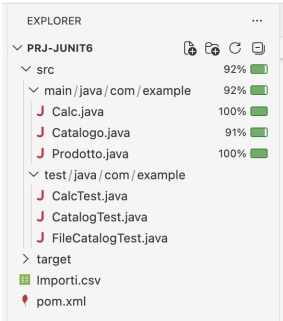
JUNIT

JUnit

- E' un framework per il test di applicazioni Java, usato per scrivere ed eseguire test su classi e metodi
- La versione JUnit 6 è la versione più recente
- JUnit fornisce un motore per l'esecuzione dei test e API per scrivere test
- Un'**annotazione** Java è un **metadato** associato a elementi di codice (classi, metodi, campi, etc.) che fornisce informazioni aggiuntive al compilatore
- JUnit usa le annotazioni Java per i metodi della classe di test
  - `@Test` identifica un metodo di test
  - `@BeforeEach` esegue il metodo prima di ogni test
  - `@BeforeAll` esegue il metodo una volta prima di tutti i test

JUnit: Asserzioni

- Per verificare il comportamento del codice (ovvero l'esecuzione) si usano i metodi della classe `Assertions`
  - `assertEquals(expected, actual)`
  - `assertTrue(condition)`
  - `assertNull(object)`
- Il codice è posizionato seguendo la **convenzione** standard di posizionamento delle cartelle di **Maven**
  - Le classi dell'applicazione sono dentro la cartella `src/main/java` e il codice di test su `src/test/java`
  - Quindi eventuali package (moduli) sono dentro la cartella java
  - Il file `pom.xml` descrive il progetto



JUnit: Esempio

```java
public class AppTest {
    private static final Catalogo catalog = new Catalogo();

    @BeforeAll
    public static void init() {
        catalog.svuota();
        catalog.inserisci("album,12.00,Rick,2025-01-15");
        catalog.inserisci("libro,10.50,Rick,2025-01-15");
        catalog.inserisci("matite,2.00,Rick,2025-01-15");
        catalog.inserisci("zaino,21.50,Rick,2025-01-15");
    }

    @Test
    public void sommaValori() {
        catalog.calcolaSomma();
        assertEquals(46.00f, catalog.getSomma());
    }

    @Test
    public void massimoValori() {
        catalog.calcolaMassimo();
        assertEquals(21.50f, catalog.getMassimo());
    }
}
```

JUnit: Esecuzione di Test Multipli

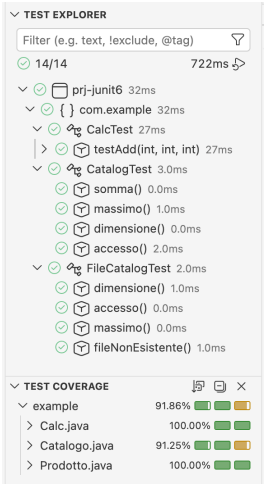
- L'annotazione `@ParameterizedTest` permette di eseguire un metodo più volte passando in input valori diversi, senza inserire i valori nel codice di test, un modo per fornire i valori è tramite `@CsvSource`

```
public class CalcTest {
    private final Calc calculator = new Calc();

    @ParameterizedTest
    @CsvSource({
            "1, 1, 2",
            "2, 3, 5",
            "10, 5, 15",
            "-1, 1, 0"
    })
    void testAdd(int a, int b, int expected) {
        int result = calculator.add(a, b);
        assertEquals(expected, result);
    }
}
```


Copertura del Codice

- Durante l'esecuzione di un test, si esegue una parte dell'applicazione
- La **copertura del codice** (code coverage) è il rapporto (espresso in percentuale) fra le linee di codice eseguite e le linee di codice totali
  - Per es., una copertura del 70% indica che il 70% delle linee di codice sono state eseguite dai test
  - Quando si raggiunge una copertura del 100% si hanno test che eseguono tutte le linee di codice
- Per calcolare la copertura del codice occorre tracciare le linee di codice eseguite
- Eseguendo JUnit da VSCode si può ottenere il valore di copertura del codice e il tracciamento delle linee

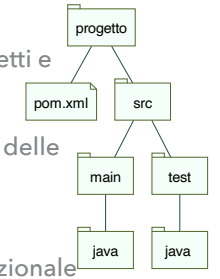



```
3x public class Catalogo {
5x     private List<Prodotto> prodotti = new ArrayList<>();
3x     private float totale = 0;
3x     private float massimo = 0;

    /**
     * Legge linea per linea il file, e riempie una lista
     *
     * @param c nome cartella
     * @param n nome file
     */
    public void leggiFile(String c, String n) {
16x         try (BufferedReader f = new BufferedReader(new FileReader(new File(c, n)))) {
2x             String riga = "";
1x             while ((riga = f.readLine()) != null)
3x                 inserisci(riga);
1x         } catch (IOException e) {
5x             System.err.println("Errore di I/O: " + e.getMessage());
        }
1x     }

    /** Inserisce in lista solo valori distinti */
    public void inserisci(String riga) {
4x         String[] campi = riga.split(regex: ",");
        if (campi.length != 4) {
            System.err.println("Riga malformata: " + riga);
            return; // Salta questa riga se non ha il formato corretto
        }
1x         if (campi[1].trim().equals(anObject: "valore"))
1x             return;
5x         String nome = campi[0].trim();
2x         float prezzo = 0;
        try {
6x             prezzo = Float.parseFloat(campi[1].trim());
1x         } catch (NumberFormatException e) {
```


Maven



- Apache Maven è uno strumento per la gestione di progetti e per automatizzare il build
- Consente di standardizzare la compilazione, la gestione delle dipendenze, il test, il packaging e il rilascio, attraverso convenzioni e configurazioni
- In Maven, un progetto ha una struttura standard convenzionale
  - La posizione dei file sorgente è predeterminata
- Il file POM (Project Object Model) descrive il progetto: nome progetto e versione, dipendenze, plugin, configurazioni
- I tag xml predefiniti racchiudono le informazioni (es. dipendenze con le librerie, come suggerito da chi le fornisce)

Maven: File POM

```xml
<project>
    <groupId>com.example</groupId>
    <artifactId>prj-junit6</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>25</maven.compiler.source>
        <maven.compiler.target>25</maven.compiler.target>
        <maven.compiler.release>25</maven.compiler.release>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
            <version>6.0.3</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>6.0.3</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```


Uso di Maven

- Si prepara un progetto Maven con l'IDE oppure con `mvn archetipe:generate`
- Si cerca la libreria che vuol usare nel repository Maven Central, e si aggiunge la dependency nel `pom.xml`, quindi maven **scarica** automaticamente la libreria e le ulteriori dipendenze necessarie
- Il codice sarà inserito su **src/main/java** e il codice di test su **src/test/java**
- Comandi e fasi (l'esecuzione di una fase esegue tutte le fasi precedenti)
  - `mvn compile` per compilare il codice
  - `mvn test` per eseguire i test di unità
  - `mvn package` per produrre un .jar distribuibile
  - `mvn verify` per i controlli di qualità
- Sul file `pom.xml` si possono inserire plugin di maven per il test, il packaging, il deploy