Università di Catania

Uni ct MATEMATICA E INFORMATICA

Unit Testing

Alessandro Midolo, PhD

Dipartimento di Matematica e Informatica

Università di Catania

alessandro.midolo@unict.it

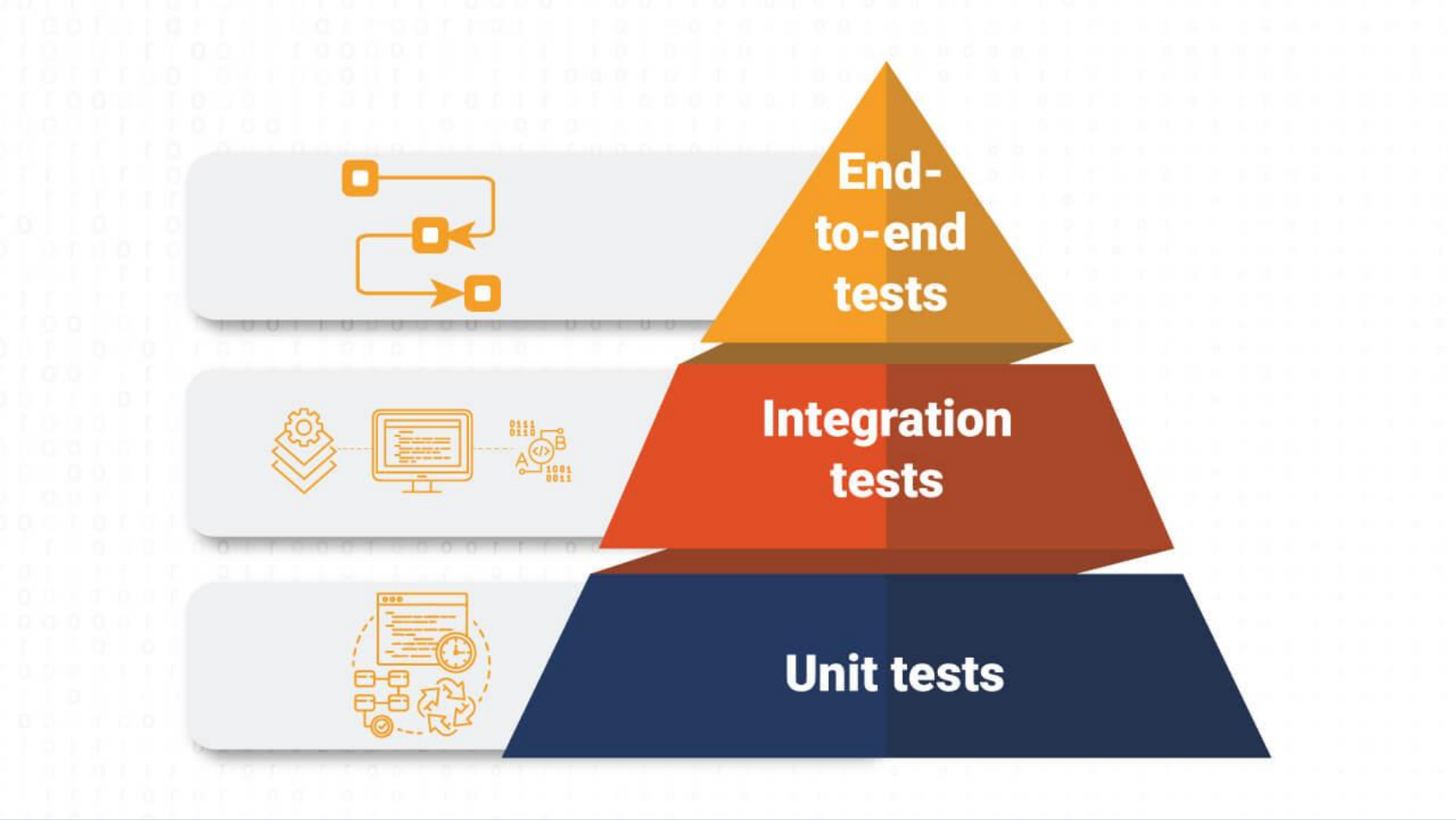
https://www.dmi.unict.it/amidolo/

A.A. 2024/2025

Unit Test

- Un **unit test** valuta il comportamento di una singola unità di lavoro, che generalmente corrisponde a un metodo specifico
- Ad esempio, se forniamo il valore x al metodo m(), restituirà y?
- L'unit test verifica che il **metodo** rispetti i **termini** definiti dal contratto della sua API, ovvero l'accordo stabilito dalla firma del metodo
- Inoltre, l'unit test funge da **documentazione eseguibile**, mostrando come un determinato metodo debba essere utilizzato
- Altre tipologie di test includono:
  - **Integration test**: verifica il corretto funzionamento di più unità che interagiscono tra loro
  - **End-to-end test**: verifica il comportamento dell'intero sistema, ad esempio, dal punto di vista dell'utente finale, simulando interazioni con l'interfaccia grafica o le API esposte

La piramide dei Test



Integration Testing

- L'***integration test*** (test di integrazione) verifica il corretto funzionamento di più componenti o unità di un sistema che interagiscono tra loro
- A differenza dei *unit test*, che testano singole unità di lavoro (come metodi o funzioni isolati), l'integration test si concentra sull'**interazione tra più moduli o servizi** per assicurarsi che comunichino correttamente e che il sistema funzioni come previsto quando le unità si integrano
- Ad esempio, un **test di integrazione** potrebbe verificare se un modulo che gestisce l'autenticazione dell'utente comunica correttamente con un modulo che gestisce il database per verificare le credenziali
- Questi test sono particolarmente utili per scoprire problemi che non sono visibili nei singoli componenti ma emergono quando questi interagiscono tra loro

End-to-End Testing

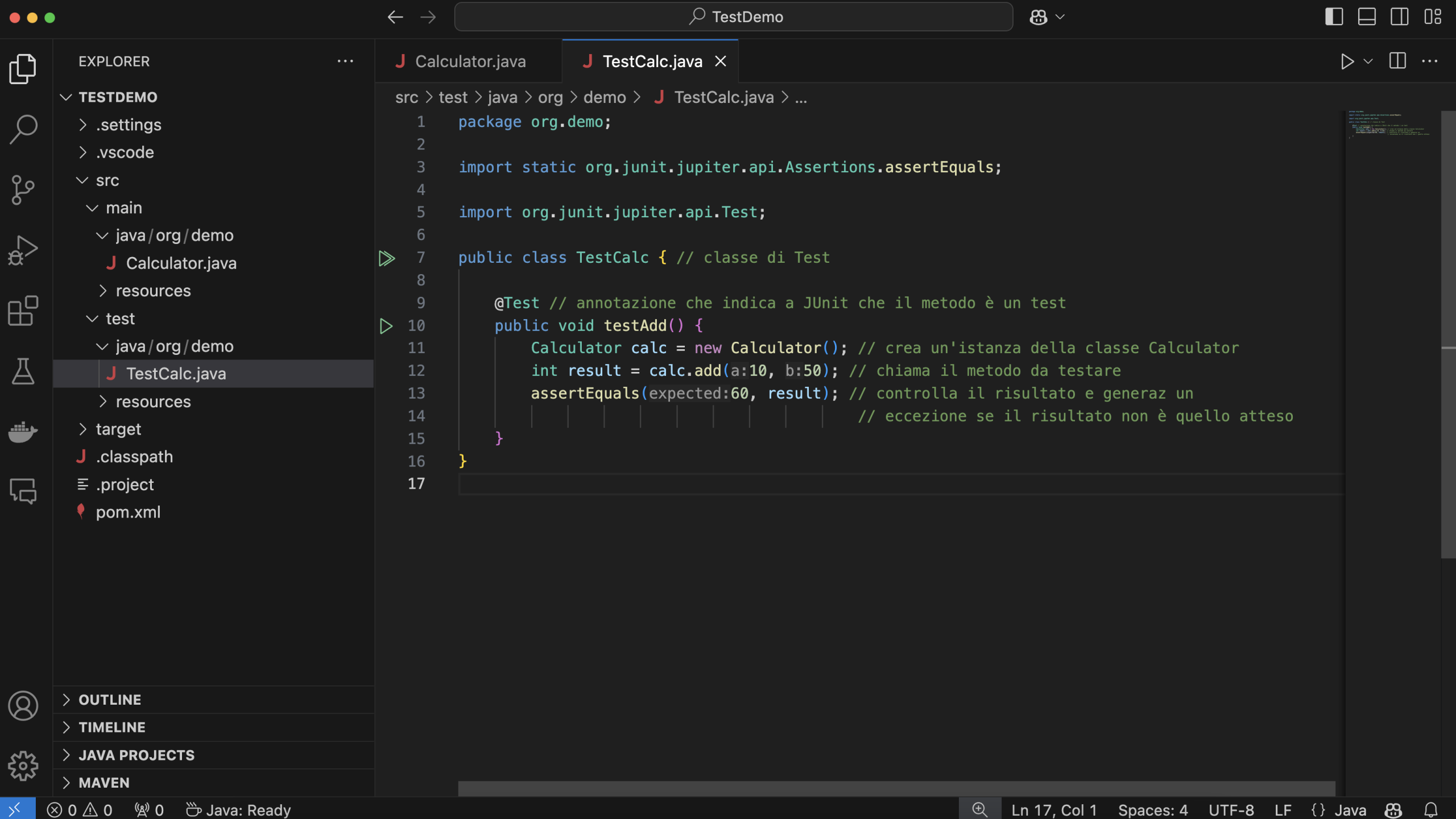
- L'***end-to-end testing*** (test end-to-end) è una metodologia di testing che verifica il funzionamento dell'intero sistema, simulando un'esperienza completa dell'utente finale
- L'obiettivo è testare l'applicazione nel suo **insieme**, dalla fase di **input** fino **all'output**, per assicurarsi che tutti i componenti del sistema funzionino correttamente insieme
- In pratica, si **simula un flusso di lavoro completo** che un utente potrebbe eseguire, come l'interazione con l'interfaccia grafica (UI) o l'uso delle API, per verificare che tutte le funzionalità siano integrate e che il sistema, nel suo complesso, produca i risultati attesi
- Questo tipo di test aiuta a identificare problemi che potrebbero sorgere quando diverse parti del sistema interagiscono tra loro

JUnit

- JUnit è un **framework di testing** per il linguaggio di programmazione Java, progettato per supportare la scrittura e l'esecuzione di test automatici, in particolare *unit test*
- È uno degli strumenti più usati per testare il codice **Java**, aiutando gli sviluppatori a garantire che il loro software funzioni correttamente
- Principi del framework **Junit**
  - Ciascun unit test è un metodo
  - Ogni metodo di test fornito è trovato ed eseguito dal framework usando la riflessione computazionale
  - Si usano istanze di classi di test separate e class loader separati, per evitare effetti collaterali, quindi per avere esecuzioni indipendenti
  - Si hanno una varietà di asserzioni per automatizzare il controllo dei risultati dei test
  - Integrazione con tool e IDE diffusi

Riflessione Computazionale e Istanziazione

- La **riflessione computazionale** è una funzionalità del linguaggio Java che permette di esaminare e manipolare il comportamento di classi, metodi, e oggetti in fase di esecuzione, senza conoscere il codice sorgente in anticipo
- In JUnit, quando si eseguono i test, il framework utilizza la riflessione per individuare i metodi di test annotati con **@Test** all'interno di una classe di test. Poi esegue questi metodi dinamicamente. Questo consente a JUnit di individuare **automaticamente** tutti i metodi di test, anche se non sono stati dichiarati esplicitamente o non sono scritti in un ordine prestabilito
- JUnit crea **una nuova istanza** della classe di test per ogni metodo di test che deve eseguire. Questo è importante perché assicura che ogni test venga eseguito su un "nuovo" stato, evitando che i test influenzino l'uno l'altro. Ad esempio, se un test modifica una variabile d'istanza, quella modifica non influenzerà altri test
- Inoltre, JUnit usa **class loader** separati per caricare le classi di test, garantendo che ogni test venga eseguito in un **ambiente isolato**. Questo isolamento aiuta a prevenire effetti collaterali tra i test, come la condivisione accidentale di risorse o variabili globali che potrebbero interferire con il comportamento di altri test


TestDemo
EXPLORER
TESTDEMO
.settings
.vscode
src
main
java/org/demo
Calculator.java
resources
test
java/org/demo
TestCalc.java
resources
target
.classpath
.project
pom.xml
OUTLINE
TIMELINE
JAVA PROJECTS
MAVEN
Calculator.java
TestCalc.java
src > test > java > org > demo > TestCalc.java > ...
1 package org.demo;
2
3 import static org.junit.jupiter.api.Assertions.assertEquals;
4
5 import org.junit.jupiter.api.Test;
6
7 public class TestCalc { // classe di Test
8
9 @Test // annotazione che indica a JUnit che il metodo è un test
10 public void testAdd() {
11 Calculator calc = new Calculator(); // crea un'istanza della classe Calculator
12 int result = calc.add(a:10, b:50); // chiama il metodo da testare
13 assertEquals(expected:60, result); // controlla il risultato e generaz un
14 // eccezione se il risultato non è quello atteso
15 }
16 }
17
0 0 0 Java: Ready
Ln 17, Col 1 Spaces: 4 UTF-8 LF Java

Fondamenti

- JUnit crea una **nuova istanza** della classe di test prima di invocare ciascun metodo annotato con ***@Test***
  - Per evitare effetti indesiderati
  - Non si possono quindi riusare variabili fra un metodo ed un altro
  - Ogni test è indipendente dagli altri
- Per verificare se il codice si comporta come ci si aspetta si usa una ***assertion***, ovvero una chiamata al metodo assert, che verifica se il risultato ottenuto (***actual***) coincide con il risultato atteso (***expected***)
- La classe che fornisce i metodi usati per valutare le esecuzioni è la classe Assert
- **assertTrue(boolean condition)** valuta se conditionè true se non è così il test ha trovato un errore
- **assertEquals(int a, int b)** verifica se due int sono uguali
- I metodi assert registrano fallimenti o errori e li riportano
- Quando si verifica un fallimento o un errore, l'esecuzione del metodo di test viene interrotta, ma verranno eseguiti gli altri metodi di test della stessa classe

Tipologie di Assert

- assertNull(actual)
- assertSame(expected, actual)
- assertTrue(boolean condition)
- assertFalse(boolean condition)
- fail(String message)

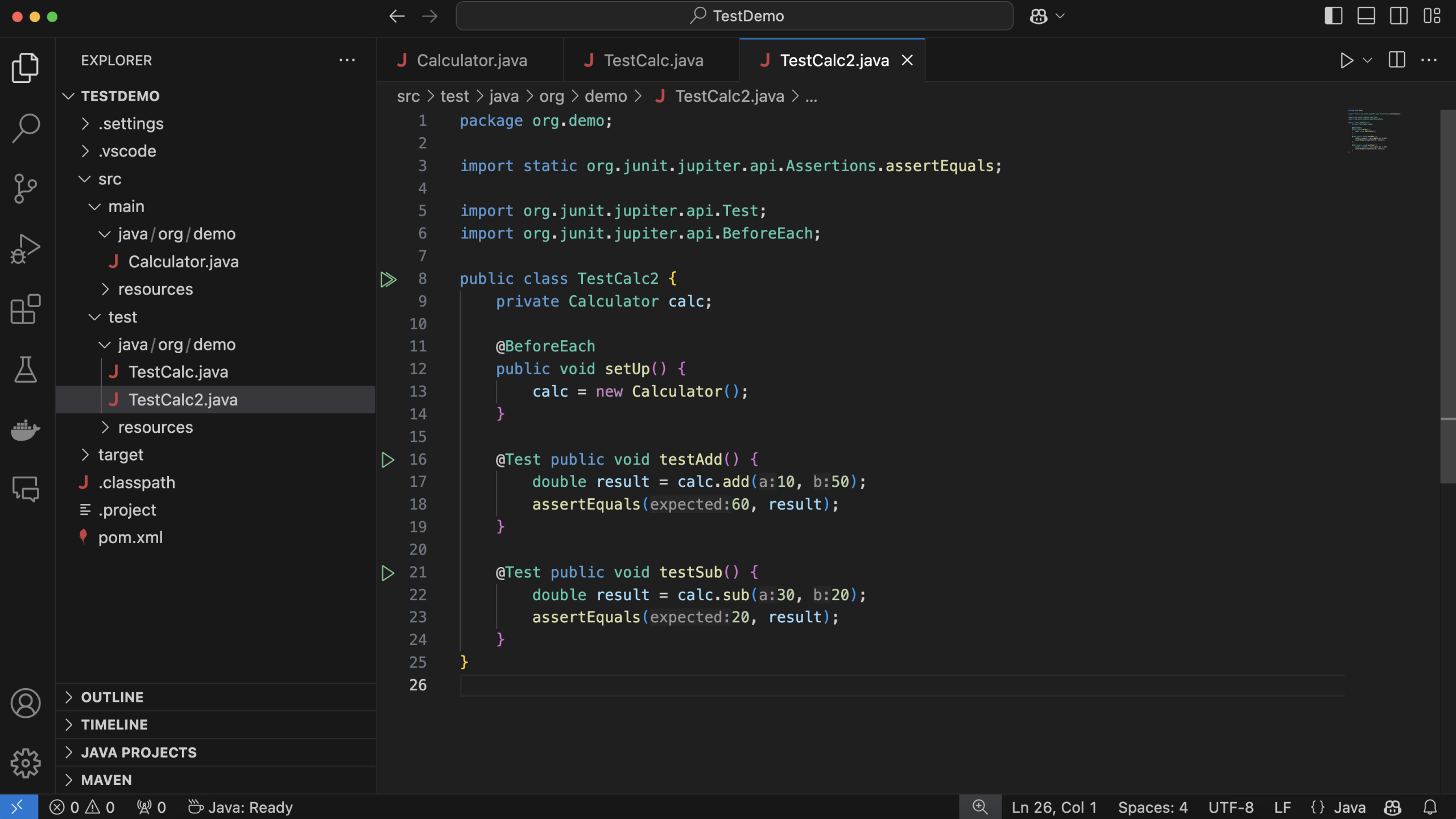
Source: https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html

Test Suite

```java
public class TestCaseA { // costruisco i casi di test
    @Test public void testA1() { ... }
}

public class TestCaseB {
    @Test public void testB1() { ... }
}

@RunWith(Suite.class) // Runner: classe che esegue i test
@SuiteClasses({TestCaseA.class}) // indico i casi di test della suite
public class TestSuiteA { }

@RunWith(Suite.class)
@SuiteClasses({ TestSuiteA.class, TestSuiteB.class })
public class MasterTestSuite { }
```

BeforeEach & AfterEach

- Un metodo annotato con ***@BeforeEach*** (setUp) viene eseguito prima dell'esecuzione di ciascun metodo @Test
  - Serve ad inizializzare lo stato prima del test
- Analogamente, un metodo ***@AfterEach*** (tearDown) viene eseguito dopo l'esecuzione di ciascun metodo @Test
  - Comunque vada l'esecuzione
  - Serve a portare il sistema ad uno stato opportuno (clean)
- ***@BeforeAll*** annota un metodo statico che verrà chiamato solo una volta prima dell'esecuzione di tutti i metodi @Test
  - Utile per eseguire operazioni costose, es. apertura connessione con un database
  - Essendo un metodo statico, può modificare solo attributi statici della classe
  - Può rendere i test meno indipendenti tra loro
- Analogamente ***@AfterAll***

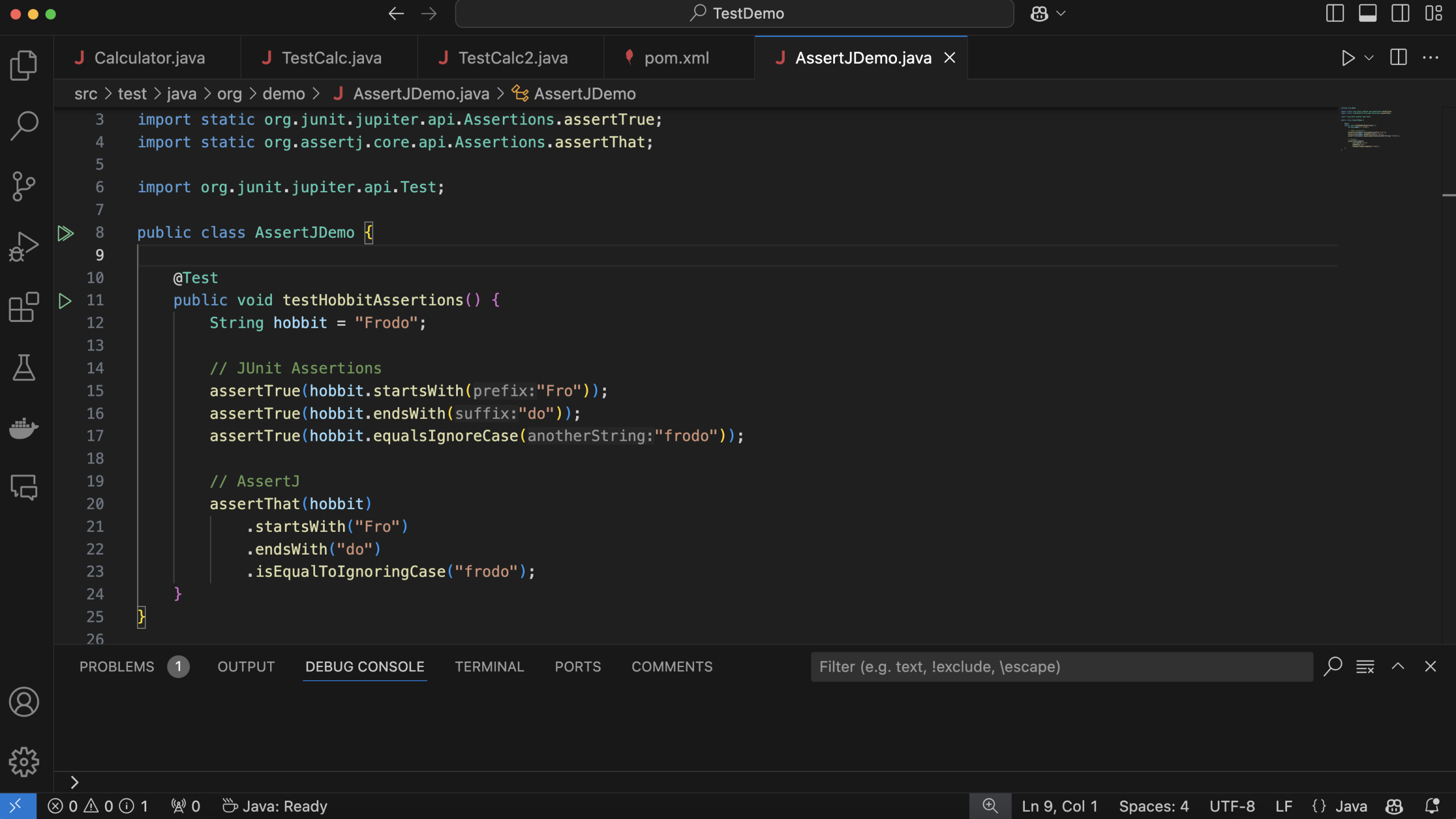

TestDemo
EXPLORER
TESTDEMO
.settings
.vscode
src
main
java/org/demo
Calculator.java
resources
test
java/org/demo
TestCalc.java
TestCalc2.java
resources
target
.classpath
.project
pom.xml
OUTLINE
TIMELINE
JAVA PROJECTS
MAVEN
Calculator.java
TestCalc.java
TestCalc2.java
src > test > java > org > demo > TestCalc2.java > ...
1 package org.demo;
2
3 import static org.junit.jupiter.api.Assertions.assertEquals;
4
5 import org.junit.jupiter.api.Test;
6 import org.junit.jupiter.api.BeforeEach;
7
8 public class TestCalc2 {
9 private Calculator calc;
10
11 @BeforeEach
12 public void setUp() {
13 calc = new Calculator();
14 }
15
16 @Test public void testAdd() {
17 double result = calc.add(a:10, b:50);
18 assertEquals(expected:60, result);
19 }
20
21 @Test public void testSub() {
22 double result = calc.sub(a:30, b:20);
23 assertEquals(expected:20, result);
24 }
25 }
26
0 0 0 Java: Ready
Ln 26, Col 1 Spaces: 4 UTF-8 LF Java

Convenzioni per il Naming

- Metodo da testare: int sum(int a, int b) throws NegativeNumbersException
- Scegliere una convenzione e usarla coerentemente per tutti i test del progetto
- **test[nome_metodo]**
  - testSum (ci limita ad un solo test case!)
- **test[feature_da_testare]**
  - testSumBiggerThenZeroNumbers
  - testSumWithNegativeNumber
- **when[stato_da_testare]_expect[comportamento_atteso]**
  - whenBiggerThenZeroNumbers_expectReturnSum
  - when_negativeNumber_expect_throwException
- **should[comportamento_atteso]_when[stato_da_testare]**
  - shouldReturnSum_whenBiggerThenZeroNumbers
  - should_returnSum_when_biggerThenZeroNumbers
- **given[precondizioni]_when[stato_da_testare]_then[comp_atteso] (BDD)**
  - given_initializedCalculator_when_negativeNumber_expect_throwException
  - given_initializedCalculator_when_biggerThenZeroNumbers_expect_returnSum

Assertj

- **AssertJ** è una libreria Java che fornisce un'API fluida e potente per scrivere asserzioni nei test
- Permette di concatenare metodi in modo leggibile
- Supporta molti tipi di asserzioni, ad esempio per stringhe, collezioni, mappe, oggetti personalizzati, numeri, file, e molto altro
- Include metodi specifici per lavorare con liste, array, set e mappe
- Consente di scrivere **asserzioni personalizzate** per le classi definite dall'utente

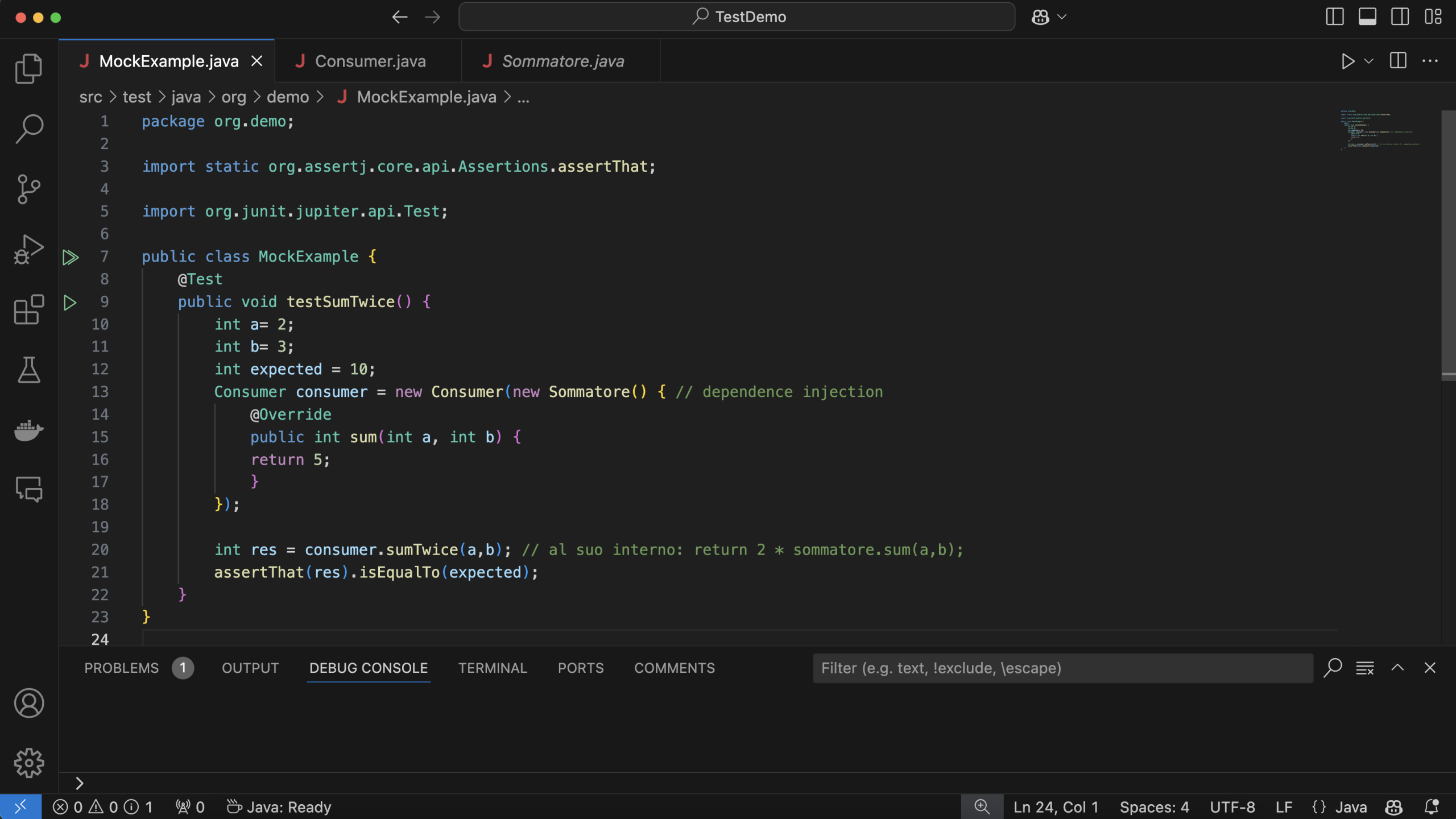

TestDemo
Calculator.java
TestCalc.java
TestCalc2.java
pom.xml
AssertJDemo.java
src > test > java > org > demo > AssertJDemo.java > AssertJDemo
3 import static org.junit.jupiter.api.Assertions.assertTrue;
4 import static org.assertj.core.api.Assertions.assertThat;
5
6 import org.junit.jupiter.api.Test;
7
8 public class AssertJDemo {
9
10 @Test
11 public void testHobbitAssertions() {
12 String hobbit = "Frodo";
13
14 // JUnit Assertions
15 assertTrue(hobbit.startsWith(prefix:"Fro"));
16 assertTrue(hobbit.endsWith(suffix:"do"));
17 assertTrue(hobbit.equalsIgnoreCase(anotherString:"frodo"));
18
19 // AssertJ
20 assertThat(hobbit)
21 .startsWith("Fro")
22 .endsWith("do")
23 .isEqualToIgnoringCase("frodo");
24 }
25 }
26
PROBLEMS 1
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
COMMENTS
Filter (e.g. text, !exclude, \escape)
0 0 1
0
Java: Ready
Ln 9, Col 1
Spaces: 4
UTF-8
LF
Java

Mock e Dependence Injection

- Uno unit test, a differenza di un integration test, deve testare una singola unità (es. metodo) in maniera indipendente dalle classi con cui **collabora**
- Necessario **simulare il comportamento** delle componenti da cui il metodo testato dipende tramite dei Mock (fantocci)
- Un Mock è un'istanza di una classe il cui comportamento di alcuni dei suoi metodi viene definito direttamente nel test
- Deve essere possibile passare il mock al metodo come **dipendenza** (es. tramite costruttore della classe, parametro, o metodo setter): questa pratica viene detta **dependence injection**
- É buona norma fare in modo che la classe si leghi ad un'interfaccia invece che ad una classe, disaccoppiandola quindi da un'implementazione specifica: il mock verrà creato come **implementazione** di questa interfaccia
- Questo permette di testare un metodo anche quando una classe da cui dipende non è stata ancora implementata o quando il comportamento della dipendenza varia ad ogni invocazione (es. generatore di valori casuali)

Esempio di Mock senza uso di librerie

- Supponiamo di voler testare il metodo sum di una classe **Consumer** che invoca al suo interno il metodo sum definito nell'interfaccia **Sommatore**
- Un'istanza compatibile con l'interfaccia sommatore viene passata in questo caso come dipendenza al costruttore della classe da testare
- Nel test possiamo definire un'implementazione dell'interfaccia come classe anonima, implementando tutti i metodi dell'interfaccia (in questo caso solo uno) in modo che si limiti a restituire al metodo da testare un risultato predefinito


TestDemo
MockExample.java
Consumer.java
Sommatore.java
src > test > java > org > demo > MockExample.java > ...
1 package org.demo;
2
3 import static org.assertj.core.api.Assertions.assertThat;
4
5 import org.junit.jupiter.api.Test;
6
7 public class MockExample {
8 @Test
9 public void testSumTwice() {
10 int a= 2;
11 int b= 3;
12 int expected = 10;
13 Consumer consumer = new Consumer(new Sommatore() { // dependence injection
14 @Override
15 public int sum(int a, int b) {
16 return 5;
17 }
18 });
19
20 int res = consumer.sumTwice(a,b); // al suo interno: return 2 * sommatore.sum(a,b);
21 assertThat(res).isEqualTo(expected);
22 }
23 }
24
PROBLEMS 1
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
COMMENTS
Filter (e.g. text, !exclude, \escape)
0 0 1
0
Java: Ready
Ln 24, Col 1
Spaces: 4
UTF-8
LF
Java

Mockito

- Mockito è una libreria per semplificare la creazione dei **Mock**
- Durante l'esecuzione di un test, mockito permette:
  - di definire il valore da restituire quando un metodo del mock viene chiamato durante l'esecuzione del test (when/given)
  - di verificare se e con quali parametri un determinato metodo del mock sia stato chiamato durante l'esecuzione del test (verify/then)
- A differenza dell'approccio tramite classi astratte, non siamo obbligati a definire il comportamento di tutti i metodi dell'interfaccia

```java
import static org.mockito.Mockito.*;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.junit.jupiter.MockitoExtension;

import static org.assertj.core.api.Assertions.*;

@ExtendWith(MockitoExtension.class)
public class ConsumerTest {

    @Mock
    Sommatore sommatore;

    @Test
    public void testSumTwice_withMockito() {
        int a = 2;
        int b = 3;
        int expected = 10;

        // given
        when(sommatore.sum(a, b)).thenReturn(value:5);

        // when
        Consumer consumer = new Consumer(sommatore);
        int res = consumer.sumTwice(a, b);

        // then
        verify(sommatore, times(wantedNumberOfInvocations:1)).sum(a, b); // Verify it was called once with "a" and "b"
        assertThat(res).isEqualTo(expected);
    }
}
```

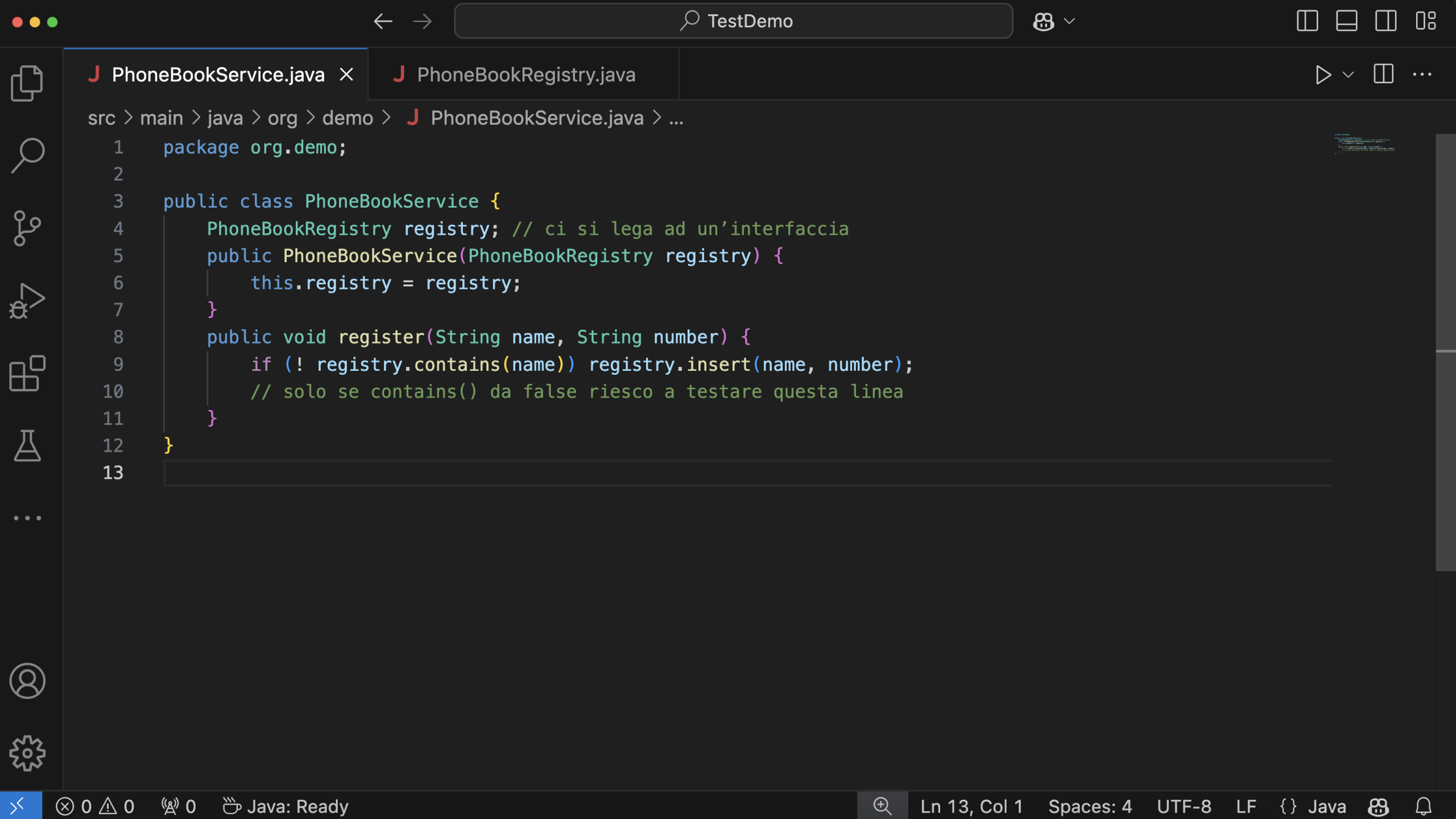
- **@ExtendWith(MockitoExtension.class)** estende la classe di test con MockitoExtension, che inizializza automaticamente tutti i mock annotati con @Mock
- **@Mock**, la dipendenza Sommatore è annotata con @Mock, quindi viene "simulata" (mockata) usando Mockito. Questo consente di controllare il comportamento della dipendenza senza dover usare un'implementazione reale
- **when** rappresenta il setup iniziale, e rappresenta una "simulazione" del comportamento del metodo reale
- quindi **creiamo** l'istanza che prende in input il mock che abbiamo creato
- **verifichiamo** che il risultato **attuale** sia uguale a quello **atteso**

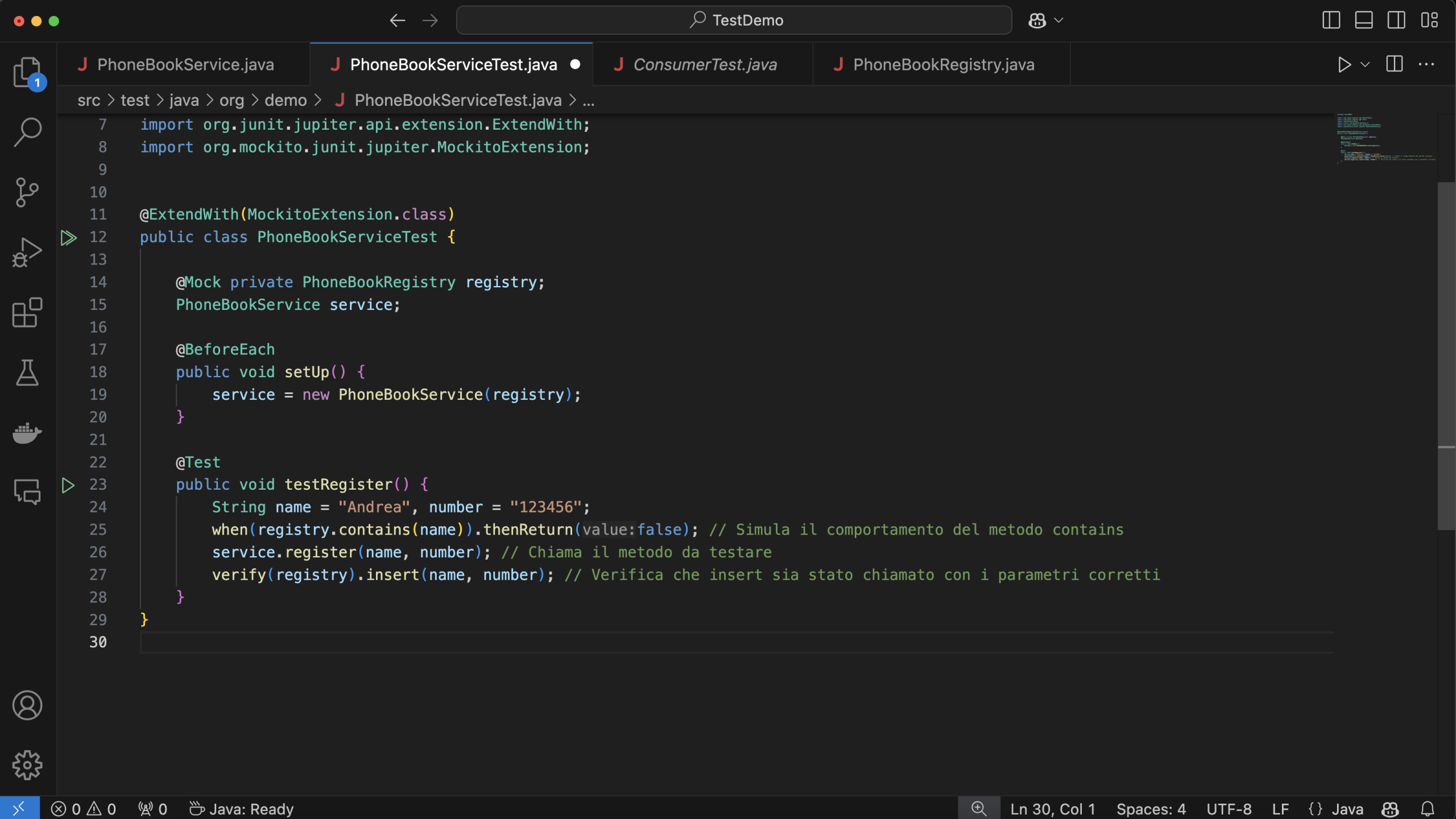
Test "Double"

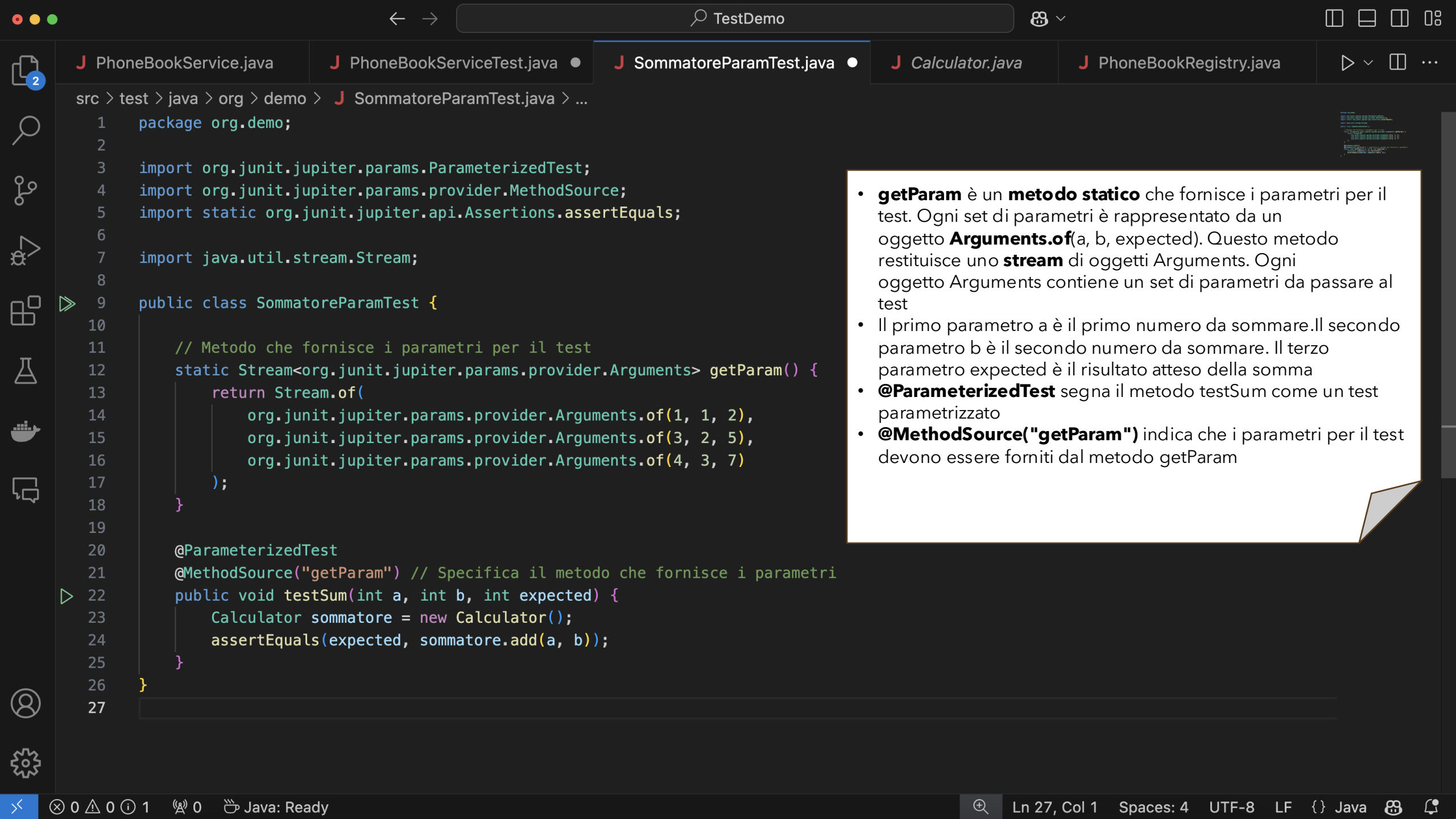
- I **Test Double** sono oggetti utilizzati nei test automatizzati per sostituire le dipendenze reali di un sistema. Questi oggetti simulano il comportamento delle dipendenze per isolare il codice da testare
- **Dummy**: Oggetti che vengono creati solo per soddisfare i requisiti del metodo o costruttore che si sta testando, ma non vengono mai realmente utilizzati nel test
- **Fake**: Oggetti che hanno un'implementazione funzionante, ma con semplificazioni che li rendono inadatti per la produzione
- **Stub**: Oggetti che restituiscono risposte preconfigurate a determinate chiamate. Non hanno una logica reale, ma rispondono solo secondo quanto programmato per il test
- **Spy**: Oggetti che combinano il comportamento di uno stub con la capacità di registrare informazioni su come vengono chiamati
- **Mock**: Oggetti configurati per aspettarsi determinate chiamate con determinati parametri. I mock verificano che il codice testato chiami i metodi giusti, con i parametri giusti, e il numero corretto di volte

State vs Behavior Verification

- Supponiamo di avere una classe da testare PhoneBookService che collabora con PhoneBookRegistry
- Il metodo da testare PhoneBookService.register() chiama i metodi di PhoneBookRegistry per aggiornarne lo stato (aggiunta nuovo contatto)
- Un approccio orientato alla **verifica dello stato** creerebbe un'**asserzione sull'output** del metodo chiamato o sul **cambiamento di stato dell'oggetto**
- In questo caso però il metodo da testare non ha un valore di ritorno e il cambiamento di stato avviene su un collaboratore (il registro)
- Un approccio orientato alla **verifica del comportamento**, invece, **verifica la correttezza della sequenza di chiamate** verso i metodi dei collaboratori (con l'uso di verify() sul mock)
- In questo caso, invece di verificare che il registro abbia un nuovo contatto, ci limitiamo a verificare che il metodo di inserimento sia stato invocato
- Il secondo approccio, reso possibile dai mock, ci permette di testare PhoneBookService indipendentemente da un'implementazione di PhoneBookRegistry


TestDemo
PhoneBookService.java
PhoneBookRegistry.java
src > main > java > org > demo > PhoneBookService.java > ...
1 package org.demo;
2
3 public class PhoneBookService {
4 PhoneBookRegistry registry; // ci si lega ad un'interfaccia
5 public PhoneBookService(PhoneBookRegistry registry) {
6 this.registry = registry;
7 }
8 public void register(String name, String number) {
9 if (! registry.contains(name)) registry.insert(name, number);
10 // solo se contains() da false riesco a testare questa linea
11 }
12 }
13
0 0 0 Java: Ready
Ln 13, Col 1 Spaces: 4 UTF-8 LF Java


TestDemo
PhoneBookService.java
PhoneBookServiceTest.java
ConsumerTest.java
PhoneBookRegistry.java
src > test > java > org > demo > PhoneBookServiceTest.java > ...
7 import org.junit.jupiter.api.extension.ExtendWith;
8 import org.mockito.junit.jupiter.MockitoExtension;
9
10
11 @ExtendWith(MockitoExtension.class)
12 public class PhoneBookServiceTest {
13
14 @Mock private PhoneBookRegistry registry;
15 PhoneBookService service;
16
17 @BeforeEach
18 public void setUp() {
19 service = new PhoneBookService(registry);
20 }
21
22 @Test
23 public void testRegister() {
24 String name = "Andrea", number = "123456";
25 when(registry.contains(name)).thenReturn(value:false); // Simula il comportamento del metodo contains
26 service.register(name, number); // Chiama il metodo da testare
27 verify(registry).insert(name, number); // Verifica che insert sia stato chiamato con i parametri corretti
28 }
29 }
30
0 0 0 Java: Ready
Ln 30, Col 1 Spaces: 4 UTF-8 LF Java


TestDemo
PhoneBookService.java
PhoneBookServiceTest.java
SommatoreParamTest.java
Calculator.java
PhoneBookRegistry.java
src > test > java > org > demo > SommatoreParamTest.java > ...
1 package org.demo;
2
3 import org.junit.jupiter.params.ParameterizedTest;
4 import org.junit.jupiter.params.provider.MethodSource;
5 import static org.junit.jupiter.api.Assertions.assertEquals;
6
7 import java.util.stream.Stream;
8
9 public class SommatoreParamTest {
10
11 // Metodo che fornisce i parametri per il test
12 static Stream<org.junit.jupiter.params.provider.Arguments> getParam() {
13 return Stream.of(
14 org.junit.jupiter.params.provider.Arguments.of(1, 1, 2),
15 org.junit.jupiter.params.provider.Arguments.of(3, 2, 5),
16 org.junit.jupiter.params.provider.Arguments.of(4, 3, 7)
17);
18 }
19
20 @ParameterizedTest
21 @MethodSource("getParam") // Specifica il metodo che fornisce i parametri
22 public void testSum(int a, int b, int expected) {
23 Calculator sommatore = new Calculator();
24 assertEquals(expected, sommatore.add(a, b));
25 }
26 }
27
• getParam è un metodo statico che fornisce i parametri per il test. Ogni set di parametri è rappresentato da un oggetto Arguments.of(a, b, expected). Questo metodo restituisce uno stream di oggetti Arguments. Ogni oggetto Arguments contiene un set di parametri da passare al test
• Il primo parametro a è il primo numero da sommare.Il secondo parametro b è il secondo numero da sommare. Il terzo parametro expected è il risultato atteso della somma
• @ParameterizedTest segna il metodo testSum come un test parametrizzato
• @MethodSource("getParam") indica che i parametri per il test devono essere forniti dal metodo getParam
0 0 1 0 Java: Ready
Ln 27, Col 1 Spaces: 4 UTF-8 LF Java

Università di Catania

Uni ct MATEMATICA E INFORMATICA

The End 

Alessandro Midolo, PhD

Dipartimento di Matematica e Informatica

Università di Catania

alessandro.midolo@unict.it

https://www.dmi.unict.it/amidolo/

A.A. 2024/2025