

# Testing di unità: JUnit

# Riferimenti e risorse

**Using JUnit in Eclipse, <http://www.cs.umanitoba.ca/~eclipse/10-JUnit.pdf>**

**An Introduction to JUnit,**

**<http://www.cs.toronto.edu/~cosmin/TA/2003/csc444h/tut/tut3.pdf>**

**JUnit Testing Utility Tutorial,**

**<http://supportweb.cs.bham.ac.uk/documentation/tutorials/docsystem/build/tutorials/junit/junit.html>**

**Tools for automated software testing,**

**[http://www.elet.polimi.it/upload/picco/Teaching/softeng/slides/test\\_tools.pdf](http://www.elet.polimi.it/upload/picco/Teaching/softeng/slides/test_tools.pdf)**

**Introduzione a Test-First Design e JUnit,**

**<http://www.lta.disco.unimib.it/didattica/Progettazione/lucidi/e09-testing+introJUnit.pdf>**

**JUnit Testing Utility Tutorial,**

**<http://supportweb.cs.bham.ac.uk/documentation/tutorials/docsystem/build/tutorials/junit/junit.html>**

**Sito ufficiale di JUnit, <http://www.junit.org/index.htm>**

**Progetto sourceforge di JUnit, <http://junit.sourceforge.net/>**

# Test di unità

- **L'Unità per definizione è la parte più piccola del software che si intende testare**
  - Il concetto di Unità coincide con il concetto di modulo utilizzato in progettazione
- **Unità possono essere:**
  - Le funzioni
  - I metodi
  - Le classi
  - I package (note le loro interfacce)
  - Gli interi sistemi
- **Il testing di unità può essere svolto:**
  - In modalità black box, se è nota solo la specifica dell'unità (ingressi/uscite/funzione realizzata)
  - In modalità white box se si conosce anche come l'unità è realizzata e si vuole sfruttare tale conoscenza per valutare anche la correttezza parziale dell'unità

# Scrittura dei test di unità

**Il testing a livello di unità dei comportamenti di una classe dovrebbe essere progettato ed eseguito dallo sviluppatore della classe, contemporaneamente allo sviluppo stesso della classe**

- Di questa opinione sono in particolare Erich Gamma e Kent Beck, meglio conosciuti come gli autori dei Design Pattern e dell'eXtreme Programming (che verrà presentato nella lezione dedicata ai cicli di vita)

## **Vantaggi:**

- Lo sviluppatore conosce esattamente le responsabilità della classe che ha sviluppato e i risultati che da essa si attende
- Lo sviluppatore conosce esattamente come si accede alla classe, ad esempio:
  - *Quali precondizioni devono essere poste prima di poter eseguire un caso di test;*
  - *Quali postcondizioni sui valori dello stato degli oggetti devono verificarsi*

## **Svantaggi:**

- Lo sviluppatore tende a difendere il suo lavoro ... troverà meno errori di quanto possa fare un tester!

# Testing Automation

**Se la progettazione dei casi di test é un lavoro duro e difficile, l'esecuzione dei casi di test é un lavoro noioso e gramo!**

**L'automatizzazione dell'esecuzione dei casi di test porta innumerevoli vantaggi:**

- Tempo risparmiato (nell'esecuzione dei test)
- Affidabilità dei test (non c'è rischio di errore umano nell'esecuzione dei test)
  - *Si può migliorare l'efficacia a scapito dell'efficienza sfruttando il fatto che l'esecuzione dei test è poco costosa*
- Riutilizzo (parziale) dei test a seguito di modifiche nella classe

# Automatizzare test di unità: Testing basato su main

**Scrivere un metodo di prova (“main”) in ogni classe contenente del codice in grado di testare i suoi comportamenti**

- Problemi
  - *Tale codice verrà distribuito anche nel prodotto finale, appesantendolo*
  - *Come strutturare i test case? Come eseguirli complessivamente?*

**Cerchiamo un approccio sistematico**

- Che separi il codice di test da quello della classe
- Che supporti la strutturazione dei casi di test in test suite
- Che consenta l’esecuzione complessiva di un’intera test suite
- Che fornisca un output separato dall’output dovuto all’esecuzione della classe

# Famiglia X-Unit

**La soluzione alle problematiche precedenti é data dai framework della famiglia X-Unit:**

- JUnit (Java)
  - *È il capostipite; fu sviluppato originariamente da Erich Gamma and Kent Beck*
- CppUnit (C++)
- csUnit (C#)
- NUnit (.NET framework)
- HttpUnit (Web Application)

# JUnit

**JUnit é un framework (in pratica consiste di un archivio .jar contenente una collezione di classi) che permette la scrittura di test in maniera ripetibile**

- **Plug-ins che supportano il processo di scrittura ed esecuzione dei test JUnit su classi Java sono previsti da alcuni ambienti di sviluppo**
  - in particolare mostreremo il funzionamento del plug-in JUnit di Eclipse



## Plug-in di Eclipse per JUnit

**Eclipse é dotato di plug-ins, di pubblico dominio, che supportano tutte le operazioni legate al testing di unità con JUnit, CppUnit, etc.. In particolare, essi forniscono dei wizard per:**

- Creare classi contenenti test cases
- Automatizzare l'esecuzione di tutti i test cases
- Mostrare i risultati dell'esecuzione dei casi di test
- Organizzare i test cases in test suites

# Componenti di un test Junit (versione 3)

## Classi di test

- Una classe di test Junit deve estendere (ereditare da) una classe denominata **TestCase** della libreria

## Metodi di test

- Ogni metodo di test deve avere un nome che inizia con la parola *test* (in minuscolo)
- Possono essere realizzati un numero qualsiasi di metodi di test

# Componenti di un test Junit (versione 3)

## Una classe di test, contiene anche:

- Un metodo *setup()* che viene eseguito prima dell'esecuzione di ogni test
  - *Utile per settare precondizioni comuni a più di un caso di test*
- Un metodo *teardown()* che viene eseguito dopo ogni caso di test
  - *Utile per resettare le postcondizioni*

# Struttura di un metodo di test

## Inizializzazione precondizioni

- Limitatamente alle precondizioni tipiche del singolo caso di test, le altre potrebbero essere nel *setup*

## Inserimento valori di input

- Tramite chiamate a metodi set oppure tramite assegnazione di valori ad attributi pubblici

## Codice di test

- Esecuzione del metodo da testare con gli eventuali parametri relativi a quel caso di test

## Valutazione delle asserzioni

- Controllo di espressioni booleani (*asserzioni*) che devono risultare vere se il test dà esito positivo, ovvero se i dati di uscita e/o le postcondizioni riscontrati sono diversi da quelli attesi

# Trasformazione di un test in codice JUnit

## Precondizioni

- Tramite il metodo setup vengono eseguite delle operazioni mirante a far diventare vere le precondizioni. Al termine del metodo vengono poste delle asserzioni che valutano le precondizioni: se non fossero verificate, i test non vengono eseguiti

## Input

- Tramite settaggi diretti di attributi oppure chiamate a metodi set (che si suppongono corretti e senza necessità di essere testati)

## Test

- Esecuzione del metodo da testare con gli eventuali ulteriori parametri di input relativi a quel caso di test

## Output

- Controllo di espressioni booleane relative alla coincidenza dei valori di output ottenuti con quelli osservati (*asserzioni*)

## Postcondizioni

- Valutazione di espressioni booleane (*asserzioni*) relative alle postcondizioni

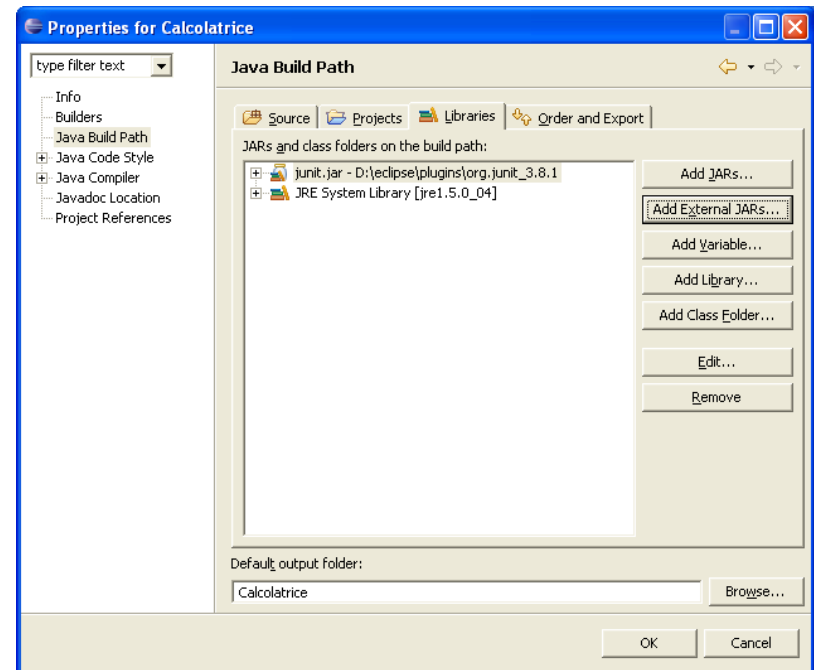
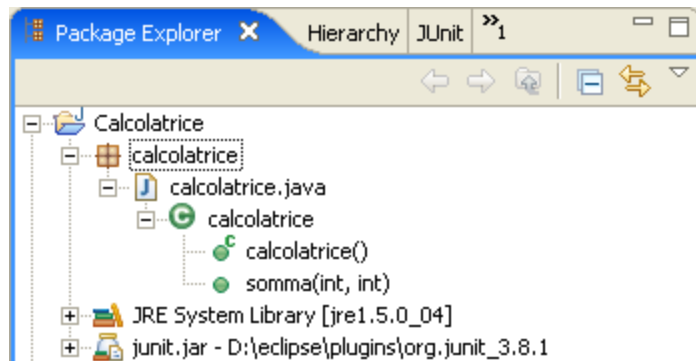
# Breve Tutorial

**Creiamo un nuovo progetto Eclipse,  
con un package (*calcolatrice*)**

**All'interno di questo package  
generiamo una classe *Calcolatrice*:**

```
package calcolatrice;  
  
public class calcolatrice {  
    public calcolatrice(){};  
  
    public int somma (int a, int b)  
    {return a+b;}  
  
}
```

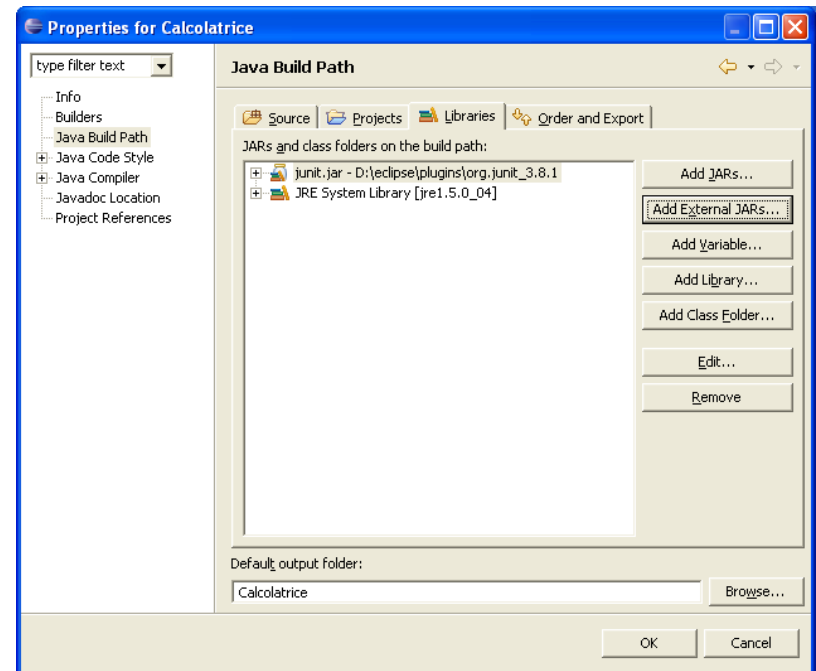
# Scrittura dei test



# Inserimento di JUnit nel progetto

- Nelle proprietà del progetto, aggiungiamo JUnit.jar tra le librerie previste nel Classpath del progetto

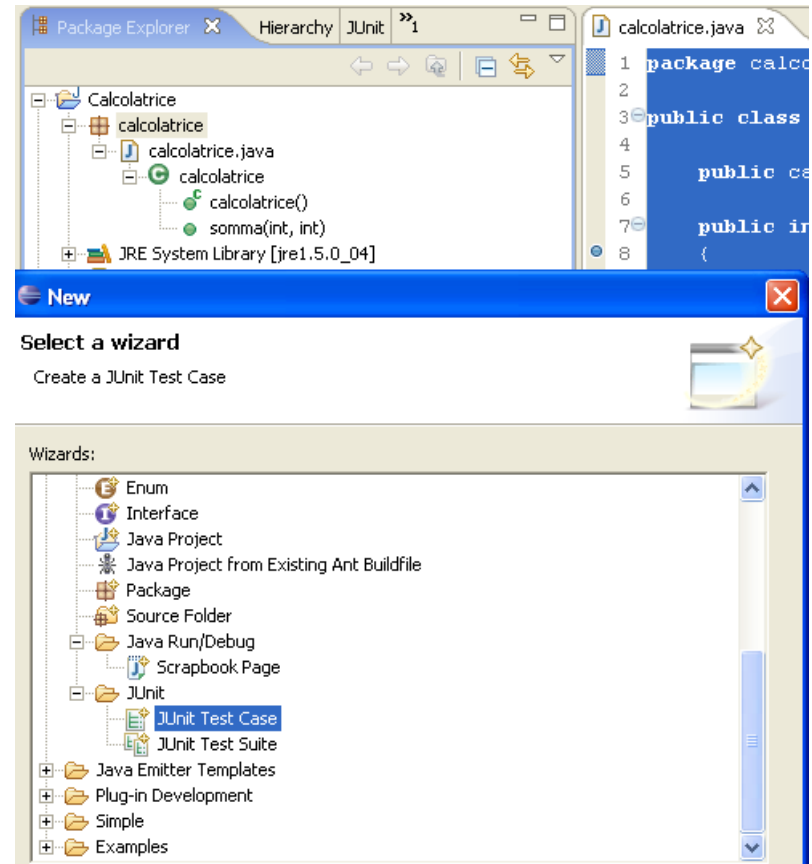
- Add External Jars ...  
Una copia di junit.jar dovrebbe trovarsi tra i plug-in di Eclipse; in alternativa la si può scaricare da [www.junit.org](http://www.junit.org)





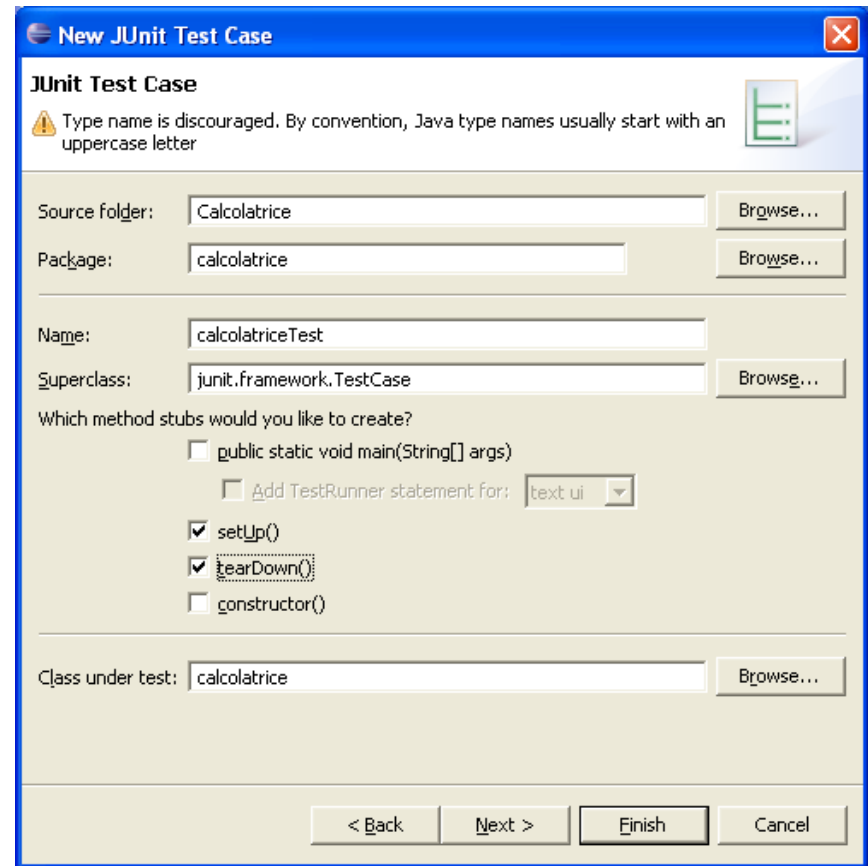
# Creazione di una classe test

1. **All'interno dello STESSO package della classe da testare, creare una nuova classe, scegliendo la tipologia JUnit Test Case**



# Generazione della classe dei test

1. Il Wizard di Eclipse ci permette di indicare il nome della classe che si vuole testare (*calcolatrice*), il nome della classe da generare (abbiamo usato per convenzione il nome *calcolatriceTest*), gli ulteriori metodi da aggiungere (abbiamo selezionato *setUp* e *tearDown*)



# Scrittura della classe test

- Il Wizard ha generato una classe che eredita dalla classe *TestCase*, cuore della libreria JUnit
- Il metodo *setUp()* può essere completato, accodando tutte quelle operazioni da effettuare preliminarmente a qualsiasi test che sarà descritto in questa classe;
- Il metodo *tearDown()* conterrà il codice relativo a tutte le operazioni comuni da effettuare dopo l'esecuzione di ogni test di questa classe (ad esempio per ripristinare lo stato della classe prima dell'esecuzione del prossimo test

```
package calcolatrice;

import junit.framework.TestCase;

public class calcolatriceTest extends TestCase {
    protected void setUp() throws Exception {
        super.setUp();
    }

    protected void tearDown() throws Exception {
        super.tearDown();
    }

    /* Test method for 'calcolatrice.calcolatrice.somma(int, int)' */
    public void testSomma() {

    }
}
```

# Scrittura di un caso di test

- Scriviamo un metodo *testSomma* che rappresenti un caso di test per il metodo *Somma*;
  - Siccome il metodo appartiene ad una classe *calcolatriceTest* nello stesso package della classe da testare, il metodo test può istanziare oggetti della classe ed accedere ai suoi metodi

```
public void testSomma() {  
  
    calcolatrice c=new calcolatrice();  
    int a=5,b=7;  
    int s=c.somma(a,b);  
    assertEquals("Somma non corretta!",12,s);  
}
```

Il metodo `assertEquals` verifica se `s` (valore ottenuto dall'esecuzione del metodo `somma`) è uguale a 12 (valore atteso); in caso contrario conta questo fatto come una *failure* e genera il messaggio di errore indicato

# Esempio più generale

L'asserzione nel Before corrisponde alla precondition: se non è verificata il test non viene eseguito

L'asserzione nell'After corrisponde alla postcondizione: se non è verificata, il test ha rilevato un malfunzionamento

L'asserzione nel Test corrisponde all'oracolo: se non è verificata, il test ha rilevato un malfunzionamento

Da notare che il test viene eseguito solo se la precondition è verificata, mentre l'After è eseguito in ogni caso.

```
public class calcolatriceTest {  
    private Calcolatrice c;  
  
    @Before  
    public void setUp() throws Exception {  
        c=new Calcolatrice();  
        assertNotNull(c);  
    }  
  
    @After  
    public void tearDown() throws Exception {  
        c=null;  
        assertNull(c);  
    }  
  
    @Test  
    public void testSomma() {  
        assertEquals("Somma Sbagliata",12,c.somma(5,7));  
    }  
}
```

# Assertzioni

## Assertzione

- affermazione che può essere vera o falsa

**I risultati attesi sono documentati con delle *asserzioni* esplicite, non con delle stampe che comunque richiedono dispendiose ispezioni visuali dei risultati**

## Se l'asserzione è

- vera: il test è andato a buon fine
- falsa: il test è fallito ed il codice testato non si comporta come atteso, quindi c'è un errore a tempo dinamico

**Le asserzioni sono utilizzate sia per verificare gli oracoli che le postcondizioni**

- Le asserzioni potrebbero essere utilizzate anche per verificare le precondizioni: se la precondizione fallisce, il test non viene proprio eseguito (e viene riportato come fallito)

# Assertzioni

## Se una asserzione non è vera il test-case fallisce

- `assertNotNull()`: afferma che il suo argomento è nullo (fallisce se non lo è)
- `assertEquals()`: afferma che il suo secondo argomento è `equals()` al primo argomento, ovvero al valore atteso
- molte altre varianti
  - `assertNotNull()`
  - `assertTrue()`
  - `assertFalse()`
  - `assertSame()`
  - ...

## assertEquals()

`assertEquals(Object expected, Object actual)`

**Va a buon fine se e solo se `expected.equals(actual)` restituisce `true`**

**`expected` è il valore atteso**

**`actual` è il valore effettivamente rilevato**

`assertEquals(String message, Object expected, Object actual)`

**In questa variante si specifica un messaggio che il *runner* stampa in caso di fallimento dell'asserzione: molto utile per localizzare immediatamente l'asserzione che causa il fallimento di un test-case ed avere i primi messaggi diagnostici**



# Principali asserzioni

**static void assertEquals(boolean expected, boolean actual)**

- Asserts that two booleans are equal.

**static void assertEquals(int expected, int actual)**

- Asserts that two ints are equal.

**static void assertEquals(java.lang.String expected, java.lang.String actual)**

- Asserts that two Strings are equal.

**static void assertFalse(boolean condition)**

- Asserts that a condition is false.

**static void assertTrue(boolean condition)**

- Asserts that a condition is true.

**static void assertNull(java.lang.Object object)**

- Asserts that an object is null.

**static void fail()**

- Fails a test with no message.

**static void fail(java.lang.String message)**

- Fails a test with the given message.

...

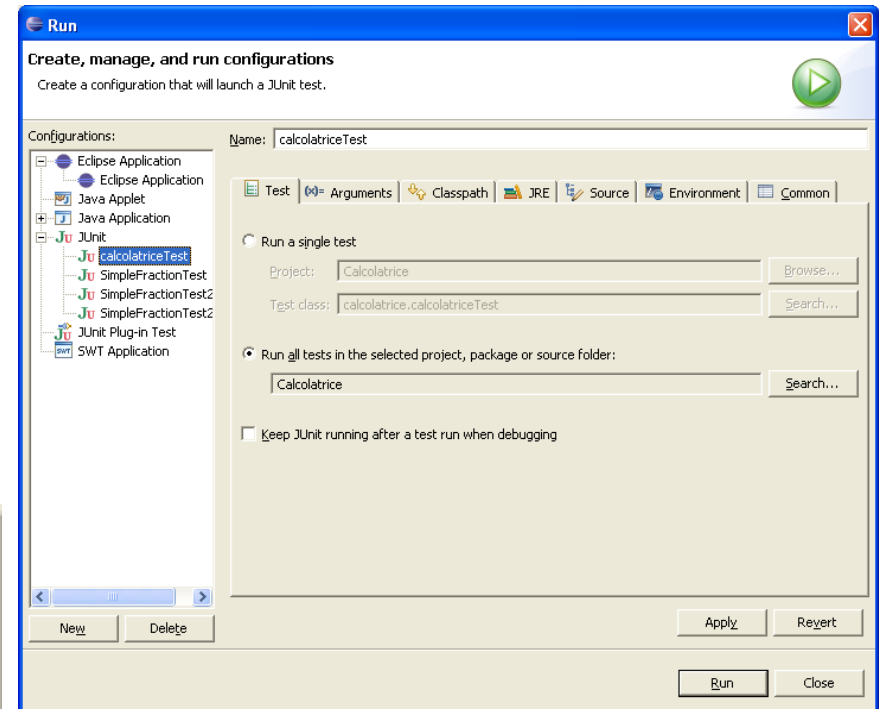
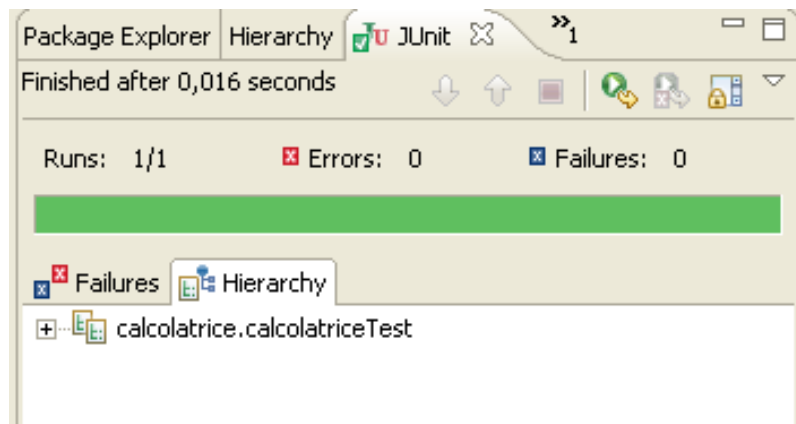
Elenco completo disponibile a:

[http://junit.sourceforge.net/javadoc\\_40/org/junit/Assert.html](http://junit.sourceforge.net/javadoc_40/org/junit/Assert.html)

# Esecuzione dei casi di test

**Per eseguire i test, basta seguire una procedura simile a quella per eseguire applicazioni**

...



# Test rilevanti errori ...

```
public double divisione (int a, int b)
{return a/b;}
```

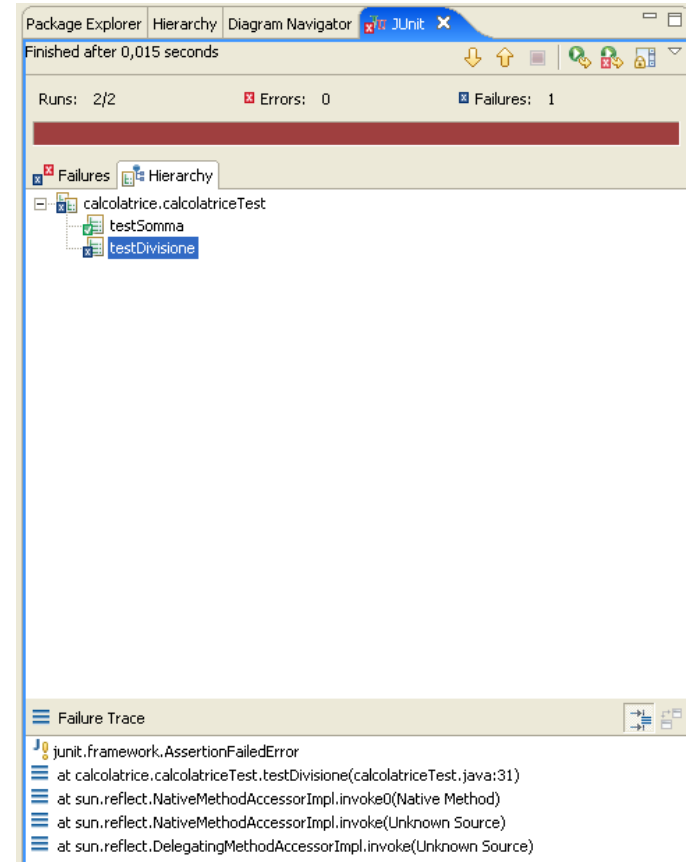
```
public void testDivisione(){

    calcolatrice c=new calcolatrice();
    int a=15,b=2;
    double s=c.divisione(a,b);
    assertTrue(s==7.5);
}
```

In realtà il metodo divisione restituisce la divisione intera ...

Grazie a JUnit possiamo prontamente trovare il rigo con l'asserzione errata e avviare il debugging ...

Quando pensiamo di aver corretto l'errore rieseguiamo il test ...



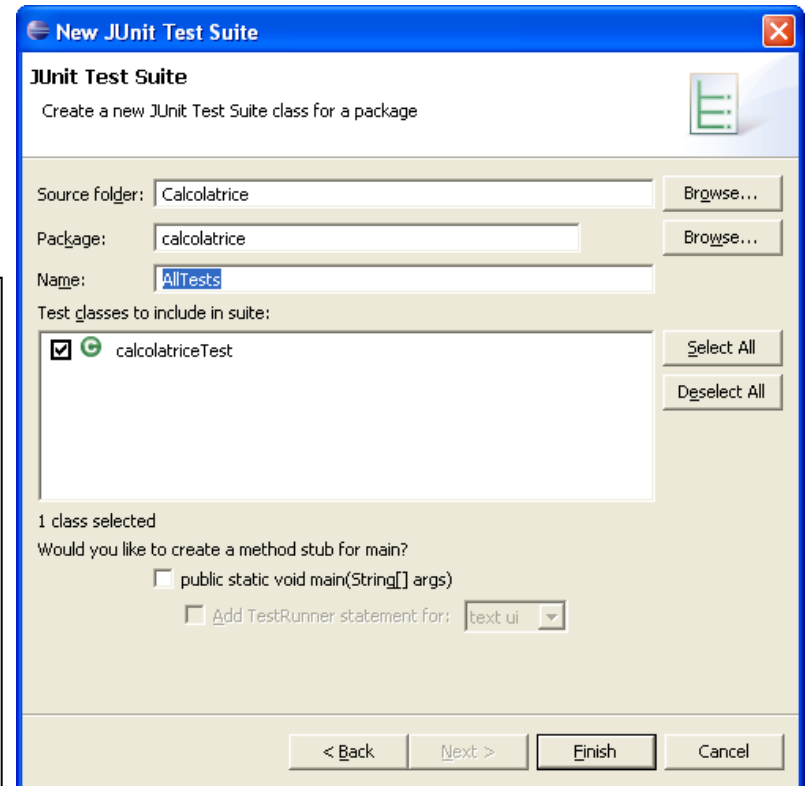
# Raggruppare test cases in test suite

- Ulteriori classi di test possono essere aggiunte in seguito modificando il codice generato ...

```
package calcolatrice;

import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTests {
    public static Test suite() {
        TestSuite suite = new TestSuite("Test for calcolatrice");
        //$JUnit-BEGIN$
        suite.addTestSuite(calcolatriceTest.class);
        //$JUnit-END$
        return suite;
    }
}
```



# JUnit versione 4

**La versione 4 del framework presenta alcune significative differenze rispetto alle versioni precedenti. Dovuto all'inclusione delle *Annotazioni* introdotte con la versione Java 5.0**

**Il meccanismo delle annotazioni permette di introdurre, direttamente nel codice, delle meta-informazioni.**

- Dichiarazioni circa il codice stesso

**Meccanismo del tutto equivalente a quello utilizzato dal parser JavaDoc per generare la documentazione.**

- Parola chiave preceduta dal carattere @

# JUnit 4 differenze con le versioni precedenti

- **Non occorre importare il package: *junit.framework.TestCase***
- **Bisogna importare le annotazioni di test (*org.junit.Test*)**
- **Non è più necessario che le classi di test estendano la classe *TestCase***
- **Non è obbligatorio aggiungere il suffisso *test* ai metodi di test: questi sono evidenziati attraverso l'utilizzo dell'annotazione *@Test***
- **È comunque possibile che il framework JUnit esegua correttamente alcune classi di test scritte per una versione precedente.**

# Annotazioni JUnit 4

## **È possibile eseguire alcune azioni**

- Prima e dopo l'esecuzione di tutti i test di una classe di test
- Prima e dopo l'esecuzione di un singolo test

## **@BeforeClass marca un metodo statico che viene eseguito prima di tutti i test contenuti in una classe di test**

- Può essere utilizzata per settare precondizioni comuni a tutti i test

## **@Before marca un metodo che viene eseguito prima del metodo di test**

- Setta precondizioni di quel singolo test

## **@After marca un metodo che viene eseguito dopo il metodo di test**

- Può essere utilizzata per ripristinare lo stato dell'applicazione dopo l'esecuzione di un test, per consentire l'esecuzione automatica di un test successivo

## **@AfterClass marca un metodo statico che viene eseguito dopo tutti i test contenuti in una classe di test**

# Test Suite

**Meccanismo usato per raggruppare logicamente dei test ed eseguirli assieme**

**L'annotazione @SuiteClasses raggruppa una lista di classi, passate come parametro, in una test suite**

**L'annotazione @RunWith permette di impostare diversi esecutori di test**

**Di solito si usa Suite, già incluso nel framework**



# Breve Tutorial Junit 4

**Creiamo un nuovo progetto Eclipse, con un package (Calendario)  
All'interno di questo package generiamo una classe *calendario*:**

```
package Calendario;

public class Calendario {

    public static String calend(int d, int m, int a)
    {
        if (m<=2)
        {
            m = m + 12;
            a--;
        };
        int f1 = a / 4;
        int f2 = a / 100;
        int f3 = a / 400;
        int f4 = (int) (2 * m + (.6 * (m + 1)));
        int f5 = a + d + 1;
        int x = f1 - f2 + f3 + f4 + f5;
        int k = x / 7;
        int n = x - k * 7;
```

```
        if (n==1) return "Lunedì";
        else if (n==2) return "Martedì";
        else if (n==3) return "Mercoledì";
        else if (n==4) return "Giovedì";
        else if (n==5) return "Venerdì";
        else if (n==6) return "Sabato";
        else if (n==0) return "Domenica";
        else return "Errore";
    }
}
```





# Generazione classi di test

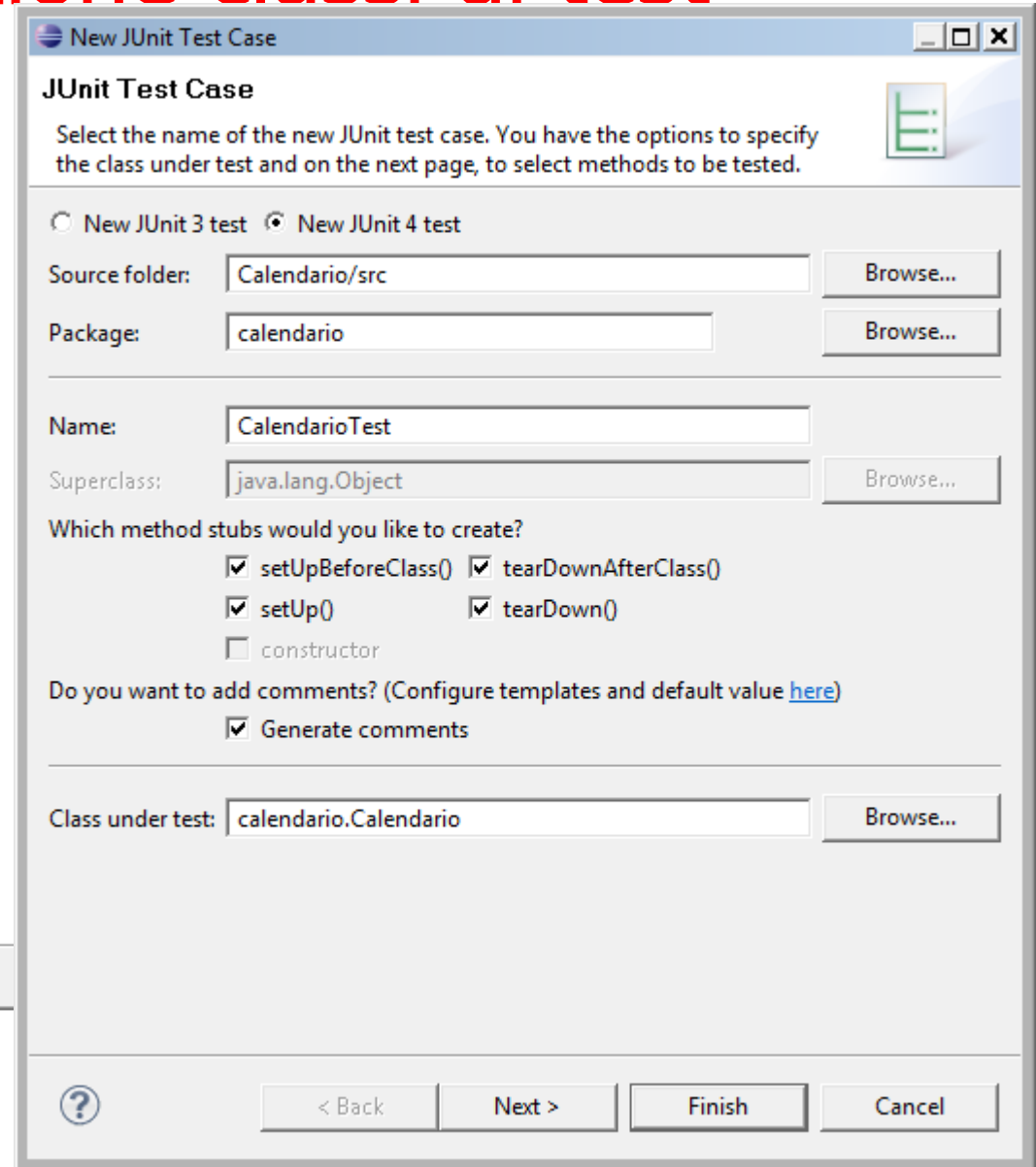
**Creiamo una  
classe  
contenitore  
di test case  
dal wizard  
New Junit  
Test Case**

## Test Methods

Select methods for which test method stubs should be created.

### Available methods:

- ☒  **Calendario**
  - ☒  **calend(int, int, int)**
  - ☐  **main(String[])**
- ☐  **Object**



**New 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

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:

# Codice di test generato (JUnit 4)

```
package calendario;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

/**
 * @author Porfirio
 *
 */
public class CalendarioTest {

    @BeforeClass
    public static void setUpBeforeClass()
        throws Exception {
    }

    @AfterClass
    public static void
        tearDownAfterClass() throws
        Exception {
    }
}
```

```
@Before
public void setUp() throws Exception {
}

@After
public void tearDown() throws Exception {
}

/**
 * Test method for {@link
 * calendario.Calendario#calend(int, int,
 * int)}.
 */
@Test
public void testCalend() {
    fail("Not yet implemented");
}
}
```

@Test è un'annotazione per marcare i metodi che si considerano di Test  
Non è (più) necessario ma è buona norma usare 'test' come prefisso del nome dei metodi di test (obbligatorio in JUnit 3)

# JUnit 4 Classi di Test

```
package mioProgetto;
import static org.junit.Assert.*;
import org.junit.Test;
public class miaClasseTest {
    @Test
    public void testprimaClasseSottoTest() {
        ...Codice

        assertEquals(valoreAtteso, valoreDaEsaminare);
    }
    @Test
    public void testsecondaClasseSottoTest() {
        ...Codice

        assertNull(valoreCheDeveEssereNullo);
    }
    ...
}
```

import di classi ed  
*annotazioni* JUnit

nome

Annotazione di metodo come test-case

Assertzione

# Test unitario in pratica

```
public void testMetodoDaTestare() {  
    miaClasse mioOggetto = new miaClasse("par1", "par2");  
    assertEquals("par1", mioOggetto.getNome());  
    assertEquals("par2", mioOggetto.getParametro());  
}
```

**mettere un “frammento” del sistema in un stato noto**

- il frammento comprende un solo oggetto `mioOggetto`

**inviare una serie di messaggi noti**

- nell’esempio solo la costruzione dell’oggetto, in generale ci potrebbero essere altre invocazioni di metodo sullo stesso

**controllare che alla fine il sistema si trovi nello stato atteso**

- tramite le asserzioni si controlla che il nome ed il parametro del comando siano quelli attesi

# Compilare i Test

**Tutto sarà semplificato con Eclipse**

**Nel classpath ci devono essere le librerie di JUnit (junit-4.4.jar).**

**Supponiamo che queste siano nella directory c:\java\lib\ :**

```
javac -cp ".;c:\java\lib\junit-4.4.jar;c:\src" miaClasseTest.java
```

# Eseguire i test

- Per eseguire dei test è necessario usare una classe runner che trova ed esegue i test-case
- JUnit 4.x include come Runner
  - `org.junit.runner.JUnitCore`accetta come argomento una o più classi di test

```
$java -cp ".;c:\java\lib\junit-4.4.jar;c:\src\mioProgetto"  
      org.junit.runner.JUnitCore mioProgetto.miaClasseTest
```

```
JUnit version 4.4
```

```
.....  
Time: 0,066
```

← Un puntino per ogni test-case andato a buon fine

```
OK (8 tests)
```

# Osservazioni

***Si è utilizzato, finora, JUnit solo e soltanto per il testing di unità di singoli metodi***

- ***Può essere utile anche nel testing black box di sistemi completi***
- ***In seguito vedremo come possa supportare problematiche relative al testing di integrazione***

**JUnit supporta il testing ma anche il debugging**

- L'utilizzo di molte asserzioni può, però, portare ad indicazioni dettagliate sulle ragioni del successo del test case
- I test case possono essere ripetuti uno step alla volta

**Si è data per scontata la correttezza del codice delle classi di test ... se avessimo voluto esserne sicuri al massimo avremmo potuto fare il test di unità delle classi di test stesse (ma il problema si sarebbe riproposto ricorsivamente!)**

- Il codice delle classi di test è comunque estremamente lineare e ripetitivo: la possibilità di sbagliare è ridotta!
- La generazione del codice delle classi di test è automatizzabile in alcuni

---

contesti



# Limiti di JUnit

**Se una classe é in associazione/dependency con altre e JUnit rileva l'errore, esso potrebbe dipendere dall'altra classe (se non é stata testata adeguatamente) oppure dall'integrazione ...**

- JUnit é uno strumento che é in grado di risolvere SOLO le problematiche relative al testing di unità ... dà solo qualche utile indicazione rispetto al testing di integrazione!

**Si é dato per scontata la correttezza del codice delle classi di test ... se avessimo voluto esserne sicuri al massimo avremmo potuto fare il test di unità delle classi di test stesse (paradossale!)**

- Il codice delle classi di test é comunque estremamente lineare e ripetitivo: la possibilità di sbagliare é ridotta!

# Limiti di JUnit

- Un metodo di test JUnit non può:
  - Accedere a metodi e attributi privati
    - A meno che non sia nella stessa classe da testare
  - Accedere a metodi e attributi con visibilità package,
    - a meno che il metodo di test non sia nello stesso package dell'attributo/metodo da testare
  - Accedere a metodi e attributi pubblici,
    - a meno che il metodo di test non sia nello stesso progetto dell'attributo/metodo
  - Solo con JUnit 4 (con la scomparsa delle classi TestCase e l'introduzione delle annotazioni) è possibile risolvere alcuni di questi problemi

# Usi possibili di JUnit

- Junit può supportare diversi task di testing:
  - Testing di unità black box
    - Nel quale i metodi di test vanno a chiamare le unità (metodi o funzioni) da provare passando parametri e valutando asserzioni sui risultati ottenuti
  - Testing di unità white box
    - Nel quale i metodi di test interagiscono con le unità da testare (metodi, ma anche oggetti) chiamando metodi e valutando asserzioni sia relative agli output che agli attributi e variabili modificate durante l'elaborazione

# Usi possibili di JUnit

- Junit può supportare diversi task di testing:
  - Testing di integrazione
    - Nel quale i metodi di test agiscono da driver che sostituiscono gli altri metodi che chiamano quelli che vogliamo testare
  - Testing della GUI
    - Nel quale i metodi di test vanno ad emulare le interazioni dell'utente col sistema
  - Testing di sistema / funzionale
    - Nel quale i metodi di test chiamano direttamente l'interfaccia del sistema
- ...

## Esercizio

**Un programma Java riceve in input una data, composta di giorno (numerico), mese (stringa che può valere gennaio ... dicembre), anno (numerico, compreso tra 1582 e 2100) e restituisce il giorno della settimana corrispondente (1= lunedì, 2= martedì ... 7=domenica)**

**Implementare con Junit le strategie di testing black box**

# Applicazione da testare

**Per esercizio, provare ad eseguire i test previsti sulla funzione riportata qui sotto**

Il codice viene riportato per comodità, ma il testing viene progettato *senza* analizzare il codice sorgente

<pre>int calend(int d,string ms,int a) {     int m=0;     if (ms=="gennaio") m=1;     else if (ms=="febbraio") m=2;     else if (ms=="marzo") m=3;     else if (ms=="aprile") m=4;     else if (ms=="maggio") m=5;     else if (ms=="giugno") m=6;     else if (ms=="luglio") m=7;     else if (ms=="agosto") m=8;     else if (ms=="settembre") m=9;     else if (ms=="ottobre") m=10;     else if (ms=="novembre") m=11;     else if (ms=="dicembre") m=12;</pre>	<pre>f (m&lt;=2) {     m = m + 12;     a--; }; int f1 = a / 4; int f2 = a / 100; int f3 = a / 400; int f4 = (int) (2 * m + (.6 * (m + 1))); int f5 = a + d + 1; int x = f1 - f2 + f3 + f4 + f5; int k = x / 7; int n = x - k * 7; return n; };</pre>
---	--

# Esempio

```
package calendario;

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

public class TestBlackBox {

    @Test
    public void testCalend1() {
        assertEquals(Calendaro.calend(13,"aprile",2011),"Mercoledi");
    }

    @Test
    public void testCalend2() {
        assertEquals(Calendaro.calend(0,"aprile",2011),"Errore");
    }

    @Test
    public void testCalend3() {
        assertEquals(Calendaro.calend(-42,"aprile",2011),"Errore");
    }

    ...
}
```