

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. 2022/2023

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

- 1 Classi, caricamento e JVM
- 2 Reflection API
- 3 Un esempio di applicazione
- 4 Annotazioni
- 5 JUnit

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)

```
1 Classfile .../Counter.class
2   Last modified Sep 20, 2018; size 361 bytes
3   MD5 checksum 2c941be1d3cfd60de02143767ce146cc
4   Compiled from "Counter.java"
5   class Counter
6     SourceFile: "Counter.java"
7     minor version: 0
8     major version: 51
9     flags: ACC_SUPER
10  Constant pool:
11    #1 = Methodref          #4.#16          // java/lang/Object."<init>":()V
12    #2 = Fieldref           #3.#17          // Counter.countValue:I
13    #3 = Class               #18            // Counter
14    #4 = Class               #19            // java/lang/Object
15    #5 = Utf8                countValue
16    #6 = Utf8                I
17    #7 = Utf8                <init>
18    #8 = Utf8                ()V
19    #9 = Utf8                Code
20    #10 = Utf8               LineNumberTable
21    #11 = Utf8               getValue
22    #12 = Utf8               ()I
23    #13 = Utf8               increment
24    #14 = Utf8               SourceFile
25    #15 = Utf8               Counter.java
26    #16 = NameAndType        #7:#8          // "<init>":()V
27    #17 = NameAndType        #5:#6          // countValue:I
28    #18 = Utf8               Counter
29    #19 = Utf8               java/lang/Object
```

Contenuto del classfile (2/2)

```
1 { public Counter();
2   flags: ACC_PUBLIC Code:
3     stack=2, locals=1, args_size=1
4     0: aload_0
5     1: invokespecial #1           // Method java/lang/Object."<init>":()V
6     4: aload_0
7     5: iconst_0
8     6: putfield      #2           // Field countValue:I
9     9: return
10   LineNumberTable:          line 7: 0 line 8: 4          line 9: 9
11
12  public int getValue();
13    flags: ACC_PUBLIC Code:
14      stack=1, locals=1, args_size=1
15      0: aload_0
16      1: getfield      #2           // Field countValue:I
17      4: ireturn
18    LineNumberTable:         line 13: 0
19  public void increment();
20    flags: ACC_PUBLIC Code:
21      stack=3, locals=1, args_size=1
22      0: aload_0
23      1: dup
24      2: getfield      #2           // Field countValue:I
25      5: iconst_1
26      6: iadd
27      7: putfield      #2           // Field countValue:I
28     10: return
29    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

```
1 public class Initializers {
2
3     static { // static initializer.. è eseguito "subito"
4         System.out.println("Static initializer executed...");
5         SET = Set.of(10,20,30); // ad esempio inizializza i campi statici
6     }
7
8     { // non-static initializer.. è eseguito prima del costruttore
9         System.out.println("Non-static initializer executed...");
10        set = Set.of(10,20,30,40);
11    }
12
13    Initializers(){
14        System.out.println("Constructor executed...");
15    }
16
17    private static final Set<Integer> SET;
18    private final Set<Integer> set;
19    // use of non-static initializers with anonymous classes
20    private final Set<Integer> set2 = new HashSet<>() {{ add(1); add(2); }};
21
22    public static void main(String[] args) {
23        System.out.println("Let's create an object..");
24        Initializers i = new Initializers();
25        System.out.println(i.set2);
26    }
27
28 }
```

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

- 1 Classi, caricamento e JVM
- 2 Reflection API**
- 3 Un esempio di applicazione
- 4 Annotazioni
- 5 JUnit

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

⇒ va usata di conseguenza.. quindi non abusarne

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 dinamico, 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)
- Altrimenti restituisce un `Class<?>`, e quindi prima o poi serviranno delle conversioni (con possibili cast “unchecked”)

Esempi

```
1 public class TryClass {
2
3     public static void main(String[] args) throws Exception {
4
5         // Unico caso di recupero del corretto tipo generico
6         final Class<String> c = String.class;
7         System.out.println(c);
8         System.out.println("oggetto "+c.newInstance()); // deprecata
9
10        // Si può ottenere una class da una stringa calcolata
11        final Class<?> c2 = Class.forName("java.lang" + ".String");
12        System.out.println("oggetto "+(String)c2.newInstance());
13
14        // Accesso alla classe di un oggetto
15        final Object o = "3";
16        final Class<?> c3 = o.getClass();
17        System.out.println(c3); //String
18
19        // Cast "unchecked" per recuperare il generico
20        final Class<Integer> c4 = (Class<Integer>)o.getClass();
21        System.out.println(c4);
22
23        // Cast "unchecked" per recuperare il generico
24        String s = cloneObject("prova");
25        java.util.Date d = cloneObject(new java.util.Date());
26        System.out.println(s.length()+" "+d);
27    }
28
29    private static <T> T cloneObject(T t) throws Exception {
30        return (T)t.getClass().newInstance();
31    }
32 }
```

Esempi

```
1 public class UseClass {
2     public static void main(String[] args) throws ClassNotFoundException {
3         final Class<String> c = String.class;
4         System.out.println(c.getName()+" "+c.getCanonicalName());
5         //java.lang.String java.lang.String
6         final Class<Integer> ci = (Class<Integer>)Integer.valueOf(5).getClass();
7         System.out.println(ci.getName()+" "+ci.getCanonicalName());
8         //java.lang.Integer java.lang.Integer
9         final Class<?> ca = new int[20].getClass();
10        System.out.println(ca.getName()+" "+ca.getCanonicalName());
11        //[I int[]
12        final Class<?> cint = ca.getComponentType();
13        System.out.println(cint.getName()+" "+cint.getCanonicalName());
14        //int int
15        final Class<?> cl = Class.forName("java.util.List");
16        System.out.println(cl.getName()+" "+cl.getCanonicalName());
17        //java.util.List java.util.List
18        final Class<?> can = new Object(){
19            public String toString(){ return "none";}
20        }.getClass();
21        System.out.println(can.getName()+" "+can.getCanonicalName());
22        //it.unibo.apice.oop.p16reflection.UseClass$1 null
23    }
24 }
```

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)
4         throws Exception {
5         return receiver.getClass().getMethod(getterName).invoke(receiver);
6     }
7
8     public static void main(String[] args) throws Exception {
9         System.out.println(callGetter("prova", "isEmpty"));
10        System.out.println(callGetter(new java.util.Date(), "getDate"));
11        System.out.println(callGetter(new Object(), "hashCode"));
12
13        class A{
14            public void createDialog() {
15                // creazione di un pannello di "dialogo", con Java Swing
16                javax.swing.JOptionPane.showMessageDialog(null, "It worked!");
17            }
18        }
19        System.out.println(callGetter(new A(), "createDialog"));
20        //System.out.println(callGetter(new A(), "createDial"));
21    }
22 }
```


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(); // Istanza 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

- 1 Classi, caricamento e JVM
- 2 Reflection API
- 3 Un esempio di applicazione**
- 4 Annotazioni
- 5 JUnit

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

```
1 public class PrintObjectsUtilities {
2
3     public static String objectToString(Object o, String... getters)
4         throws Exception {
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            final Method m = o.getClass().getMethod("get" + getter);
11            final Object res = m.invoke(o);
12            // Aggiungo la stringa
13            out += " " + getter + " -> ";
14            out += res.getClass().isArray() ? Arrays.deepToString((Object[]) res)
15                : res.toString();
16            out += " |";
17        }
18        return out.substring(0, out.length() - 2);
19    }
```

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 toString

```
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,"PC","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        Counter c = new Counter();
14        c.increment();
15        c.increment();
16        System.out.println(objectToString(c,"value"));
17    }
18 }
19
20 }
```

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

Outline

- 1 Classi, caricamento e JVM
- 2 Reflection API
- 3 Un esempio di applicazione
- 4 Annotazioni**
- 5 JUnit

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`

- Dichiarare 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

```
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     String customName() default ""; // propr. specificabili
14
15     String associationSymbol() default "->";
16
17     String separator() default ",";
18 }
```


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

```
1 public class Product {
2
3     private final String name;
4     private final int id;
5     private final double quantity;
6
7     public Product(final String name, final int id, final double quantity) {
8         this.name = name;
9         this.id = id;
10        this.quantity = quantity;
11    }
12
13    @ToString( separator = ";" )
14    public String getName() {
15        return name;
16    }
17
18    @ToString( separator = ":", associationSymbol = ":" )
19    public int getId() {
20        return id;
21    }
22
23    @ToString( separator = ";", customName = "qty" )
24    public double getQuantity() {
25        return quantity;
26    }
27 }
```

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
14                     out += annotation.associationSymbol();
15                     out += m.invoke(o) + annotation.separator() + " ";
16                 }
17             }
18             return out;
19         } catch (Exception e) {
20             return null;
21         }
22     }
23 }
```

Use toString()

```
1 public class UsePrintObjectsUtilities {
2
3     public static void main(String[] args) {
4         final Person p = new Person("Marco", 100);
5         System.out.println(PrintObjectsUtilities.toString(p));
6         // getName->Marco, getId->100,
7
8         final Product pr = new Product("Pr", 200, 100000);
9         System.out.println(PrintObjectsUtilities.toString(pr));
10        // qty->100000.0; getName->Pr; getId:200;
11    }
12 }
```

Outline

- 1 Classi, caricamento e JVM
- 2 Reflection API
- 3 Un esempio di applicazione
- 4 Annotazioni
- 5 JUnit**

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

```
1 import static org.junit.Assert.*;
2
3 import org.junit.Test;
4
5 public class CounterTest {
6
7     @Test
8     public void test1() {
9         Counter c = new Counter();
10        c.increment();
11        c.increment();
12        assertTrue("Increment does not work wrt getValue", c.getValue()==2);
13    }
14
15    @Test
16    public void test2() {
17        Counter c = new Counter();
18        assertTrue("getValue is not initially zero", c.getValue()==0);
19    }
20
21
22 }
```

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 }
```


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: 1/2

```
1 package it.unibo.apice.oop.p14reflection.annotations;
2
3 import static org.junit.Assert.*;
4 import java.util.*;
5
6 import org.junit.Test;
7
8 public class TestRangeIterator {
9
10     @Test
11     public void testStandardBehaviour() {
12         final Iterator<Integer> it = new RangeIterator(5,7);
13         assertTrue(it.hasNext());
14         assertEquals(5,it.next().intValue());
15         assertTrue(it.hasNext());
16         assertEquals(6,it.next().intValue());
17         assertTrue(it.hasNext());
18         assertEquals(7,it.next().intValue());
19         assertFalse(it.hasNext());
20     }
21
22     @Test
23     public void testStartEqualsStop() {
24         Iterator<Integer> it = new RangeIterator(5,5);
25         assertTrue(it.hasNext());
26         assertEquals(5,it.next().intValue());
27         assertFalse(it.hasNext());
28     }
```

Possibile codice JUnit: 2/2

```
1  @Test
2
3  public void testRemoveRaisesException() {
4      Iterator<Integer> it = new RangeIterator(5,10);
5      try {
6          it.remove();
7          fail();
8      } catch (UnsupportedOperationException e) {
9      }
10 }
11
12 // Variante migliorativa
13 @Test(expected = UnsupportedOperationException.class)
14 public void testRemoveRaisesException2() {
15     Iterator<Integer> it = new RangeIterator(5,10);
16     it.remove();
17 }
18
19 @Test(expected = NoSuchElementException.class)
20 public void testNoSuchElement() {
21     Iterator<Integer> it = new RangeIterator(5,6);
22     it.next();
23     it.next();
24     it.next();
25 }
26
27 @Test(expected = IllegalArgumentException.class)
28 public void testIllegalArguments() {
29     new RangeIterator(6,4);
30 }
31 }
```

Altro esempio: esercizio Graph

```
1 import java.util.*;
2
3 public interface Graph<N> {
4
5     // Adds a node: nothing happens if node is null or already there
6     void addNode(N node);
7
8     // Adds an edge: nothing happens if source or target are null
9     void addEdge(N source, N target);
10
11     // Returns all the nodes
12     Set<N> nodeSet();
13
14     // Returns all the nodes directly targeted from node
15     Set<N> linkedNodes(N node);
16
17     // Gets one sequence of nodes connecting source to path
18     List<N> getPath(N source, N target);
19
20 }
```

Un esempio di semplicissimo test

```
1 public class TestGraphImpl {
2
3     @Test
4     public void testNodeSet() {
5         final Graph<String> g = null; //new GraphImpl<>();
6         assertEquals(g.nodeSet().size(),0);
7
8         g.addNode("a");
9         g.addNode("b");
10        g.addNode("c");
11        g.addNode("d");
12        g.addNode("e");
13        assertEquals(g.nodeSet().size(),5);
14        assertTrue(g.nodeSet(),hasItems("a","b","c","d","e"));
15        assertEquals(g.linkedNodes("c").size(),0);
16
17        g.addEdge("a","b");
18        g.addEdge("b","c");
19        g.addEdge("c","d");
20        g.addEdge("d","e");
21        g.addEdge("c","a");
22        assertEquals(g.linkedNodes("c").size(),2);
23        assertTrue(g.linkedNodes("c"),hasItems("a","d"));
24
25        assertEquals(g.getPath("b", "a"),Arrays.asList("b","c","a"));
26    }
27
28 }
```

Variante migliorativa (1/2)

```
1 public class TestGraphImplImproved {
2
3     private Graph<String> g;
4
5     @Before    // initialisation code executed prior to each test
6     public void initialise() {
7         //g = new GraphImpl<>();
8         g.addNode("a");
9         g.addNode("b");
10        g.addNode("c");
11        g.addNode("d");
12        g.addNode("e");
13        g.addEdge("a", "b");
14        g.addEdge("b", "c");
15        g.addEdge("c", "d");
16        g.addEdge("d", "e");
17        g.addEdge("c", "a");
18    }
19
20    @Test
21    public void testNodeSetSize() {
22        assertEquals(5, g.nodeSet().size());
23    }
24
25    @Test
26    public void testNodeSetElements() {
27        assertThat(g.nodeSet(), hasItems("a", "b", "c", "d", "e"));
28    }
```

Variante migliorativa (2/2)

```
1  @Test
2
3  public void testLinkedNodes() {
4      assertEquals(2, g.linkedNodes("c").size());
5      assertThat(g.linkedNodes("c"), hasItems("a", "d"));
6  }
7
8  @Test
9  public void testGetPath() {
10     assertEquals(Arrays.asList("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