Java

riconoscere i tipi

G. Prencipe
prencipe@di.unipi.it

Introduzione

- Il run-time type identification è quella parte di Java che si occupa di controllare i tipi a run-time
  - In particolare determina il tipo esatto di un oggetto avendo solo un riferimento a un oggetto della superclasse
- È strettamente legato al polimorfismo

RTTI

- Nel tipico esempio delle forme geometriche, quando si creano forme specifiche (**Triangolo**, **Cerchio**) e poi si invoca un metodo **disegna()** definito nella superclasse, abbiamo che questo metodo funziona correttamente grazie al polimorfismo
  - Ogni oggetto della sottoclasse è anche oggetto della superclasse

RTTI

- Supponiamo di creare un array di **Object** contenenti forme geormetriche specifiche

```
Object[] shapeList = {
   new Circle(), new Square(), new Triangle()
}
```

- Dato che tutti gli oggetti sono **Object**, il codice scritto su funziona
- Quando però accediamo gli oggetti dell'array, otteniamo **Object**
  - In altre parola abbiamo perso la specializzazione delle forme geometriche

RTTI

- Per specificare che in realtà trattiamo con forme geometriche, bisogna fare un cast a **Shape** (downcast) che verrà controllato a run-time dal RTTI
- A questo punto sull'oggetto restituito (una **Shape**) possiamo invocare i metodi scritti per la superclasse (upcast)
  - Anche se gli oggetti sappiamo essere degli oggetti delle sottoclassi di **Shape**
- A volte però vorremmo sapere il tipo *esatto* di un certo riferimento

L'oggetto **Class**

- Per capire meglio come fare questo, bisogna introdurre uno speciale oggetto chiamato l'oggetto **Class**, che contiene informazioni sulla classe
- Questo oggetto è utilizzato per creare tutti gli oggetti di una certa classe
- C'è un oggetto **Class** per ogni classe del nostro programma
  - Cioè, ogni volta che si scrive e compila una nuova classe, un singolo oggetto **Class** è creato
  - Viene memorizzato in un file **.class** chiamato con lo stesso nome

L'oggetto **Class**

- A run-time, quando si vuole creare un oggetto di una certa classe, la JVM che esegue il programma controlla prima se l'oggetto **Class** per quel tipo è caricato
- Se no, la JVM lo carica cercando il file **.class** con lo stesso nome della classe
- Quindi un programma Java non è caricato completamente prima che inizi l'esecuzione

L'oggetto **Class**

- Una volta che l'oggetto Class per quel tipo è in memoria, viene utilizzato per creare tutti gli oggetti di quel tipo
- Il metodo statico **forName(String s)** della classe **Class**, passata una stringa contenente il nome *esatto* di una classe, restituisce un riferimento all'oggetto **Class** per quel tipo
- Vediamo un esempio

Esempio -- l'oggetto **Class**

```
class Candy {
  static {System.out.println("Loading Candy");}
}
class Gum {
  static {System.out.println("Loading Gum");}
}
class Cookie {
  static {System.out.println("Loading Cookie");}
}
```

- Ognuna di queste classi ha una clausola **static** che viene eseguita appena la classe è caricata in memoria la prima volta

Esempio -- l'oggetto **Class**

```
public class SweetShop {
  public static void main(String[] args) {
    System.out.println("inside main");
    new Candy();
    System.out.println("After creating Candy");
    try {
       // Restituisce un riferimento all'oggetto Class di questo tipo
       // In questo caso il riferimento è ignorato
       // Il metodo è invocato per provocare il caricamento in memoria della classe
       Class.forName("Gum"); } catch(ClassNotFoundException e) {
      System.out.println("Couldn't find Gum");
    }
    System.out.println("After Class.forName(\"Gum\")");
    new Cookie();
    System.out.println("After creating Cookie");
  }} ///:~
```

L'oggetto **Class**

- Un secondo modo per ottenere un riferimento a un oggetto Class è fornito dall'utilizzo dei *letterali di classe*
- Nell'esempio di prima, avremmo scritto

  Gum.class;
- Questi letterali funzionano con classi regolari, interfacce, arrays e tipi primitivi

L'oggetto **Class**

- Inoltre esiste un campo **TYPE** per ognuna delle classi associate ai tipi primitivi, che produce un riferimento a un oggetto **Class** per il tipo primitivo associato
  - Boolean.class equivale a Boolean.TYPE
  - Int.class equivale a Integer.TYPE

RTTI

- Fino ad ora abbiamo visto due forme per l'RTTI
  - Nel caso di cast (come in (Shape)Triangle), che utilizza RTTI per accertarsi che il **cast** sia corretto. In caso contrario viene generata una **ClassCastException**
  - L'oggetto **Class** che rappresenta il tipo degli oggetti. Può essere interrogato per ottenere informazioni utili a run-time
- Esiste una terza forma per l'RTTI in Java, ottenuta con la parola chiave **instanceof**

Parola chiave **instanceof**

- Questa parola chiave serve a sapere se un oggetto è una istanza di un particolare tipo
- Restituisce un boolean

```
if(x instanceof Cane) ((Cane)x).abbaia();
```

- Il controllo viene effettuato prima di operare un downcast

Esercizio

- Proviamo a sviluppare insieme un programma (leggermente più intricato) che mostri l'utilizzo di instanceof
- Iniziamo creando un nuovo progetto **TestInstanceOf** contenente un pacchetto chiamato **c10** in cui mettiamo per i momento le seguenti classi tutte **public**
  - Animale, Cane **extends** Animale, Lupo **extends** Cane, Gatto **extends** Animale, Roditore **extends** Animale, Castoro **extends** Roditore, Scoiattolo **extends** Roditore
  - Lasciare le classi vuote

Esercizio -- continua

- Quello che vogliamo fare è tenere traccia di quanti oggetti di un certo tipo sono stati creati
- Per fare questo, aggiungiamo in **c10** una classe public **Counter** che contiene un **int** i e un metodo **toString** che restituisce i come **String**a
  - Invocare il metodo **statico** **Integer.toString(i)**

Esercizio -- continua

- A questo punto abbiamo bisogna di una memoria associativa che contenga per ogni tipo di **Animale** disponibile il numero di oggetti creati
- Utilizziamo il codice già scritto per realizzare una memoria associativa con **Vector**
  - In questo caso però, invece di coppie (**int**,**int**) ci occorre memorizzare coppie di **Object**

Esercizio -- continua

- A questo punto siamo pronti per scrivere la classe public **ContaAnimali**
  - Creiamo un generatore di numeri casuali **rand**
  - Creiamo un array di **Object animali** contenente 15 elementi
  - Creiamo un array di **Class tipiAnimali** contenente i riferimenti ai **.class** delle classi di **Animali** create (utilizzare i letterali di classe)
  - Inserire in **Animali** 15 animali creati casualmente. Creare le istanze con il metodo **newInstance()** di **Class**
  - Creare un array associativo in cui si inseriscono coppie (*chiave*,*valore*), dove
    - Le *chiavi* sono i tipi di possibili **Animali**
    - I *valori* sono **Counter**

Esercizio -- continua

- A questo punto si scandisce l'array **animali**
  - Utilizzando **istanceof** si controlla il tipo degli elementi, si accede alla memoria associativa nella posizione relativa a quel tipo, e si incrementa il contatore associato
- Inserire infine delle stampe per visualizzare il contenuto di **animali** e dell'array associativo

instanceof dinamico

- Il metodo **Class.isInstance()** fornisce un modo per invocare dinamicamente l'operatore **instanceof**
- Le chiamate a **instanceof** dell'esercizio fatto diventano

```
for(int i = 0; i < animali.length; i++) {
    Object o = animali[i];
    for(int j = 0; j < tipiAnimali.length; ++j)
      if(tipiAnimali[j].isInstance(o))
      ((Counter)map.get(tipiAnimali[j].toString())).i++;
}
```

instanceof dinamico

- In questo modo è anche possibili aggiungere altri tipi a **tipiAnimali** senza dover modificare il codice

instanceof vs equivalenza

- C'è una differenza sostanziale tra comparare due oggetti **Class** con **instanceof** e **equals()**
  - Con **instanceof** ci si chiede se un certo oggetto è di una certa classe o di una classe da essa derivata (come possiamo anche osservare dall'esercizio sugli **Animali** con, ad esempio, **Cane** e **Lupo**
  - Con **equals** ci si chiede se due oggetti sono dello stesso tipo o no (cioè, non ci importa se sono "parenti")
- **Instanceof** e **isInstance()** producono lo stesso risultato, così come **==** e **equals()** quando chiamati su oggetti **Class**, come si nota in **FamilyVsExactType.java**

Sintassi RTTI

- Java effettua RTTI utilizzando oggetti **Class**
- Alcuni dei modi per utilizzare RTTI li abbiamo già visti
- Per ottenere un riferimento a un oggetto **Class**
  - **Class.forName()**
    - Utile quando non si ha una istanza dell'oggetto di una certa classe. È sufficiente passare come argomento una Stringa (il nome della classe)
  - Con il metodo **getClass()** in **Object** che restituisce il riferimento all'oggetto **Class** dell'oggetto su cui è invocato

Sintassi RTTI

- Altri metodi interessanti in Class sono
  - **getInterfaces()**
    - Restituisce un array di oggetti **Class** che rappresenta le interfacce che sono contenute nell'oggetto **Class**e di interesse
  - **getSuperclass()**
    - Dato un oggetto Class, restituisce un riferimento all'oggetto Class relativo alla sua superclasse
  - **newInstance()**
    - Dato un riferimento Class, crea una nuova istanza della classe riferita
    - Nota: la classe di cui viene creato l'oggetto deve avere un costruttore di default (visto che non gli si passano parametri)

Reflection

- RTTI ci permette di analizzare i tipi che creiamo
- La cosa importante per poter utilizzare RTTI è che i tipi che vogliamo analizzare a run-time siano tutti disponibili già a tempo di compilazione
  - Cioè, dobbiamo avere disponibili i **.class**

Reflection

- Ci sono caso dove però questo non è possibile
  - Ad esempio se ci arrivano dei dati da rete e ci viene detto che rappresentano classi. A compilazione non li abbiamo: sono dato che avremo solo a run-time
  - Con le *Remote Method Invocation* (RMI) possiamo invocare metodi situati su macchine remote. Quindi, in locale e a tempo di compilazione, non conosciamo i tipi degli oggetti su cui invocheremo questi metodi

Reflection

- In tutti questi casi ci vene in aiuto la *reflection*, offerta dalla libreria **java.lang.reflect**
- Ha classi **Field**, **Method** e **Constructor**
- Oggetti di questo tipo so creati dalla JVM a run-time per rappresentare il corrispondente membro nella classe sconosciuta a compilazione
  - È fondamentale però che i **.class** corrispondenti siano disponibili a run-time
- La differenza sostanziale tra RTTI e *reflection* è che i **.class** devono essere disponibili a compilazione nel primo caso, mentre nel secondo vengono aperti ed esaminati solo a run-time

Esercizi

1. Scrivere una gerarchia di forme geometriche
   1. Superclasse **FormeGeom**, con sottoclassi **Cerchio**, **Quadrato** e **Tiangolo**, con un flag **boolean acceso** (inizializzato a **false**) e metodi **toString()**. Inserire in **FormeGeom** un metodo **disegna()**
   2. Creare un array di **FormeGeom** con 5 elementi, e riempirlo con varie forme
   3. Settare **acceso** a **true** solo per un particolare tipo di forme
   4. Stampare il contenuto dell'array e dei flag

Esercizi

2. Scrivere un metodo in **ClassInfo.java** che prende come argomento un oggetto e ricorsivamente stampi tutte le classi nella gerarchia di quell'oggetto
   1. Utilizzare **getDeclaredFields()** in Class per mostrare le informazioni sui campi della classe
   2. Provare a sostituire **getDeclaredFileds()** con **getDeclaredMethods()**

Java

riconoscere i tipi

fine