14
Reflection
e informazioni run-time sui tipi, annotazioni e testing

Mirko Viroli
mirko.viroli@unibo.it

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

a.a. 2025/2026

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 desunto (per compilazione) da quello del `.java`, solo espresso in un linguaggio diverso che garantisce la correttezza del contenuto e le performance di chi deve interpretarlo (JVM)

È possibile ispezionare il contenuto di un `.class`

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

Classe Counter

```
1  class Counter{
2
3      /* Il campo è reso inaccessibile direttamente */
4      private int countValue;
5
6      /* E' il costruttore che inizializza i campi */
7      public Counter(){
8        this.countValue=0;
9      }
10
11     /* Unico modo per osservare lo stato */
12     public int getValue(){
13       return this.countValue;
14     }
15
16     /* Unico modo per modificare lo stato */
17     public void increment(){
18       this.countValue++;
19     }
20 }
```

Contenuto del classfile (1/2)

```
Classfile .../Counter.class
  Last modified Sep 20, 2018; 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++
- HotSpot di OpenJDK ($>$ 250K linee): `https://hg.openjdk.java.net/jdk/`
- 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 (e chiamando l'inizializzatore statico se definito)

Nota: esiste anche l'inizializzatore "non statico"

- viene richiamato all'atto della creazione dell'oggetto

Una parentesi: gli `initializer`

Static initializer

- sintassi `static{...}`
- eseguito "subito", prima di qualunque accesso alla classe
- in genere inizializza campi statici
- potrebbe eseguire altro codice di inizializzazione

Non-static initializer

- sintassi `{...}`
- eseguito prima di chiamare il costruttore
- in genere inizializza campi istanza
- potrebbe eseguire altro codice di inizializzazione

Inizializzatori all'opera

```java
public class Initializers {

  static { // static initializer.. è eseguito "subito"
    System.out.println("Static initializer executed...");
    SET = Set.of(10,20,30); // ad esempio inizializza i campi statici
  }

  { // non-static initializer.. è eseguito prima del costruttore
    System.out.println("Non-static initializer executed...");
    set = Set.of(10,20,30,40);
  }

  Initializers(){
    System.out.println("Constructor executed...");
  }

  private static final Set<Integer> SET;
  private final Set<Integer> set;
  // use of non-static initializers with anonymous classes
  private final Set<Integer> set2 = new HashSet<>() {{ add(1); add(2); }};

  public static void main(String[] args) {
    System.out.println("Let's create an object..");
    Initializers i = new Initializers();
    System.out.println(i.set2);
  }

}
```

Ispezioniamo la dinamica di caricamento (e istanziazione)

```
1  class A{
2    static { System.out.println("A.class caricato");}
3    { System.out.println("A.class istanziato"); }
4    { System.out.println("A.class istanziato"); }
5  }
6
7  class B extends A{
8    static { System.out.println("B.class caricato");}
9    { System.out.println("B.class istanziato"); }
10 }
11
12 public class Loading {
13   static { System.out.println("Loading.class caricato");}
14
15   public static void main(String[] args) {
16     System.out.println("main partito");
17     new B();
18     System.out.println("creato un oggetto di B");
19   }
20   /*  Loading.class caricato
21     main partito
22     A.class caricato
23     B.class caricato
24     A.class istanziato
25     A.class istanziato
26     B.class istanziato
27     creato un oggetto di B  */
28 }
```

Perché questa gestione "by-need" del caricamento?

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
- In scenari avanzati si potrebbero anche "scaricare" (togliere dalla JVM) le classi che sembrano non servire più, o addirittura fare "hot-swapping" (modifica di una classe)
- Alcune classi potrebbe essere aggiunte al volo per aumentare funzionalità senza spegnere l'applicazione
- 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 (ma ottimizzabile)
- non interagisce con i controlli del compilatore (genera eccezioni..)
- pone problemi di security (che non analizzeremo in dettaglio..)
- consente di programmare accessi a classi non note a priori

$\Rightarrow$ 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
- Interagire con oggetti in modo dnamico, bypassando i test del compilatore

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 tre 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>`, che avrà quindi metodi che "sanno" di lavorare su stringhe (ad esempio per creare oggetti di tipo stringa)
- Altrimeni restituisce un `Class<?>`, e quindi prima o poi serviranno delle conversioni (con possibili cast "unchecked")

Esempi

```java
public class TryClass {

  public static void main(String[] args) throws Exception {

    // Unico caso di recupero del corretto tipo generico
    final Class<String> c = String.class;
    System.out.println(c);
    System.out.println("oggetto "+c.newInstance()); // deprecata

    // Si può ottenere una class da una stringa calcolata
    final Class<?> c2 = Class.forName("java.lang" + ".String");
    System.out.println("oggetto "+(String)c2.newInstance());

    // Accesso alla classe di un oggetto
    final Object o = "3";
    final Class<?> c3 = o.getClass();
    System.out.println(c3); //String

    // Cast "unchecked" per recuperare il generico
    final Class<Integer> c4 = (Class<Integer>)o.getClass();
    System.out.println(c4);

    // Cast "unchecked" per recuperare il generico
    String s = cloneObject("prova");
    java.util.Date d = cloneObject(new java.util.Date());
    System.out.println(s.length()+" "+d);
  }

  private static <T> T cloneObject(T t) throws Exception {
    return (T)t.getClass().newInstance();
  }
}
```

Esempi

```java
public class UseClass {
  public static void main(String[] args) throws ClassNotFoundException {
    final Class<String> c = String.class;
    System.out.println(c.getName()+" "+c.getCanonicalName());
    //java.lang.String java.lang.String
    final Class<Integer> ci = (Class<Integer>)Integer.valueOf(5).getClass();
    System.out.println(ci.getName()+" "+ci.getCanonicalName());
    //java.lang.Integer java.lang.Integer
    final Class<?> ca = new int[20].getClass();
    System.out.println(ca.getName()+" "+ca.getCanonicalName());
    //[I int[]
    final Class<?> cint = ca.getComponentType();
    System.out.println(cint.getName()+" "+cint.getCanonicalName());
    //int int
    final Class<?> cl = Class.forName("java.util.List");
    System.out.println(cl.getName()+" "+cl.getCanonicalName());
    //java.util.List java.util.List
    final Class<?> can = new Object(){
      public String toString(){ return "none";}
    }.getClass();
    System.out.println(can.getName()+" "+can.getCanonicalName());
    //it.unibo.apice.oop.p16reflection.UseClass$1 null
  }
}
```

Alcuni metodi di java.lang.Class

```
1  public final class Class<T> implements ...  {
2
3      public static Class<?> forName(String className) throws ClassNotFoundException {...}
4
5      public boolean isInterface();
6      public boolean isArray();
7      public boolean isPrimitive();
8      public boolean isAnonymousClass() {...}
9      public boolean isLocalClass() {...}
10     public boolean isMemberClass() {...}
11     public boolean isEnum() {...}
12
13     public T[] getEnumConstants() {...}
14     public Class<?> getComponentType();
15     public T cast(Object obj) {...}
16
17     public T newInstance() throws ...{...}
18
19     // Accessing the structure.. can throw SecurityException
20     public Field[] getFields() throws ... {...}
21     public Method[] getMethods() throws ... {...}
22     public Constructor<?>[] getConstructors() throws ... {...}
23
24     public Field getField(String name) throws NoSuchFieldException, .. {...}
25     public Method getMethod(String name, Class<?>... parameterTypes)
26         throws NoSuchMethodException, .. {...}
27     public Constructor<T> getConstructor(Class<?>... parameterTypes)
28         throws NoSuchMethodException, .. {...}
29
30     public Field[] getDeclaredFields() throws SecurityException {...}
31     ... // All versions with 'Declared'
32 }
```

Alcuni metodi di java.lang.reflect.Field

```
1  public final class Field extends ... {
2
3      public Class<?> getDeclaringClass() {...}
4      public String getName() {...}
5      public int getModifiers() {...}
6      public boolean isEnumConstant() {...}
7      public Class<?> getType() {...}
8
9      public Object get(Object obj)
10         throws IllegalArgumentException, IllegalAccessException{...}
11     public boolean getBoolean(Object obj)
12         throws IllegalArgumentException, IllegalAccessException{...}
13     public byte getByte(Object obj)
14         throws IllegalArgumentException, IllegalAccessException{...}
15     ...
16
17     public void set(Object obj, Object value)
18         throws IllegalArgumentException, IllegalAccessException{...}
19     ...
20 }
```

Alcuni metodi di java.lang.reflect.Constructor

```
1  public final class Constructor<T> extends ... {
2
3      public Class<T> getDeclaringClass() {...}
4      public String getName() {...}
5      public int getModifiers() {...}
6      public Class<?>[] getParameterTypes() {...}
7      public Class<?>[] getExceptionTypes() {...}
8
9      public T newInstance(Object ... initargs)
10         throws InstantiationException, IllegalAccessException,
11                IllegalArgumentException, InvocationTargetException {...}
12
13     public boolean isVarArgs() {...}
14     ...
15 }
```

Alcuni metodi di java.lang.reflect.Method

```
1  public final class Method extends ... {
2
3      public Class<?> getDeclaringClass() {...}
4      public String getName() {...}
5      public int getModifiers() {...}
6      public Class<?> getReturnType() {...}
7      public Class<?>[] getParameterTypes() {...}
8      public Class<?>[] getExceptionTypes() {...}
9      public boolean isVarArgs() {...}
10
11     public Object invoke(Object obj, Object... args)
12         throws IllegalAccessException, IllegalArgumentException,
13            InvocationTargetException {...}
14 }
```

Esempio di chiamata dinamica di metodo

```
1  public class UseReflection {
2
3    private static Object callGetter(Object receiver, String getterName)
       throws Exception {
4      return receiver.getClass().getMethod(getterName).invoke(receiver);
5    }
6
7    public static void main(String[] args) throws Exception {
8      System.out.println(callGetter("prova","isEmpty"));
9      System.out.println(callGetter(new java.util.Date(),"getDate"));
10     System.out.println(callGetter(new Object(),"hashCode"));
11
12     class A{
13       public void createDialog() {
14         // creazione di un pannello di "dialogo", con Java Swing
15         javax.swing.JOptionPane.showMessageDialog(null, "It worked!");
16       }
17     }
18     System.out.println(callGetter(new A(),"createDialog"));
19     //System.out.println(callGetter(new A(),"createDial"));
20   }
21 }
```

Esempio di creazione dinamica di un oggetto

```
1  public class UseReflection2 {
2
3    private static void check(boolean condition, Class<? extends
       RuntimeException> exClass, String message){
4      if (!condition) {
5        try {
6          throw exClass.getConstructor(String.class).newInstance(message);
7        } catch (InstantiationException | IllegalAccessException
8            | IllegalArgumentException  | InvocationTargetException
9            | NoSuchMethodException | SecurityException e1) {
10         throw new IllegalArgumentException("incorrect class type");
11       }
12     }
13   }
14
15   private static int getFromArray(int[] a, int index){
16     check(a != null, NullPointerException.class, "Array can't be null");
17     check(index >= 0 && index < a.length, IndexOutOfBoundsException.class,
18         "Index "+index+ " out of bounds");
19     return a[index];
20   }
21
22   public static void main(String[] args) {
23     getFromArray(new int[]{10,20,30},4);
24   }
25 }
```

Reflection e esecuzione "dinamica"

Applicazioni dinamiche

- Sono applicazioni che riescono ad eseguire codice aggiunto dinamicamente dopo che l'applicazione stessa è partita
- Questo tipo di meccanismo è molto importante in sistemi che non è possibile "spegnere"

Tecnica

- Un gestore dell'applicazione compila nuove classi e le aggiunge al CLASSPATH
- L'applicazione è già stata costruita in modo da accedere a certe classi da eseguire tramite il loro nome, eseguendone certi metodi

Caricamento ed esecuzione automatica

```
1
2  public class DynamicExecution {
3
4    private static final String Q_CLASS = "Insert fully-qualified class name: ";
5    private static final String Q_METH = "Insert name of method to call: ";
6    private static final String L_OK = "Everything was ok! The result is..";
7    private static final String E_RET = "Wrong return type";
8
9    public static void main(String[] s) throws Exception {
10     while (true) {
11       System.out.println(Q_CLASS);
12       final String className = System.console().readLine();
13       System.out.println(Q_METH);
14       final String methName = System.console().readLine();
15       final Class<?> cl = Class.forName(className); // Ottiene la classe
16       final Constructor<?> cns = cl.getConstructor(); // Ottiene il costruttore
17       final Method met = cl.getMethod(methName);      // Ottiene il metodo
18       if (!met.getReturnType().isAssignableFrom(String.class)) {
19         throw new NoSuchMethodException(E_RET);   // Il metodo deve tornare String
20       }
21       final Object o = cns.newInstance();             // Istanzia l'oggetto
22       final String result = (String) met.invoke(o);   // Chiama il metodo
23       System.out.println(L_OK);
24       System.out.println(result);
25       System.out.println();
26     }
27   }
28 }
```

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 soluzione 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 della proprietà da ritrovare (assumendo ci sia un getter)
- Ritorna la stringa creata

objectToString

```java
public class PrintObjectsUtilities {

  public static String objectToString(Object o, String... getters)
      throws Exception {
    String out = o.getClass().getSimpleName() + ": ";
    for (String getter : getters) {
      // Sistemo la maiuscola iniziale
      getter = getter.substring(0, 1).toUpperCase() + getter.substring(1);
      // Trovo il getter e lo invoco
      final Method m = o.getClass().getMethod("get" + getter);
      final Object res = m.invoke(o);
      // Aggiungo la stringa
      out += " " + getter + " -> ";
      out += res.getClass().isArray()
          ? Arrays.deepToString((Object[]) res) : res.toString();
      out += " |";
    }
    return out.substring(0, out.length() - 2);
  }
}
```

Classi di prova

```java
public class Person {
    ...

  public Person(String name, int id) {..}

  public String getName() {..}

  public int getId() {..}

}
```

```java
public class Teacher extends Person {
    ...

  public Teacher(String name, int id, String... courses) {..}

  public String[] getCourses() {..}
}
```

Uso di objectToString

```java
import static it.unibo.apice.oop.p14reflection.classes.PrintObjectsUtilities
    .*;

public class UsePrintObjectUtilities {

  public static void main(String[] args) throws Exception {
    Person p = new Person("Mario",101);
    Teacher t = new Teacher("Gino",102,"PC","OOP");
    System.out.println(objectToString(p));
    System.out.println(objectToString(p,"name"));
    System.out.println(objectToString(p,"name","id"));
    System.out.println(objectToString(t,"name","id","courses"));

    Counter c = new Counter();
    c.increment();
    c.increment();
    System.out.println(objectToString(c,"value"));

  }

}
```

```
Person
Person:  Name -> Mario
Person:  Name -> Mario | Id -> 101
Teacher:  Name -> Gino | Id -> 102 | Courses -> [PC, 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 piccole 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
- `java.lang.FunctionalInterface`: verifica che un solo metodo richieda implementazione
- 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     }
```

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

```java
package it.unibo.apice.oop.p14reflection.annotations;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Documented // genera documentazione javadoc
@Target(ElementType.METHOD) // potrà essere usato solo nei metodi
@Retention(RetentionPolicy.RUNTIME) // sarà visibile a run-time
public @interface ToString {
    String customName() default ""; // propr. specificabili

    String associationSymbol() default "->";

    String separator() default ",";
}
```

Esempio di come si usa l'annotazione

```
1  public class Person {
2
3    private final String name;
4    private final int id;
5
6    public Person(final String name, final int id) {
7      this.name = name;
8      this.id = id;
9    }
10
11   @ToString
12   public String getName() {
13     return name;
14   }
15
16   @ToString
17   public int getId() {
18     return id;
19   }
20 }
```

Altro esempio

```java
public class Product {

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

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

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

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

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

Metodo objectToString()

```
1  import java.lang.reflect.Method;
2
3  public class PrintObjectsUtilities {
4
5    public static String objectToString(Object o) {
6      try {
7        String out = "";
8        for (final Method m : o.getClass().getMethods()) {
9          if (m.isAnnotationPresent(ToString.class) && m.getParameterTypes().length==0){
10           final ToString annotation = m.getAnnotation(ToString.class);
11           out += annotation.customName().equals("") ? m.getName()
12                                                     : annotation.customName();
13           out += annotation.associationSymbol();
14           out += m.invoke(o) + annotation.separator() + " ";
15         }
16       }
17       return out;
18     } catch (Exception e) {
19       return null;
20     }
21   }
22 }
```

Uso objectToString()

```java
public class UsePrintObjectsUtilities {

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

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

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", dopo aver collegato la libreria Junit al "build path" del progetto

Classe per il collaudo di un contatore

```java
package it.unibo.apice.oop.p14reflection.classes;

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

public class CounterTest {

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

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


}
```

Testiamo il RangeIterator: e modifichiamo se serve..

```
1  public class RangeIterator implements java.util.Iterator<Integer>{
2
3      private int current;
4      final private int stop;
5
6      public RangeIterator(final int start, final int stop){
7        if (start>stop){
8          throw new IllegalArgumentException();
9        }
10       this.current = start;
11       this.stop = stop;
12     }
13
14     public Integer next(){
15       if (!this.hasNext()){  // next() oltre i limiti
16           throw new java.util.NoSuchElementException();
17       }
18       return this.current++;
19     }
20
21     public boolean hasNext(){
22       return this.current <= this.stop;
23     }
24
25     public void remove(){
26       throw new UnsupportedOperationException();
27     }
28 }
```

Possibile codice JUnit: 1/3

```java
package it.unibo.apice.oop.p14reflection.annotations;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.function.Executable;
import static org.junit.jupiter.api.Assertions.*;
import java.util.*;

public class TestRangeIterator {

  @Test
  public void testStandardBehaviour() {
    final Iterator<Integer> it = new RangeIterator(5,7);
    assertTrue(it.hasNext());
    assertEquals(5,it.next().intValue());
    assertTrue(it.hasNext());
    assertEquals(6,it.next().intValue());
    assertTrue(it.hasNext());
    assertEquals(7,it.next().intValue());
    assertFalse(it.hasNext());
  }

  @Test
  public void testStartEqualsStop() {
    Iterator<Integer> it = new RangeIterator(5,5);
    assertTrue(it.hasNext());
    assertEquals(5,it.next().intValue());
    assertFalse(it.hasNext());
  }
```

Possibile codice JUnit: 2/3

```
1   @Test
2   public void testRemoveRaisesException() {
3     Iterator<Integer> it = new RangeIterator(5,10);
4     try {
5       it.remove();
6       fail();
7     } catch (UnsupportedOperationException e) {
8     }
9   }
10
11  // Variante migliorativa
12  @Test
13  public void testRemoveRaisesException2() {
14    Iterator<Integer> it = new RangeIterator(5,10);
15    assertThrows(UnsupportedOperationException.class, new Executable() {
16      @Override
17      public void execute() throws Throwable {
18        it.remove();
19      }
20    });
21    // with lambdas:
22    // assertThrows(UnsupportedOperationException.class, () -> {it.remove()
      ;});
23  }
```

Possibile codice JUnit: 3/3

```
1   @Test
2   public void testNoSuchElement() {
3     Iterator<Integer> it = new RangeIterator(5,6);
4     assertThrows(NoSuchElementException.class, new Executable() {
5       @Override
6       public void execute() throws Throwable {
7         it.next();
8         it.next();
9         it.next();
10      }
11    });
12
13  }
14
15  @Test
16  public void testIllegalArguments() {
17    assertThrows(IllegalArgumentException.class, new Executable() {
18      @Override
19      public void execute() throws Throwable {
20        new RangeIterator(6,4);
21      }
22    });
23  }
24 }
```

Altro esempio: esercizio Graph

```java
import java.util.*;

public interface Graph<N> {

  // Adds a node: nothing happens if node is null or already there
  void addNode(N node);

  // Adds an edge: nothing happens if source or target are null
  void addEdge(N source, N target);

  // Returns all the nodes
  Set<N> nodeSet();

  // Returns all the nodes directly targeted from node
  Set<N> linkedNodes(N node);

  // Gets one sequence of nodes connecting source to path
  List<N> getPath(N source, N target);

}
```

Un esempio di semplicissimo test

```
1
2    @Test
3    public void testNodeSet() {
4      final Graph<String> g = null; //new GraphImpl<>();
5      assertTrue(g.nodeSet().isEmpty());
6
7      g.addNode("a");
8      g.addNode("b");
9      g.addNode("c");
10     g.addNode("d");
11     g.addNode("e");
12     assertEquals(5, g.nodeSet().size());
13     assertEquals(Set.of("a","b","c","d","e"), g.nodeSet());
14     assertTrue(g.linkedNodes("c").isEmpty());
15
16     g.addEdge("a","b");
17     g.addEdge("b","c");
18     g.addEdge("c","d");
19     g.addEdge("d","e");
20     g.addEdge("c","a");
21     assertEquals(2, g.linkedNodes("c").size());
22     assertEquals(Set.of("a","d"), g.linkedNodes("c"));
23     assertEquals(List.of("b","c","a"), g.getPath("b", "a"));
24   }
25
26 }
```

Variante migliorativa (1/2)

```java
public class TestGraphImplImproved {

    private Graph<String> g;

    @BeforeEach     // initalisation code executed prior to each test
    public void initialise() {
        //g = new GraphImpl<>();
        g.addNode("a");
        g.addNode("b");
        g.addNode("c");
        g.addNode("d");
        g.addNode("e");
        g.addEdge("a", "b");
        g.addEdge("b", "c");
        g.addEdge("c", "d");
        g.addEdge("d", "e");
        g.addEdge("c", "a");
    }

    @Test
    public void testNodeSetSize() {
        assertEquals(5, g.nodeSet().size());
    }

    @Test
    public void testNodeSetElements() {
        assertEquals(Set.of("a", "b", "c", "d", "e") , g.nodeSet());
    }
```

Variante migliorativa (2/2)

```
1
2       @Test
3       public void testLinkedNodes() {
4           assertEquals(2, g.linkedNodes("c").size());
5           assertEquals(Set.of("a", "d") , g.linkedNodes("c"));
6       }
7
8       @Test
9       public void testGetPath() {
10          assertEquals(List.of("b", "c", "a"), g.getPath("b", "a"));
11      }
12  }
```

Sul testing con JUnit

In generale

- il testing è un aspetto fondamentale della software engineering
- il testing è un aspetto fondamentale della programmazione
- JUnit è uno strumento estremamente potente
- tutte queste questioni verranno affrontate alla magistrale

Linee guida base per i vostri test

- ogni classe abbia una sua classe di test
- la classe di test sia scritta molto bene e sia comprensibile
- si abbiano metodi di test corti, specifici e con un buon nome
- si cerchi massima copertura dei test