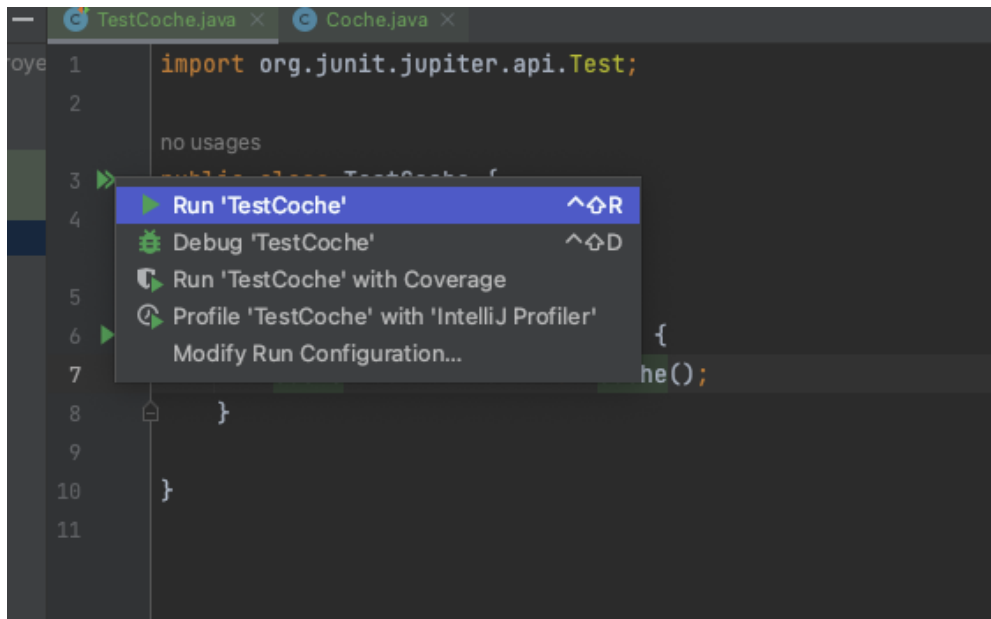
Cannot resolve symbol 'Coche'

Create class 'Coche' More actions...

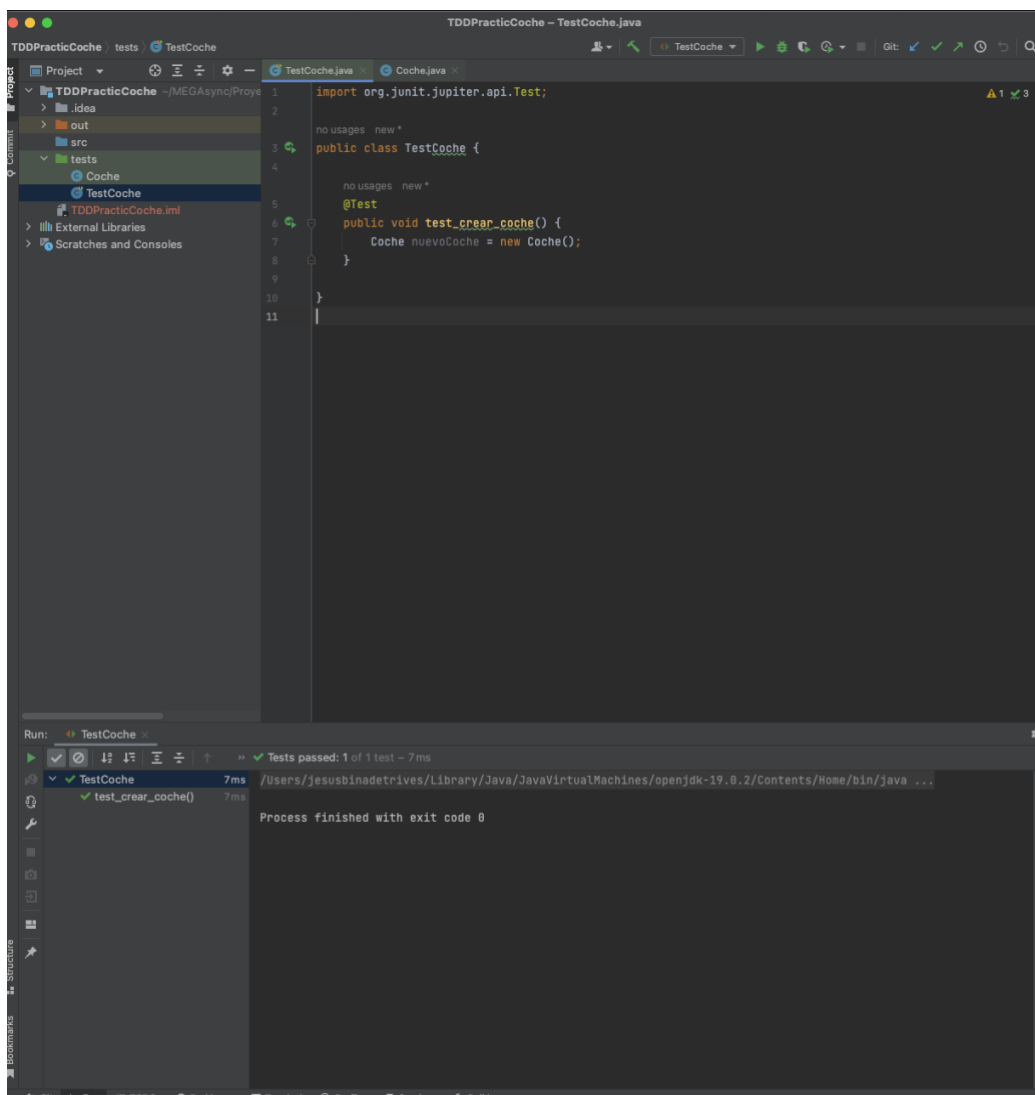
TestCoche.java x Coche.java x

```
1 2 usages new *
2 public class Coche {
3     |
4 }
```

Creada la clase ejecutamos el test.



Ejecutamos el test y vemos como se completa satisfactoriamente.



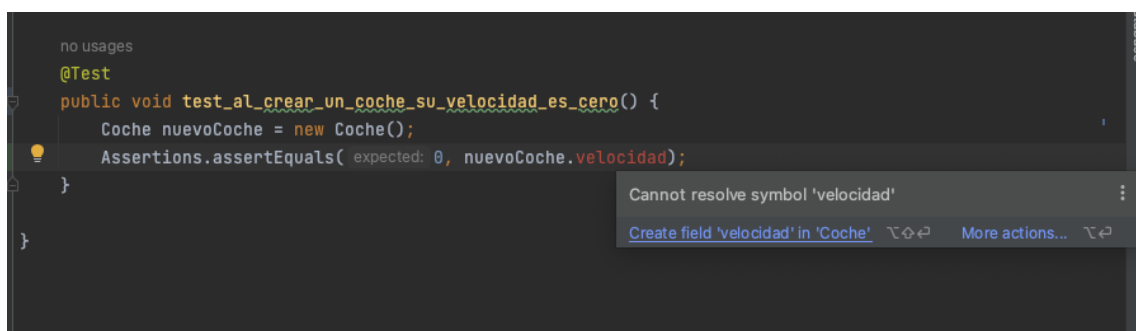
Ahora vamos a mejorar un poco el test haciendo que no sea tan simple utilizando una aserción.



```
1 import org.junit.jupiter.api.Assertions;
2 import org.junit.jupiter.api.Test;
3
4 public class TestCoche {
5
6     @Test
7     public void test_al_crear_un_coche_su_velocidad_es_cero() {
8         Coche nuevoCoche = new Coche();
9         Assertions.assertEquals(0, nuevoCoche.velocidad);
10    }
11
12 }
13
```

Hemos modificado el método estableciendo que al crear un coche su velocidad sea cero y hemos utilizado la aserción de manera que el valor que esperamos sea cero y el valor real sea el que se le pasa. Además, como vemos, velocidad aparece en rojo y es debido a que no hemos creado dicho atributo en la clase Coche.

De igual manera que anteriormente con la clase Coche, creamos el atributo velocidad.



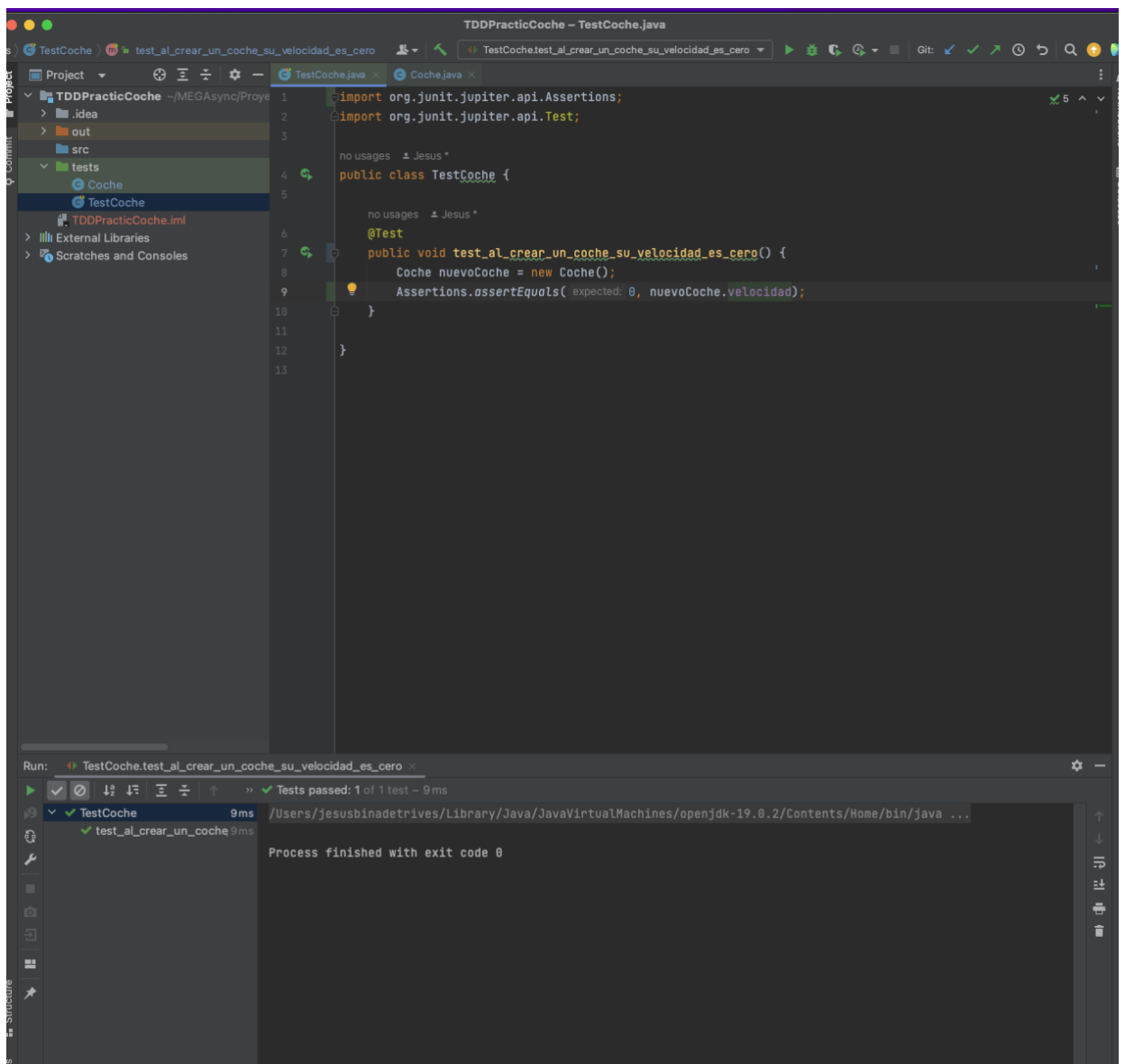
```
no usages
@Test
public void test_al_crear_un_coche_su_velocidad_es_cero() {
    Coche nuevoCoche = new Coche();
    Assertions.assertEquals( expected: 0, nuevoCoche.velocidad);
}
}
```

Cannot resolve symbol 'velocidad'

Create field 'velocidad' in 'Coche' More actions...



Hecho esto, ejecutamos el test y observamos como lo hemos pasado correctamente.



Una vez hecho esto, vamos a implementar un método para acelerar la velocidad del coche.

```
10
11
12     no usages new *
13     @Test
14     public void test_al_acelerar_un_coche_su_velocidad_aumenta() {
15         Coche nuevoCoche = new Coche();
16         nuevoCoche.acelerar(30);
17         Assertions.assertEquals( expected: 30, nuevoCoche.velocidad);
18     }
19
20 }
```

Hemos creado el método para acelerar la velocidad del coche, pero como vemos acelerar aparece en rojo lo que significa que no existe y debemos crear el método en la clase Coche, además, le hemos dicho que su velocidad es de 30.

```
@Test
public void test_al_acelerar_un_coche_su_velocidad_aumenta() {
    Coche nuevoCoche = new Coche();
    nuevoCoche.acelerar(30);
    Assertions.as
}
```

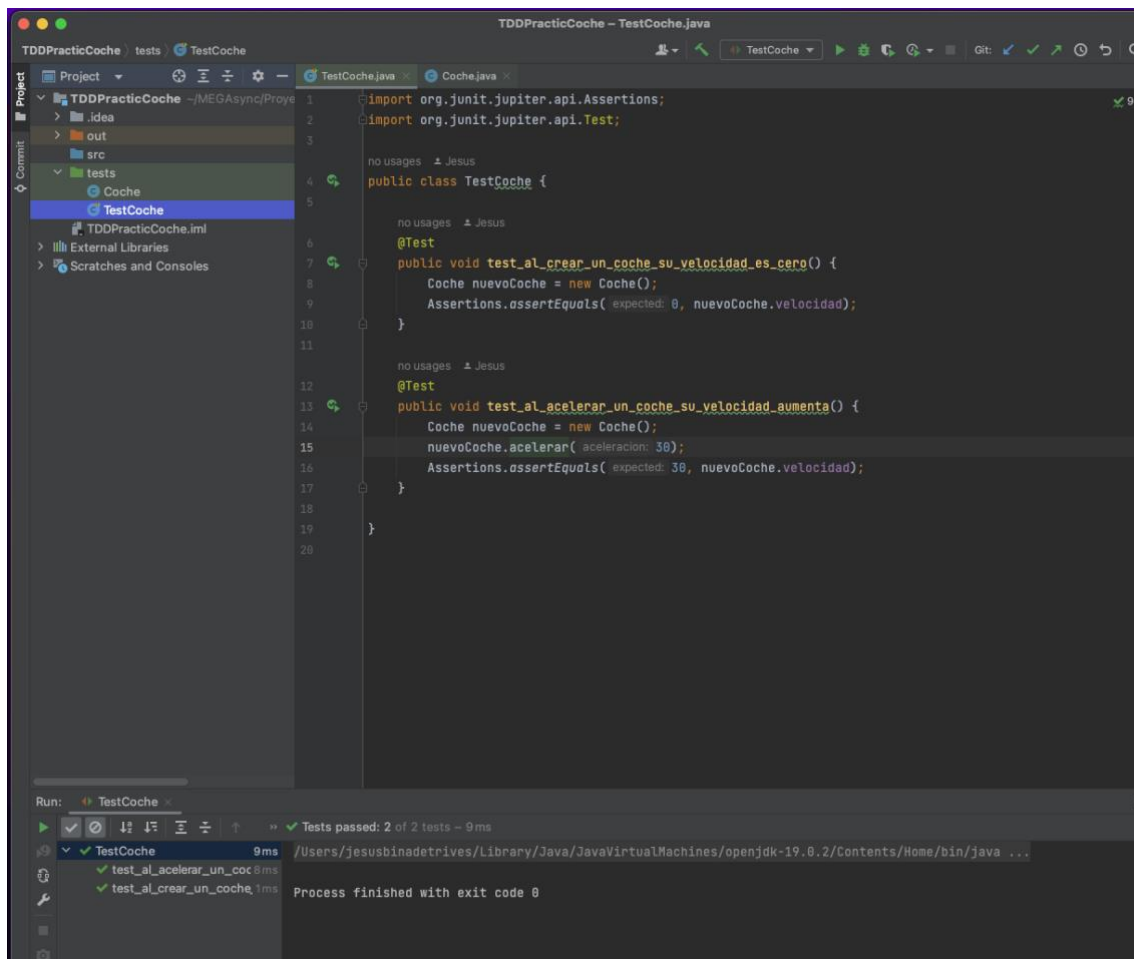
Cannot resolve method 'acelerar' in 'Coche'
[Create method 'acelerar' in 'Coche'](#) More actions...
No candidates found for method call **nuevoCoche.acelerar(30)**.
TDDPracticCoche

Creamos el método y le decimos que velocidad aumente en aceleración.

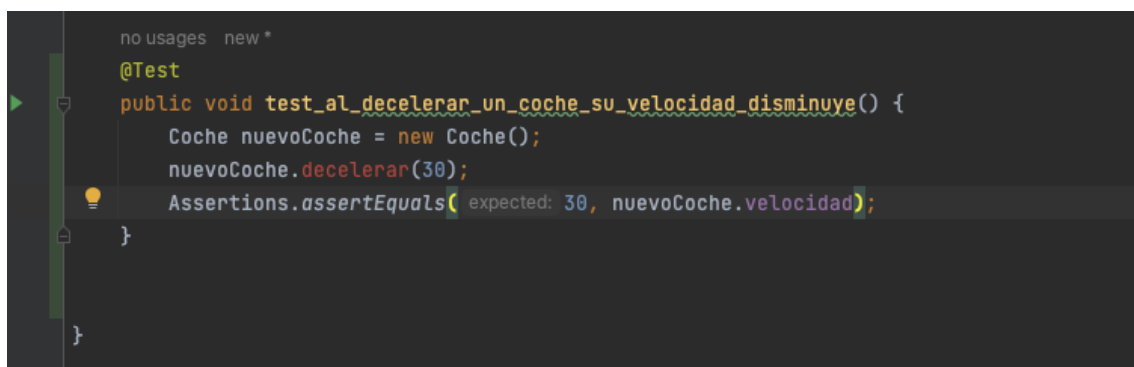
```
1 usage new *
public void acelerar(int aceleracion) {
    velocidad += aceleracion;
}
```

Click or press ↵

Ejecutamos y vemos como hemos pasado el test correctamente.



Después del método acelerar procedemos a implementar otro método para decelerar.



Hemos creado el método, pero si lo dejamos así no funcionaría puesto cuando creamos el coche este parte de una velocidad de 0 y aquí le hemos establecido 30, por lo que vamos a establecerle una velocidad mayor para que vaya decelerando. Además, debemos crear el método decelerar.

```
no usages new *
@Test
public void test_al_decelerar_un_coche_su_velocidad_disminuye() {
    Coche nuevoCoche = new Coche();
    nuevoCoche.velocidad = 50;
    nuevoCoche.decelerar(30);
    Assertions.assertThat(nuevoCoche.velocidad).isEqualTo(20);
}
```

Cannot resolve method 'decelerar' in 'Coche'
[Create method 'decelerar' in 'Coche'](#) [More actions...](#)

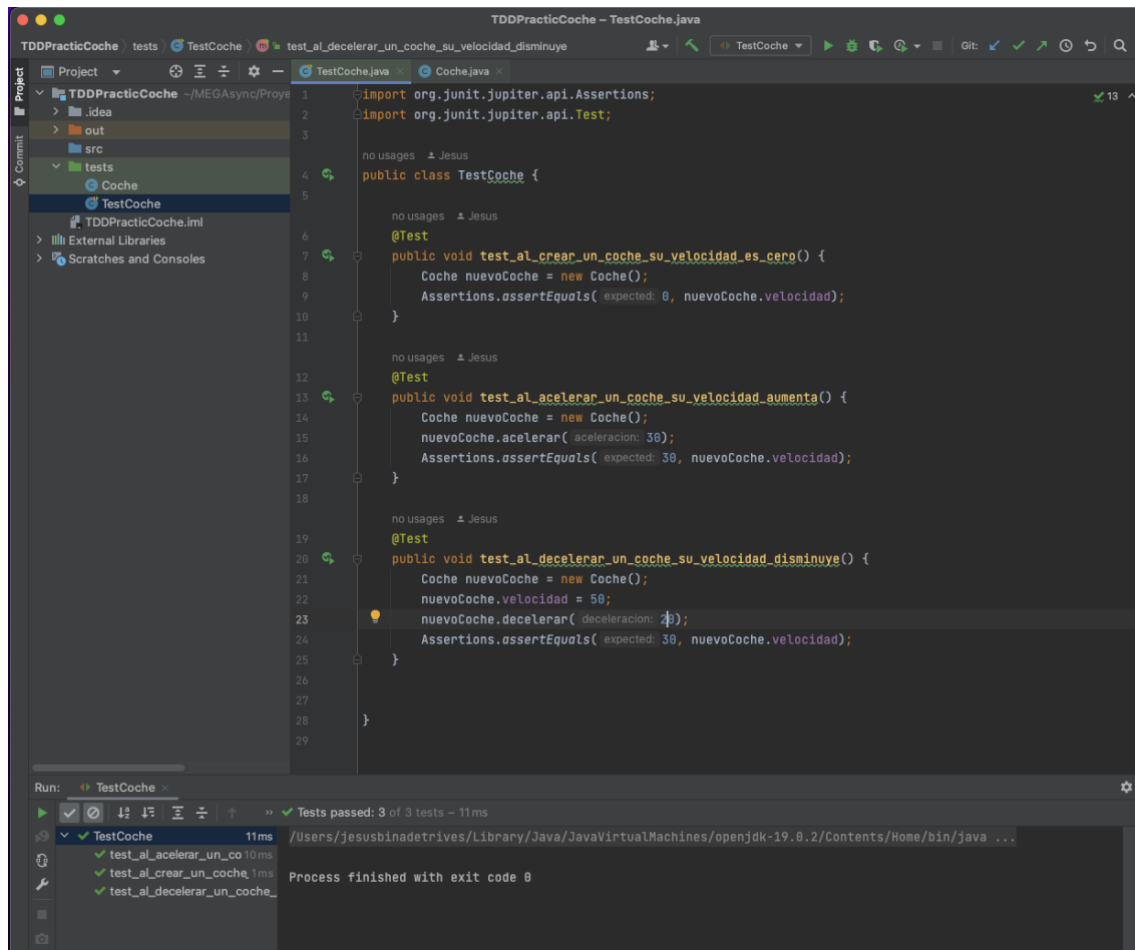
No candidates found for method call `nuevoCoche.decelerar(30)`.
 TDDPracticCoche

Creamos el método y le decimos que la velocidad disminuya en la deceleración.

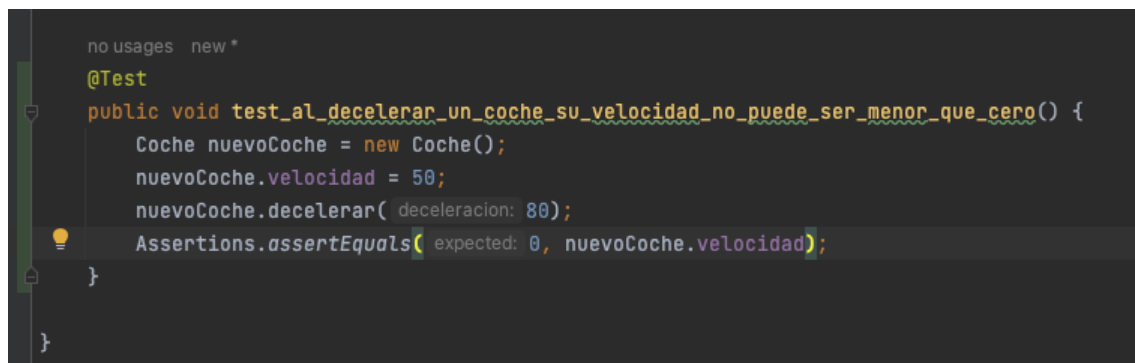
```
8
1 usage new *
9 public void decelerar(int deceleracion) {
10     velocidad -= deceleracion;
11 }
12 }
```

Hecho esto, tenemos el método para disminuir la velocidad partiendo en 50 y teniendo que decelerar hasta 20 por lo que, el valor esperado es de 30.

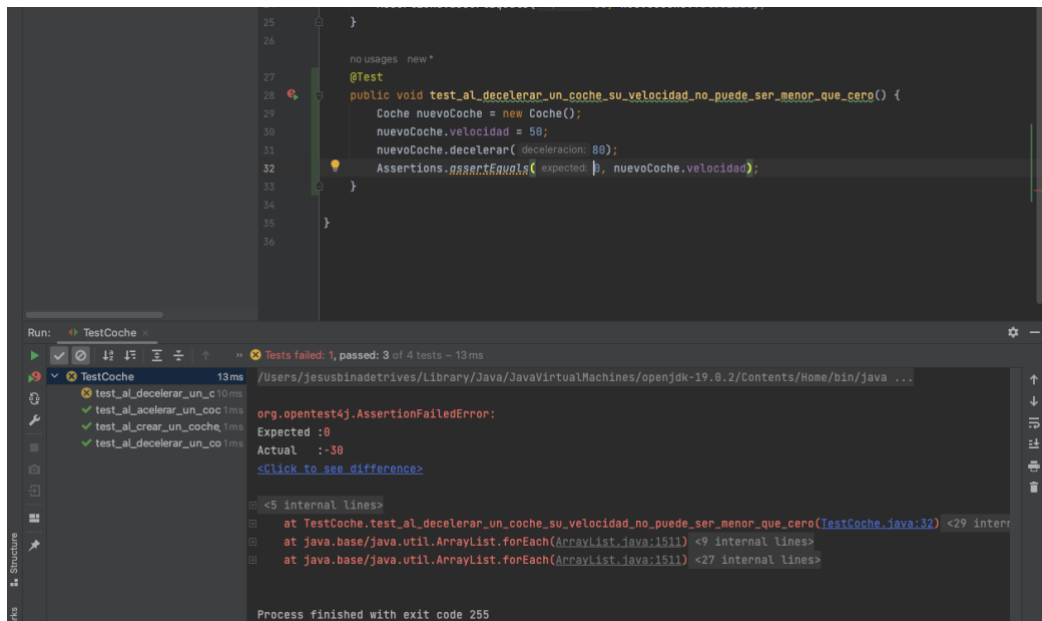
Ahora, ejecutamos el test y observamos como lo hemos pasado correctamente.



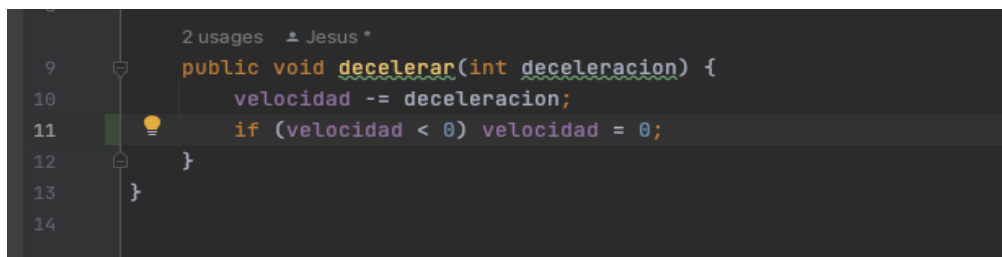
Por último, vamos a implementar un método para que la velocidad del coche no pueda ser menor que cero y así no obtener una velocidad negativa.



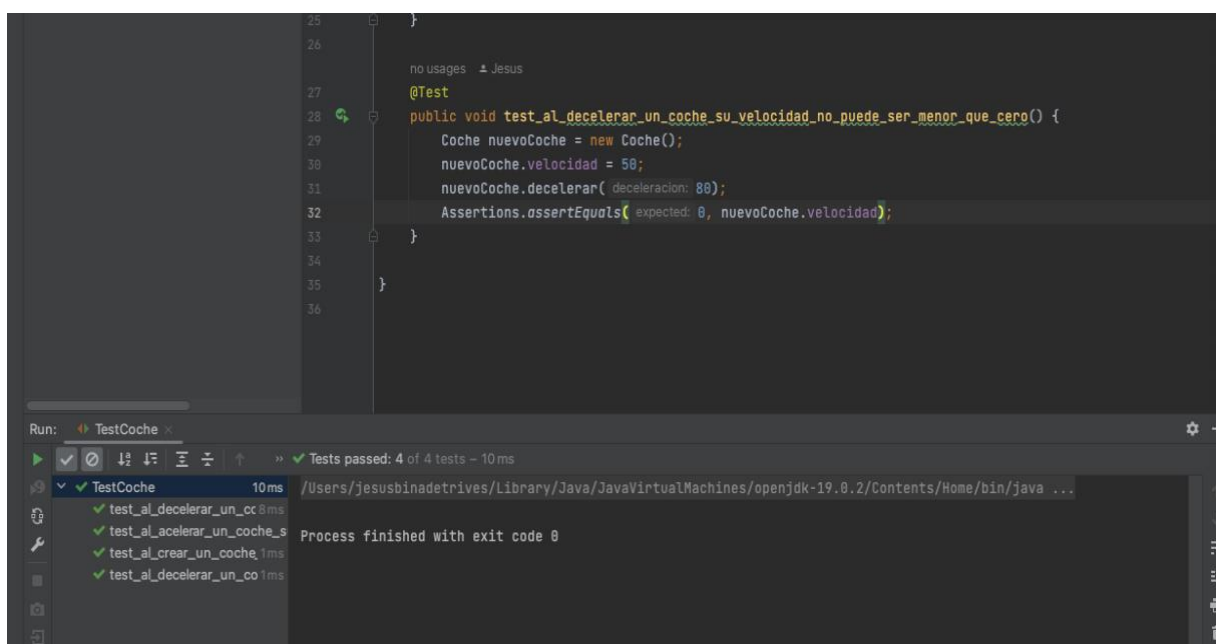
Hemos creado el método estableciendo que el coche lleva una velocidad de 50 y su deceleración en de 80 indicando que el valor esperado se de 0 porque como hemos dicho el valor no puede ser menor que cero. Observamos que el método se puede compilar, pero falla debido a que esperaba una velocidad de 0 y su actual es de -30.



Para resolver esto, debemos irnos a la clase coche y en el método decelerar tendremos que indicar que la velocidad no pueda ser menor que cero.

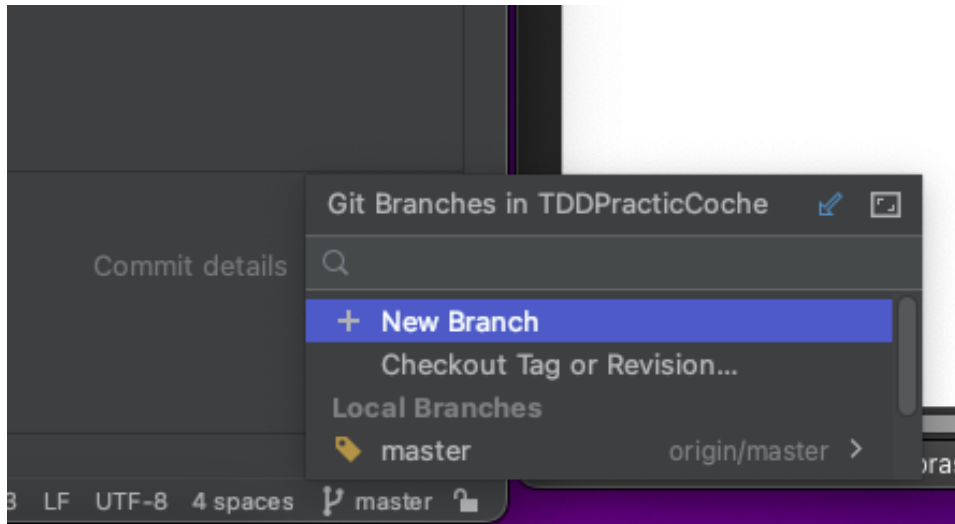


Hecho esto, ejecutamos de nuevo y vemos como ahora si hemos pasado el test satisfactoriamente.



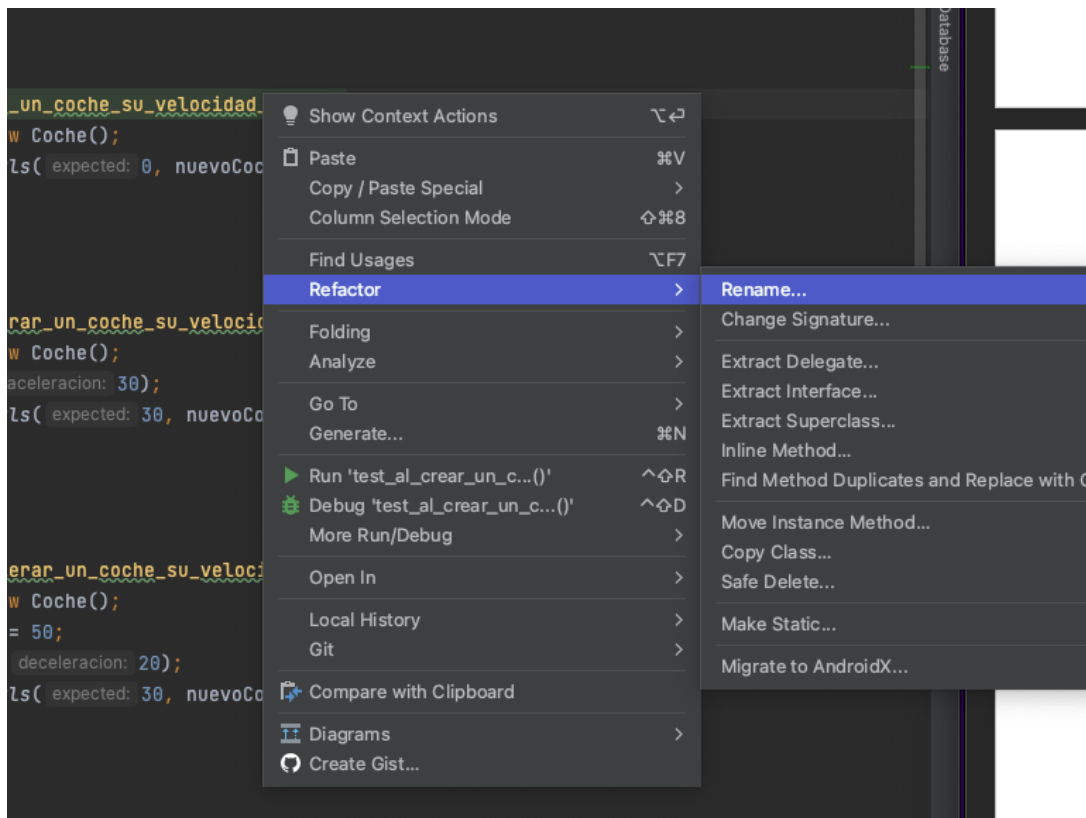
Realizados todos los test, procedemos a crear una nueva rama llamada Refactorizado en la que haremos una refactorización de todos los métodos añadiendo el nombre al final del método.

Para ello, en la esquina inferior derecha de IntelliJ hacemos click donde indica “master” y nos aparece lo siguiente.



Hacemos click en “new branch” y creamos la nueva rama con el nombre indicado.

Trabajando sobre la nueva rama, nos posicionamos sobre los nombres de los métodos y hacemos click derecho y seleccionamos la opción “refactor – rename”.



El método se muestra de este color y solamente deberemos añadir el nombre al final.

```
pages  Jesus
st
public void test_al_decelerar_un_coche_su_velocidad_disminuye_Jesus // () {
    Coche nuevoCoche = new Coche();
    nuevoCoche.velocidad = 50;
    nuevoCoche.decelerar( deceleracion: 20);
    Assertions.assertEquals( expected: 30, nuevoCoche.velocidad);
}
```

```
no usages  Jesus *
6
7  ▶  @Test
8      public void test_al_crear_un_coche_su_velocidad_es_cero_Jesus() {
9          Coche nuevoCoche = new Coche();
10         Assertions.assertEquals( expected: 0, nuevoCoche.velocidad);
11     }
12
13  no usages  Jesus *
14  ▶  @Test
15      public void test_al_acelerar_un_coche_su_velocidad_aumenta_Jesus() {
16          Coche nuevoCoche = new Coche();
17          nuevoCoche.acelerar( aceleracion: 30);
18          Assertions.assertEquals( expected: 30, nuevoCoche.velocidad);
19      }
20
21  no usages  Jesus *
22  ▶  @Test
23      public void test_al_decelerar_un_coche_su_velocidad_disminuye_Jesus() {
24          Coche nuevoCoche = new Coche();
25          nuevoCoche.velocidad = 50;
26          nuevoCoche.decelerar( deceleracion: 20);
27          Assertions.assertEquals( expected: 30, nuevoCoche.velocidad);
28      }
29
30  no usages  Jesus *
31  ▶  @Test
32      public void test_al_decelerar_un_coche_su_velocidad_no_puede_ser_menor_que_cero_Jesus() {
33          Coche nuevoCoche = new Coche();
34          nuevoCoche.velocidad = 50;
35          nuevoCoche.decelerar( deceleracion: 80);
36      }
37  }
```

```
1 usage  Jesus *
public void acelerar_Jesus(int aceleracion) {
    velocidad += aceleracion;
}

2 usages  Jesus *
public void decelerar_Jesus(int deceleracion) {
    velocidad -= deceleracion;
    if (velocidad < 0) velocidad = 0;
}
}
```

Observamos como en los métodos de test el nombre de los métodos acelerar y decelerar ha cambiado.

```
nuevoCoche.velocidad = 50;  
nuevoCoche.decelerar_Jesus( deceleracion: 20);  
Assertions.assertEquals( expected: 30, nuevoCoche.velocidad);  
}
```