

## Sumar

1. Configurarea JUnit 5.x pentru proiectul Maven în IntelliJ IDEA.....	2
2. Crearea unei clase de testare folosind JUnit.....	2
3. Scrierea unui test folosind JUnit.....	3
4. Execuția testelor folosind JUnit.....	4
5. Localizarea bug-urilor. <i>Test Case</i> vs. <i>Tested Method</i> .....	5
6. <i>JUnit 4</i> vs. <i>JUnit 5</i> .....	5

## Lista de Figuri

Figure 1. Dependențele incluse în fișierul <i>pom.xml</i> , necesare utilizării JUnit 5.x în cadrul unui proiect Maven.....	2
Figure 2. Crearea unei clase pentru testarea entității Task .....	3
Figure 3. Alegerea framework-ului de testare și configurarea clasei de testare .....	3
Figure 4. Afișarea rezultatului execuției testelor .....	4

## Lista de Tabele

Table 1. JUnit 4 vs. JUnit 5.....	5
-----------------------------------	---

## Lista de Snippets

Snippet 1 Dependența inclusă în fișierul <i>pom.xml</i> pentru utilizarea JUnit 5.x într-un proiect Maven.....	2
--	---

**Tutorialul pentru utilizarea platformei de testare JUnit 5 in IntelliJ IDEA poate conține anumiți pași care pot fi omiși.**

## 1. Configurarea JUnit 5.x pentru proiectul Maven în IntelliJ IDEA

În proiectul Maven, în fișierul **pom.xml** se verifică includerea dependenței pentru **junit-jupiter-engine** pentru **JUnit 5.x** (vezi Figure 1 și Snippet 1).

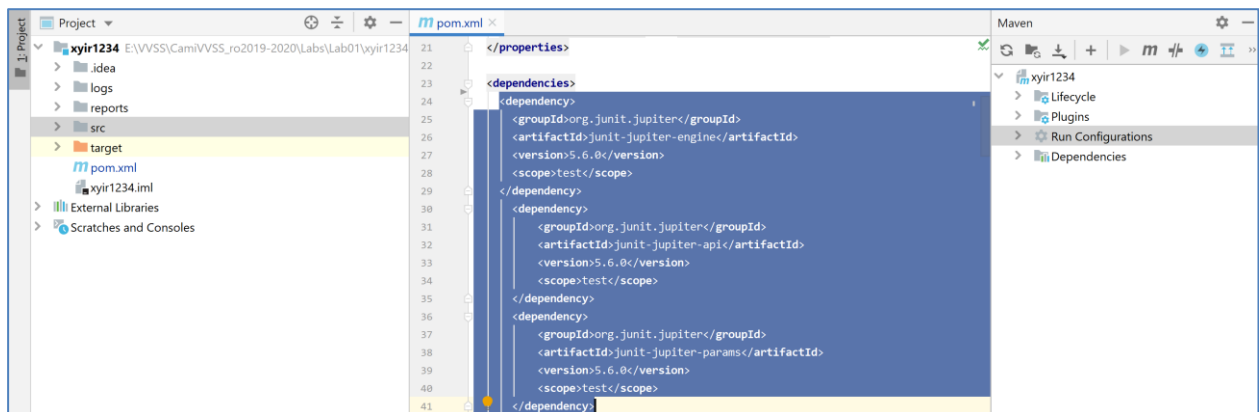


Figure 1. Dependențele incluse în fișierul **pom.xml**, necesare utilizării JUnit 5.x în cadrul unui proiect Maven

```

<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>5.6.0</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <version>5.6.0</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-params</artifactId>
  <version>5.6.0</version>
  <scope>test</scope>
</dependency>
  
```

Snippet 1 Dependența inclusă în fișierul **pom.xml** pentru utilizarea JUnit 5.x într-un proiect Maven

## 2. Crearea unei clase de testare folosind JUnit

1. în **IntelliJ IDEA**, se poziționează cursorul pe numele clasei pentru care se creează testul;
2. se folosește combinația de taste **Alt+Enter**;
3. din meniul pop-up care apare se selectează opțiunea **Create Test** (vezi Figure 2);
4. se va deschide o fereastră care permite (vezi Figure 3):
  - alegerea platformei de testare:
    - în cadrul activităților de laborator se va utiliza **JUnit 5.x**;
    - Table 1 prezintă o parte dintre diferențele de utilizare la nivelul adnotărilor între **JUnit 4.x** și **JUnit 5.x**;
  - stabilirea numelui clasei care va conține teste (**Class name**): **TaskTest**;
  - (opțional) se poate bifa utilizarea metodelor:
    - **setUp()** – pentru inițializarea stării *înainte* de execuția fiecărui test;

- `tearDown()` – pentru finalizarea testului (e.g., revenirea la starea anterioară execuției testului), apelată *după* execuția fiecărui test;
- **cele două metode se apelează implicit înainte și după fiecare test rulat;**
- (opțional) se pot alege metodele din clasă care vor fi testate și pentru care se vor crea teste stub (i.e., metode care vor fi descrise ulterior);
- la utilizarea **JUnit 4.x/5.x**, testele *stub* generate vor fi adnotate cu **@Test**;
- click **OK** pentru crearea clasei cu teste.

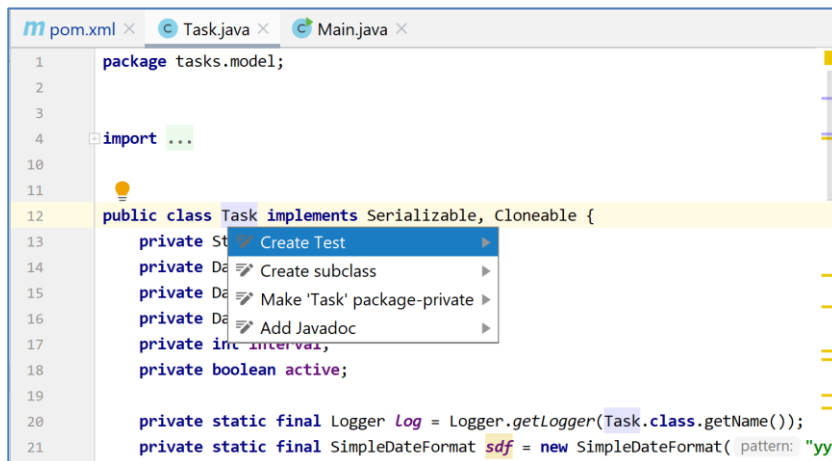


Figure 2. Crearea unei clase pentru testarea entității Task

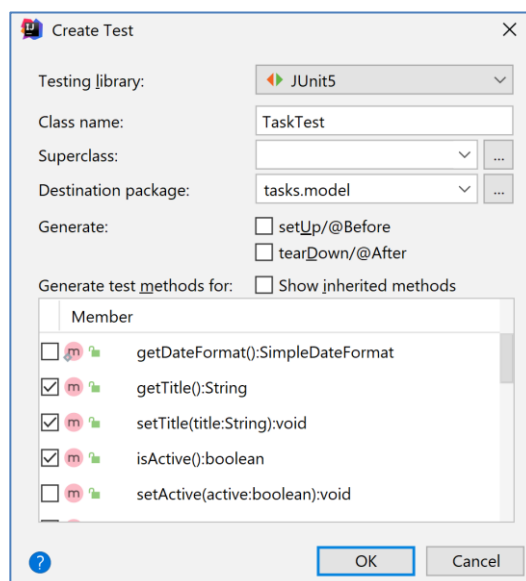


Figure 3. Alegerea framework-ului de testare și configurarea clasei de testare

### 3. Scrierea unui test folosind JUnit

**Exemplu 1:** testarea metodei `getTitle()`:

- se declară referințe private ale tipurilor utilizate în cadrul testului, i.e., `Task`, `SimpleDateFormat`, `Date` în clasa `TaskTest`:

```
private Task task;
private Date start, end;
private SimpleDateFormat sdf;
```

- referințele se inițializează în metoda `setUp()`:

```
sdf= Task.getDateFormat();
try {
    start=sdf.parse("2020-02-30 12:00");
    end=sdf.parse("2020-02-03 10:00");
} catch (ParseException e) {
    fail(e.getMessage());
}
task = new Task("seminar", start, end, 1);
```

- **înainte de execuția fiecărui test** se execută metoda `setUp()` care va instanția obiectele `sdf`, `start`, `end` și `task`;
- în testul `getTitle()` se adaugă codul de testare pentru metoda `getTitle()` din clasa `Task`:

```
assertEquals("seminar", task.getTitle(), "Task title name should be 'seminar'");
```

- dacă în cadrul unui test o construcție `assert` eșuează, execuția testului se încheie imediat, iar JUnit setează statusul acestuia ca **failed**;

#### Exemplu 2: testarea constructorului:

- în testul `createTask()` se adaugă codul de testare pentru testarea construirii unui obiect de tip `Task`:

```
@Test
public void createTask() {
    Task task1 = new Task("lab", start, end, 1);

    //assertNotEquals(task1, null);
    assert task1 != null;
}
```

## 4. Execuția testelor folosind JUnit

1. în **IntelliJ IDEA**, având clasa `TaskTest` ca și clasă curentă:
  - din meniul **Run** ---> **Run 'TaskTest'** sau
  - click dreapta pe numele clasei `TaskTest` în **Project Explorer** și se alege opțiunea **Run 'TaskTest'**;
2. *Java Perspective* se modifică prin apariția tab-ului **Run** sub **Project Explorer** (vezi Figure 4);

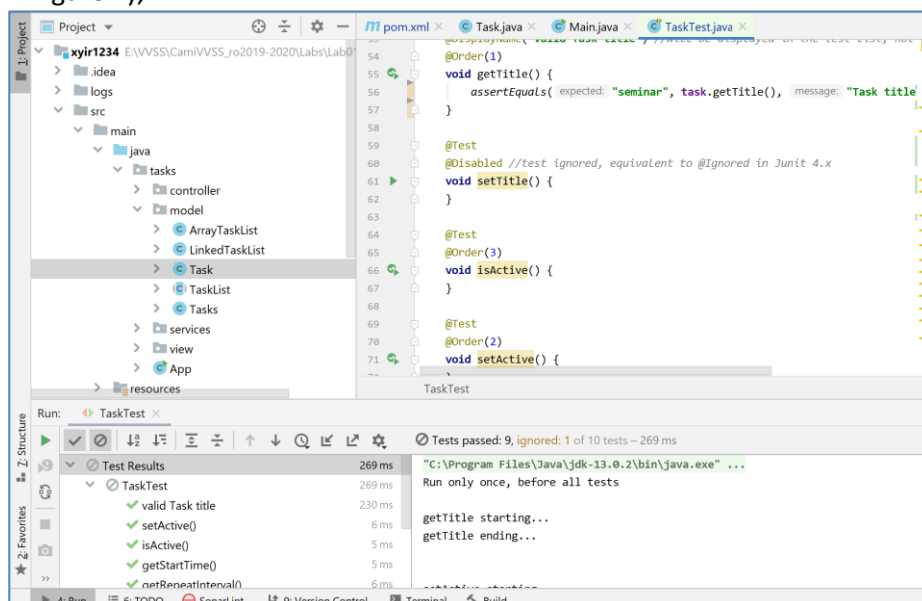


Figure 4. Afișarea rezultatului execuției testelor

- la selectarea unui test din lista de teste executate, în frame-ul din partea dreaptă sunt oferite detalii cu privire la execuție (i.e., valoare așteptată, valoare obținută, excepții aruncate, etc.).

## 5. Localizarea bug-urilor. *Test Case vs. Tested Method*

- execuția eșuată a unui test este determinată de bug-uri care pot apărea în:
  - codul sursă testat (e.g., `getTitle()`) – **[tested\_method#bug]**;
  - testul propriu-zis (e.g., `testSetTitle()`) – **[test\_case#bug]**;
- se verifică dacă datele de intrare din etapa de proiectare a testului au fost preluate corect în implementare;
- după verificare și re-execuția testului:
  - success** ---> pentru datele de intrare furnizate, metoda obține rezultatele așteptate **[test\_case#bug: fixed]**;
  - failed** ---> există un bug în metoda testată **[tested\_method#bug: needs debugging]**.

## 6. JUnit 4 vs. JUnit 5

Îmbunătățirile disponibile în **JUnit 5** fac referire la diferite aspecte:

- modalitatea de execuție a testelor prin folosirea unor adnotări care stabilesc ordinea de execuție a testelor (e.g., ordine alfabetică, pre-stabilită, aleator), folosind adnotări ca: `@TestMethodOrder(<tip de ordonare>.class)`, `@Order(<număr>)`.
- introducerea de adnotări noi sau adaptarea celor existente, ca în Table 1.

JUnit 4	JUnit 5
<code>@BeforeClass</code>	<code>@BeforeAll</code>
<code>@AfterClass</code>	<code>@AfterAll</code>
<code>@Before</code>	<code>@BeforeEach</code>
<code>@After</code>	<code>@AfterEach</code>
<code>@Ignore</code>	<code>@Disabled</code>
<code>@Test &lt;cu diverși parametri&gt;</code>  e.g.: <code>@Test (timeout = &lt;&lt;value&gt;&gt;)</code>  <code>@Test (expected = &lt;&lt;ClassException&gt;&gt;.class)</code>	<code>@Test [fără parametri]</code> Pentru simularea aceluiași comportament din <b>JUnit 4</b> se introduc adnotări sau metode <code>assert</code> particulare  e.g.: <code>@Timeout</code> <code>assertTimeout(ofMinutes(2), () -&gt; { /*...*/ });</code>  <code>assertThrows(&lt;&lt;ClassException&gt;&gt;.class, () -&gt; { /*...*/ });</code>
<code>@Category</code>	<code>@Tag</code>

Table 1. JUnit 4 vs. JUnit 5

- descrierea facilă a testelor parametrizate prin utilizarea unor adnotări specifice (e.g.: `@ParameterizedTest`, `@ValueSource`) și includerea dependenței către `junit-jupiter-params` (similar cu includerea dependenței pentru `junit-jupiter-engine`).
- execuția repetată a testelor prin utilizarea adnotării `@RepeatedTest` cu diferite semnături;
- alte facilități documentate în <https://junit.org/junit5/docs/current/user-guide/#writing-tests>.