



# Modulo 4

# **Integration Tier**

## JUnit

Presented by

*Paolo Locorotondo*

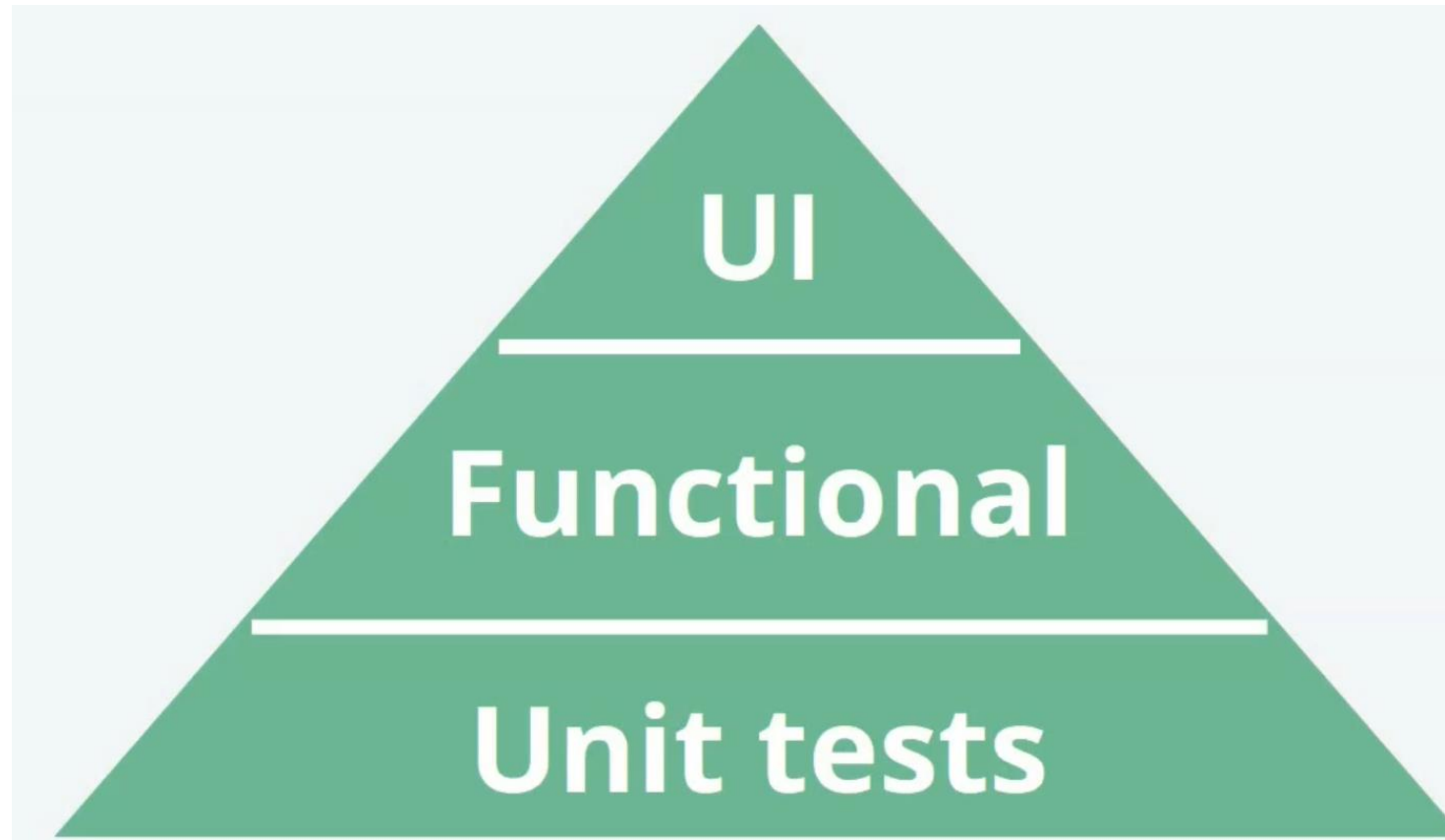
IBM Client Innovation Center - Italy

06/12/2023

IBM Client Innovation Center  
**Italy**

JUnit

# Introduzione



# Introduzione

## Unit Test

Attività di collaudo di singole unità di un software

Consente di testare il codice in modo isolato e di evitare che l'esecuzione del test porti a stati di inconsistenza che ne impediscano la ripetitività nel tempo

Può essere svolta:

- manualmente
- automaticamente

# Introduzione

JUnit è uno dei più popolari framework per unit-testing per il linguaggio di programmazione Java

Caratteristiche principali:

- Integrazione con gli IDE più diffusi (Eclipse, IntelliJ, VS Code)
- Ogni unit test è rappresentato da un metodo
- Vengono usate classi di test separate per avere esecuzioni indipendenti
- Sono disponibili diverse asserzioni per automatizzare il controllo dei risultati dei test

# Esempio

```
public class TestCalc { // classe di test
    @Test
    public void testAdd() { // l'annotazione indica il test
        Calculator c = new Calculator(); // crea istanza
        double result = c.add(10, 50); // chiama metodo da testare
        assertEquals(60, result, 0); // controlla il risultato e
    }                                // genera eccezione se result != 60
}
```

# Assertzioni

Metodi di utilità per supportare la verifica di condizioni durante l'esecuzione di test

Accessibili tramite:

- Assert in JUnit 4
- Assertions in JUnit 5

# Assertzioni

## **assertEquals**

@Test

```
void whenAssertingEquality_thenEqual() {  
    int expected = 2 * 2;  
    float actual = 2 * 2;  
    Assertions.assertEquals(expected, actual);  
}
```

## **assertArrayEquals**

@Test

```
public void whenAssertingArraysEquality_thenEqual() {  
    char[] expected = { 'J', 'U', 'n', 'i', 't' };  
    char[] actual = "JUnit".toCharArray();  
    Assertions.assertArrayEquals(expected, actual, "Arrays should be equal");  
}
```

# Assertzioni

## **assertTrue e assertFalse**

@Test

```
void whenAssertingConditions_thenVerified() {  
    Assertions.assertTrue(5 > 4, "5 is greater than 4");  
    Assertions.assertTrue(null == null, "null is equal to null");  
    Assertions.assertFalse(5 > 6, "5 is not greater than 6");  
}
```

## **assertNull e assertNotNull**

@Test

```
void whenAssertingConditions_thenVerified() {  
    Object obj = new Object();  
    Assertions.assertNotNull(obj);  
}
```



# Assertzioni

## **assertThrows**

Consente di verificare se un eseguibile genera il tipo di eccezione atteso

L'asserzione avrà esito negativo se non viene generata alcuna eccezione o se viene generata un'eccezione di tipo diverso

@Test

```
void whenAssertingException_thenThrown() {  
    Throwable exception = assertThrows(  
        IllegalArgumentException.class,  
        () -> {  
            throw new IllegalArgumentException("Exception message");  
        }  
    );  
    assertEquals("Exception message", exception.getMessage());  
}
```

# AssertJ

Libreria che può essere utilizzata insieme a JUnit per definire asserzioni tramite concatenazione di più metodi, migliorando la leggibilità del codice

```
@Test
void whenAssertingStrConditions_thenVerified() {
    String mytest = "MyAssertJTest";
    assertThat(mytest)
        .startsWith("MyAssertJ")
        .endsWith("Test")
        .isEqualToIgnoringCase("myassertjtest");
}
```

# Annotazioni

- *Test*: identifica un metodo associato ad un test case
- *RepeatedTest(n)*: come l'annotazione *Test*, ma ripete l'esecuzione del test case *n* volte
- *ParameterizedTest*: identifica un caso di test che necessita di alcuni parametri in input. Diverse annotazioni per fornire i parametri:
  - *ValueSource(ints = {1, 3, 5, -3, 15, Integer.MAX\_VALUE})*
  - *NullSource*
  - *EmptySource*
  - *NullAndEmptySource*
  - *EnumSource(Month.class)*
  - *CsvSource({"test,TEST", "tEst,TEST", "Java,JAVA"})*

# Annotazioni

Sono disponibili delle annotazioni che consentono di stabilire quando verrà eseguito il metodo su cui sono applicate:

- *BeforeAll*: prima di tutti i test della classe
- *BeforeEach*: prima di ogni metodo di test
- *AfterEach*: dopo ogni metodo di test
- *AfterAll*: al termine dell'esecuzione di tutti i test della classe

## JUnit Mock

L'isolamento dei test unitari può essere ottenuto utilizzando degli oggetti (mock) che simulano il comportamento degli oggetti reali

Il mock deve esporre la stessa interfaccia dell'oggetto che simula consentendo al client di ignorare se sta interagendo con l'oggetto reale o con quello simulato

# JUnit Mockito

Framework che semplifica la creazione dei mock

Costrutti principali:

- *mock()* o *@Mock*: consentono la definizione di un oggetto mock al quale è possibile associare un comportamento
- *spy()* o *@Spy*: consentono di realizzare un mock parziale dell'oggetto
- *verify()*: consente di verificare la corretta invocazione dei metodi durante l'esecuzione del test

# Mockito

## Creazione oggetto mock

```
Dao dao = Mockito.mock(Dao.class);
```

## Utilizzo dell'oggetto mockato












```
Service service = new Service(dao);
```

Modifica del comportamento del mock per il metodo `dao.existsPerson`: per qualsiasi intero fornito in input il metodo restituirà true

```
Mockito.when( dao.existsPerson( Mockito.anyInt() ) ).thenReturn(true);
```

# Code coverage

Metrica tramite cui ottenere la percentuale del codice coperta da una suite di test

Element	Coverage
▼ BugTrap	 71,5 %
▼ src	 71,5 %
▼ Base	 85,3 %
> BugReportTest.java	 92,7 %
> BugTrap.java	 80,7 %
> BugReport.java	 96,3 %
> Project.java	 69,2 %
> PerformanceTest.java	 93,4 %
> HealthTest.java	 97,2 %
> Performance.java	 99,4 %
> Subsystem.java	 74,6 %



# Vediamo un esempio

*(junit\_esempio)*

