

Linguaggi

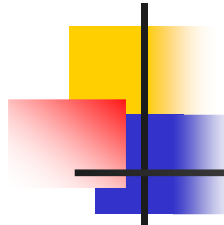
*Corso M-Z - Laurea in Ingegneria Informatica
A.A. 2009-2010*

Alessandro Longheu

<http://www.diit.unict.it/users/alongheu>

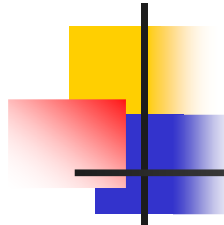
alessandro.longheu@diit.unict.it

Reflection in Java



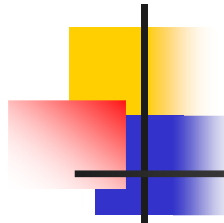
Reflection

- Riflessione
 - funzionalità per cui è possibile scrivere codice di un linguaggio la cui funzione è analizzare codice dello stesso linguaggio
 - “il codice riflette su sè stesso”
- Ad esempio, nel caso di Junit, il testRunner “carica” classi Java (ovvero bytecode) e lo analizza per capire quali i metodi di test da eseguire



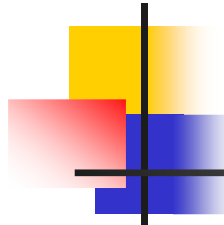
Reflection

- Perché esiste la riflessione in Java ?
 - perchè la filosofia della piattaforma è che tutti gli strumenti collegati al codice sono scritti essi stessi in Java
 - ovvero sono componenti Java che hanno bisogno di manipolare altri componenti Java
 - es: compilatore, caricatore ("classloader")
 - analogamente per gli IDE



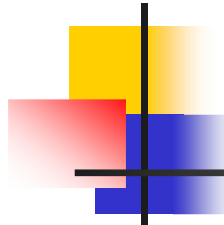
Reflection

- Il supporto alla riflessione consente a un programma di ispezionarsi ed operare su se stesso.
- Tale supporto comprende:
 - la classe `Class` nel package `java.lang`;
 - l'intero package `java.lang.reflect` che introduce le classi `Method`, `Constructor` e `Field`.
- La riflessione si usa :
 - per ottenere informazioni su una classe e sui suoi membri
 - per ottenere informazioni sulle proprietà e sugli eventi di un Java Bean
 - per manipolare oggetti.
- In particolare:
 - la classe `Field` permette di scoprire e impostare valori di singoli campi
 - la classe `Method` consente di invocare metodi
 - la classe `Constructor` permette di creare nuovi oggetti.
 - Altre classi accessorie sono `Modifier`, `Array` e `ReflectPermission`.
- NB: i nomi degli argomenti dei metodi non sono memorizzati nella classe, e quindi non sono recuperabili via riflessione.



Reflection

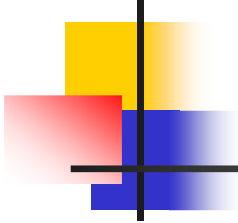
- L'idea alla base della riflessione
 - esiste una classe `java.lang.Class`
 - ad ogni classe Java corrisponde un oggetto della classe `java.lang.Class`
 - attraverso i metodi della classe è possibile analizzare tutte le caratteristiche della classe
- In effetti `java.lang.Class` si può definire una "metaclass", ovvero una classe le cui istanze (oggetti) rappresentano altre classi
 - Per ogni classe del linguaggio esiste un oggetto di tipo `java.lang.Class` che descrive il contenuto del file `.class` contenente il codice oggetto della classe



Reflection

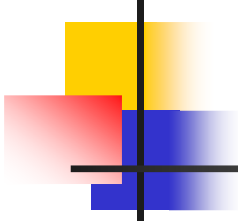
- Il riferimento all'oggetto di tipo Class
 - è accessibile in vari modi
- Proprietà statica class
 - tutte le classi hanno una proprietà pubblica e statica chiamata class che mantiene un riferimento all'oggetto di tipo `java.lang.Class`

es: `java.lang.Class classe =Counter.class;`



La Classe `java.lang.Class`

- Inizializzazione della proprietà class
 - viene effettuata automaticamente dalla macchina virtuale al caricamento della classe
- In particolare
 - viene caricato il bytecode di `java.lang.Class`
 - viene creato un oggetto di tipo `Class`
 - viene caricato il bytecode della classe
 - viene inizializzata la proprietà class



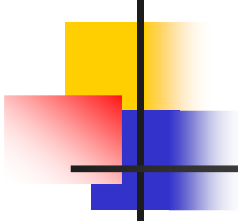
La Classe `java.lang.Class`

Il metodo `getClass()` di `Object`

- alternativa alla proprietà statica `class`
- ereditato da tutti gli oggetti
- consente di ottenere il riferimento all'oggetto di tipo `java.lang.Class` a partire da un oggetto invece che dalla classe

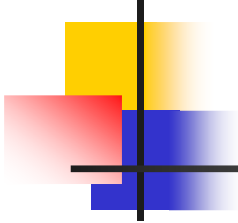
es: `Integer integer = new Integer();`

`java.lang.Class classe = integer.getClass();`



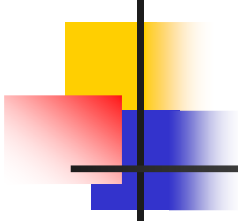
La Classe `java.lang.Class`

- Un esempio di metodi di `java.lang.Class`
 - `String getName()`: restituisce il nome della classe
- Due metodi interessanti
 - `static java.lang.Class.forName(String nome)`: cerca ed eventualmente carica l'oggetto `Class` di una classe dato il nome sotto forma di stringa
- `Class.forName("it.unict.utilita.Logger");`
 - `Object newInstance()`: crea un nuovo oggetto della classe e ne restituisce l'identificatore



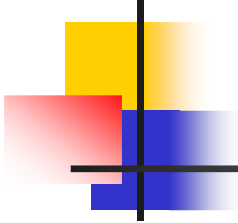
La Classe `java.lang.Class`

- Creare gli oggetti: due modi diversi
 - Modo tradizionale chiamare il costruttore richiede di conoscere staticamente (a tempo di compilazione) il nome della classe
 - Modo basato sulla riflessione utilizzando `forName` e `newInstance` sono in grado di creare oggetti di classi arbitrarie



Il Package `java.lang.Reflect`

- In realtà il sistema di classi che è possibile usare per la riflessione è molto più ampio. Il package `java.lang.reflect` fornisce tutte le classi necessarie allo studio di una classe Java attraverso la riflessione
- Principali classi
 - `java.lang.reflect.Field` per le proprietà
 - `java.lang.reflect.Method` per i metodi
 - `java.lang.reflect.Constructor` per i costruttori
- Inoltre
 - `java.lang.reflect.Modifier`
 - `java.lang.reflect.Array`



Il Package `java.lang.Reflect`

- Attraverso l'oggetto class
 - è possibile ottenere per una classe i riferimenti agli oggetti di tipo Field, Method, Constructor ecc.
 - analizzarne le caratteristiche (anche se sono privati !)
 - e utilizzarli dinamicamente (cambiare una proprietà, eseguire un metodo ecc.)
- Un esempio
 - Una classe Ispezionatore òriceve in ingresso il nome di una classe raggiungibile attraverso il classpath e ne studia il contenuto utilizzando la riflessione
- NOTA: lavora sul bytecode, non richiede la presenza del sorgente

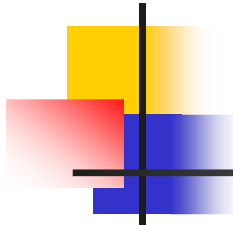


ESEMPIO

- Ipotesi: MostraClasse si invoca dalla linea di comando con un argomento che rappresenta il nome della classe da ispezionare.

```
import java.lang.reflect.*;  
public class MostraClasse {  
    public static void main(String args[]) throws ClassNotFoundException {  
        Class c = Class.forName(args[0]);  
        // recupera un riferimento a quella classe  
        if (c.isInterface()) // è un'interfaccia  
            System.out.print(Modifier.toString(c.getModifiers())+c.getName());  
        else // è una classe  
            System.out.print(Modifier.toString(c.getModifiers())+" class "  
                +c.getName()+" extends " + c.getSuperclass().getName());  
    }  
}
```

...



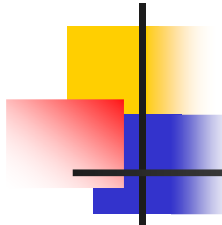
ESEMPIO

```
Class[] interfacce = c.getInterfaces();  
if ((interfacce!=null)&&(interfacce.length>0)){  
if (c.isInterface()) System.out.println(" extends ");  
else System.out.println(" implements ");  
}  
for(int i=0; i<interfacce.length; i++){  
if (i>0) System.out.print(", ");  
System.out.print(interfacce[i].getName());  
}
```



ESEMPIO

```
// elenco dei metodi della classe
System.out.println(" { "); System.out.println(" // Costruttori");
Constructor[] costruttori = c.getDeclaredConstructors();
    for(int i=0; i<costruttori.length; i++) visualizza(costruttori[i]);
System.out.println(" // Campi dati");
Field[] campiDati = c.getDeclaredFields();
    for(int i=0; i<campiDati.length; i++)
        System.out.println("
        Modifier.toString(campiDati[i].getModifiers())
        +nomeTipo(campiDati[i].getType()) +
        " " + campiDati[i].getName() + ";" );
System.out.println(" // Metodi");
Method[] metodi = c.getDeclaredMethods();
    for(int i=0; i<metodi.length; i++) visualizza(metodi[i]);
System.out.println(" } ");}
```



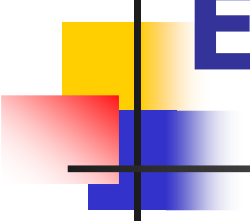
ESEMPIO

- Invocazione:
 - `java MostraClasse classeDaIspezionare`

- Ad esempio:
 - `java MostraClasse MostraClasse`

- Output:

```
public class MostraClasse extends java.lang.Object {  
    // Costruttori  
    public MostraClasse();  
    // Campi dati  
    // Metodi  
    public static void main(java.lang.String[]) throws  
        java.lang.ClassNotFoundException;  
    public static java.lang.String nomeTipo(java.lang.Class);  
    public static void visualizza(java.lang.reflect.Member);
```



ESEMPIO

```
// ----- metodi accessori -----  
public static String nomeTipo(Class t){  
    String quadre="";  
    while (t.isArray()) {  
        quadre = quadre + "[";]  
        t = t.getComponentType();  
    }  
    return t.getName() + quadre; }
```



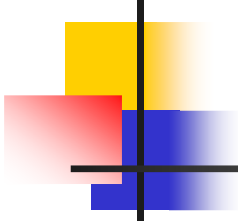
ESEMPIO

```
// ----- metodi accessori -----
public static void visualizza(Member m){
// stampa la dichiarazione di un metodo o di un costruttore
Class tipoRit = null;      Class[] parametri;      Class[] eccezioni;
    if (m instanceof Method) { // è un metodo
        Method metodo = (Method) m;
        tipoRit = metodo.getReturnType();
        parametri = metodo.getParameterTypes();
        eccezioni = metodo.getExceptionTypes();
    } else { // è un costruttore
        Constructor costr = (Constructor) m;
        parametri = costr.getParameterTypes();
        eccezioni = costr.getExceptionTypes();
    }
    System.out.print(" " + Modifier.toString(m.getModifiers()) + " " +
(tipoRit!=null ? nomeTipo(tipoRit) + " ": "") + m.getName() + "(" ); 18
```



ESEMPIO

```
for(int i=0; i<parametri.length; i++){  
    if (i>0) System.out.print(", ");  
    System.out.print(nomeTipo(parametri[i]));  
}  
System.out.print(")");  
if (eccezioni.length>0)  
    System.out.print(" throws ");  
for(int i=0; i<eccezioni.length; i++){  
    if (i>0) System.out.print(", ");  
    System.out.print(nomeTipo(eccezioni[i]));  
}  
System.out.println(";"); }}
```



Conclusione

- la riflessione permette di invocare indirettamente un metodo passato per nome tramite reflection (tecnica comunque non efficiente); questo permette di creare un codice che chiama un metodo senza saperne il nome (almeno a compile time), quindi si realizza codice "generico", similmente ai puntatori a funzione (linguaggio C) o delegati (linguaggio C#)
- La riflessione permette di fare lavorare insieme diversi software di cui uno non è ancora noto all'altro o addirittura ancora da sviluppare
- La riflessione permette di effettuare la decompilazione del software (reverse engineering)