

IL TEST DI UNITÀ CON JUnit

Per agevolare il programmatore nel compito sempre poco piacevole del testing, per tutti gli ambienti di sviluppo moderni sono disponibili framework che offrono un aiuto per la automatizzazione di test di unità: la soluzione viene data dalla famiglia **X-Unit**, dove **JUnit (Java)** è il capostipite, sviluppato originariamente da **Erich Gamma** and **Kent Beck**, gli autori dei **Design Pattern** e dell'**eXtreme Programming**; a esso si affiancano **CppUnit (C++)**, **csUnit (C#)**, **NUnit (.NET framework)** e **HttpUnit (Web Application)**.

Il vantaggio di automatizzare i test è notevole per lo sviluppatore:

- innanzitutto il risparmio di tempo per la loro esecuzione e l'affidabilità della loro esecuzioni in quanto, una volta impostati, non è più necessario l'intervento umano;
- inoltre la batteria di test può essere riutilizzata, almeno parzialmente, in caso si portino modifiche nella classe.

La parte più piccola che si vuole testare è chiamata **unità**, ma a seconda del progetto che si sta realizzando questa può corrispondere a una funzione, a un metodo, a una classe, a un package oppure a un intero sistema; nel nostro esempio noi considereremo come unità la classe.

Struttura di un metodo di test

In un metodo di test possiamo individuare quattro componenti:

- 1 **inizializzazione precondizioni**: limitatamente alle precondizioni tipiche del singolo caso di test, le altre potrebbero essere nel setup;
- 2 **inserimento valori di input**: tramite chiamate a metodi `set()` oppure tramite assegnazione di valori ad attributi pubblici;
- 3 **codice di test**: esecuzione del metodo da testare con gli eventuali parametri relativi a quel caso di test;
- 4 **valutazione delle asserzioni**: controllo di espressioni booleani (asserzioni) che devono risultare vere se il test dà esito positivo, ovvero se i dati di uscita e/o le postcondizioni riscontrati sono diversi da quelli attesi.

Abbiamo a disposizione un insieme di asserzioni che effettuano ciascuna uno specifico controllo: se una asserzione non è vera il test case fallisce.

L'asserzione più utilizzata è sicuramente `assertEquals()`, che restituisce **true** in caso di successo del confronto tra i suoi argomenti:

- se l'assert è vero, il metodo continua l'esecuzione;
- se vi è un assert falso, il metodo ferma l'esecuzione in quel punto, e il risultato del test case sarà "fallito".

Un esempio di `assertEquals()` è il seguente:

```
assertEquals(Object atteso, Object attuale)
```

Una variante molto comoda è quella riportata di seguito che mostra sullo schermo un messaggio predefinito in caso di fallimento, in modo da agevolare la localizzazione del difetto.

```
assertEquals(String messaggio, Object atteso, Object attuale)
```

Sono anche disponibili metodi appositi per il controllo di tipi specifici di dati:

METODO	OPERAZIONE
<code>static void assertEquals(boolean atteso, boolean attuale)</code>	Test tra variabili booleane
<code>static void assertEquals(int atteso, int attuale)</code>	Test tra variabili intere
<code>static void assertEquals(java.lang.String atteso, java.lang.String attuale)</code>	Test tra variabili Stringhe
<code>static void assertFalse(boolean condizione)</code>	Verifica valore falso della condizione
<code>static void assertTrue(boolean condizione)</code>	Verifica valore vero della condizione
<code>static void assertNull(java.lang.Object object)</code>	Verifica che oggetto sia Null

.....

Automaticamente viene generata la classe **CalcolatriceTest.java** (classe ereditata da **JUnit** che estende una classe denominata **TestCase** della libreria) e vengono predisposti automaticamente al suo interno un "campione" di metodo di test per ciascun metodo della classe; riportiamo come esempio quello predisposto per la somma, che testa di default un caso elementare, come si può vedere nella figura che segue.

```
@Test
public void testSomma() {
    System.out.println("somma");
    int a = 0;
    int b = 0;
    int expectedResult = 0;
    int result = Calcolatrice.somma(a, b);
    assertEquals(expectedResult, result);
    // TODO review the generated test code and remove the default
    fail("The test case is a prototype.");
}
```

Modifichiamo per esempio il primo metodo, quello che effettua la somma di due addendi in questo modo:

```
@Test
public void testSomma() {
    System.out.println("somma");
    int a = 3;
    int b = 12;
    int expectedResult = 15;
    int result = calcolatrice.somma(a, b);
    assertEquals("errore nella somma(3+12)", expectedResult, result);
}
```

Il metodo **assertEquals()** verifica se il valore ottenuto dall'esecuzione del metodo **somma** è uguale a 15 (valore atteso); in caso contrario viene incrementato il contatore degli errori considerando questa situazione come una failure e genera il messaggio di errore indicato.

Dopo aver completato di modificare e/o inserire i metodi desiderati che testano la nostra classe **Calcolatrice** mandiamo in esecuzione la classe **CalcolatriceTest** cliccando sull'opzione **Run File** dal menu che appare cliccando col tasto destro

.....

Automaticamente vengono eseguiti tutti i metodi presenti nella classe di test indicati dalla istruzione **@Test**: se non vengono riscontrati errori, nella sezione inferiore viene visualizzata una **barra di colore verde** che indica anche il numero di test eseguiti con successo, altrimenti la **barra è rossa** e su di essa viene indicato la percentuale di test che hanno avuto successo ed elencati i metodi test che hanno fallito, per i quali viene anche riportata la motivazione della difformità, in modo da permettere agevolmente il debugging.



Oltre alla predisposizione dei metodi per testare le singole operazioni, in automatico sono predisposti metodi per l'inizializzazione della classe prima/dopo l'esecuzione di tutti i test di una classe di test o delle singole istanze; li riportiamo per completezza:

- un metodo `setUpClass()` è un metodo `setUp()` che viene eseguito prima dell'esecuzione di ogni test, necessario per settare precondizioni comuni a più di un caso di test (**before**);
- un metodo `tearDownClass()` è un metodo `tearDown()` che viene eseguito dopo ogni caso di test che serve per resettare le postcondizioni (**after**).

Vediamo un possibile utilizzo dei metodi `setUpClass()` e `tearDown()`.

ESEMPIO

```
@BeforeClass
public static void setUpClass() throws Exception {
    Calcolatrice miaCalc = new Calcolatrice();
    assertNotNull(miaCalc);
}

@After
public void tearDown() throws Exception {
    Calcolatrice miaCalc = null;
    assertNull(miaCalc);
}
```

Riportiamo le annotazioni che possiamo porre prima dei metodi, in modo da stabilire quando questi devono essere richiamati oppure la specifica del test vero e proprio:


- `@BeforeClass`: richiamato all'inizializzazione della classe di test;
- `@Before`: richiamato all'inizio di ogni metodo della classe di test;
- `@After`: richiamato alla fine di ogni metodo della classe di test;
- `@AfterClass`: richiamato alla chiusura della classe di test;
- `@Test`: identifica un metodo di test vero e proprio;
- `@Test (timeout = 100)`: identifica un metodo di test vero e proprio che fallisce qualora superi la durata di 100 millisecondi;
- `@Test (expected = Exception.class)`: identifica un metodo di test vero e proprio che si supera solo se viene rilevata l'eccezione descritta.


Come usare JUnit in IntelliJ

Add dependencies

For our project to use JUnit features, we need to add JUnit as a dependency.

1. Open `pom.xml` in the root directory of your project.

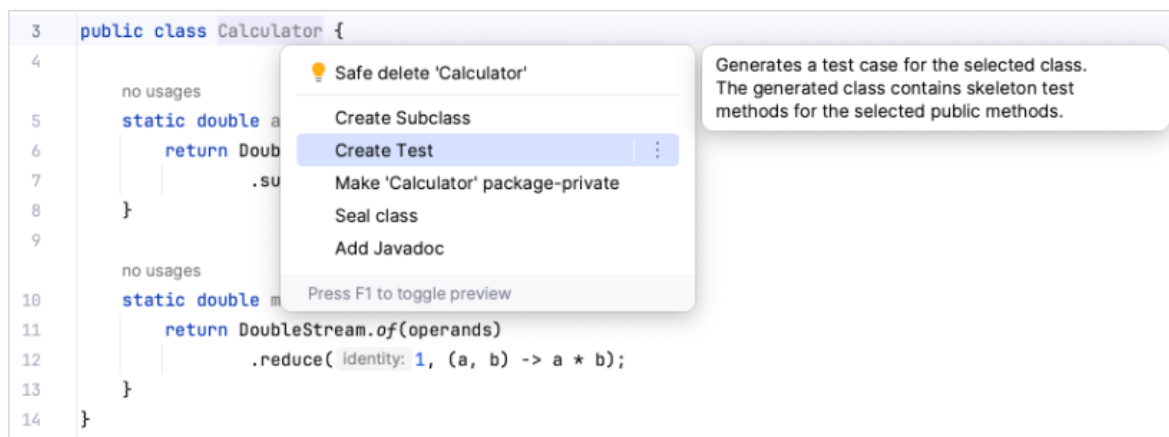
 To quickly navigate to a file, press `Ctrl` `Shift` `N` and enter its name.

2. In `pom.xml`, press `Alt` `Insert` and select **Dependency**.
3. In the dialog that opens, type `org.junit.jupiter:junit-jupiter` in the search field.
Locate the necessary dependency in the search results and click **Add**.
4. When the dependency is added to `pom.xml`, press `Ctrl` `Shift` `O` or click  in the **Maven** tool window to import the changes.

Create tests

Now let's create a test. A **test** is a piece of code whose function is to check if another piece of code is operating correctly. In order to do the check, it calls the tested method and compares the result with the predefined **expected result**. An expected result can be, for example, a specific return value or an exception.

1. Place the caret at the `Calculator` class declaration and press `Alt` `Enter`. Alternatively, right-click it and select **Show Context Actions**. From the menu, select **Create test**.



```
3 public class Calculator {
4
5     no usages
6     static double a
7     return Double
8 }
9
10 no usages
11 static double m
12 return DoubleStream.of(operands)
13     .reduce(Identity: 1, (a, b) -> a * b);
14 }
```

Safe delete 'Calculator'

- Create Subclass
- Create Test
- Make 'Calculator' package-private
- Seal class
- Add Javadoc

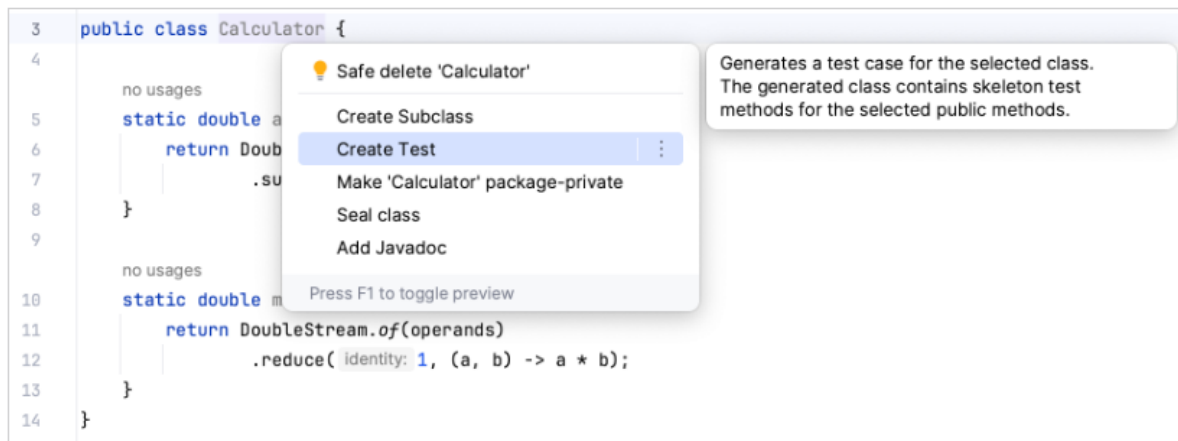
Press F1 to toggle preview

Generates a test case for the selected class. The generated class contains skeleton test methods for the selected public methods.

Create tests

Now let's create a test. A **test** is a piece of code whose function is to check if another piece of code is operating correctly. In order to do the check, it calls the tested method and compares the result with the predefined **expected result**. An expected result can be, for example, a specific return value or an exception.

1. Place the caret at the `Calculator` class declaration and press `Alt` `Enter`. Alternatively, right-click it and select **Show Context Actions**. From the menu, select **Create test**.



2. Select the two class methods that we are going to test.

Create Test

Testing library: JUnit5

Class name: CalculatorTest

Superclass:

Destination package:

Generate: ☐ setUp/@Before ☐ tearDown/@After

Generate test methods for: ☐ Show inherited methods

...	Member
<input checked="" type="checkbox"/>	add(operands:double...):double
<input checked="" type="checkbox"/>	multiply(operands:double...):double

? Cancel OK


3. The editor takes you to the newly created test class. Modify the `add()` test as follows:

```
@Test
@DisplayName("Add two numbers")
void add() {
    assertEquals(4, Calculator.add(2, 2));
}
```


This simple test will check if our method correctly adds 2 and 2. The `@DisplayName` annotation specifies a more convenient and informative name for the test.

Run tests and view their results

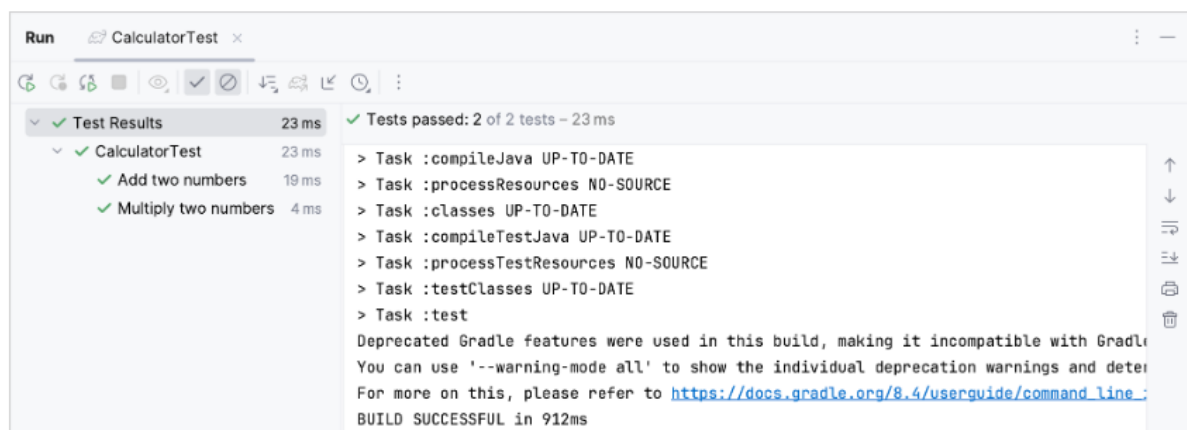
After we have set up the code for the testing, we can run the tests and find out if the tested methods are working correctly.

- To run an individual test, click  in the gutter and select **Run**.



- To run all tests in a test class, click  against the test class declaration and select **Run**.

You can view test results in the **Run** tool window.



More options

To view more options for how to run unit tests that are based on the JUnit testing framework, select **Modify options** in the **Run/Debug Configurations** dialog. This will open the **Add Run Options** dialog.

