

1. A legfontosabb rész:

```
@Test
```

Ez mondja meg a JUnitnak, hogy ez egy tesztmetódus. Az assertion: `assertEquals(5, result);`

azt jelenti: elvárt érték = 5 tényleges érték = result

2. A legfontosabb assertionök Ezeket fogod a legtöbbször használni:

```
assertEquals(expected, actual);
assertNotEquals(expected, actual);

assertTrue(condition);
assertFalse(condition);

assertNull(value);
assertNotNull(value);
```

Például: `@Test void simpleAssertions() {`

```
    assertEquals(4, 2 + 2);
```

```
    assertTrue(10 > 5);
```

```
    assertFalse(3 > 10);
```

```
    String name = "Oliver";
    assertNotNull(name);
```

```
}
```

Nagyon fontos szokás: `assertEquals(expected, actual);`

tehát először az elvárt, utána a kapott érték. 3. Több teszt ugyanarra az osztályra Például bővítsük a Calculator osztályt: `public class Calculator {`

```
    public int add(int a, int b) {
        return a + b;
    }
```

```
    public int subtract(int a, int b) {
        return a - b;
    }
```

```
    public int multiply(int a, int b) {
        return a * b;
    }
```

```
}
```

A teszt: class CalculatorTest {

```
@Test
void additionWorks() {
    Calculator calculator = new Calculator();

    assertEquals(5, calculator.add(2, 3));
}

@Test
void subtractionWorks() {
    Calculator calculator = new Calculator();

    assertEquals(7, calculator.subtract(10, 3));
}

@Test
void multiplicationWorks() {
    Calculator calculator = new Calculator();

    assertEquals(20, calculator.multiply(4, 5));
}
}
```

Mavenből: mvn test

VS Code-ban pedig az egyes @Test metódusok mellett is megjelenhet a Run Test gomb; a Java Test Runner JUnit 5-öt is támogatja. Visual Studio Code 4. @BeforeEach Gyakran ugyanazt az objektumot kell minden teszthez létrehoznunk. Ez: @Test void additionWorks() {

```
    Calculator calculator = new Calculator();

    assertEquals(5, calculator.add(2, 3));
}

@Test void subtractionWorks() {

    Calculator calculator = new Calculator();

    assertEquals(7, calculator.subtract(10, 3));
}
```

ismétli ezt: Calculator calculator = new Calculator();

Erre való a: @BeforeEach

Például: import org.junit.jupiter.api.BeforeEach; import org.junit.jupiter.api.Test;

```
import static org.junit.jupiter.api.Assertions.assertEquals;
```

```
class CalculatorTest {
```

```
    private Calculator calculator;
```

```
    @BeforeEach
    void setUp() {
        calculator = new Calculator();
    }
```

```
    @Test
    void additionWorks() {
        assertEquals(5, calculator.add(2, 3));
    }
```

```
    @Test
    void subtractionWorks() {
        assertEquals(7, calculator.subtract(10, 3));
    }
```

```
}
```

A setUp() minden egyes teszt előtt lefut. Tehát: @BeforeEach

↓

@Test #1

@BeforeEach

↓

@Test #2

@BeforeEach

↓

@Test #3

Ez nagyon fontos unit testing minta. 5. @BeforeAll és @AfterAll Van olyan inicializáció, amit csak egyszer akarunk lefuttatni. @BeforeAll static void beforeAllTests() {

```
    System.out.println("Tests starting...");
```

```
}
```

és a végén: @AfterAll static void afterAllTests() {

```
    System.out.println("Tests finished.");
```

```
}
```

Teljes példa: `import org.junit.jupiter.api.AfterAll; import org.junit.jupiter.api.BeforeAll; import org.junit.jupiter.api.Test;`

```
class ExampleTest {
```

```
    @BeforeAll
    static void beforeAll() {
        System.out.println("START");
    }
```

```
    @Test
    void testOne() {
        System.out.println("testOne");
    }
```

```
    @Test
    void testTwo() {
        System.out.println("testTwo");
    }
```

```
    @AfterAll
    static void afterAll() {
        System.out.println("END");
    }
```

```
}
```

6. Kivételek tesztelése: `assertThrows` Ez különösen fontos. Tegyük fel, hogy van osztás: `public int divide(int a, int b) {`

```
    if (b == 0) {
        throw new IllegalArgumentException("Division by zero");
    }
```

```
    return a / b;
```

```
}
```

Azt szeretnénk tesztelni, hogy nullával osztás esetén valóban kivételt kapunk: `@Test void divisionByZeroThrowsException() {`

```
    Calculator calculator = new Calculator();
```

```
    assertThrows(
        IllegalArgumentException.class,
        () -> calculator.divide(10, 0)
    );
```

```
}
```

Ez nagyon tipikus JUnit 5 kód. Még jobb: `@Test void divisionByZeroThrowsException() {`

```
IllegalArgumentExcepTion exception =
    assertThrows(
        IllegalArgumentExcepTion.class,
        () -> calculator.divide(10, 0)
    );
```

```
assertEquals(
    "Division by zero",
    exception.getMessage()
);
```

```
}
```

Itt már nemcsak azt ellenőrizzuk, hogy volt exception, hanem az üzenetét is. 7. `assertAll` Több dolgot egyszerre is ellenőrizhetsz: `@Test void severalAssertions() {`

```
int result = 10;
```

```
assertAll(
    () -> assertTrue(result > 0),
    () -> assertTrue(result < 100),
    () -> assertEquals(10, result)
);
```

```
}
```

Előnye, hogy a JUnit megpróbálja az összes assertiont kiértékelni. 8. `@DisplayName` Olvasható nevet adhatsz a tesztnek: `import org.junit.jupiter.api.DisplayName; import org.junit.jupiter.api.Test;`

```
class CalculatorTest {
```

```
@Test
@DisplayName("2 + 3 eredménye 5")
void additionWorks() {
```

```
    Calculator calculator = new Calculator();
```

```
    assertEquals(5, calculator.add(2, 3));
}
```

```
}
```

A VS Code Testing nézetében így sokkal érthetőbb lehet a teszt. 9. `@Disabled` Ideiglenesen kikapcsolhatsz egy tesztet: `import org.junit.jupiter.api.Disabled;`

```
@Test @Disabled void unfinishedTest() {
```

```
    // még nincs kész
```

```
}
```

Vagy indoklással: `@Disabled("A funkció még nincs implementálva")`

10. Paraméterezett tesztek Ez a JUnit 5 egyik nagyon hasznos funkciója. Tegyük fel, hogy ezt akarjuk tesztelni: `calculator.add(...)`

Nem akarunk öt külön tesztet írni: `assertEquals(3, calculator.add(1, 2)); assertEquals(7, calculator.add(3, 4)); assertEquals(0, calculator.add(-1, 1));`

Használhatunk `@ParameterizedTest`-et: `import org.junit.jupiter.params.ParameterizedTest; import org.junit.jupiter.params.provider.CsvSource;`

```
import static org.junit.jupiter.api.Assertions.assertEquals;
```

```
class CalculatorTest {
```

```
    private final Calculator calculator = new Calculator();
```

```
    @ParameterizedTest
    @CsvSource({
        "1, 2, 3",
        "3, 4, 7",
        "-1, 1, 0",
        "10, 20, 30"
    })
    void additionWorks(
        int a,
        int b,
        int expected) {
```

```
        assertEquals(
            expected,
            calculator.add(a, b)
        );
    }
```

```
}
```

Ez gyakorlatilag négyszer futtatja ugyanazt a tesztet különböző adatokkal. 11. `@ValueSource` Egyszerű paraméterlistához: `@ParameterizedTest @ValueSource(ints = {1, 2, 3, 10, 100}) void positiveNumbersArePositive(int number) {`

```
    assertTrue(number > 0);
```

```
}
```

12. Jó teszt felépítése: Arrange - Act - Assert Ezt érdemes megszokni. `@Test void additionWorks() {`

```
    // Arrange
    Calculator calculator = new Calculator();
```

```
// Act  
int result = calculator.add(2, 3);
```

```
// Assert  
assertEquals(5, result);
```

```
}
```

A három rész: Arrange ↓ előkészítjük a tesztet

Act ↓ meghívjuk a tesztelendő kódot

Assert ↓ ellenőrizzük az eredményt

Ez az egyik legfontosabb unit testing gondolkodásmód.

From:

<https://edu.iit.uni-miskolc.hu/> - **Institute of Information Science - University of Miskolc**

Permanent link:

<https://edu.iit.uni-miskolc.hu/tanszek:oktatas:junit5?rev=1790849039>

Last update: **2026/10/01 10:03**

