8.
Confronto e ordinamento tra oggetti

Marco Faella

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

Corso di Linguaggi di Programmazione II

Confronto tra oggetti: l'interfaccia Comparable

- Esaminiamo il problema di **ordinare un insieme di oggetti** secondo un criterio di ordinamento

- Innanzitutto, vediamo qual è il modo standard di impostare un criterio di ordinamento per gli oggetti di una classe
- La libreria standard Java fornisce **due interfacce** a questo scopo

- La prima è l'interfaccia *Comparable*

```
public interface Comparable<T> {
     public int compareTo(T x);
}
```

- Quando si voglia dotare una classe di un criterio di ordinamento, si può far implementare alla classe l'interfaccia Comparable

Il contratto di Comparable

- Il **contratto** del metodo compareTo con argomento x è il seguente
  - **Pre-condizione:** l'oggetto x è confrontabile con this
  - **Post-condizione:**
    restituisce:
    - un valore negativo se this è minore di x
    - 0 se this è uguale a x
    - un valore positivo se this è maggiore di x

    Inoltre, deve rappresentare una **relazione d'ordine** tra oggetti (si veda dopo)
  - Il metodo dovrebbe lanciare l'eccezione (non verificata) *ClassCastException* se riceve un oggetto che, a causa del suo tipo effettivo, non è confrontabile con this

- L'uso dell'interfaccia Comparable è indicato quando la classe da ordinare possiede **un unico criterio** di ordinamento naturale

Il metodo strcmp del linguaggio C

- E' evidente l'analogia tra il contratto di compareTo e la funzione **strcmp** del linguaggio C
- Nella libreria standard del C (header string.h), si trova la seguente funzione

```
int strcmp(const char *s1, const char *s2)
```

- La funzione confronta alfabeticamente le stringhe puntate da s1 ed s2, restituendo un valore intero secondo le stesse regole del contratto di compareTo
  - Ovvero, un numero negativo se s1 precede s2 in ordine alfabetico, e così via

Esempio

- Consideriamo la classe Employee, con campi **nome** (String) e **salario** (int)

- Se siamo certi che gli Employee saranno sempre confrontati alfabeticamente per nome, faremo in modo che Employee implementi l'interfaccia Comparable, come segue:

```
public class Employee implements Comparable<Employee> {
   private int salary;
   private String name;

   @Override
   public int compareTo(Employee x) {
      return name.compareTo(x.name);
   }
}
```

- Come si vede, sfruttiamo il fatto che la classe String implementi a sua volta Comparable<String>

Confronto tra oggetti: l'interfaccia Comparator

- In alternativa, si può realizzare una seconda classe che implementi l'interfaccia Comparator

```
public interface Comparator<T> {
     public int compare(T x, T y);
}
```

- Il **contratto** del metodo compare di Comparator è analogo a quello di compareTo di Comparable:
    - **Pre-condizione:** gli oggetti x e y sono confrontabili
    - **Post-condizione:**
      restituisce:
      - un valore negativo se x è minore di y
      - 0 se x è uguale a y
      - un valore positivo se x è maggiore di y

      inoltre, deve rappresentare una **relazione d'ordine** tra oggetti (si veda dopo)

Confronto tra oggetti: l'interfaccia Comparator

- L'uso di Comparator è indicato quando:
  - la classe da ordinare **non ha un unico criterio** di ordinamento naturale, oppure
  - la classe da ordinare è già stata realizzata e **non si può o non si vuole modificarla**

Esempio

- Supponiamo adesso di voler offrire **due diversi criteri di ordinamento** per gli Employee: per nome e per salario
- Per farlo, dobbiamo utilizzare l'interfaccia Comparator, come segue

```
public class Employee {
   private int salary;
   private String name;

   public static final Comparator<Employee> comparatorByName = new Comparator<>() {
      public int compare(T a, T b) {
         return a.name.compareTo(b.name);
      }
   };
   public static final Comparator<Employee> comparatorBySalary = new Comparator<>() {
      ...
   };
}
```

- Si noti l'uso di una classe anonima per l'inizializzazione di un campo statico

- In alternativa, si potrebbe usare anche una *lambda-espressione*, un costrutto illustrato in una lezione successiva

Il caso della classe String

- Le stringhe sono dotate di un **ordinamento naturale** che è quello **alfabetico** (o *lessicografico*)

- La classe String fornisce questo criterio di confronto implementando Comparable

- Nell'ordinamento naturale delle stringhe, le lettere maiuscole vengono prima delle minuscole
  - L'ordinamento naturale si basa sui codici UNICODE dei caratteri, che a loro volta si basano sui codici ASCII per i caratteri di base

Il caso della classe String

- Può essere utile anche ordinare stringhe senza considerare la distinzione tra minuscole e maiuscole *(case insensitive)*

- Poiché *non è possibile implementare Comparable in due modi diversi*, questo criterio di confronto alternativo deve essere fornito da un oggetto di tipo Comparator

- Difatti, nella classe String troviamo la seguente costante:

```
public static final Comparator<String> CASE_INSENSITIVE_ORDER;
```

- Riassumendo, la classe String offre un criterio di confronto "**naturale**", fornito dal metodo compareTo, e un criterio **alternativo**, sotto forma di oggetto Comparator disponibile ai client tramite una costante di classe

Proprietà dell'ordinamento

- Affinché l'implementazione di Comparable o Comparator definisca effettivamente un criterio di ordinamento tra oggetti, essa dovrà rispettare le seguenti proprietà

- Dato un numero reale a, definiamo la fuzione *segno* sgn(a):

$$\text{sgn}(a)=\begin{cases} 1 \text{ se } a>0 \\ 0 \text{ se } a=0 \\ -1 \text{ se } a<0 \end{cases}$$

- Dati tre oggetti x, y e z, appartenenti ad una classe che implementa Comparable, deve valere:

1) sgn(x.compareTo(y)) == -sgn(y.compareTo(x))
2) se x.compareTo(y)<0 e y.compareTo(z)<0 allora x.compareTo(z)<0
3) se x.compareTo(y)==0 allora sgn(x.compareTo(z))==sgn(y.compareTo(z))

- Condizioni analoghe devono valere per le implementazioni di Comparator

Proprietà dell'ordinamento

- Queste condizioni sono simili a quelle di una relazione d'ordine (riflessività, antisimmetria e transitività)

  1) sgn(x.compareTo(y)) == -sgn(y.compareTo(x)) **(riflessività e antisimmetria)**

  2) se x.compareTo(y)<0 e y.compareTo(z)<0 allora x.compareTo(z)<0 **(transitività)**

  3) se x.compareTo(y)==0 allora sgn(x.compareTo(z))==sgn(y.compareTo(z))

- La **relazione d'ordine indotta da Comparable** è x <= y sse x.compareTo(y) <=0

- Le 3 proprietà implicano che questa relazione è riflessiva, antisimmetrica e transitiva

- Esercizio: mostrare che la proprietà 1 implica la riflessività

Coerenza con equals

- E' preferibile, ma non obbligatorio, che le implementazioni di Comparable e Comparator siano *coerenti con equals*
- Nel caso di Comparable, questo vuol dire che

x.compareTo(y) == 0 se e solo se x.equals(y)==true

- Questo vincolo trova giustificazione nella Java Collection Framework, oggetto di lezioni successive
- Nota: Il comparatore `String.CASE_INSENSITIVE_ORDER` non è coerente con equals

Esercizio (esame 18/6/2012, #1)

La classe Point rappresenta un punto del piano cartesiano con coordinate intere:

```
public class Point {
  private int x, y;
  ...
}
```

Spiegare quali delle seguenti sono implementazioni valide per il metodo compare(Point a, Point b) di Comparator<Point>:

(Per semplicità, le seguenti pseudo-implementazioni accedono direttamente ai campi privati della classe Point)

1) return a.x-b.x;
2) return a.x-b.y;
3) return ((a.x*a.x)+(a.y*a.y)) - ((b.x*b.x)+(b.y*b.y));
4) return (a.x-b.x)+(a.y-b.y);
5) return (a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y);

Esercizio (esame 14/9/2010, #1)

Implementare la classe *Time*, che rappresenta un orario della giornata (dalle 00:00:00 alle 23:59:59). Gli orari devono essere confrontabili secondo Comparable. Il metodo *minus* accetta un altro orario *x* come argomento e restituisce la differenza tra questo orario e *x*, sotto forma di un nuovo oggetto *Time*. La classe fornisce anche gli orari predefiniti *MIDDAY* e *MIDNIGHT*.

Esempio d'uso:

```
Time t1 = new Time(14,35,0);
Time t2 = new Time(7,10,30);
Time t3 = t1.minus(t2);

System.out.println(t3);
System.out.println(t3.compareTo(t2));
System.out.println(t3.compareTo(Time.MIDDAY));
```

Output dell'esempio d'uso:

```
7:24:30
1
-1
```

Uso di comparatori per ordinare array

- Nell'API Java sono presenti dei metodi che utilizzano le interfacce Comparable e Comparator per fornire algoritmi di ordinamento di **array** e di **liste**
- Per quanto riguarda gli array, tali metodi si trovano nella classe *java.util.**Arrays***, una classe che contiene solo metodi statici
- Si tratta dei seguenti due metodi:

```
1) public static void sort(Object[] a)
```

- ordina l'array a in senso non-decrescente, in base all'**ordinamento naturale** tra i suoi elementi
- ovvero, suppone che tutti gli elementi contenuti siano confrontabili tra loro tramite l'interfaccia Comparable

```
2) public static <T> void sort(T[] a, Comparator<T> c) (semplificata)
```

- ordina la lista l in senso non-decrescente, in base all'ordinamento indotto dal **comparatore** c

- In entrambi i casi, l'ordinamento è *in-place* e *stabile*
  - cioè, l'array viene modificato senza utilizzare strutture di appoggio e gli elementi equivalenti secondo l'ordinamento mantengono l'ordine che avevano originariamente

- L'algoritmo usato è una versione ottimizzata del *merge sort*

Uso di comparatori per ordinare liste

- Per quanto riguarda le liste, i metodi di ordinamento si trovano nella classe *java.util.**Collections***, una classe che contiene solo metodi statici
- Si tratta dei seguenti due metodi:

```
public static void sort(List l)
```

- ordina la lista l in senso non-decrescente, in base all'ordinamento naturale tra i suoi elementi
- ovvero, suppone che tutti gli elementi contenuti siano confrontabili tra loro tramite l'interfaccia Comparable

```
public static void sort(List l, Comparator c)
```

- ordina la lista l in senso non-decrescente, in base all'ordinamento indotto dal comparatore c

- Come per gli array, l'ordinamento è *in-place* e *stabile*
- L'algoritmo usato è una versione ottimizzata del *merge sort*

- Nota: Per semplicità, in questa slide sono presentate le versioni *grezze* dei metodi, che non usano i parametri di tipo introdotti da Java 1.5

Ordinare in C

- E' interessante confrontare i metodi sort di Java con il metodo sort della libreria standard del linguaggio C
- Nel file header stdlib.h troviamo la seguente dichiarazione:

```
void qsort(void* base, size_t nmemb, size_t size, int (*compar)(const void*, const void*) )
```

- I primi tre argomenti servono a passare un array di tipo arbitrario, specificandone l'indirizzo di base, la lunghezza e la dimensione di ciascuna cella

- L'ultimo argomento è un puntatore a una funzione che accetta due elementi generici da confrontare (void *) e restituisce un intero

- L'analogia con Comparator è evidente
  - naturalmente, la funzione qsort è storicamente precedente al linguaggio Java