22.
Le classi enumerate

Marco Faella

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

Corso di Linguaggi di Programmazione II

Il problema dei tipi enumerati

- Un'esigenza ricorrente nella programmazione consiste nel creare un tipo di dati che può assumere **un numero fisso e limitato di valori**

- Ad esempio:
  - **giorno della settimana:** 7 possibili valori,
  - **seme di una carta** da gioco: 4 valori,
  - **mese dell'anno**: 12 valori,
  - etc.

Carte da gioco

- Supponiamo di dover implementare una classe **Card**, che rappresenta una carta da gioco francese
- Ogni carta è caratterizzata da un valore, da 1 a 13, ed uno dei quattro semi (in Inglese, *suit*)
- Che tipo di dati utilizzare per questi due attributi?
- Una soluzione ingenua (e fortemente sconsigliata per motivi che vedremo in seguito) è la seguente:

```
public class Card {
   private int val, suit;
   ...
}
```

- Un'altra soluzione simile è:

```
public class Card {
   private int val;
   private String suit;
   ...
}
```

Il problema dei tipi enumerati

- Nelle due implementazioni proposte nella slide precedente, si utilizza un tipo che ha **poca attinenza** con i valori rappresentati
- Così facendo, non riusciamo a distinguere una variabile che contenga un seme, da una che contiene un qualsiasi intero, o una qualsiasi stringa

- Ad esempio, se un metodo di Card come *setSuit* deve accettare un seme come argomento, deve necessariamente accettare un qualsiasi intero (o stringa), almeno in fase di compilazione

- Quindi, **il compilatore non può accorgersi** che invocazioni come *card.setSuit(5)* oppure *card.setSuit("Quori")* non hanno senso

Il problema dei tipi enumerati

- Riassumendo, utilizzare un tipo generico come int o String ha i seguenti problemi:
  1) int e String hanno molti più valori possibili dei 4 semi previsti
  2) non c'è una corrispondenza naturale tra i 4 semi e i valori di tipo int e String

- Per quanto riguarda il problema 2, potrebbe sembrare che associare ad ogni seme il suo nome sia una corrispondenza naturale
- Però, il nome di un seme si può scrivere in tanti modi diversi, come “Cuori”, “cuori”, “CUORI”, o “Hearts”

Il problema dei tipi enumerati

- Vista la discussione della slide precedente, sarebbe preferibile disporre di un tipo di dati che preveda **un insieme fissato di valori**

- I principali linguaggi di programmazione offrono supporto specifico a questa esigenza
- Ad esempio, i linguaggi C/C++ offrono il costrutto **enum**
- Il linguaggio ML offre il costrutto (molto più generale) **datatype**

- Fino alla versione 1.4, Java *non* offriva un simile supporto
- Tuttavia, anche in Java 1.4 è possibile *simulare* questo supporto
- La prossima slide illustra la soluzione standard a questo problema, che prende il nome di *Typesafe Enum Pattern*

Il Typesafe Enum Pattern

- Supponiamo di voler implementare la classe Card in Java 1.4 (senza enum)
- Invece di utilizzare un tipo base per rappresentare il seme, creiamo una classe **Suit** apposita
- Il nostro scopo consiste nell'assicurarci che la classe Suit possa avere **solo 4 istanze** predefinite, e che ciascun riferimento di tipo Suit possa puntare solo ad una di queste 4 istanze (oltre che a *null*)

Il Typesafe Enum Pattern

- Il pattern suggerisce di implementare una classe secondo il seguente schema:

```
public class SuitClass {
     private SuitClass(String name) { this.name = name; }
     public final String name;
     public static final SuitClass HEARTS   = new SuitClass(“Hearts”);
     public static final SuitClass SPADES   = new SuitClass(“Spades”);
     public static final SuitClass CLUBS    = new SuitClass(“Clubs”);
     public static final SuitClass DIAMONDS = new SuitClass(“Diamonds”);
}
```

- Riassumendo, le regole da seguire sono le seguenti:
  - La classe avrà **solo costruttori privati**
  - Ogni possibile valore del tipo enumerato corrisponderà ad una **costante** pubblica di **classe**, cioè un campo *public static final*
  - Ciascuna di queste costanti viene inizializzata usando uno dei costruttori privati

- Si noti che in questo caso specifico non è necessario fornire un metodo accessore per il campo immutabile “name”

Osservazioni

- Con la classe SuitClass abbiamo raggiunto lo scopo che ci eravamo prefissati: abbiamo creato un tipo di dati che *può assumere solo uno di quattro valori possibili*
- Infatti, il costruttore privato assicura che non sia possibile creare ulteriori istanze

- Il lettore attento potrebbe chiedersi perché non abbiamo riservato lo stesso trattamento anche all'attributo "valore" delle carte
- Anche in questo caso, si tratta di un attributo che può assumere un numero fisso e piuttosto limitato di valori validi: i numeri da 1 a 13
- Tuttavia, a differenza del seme, il valore di una carta **è intrinsecamente un numero**, e quindi rappresentarlo come tale presenta dei vantaggi, come la facilità di confrontare o sommare due valori

- Comunque, niente impedisce di trattare anche il valore allo stesso modo, definendo una ulteriore classe enumerata con 13 possibili valori
  - il tutorial ufficiale sulle enumerazioni Java suggerisce di procedere proprio così

Le classi enumerate in Java 1.5

- A partire dalla versione 1.5 di Java, il linguaggio offre supporto nativo ai tipi enumerati, tramite il concetto di ***classe enumerata***
- Una classe enumerata è un tipo particolare di classe, introdotta dalla parola chiave ***enum***, che prevede un numero fisso e predeterminato di istanze
- Continuando l'esempio dei semi, la corrispondente classe enumerata può essere definita come segue:

```
public enum SuitEnum {
   HEARTS, SPADES, CLUBS, DIAMONDS;
}
```

- Una classe enumerata può consistere semplicemente di un elenco di valori possibili
- Per molti versi, SuitEnum si comporta come SuitClass
- In particolare, i quattro valori dichiarati si utilizzano proprio come costanti pubbliche di classe, come ad esempio in:

```
SuitEnum s = SuitEnum.HEARTS;
```

- Però, come vedremo, SuitEnum dispone di **molte più funzionalità** di SuitClass, offerte in modo del tutto automatico e trasparente per il programmatore
- A differenza di SuitClass, **l'ordine** in cui i valori sono definiti in SuitEnum **è significativo**

Funzionalità delle classi enumerate

La classe enumerata SuitEnum gode delle seguenti funzionalità implicite (*built-in*):

```
SuitEnum x = SuitEnum.DIAMONDS;

int i = x.ordinal();

String name = x.name();


SuitEnum[] allSuits = SuitEnum.values();

SuitEnum y = Enum.<SuitEnum>valueOf(SuitEnum.class, “HEARTS”);
```

Potenzialità delle classi enumerate

- Una classe enumerata può essere molto più ricca della classe SuitEnum
- Può contenere **campi**, **metodi** e **costruttori**, come una classe normale
- Le uniche restrizioni riguardano i costruttori:
  - tutti i costruttori devono avere visibilità **privata** o **default** (di pacchetto)
  - non è possibile invocarli esplicitamente con new, neanche all'interno della classe stessa

- Ad esempio, supponiamo di voler distinguere i semi rappresentati sulle carte con il colore rosso (cuori e quadri) da quelli rappresentati in nero (fiori e picche)
- Aggiungiamo quindi un campo, un metodo e un costruttore, come segue:

```
public enum SuitEnum {
   HEARTS(true), SPADES(false), CLUBS(false), DIAMONDS(true);
   private final boolean red;
   private SuitEnum(boolean red) {
      this.red = red;
   }
   public boolean isRed() {
      return red;
   }
}
```

Proprietà delle classi enumerate

- Nell'esempio precedente, l'unica novità consiste nella sintassi usata per invocare un costruttore
- Se una classe enumerata ha più costruttori, ciascun valore può essere costruito con un costruttore diverso

- Inoltre, valgono le seguenti proprietà:

1) In una classe enumerata, la **prima riga** deve contenere l'elenco dei valori possibili
2) Le classi enumerate estendono automaticamente la classe parametrica **Enum**
    - quindi, le classi enumerate non possono estendere altre classi
    - precisamente, ogni classe enumerata E estende Enum<E>
3) Le classi enumerate sono automaticamente **final**

Ordinali dei valori enumerati

- Ad ogni valore di una classe enumerata è associato un **numero intero**, chiamato **ordinale**, che rappresenta il suo posto nella sequenza dei valori, a partire da zero
  - nel caso di SuitEnum, a HEARTS è associato 0, a SPADES 1, e così via
- Vediamo come si passa da ordinale a valore enumerato, e viceversa

- Per passare **da valore enumerato a ordinale**, si usa il seguente metodo della classe Enum

```
public int ordinal()
```

- Essendo pubblico, questo metodo viene ereditato da tutte le classi enumerate

- Per l'**operazione inversa**, si usa il seguente metodo statico, che ogni classe enumerata E possiede automaticamente (non appartiene alla classe Enum)

```
public static E[] values()
```

- Il metodo restituisce un array contenente tutti i possibili valori di E
- Quindi, per ottenere il valore di posto i-esimo, è sufficiente accedere all'elemento i-esimo dell'array restituito da values

Nomi dei valori enumerati

- E' anche possibile passare da un valore enumerato alla stringa che contiene il suo **nome**, come definito nel codice sorgente, e viceversa

- Per passare da valore a stringa, si usa il seguente metodo della classe Enum

```
public String name()
```

- Restituisce il nome di questo valore enumerato

- Per il passaggio inverso, la classe Enum offre il seguente metodo statico parametrico

```
public static <T extends Enum<T>> T valueOf(Class<T> enumType, String name)
```

- Restituisce il valore enumerato della classe *enumType* che ha nome *name*
- Il parametro di tipo del metodo rappresenta la classe enumerata a cui lo si applica
- Esempio:

```
SuitEnum x = Enum.<SuitEnum>valueOf(SuitEnum.class, “HEARTS”);
```

Specializzazione dei valori enumerati

- E' possibile *specializzare* il comportamento di un valore enumerato rispetto agli altri valori della stessa enumerazione
- In particolare, è possibile che un valore enumerato abbia una **versione particolare di un metodo** che è comune a tutta l'enumerazione
- Ad esempio, l'esempio precedente dei 4 semi divisi in rossi e neri si può anche realizzare nel modo seguente

```
public enum SuitEnum {
   HEARTS {
      public boolean isRed() { return true; }
   }, SPADES, CLUBS, DIAMONDS {
      public boolean isRed() { return true; }
   };
   public boolean isRed() {
      return false;
   }
}
```

- Per specializzare il comportamento di un valore, si inserisce il codice relativo subito dopo la dichiarazione di quel valore, racchiuso tra parentesi graffe
- Le versioni specializzate di isRed rappresentano un **overriding** del metodo presente in SuitEnum

Specializzazione dei valori enumerati

- Come ulteriore esempio, consideriamo una classe enumerata **BoolOp**, che rappresenta i principali **operatori booleani** binari (AND e OR)
- Vogliamo dotare la classe di un metodo **eval**, che accetta due valori booleani e calcola il risultato dell'operatore applicato a questi valori
- Per farlo, è possibile utilizzare la seguente sintassi:

```
public enum BoolOp {
   AND { public boolean eval(boolean a, boolean b) { return a && b; } },
   OR  { public boolean eval(boolean a, boolean b) { return a || b; } };
   public abstract boolean eval(boolean a, boolean b);
}
```

- Le enumerazioni possono avere metodi astratti pur non essendo astratte esse stesse

Specializzazione come classe anonima

Abbiamo già visto che un'enumerazione come:

```
public enum MyEnum { A, B, C; }
```

è molto simile alla seguente classe:

```
public class MyEnum {
   public static final MyEnum A = new MyEnum();
   ...
}
```

Specializzazione come classe anonima

- Un valore specializzato, come ad esempio:

```
public enum MyEnum { A { /* codice specifico per A */ }, B, C; }
```

va interpretato alla stregua di:

```
public class MyEnum {
   public static final MyEnum A = new MyEnum() {
      /* codice specifico per A */
   };
   ...
}
```

- Il codice specifico di un valore si comporta come se fosse inserito in una apposita **classe anonima**
- Infatti, nell'ambito del codice specifico di un certo valore è possibile inserire tutti i costrutti che possono comparire in una classe anonima, come **campi** e **metodi**

Collezioni per tipi enumerati

Collezioni per tipi enumerati

- Il Java Collection Framework offre delle collezioni specificamente progettate per le classi enumerate:
  - **EnumSet**, versione specializzata di Set
  - **EnumMap**, versione specializzata di Map

- EnumSet è una classe che implementa Set ed è ottimizzata per contenere elementi di una classe enumerata

- La sua intestazione completa è

```
public abstract class EnumSet<E extends Enum<E>>  extends AbstractSet<E>
                                                  implements Cloneable, Serializable
```

- Il limite superiore del parametro di tipo impone che tale tipo estenda Enum di se stesso
- Questo requisito è soddisfatto da tutte le classi enumerate

EnumSet e la sua implementazione

- Internamente, un EnumSet è un **vettore di bit** (*bit vector*)
- Ovvero, se un EnumSet dovrà contenere elementi di una classe enumerata che prevede **n valori** possibili, esso conterrà internamente un vettore di **n valori booleani**
- L'i-esimo valore booleano sarà vero se l'i-esimo valore enumerato appartiene all'insieme, e falso altrimenti
- Questa rappresentazione permette di realizzare in modo **estremamente efficiente** (tempo costante) tutte le operazioni base sugli insiemi previste dall'interfaccia Collection (add, remove, contains)
- E' facile rendersi conto che questa tecnica implementativa non potrebbe funzionare su classi non enumerate, che non hanno un numero prefissato e limitato di valori possibili

Creare un EnumSet

- Il discorso della slide precedente implica che un EnumSet **deve conoscere il numero di valori** possibili che ospiterà

- Inoltre, per motivi di performance la libreria offre due diverse versioni (sottoclassi di EnumSet):
  - Una per enumerazioni con **64 elementi o meno**
  - Una per enumerazioni con più di 64 elementi

- Pertanto, la classe EnumSet è astratta e non ha costruttori pubblici

- Per istanziarla, si utilizzano dei metodi statici, chiamati *metodi factory*

Creare un EnumSet

- Uno dei metodi factory è il seguente:

```
public static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> elemType)
```

- Questo metodo crea un EnumSet **vuoto**, predisposto per contenere elementi della classe enumerata elemType

- Passare l'oggetto di tipo class corrispondente alla classe enumerata permette all'EnumSet di **conoscere il numero di valori possibili** che ospiterà

- Ad esempio:

```
Collection<SuitEnum> c = EnumSet.noneOf(SuitEnum.class);
```

- Possiamo affidarci alla type inference per dedurre il parametro di tipo appropriato (cioè, E = SuitEnum)

Costruttori vs Metodi factory

- Un metodo factory ha lo scopo di creare oggetti di una certa classe
- Il caso più comune è quello di un metodo statico che crea oggetti della classe in cui si trova

- Vantaggi rispetto ad un costruttore:
  1) Un metodo statico non è obbligato a restituire un oggetto nuovo
     - Può quindi restituire un oggetto "riciclato"
     - E' il caso del metodo factory `Integer.valueOf`
  2) Un metodo statico non è obbligato a restituire uno specifico tipo effettivo
     - Può scegliere di restituire una o più sottoclassi
     - E' il caso del metodo factory `EnumSet.noneOf`

EnumMap

- EnumMap è una classe che implementa Map, ed è ottimizzata per i casi in cui le chiavi appartengano ad una classe enumerata
- Internamente, una EnumMap con valori di tipo V è semplicemente **un array di riferimenti di tipo V**
- L'ordinale delle chiavi funge da indice nell'array
- La sua intestazione completa è

```
public class EnumMap<K extends Enum<K>,V> extends AbstractMap<K,V>
                                          implements Serializable, Cloneable
```

- A differenza di EnumSet, EnumMap **ha costruttori** pubblici
- Il più semplice costruttore di EnumMap<K, V> è il seguente

```
public EnumMap(Class<K> keyType)
```

- Questo metodo crea una EnumMap vuota, predisposta per contenere chiavi della classe enumerata keyType
- Ad esempio:

```
Map<SuitEnum,Integer> m = new EnumMap<>(SuitEnum.class);
```

Esercizio (esame 19/6/2009, #1)

Implementare l'enumerazione *Cardinal*, che rappresenta le 16 direzioni della rosa dei venti, chiamate N (per Nord), NNE (per Nord Nord Est), NE, ENE, E, etc.

Il metodo *isOpposite* prende come argomento un punto cardinale *x* e restituisce *vero* se questo punto cardinale è diametralmente opposto ad *x*, e *falso* altrimenti.

Il metodo statico *mix* prende come argomento due punti cardinali, non opposti, e restituisce il punto cardinale intermedio tra i due. Se i due punti cardinali sono opposti, viene lanciata un'eccezione.

Esempio d'uso:

```
Cardinal nord = Cardinal.N;
System.out.println(nord.isOpposite(Cardinal.S));
Cardinal nordest = Cardinal.mix(Cardinal.N, Cardinal.E);
assert nordest==Cardinal.NE : "Errore inaspettato!";
Cardinal nordnordest = Cardinal.mix(nordest, Cardinal.N);
System.out.println(nordnordest);
```

Output dell'esempio d'uso:

```
true
NNE
```