Linguaggi

Corso M-Z - Laurea in Ingegneria Informatica
A.A. 2008-2009

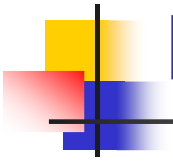
Alessandro Longheu
http://www.diit.unict.it/users/alongheu
alessandro.longheu@diit.unict.it

- lezione 24 -

Reflection

Reflection

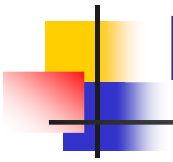
- JUnit, un aspetto misterioso
- come fa il testRunner a sapere quali sono i metodi di test da eseguire ?
- che cosa vuol dire TestXXX.class nel metodotestSuite() ?
- In effetti si tratta di tecniche di programmazione avanzate, basate sulla "riflessione"



Reflection

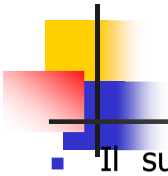
- Riflessione
  - funzionalità per cui è possibile scrivere codice di un linguaggio la cui funzione è analizzare codice dello stesso linguaggio
  - "il codice riflette su sè stesso"
- Ad esempio, nel caso di Junit, il testRunner "carica" classi Java (ovvero bytecode) e lo analizza per capire quali i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

A. Longheu – Linguaggi M-Z – Ing. Inf. 2008-2009



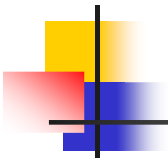
Reflection

- Perchè 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



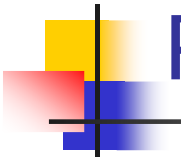
Reflection

- 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 supportati da 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

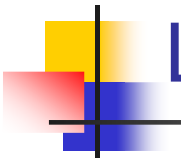
- L'idea alla base della riflessione
  - esiste una classe java.lang.Class
  - ad ogni classe Java corrisponde un oggetto della classe java.lang.Class
  - attraverso i metodi della classe è possibile analizzare tutte le caratteristiche della classe
- In effetti java.lang.Class si può definire una "metaclasse", 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

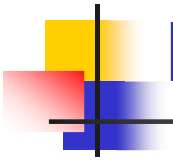
- Il riferimento all'oggetto di tipo Class
  - è accessibile in vari modi
- Proprietà statica class
  - tutte le classi hanno una proprietà pubblica e statica chiamata class che mantiene un riferimento all'oggetto di tipo java.lang.Class

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



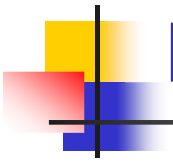
La Classe java.lang.Class

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 oggettoinvece che dalla classse

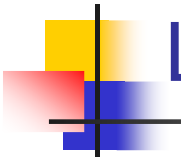
es: Integer integer = new Integer();

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



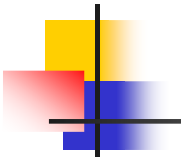
La Classe java.lang.Class

- 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 ðriceve 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());
```

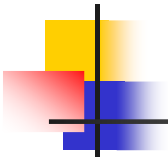
ESEMPIO

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

```
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.print(" " + 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(" } ");
}
```

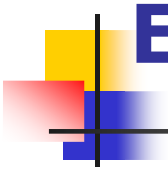


ESEMPIO

- 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);
```

A. Longheu – Linguaggi M-Z – Ing. Inf. 2008-2009



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() + "(" );
```

A. Longheu – Linguaggi M-Z – Ing. Inf. 2008-2009

ESEMPIO

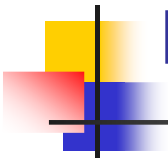
```
for(int i=0; i<parametri.length; i++){
        if (i>0) System.out.print(", ");
        System.out.print(nomeTipo(parametri[i]));
        }
System.out.print(")");
if (eccezioni.length>0)
        System.out.print(" throws ");
for(int i=0; i<eccezioni.length; i++){
        if (i>0) System.out.print(",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

Invocazione Indiretta Di Metodi

- Sebbene Java non consenta di passare metodi come dati (non esiste l'analogo dei "puntatori a funzione" del C), la riflessione permette di invocare indirettamente un metodo passato *per nome*.
- Attenzione: questa tecnica *non è molto efficiente*, quindi va usatasolo se l'efficienza non è un problema (es. gestione di eventi).
- Lo schema di invocazione indiretta è

res = m.invoke(oggettoTarget, args)

- dove m è un'istanza di Method, args è un array di Object che rappresentano gli argomenti da passare al metodo, e res è un Object che rappresenta il risultato del metodo.
- NB: invoke() è in grado di convertire automaticamente i tipi primitivi nei corrispondenti tipi "wrapper" e viceversa, così da poter chiamare anche metodi con parametri int, float, etc.



Problema:

- come ottenere l'oggetto m, di classe Method, che rappresenta il metodo da chiamare?
- Solitamente, l'oggetto metodo viene recuperato tramite una chiamata a getMethod():

m = c.getMethod(nomeMetodo, parametri);

- dove c è un'istanza di Class che rappresenta la classe dell'oggetto target su cui il metodo verrà chiamato, nomeMetodo è una String che contiene il nome del metodo, e parametri è un array di Class che rappresenta la lista dei parametri formali delmetodo.

Ad esempio:

- Class parametri[] = {int.class, int.class};
- m = c.getMethod("setLocation", parametri);

Invocazione indiretta di metodi

```
Object res=null;      // recupero della classe dell'oggetto target
Class c = ogg.getClass();   // preparazione array dei parametri formali
Class[] parametriFormali;
...
// recupero del metodo
Method m = null;
try {m = c.getMethod(nomeMetodo, parametriFormali);}
catch (NoSuchMethodException e){}
// predisposizione array degli parametri attuali
Object[] parametriAttuali;
...
// invocazione del metodo
try {res = m.invoke(ogg, parametriAttuali);}
catch(IllegalAccessException e){}
catch(InvocationTargetException e){}
```

nomeMetodo è il nome del metodo da invocare

parametriFormali è l'array di Class che contiene i paramatri *formali*

parametriAttuali è l'array di **Object** che contiene i paramatri ***attuali***

ogg è l'oggetto (target) su cui va invocato il metodo

ESEMPIO 1 - Invocazione di metodi senza argomenti

```
import java.lang.reflect.*;
import java.awt.Point;
public class ProvaInvoke1 {
    public static Object chiama0(Object ogg,String nomeMetodo) {
    // senza parametri
    Object res = null;
    Class c = ogg.getClass();
    Class parametriFormali[] = null; // senza param
    System.out.println("Calling "+nomeMetodo+"()");
    Method m = null;
    try {m = c.getMethod(nomeMetodo, parametriFormali);}
    catch (NoSuchMethodException e){
    System.err.println("Metodo inesistente"); }
    Object[] parametriAttuali = null;
    try {res = m.invoke(ogg, parametriAttuali);}
```

ESEMPIO 1 - Invocazione di metodi senza argomenti

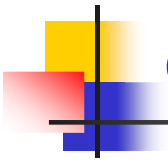
```
catch(InvocationTargetException e){
        System.err.println("Target illegale");
    }
catch(IllegalAccessException e){
        System.err.println("Accesso illegale");
    }
return res;
}

public static void main(String args[]) {
    Point p = new Point(10,20);
    Double x = (Double) chiama0(p,"getX");
    Double y = (Double) chiama0(p,"getY");
    System.out.println(" X = " + x + ", Y = " +y);
}}
```


IL PROBLEMA CON invoke() e getmethod()

- Mentre invoke() è in grado di convertire automaticamente i tipi primitivi (int, float,…) nei corrispondenti tipi "wrapper" (Integer, Float,…) e viceversa, getMethod() non lo fa: i parametri formali devono corrispondere esattamente al metodo cercato, altrimenti esso non verrà trovato.
- Questo ha come conseguenza una difficoltà nell'invocare metodi con parametri di tipi primitivi, in quanto: · invoke() pretende che i parametri attuali siano Object (l'array che li contiene è un array di Object), e i tipi primitivi non lo sono ® occorre passare i corrispondenti tipi "wrapper" (che invoke() convertirà, peraltro, automaticamente)
- getMethod() pretende invece che i parametri formali siano perfettamente corrispondenti a quelli dichiarati nel metodI
  - tipi primitivi e tipi wrapper sono diversi!
- Ciò rende impossibile scrivere una funzione call() generale, in grado di chiamare qualunque metodo.

IL PROBLEMA CON invoke() e getmethod()

- È però possibile:
  - invocare qualunque metodo (anche contenente tipi primitivi) chiamando getMethod() "ad hoc"
    - ESEMPIO 2
  - scrivere una funzione call() "quasi generale", atta a invocare qualunque metodo che non usi tipi primitivi
    - ESEMPIO 3
- Fortunatamente, per quasi tutti i metodi Java che usano tipi primitivi esiste anche una versione che li incapsula in un qualche oggetto. Ad esempio, la classe Point definisce
  - sia il metodo void setLocation(int, int)
  - sia il metodo void setLocation(Point)



ESEMPIO 2 - Invocazione di metodo con due argomenti int

```
import java.lang.reflect.*;
mport java.lang.reflect.*;
import java.awt.Point;
public class ProvaInvoke2 {
    public static Object chiama1(Object ogg,
            Method m, Object args[]) {
            Object res = null;
            try { res = m.invoke(ogg, args); }
            catch(InvocationTargetException e){
                    System.err.println("Target illegale");
            }
            catch(IllegalAccessException e){
                    System.err.println("Accesso illegale");
            }
            return res;
    }
```

ESEMPIO 2 - Invocazione di metodo con due argomenti int

```
public static void main(String args[]) {
        Point p = new Point(10,20);
        Method getX = null, getY = null, sL = null;
        try { // due modi alternativi
           getX = p.getClass().getMethod("getX",null);
           getY = Point.class.getMethod("getY",null);
        } catch(NoSuchMethodException e) {}
           System.out.println(" X = " + x + ", Y = " +y);
        Class intintArgs[] = {int.class,int.class};
        try {
         sL=Point.getMethod("setLocation",intintArgs);
        } catch(NoSuchMethodException e) {}
        Integer argomenti[] =
                  { new Integer(30), new Integer(40) };
        chiama1(p,setLoc, argomenti);
        Double x = (Double) chiama1(p, getX, null);
        Double y = (Double) chiama1(p, getY, null);
        System.out.println(" X = " + x + ", Y = " +y);    }}
```


ESEMPIO 3 - Invocazione di metodo con un argomento Point

```
import java.lang.reflect.*;
import java.awt.Point;
public class ProvaInvoke3 {
        public static Object chiama(Object ogg,
        String metodo, Object args[]) {
        Object res = null;
        Class c = ogg.getClass();
        Class parametriFormali[] = null;
        int argNum = 0;
        if (args==null) {
                System.out.println("Calling "+ metodo +"()");
        } else {
                System.out.print("Calling " + metodo + "(");
                argNum = args.length;
        parametriFormali = new Class[argNum];
        for (int i=0; i<argNum-1; i++) {
                parametriFormali[i] = args[i].getClass();
                System.out.print(args[i].getClass().getName()    + ",");  }
```

ESEMPIO 3 - Invocazione di metodo con un argomento Point

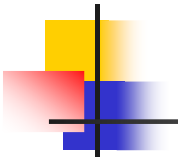
```
parametriFormali[argNum-1] = args[argNum-1].getClass();
    System.out.println(args[argNum-1].getClass().getName()
    +")");}
Method m = null;
try { // trova solo metodi senza tipi primitivi
    m = c.getMethod(metodo, parametriFormali);}
catch (NoSuchMethodException e){
    System.err.println("Metodo inesistente");}
try {res = m.invoke(ogg, args);}
catch(InvocationTargetException e){
        System.err.println("Target illegale");}
catch(IllegalAccessException e){
        System.err.println("Accesso illegale");}
return res;
}
```

A. Longheu – Linguaggi M-Z – Ing. Inf. 2008-2009

ESEMPIO 3 - Invocazione di metodo con un argomento Point

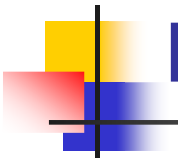
```
public static void main(String args[]) {
  Point p = new Point(10,20);
  Double x = (Double) chiama(p,"getX", null);
  Double y = (Double) chiama(p,"getY", null);
  System.out.println(" X = " + x + ", Y = " +y);
  Point argomenti[] = { new Point(30,40) };
       chiama(p,"setLocation", argomenti);
  // uso setLocation(Point), perché l'altro
  // setLocation(int,int) non verrebbe trovato
  chiama(p,"setLocation", argomenti);
  x = (Double) chiama(p,"getX", null);
  y = (Double) chiama(p,"getY", null);
  System.out.println(" X = " + x + ", Y = " +y);}}
```

Decompilatori e Offuscatori



- Riassumendo
  - a partire dal bytecode è possibile risalire a moltissime delle caratteristiche del codice sorgente originale
  - anche senza avere a disposizione il .java
- In effetti
  - nei linguaggi basati sulla riflessione, questo processo può essere spinto molto avanti

Decompilatore



- Decompilatore
  - applicazione che ispeziona il codice oggetto di una classe
  - utilizza la riflessione per estrarne le caratteristiche e disassembla il codice oggetto per ricostruire la struttura dei metodi
  - in altri termini, ricostruisce integralmente il codice sorgente originale dal codice oggetto
- Offuscatore
  - applicazione che riorganizza il codice oggetto di una classe per renderlo più difficilmente manipolabile dal decompilatore
  - in sostanza tende a confondere nomi e identificatori

  Un esempio: ProGuard, progetto open source