

# 18.

## La riflessione

Marco Faella

Dip. Ing. Elettrica e Tecnologie dell'Informazione  
Università di Napoli “Federico II”

Corso di Linguaggi di Programmazione II

- La riflessione (o *introspezione*) è quella caratteristica di un linguaggio che permette ai programmi di investigare a tempo di esecuzione sui tipi effettivi degli oggetti manipolati
- Java fornisce ampio supporto alla riflessione
- Il cardine della riflessione è rappresentato dalla **classe Class**
- Ciascun oggetto di classe Class rappresenta una delle classi del programma
  - la JVM si occupa di istanziare un oggetto Class per ogni nuova classe caricata in memoria
  - solo la JVM può istanziare la classe Class
  - un oggetto di tipo Class contiene tutte le informazioni relative alla classe che esso rappresenta: i costruttori, i metodi, i campi, ed anche le eventuali classi interne
  - tramite questo oggetto, è possibile conoscere a run-time le caratteristiche di una classe che non è nota al momento della compilazione
- Pur non essendo classi, anche i tipi primitivi hanno un corrispondente oggetto di tipo Class

# Ottenere riferimenti agli oggetti di tipo Class

Per ottenere un riferimento ad un oggetto di tipo Class, è possibile utilizzare tre tecniche:

- 1) Il metodo **getClass** della classe Object
- 2) L'operatore **.class**
- 3) Il metodo statico **forName** della classe Class

- Nella classe Object, è presente il metodo

```
public Class<?> getClass()
```

- Questo metodo restituisce l'oggetto Class corrispondente al tipo **effettivo** di questo oggetto
- Il **tipo restituito** di getClass merita attenzione (tra qualche slide...)

- La classe Class ha un parametro di tipo, che chiameremo "T"
- Come un serpente che si morde la coda, il parametro di tipo di un oggetto Class indica il tipo che questo oggetto rappresenta
- Ad esempio, se x è l'oggetto Class relativo alla classe Employee, il tipo di x è `Class<Employee>`
- Nelle prossime slide, vedremo come il parametro di tipo viene utilizzato dai metodi di Class

# Alcuni metodi della classe Class

- Esaminiamo alcuni metodi della classe `Class<T>`:

`public String getName()`

- Restituisce il nome di questa classe, completo di eventuali nomi di pacchetti (*fully qualified*)

`public T newInstance()`

- Crea e restituisce un nuovo oggetto di questa classe, invocando un costruttore senza argomenti, che questa classe deve possedere

`public static Class<?> forName(String name)`

- Restituisce l'oggetto `Class` corrispondente alla classe di nome "name"; la stringa "name" deve contenere anche l'indicazione degli eventuali pacchetti cui la classe appartiene

`public Class<? super T> getSuperclass()`

- Restituisce l'oggetto `Class` corrispondente alla superclasse diretta di questa; se questa è `Object`, oppure è un'interfaccia o un tipo base, il metodo restituisce `null`

`public boolean isInstance(Object x)`

- Restituisce vero se (e solo se) il tipo effettivo di `x` è sottotipo di questa classe
- E' la versione riflessiva di `instanceof` (dinamica anche nell'operando destro)

- Supponiamo di applicare getClass ad un oggetto di tipo dichiarato Employee:

```
Employee e = ... ;  
??? = e.getClass();
```

- A che tipo di variabile ci aspettiamo di poter assegnare il risultato della chiamata?
- A prima vista, la risposta sembrerebbe Class<Employee>
- A pensarci meglio, se il tipo effettivo di "e" fosse Manager, l'oggetto restituito da getClass sarebbe di tipo Class<Manager>, che non è assegnabile a Class<Employee>!
- Pertanto, il tipo più appropriato (cioè, più specifico) a cui vorremmo assegnare il risultato è:

Class<? extends Employee>

- Ma Class<?> (tipo di ritorno di getClass) **non** è assegnabile a Class<? extends Employee>

**Come fare?**

- Il tipo restituito di getClass dovrebbe esprimere il seguente concetto:

*Applicato ad un'espressione di tipo dichiarato A, questo metodo restituisce un oggetto di tipo `Class<? extends A>`*

- Purtroppo, **non è possibile** esprimere in Java questa proprietà
- Quindi, il type-checker tratta il metodo getClass **in modo particolare**, simulando il tipo restituito che il linguaggio non è in grado di esprimere
- Nota: Fino a Java 5 (1.5), il tipo di ritorno di getClass era dichiarato "`Class<? extends Object>`"



- Precisamente, il tipo di ritorno di `exp.getClass()` è l'*erasure* del tipo dichiarato dell'espressione `exp`
- Quindi, l'esempio seguente non compila:

```
public static <T> Class<? extends T> myGetClass(T x) {  
    return x.getClass(); // errore qui  
}
```

Infatti:

- Il tipo dichiarato di `x` è `T`
- L'*erasure* di `T` è `Object`
- Il tipo di ritorno di `x.getClass()` è `Class<? extends Object>`
- `Class<? extends Object>` non è assegnabile a `Class<? extends T>`

- La seconda tecnica per ottenere un riferimento ad un oggetto di tipo Class sfrutta l'operatore **".class"**
- Tale operatore si applica al **nome di una classe** o di un tipo primitivo, come in:

```
Employee.class  
String.class  
java.util.LinkedList.class  
int.class
```

- Pertanto, è un *operatore unario postfisso*

- L'operatore .class ha carattere *statico*, nel senso che il suo valore è **noto al momento della compilazione**
- Quindi, applicato ad una classe A, forma un'espressione di tipo `Class<A>`
- Ad esempio:

```
Class<String> c1 = String.class;  
Class<Integer> c2 = Integer.class;  
Class<Integer> c3 = int.class;
```

Nota: nell'esempio qui sopra c2 e c3 sono due oggetti differenti

- La terza tecnica è rappresentata dal metodo statico **`forName`** della classe `Class`, che abbiamo già presentato in questa lezione
- Questa tecnica ha carattere *dinamico*, in quanto il valore restituito da `forName` non è noto al momento della compilazione

Data la dichiarazione:

```
Employee e = new Manager(...);
```

Per ognuna delle seguenti espressioni, dire se è corretta o meno, e in caso affermativo calcolarne il valore

- 1) `e instanceof Employee`
- 2) `e instanceof Manager`
- 3) `e.class instanceof Employee`
- 4) `e.getClass() == Manager`
- 5) `e.getClass() == Employee.class`
- 6) `e.getClass() == "Manager" (*)`
- 7) `e.getClass() == Manager.class`
- 8) `e.getClass().equals(Manager.class)`

(\*) La risposta dipende dalle regole di tipo dell'operatore "=="

Presentiamo un metodo statico "fill", che accetta un array e un oggetto Class e riempie l'array di nuove istanze della classe corrispondente

Ecco una possibile invocazione:

```
Employee[] a = new Employee[10];  
Test.<Employee>fill(a, Manager.class);
```

Questo frammento dovrebbe riempire l'array con 10 nuovi oggetti Manager, creati con il costruttore senza argomenti (se esiste)

Implementazione, all'interno di una classe Test:

```
public static <T> void fill(T[] arr, Class<? extends T> c)
    throws InstantiationException, IllegalAccessException
{
    for (int i=0; i<arr.length; i++) {
        T x = c.newInstance();
        arr[i] = x;
    }
}
```

Il metodo `newInstance` può lanciare diverse eccezioni verificate, ad esempio nel caso in cui la classe in questione non possenga un costruttore senza argomenti, o se tale costruttore non sia accessibile.

Si noti il tipo di `Class`, che consente di istanziare oggetti di una sottoclasse del tipo dell'array.

Nota: purtroppo, anche la seguente invocazione è lecita: `<Object>fill(a, String.class)`

Consideriamo una classe per coppie di oggetti dello stesso tipo

Con generics:

```
public class Pair<S> {  
    private S first, second;  
  
    public Pair(S a, S b) {  
        first = a;  
        second = b;  
    }  
    public void setFirst(S a) { first = a; }  
    public S getFirst()      { return first; }  
  
    @Override  
    public boolean equals(Object other) {  
        if (!(other instanceof Pair))  
            return false;  
        Pair<?> p = (Pair) other;  
        return first.equals(p.first) && second.equals(p.second);  
    }  
}
```

Siamo costretti  
a usare la classe grezza



Ora proviamo a simulare i generics con la **riflessione**:

```
public class Pair {  
    private Object first, second;  
    private final Class<?> type;  
  
    public Pair(Class<?> c, Object a, Object b)  
    {  
        if (!c.isInstance(a) || !c.isInstance(b))  
            throw new IllegalArgumentException();  
        type = c;  
        first = a;  
        second = b;  
    }  
    public void setFirst(Object a) {  
        if (!type.isInstance(a))  
            throw new IllegalArgumentException();  
        first = a;  
    }  
    public Object getFirst() { return first; }  
}
```

Controlliamo a runtime  
quello che i generics controllano  
in fase di compilazione

*continua...*

Ora proviamo a simulare i generics con la riflessione:

```
public class Pair {  
    private Object first, second;  
    private final Class<?> type;  
  
    ...  
  
    @Override  
    public boolean equals(Object other) {  
        if (!(other instanceof Pair))  
            return false;  
        Pair p = (Pair) other;  
        if (p.type != type)  
            return false;  
        return first.equals(p.first) && second.equals(p.second);  
    }  
}
```

Possiamo controllare  
il tipo effettivo di un'altra coppia

Attenzione: così una coppia di Manager risulta  
diversa da una coppia di Employee, anche se  
contengono gli stessi oggetti (due Manager)

Possiamo anche **combinare i due approcci**

Generics + riflessione:

```
public class Pair<S> {  
    private S first, second;  
    private final Class<S> type;  
  
    public Pair(Class<S> c, S a, S b)  
    {  
        type = c;  
        first = a;  
        second = b;  
    }  
    public void setFirst(S a) { first = a; }  
    public S getFirst() { return first; }  
    ...  
}
```

Usare questa classe:

```
Pair<String> p = new Pair<>(String.class, "uno", "due");      OK
```

```
Pair<String> p = new Pair<>("pippo".getClass(), "uno", "due");  ??
```

Esplorare il contenuto di una classe

# Ottenere informazioni su una classe

- I metodi della classe `Class` permettono di ricavare numerose informazioni sulla classe in questione
- In particolare, è possibile conoscere l'elenco di tutti i **campi**, **metodi** e **costruttori** appartenenti alla classe
- A tale scopo, esistono le classi **Field**, **Method** e **Constructor**, che rappresentano gli elementi omonimi di una classe

# Ottenere informazioni su una classe

- Per ottenere tali informazioni, sono utili i seguenti metodi di Class:

<code>public Field[] getFields()</code>	Tutti i campi pubblici (anche ereditati)
<code>public Field[] getDeclaredFields()</code>	Tutti i campi dichiarati qui (anche privati)

<code>public Method[] getMethods()</code>
<code>public Method[] getDeclaredMethods()</code>

<code>public Constructor[] getConstructors()</code>
<code>public Constructor[] getDeclaredConstructors()</code>

- Il metodo `getFields` restituisce tutti i campi **pubblici** di questa classe, anche ereditati
  - `getMethods` è analogo a `getFields`
  - `getConstructors` restituisce semplicemente i costruttori pubblici di questa classe
- Il metodo `getDeclaredFields` restituisce tutti i campi dichiarati in questa classe (e non nelle superclassi), indipendentemente dalla loro visibilità
  - similmente, `getDeclaredMethods` e `getDeclaredConstructors`

- La classe **Field** rappresenta un **campo** di una classe
- Essa dispone di metodi per leggere e modificare il contenuto di un campo, conoscere il suo nome e il suo tipo
- In particolare, abbiamo:

public String <b>getName()</b>	Restituisce il nome di questo campo.
public Object <b>get</b> (Object x) throws IllegalAccessException	Restituisce il valore di questo campo nell'oggetto x. Se questo campo è di un tipo base, il suo valore viene racchiuso nel corrispondente tipo riferimento (ad es., int -> Integer). Se questo campo è statico, il parametro x viene ignorato.
public void <b>set</b> (Object x, Object val) throws IllegalAccessException	Imposta a val il valore di questo campo nell'oggetto x. Se questo campo è statico, il parametro x viene ignorato.
public Class<?> <b>getType()</b>	Restituisce il tipo di questo campo.

- Alcuni metodi sollevano l'eccezione verificata IllegalAccessException, quando si tenta di accedere ad un campo che non è accessibile a causa della sua visibilità

- I metodi get e set di Field (così come altri metodi simili delle classi Method e Constructor) applicano le regole di visibilità previste dal linguaggio
- Cioè, lanciano l'eccezione verificata `IllegalAccessException` se si tenta di accedere ad un campo che non è visibile dalla classe in cui ci si trova
- E' possibile disattivare questo controllo utilizzando i metodi della classe **`AccessibleObject`**, superclasse comune a Field, Method e Constructor
- La possibilità di aggirare i controlli è soggetta al Security Manager, l'oggetto che è responsabile dei permessi per le operazioni a rischio
- Di default, la JVM si avvia senza un Security Manager, consentendo alle applicazioni di compiere qualsiasi operazione
- Invece, i browser che eseguono applet Java utilizzano dei Security Manager particolarmente restrittivi



- La classe **Method** rappresenta un metodo di una classe
- Essa dispone di metodi per conoscere il nome del metodo, il numero e tipo dei parametri formali e il tipo di ritorno
- Inoltre, è possibile invocare il metodo stesso
- In particolare, abbiamo:

public String <b>getName</b> ()	Restituisce il nome del metodo rappresentato.
public Object <b>invoke</b> (Object x, Object...args) throws IllegalAccessException	Invoca sull'oggetto x il metodo rappresentato, passandogli i parametri attuali args. Se il metodo rappresentato è statico, il parametro x viene ignorato. Restituisce il valore restituito dal metodo rappresentato.

- La sintassi "Object...args" indica che invoke accetta un numero variabile di argomenti

Metodi variadici

# Metodi con numero variabile di argomenti

- Dalla versione 1.5, Java prevede un meccanismo per dichiarare metodi con un numero variabile di argomenti (metodi *variadici*, o, in breve, *varargs*)

- Se T è un tipo di dati, con la scrittura

`f(T ... x)`

- si indica che f accetta un numero variabile di argomenti (anche zero), tutti di tipo T
- I puntini sospensivi devono essere necessariamente tre
- Gli argomenti possono essere passati separatamente, come in `f(x1, x2, x3)`, oppure tramite un array, come in `f(new T[] {x1, x2, x3})`
- All'interno del metodo f, si può accedere agli argomenti utilizzando x come un array di tipo T
- Ogni metodo può avere un solo argomento variadico, che deve essere l'**ultimo** della lista, come il metodo *invoke* della classe Method, illustrato nella slide precedente

```
public Object invoke(Object x, Object...args)
```

In altri linguaggi

- Il C non fornisce alcun supporto alla riflessione
- Il C++ fornisce un supporto parziale alla riflessione, chiamato Run-Time Type Information (RTTI)
- Diremo che un oggetto “conosce il proprio tipo” se a partire dal suo indirizzo è possibile risalire all’identità del suo tipo
- Ovvero, se nel memory layout dell’oggetto è presente un puntatore o un identificativo della sua classe
- In C, nessun oggetto (ad es., struct) conosce il proprio tipo
- In Java, tutti gli oggetti conoscono il proprio tipo
- In C++, RTTI agisce solo su classi che hanno *almeno un metodo virtuale* (cioè sovrascrivibile)
  - Gli oggetti di queste classi conoscono il proprio tipo, per permettere il binding dinamico
  - Gli oggetti delle altre classi non conoscono il proprio tipo

- **Operatore typeid:** `typeid(exp)`
  - Restituisce un oggetto di tipo `std::type_info` corrispondente al tipo effettivo di "exp"
  - Non compila se il tipo dichiarato di `exp` non conosce il proprio tipo
  - Simile a `getClass`
- **Dynamic cast:** `dynamic_cast<type>(exp)`
  - Un cast che controlla a runtime se "exp" è di tipo effettivo "type"
  - Se non lo è, restituisce `nullptr` (in caso di tipo puntatore) o lancia un'eccezione (in caso di tipo riferimento)
  - Non compila se il tipo dichiarato di `exp` non conosce il proprio tipo
  - Simile a un cast Java tra riferimenti

- 1) Implementare un metodo, chiamato *reset*, che prende come argomento un oggetto ed imposta a zero tutti i suoi campi interi pubblici
- 2) Implementare un metodo che, dato un oggetto, parte dalla classe che rappresenta il tipo effettivo dell'oggetto e ne restituisce la superclasse più generale, escludendo Object (quindi, la penultima classe, prima di arrivare a Object)