14
Reflection
e informazioni run-time sui tipi

Mirko Viroli
mirko.viroli@unibo.it

C.D.L. Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Cesena

a.a. 2014/2015

Outline

Goal della lezione

- Illustrare il concetto di run-time type information
- Mostrare le principali funzionalità della Reflection
- Descrivere il meccanismo delle annotazioni

Argomenti

- Oggetti `Class` e loro uso
- Reflection API
- Annotazioni di Java
- Testing con `JUnit`

Outline

Il classfile

Classi e JVM

- Ogni classe Java (interfaccia, enumerazione, inner o outer) produce un file `.class` ad opera del compilatore
- Nel caso di classi "inner" il nome di tale `.class` è `<outer>$<inner>`, nel caso di anonima è `<outer>$<numero>`
- Tale `.class` è disponibile nel folder di uscita e innestato a seconda del suo package
- Il contenuto informativo del `.class` è quasi identico a quello del `.java`, solo espresso in un linguaggio diverso che garantisce la correttezza del contenuto e le performance di chi deve leggerlo

È possibile ispezionare il contenuto di un `.class`

- Esempio di comando: `javap -v Counter.class`
- Comando in modalità verbose

Classe Counter

```java
class Counter{

    /* Il campo è reso inaccessibile direttamente */
    private int countValue;

    /* E' il costruttore che inizializza i campi */
    public Counter(){
      this.countValue=0;
    }

    /* Unico modo per osservare lo stato */
    public int getValue(){
      return this.countValue;
    }

    /* Unico modo per modificare lo stato */
    public void increment(){
      this.countValue++;
    }
}
```

Contenuto del classfile (1/2)

```
Classfile /media/dati/work/Lessons/2013-2014/oop/lex/slides/13/13/loading/Counter.class
  Last modified Sep 20, 2013; size 361 bytes
  MD5 checksum 2c941be1d3cfd60de02143767ce146cc
  Compiled from "Counter.java"
class Counter
  SourceFile: "Counter.java"
  minor version: 0
  major version: 51
  flags: ACC_SUPER
Constant pool:
   #1 = Methodref          #4.#16         //  java/lang/Object."<init>":()V
   #2 = Fieldref           #3.#17         //  Counter.countValue:I
   #3 = Class              #18            //  Counter
   #4 = Class              #19            //  java/lang/Object
   #5 = Utf8               countValue
   #6 = Utf8               I
   #7 = Utf8               <init>
   #8 = Utf8               ()V
   #9 = Utf8               Code
  #10 = Utf8               LineNumberTable
  #11 = Utf8               getValue
  #12 = Utf8               ()I
  #13 = Utf8               increment
  #14 = Utf8               SourceFile
  #15 = Utf8               Counter.java
  #16 = NameAndType        #7:#8          //  "<init>":()V
  #17 = NameAndType        #5:#6          //  countValue:I
  #18 = Utf8               Counter
  #19 = Utf8               java/lang/Object
```

Contenuto del classfile (2/2)

```
{  public Counter();
    flags: ACC_PUBLIC   Code:
      stack=2, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: aload_0
         5: iconst_0
         6: putfield      #2                  // Field countValue:I
         9: return
      LineNumberTable:        line 7: 0 line 8: 4        line 9: 9

  public int getValue();
    flags: ACC_PUBLIC Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: getfield      #2                  // Field countValue:I
         4: ireturn
      LineNumberTable:    line 13: 0
  public void increment();
    flags: ACC_PUBLIC Code:
      stack=3, locals=1, args_size=1
         0: aload_0
         1: dup
         2: getfield      #2                  // Field countValue:I
         5: iconst_1
         6: iadd
         7: putfield      #2                  // Field countValue:I
        10: return
      LineNumberTable:          line 18: 0        line 19: 10   }
```

Caricamento delle classi e la JVM (1/2)

Caricamento: chi?

- La JVM è un programma solitamente scritto in C/C++
- Dispone di uno o più class-loader (realizzabili anche in Java dall'utente, estendendo `java.lang.ClassLoader`)
- Hanno il compito di cercare i classfile che servono, e di "caricarli" nella JVM

Caricamento: quando?

- Tale caricamento **non** avviene necessariamente all'avvio: ogni classe viene caricata al momento del suo primo utilizzo!! (Schema by-need)
- Alla prima `new`, o chiamata statica, o se serve una sottoclasse!
- (o con una richiesta esplicita come vedremo)

Caricamento delle classi e la JVM (2/2)

Caricamento: da dove?

- Dal file system, attraverso il classpath e navigando i package
- Eventualmente dentro ai file JAR
- In alcune modalità, può caricare le classi anche via rete!

Caricamento: cosa succede?

- La JVM prepara una opportuna struttura dati in memoria
- Inizializza i campi statici (chiamando l'inizializzatore statico)

Ispezioniamo la dinamica di caricamento

```
1  class A{
2    static { System.out.println("A.class caricato");}
3  }
4
5  class B extends A{
6    static { System.out.println("B.class caricato");}
7  }
8
9  public class Loading {
10   static { System.out.println("Loading.class caricato");}
11
12   public static void main(String[] args) {
13     System.out.println("main partito");
14     new B();
15     System.out.println("creato un oggetto di B");
16   }
17   /*  Loading.class caricato
18     main partito
19     A.class caricato
20     B.class caricato
21     creato un oggetto di B */
22 }
```

Perché questa gestione?

Alcune motivazioni

- Permette alle applicazioni di "partire" più velocemente, in quanto non si carica tutto, solo quello che serve mano a mano
- In scenari di rete, consente di dover caricare da remoto solo sottoparti di applicazioni
- Class-loader avanzati potrebbero anche "scaricare" (togliere dalla JVM) le classi che sembrano non servire più
- È possibile ispezionare e usare il contenuto delle classi via reflection

Outline

Reflection

Packages `java.lang` e `java.lang.reflect`

Forniscono una libreria che interagisce con la JVM per..

- dare una rappresentazione "ad oggetti" del contenuto di una classe
- direttamente istanziare un oggetto, invocare metodi, accedere a campi
- forzare il caricamento di una classe

Sulla modalità "via reflection"

È più flessibile ma..

- è più lenta, anche di 1-2 ordini di grandezza
- non interagisce con i controlli del compilatore (genera eccezioni..)
- pone problemi di security

$\Rightarrow$ va usata di conseguenza.. quindi non abusarne

Motivazioni

Tecniche messe a disposizione

- Unire classi non note ad una applicazione durante il suo funzionamento
- Trattare una stringa come identificatore del linguaggio

Applicazioni della reflection

- Estendibilità: Si può fare uso di classi esterne create/caricate "al volo", per modificare dinamicamente il comportamento di una applicazione
- Ambienti di sviluppo: Poter ispezionare la struttura di una classe o libreria
- Framework di Java: annotazioni, serializzazione, dynamic proxies,...

La classe `java.lang.Class`

Ogni suo oggetto rappresenta un tipo disponibile nella JVM

- una classe (outer o inner, astratta o non), una interfaccia, una enumerazione, un array, i tipi primitivi e anche void
- non i tipi generici (`ArrayList<String>`) ma le classi generiche si

Come si ottiene un oggetto di `Class`? In vari modi:

- `String.class`
- `new String("this is a string").getClass()`
- `Class.forName("java.lang.String")`

Genericità di `Class`

- Solo via `String.class` si può ottenere un oggetto di tipo `Class<String>`
- Altrimeni si può recuperarlo con un cast "unchecked"

Esempi

```
1  public class TryClass {
2
3    public static void main(String[] args) throws ClassNotFoundException  {
4
5      // Unico caso di recupero del corretto tipo generico
6      Class<Integer> c = Integer.class;
7      System.out.println(c);
8
9      // Si può ottenere una class da una stringa calcolata
10     Class<?> c2 = Class.forName("java.lang" + ".Integer");
11     System.out.println(c2);
12
13     // Accesso alla classe di un oggetto
14     Object o = 3;
15     Class<?> c3 = o.getClass();
16     System.out.println(c3);
17
18     // Cast "unchecked" per recuperare il generico
19     Class<Integer> c4 = (Class<Integer>)o.getClass();
20     System.out.println(c3);
21   }
22 }
```

Esempi

```
public class UseClass {
  public static void main(String[] args) throws ClassNotFoundException {
    Class<String> c = String.class;
    System.out.println(c.getName()+" "+c.getCanonicalName());
    //java.lang.String java.lang.String
    Class<Integer> ci = (Class<Integer>)new Integer(5).getClass();
    System.out.println(ci.getName()+" "+ci.getCanonicalName());
    //java.lang.Integer java.lang.Integer
    Class<?> ca = new int[20].getClass();
    System.out.println(ca.getName()+" "+ca.getCanonicalName());
    //[I int[]
    Class<?> cint = ca.getComponentType();
    System.out.println(cint.getName()+" "+cint.getCanonicalName());
    //int int
    Class<?> cl = Class.forName("java.util.List");
    System.out.println(cl.getName()+" "+cl.getCanonicalName());
    //java.util.List java.util.List
    Class<?> can = new Object(){
      public String toString(){ return "none";}
    }.getClass();
    System.out.println(can.getName()+" "+can.getCanonicalName());
    //it.unibo.apice.oop.p14reflection.UseClass$1 null
  }
}
```

Alcuni metodi di `java.lang.Class`

```
public final class Class<T> implements ...  {

    public static Class<?> forName(String className) throws ClassNotFoundException {...}

    public boolean isInterface();
    public boolean isArray();
    public boolean isPrimitive();
    public boolean isAnonymousClass() {...}
    public boolean isLocalClass() {...}
    public boolean isMemberClass() {...}
    public boolean isEnum() {...}

    public T[] getEnumConstants() {...}
    public Class<?> getComponentType();
    public T cast(Object obj) {...}

    public T newInstance() throws ...{...}

    // Accessing the structure.. can throw SecurityException
    public Field[] getFields() throws ... {...}
    public Method[] getMethods() throws ... {...}
    public Constructor<?>[] getConstructors() throws ... {...}

    public Field getField(String name) throws NoSuchFieldException, .. {...}
    public Method getMethod(String name, Class<?>... parameterTypes)
        throws NoSuchMethodException, .. {...}
    public Constructor<T> getConstructor(Class<?>... parameterTypes)
        throws NoSuchMethodException, .. {...}

    public Field[] getDeclaredFields() throws SecurityException {...}
    ... // All versions with 'Declared'
}
```

Alcuni metodi di java.lang.reflect.Field

```
public final class Field extends ... {

    public Class<?> getDeclaringClass() {...}
    public String getName() {...}
    public int getModifiers() {...}
    public boolean isEnumConstant() {...}
    public Class<?> getType() {...}

    public Object get(Object obj)
        throws IllegalArgumentException, IllegalAccessException{...}
    public boolean getBoolean(Object obj)
        throws IllegalArgumentException, IllegalAccessException{...}
    public byte getByte(Object obj)
        throws IllegalArgumentException, IllegalAccessException{...}
    ...

    public void set(Object obj, Object value)
        throws IllegalArgumentException, IllegalAccessException{...}
    ...
}
```

Alcuni metodi di java.lang.reflect.Constructor

```
public final class Constructor<T> extends ... {

    public Class<T> getDeclaringClass() {...}
    public String getName() {...}
    public int getModifiers() {...}
    public Class<?>[] getParameterTypes() {...}
    public Class<?>[] getExceptionTypes() {...}

    public T newInstance(Object ... initargs)
        throws InstantiationException, IllegalAccessException,
               IllegalArgumentException, InvocationTargetException {...}

    public boolean isVarArgs() {...}
    ...
}
```

Alcuni metodi di `java.lang.reflect.Method`

```
public final class Method extends ... {

    public Class<?> getDeclaringClass() {...}
    public String getName() {...}
    public int getModifiers() {...}
    public Class<?> getReturnType() {...}
    public Class<?>[] getParameterTypes() {...}
    public Class<?>[] getExceptionTypes() {...}
    public boolean isVarArgs() {...}

    public Object invoke(Object obj, Object... args)
        throws IllegalAccessException, IllegalArgumentException,
           InvocationTargetException {...}
}
```

Caricamento ed esecuzione automatica

```
public class DynamicExecution {

  public final static String Q_CLASS = "Insert fully-qualified class name: ";
  public final static String Q_METH = "Insert name of method to call: ";
  public final static String L_OK = "Everything was ok! The result is..";
  public final static String E_RET = "Wrong return type";

  public static void main(String[] s) {
    BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
    while (true) {
      try {
        System.out.println(Q_CLASS);
        String className = reader.readLine();
        System.out.println(Q_METH);
        String methName = reader.readLine();

        Class<?> cl = Class.forName(className);
        Constructor<?> cns = cl.getConstructor();
        Method met = cl.getDeclaredMethod(methName);
        if (!met.getReturnType().isAssignableFrom(String.class)) {
          throw new NoSuchMethodException(E_RET);
        }
        Object o = cns.newInstance();
        String result = (String) met.invoke(o);
        System.out.println(L_OK);
        System.out.println(result);
        System.out.println();
      } catch (Exception e) {
        System.out.println(e);
      }
    }
```

Outline

Sviluppiamo un semplice esempio

Realizzazione automatizzata del `toString` – un mero esempio

- Scrivere i `toString` è piuttosto noioso e ripetitivo
- Alcuni IDE (come Eclipse) lo generano automaticamente
- Come potremmo fornire un supporto programmato?
- Il principale problema è indicare dinamicamente, di volta in volta, come produrre la stringa sulla base delle proprietà di interesse
- Forniamo una soluazione base, facilmente estendibile dallo studente

Idea

- Fornisco un metodo statico in una classe di funzionalità varie
- Accetta l'oggetto da stampare e una descrizione di cosa stampare
- Esempio: il nome dei getter da chiamare
- Ritorna la stringa creata

objectToString

```
1  public class PrintObjectsUtilities {
2
3    public static String objectToString(Object o, String... getters) {
4      try {
5        String out = o.getClass().getSimpleName()+": ";
6        for (String getter : getters) {
7          // Sistemo la maiuscola iniziale
8          getter = getter.substring(0,1).toUpperCase() + getter.substring(1);
9          // Trovo il getter e lo invoco
10         Method m = o.getClass().getMethod("get" + getter);
11         Object res = m.invoke(o);
12         // Aggiungo la stringa
13         out += " " + getter + " -> ";
14         out += res.getClass().isArray()
15               ? Arrays.deepToString((Object[])res)
16               : res.toString();
17         out += " |";
18       }
19       return out.substring(0,out.length()-2);
20     } catch (Exception e) {
21       return null;
22     }
23   }
24 }
```

Classi di prova

```
1  public class Person {
2      ...
3
4    public Person(String name, int id) {..}
5
6    public String getName() {..}
7
8    public int getId() {..}
9
10 }
```

```
1  public class Teacher extends Person {
2      ...
3
4    public Teacher(String name, int id, String... courses) {..}
5
6    public String[] getCourses() {..}
7  }
```

Uso di objectToString

```
1  import static it.unibo.apice.oop.p14reflection.classes.PrintObjectsUtilities
       .*;
2
3  public class UsePrintObjectUtilities {
4
5    public static void main(String[] args) {
6      Person p = new Person("Mario",101);
7      Teacher t = new Teacher("Gino",102,"LMC","OOP");
8      System.out.println(objectToString(p));
9      System.out.println(objectToString(p,"name"));
10     System.out.println(objectToString(p,"name","id"));
11     System.out.println(objectToString(t,"name","id","courses"));
12   }
13
14 }
```

```
1  Person
2  Person:  Name -> Mario
3  Person:  Name -> Mario | Id -> 101
4  Teacher:  Name -> Gino | Id -> 102 | Courses -> [LMC, OOP]
```

Outline

Il Java annotation framework

Annotazioni in Java

- Sono un meccanismo usato per "annotare" pezzi di codice
- Il compilatore di default ignora queste annotazioni
- A run-time, via reflection, è possibile verificare quali annotazioni e dove sono presenti
- È anche possibile istruire il compilatore a rigettare annotazioni mal formate
- Java fornisce alcune annotazioni standard

Motivazioni

- Rendere il linguaggio più flessibile
- Consentire di realizzare piccola aggiunte "programmate" al linguaggio

Un primo esempio di annotazione: `Override`

```
1 class MyClass extends SuperClass{
2     ...
3     @Override
4     public void myMethod(...){...}
```

Elementi principali

- Abbiamo annotato con `@Override` il metodo `myMethod`
- `javac` è istruito a rigettare questo codice se `SuperClass` non definisce `myMethod`
  - stessa cosa quando si implementa il metodo di una interfaccia
- Serve a evitare errori di nome nell'indicazione di `myMethod`
- È prassi usarli sempre quando si fa overriding
- Eclipse li aggiunge e segnala eventuali errori

Esempio @Override

```
1 public class A {
2   void metodo(int x){}
3 }
4
5 class B extends A{
6   @Override
7   void metod(int x){} // Il compilatore segnala errore
8 }
```

Altre annotazioni di libreria

Cosa sono le annotazioni?

- Sono indicabili come sorta di interfacce
- Ogni package può esporre le proprie
- Le librerie di Java ne espongono varie

`java.lang.Override`

- Controllo statico del corretto overriding

`java.lang.SuppressWarnings`

- Dichiara del codice essere corretto: non genererà warning!
- Vuole come parametro il warning da disabilitare

Altri

- `java.lang.Deprecated`: marca "deprecato" il metodo
- Varie definite nel package `java.lang.annotation`

Esempio @SuppressWarnings

```
1  public class Vector<X>{
2      private Object[] elements = new Object[10]; // Deposito elementi
3      private int size = 0;   // Numero di elementi
4
5      public void addElement(X e){
6        if (this.size == elements.length){
7            this.expand();  // Se non c'è spazio
8        }
9        this.elements[this.size] = e;
10       this.size++;
11     }
12
13     @SuppressWarnings("unchecked")
14     public X getElementAt(int position){
15       return (X)this.elements[position];
16     }
17
18     public int getLength(){
19       return this.size;
20     }
21
22     private void expand(){  // Raddoppio lo spazio..
23       Object[] newElements = new Object[this.elements.length*2];
24       for (int i=0; i < this.elements.length; i++){
25           newElements[i] = this.elements[i];
26       }
27       this.elements = newElements;
28     }
29 }
```


Annotazioni custom

Definire le proprie annotazioni

- È possibile definire le proprie annotazioni, con una sintassi che ricalca molto da vicino quella delle interfacce
- È possibile via reflection capire quali metodi/campi/classi/costruttori sono stati annotati

Dove si possono inserire?

Per annotare la dichiarazione di classi, campi, metodi e costruttori
Non è al momento consentito annotare anche i tipi mentre li si usano

Come li si dichiara

- L'uso di una annotazione genera un oggetto ispezionabile
- L'annotazione dichiara l'interfaccia di tale oggetto, e come lo si deve inizializzare
- Si forniscono anche informazioni ulteriori

Un esempio di applicazione

Riprendiamo l'esempio `objectToString()`

- Costruiamo una gestione delle annotazioni che permetta di annotare alcuni metodi getter
- E costruiamo una nuova `objectToString` che stampi controllando tali annotazioni

Dichiarazione `@ToString`

```
1  package it.unibo.apice.oop.p14reflection.annotations;
2
3  import java.lang.annotation.Documented;
4  import java.lang.annotation.ElementType;
5  import java.lang.annotation.Retention;
6  import java.lang.annotation.RetentionPolicy;
7  import java.lang.annotation.Target;
8
9  @Documented            // genera documentazione javadoc
10 @Target(ElementType.METHOD)   // potrà essere usato solo nei metodi
11 @Retention(RetentionPolicy.RUNTIME)   // sarà visibile a run-time
12 public @interface ToString {
13   boolean showMethodName() default true;    // proprietà
       specificabili
14   String customName() default "";
15   String associationSymbol() default "->";
16   String separator() default "|";
17 }
```

Esempio di come si usa l'annotazione

```java
public class Person {

  private String name;
  private int id;

  public Person(String name, int id) {
    super();
    this.name = name;
    this.id = id;
  }

  @ToString
  public String getName() {
    return name;
  }

  @ToString
  public int getId() {
    return id;
  }
}
```

Altro esempio

```java
public class Product {

  private String name;
  private int id;
  private double quantity;

  public Product(String name, int id, double quantity) {
    this.name = name;
    this.id = id;
    this.quantity = quantity;
  }

  @ToString( separator = " " )
  public String getName() {
    return name;
  }

  @ToString( showMethodName = false, associationSymbol = ":", separator = "
    " )
  public int getId() {
    return id;
  }

  @ToString( showMethodName = false, customName = "qty", separator = " " )
  public double getQuantity() {
    return quantity;
  }
}
```

Metodo objectToString()

```java
import java.lang.reflect.Method;

public class PrintObjectsUtilities {

  public static String objectToString(Object o) {
    try {
      String out = "";
      for (Method m : o.getClass().getMethods()) {
        if (m.isAnnotationPresent(ToString.class) && m.getParameterTypes().length==0){
          ToString annotation = m.getAnnotation(ToString.class);
          if (annotation.showMethodName()){
            out += m.getName() + annotation.associationSymbol();
          }
          if (!annotation.showMethodName() && annotation.customName()!=null){
            out += annotation.customName() + annotation.associationSymbol();
          }
          out += m.invoke(o) + annotation.separator();
        }
      }
      return out;
    } catch (Exception e) {
      return null;
    }
  }
}
```

Uso objectToString()

```java
public class UsePrintObjectsUtilities {
  public static void main(String[] args) {
    Person p = new Person("Marco", 100);
    System.out.println(PrintObjectsUtilities.objectToString(p));
    // getName->Marco|getId->100|

    Product pr = new Product("Pr", 200, 100000);
    System.out.println(PrintObjectsUtilities.objectToString(pr));
    // qty->100000.0 getName->Pr :200
  }
}
```

Un approfondimento: `AnnotationProcessor`

I processori di annotazioni

- Sono delle classi che è possibile costruire, che vengono indicate a `javac`
- Sono usate da `javac` per verificare se stiamo usando certe annotazioni nel modo giusto
- Hanno accesso completo ai dettagli interni del compilatore, quindi possono verificare errori e anche "interpretare" a piacimento le annotazioni

La classe `javax.annotation.processor.AnnotationProcessor`

- É una classe astratta che richiede di implementare un metodo `process()`
- Questo riceve le informazioni sullo stato del codice che si compila
- Da questo è possibile controllare l'uso delle annotazioni
- Serve conoscere le API interne del compilatore

`ToStringChecker`: preambolo

```
1  package annotations;
2
3  import javax.annotation.processing.AbstractProcessor;
4  import javax.annotation.processing.RoundEnvironment;
5  import javax.annotation.processing.SupportedAnnotationTypes;
6  import javax.annotation.processing.SupportedSourceVersion;
7  import javax.lang.model.element.TypeElement;
8  import javax.lang.model.element.Element;
9  import javax.lang.model.SourceVersion;
10 import javax.tools.Diagnostic.Kind;
11 import com.sun.tools.javac.code.TypeTags;
12 import com.sun.tools.javac.code.Symbol.MethodSymbol;
13 import java.util.Set;
14
15 @SupportedSourceVersion(SourceVersion.RELEASE_7)
16 @SupportedAnnotationTypes("it.unibo.apice.oop.p13annotations.
       ToString")
17 public class ToStringChecker extends AbstractProcessor {
```

ToStringChecker: classe

```java
public class ToStringChecker extends AbstractProcessor {

    @Override
  public boolean process(Set<? extends TypeElement> annotations,
                         RoundEnvironment roundEnv) {
      boolean out = true;
      for (Element e : roundEnv.getElementsAnnotatedWith(ToString.class)){
        if (!(e instanceof MethodSymbol) ||
           !((MethodSymbol)e).params.isEmpty() ||
           ((MethodSymbol)e).getReturnType().tag==TypeTags.VOID){
          out = false;
          String msg = "@ToString cannot be applied to "+e+": ";
          msg + = "it must be a getter method";
        processingEnv.getMessager().printMessage(Kind.ERROR,msg);
      }
    }
    return out;
  }

}
// Dettagli compilazione da linea di comando:
// javac -classpath .:/usr/lib/jvm/java-7-openjdk-amd64/lib/tools.jar\
// -processorpath ../bin\
// -processor it.unibo.apice.oop.p13annotations.ToStringChecker\
// it/unibo/apice/oop/p13annotations/*.java
// error: @ToString cannot be applied to m(): it must be a getter method
// 1 error
```


Outline

Il framework per i collaudi software JUnit

Una applicazione delle annotazioni di Java... Idea:

- Creare delle classi di test, con metodi che realizzano degli scenari d'uso di certe classi da testare
- Tali metodi sono propriamente annotati
- Alla fine del loro lavoro tali metodi asseriscono se il risultato atteso è giusto
- Da Eclipse semplice esecuzione della classe come "JUnit test"

Per verificare se e quanti test passano.. da linea di comando

- Compilazione specificando la libreria JUnit
- Esecuzione del framework JUnit

```
javac -cp .:/opt/java/junit-4.11.jar *.java
java -cp .:/opt/java/junit-4.11.jar:\
/opt/java/hamcrest-core-1.3.jar\
org.junit.runner.JUnitCore CounterTest
```


Classe per il collaudo di un contatore

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

import org.junit.Test;

public class CounterTest {

  @Test
  public void test1() {
    Counter c = new Counter();
    c.increment();
    c.increment();
    assertTrue("Increment does not work wrt getValue", c.getValue()==2);
  }

  @Test
  public void test2() {
    Counter c = new Counter();
    assertTrue("getValue is not initially zero", c.getValue()==0);
  }


}
```

Altre modalità di asserire il risultato di un test

Si veda: https://github.com/junit-team/junit/wiki/Assertions

```
1  // many things to import
2
3  public class AssertTests {
4    @Test
5    public void testAssertFalse() {
6      org.junit.Assert.assertFalse("failure - should be false", false);
7    }
8
9    @Test
10   public void testAssertSame() {
11     Integer aNumber = Integer.valueOf(768);
12     org.junit.Assert.assertSame("should be same", aNumber, aNumber);
13   }
14
15   // JUnit Matchers assertThat
16   @Test
17   public void testAssertThatEveryItemContainsString() {
18     org.junit.Assert.assertThat(
19         Arrays.asList(new String[] { "fun", "ban", "net" }),
20         everyItem(containsString("n")));
21   }
22   ..
23 }
```