

Esercitazioni Ing.Sw

BARESI LUCIANO / Gian Enrico Conti

Feb/Mar 2021

Test e jUnit

Breve introduzione

Test

- Si fanno esperimenti con il programma allo scopo di scoprire eventuali errori
 - Si campionano i comportamenti
 - Fornisce indicazioni parziali relative al particolare esperimento
 - Il programma è effettivamente provato solo per quei dati
- Il test è una tecnica dinamica rispetto alle verifiche statiche fatte dal compilatore

Test black-box e white-box testing

- Black-box o *funzionale*
 - Casi di test determinati in base a ciò che il componente deve fare
 - La sua *specifica*
- White-box o *strutturale*
 - Casi di test determinati in base a come il componente è implementato
 - Il suo *codice*

Dijkstra (1972)

Program testing can be used to show the presence of bugs, but never to show their absence



“Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead.”

Martin Fowler

JUnit (1)

- JUnit è un framework per scrivere test
 - Proposto da Erich Gamma (Design Patterns) e Kent Beck (Extreme Programming)
- JUnit usa la riflessività di Java per
 - Integrare codice e test
 - Eseguire test e test suite

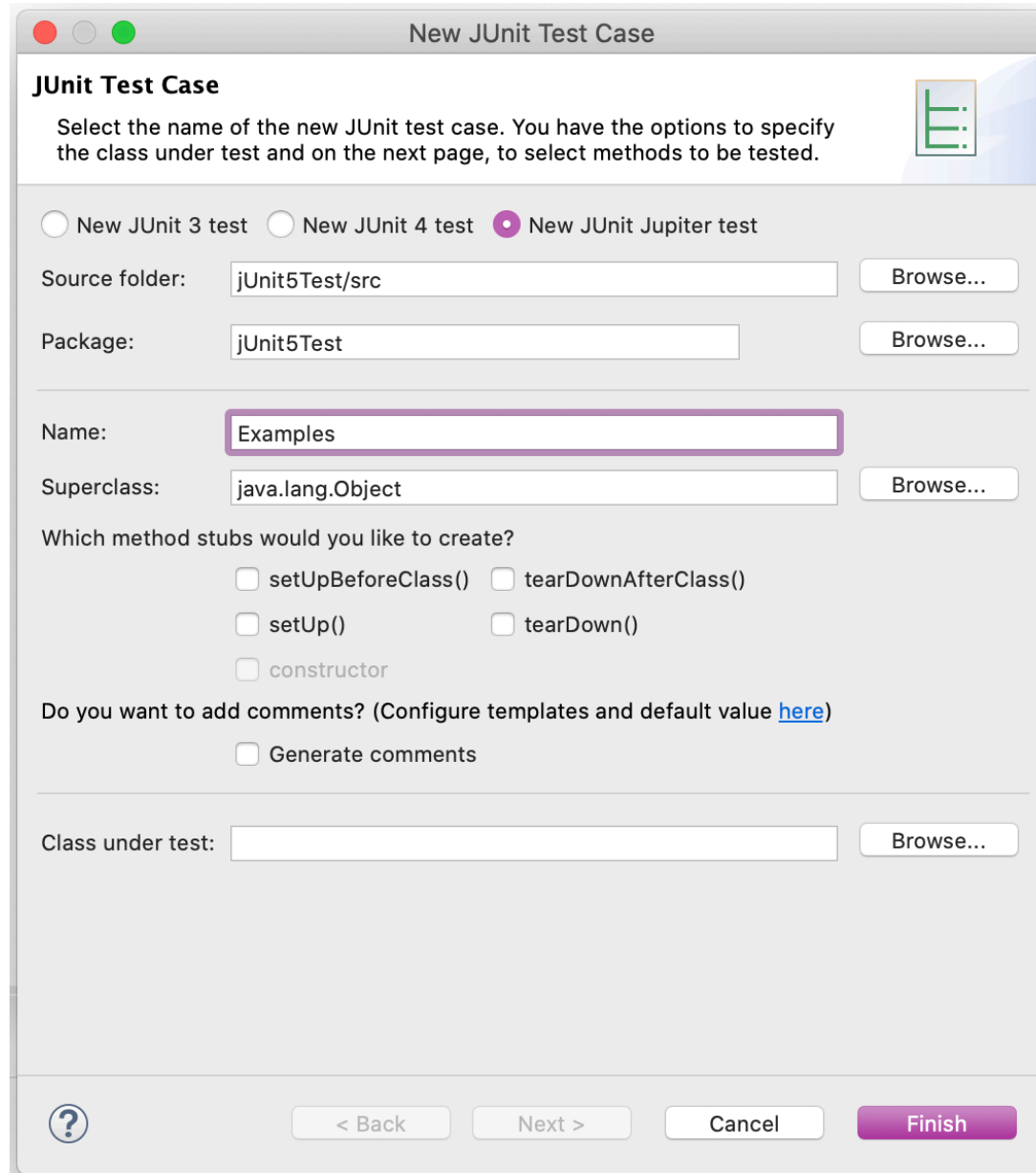
JUnit (2)

- JUnit non è parte di nessun SDK, ma tutti i principali IDE lo includono
- JUnit 5 = Platform + Jupiter + Vintage
- Estendibile con framework esterni
 - Ad esempio, AssertJ

Idea di base

- Data una classe Foo, creare un'altra classe FooTest per eseguire il test della precedente attraverso opportuni metodi
 - Ogni metodo è un test
- JUnit mette a disposizione metodi assert per la scrittura dei test
 - Questi metodi vanno chiamati nei metodi test per controllare quanto di interesse (oracolo)

JUnit e Eclipse



The screenshot shows the 'New JUnit Test Case' dialog box in Eclipse. The title bar reads 'New JUnit Test Case'. Inside, the 'JUnit Test Case' section has a description: 'Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.' Below this, three radio buttons are present: 'New JUnit 3 test', 'New JUnit 4 test', and 'New JUnit Jupiter test', with the latter being selected. The 'Source folder' is 'jUnit5Test/src' and the 'Package' is 'jUnit5Test', both with 'Browse...' buttons. The 'Name' field is 'Examples' and the 'Superclass' is 'java.lang.Object', also with a 'Browse...' button. A section titled 'Which method stubs would you like to create?' contains checkboxes for 'setUpBeforeClass()', 'tearDownAfterClass()', 'setUp()', 'tearDown()', and 'constructor'. Below this, a checkbox for 'Generate comments' is shown next to the text 'Do you want to add comments? (Configure templates and default value [here](#))'. At the bottom, the 'Class under test' field is empty with a 'Browse...' button. The footer contains a help icon, '< Back', 'Next >', 'Cancel', and a highlighted 'Finish' button.

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

☐ New JUnit 3 test ☐ New JUnit 4 test ☒ New JUnit Jupiter test

Source folder:

Package:

Name:

Superclass:


Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()
☐ setUp() ☐ tearDown()
☐ constructor

Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

Class under test:

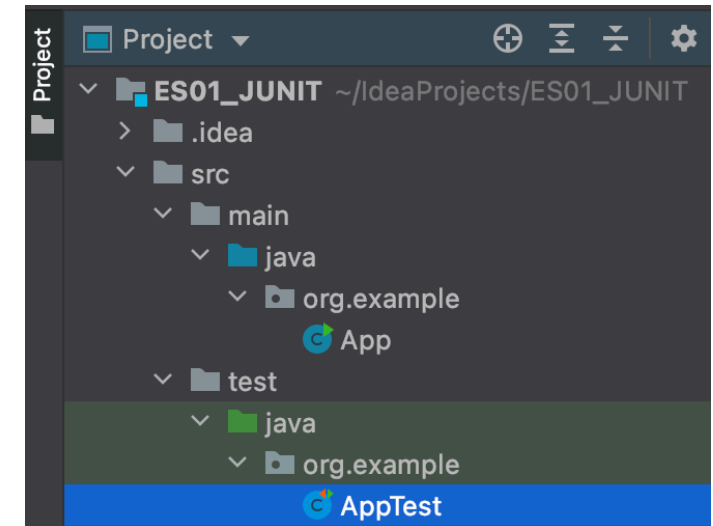




JUnit e IntelliJ

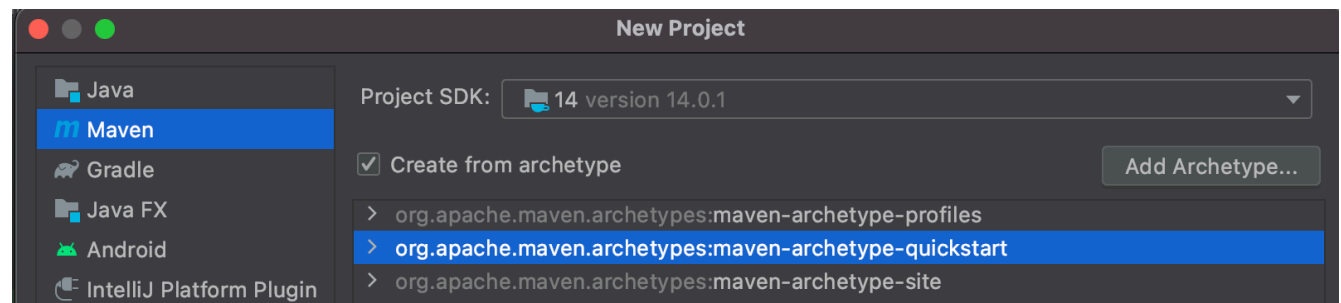
(<https://junit.org/junit5/>)

Solito progetto ...



Settare come cartella x tests..

(usualmente già settata da Maven quickstart)

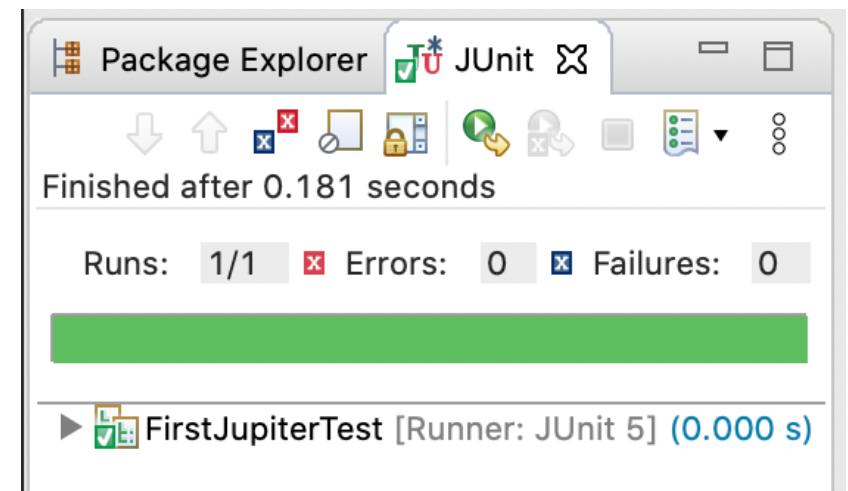


Un primo test

```
package junit5Test;
```

```
import static org.junit.jupiter.api.Assertions.*;  
import org.junit.jupiter.api.Test;
```

```
class FirstJupiterTest {  
    @Test  
    void theAnswer() {  
        assertEquals(42, 22 + 20);  
    }  
}
```





Un primo test

```
public class AppTest
{
    /**
     * Rigorous Test :-)
     */
    @Test
    public void shouldAnswerWithTrue()
    {
        assertTrue( true );
    }

    class FirstJupiterTest {
        @Test
        void theAnswer() {
            assertEquals(42, 22 + 20);
        }
    }
}
```

Errors? ...



IJ Test

- Se appare:

```
class FirstJupiterTest {  
    @Test  
    void theAnswer() {  
        assertEquals(42, 22 + 20);  
    }  
}
```

Cannot resolve method 'assertEquals' in 'FirstJupiterTest' ⋮
Create method 'assertEquals' ↵ ↵ More actions... ↵ ↵

Alt Invio...

Mette quello che non avete messo a mano...

```
import org.junit.Test; // Junit 4.
```

```
import static org.junit.jupiter.api.Assertions.*; // Junit 5
```

```
import org.junit.jupiter.api.Test; .....
```

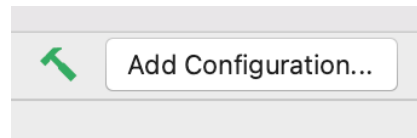
```
...import org.junit.jupiter.api.BeforeEach;
```



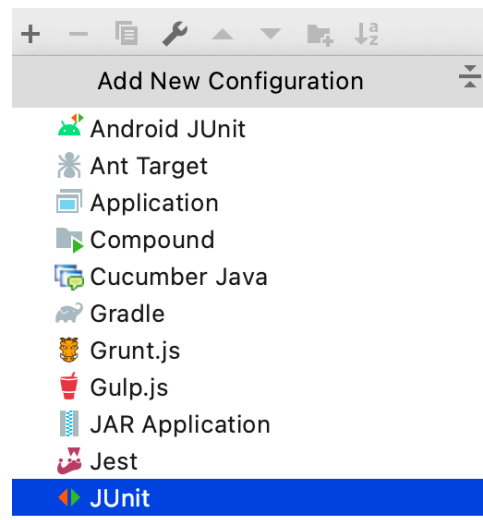
Lavoriamo "comodi": Configuration

IJ Test

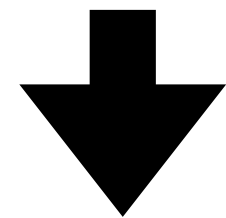
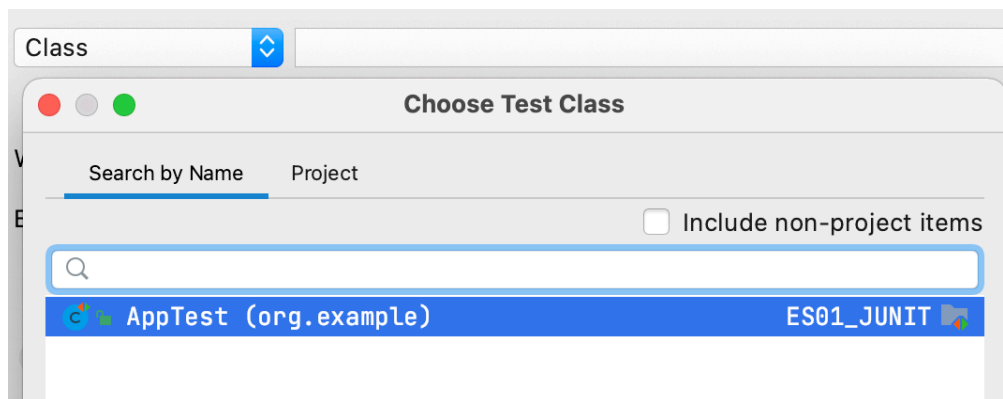
1) Add



2) Junit



3) Set class



@Test

- @Test indica un metodo test
- Questi metodi
 - Non possono essere **private** o **static**
 - Possono dichiarare parametri
- Jupiter consente le meta-annotazioni
- Assertion e Assumption
- Junit5 non supporta le test suite come Junit4

Alcune asserzioni



| | |
|--|--|
| <code>assertTrue(test)</code> | Fallisce se il test booleano è false |
| <code>assertFalse(test)</code> | Fallisce se il test booleano è true |
| <code>assertEquals(expected, actual)</code> | Fallisce se i due valori non sono uguali |
| <code>assertSame(expected, actual)</code> | Fallisce se i due oggetti non sono identici (==) |
| <code>assertNotSame(expected, actual)</code> | Fallisce se i due oggetti sono identici (==) |
| <code>assertNull(value)</code> | Fallisce se il valore non è null |
| <code>assertNotNull(value)</code> | Fallisce se il valore è null |
| <code>fail()</code> | Il test fallisce immediatamente |




- Ogni metodo accetta anche un parametro messaggio:
 - Esempio: `assertEquals(atteso, effettivo, messaggio)`

Proviamo ArrayList

```
class TestArrayIntList {  
    @Test  
    public void testAddGet1() {  
        ArrayList<Integer> list = new ArrayList<Integer>();  
        list.add(42); list.add(-3); list.add(15);  
  
        assertEquals(Integer.valueOf(42), list.get(0));  
        assertEquals(Integer.valueOf(-3), list.get(1));  
        assertEquals(Integer.valueOf(15), list.get(2));  
    }  
}
```

```
    @Test  
    public void testIsEmpty() {  
        ArrayList<Integer> list = new ArrayList<Integer>();  
        assertTrue(list.isEmpty());  
        list.add(123);  
        assertFalse(list.isEmpty());  
        list.remove(0);  
        assertTrue(list.isEmpty());  
    }  
}
```

Runs: 2/2  Errors: 0  Failures: 0

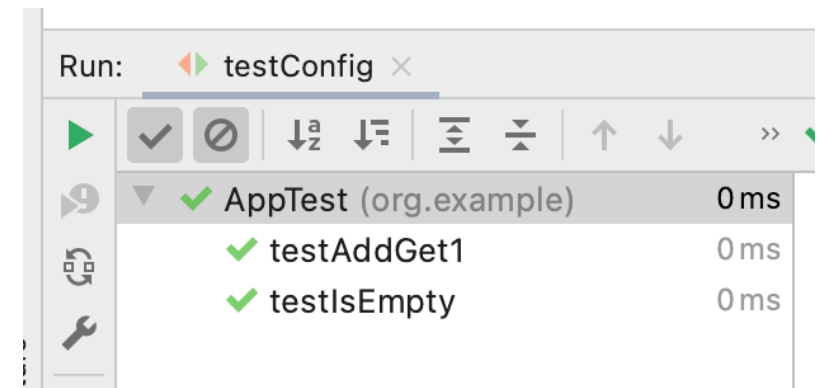
▼  TestArrayIntList [Runner: JUnit 5] (0.178 s)
  testAddGet1() (0.149 s)
  testIsEmpty() (0.029 s)



Proviamo ArrayList

```
class TestArrayIntList {  
    @Test  
    public void testAddGet1() {  
        ArrayList<Integer> list = new ArrayList<Integer>();  
        list.add(42); list.add(-3); list.add(15);  
  
        assertEquals(Integer.valueOf(42), list.get(0));  
        assertEquals(Integer.valueOf(-3), list.get(1));  
        assertEquals(Integer.valueOf(15), list.get(2));  
    }  
}
```

```
    @Test  
    public void testIsEmpty() {  
        ArrayList<Integer> list = new ArrayList<Integer>();  
        assertTrue(list.isEmpty());  
        list.add(123);  
        assertFalse(list.isEmpty());  
        list.remove(0);  
        assertTrue(list.isEmpty());  
    }  
}
```




Le nostre amate Date

```
class DateTest {  
    @Test  
    public void test1() {  
        Date d = new Date(2050, 2, 15);  
        d.addDays(4);  
        assertEquals(2050, d.getYear());  
        assertEquals(2, d.getMonth());  
        assertEquals(19, d.getDay());  
    }  
}
```

```
    @Test  
    public void test2() {  
        Date d = new Date(2050, 2, 15);  
        d.addDays(13);  
        assertEquals(2050, d.getYear(), "year after 13 days");  
        assertEquals(2, d.getMonth(), "month after 13 days");  
        assertEquals(28, d.getDay(), "day after 13 days");  
    }  
}
```

Finished after 0.127 seconds

Runs: 2/2  Errors: 0  Failures: 0

 DataTest [Runner: JUnit 5] (0.024 s)

Altro esempio

```
class DateTest {  
    @Test  
    public void test1() {  
        Date d = new Date(2050, 2, 15);  
        d.addDays(4);  
        assertEquals(2050, d.getYear());  
        assertEquals(2, d.getMonth());  
        assertEquals(19, d.getDay());  
    }  
  
    @Test  
    public void test2() {  
        Date d = new Date(2050, 2, 15);  
        d.addDays(14);  
        assertEquals(2050, d.getYear(), "year after 14 days");  
        assertEquals(3, d.getMonth(), "month after 14 days");  
        assertEquals(1, d.getDay(), "day after 14 days");  
    }  
}
```

Finished after 0.144 seconds

Runs: 2/2  Errors: 0  Failures: 1

▼  DataTest [Runner: JUnit 5] (0.023 s)
  test1() (0.014 s)
  test2() (0.009 s)

Oggetti attesi

- Dobbiamo implementare toString() e equals()

```
public class DataTest {  
    @Test  
    public void test1() {  
        Date d = new Date(2050, 2, 15);  
        d.addDays(4);  
        Date expected = new Date(2050, 2, 19);  
        assertEquals(expected, d);  
    }  
  
    @Test  
    public void test2() {  
        Date d = new Date(2050, 2, 15);  
        d.addDays(14);  
        Date expected = new Date(2050, 3, 1);  
        assertEquals(expected, d, "date after +14 days");  
    }  
}
```

Meta-annotazioni

- Stanno in `org.junit.jupiter.api`
- Le più comuni sono
 - `@BeforeEach`
 - `@AfterEach`
 - `@BeforeAll` (il metodo deve essere static)
 - `@AfterAll` (il metodo deve essere static)
 - `@DisplayName`
 - `@Tag`
 - `@Disabled`
- Altre più strane sono
 - `@Nested`
 - `@ParameterizedTest`
 - `@RepeatedTest`

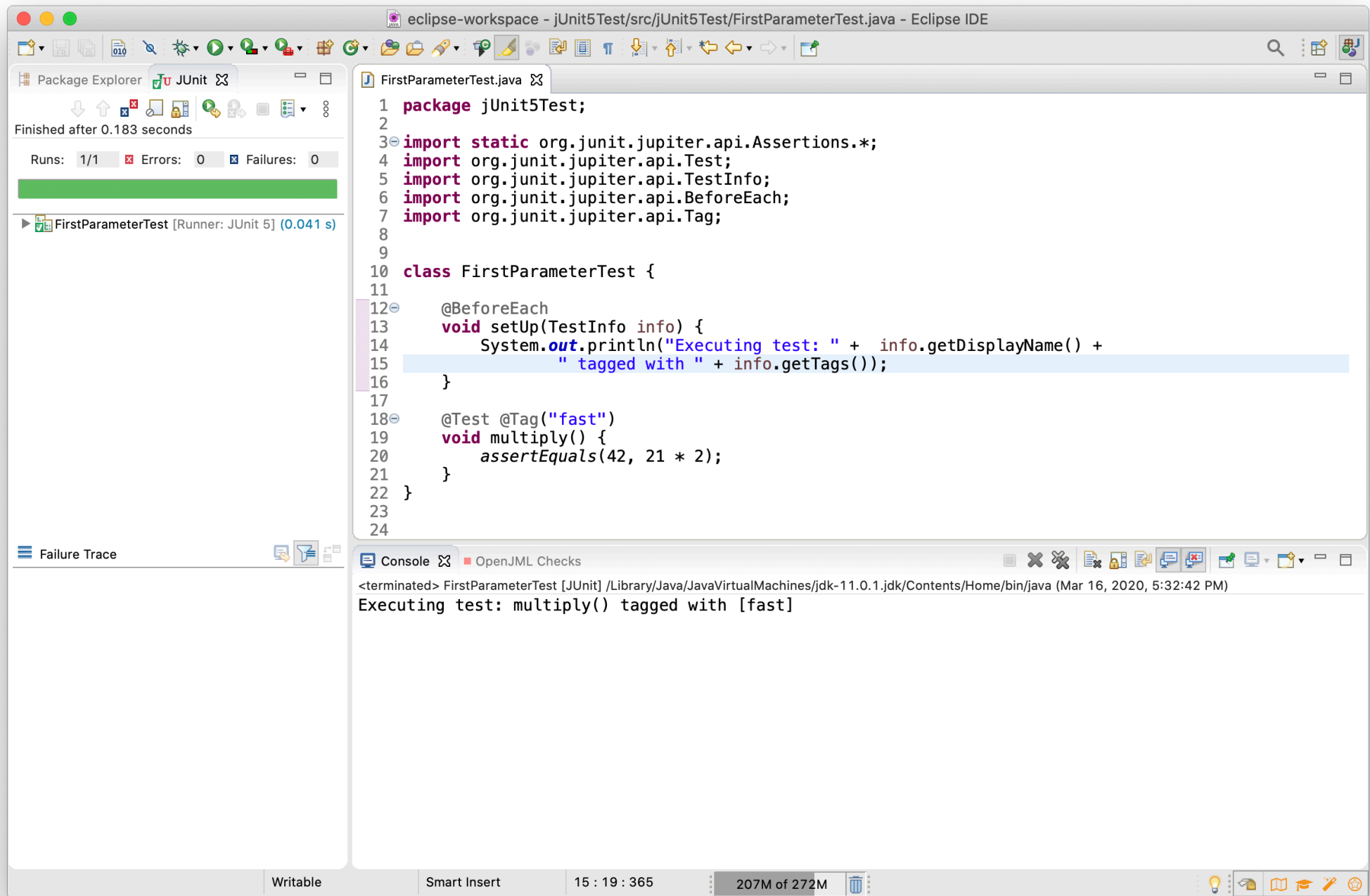
| JUnit 5 | JUnit 4 | Description |
|-------------|--------------|---|
| @Test | @Test | The annotated method is a test method. No change from JUnit 4. |
| @BeforeAll | @BeforeClass | The annotated (static) method will be executed once before any @Test method in the current class. |
| @BeforeEach | @Before | The annotated method will be executed before each @Test method in the current class. |
| @AfterEach | @After | The annotated method will be executed after each @Test method in the current class. |
| @AfterAll | @AfterClass | The annotated (static) method will be executed once after all @Test methods in the current class. |
| @Disabled | @Ignore | The annotated method will not be executed (it will be skipped), but reported as such. |

@DisplayName

```
import java.util.concurrent.ThreadLocalRandom;
import java.time.Duration;

class FirstJupiterTest {
    @Test
    @DisplayName("First test")
    void allYourBase() {
        assertAll(
            () -> assertTrue(true),
            () -> assertEquals("42", "4" + "2"),
            () -> assertTimeout(Duration.ofSeconds(1), () -> 42 * 42)
        );
    }

    @Test
    @DisplayName("Second test")
    void toInfinity() {
        Integer result = assertDoesNotThrow(() ->
            ThreadLocalRandom.current().nextInt(42)); assertTrue(result < 42);
    }
}
```



@TestInstance

@TestInstance is a type-level annotation that is used to configure the **lifecycle** of test instances for the annotated test class or test interface.



By default, both JUnit4/5 create a new instance of the test class before running each test method.

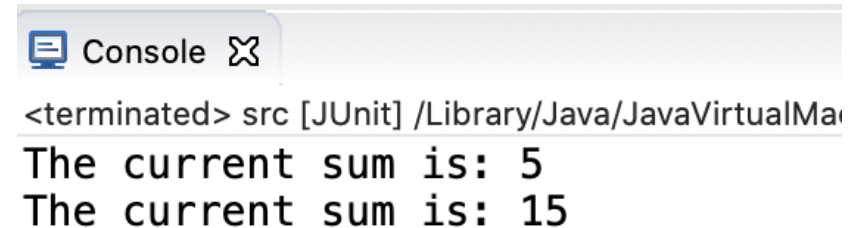
@TestInstance

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
class PerClassTestLifecycleTest {
    private int sum = 0;

    @Test
    void add5() { sum += 5; }

    @Test
    void add10() { sum += 10; }

    @AfterEach
    void tearDown() {
        System.out.println("The current sum is: " + sum);
    }
}
```



The screenshot shows a console window titled "Console" with a close button. It displays the output of a test run. The first line is "<terminated> src [JUnit] /Library/Java/JavaVirtualMa". The second line is "The current sum is: 5". The third line is "The current sum is: 15".

```
<terminated> src [JUnit] /Library/Java/JavaVirtualMa
The current sum is: 5
The current sum is: 15
```

Provate a rimuovere/commentare la riga..

@Nested

@Nested is used to signal that the annotated class is a nested, non-static test class (i.e., an *inner class*) that can share setup and state with an instance of its **enclosing class**.



```
@DisplayName("An ArrayList")
class NestedTest {
    private ArrayList<String> list;
```

```
@Nested
@DisplayName("when new")
class WhenNew {
    @BeforeEach
    void createNewList() {list = new ArrayList<>();}
    @Test
    @DisplayName("is empty")
    void isEmpty() {assertTrue(list.isEmpty());}
```

```
@Nested
@DisplayName("after adding an element")
class AfterAdding {
    String anElement = "an element";

    @BeforeEach
    void addAnElement() {list.add(anElement);}
    @Test
    @DisplayName("it is no longer empty")
    void isEmpty() {assertFalse(list.isEmpty());}
```

```
    }
}
}
```

@Nested

Finished after 0.173 seconds

Runs: 2/2 ✖ Errors: 0 ❌ Failures: 0

```
▼ [icon] An ArrayList [Runner: JUnit 5] (0.024 s)
  ▼ [icon] when new (0.024 s)
    ▼ [icon] is empty (0.013 s)
      ▼ [icon] after adding an element (0.011 s)
        [icon] it is no longer empty (0.011 s)
```

Test parametrici

- Altre sorgenti
 - Stringhe CSV
 - file CSV
 - interfaccia ArgumentsProvider

```
@ParameterizedTest
@ValueSource(ints = {2, 4, 6, 8, 9})
void evens(int value) {
    assertEquals(0, value % 2);
}
```

```
@ParameterizedTest
@MethodSource("supplyOdds")
void odds(int value) {
    assertEquals(1, value % 2);
}
```


```
private static Stream<Integer>
supplyOdds() {
    return Stream.of(1, 3, 5, 7, 9, 11);
}
```

Finished after 0.24 seconds

Runs: 11/11  Errors: 0  Failures: 1


ParametersTest [Runner: JUnit 5] (0.069 s)


odds(int) (0.041 s)

 [1] 1 (0.041 s)

 [2] 3 (0.003 s)


 [3] 5 (0.004 s)


 [4] 7 (0.002 s)

 [5] 9 (0.006 s)


 [6] 11 (0.009 s)

evens(int) (0.002 s)

 [1] 2 (0.001 s)


 [2] 4 (0.002 s)

 [3] 6 (0.003 s)

 [4] 8 (0.003 s)

 [5] 9 (0.009 s)

Failure Trace

 org.opentest4j.AssertionFailedError: expected: <0> but was: <1>

 at nested.ParametersTest.evens(ParametersTest.java:15)

Test ripetuti

`@RepeatedTest(value =)`



Each invocation of the repeated test behaves like the execution of a regular `@Test` method with full support for the same lifecycle callbacks and extensions. In addition, the current repetition and total number of repetitions can be accessed by having the `RepetitionInfo` injected.





Test ripetuti


```
@DisplayName("Repeated test")
@RepeatedTest(value = 10)
void repeatedTestWithInfo(RepetitionInfo repetitionInfo) {
    assertTrue(repetitionInfo.getCurrentRepetition() <= repetitionInfo.getTotalRepetitions());
    assertEquals(10, repetitionInfo.getTotalRepetitions());
}
```











Finished after 0.2 seconds

Runs: 10/10  Errors: 0  Failures: 0



▼  Test [Runner: JUnit 5] (0.028 s)

▼  Repeated test (0.028 s)

-  repetition 1 of 10 (0.028 s)
-  repetition 2 of 10 (0.003 s)
-  repetition 3 of 10 (0.003 s)
-  repetition 4 of 10 (0.005 s)
-  repetition 5 of 10 (0.003 s)
-  repetition 6 of 10 (0.003 s)
-  repetition 7 of 10 (0.005 s)
-  repetition 8 of 10 (0.003 s)
-  repetition 9 of 10 (0.002 s)
-  repetition 10 of 10 (0.007 s)

Enabled and Disabled

```
import org.junit.jupiter.api.Disabled;  
import org.junit.jupiter.api.condition.DisabledOnJre;  
import org.junit.jupiter.api.condition.EnabledOnOs;  
import org.junit.jupiter.api.condition.OS;  
import org.junit.jupiter.api.condition.JRE;
```

```
@Test
```

```
@Disabled("for demonstration purposes")
```

```
@EnabledOnOs({OS.MAC, OS.LINUX})
```

```
@DisabledOnJre(JRE.JAVA_12)
```

Eccezioni

```
class ExceptionTest {  
    @Test  
    void assertThrowsException() {  
        String str = null;  
        assertThrows(IllegalArgumentException.class, () -> {Integer.valueOf(str);});  
    }  
  
    @Test  
    void exceptionTesting() {  
        Exception exception;  
  
        exception = assertThrows(ArithmeticException.class, () -> Calculator.divide(1, 0));  
        assertEquals("/ by zero", exception.getMessage());  
    }  
}
```

Timeout

```
@BeforeEach @Timeout(5)
void setUp() {
    // fails if execution time exceeds 5 seconds
}
```

```
@Test
@Timeout(value = 100, unit = TimeUnit.MILLISECONDS)
void failIfExecutionTimeExceeds100Milliseconds() {
    // fails if execution time exceeds 100 milliseconds
}
```

```
@Test
void timeoutNotExceeded() {
    assertTimeout(Duration.ofMinutes(2), () -> {
        // Perform task that takes less than 2 minutes.
    });
}
```

Assunzioni

- Se l'ipotesi non è valida, l'esecuzione del test viene abortita

```
import static org.junit.jupiter.api.Assertions.*;
import static org.junit.jupiter.api.Assumptions.*;
import org.junit.jupiter.api.Test;
```

```
class AssumptionTest {
    @Test
    void trueAssumption() {
        assumeTrue(5 > 1);
        assertEquals(5 + 2, 7);
    }

    @Test
    void falseAssumption() {
        assumeFalse(5 < 1);
        assertEquals(5 + 2, 7);
    }
}
```

```
    @Test
    void assumptionThat() {
        String someString = "Just a string";
        assumingThat(someString.equals("Just a string"),
            () -> assertEquals(2 + 2, 4));
    }
}
```

Lambda

```
class TestFactoryTest {  
    @TestFactory  
    public Stream<DynamicTest> testOneThroughTen() {  
        return Stream.of(1,2,3,4,5,6,7,8,9,10)  
            .map((i) -> DynamicTest.dynamicTest("input: " + i, () -> assertTrue(i != 4)));  
    }  
}
```

```
class LabdaTest {  
    @Test  
    void lambdaExpressions() {  
        assertTrue((Stream.of(1, 2, 3))  
            .mapToInt(i -> i)  
            .sum() > 5);  
    }  
}
```

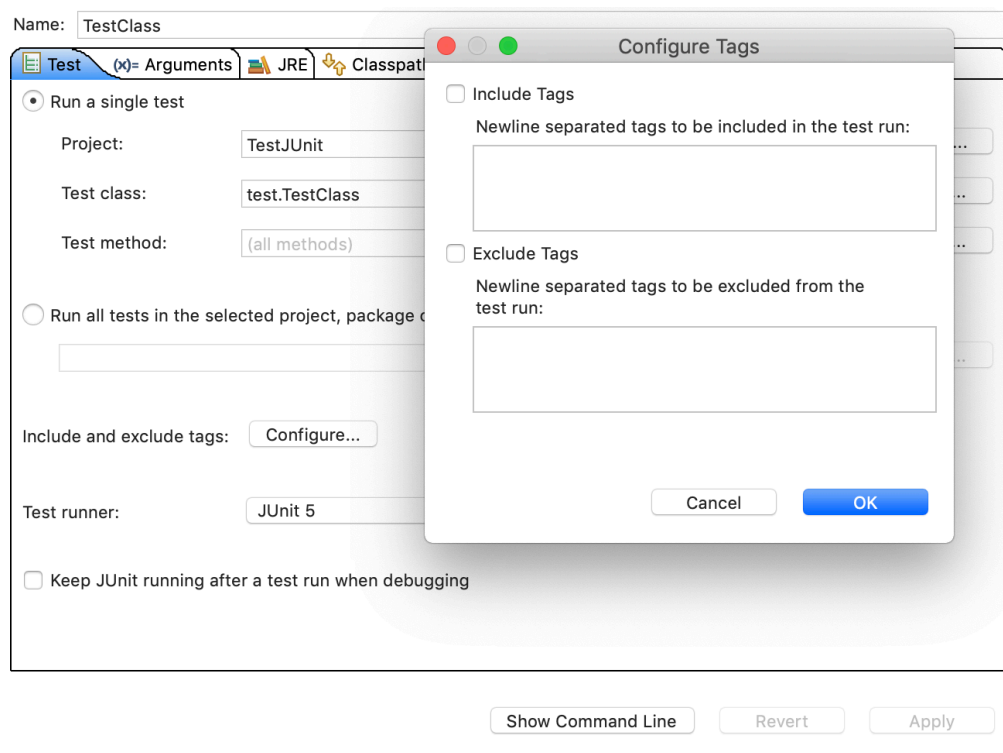
Runs: 10/10 Errors: 0 Failures: 1

TestFactoryTest [Runner: JUnit 5] (0.010 s)

- testOneThroughTen() (0.010 s)
 - input: 1 (0.010 s)
 - input: 2 (0.002 s)
 - input: 3 (0.017 s)
 - input: 4 (0.009 s)
 - input: 5 (0.001 s)
 - input: 6 (0.001 s)
 - input: 7 (0.000 s)
 - input: 8 (0.001 s)
 - input: 9 (0.001 s)
 - input: 10 (0.010 s)

Tag

- Su classe e su metodo
- includeTags e excludeTags



```
class TestClass {  
    @Test  
    @Tag("development")  
    @Tag("production")  
    void testA() {}  
  
    @Test  
    @Tag("development")  
    void testB() {}  
  
    @Test  
    @Tag("development")  
    void testC() {}  
}
```

Un esempio Black Box

- Vi hanno dato una semplice classe Calculator
- Dovete testare il fattoriale
- (E magari poi aprire.. diventa **white**...)
- Casi noti:
- Input 0 e neg...
- Input 10
- Basta? (proviamo..)

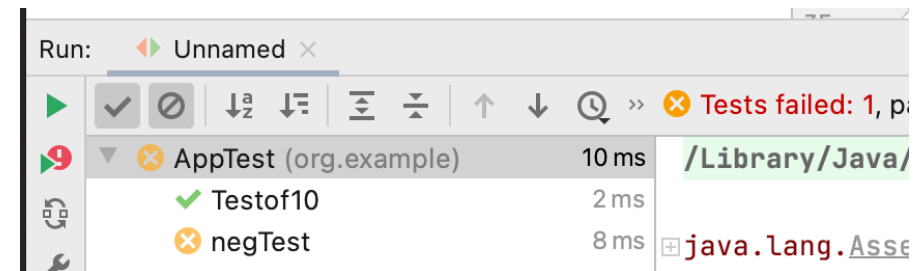
Un esempio Black Box II

```
public void negTest()  
{  
    Calculator c = new Calculator();  
    assertTrue( c.Factorial(-1) > 0 );  
}  
@Test  
public void Testof10()  
{  
    Calculator c = new Calculator();  
    assertTrue( c.Factorial(10) == 3628800 );  
}
```

Un esempio Black Box III

Ok, un fallimento.

Fixiamo codice: (WHITE...)



```
int Factorial( int number ) {  
    return number <= 1 ? number : Factorial( number - 1 ) * number;  
}
```

Diventa:

```
int Factorial( int number ) {  
    return number <= 1 ? 1 : Factorial( number - 1 ) * number;  
}
```

Un esempio Black Box IV

Run.. ok.

Ma siamo sicuri?

Aggiungiamo:

```
@Test
    public void Testof18()
    {
        Calculator c = new Calculator();
        assertTrue( c.Factorial(18) > 0 );
    }
}
```



IntelliJ: creazione dei test

Possiamo velocizzare la scrittura dei test?



IntelliJ: creazione dei test

ES: scriviamo una semplice classe Divider:

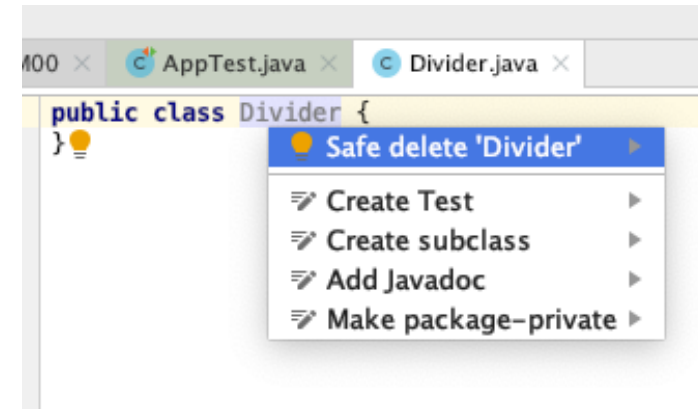
```
public class Divider {  
    double doCalc(int x, int y){  
        return x/y;  
    }  
}
```



IntelliJ: creazione dei test

IntelliJ ci "aiuta":

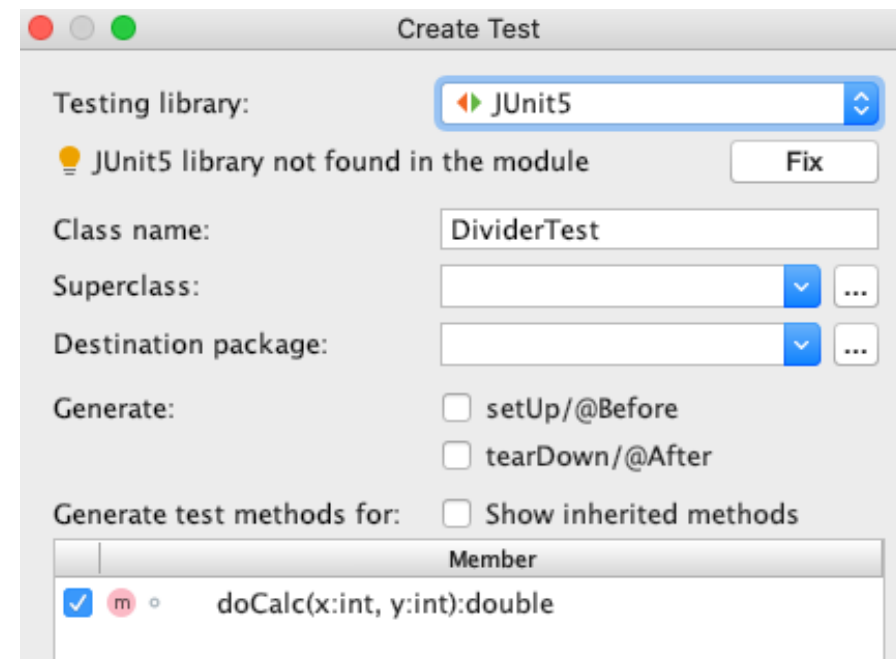
Alt Invio



JUnit 5.

Scegliete quali metodi testare.

Se appare Fix. ok!



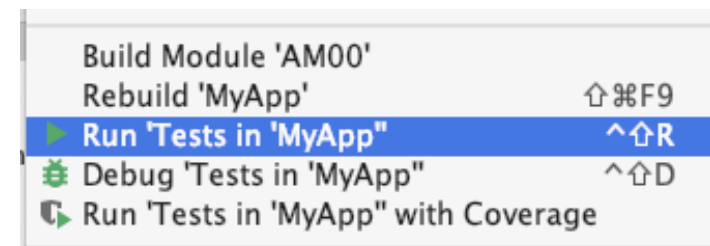


IntelliJ: creazione dei test

modifichiamo:

```
class DividerTest {  
  
    @Test  
    void doCalc() {  
  
        Divider divider = new Divider();  
        double r = divider.doCalc(3, 2);  
        assertEquals(1.5, r);  
  
    }  
}
```

run...





IntelliJ: creazione dei test

Perche' non funziona?

Git repo:
<https://github.com/ingconti/JUnitIntro>

... per finire

- La comunicazione tra le asserzioni ed il framework di test è lasca e avviene per mezzo di eccezioni
- JUnit 5 consente l'uso di librerie di asserzioni diverse
 - Ad esempio, Hamcrest, AssertJ, o Google Truth potrebbero essere usate con JUnit 5 senza necessità di cambiamenti

Alcuni consigli

- Una cosa per volta e per metodo test
 - Dieci piccoli test sono molto meglio di un solo test dieci volte più grande
- Ogni metodo test dovrebbe avere pochi (un) metodo assert
 - La prima assert che fallisce blocca il test
 - Non si riuscirebbe quindi a sapere se eventuali asserzioni successive fallirebbero
- I test dovrebbero evitare logiche complesse
 - Minimizzare l'uso di if, loops, switch, ecc
 - Evitare try/catch
 - Se l'eccezione fosse sollevabile, bisognerebbe usare expected
 - Altrimenti sarebbe meglio lasciare che JUnit gestisca l'eccezione
- Test complessi vanno bene, ma solo in aggiunta a quelli semplici

“Bad smells”

- Ogni test dovrebbe essere auto-contenuto e non considerare gli altri
- Cose da evitare
 - Vincolare l'ordine di applicazione dei test
 - I test si chiamano a vicenda
 - Uso di oggetti condivisi

Debugging

Debugging

- Trovare il difetto del programma che dà origine a comportamento erraneo rivelato dal test
- Tecniche di debugging riconducibili a due tipi di azioni
 - Identificazione della causa effettiva usando dati di test più semplici possibili
 - Localizzazione della porzione di codice difettoso osservando gli stati intermedi della computazione
- Il costo del debugging (spesso "contabilizzato" sotto la voce: test) può essere parte preponderante del costo di sviluppo
 - È molto importante sviluppare il software in modo sistematico per minimizzare sforzo speso in debugging

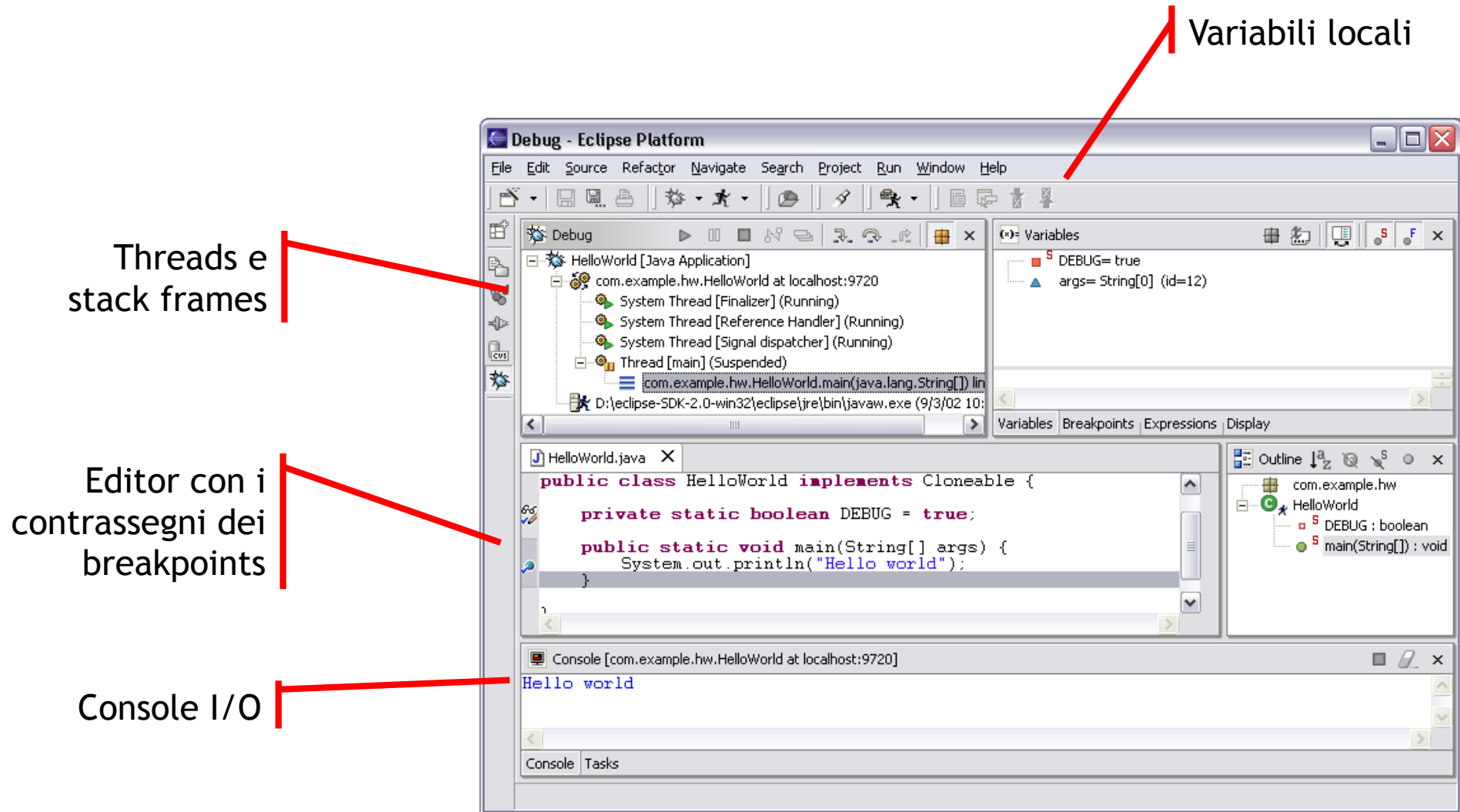
Debugging

- Debugging è attività difficile da rendere sistematica
 - L'efficienza dipende dalle persone ed è poco prevedibile
- Sistematicità
 - Identificare almeno uno stato corretto S1 e uno non corretto S2
 - Cercare di capire quali stati intermedi tra S1 e S2 sono corretti e quali no
- Debugger
 - breakpoint
 - esecuzione passo-passo
 - visualizzazione e modifica di variabili

Funzionalità essenziali

- Breakpoint: permettono di interrompere l'esecuzione in un certo punto
- Esecuzione passo passo: permette di avanzare l'esecuzione di un passo per volta
- Esame dello stato intermedio: permette di visualizzare il valore delle singole variabili
- Modifica dello stato: permette di modificare il valore di una o più variabili prima di riprendere l'esecuzione
- Oggi si usano debugger “simbolici” che consentono di operare al livello del linguaggio di programmazione
 - variabile = variabile del linguaggio, non cella di memoria
 - passo = istruzione del linguaggio

Il debugger di Eclipse



Programmazione difensiva

- Un pizzico di paranoia può essere utile
- Possiamo/dobbiamo scrivere i programmi in modo che scoprano e gestiscano ogni possibile situazione anomala:
 - procedure chiamate con parametri attuali scorretti,
 - file: devono essere aperti ma sono chiusi, devono aprirsi e non si aprono...
 - riferimenti ad oggetti null, array vuoti ...
- Il meccanismo delle eccezioni è un aiuto utile
- Essere scrupolosi con il test
 - ricordarsi che l'obiettivo è trovare gli errori, non essere contenti di non trovarne
- Può convenire dare ad altri il compito di collaudare i propri programmi