

# Programmazione Orientata agli Oggetti

---

Qualità del Codice:  
Introduzione Unit-Testing



# Sommario

- Errori nel software
  - errori di compilazione vs bug
  - località dei bug
- Testing
  - motivazioni
  - le tre fasi di un test
- Unit-Testing
- Introduzione a JUnit
- Qualità dei test
- Testing Continuo
- TDD

# Software ed Errori

- I primi errori con i quali ci scontriamo di solito sono *errori di sintassi*
  - Ci vengono indicati dal compilatore
- Successivamente incorriamo in *errori logici*
  - Il compilatore non ci può aiutare
  - Sono noti anche come “*bug*” (*bachì*)
- Alcuni errori logici non si manifestano immediatamente
  - Il software è estremamente complesso
  - Anche il software commerciale non è affatto privo di errori

# Errori di Compilazione

- Il compilatore ci dà indicazioni precise e molto utili a correggere l'errore
- Il messaggio di errore del compilatore  
**VA LETTO E CAPITO**

LINEA DI CODICE IN CUI E' STATO RISCONTRATO L'ERRORE

ERRORE RISCONTRATO

```
Diadia.java:27:invalid method declaration;  
return type required  
private creaStanze() {  
                ^
```

**1 error**

# Errori a Tempo di Esecuzione

- Anche in questo caso abbiamo informazioni molto precise (dalla macchina virtuale)

```
Exception in thread "main"  
java.lang.NullPointerException
```

```
    at Diadia.vaiNellaStanza(Gioco.java:176)  
    at Diadia.processaComando(Gioco.java:117)  
    at Diadia.gioca(Gioco.java:71)  
    at Diadia.main(Gioco.java:209)
```

# Motivazioni del Testing

- I programmi sono descrizioni “statiche” a cui possono corrispondere molteplici esecuzioni “dinamiche”
- I compilatori moderni sono in grado di indicare esattamente posizione e motivo degli errori di compilazione
  - addirittura già mentre si scrive! (compilazione *incrementale*)
- Al contrario i compilatori NON possono prevedere come evolverà l'esecuzione di un programma e non sono in grado di individuare gli errori dei programmatori (né possono sapere cosa intendessero esprimere con il proprio codice)
- In sintesi:
  - il compilatore ci aiuta sugli aspetti statici (ad es. analizzando i tipi)
  - il compilatore non dice nulla di nuovo sugli aspetti dinamici (più di quanto non sia già implicato dagli aspetti statici)

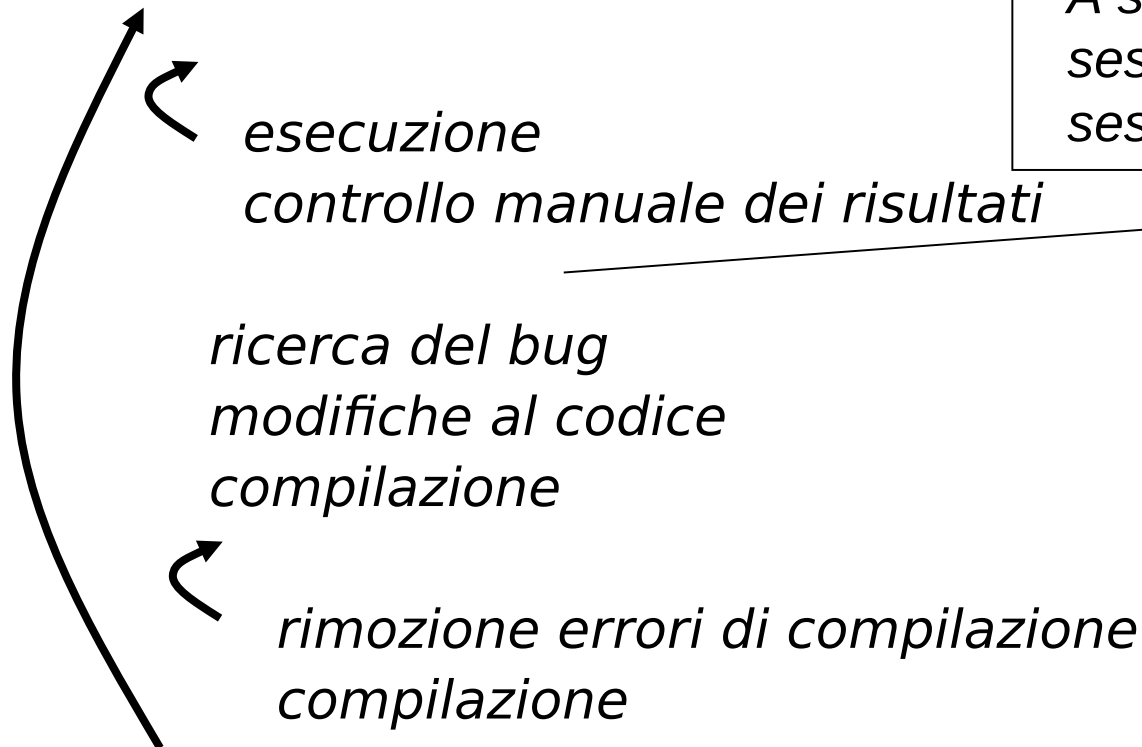
# I Bug

- I bug sono errori nell'evoluzione dinamica di un programma su cui il compilatore non ha potuto prevedere e dire nulla
- Il debugging è completamente a carico del programmatore
- Il costo di debugging è ritenuto di gran lunga la componente principale nel costo dei moderni progetti software



# Ciclo di Debugging

- Come si effettua il debugging di un programma che compila? Con estenuanti cicli:

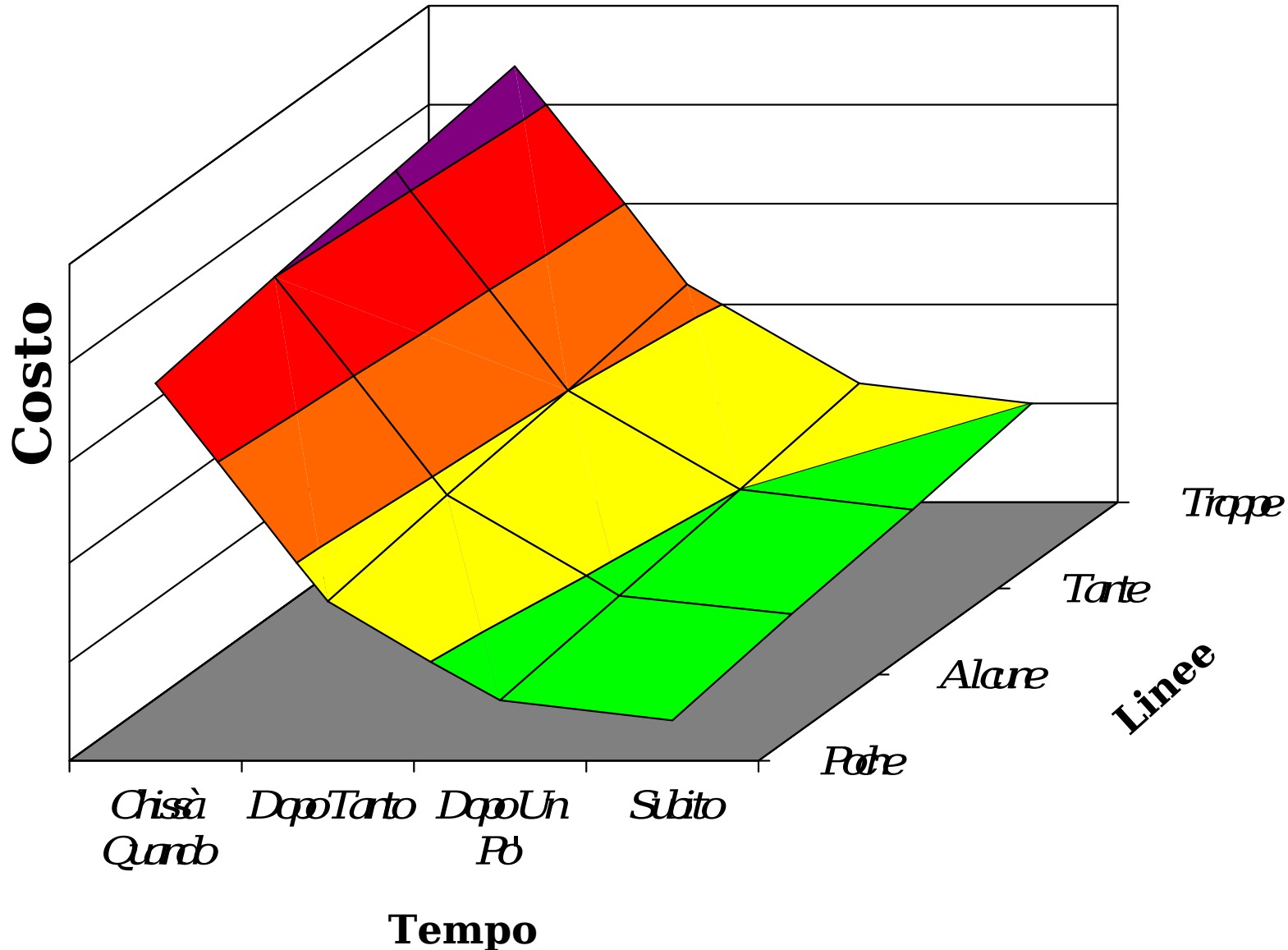


*A sua volta può richiedere:  
sessioni di tracing/logging  
sessioni con il debugger*

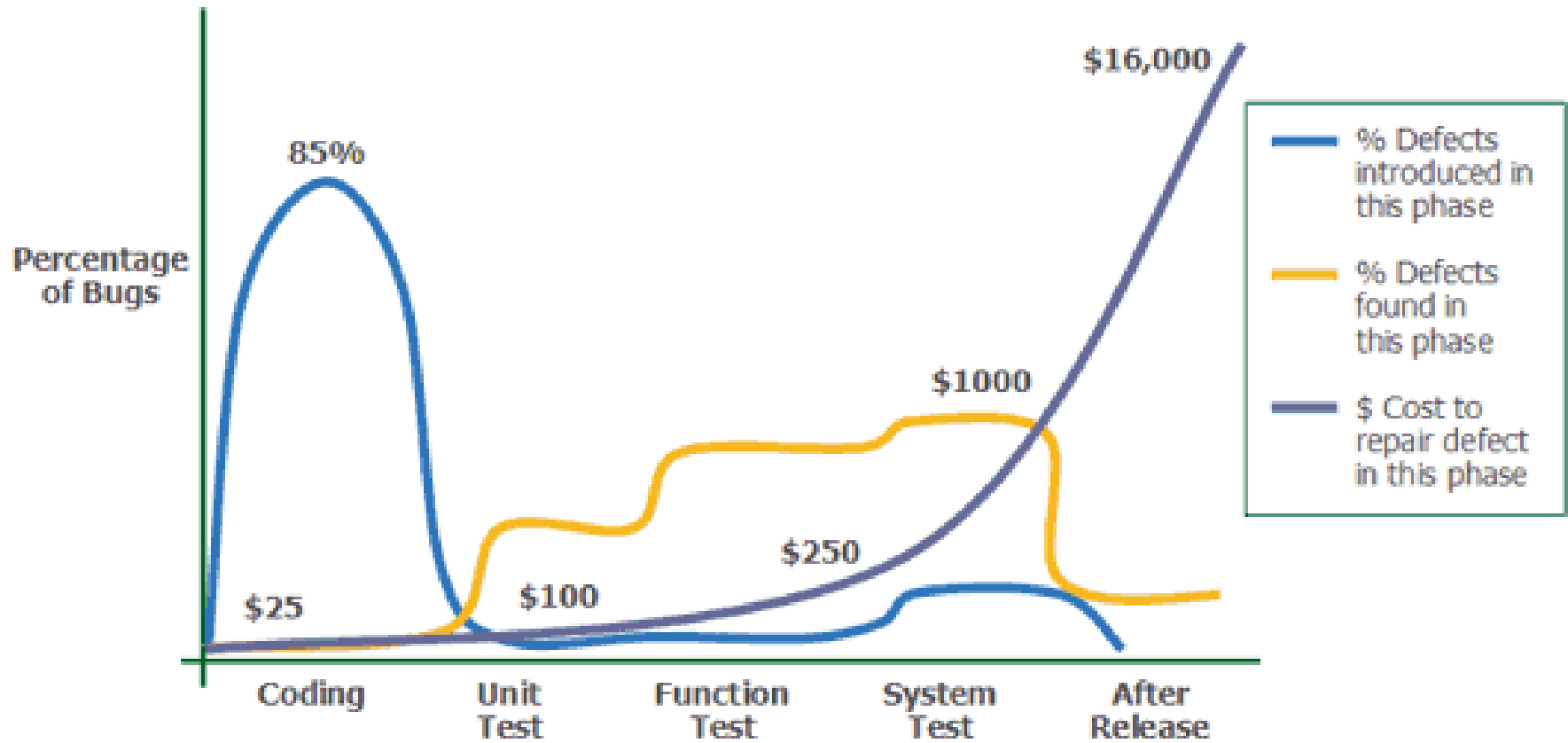
# Costo del Debugging

- Debugging del codice: operazione molto costosa (nonostante gli ausili dell'IDE)
- E' noto che il costo della correzione di bug dipende da almeno due grandezze che ne determinano la *località* :
  - le “dimensioni” del contesto
    - ✓ numero di linee di codice in cui il bug può annidarsi
  - il “tempo” che il bug impiega per manifestarsi
    - ✓ misura temporale di quanto dista la causa del bug (durante un'esecuzione del codice) ed il rilevamento dei suoi effetti

# Costo di un Bug e “Località”



# Tipologie di Test



# Le Tre Fasi di un Test

- Tutte le tipologie di test di un qualsiasi sistema prevedono la costruzione di uno *scenario di testing* che si articola sempre in tre fasi strettamente sequenziali
  1. mettere il sistema in un stato iniziale ben noto
  2. inviare una serie di sollecitazioni
  3. controllare che alla fine il sistema si trovi nello stato atteso
- I test possono fallire od avere successo

# Obiettivi del Testing

- Se ben progettati e mantenuti, i test aiutano a confinare i bug nella “zona verde”, ovvero con spiccata località
  - i bug si manifestano immediatamente e palesemente
  - la ricerca del bug è confinabile in poche linee facilmente localizzabili
- Le esecuzioni che palesano un bug non sono mai troppo lunghe e complesse

# Il Valore Aggiunto dal Testing

- Se il test ha successo:
  - si possiede una garanzia sul comportamento dinamico del codice (assenza di bug)
- Se il test non ha successo:
  - il bug dovrebbe risultare facilmente localizzabile nell'unità di codice sollecitata dal test
- In *entrambi* i casi c'è un significativo valore aggiunto

# Testing vs Regression

- Se il test
  - funzionava subito prima di effettuare un modifica
  - ma smette di funzionare subito dopo aver effettuato la modifica
- E' altamente probabile che l'errore è stato appena introdotto con la modifica
  - Ricerca «locale» e quindi economica
- E' possibile prevenire la *regressione*



# Esercizio

- Supponiamo di voler testare il metodo **max()** della classe **Sequenza** (quiz di benvenuto al corso)
  - Scriviamo in un documento di testo (.txt) diverse istanze dell'array di interi, per ogni sequenza scriviamo il massimo atteso
  - facciamo girare il programma su ciascuna sequenza e verifichiamo che il risultato sia quello atteso
- Osservazione:
  - possiamo scrivere i test senza preoccuparci dell'algoritmo per il calcolo del massimo (ovviamente è necessario scegliere con cura gli array di test)
- Conseguenza:
  - possiamo scrivere i test prima di scrivere il programma!

# Testo QUIZ

- Scrivere il codice del metodo `massimo()` che deve restituire il più grande valore presente nella variabile di istanza `sequenza`, un array:

```
public class Sequenza {  
    private int[] sequenza;  
  
    public Sequenza(int n){  
        sequenza = new int[n];  
    }  
  
    public int massimo(){  
        // scrivere il codice di questo metodo:  
        // deve restituire il valore piu' grande  
        // presente nell'array sequenza  
    }  
  
    public void setElemento(int indice, int valore) {  
        sequenza[indice] = valore;  
    }  
}
```

# Codice di Test

- Nella pratica, accanto al *codice di produzione* si sviluppa sempre del *codice di test*
- Unico motivo di esistere del codice di test è quello di verificare la correttezza a tempo di esecuzione del codice principale
- Il codice di test accompagna e supporta lo sviluppo del codice di produzione ma non fa parte del codice consegnato a fine progetto

# Test Unitari Automatici

- Esistono diverse tipologie di test
- Nostra attenzione è limitata ad una in particolare: *unit-testing automatico*
  - test che si focalizzano su *frammenti (unità)* del sistema
  - senza alcun intervento umano (tranne la richiesta di esecuzione)
- Praticamente i test unitari si codificano nel medesimo linguaggio di programmazione utilizzato per lo sviluppo (Java)

# Test Unitari - Unit Testing

- Test su *frammenti* di un sistema piuttosto che sull'intero sistema
- Concettualmente un test unitario si articola in questi passi
  - 1) mettere uno o più oggetti da testare in un stato iniziale ben noto
  - 2) invocare i metodi degli oggetti
  - 3) controllare che alla fine gli oggetti si trovino nello stato atteso

# Automazione dei Test

- Abbiamo già eseguito test manuali (cfr. Esercizi precedenti)
  - basta riportare i valori di ingresso ed i valori di output attesi in un documento di testo e verificare che ogni esecuzione produca quanto atteso
- Chiaramente ogni esecuzione manuale richiede uno sforzo sia per inserire l'input che per ispezionare visivamente i risultati
  - ✓ Difficilmente si è disposti a ripetere l'operazione troppe volte
- L'automazione dei test è fondamentale
  - altrimenti viene meno la loro economicità

# Automazione Artigianale

- Una possibile soluzione

- molto artigianale
- automatica

chiarisce il funzionamento dei test unitari

- Scriviamo un programma in cui

- inizializziamo un certo numero di oggetti con sequenze di interi su cui basare il test (*fixture* )
- invochiamo il metodo da testare e verifichiamo che il risultato restituito sia uguale a quello atteso

- Successivamente vedremo un framework (JUnit) che rende l'automazione ancora più spedita>>

# Esempio Soluzione Artigianale (1)

```
public static void main(String[] args){
    Sequenza positivi;
    Sequenza negativi;
    Sequenza negEpos;
    Sequenza negEzero;
    Sequenza inPrimaPos;
    Sequenza inUltimaPos;

    positivi = new Sequenza(5);
    positivi.setElemento(0,1);
    positivi.setElemento(1,5);
    positivi.setElemento(2,8); // MAX!
    positivi.setElemento(3,3);
    positivi.setElemento(4,4);
    negativi = new Sequenza(5);
    negativi.setElemento(0,-6);
    negativi.setElemento(1,-1); // MAX!
    negativi.setElemento(2,-8);
    negativi.setElemento(3,-13);
    negativi.setElemento(4,-10);
```



# Esempio Soluzione Artigianale (2)

```
negEpos = new Sequenza(5);  
negEpos.setElemento(0,100);  
negEpos.setElemento(1,-5);  
negEpos.setElemento(2,-80);  
negEpos.setElemento(3,1000); // MAX!  
negEpos.setElemento(4,10);
```

```
negEzero = new Sequenza(5);  
negEzero.setElemento(0,-1);  
negEzero.setElemento(1,0); // MAX!  
negEzero.setElemento(2,-80);  
negEzero.setElemento(3,-10);  
negEzero.setElemento(4,-10);
```

```
inPrimaPos = new Sequenza(5);  
inPrimaPos.setElemento(0, 1000); // MAX!  
inPrimaPos.setElemento(1, 0);  
inPrimaPos.setElemento(2, 80);  
inPrimaPos.setElemento(3,-10);  
inPrimaPos.setElemento(4,-10);
```

```
inUltimaPos = new Sequenza(5);  
inUltimaPos.setElemento(0, 1);  
inUltimaPos.setElemento(1, 0);  
inUltimaPos.setElemento(2, 80);  
inUltimaPos.setElemento(3,-10);  
inUltimaPos.setElemento(4, 1000); // MAX!
```

# Esempio Soluzione Artigianale (3)

```
boolean esito = true;
esito &= (positivi.massimo() == 8);
System.out.println(positivi.massimo() == 8);
esito &= (negativi.massimo() == -1);
System.out.println(negativi.massimo() == -1);
esito &= (negEpos.massimo() == 1000);
System.out.println(negEpos.massimo() == 1000);
esito &= (negEzero.massimo() == 0);
System.out.println(negEzero.massimo() == 0);
esito &= (inPrimaPos.massimo() == 1000);
System.out.println(inPrimaPos.massimo() == 1000);
esito &= (inUltimaPos.massimo() == 1000);
System.out.println(inUltimaPos.massimo() == 1000);

System.out.println(esito);
}
```

# Soluzione Artigianale

- La soluzione presentata, benché chiaramente *artigianale*, è completamente automatica
  - dopo ogni modifica al metodo sotto test possiamo velocemente far rigirare il programma di test e verificare se ci sono cambiamenti (regressioni) nei risultati
  - in presenza di fallimenti la ricerca degli errori risulta fortemente semplificata dalle informazioni desumibili già dal test fallito stesso

# Automazione del Testing

- I test devono essere:
  - *automatici* (per mantenere rapido il ciclo di feedback)
    - devono essere eseguiti molte volte al giorno
  - *efficienti*
    - devono essere convenienti rispetto alle ispezioni manuali
  - *isolati* e che garantiscano la *località* degli errori
    - dal fallimento di un test alla rimozione del bug deve trascorrere poco tempo grazie alle caratteristiche di forte località del test per gli errori che rilevano
  - ed inoltre:
    - separati dal codice applicativo
    - eseguibili e verificabili separatamente

# Automazione del Testing: JUnit

- Esistono vari strumenti per assistere il programmatore nel testing, ed in particolare nello unit-testing
- JUnit: <http://www.junit.org>
- Il più noto ed utilizzato framework (insieme di classi e convenzioni d'uso) per la scrittura di test unitari
- Fortemente integrato con gli ambienti di sviluppo più diffusi come Eclipse>>

# JUnit: Test del Metodo massimo()

```
import static org.junit.jupiter.api.Assertions.*;  
import org.junit.jupiter.api.Test;  
public class SequenzaTest {
```

*import di classi ed  
annotazioni Junit 5*

*nome*

**@Test**

*Annotazione di metodo come test-case*

```
public void testMassimoPositivi() {  
    Sequenza seq = new Sequenza(5);  
    seq.setElemento(0,1);  
    seq.setElemento(1,5);  
    seq.setElemento(2,8);  
    seq.setElemento(3,3);  
    seq.setElemento(4,4);  
    assertEquals(8, seq.massimo());  
}
```

*Asserzione*

*test-case*

**@Test**

```
public void testMassimoNegativi() {  
    ...  
    ...  
}...}
```

*test-case*

# JUnit: Struttura Classi di Test (1)

- Tutte le classi di test che scriveremo avranno questa struttura
- Ovviamente le classi di test vanno progettate sulla base delle peculiarità della classe testata
- Collochiamo la classe di test nello stesso package della classe che si sta testando
- Convenzione sui nomi basata sul suffisso:

Classe

**Sequenza → SequenzaTest**

Classe di Test

# JUnit: Struttura Classi di Test (2)

- `import static org.junit.jupiter.api.Assertions.*;`  
Serve per importare vari metodi statici del framework che permettono di fare *asserzioni*>>
- `import org.junit.jupiter.api.Test;`  
Serve per importare l'*annotazione* del framework **@Test** utile a marcare i metodi i test-case
- Non è (più) necessario ma è (tuttora) buona norma usare '**test**' come prefisso del nome dei test-case

**@Test**

```
public void testCostruzioneComandiInvalidi() {  
    ...  
}
```



# JUnit: Asserzioni

- Asserzione:  
*affermazione che può essere vera o falsa*
- I risultati attesi sono documentati con delle *asserzioni* esplicite, non mediante stampe
  - richiederebbero dispendiose ispezioni visuali
- Se l'asserzione è
  - *vera* : il test ha avuto successo, 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

# JUnit: Metodi `assertXYZ()`

- Una asserzione non vera fa fallire il test-case
  - **`assertEquals(Object expected, Object actual)`**: afferma l'«uguaglianza» degli argomenti (>>)
  - **`assertNull(Object object)`**: afferma che il suo argomento è nullo (fallisce se non lo è)
  - molte altre varianti
    - **`assertNotNull()`**
    - **`assertTrue()`**
    - **`assertFalse()`**
    - **`assertSame()...`**

tutte sovraccariche ... e talvolta facilmente intercambiabili...
- Usare sempre la versione più pertinente!
  - meglio **`assertFalse(b)`** di **`assertTrue(b==false)`**,
  - meglio **`assertNotNull(o)`** di **`assertTrue(o!=null)`**

# JUnit: assertEquals()

- **assertEquals(Object expected, Object actual)**  
«expected» è il valore atteso, che ci si aspetta normalmente  
«actual» è il valore effettivo, reale, quello ottenuto
  - afferma che il suo secondo argomento è uguale al primo argomento
  - va a buon fine se e solo se **expected.equals(actual)** restituisce **true**
- Una variante, spesso preferibile

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

- un messaggio diagnostico da stampare solo in caso di fallimento
- se ben ideato, dovrebbe permettere di comprendere il motivo del fallimento senza nemmeno aprire il debugger

# Le “Tre Fasi” in Pratica

@Test

```
public void testMassimoPositivi() {  
    Sequenza seq = new Sequenza(5);  
    seq.setElemento(0,1);  
    seq.setElemento(1,5);  
    seq.setElemento(2,8);  
    seq.setElemento(3,3);  
    seq.setElemento(4,4);  
    assertEquals(8, seq.massimo());  
}
```

1) mettere un “frammento” del sistema in un stato *iniziale* noto

- il frammento comprende un *solo* oggetto **Sequenza** opportunamente popolato di valori (nell'es. si tratta di interi tutti positivi)

2) inviare una serie di sollecitazioni (`seq.massimo()`)

3) controllare tramite asserzioni che si raggiunga lo stato *finale* atteso (`assertEquals(...)`)

# JUnit: Compilare i Test

Per compilare ed eseguire i test:

- Nel classpath devono essere presenti le librerie di JUnit
- ✓ Eclipse snellisce molti di questi passaggi sino a renderli quasi trasparenti
  - ed arriva a suggerire di aggiungere la libreria nel build path del vostro progetto non appena ne nota l'utilizzo

# JUnit ed Eclipse

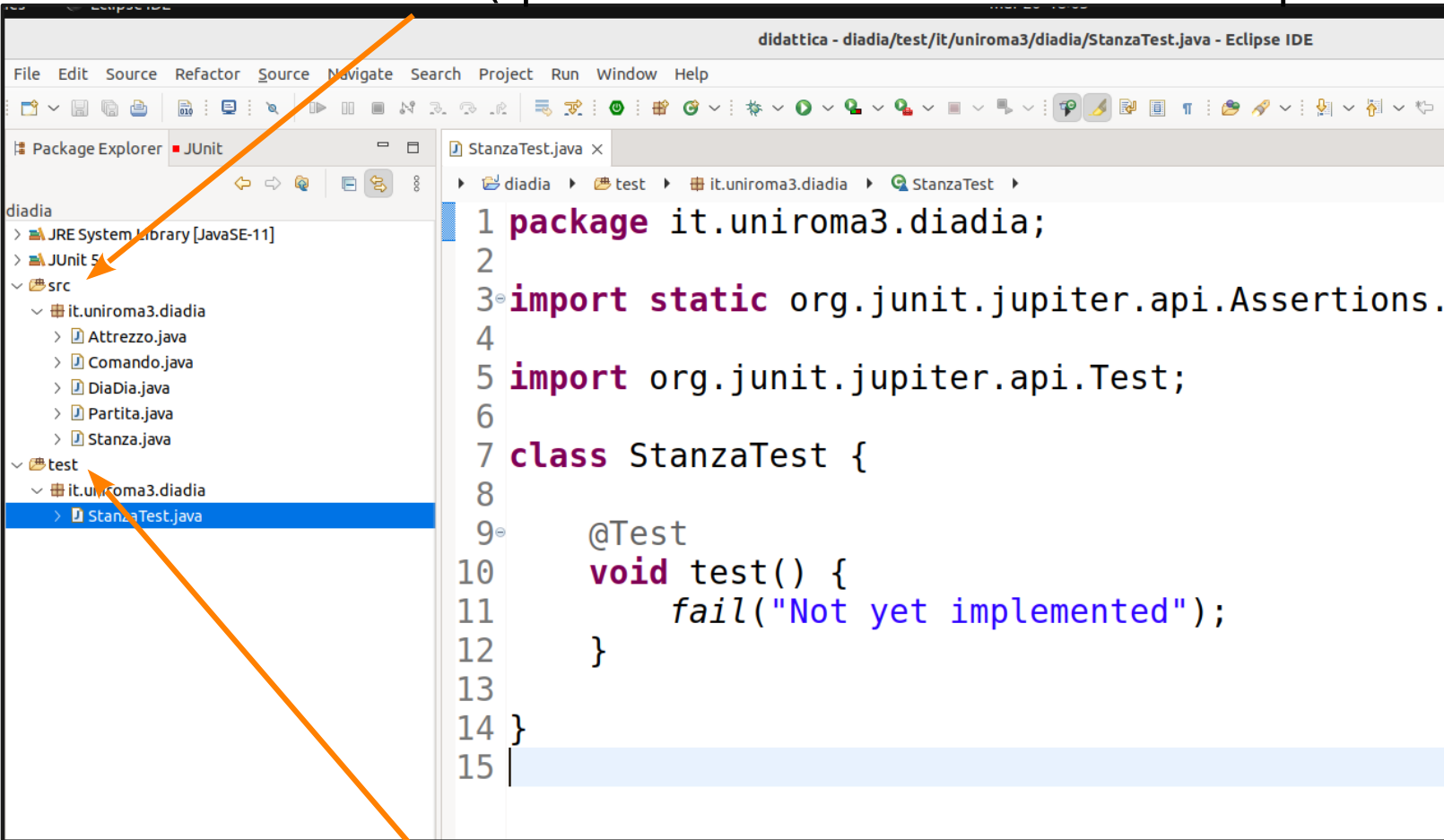
- JUnit è talmente popolare da venire fornito già integrato e fortemente supportato negli IDE
- Nel caso di Eclipse:
  - per creare una classe di test  
click con tasto destro del mouse sulla classe  
new-> JUnit Test Case (spuntare new JUnit 5)
  - per eseguire una classe di test  
click con tasto destro del mouse sulla classe di test  
run as -> JUnit test
    - barra verde: il test è andato a buon fine
    - barra blue: il test è fallito violando una asserzione
    - barra rossa: il test è fallito causando un errore

# JUnit ed Eclipse

- Le classi di test devono essere nello stesso package delle classi da testare. Scomodo!
  - Si pensi ad una consegna del solo codice di produzione
- Ma in Eclipse possiamo collocare uno stesso package anche in directory (*source folder*) diverse
  - In ogni progetto,
    - nella source folder **src** mettiamo il codice di produzione
    - nella source folder **test** mettiamo le classi di test (organizzate con gli stessi package delle classi di produzione)
  - per creare una source folder: tasto destro del mouse sul progetto, quindi new->Source Folder
    - dentro la source folder **test** creiamo una copia “parallela” dei package del codice di produzione (new->package)

# src vs test Cartella/Folder

Source folder **src** (qui vanno le classi del codice di produzione)



Source folder **test** (qui vanno le classi del codice di test)



# JUnit: Fixture

- Per facilitare la scrittura dei test-case, è spesso comodo creare degli oggetti in uno stato iniziale noto e «pronto» per l'utilizzo da parte di tutti i test-case
- Può convenire fattorizzare il codice di creazione di questi oggetti. Ad es.
  - utilizzando metodi **setUp()**
  - mediante i cosiddetti *factory methods*
- Le *fixture* sono oggetti in uno stato iniziale noto ed ospitati in variabili d'istanza che le classi di test predispongono allo scopo

# Fixture e JUnit

- Attraverso l'annotazione **@BeforeEach** è possibile indicare quali metodi eseguire *prima di ciascuna* invocazione di un test-case
- Tipicamente questi metodi inizializzano le fixture
- Spesso, ma non più obbligatoriamente, questi metodi vengono tuttora denominati **setUp()**
  - perché con le versioni di JUnit pre-annotazioni Java (JUnit 3.x) era un nome imposto per convenzione

# Fixture e Metodo setUp() (1)

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

**Fixture**



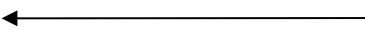
```
public class SequenzaTest {
    private Sequenza positivi;
    private Sequenza negativi;
```

**@BeforeEach**

```
public void setUp() {
    this.positivi = new Sequenza(5);
    this.positivi.setElemento(0,1);
    this.positivi.setElemento(1,5);
    this.positivi.setElemento(2,8);
    this.positivi.setElemento(3,3);
    this.positivi.setElemento(4,4);

    this.negativi = new Sequenza(5);
    this.negativi.setElemento(0, -6);
    ...
}
```

*Metodo eseguito prima di ogni  
invocazione di test-case*



```
@Test
public void testMassimoPositivi() {...}
```

```
@Test
public void testMassimoNegativi() {...}
```

```
}
```

# Fixture e Metodo setUp() (2)

```
...
public class SequenzaTest {
    ...
    @Test
    public void testMassimoPositivi() {
        assertEquals(8, this.positivi.massimo());
    }

    @Test
    public void testMassimoNegativi() {
        assertEquals(-1, this.negativi.massimo());
    }

    ...
}
```

# setUp() e l'Importanza dei Nomi

- *Dopo diverso tempo dalla prima scrittura, a meno che il nome della fixture `this.positivi` sia perfettamente indicativo di “sequenza non vuota di interi tutti positivi” si finirà per dover leggere il corpo del metodo `setUp()` per poterne comprendere appieno il significato*
- Una situazione migliore:

```
...
public class SequenzaTest {
    ...
    @Test
    public void
    testMassimoDiSequenzaNonVuotaDiInteriTuttiPositivi() {
        assertEquals(8, this.positivi.massimo());
    }
    ...
}
```

# setUp(): Controindicazioni

- Se nel **setUp()** si accumulano le fixture di diversi test-case (seq. positive, negative ecc. ecc. ...) si creano tanti oggetti che non hanno nulla a che vedere con il singolo test appena fallito, ad es. **testMassimoPositivi()**
- Si è costretti a leggere un lungo **setUp()** solo per comprendere un breve test-case
- Mettendo a fattor comune tutte le fixture utilizzate una sola volta si sta ledendo la leggibilità dei test-case rendendoli meno autocontenuti
- Tramite il **setUp()** si finisce per creare impliciti ma sottili accoppiamenti tra test-case

# setUp(): Indicazioni

- In genere, nel `setUp()` ha senso fattorizzare solo la creazione di oggetti che vengano utilizzati da almeno **due** test-case distinti
- In tutti gli altri casi meglio non distribuire il codice di uno scenario di test in due distinti metodi `test()` + `setUp()`. Altrimenti:
  - Il test non è *isolato*: per comprenderne uno si finisce per dover capire (almeno una parte) di tutti
  - Il test non è *autocontenuto*: per ricostruire lo scenario di test bisogna cercare ben oltre il corpo del test-case stesso
- In definitiva, la *località* del test potrebbe risentirne

# Qualità dei Test-Case (1)

- ATTENZIONE: la qualità di un test si avverte in particolare quando il test smette di funzionare e bisogna trovare l'errore all'origine del fallimento
  - *può capitare anche dopo molto tempo* dalla scrittura iniziale del codice
  - quando oramai lo stesso non risulta affatto “familiare”
  - magari subito dopo avere effettuato un refactoring...
- Lo sforzo necessario per rimuovere un errore appena introdotto è uno dei più importanti indicatori della qualità dei test che smettono di funzionare
- ✓ Questo sforzo, abbiamo già visto, dipende largamente dalla *località* di un test



# Qualità dei Test-Case (2)

- Qual'è la lunghezza ottimale di un test-case?
  - 1 (dicesi **UNA**) – linea di codice totale!
- E' possibile perseguire questo obiettivo utilizzando alcuni accorgimenti
  - fixtures
  - factory methods
  - *minimalità*>>
- ✓ N.B. per la località dei test non è solo utile conseguire il risultato di aver test monolinea, ma è forse ancora più importante ricordarsi di perseguirlo
  - meno linee possibili per test-case
  - *molto* meglio 10 test-case con 1 asserzione ciascuno che 1 test-case con 10 asserzioni!

# Factory Methods

- Per favorire la semplicità dei test si può pensare di fattorizzare il codice di creazione della fixture

```
public class SequenzaTest {
```

```
...  
    private Sequenza sequenza(int... array) {  
        Sequenza risultato = new Sequenza(array.length);  
        for(int i=0; i<array.length; i++) {  
            risultato.setElemento(i,array[i]);  
        }  
        return risultato;  
    }  
}
```

**Equivale a**  
`private Sequenza sequenza(int[] array)`

```
@Test
```

```
public void testMassimoDiSequenzaNonVuotaDiInteriTuttiPositivi() {  
    assertEquals(8, sequenza(1,5,8,3,4).massimo());  
}
```

```
...
```

```
}
```

# Testing - Punto di Vista di un Programmatore-Utilizzatore (1)

- Il factory method **sequenza()** ha reso evidente quanto sia “faticoso” creare una sequenza per gli utilizzatori della classe
  - la classe di test è solo una delle possibili classi *utilizzatrici/clienti*
- A ben vedere anche altri utilizzatori della classe possono condividere la stessa esigenza
- Se cambiamo **Sequenza** per facilitare il testing, rendiamo più semplice l’uso della classe da parte anche di tutti gli altri utilizzatori
  - Basta aggiungere il costruttore **Sequenza(int[] e)?**

# Autocontenimento dei Test

```
...
public class SequenzaTest {
    ...
    @Test
    public void testMassimoDiSequenzaNonVuotaDiInteriTuttiPositivi() {
        assertEquals(8, new Sequenza(1,5,8,3,4).massimo());
    }
    ...
    @Test
    public void testMassimoDiSequenzaNonVuotaDiInteriTuttiNegativi() {
        assertEquals(-1, new Sequenza(-6,-1,-8,-13,-10).massimo());
    }
    ...
    @Test
    public void testMassimoInPrimaPosizione() {
        assertEquals(1000, new Sequenza(1000,0,80,-10,-10).massimo());
    }
    ...
}
```

# Minimalità (1)

- I test visti sinora non sono *minimali*
- E' possibile esprimere lo stesso scenario di testing con test-case più brevi, e che fanno uso di oggetti di stato meno complesso
- Quale di questi test-case preferire? perché?

@Test

```
public void testMassimoDiSequenzaNonVuotaDiInteriTuttiPositivi() {  
    assertEquals(8, new Sequenza(1, 5, 8, 3, 4).massimo());  
}
```

***...oppure...***

@Test

```
public void testMassimoDiSequenzaNonVuotaDiInteriTuttiPositivi() {  
    assertEquals(2, new Sequenza(1, 2).massimo());  
}
```

**???**

# Minimalità (2)

- L'uso di test minimali rende molto più semplice la ricerca degli errori
- Nulla è più minimale di una sequenza vuota!

```
@Test  
public void testMassimoDiSequenzaVuota() {  
    assertEquals(???, new Sequenza().massimo());  
}
```

- Qual'è il massimo di una sequenza vuota?

# Testing - Punto di Vista di un Programmatore-Utilizzatore (2)

- Di nuovo il testing ha evidenziato un problema nel contratto di utilizzo che il metodo `massimo()` espone agli utilizzatori

JUnit 4

```
public class Sequenza {  
    ...  
    public int massimo() {  
        if (this.sequenza.length==0)  
            throw new java.util.NoSuchElementException();  
        // ...  
    }  
}
```

```
@Test(expected = java.util.NoSuchElementException.class)  
public void testMassimoDiSequenzaVuota() {  
    new Sequenza().massimo();  
}
```

# ***“Si nasce minimali, non ci si diventa”***

- Conviene sempre partire dai test più semplici, perché sono quelli che permetteranno di rimuovere la *maggior* parte dei bug il *minor* sforzo possibile
- E' poi naturale aumentare, via via, la complessità degli scenari di testing
  - ✓ per aumentare la propria confidenza sulla correttezza del proprio codice
- Ad esempio, se ipotizziamo di ordinare tutti i test-case secondo la complessità dello scenario di testing trattato...



# Scenari di Testing di Complessità Crescente

```
...
public class SequenzaTest {
    @Test(expected = java.util.NoSuchElementException.class)
    public void testMassimoDiSequenzaVuota() {
        new Sequenza().massimo();
    }
    @Test
    public void testMassimoDiSequenzaSingleton() {
        assertEquals(1, new Sequenza(1).massimo());
    }
    @Test
    public void testMassimoInPrimaPosizione() {
        assertEquals(2, new Sequenza(2,1).massimo());
    }
    @Test
    public void testMassimoInSecondaPosizione() {
        assertEquals(2, new Sequenza(1,2).massimo());
    }
    @Test
    public void testMassimoDiSequenzaNonVuotaDiInteriTuttiNegativi() {
        assertEquals(-1, new Sequenza(-2,-1).massimo());
    }
    ...
}
```

*Fixture di complessità  
crescente*



# Quando Scrivere i Test?

- Uno dei più gravi e purtroppo frequenti errori di chi viene introdotto allo unit-testing è aspettare la fine della scrittura di tutto il codice principale per cominciare a scrivere il codice di test
- E' la scelta peggiore! si massimizzano i costi di scrittura dei test e si minimizzano i benefici
  - ✓ potranno esistere contemporaneamente molteplici errori, anche correlati, e se tanti test falliscono, non è più chiaro da dove cercarli... in due parole: *scarsa località*
- ✓ Risulta più conveniente scriverli continuamente, *durante* o addirittura *prima* della scrittura del codice principale (>>)

# Testing Continuo

- Il testing deve essere una attività associata ed affiancata all'ordinario sviluppo del codice principale: avviene progressivamente e continuativamente
- Principali motivazioni legate ai costi
  - la rimozione precoce degli errori riduce i costi di sviluppo
  - si costruisce contestualmente al codice principale un ambiente di test
  - si accumulano *batterie di test* molto utili per lo sviluppo e la manutenzione efficace del codice principale
  - i test possono essere riutilizzati durante la manutenzione del software ad esempio per evitare regressioni

# Testing - Punto di Vista di un Programmatore-Utilizzatore (3)

- Chi scrive test si costringe nel ruolo del *Programmatore-Utilizzatore* e si focalizza sulla semplicità di utilizzo del proprio codice
- Per questo motivo il testing aiuta a cambiare la prospettiva di visione sul proprio codice, a concentrarsi sulle interfacce delle proprie classi e sulla distribuzione delle responsabilità
- Tipicamente il codice di qualità è più *testabile* e viceversa
- Esistono metodologie di sviluppo che portano all'estremo questa attitudine: *T.D.D.*

# Test Driven Development

- Promuove l'uso dei test anche come strumento di progettazione
  - i test guidano lo sviluppo verso codice che sia semplice, facilmente testabile e di qualità
- Predica la scrittura dei test-case **prima** della scrittura del codice testato
  - anticipa nel tempo ed evidenzia il punto di vista del Programmatore-Utilizzatore
  - predilige micro-iterazioni
    - testing-coding-testing-coding...*
    - che incentivano la minimalità dei test

# Sviluppo Guidato dai Test

- Se scriviamo il codice di test prima del codice stesso siamo incentivati a:
  - precisare i metodi visibili all'esterno in quanto il codice di test è codice *cliente*
    - esterno alla classe alla stregua di tutte le altre classi clienti del codice testato
  - chiarire la semantica dei metodi
  - cercare di semplificare al massimo l'utilizzo del codice
  - Individuare i casi limite e chiarire la gestione delle situazioni anomale

# Motivazioni del Testing: Conclusioni

- La rimozione precoce degli errori riduce i costi di sviluppo e migliora la qualità del codice
- I test inducono ad assumere anticipatamente il punto di vista del Programmatore-Utilizzatore e spingono gli sviluppatori verso soluzioni più semplici per gli utilizzatori
- Si documenta in maniera formale e precisa il funzionamento del codice

# Jupiter - JUnit 5 vs JUnit 4

| JUnit 4  | JUnit 5  |
|--|--|
| <code>@BeforeClass, @AfterClass</code>   | <code>@BeforeAll, @AfterAll</code>   |
| <code>@Before, @After</code>   | <code>@BeforeEach, @AfterEach</code>   |
| <code>@Ignore</code>   | <code>@Disabled</code>   |
| <code>@Category</code>   | <code>@Tag</code>  |
| Assert class.<br><br>Optional assertion message is the first parameter.              | Assertions class.<br><br>Optional assertion message is the last parameter.                                 |
| Assume class.<br><br><code>assumeNotNull</code> and <code>assumeNoException</code> . | Assumptions class.<br><br><code>assumeNotNull</code> and <code>assumeNoException</code> have been removed. |

- Molti miglioramenti, nessuna rivoluzione, perché già JUnit 4 funzionava benissimo
- Alcune rendono il testing più facile in particolari scenari
  - test parametrici (>>)



# JUnit 5 vs JUnit 4: In Pratica

- Nella pratica serve conoscere entrambe le versioni
  - “Greenfield” project: usare Jupiter – JUnit 5
  - “Brownfield” project: continuare ad usare JUnit 4
- Conviene cambiare?
  - Soprattutto per un uso basico, JUnit 4 già funzionava benissimo
  - Per utilizzi più avanzati, le soluzioni JUnit 4 risultano, in genere, sensibilmente più “macchinose” delle equivalenti in JUnit 5
- Un project “brownfield” su tutti: il SISTEMA QUIZ
  - Si basa su JUnit 4 e non vale la pena di aggiornarlo, al momento...
- Se si decide di aggiornare un progetto, come prima cosa ricordarsi di cambiare gli import...
  - Da JUnit 4:
    - `import static org.junit.Assert.*; // Asserzioni assertXYZ...`
    - `import org.junit.*; // Annotazioni @...`
  - A JUnit 5:
    - `import static org.junit.jupiter.api.Assertions.*; // assertXYZ...`
    - `import org.junit.jupiter.api.*; // Annotazioni @...`

# Esercizi

- Scrivere (con Eclipse) una classe di test JUnit per la classe **Persone** (dal Quiz di preparazione alla prima verifica)
- In particolare testare il metodo `int conta0monimiDi(String nome)`
- Scrivere il codice del metodo `int conta0monimiDi(String nome)`
- Eseguire la classe di test JUnit (se il test fallisce, correggere il metodo sotto test e far girare nuovamente la classe di test)

```
public class Persone {  
    private String[] nomi;  
  
    public Persone(int n) {  
        this.nomi = new String[n];  
    }  
  
    public int conta0monimiDi(String nome) {  
        // metodo da scrivere  
    }  
  
    public void aggiungiNome(int indice, String nome){  
        this.nomi[indice] = nome;  
    }  
}
```