# Test di unità con JUnit4

# Richiamo sul test di unità

- Il test d'unità è una metodologia che permette di verificare il corretto funzionamento di singole unità di codice in determinate condizioni.
- Nel caso di Java (e più in generale nella programmazione orientata agli oggetti) si considera come unità di codice un metodo o un gruppo di metodi che implementano una singola funzionalità.
- Il test di unità si un'applicazione può essere organizzato in **Test Case** e **Test Suite**.
  - Test Case: test che verifica una singola unità di codice
  - Test Suite: gruppo di Test Case che verificano funzionalità correlate.

# Progettare un Test Case

1. Individuare l'unità di codice da testare (es. un metodo).
2. Definire un modo con cui può essere eseguito il codice dell'unità da testare.
3. Definire un **oracolo** che, assumendo che il codice da testare sia corretto, preveda il risultato dell'esecuzione dell'unità da testare.
4. Scrivere l'implementazione del Test Case:
   - Eseguire il codice dell'unità da testare.
   - Verificare che il risultato della sua esecuzione sia conforme a quanto previsto dall'oracolo.

# Individuare l'unità da testare

- In generale ogni unità di codice contenente una funzionalità che può essere eseguita deve essere testata.
- Esempio:
  - Componenti algoritmiche dell'applicazione.
  - Tutto ciò che può modificare lo stato dell'applicazione.
  - ...

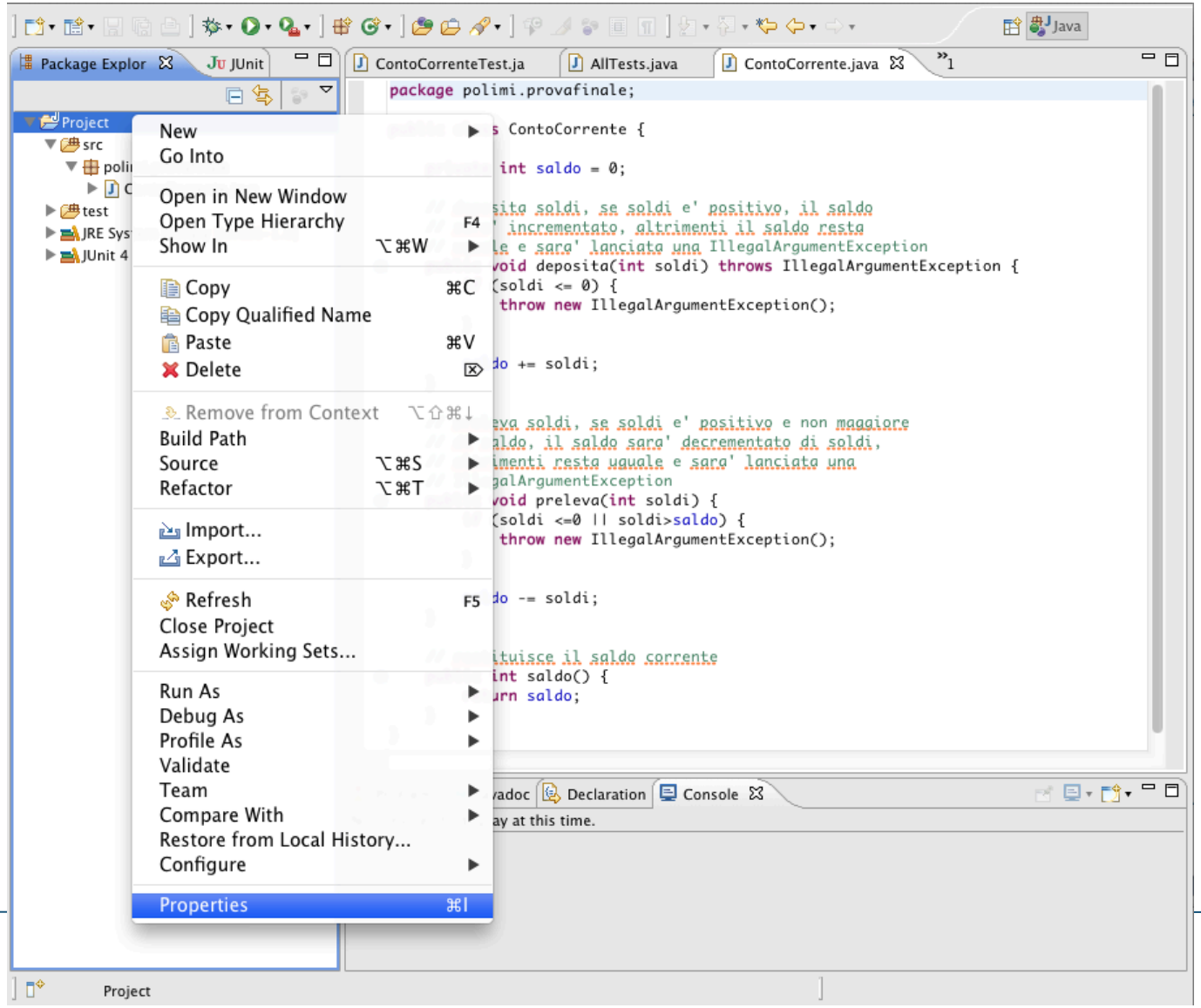
# Come testare un'unità

- Test Black-Box (funzionale)
  - È indispensabile quando non si ha a disposizione l'implementazione oppure quando si scrive il test prima dell'implementazione (Test Driven Development)
- Test White-Box (strutturale)
  - Questa tecnica può essere usata nelle sue diverse forme (vedere teoria...) quando si ha a disposizione l'implementazione dell'unità da testare.
  - Può essere affiancata ai test Black-Box per migliorare ulteriormente la qualità del software.

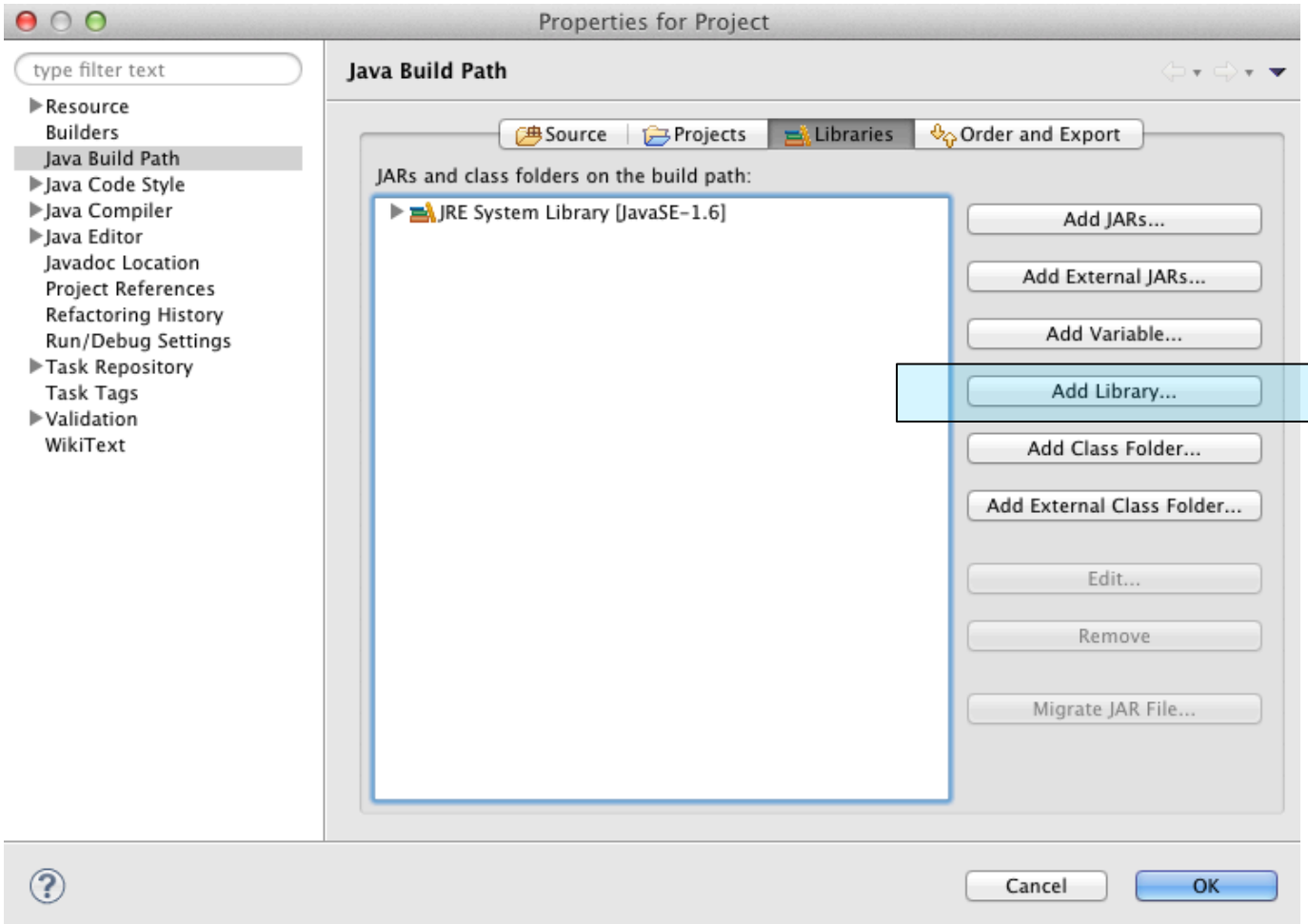
# JUnit

- È un framework open-source che supporta la scrittura e l'esecuzione automatica di **Test Case** e **Test Suite**.
- Richiede l'utilizzo di una libreria (solitamente già inclusa nella propria distribuzione di Java/Eclipse)
  - http://www.junit.org
- Si integra con Eclipse
- Ne esistono due versioni principali il cui modo d'uso è diverso: JUnit3 (diffuso prima Java 5 e tutt'ora supportato) e JUnit4 (richiede Java 5 o superiore).
  - Nel laboratorio useremo **JUnit4**.

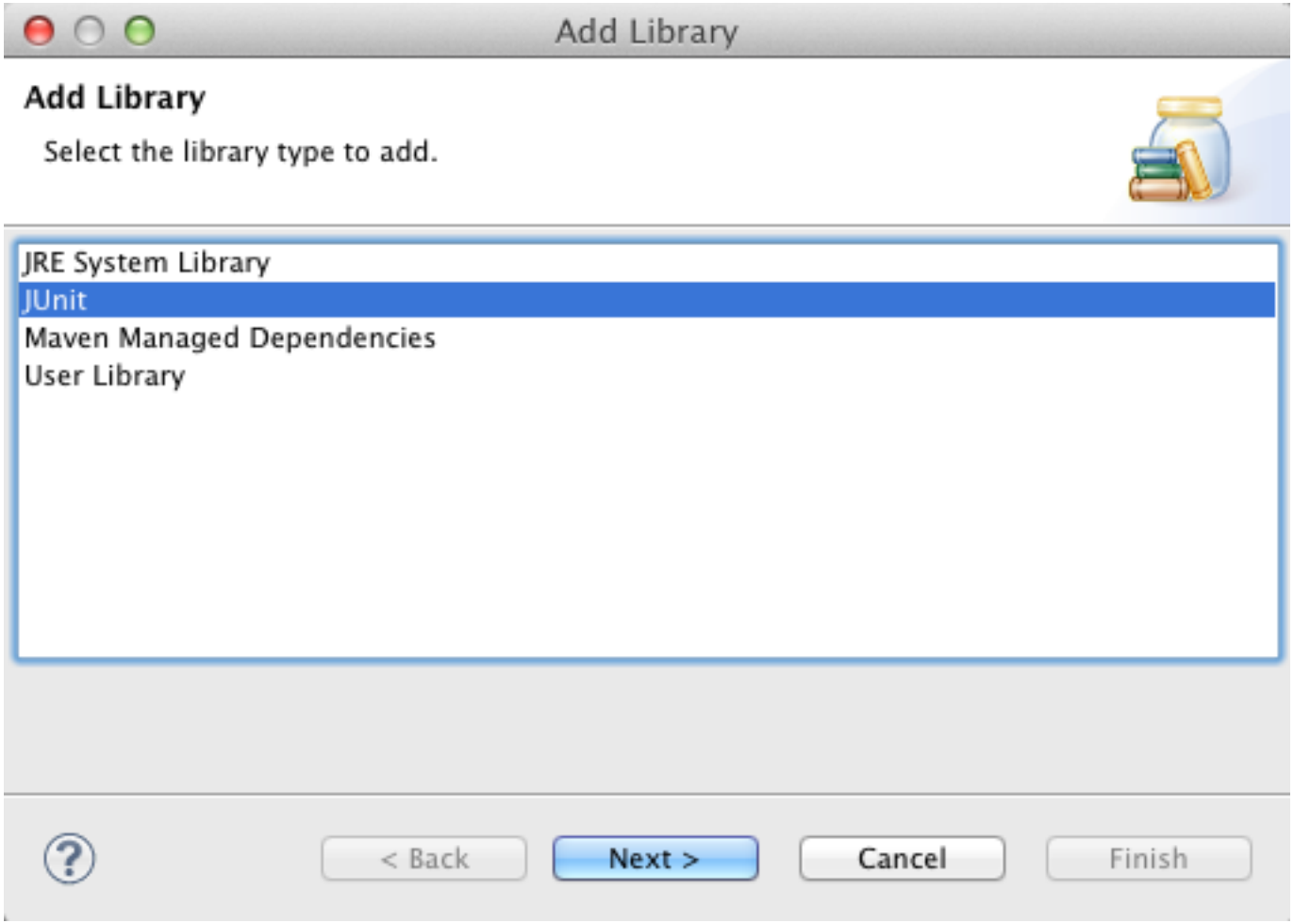
# Aggiungere libreria JUnit4 in Eclipse (1)



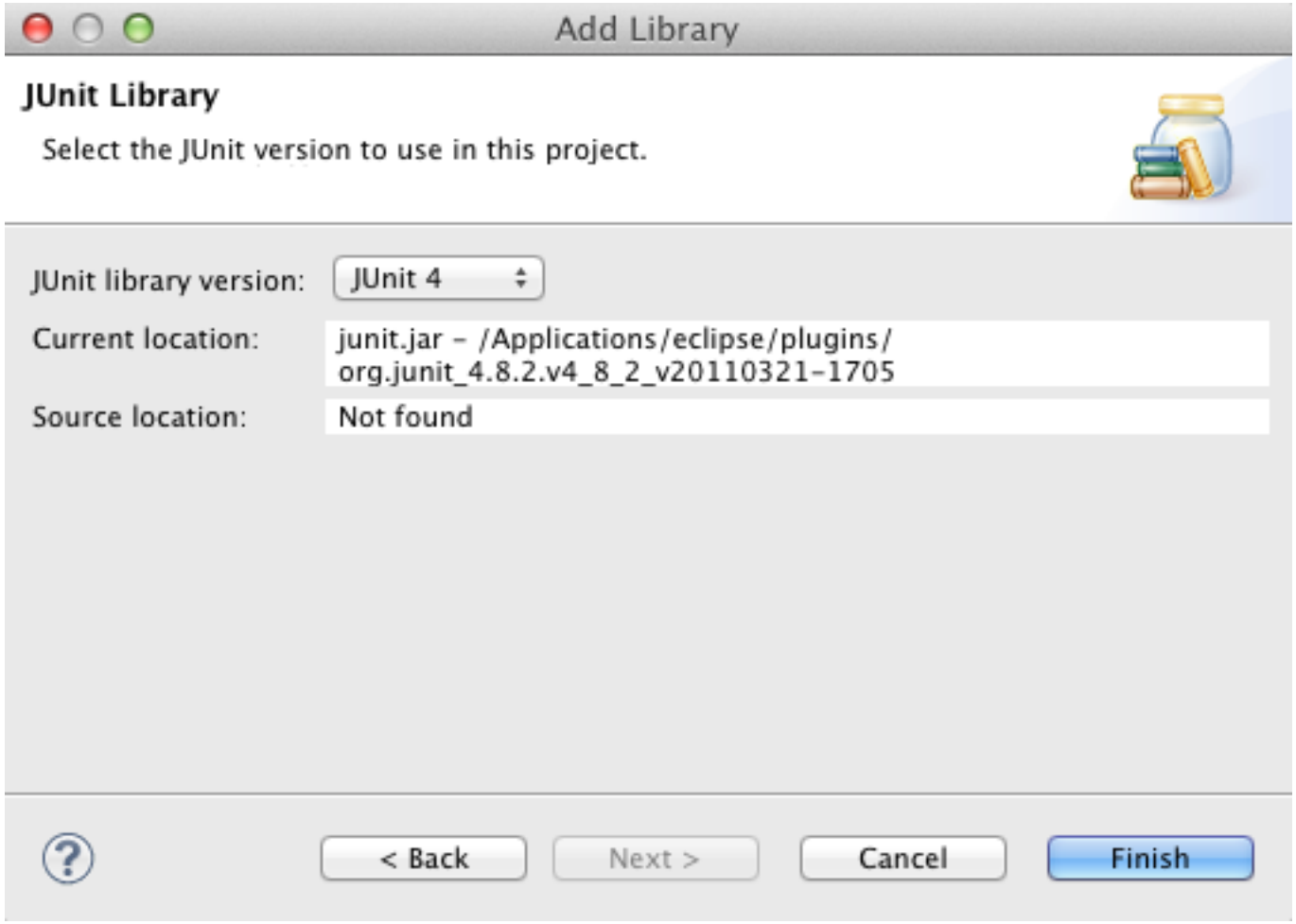
# Aggiungere libreria JUnit4 in Eclipse (2)



# Aggiungere libreria JUnit4 in Eclipse (3)



# Aggiungere libreria JUnit4 in Eclipse (4)



# JUnit: Test Case e Test Suite

- È necessario creare:
  1. Per ogni classe da testare: una classe **Test Case** che conterrà tutti i test necessari per testare la classe.
  2. Per ogni gruppo di Test Case correlati: una classe **Test Suite** che conterrà il riferimento a tutte le classi Test Case associate ad essa.
- È buona norma mettere le classi usate nel testing in una gerarchia di directory separata dal resto del codice.

# Classe Test Case (1)

- Ogni classe Test Case è solitamente associata ad una classe da testare.
- Ogni classe Test Case può avere un nome arbitrario, ma normalmente si usa il nome della classe testata seguito da Test.
- Solitamente la classe Test Case appartiene allo stesso package della classe che viene testata.
- Tutti i test contenuti in una classe Test Case possono essere eseguiti direttamente dal menu **Run As** di Eclipse

# Classe Test Case (2)

- Una classe Test Case non deve definire un costruttore e può avere i propri metodi annotati nel seguente modo:
  - **@Test** (richiede import **org.junit.Test**)
    Specifica che il metodo consiste in un test.
  - **@Before** (richiede import **org.junit.Before**)
    Specifica che il metodo deve essere eseguito **prima** di **ogni** test.
  - **@After** (richiede import **org.junit.After**)
    Specifica che il metodo deve essere eseguito **dopo ogni** test.
  - **@BeforeClass** (richiede import **org.junit.BeforeClass**)
    Specifica che il metodo deve essere eseguito prima di eseguire il primo test.
  - **@AfterClass** (richiede import **org.junit.AfterClass**)
    Specifica che il metodo deve essere eseguito dopo aver eseguito l'ultimo test.
- **NB**: i metodi annotati devono essere **di tipo void**, **senza parametri**, e devono avere la **clausola throws Exception**.

# Classe **Assert** di JUnit

- Il risultato di ogni test deve essere confrontato con quello previsto dall'oracolo.
- Il confronto tra il risultato ottenuto e quello previsto si effettua usando i metodi statici della classe **Assert** di JUnit.
- Se tutti i confronti hanno esito positivo, il test avrà successo, altrimenti il test fallisce.
- I metodi statici della classe Assert possono essere resi visibili come metodi statici privati della classe Test Case utilizzando questo import statico:
  - **import static org.junit.Assert.*;**

# Metodi importanti della classe **Assert**

- **fail(String msg)**
  fa fallire il test mostrando **msg** come motivo.
- **assertTrue(String msg, boolean cond)**
  verifica che la condizione sia vera
- **assertFalse(String msg, boolean cond)**
  verifica che la condizione sia falsa
- **assertNull(String msg, Object obj)**
  verifica che l'oggetto sia nullo
- **assertEquals(String msg, Object expected, Object actual)**
  verifica che **actual.equals(expected)** sia vera
- **assertSame(String msg, Object expected, Object actual)**
  verifica che **actual==expected** sia vera

# Esempio: ContoCorrente

```
// classe che rappresenta un conto corrente (inizialmente a 0)
public class ContoCorrente {

    // deposita soldi, se soldi e' positivo, il saldo
    // sara' incrementato, altrimenti il saldo resta
    // uguale e sara' lanciata una IllegalArgumentException
    public void deposita(int soldi) throws
                                IllegalArgumentException { ... }

    // preleva soldi, se soldi e' positivo e non maggiore
    // di saldo, il saldo sara' decrementato di soldi,
    // altrimenti resta uguale e sara' lanciata una
    // IllegalArgumentException
    public void preleva(int soldi) { ... }

    // restituisce il saldo corrente
    public int saldo() { ... }
}
```

# Classe Test Case Conto Corrente (1)

```
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class ContoCorrenteTest {

    private ContoCorrente contoCorrente;

    @Before
    public void setUp() throws Exception {
        // codice comune a tutti i test
        contoCorrente = new ContoCorrente();
    }
    ...
}
```

Il metodo annotato con **@Before** crea una istanza di **ContoCorrente** prima di ogni test.

# Classe Test Case Conto Corrente (2)

- Descrizione test:
  - **un conto corrente nuovo deve avere saldo 0**.

```
@Test
public void contoHaSaldoInizialeZero() {
    assertEquals("saldo iniziale non nullo",
            0, contoCorrente.saldo());
}
```

# Classe Test Case Conto Corrente (3)

```
@Test
public void testDeposita() {
    contoCorrente.deposita(1);
    assertEquals("saldo 0 non diventa 1 dopo deposito di 1",
                 1,contoCorrente.saldo());
    contoCorrente.deposita(1);
    assertEquals("saldo 1 non diventa 2 dopo deposito di 1",
                 2,contoCorrente.saldo());
    try {
        contoCorrente.deposita(0);
        fail("sono riuscito a depositare zero");
    } catch (IllegalArgumentException e) {
        assertEquals("deposito fallito ha cambiato il saldo",
                     contoCorrente.saldo(), 2);
    }
    ...
}
```

# Classe Test Case Conto Corrente (4)

```
@Test
public void testPreleva() {
    try {
        contoCorrente.preleva(1);
        fail("ho prelevato 1 da un conto con saldo zero");
    } catch (IllegalArgumentException e) {
        assertEquals("prelievo fallito ha cambiato il saldo",
                     0, contoCorrente.saldo());
    }
    contoCorrente.deposita(50);
    contoCorrente.preleva(25);
    assertEquals("saldo 50 non diventa 25 dopo prelievo di 25",
                 25, contoCorrente.saldo());
```

# Classe Test Suite (1)

- È associata ad un insieme di classi Test Case
- Può avere un nome arbitrario, ma solitamente si usa un nome che finisce con **Tests**.
- Deve importare le seguenti classi di JUnit:
  - **import org.junit.runner.RunWith;**
  - **import org.junit.runners.Suite;**
  - **import org.junit.runners.Suite.SuiteClasses;**
- Se in diverso package deve importare anche tutte le classi Test Case.

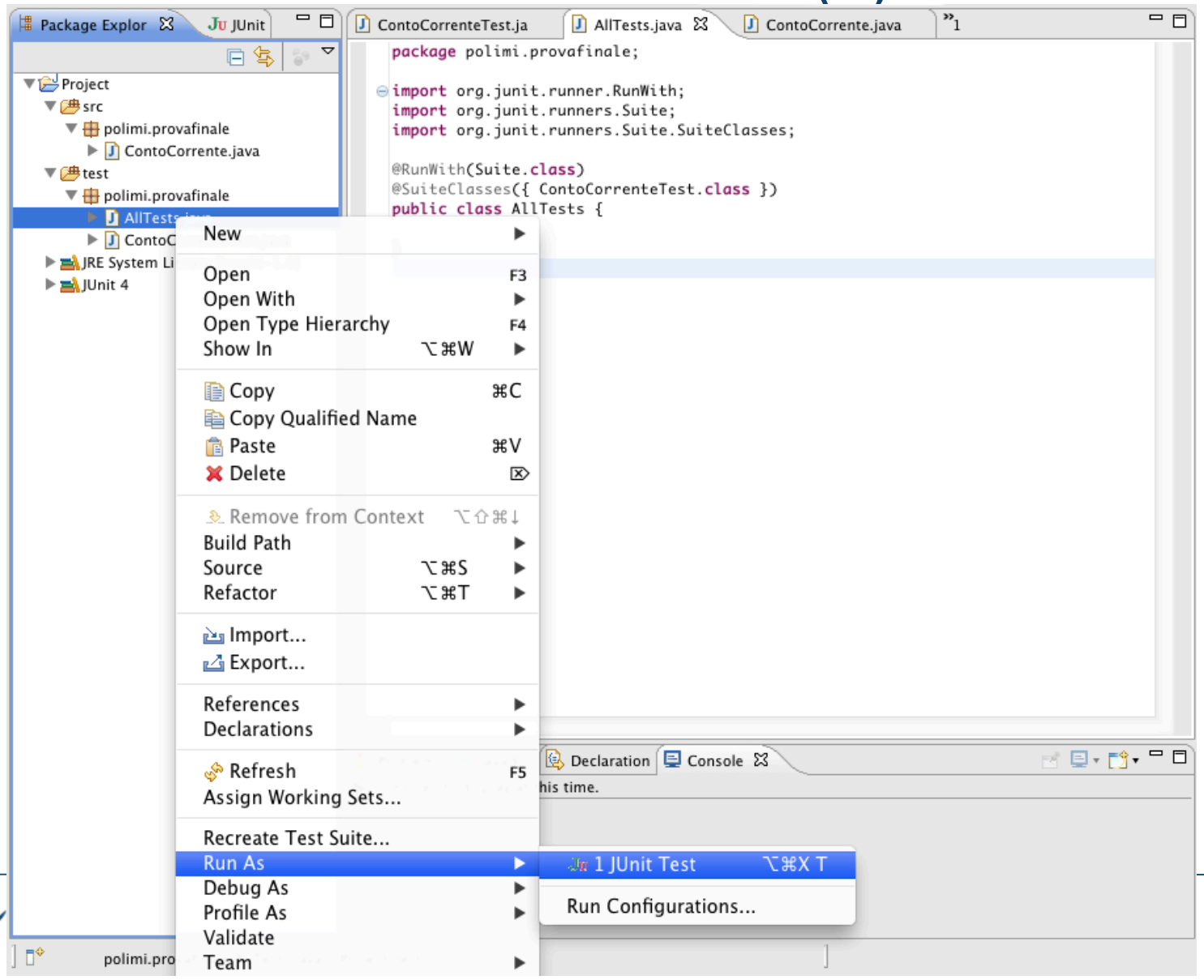
# Classe Test Suite (2)

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;
@RunWith(Suite.class)
@Suite.SuiteClasses( { FirstTest.class,
                       SecondTest.class,
                       ...

})

public class AllTests {
}
```

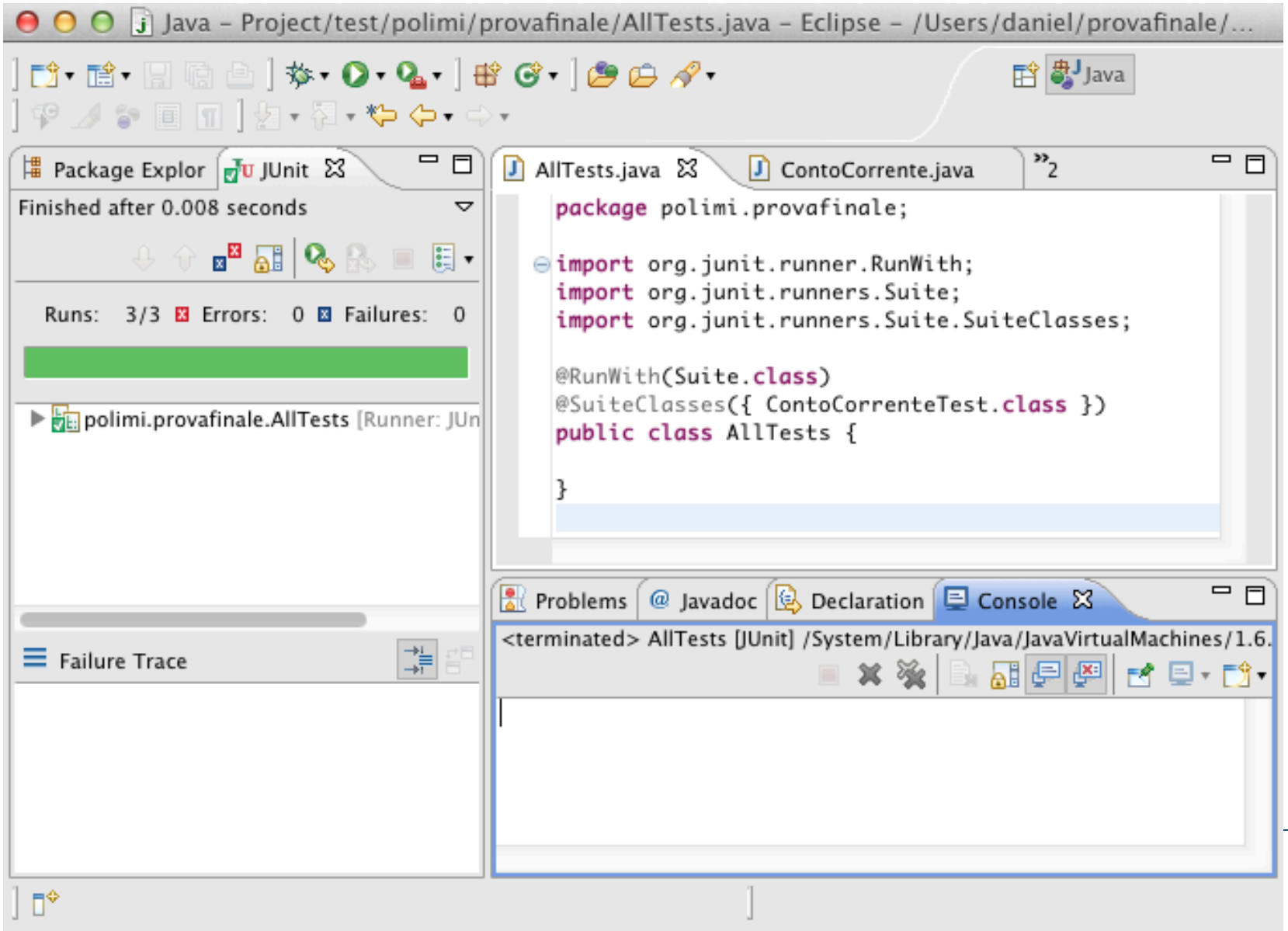
# Classe Test Suite Conto Corrente

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({ ContoCorrenteTest.class })

public class AllTests {
}
```

# Esecuzione dei test (1)



# Esecuzione dei test (2)



# Riferimenti JUnit

- http://www.junit.org

# Documentazione del codice con JavaDoc

- JavaDoc specifica una convenzione nella scrittura dei commenti della propria applicazione.
- Permette di generare automaticamente delle pagine HTML con la documentazione delle proprie classi.
- Lo stile della documentazione creata è lo stesso della documentazione ufficiale delle API Java:
  - http://docs.oracle.com/javase/6/docs/api/

# Esempio: JavaDOC di java.lang.String

**Java™ Platform Standard Ed. 6**

All Classes

Packages
java.applet
java.awt
java.awt.color
java.awt.datatransfer
java.awt.dnd
java.awt.event

**All Classes**
AbstractAction
AbstractAnnotationValueVisito
AbstractBorder
AbstractButton
AbstractCellEditor
AbstractCollection
AbstractColorChooserPanel
AbstractDocument
*AbstractDocument.AttributeCo*
*AbstractDocument.Content*
AbstractDocument.ElementEd
AbstractElementVisitor6
AbstractExecutorService
AbstractInterruptibleChannel
AbstractLayoutCache

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

java.lang

## Class String

java.lang.Object
  └ java.lang.String

**All Implemented Interfaces:**
Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"` this class.

Strings are constant; their values cannot be changed after they are created. String buffers support objects are immutable they can be shared. For example:

```
String str = "abc";
```

# Commenti Javadoc

- Possono riferirsi a classi o metodi (ma anche attributi)
- Sono nella forma:

```
/**
 ...
 */
```

- Dei tag permettono di specificare delle proprietà particolari: `@tag`

# Documentare una classe

```
/**
 DOCUMENTAZIONE:
 - COSA RAPPRESENTA LA CLASSE
 - A COSA SERVE
 - COME SI USA

 @author Dina Lampa
 @version 1.0
*/
public class Classe { ... }
```

# Esempio Documentazione Classe ContoCorrente

```
/**
 * La classe ContoCorrente rappresenta un conto corrente.
 * Lo stato della classe consiste nel saldo del conto corrente
 * bancario, il cui valore e' un intero inizialmente nullo.
 *
 * La classe permette di interrogare il saldo residuo e di
 * effettuare prelievi e depositi.
 *
 * @author Daniel J. Dubois
 * @version 1.0
 */
public class ContoCorrente {

    private int saldo = 0;
    ...
}
```

# Documentare un metodo

```
/**
 DOCUMENTAZIONE:
 - COSA FA IL METODO
 - COME USARLO

 @param x    COSA RAPPRESENTA x?
 @return     COSA RESTITUISCE
 @throws Ex  QUANDO VIENE LANCIATA?
*/
public int metodo(int x) throws Ex {...}
```

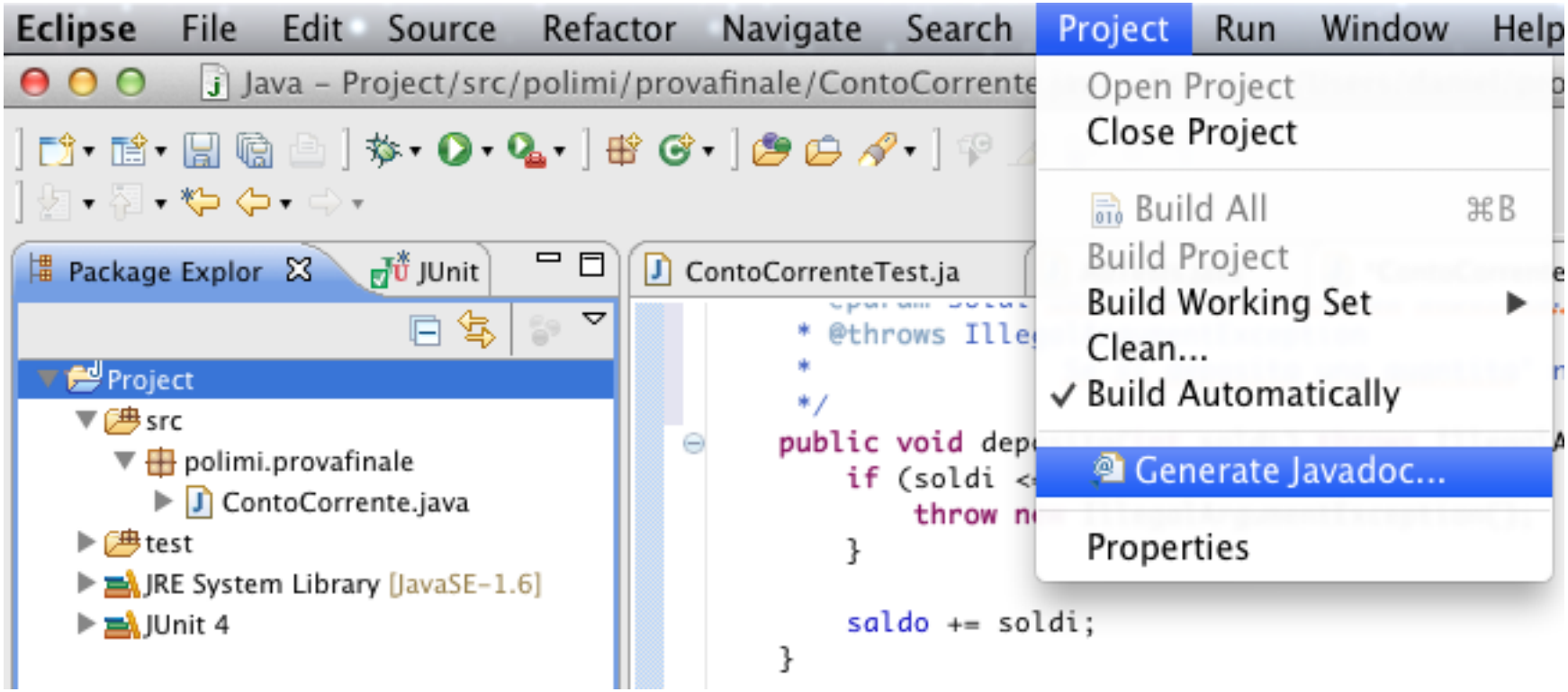
# Esempio documentazione metodi nella classe ContoCorrente (1)

```
/**
 * Deposito dei soldi nel conto corrente.
 * Dopo l'invocazione di questo metodo, in assenza
 * di eccezioni, il saldo sara' incrementato della
 * quantita' di soldi depositata.
 *
 * @param  soldi Quantita' di soldi da depositare
 * @throws IllegalArgumentException
 *             Se si deposita una quantita' non positiva
 */
public void deposita(int soldi) throws
                        IllegalArgumentException { ... }
```

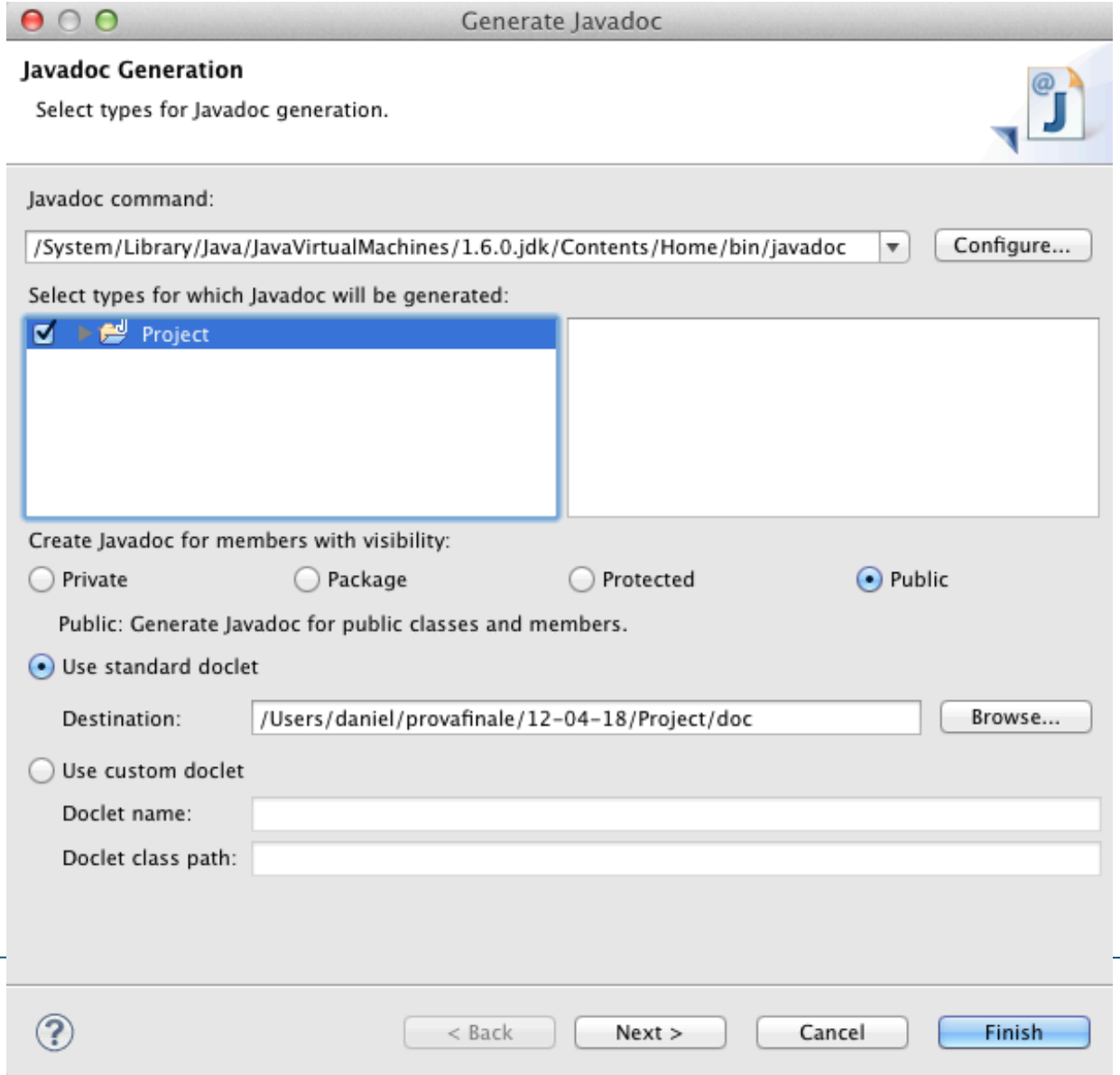
# Esempio documentazione metodi nella classe ContoCorrente (2)

```
/**
 * Restituisco il saldo residuo in questo
 * conto corrente.
 *
 * @return Saldo residuo
 */
public int saldo() {
    return saldo;
}
```

# Generazione Javadoc con Eclipse (1)



# Generazione Javadoc con Eclipse (2)



# Generazione Javadoc con Eclipse (3)

**polimi.provafinale**

## Class ContoCorrente

```
java.lang.Object
  └ polimi.provafinale.ContoCorrente
```

---

```
public class ContoCorrente
extends java.lang.Object
```

La classe ContoCorrente rappresenta un conto corrente bancario. Lo stato della classe consiste nel saldo del conto corrente bancario, il cui valore e' un intero inizialmente nullo. La classe permette di interrogare il saldo residuo e di effettuare prelievi e depositi.

**Version:**
1.0
**Author:**
Daniel J. Dubois

---

### Constructor Summary

| |
|---|
| `ContoCorrente()` |

### Method Summary

| | |
|---|---|
| void | `deposita(int soldi)`<br>Deposito dei soldi nel conto corrente. |
| void | `preleva(int soldi)`<br>Prelevo dei soldi dal conto corrente. |
| int | `saldo()`<br>Restituisco il saldo residuo in questo conto corrente. |

# Generazione JavaDoc: altri metodi

- E' anche possibile generare i file HTML con la documentazione utilizzando:
  - usando direttamente javadoc da linea di comando
  - automaticamente con Maven

# Riferimenti

- http://docs.oracle.com/javase/1.5.0/docs/guide/javadoc