12
Meccanismi Avanzati
Classi innestate e enumerazioni

Mirko Viroli
mirko.viroli@unibo.it

C.D.L. Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Cesena

a.a. 2014/2015

Outline

Goal della lezione

- Illustrare meccanismi avanzati della programmazione OO
- Dare linee guida sul loro utilizzo

Argomenti

- Enumerazioni
- Classi innestate statiche
- Inner class
- Classi locali
- Classi anonime
- Mappe del Collection Framework

Outline

Classi innestate statiche – idea

Principali elementi

- Dentro una classe `Outer`, è possibile innestare la definizione di un'altra classe `StaticNested`
- `StaticNested` è vista come se fosse una proprietà `statica` di `Outer` al pari di altri campi o metodi statici

```
class Outer {
    ...
    static class StaticNested {
        ...
    }
}
```

Classi innestate statiche – casistica

Possibilità di innestamento

- Anche una interfaccia può fungere da `Outer`
- Si possono innestare anche interfacce
- Il nesting può essere multiplo e/o multilivello
- L'accesso alle classi/interfacce innestate statiche avviene con sintassi `Outer.A`, `Outer.B`, `Outer.I`, `Outer.A.C`

```java
class Outer {
    ...
    static class A { .. static class C{..} ..}
    static class B {..}
    interface I {..} // static è implicito
}
```

Classi innestate statiche – accesso

Uso

- L'accesso alle classi/interfacce innestate statiche avviene con sintassi `Outer.StaticNested`
- Da dentro `Outer` si può accedere anche direttamente con `StaticNested`
- L'accesso da fuori `Outer` di `StaticNested` segue le regole del suo modificatore d'accesso
- Esterna e interna si vedono a vicenda anche le proprità `private`

```java
class Outer {
    ...
    static class StaticNested {
        ...
    }
}
..
Outer.StaticNested obj = new Outer.StaticNested(...);
```

Motivazioni

Una necessità generale

Vi sono situazioni in cui per risolvere un singolo problema è opportuno generare più classi, e non si vuole affiancarle così come ogni coppia di classi del package

Almeno) tre motivazioni (non necessariamente contemporanee)

- Evitare il proliferare di classi in un package, specialmente quando solo una di queste debba essere pubblica
- Migliorare l'incapsulamento, consentendo un meccanismo per consentire un accesso locale anche a proprietà `private`
- Migliorare la leggibilità, inserendo classi là dove serve (con nomi qualificati, quindi più espressivi)
- ..meglio comunque non abusare di questo meccanismo

Caso 1

Specializzazioni come classi innestate

- La classe astratta, o comunque base, è la outer
- Alcune specializzazioni ritenute frequenti e ovvie vengono innestate, ma comunque rese pubbliche

Esempio

- `Counter`, `Counter.Bidirectional`, `Counter.Multi`

Note

Un sintomo della possibilità di usare le classi nested per questo caso è quando ci si trova a costruire classi diverse costuite da un nome composto con una parte comune (`Counter`, `BiCounter`, `MultiCounter`)

Classe Counter e specializzazioni innestate (1/2)

```
public class Counter {

  protected int value;

  public Counter(int initialValue) {
    this.value = initialValue;
  }

  public void increment() {
    this.value++;
  }

  public int getValue() {
    return this.value;
  }

  public static class Multi extends Counter{
      ... // solito codice
  }

  public static class Bidirectional extends Counter{
      ... // solito codice
  }
}
```


Classe Counter e specializzazioni innestate (2/2)

```
public class Counter {

  ...
  // Codice della classe senza modifiche..
  public static class Multi extends Counter{

      public Multi(int initialValue){
        super(initialValue);
      }

      public void multiIncrement(int n){
        for (int i=0;i<n;i++){
            this.increment();
        }
      }
  }
    ...
  public static class Bidirectional extends Counter{
      ... // solito codice
  }
}
```

Uso di Counter e specializzazioni innestate

```
1  import java.util.*;
2
3  public class UseCounter {
4
5    public static void main(String[] args) {
6      List<Counter> list = new ArrayList<>();
7      list.add(new Counter(100));
8      list.add(new Counter.Bidirectional(100));
9      list.add(new Counter.Multi(100));
10
11     for (Counter c : list){
12       c.increment();
13     }
14
15   }
16
17 }
```

Caso 2

Necessità di una classe separata ai fini di ereditaerità

In una classe potrebbero servire sotto-comportamenti che debbano:

- implementare una data interfaccia
- estendere una data classe

Esempio

- `Range`, `Range.Iterator`

Nota

In tal caso spesso tale classe separata non deve essere visibile dall'esterno, quindi viene indicata come `private`

Classe Range e suo iteratore (1/2)

```java
public class Range implements Iterable<Integer>{

    final private int start;
    final private int stop;

    public Range(int start, int stop){
      this.start = start;
      this.stop = stop;
    }

    public java.util.Iterator<Integer> iterator(){
      return new Iterator(this.start,this.stop);
    }

    private static class Iterator
              implements java.util.Iterator<Integer>{
        ...
    }
}
```

Classe Range e suo iteratore (2/2)

```java
public class Range implements Iterable<Integer>{
    ...
    private static class Iterator
              implements java.util.Iterator<Integer>{

        private int current;
        private int stop;

        public Iterator(int start, int stop){
          this.current = start;
          this.stop = stop;
        }

        public Integer next(){
          return this.current++;
        }

        public boolean hasNext(){
          return this.current <= this.stop;
        }

        public void remove(){}
    }
}
```

Uso di Range

```java
public class UseRange{
    public static void main(String[] s){
      for (int i: new Range(5,12)){
          System.out.println(i);
          // 5 6 7 8 9 10 11 12
      }
    }
}
```

Caso 3

Necessità di comporre una o più classi diverse

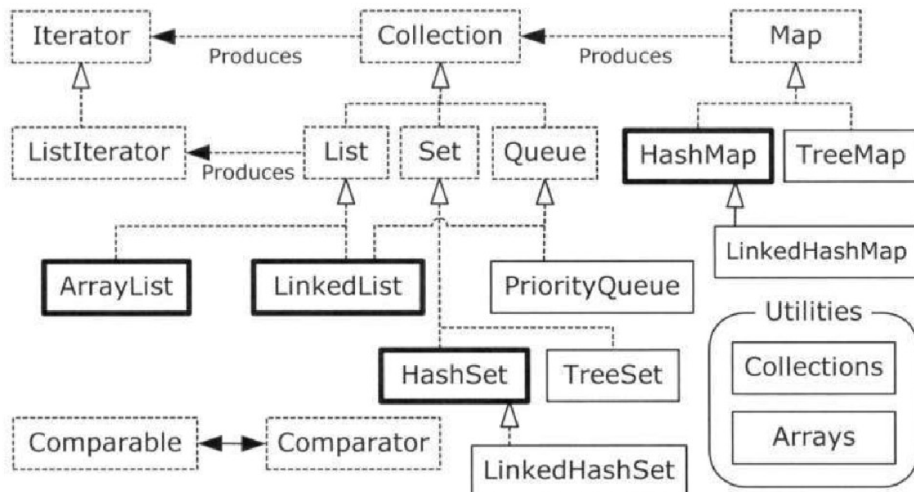
- Ognuna realizzi un sotto-comportamento
- Per suddividere lo stato dell'oggetto
- Tali classi non utilizzabili indipendentemente dalla outer

Esempio tratto dal Collection Framework

- `Map`, `Map.Entry`
- (una mappa è un osservabile come set di entry)

Outline

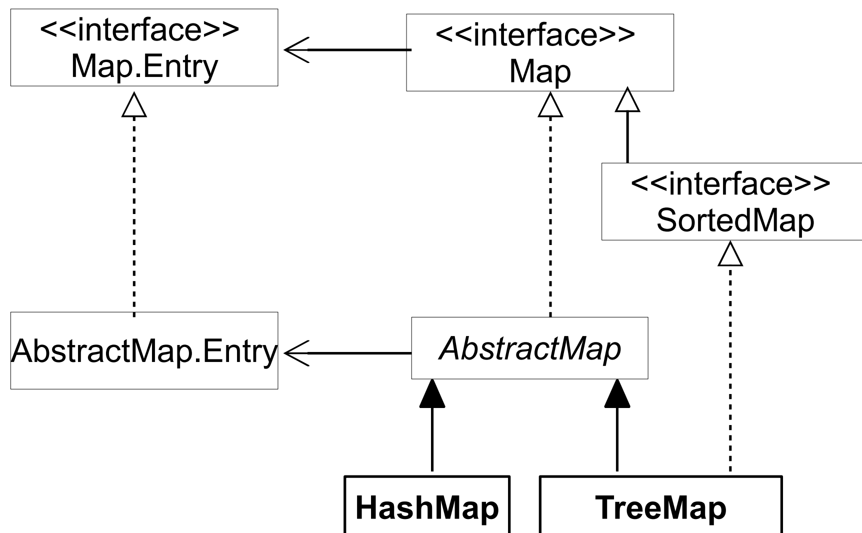
JCF – struttura semplificata



Map

```java
public interface Map<K,V> {

    // Query Operations
    int size();
    boolean isEmpty();
    boolean containsKey(Object key);         // usa Object.equals
    boolean containsValue(Object value);     // usa Object.equals
    V get(Object key);                       // accesso a valore

    // Modification Operations
    V put(K key, V value);          // inserimento chiave-valore
    V remove(Object key);           // rimozione chiave(-valore)

    // Bulk Operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear();                   // cancella tutti

    // Views
    Set<K> keySet();                    // set di valori
    Collection<V> values();             // collezione di chiavi
    Set<Map.Entry<K, V>> entrySet();    // set di chiavi-valore

    interface Entry<K,V> {...}          // public static implicito!
}
```


Implementazione mappe – UML



Map.Entry

Ruolo di Map.Entry

- Una mappa può essere vista come una collezione di coppie chiave-valore, ognuna incapsulata in un Map.Entry
- Quindi, una mappa è composta da un set di Map.Entry

```
public interface Map<K,V> {

    ...

    Set<Map.Entry<K, V>> entrySet();

    interface Entry<K,V> { // public e static implicite!

        K getKey();
        V getValue();
        V setValue(V value);

    }
}
```


Uso di Map.Entry

```
package map;

import java.util.*;

public class UseMap2 {
  public static void main(String[] args) {
    // Uso una incarnazione, ma poi lavoro sull'interfaccia
    Map<Integer, String> map = new HashMap<>();
    // Una mappa è una funzione discreta
    map.put(345211, "Bianchi");
    map.put(345122, "Rossi");
    map.put(243001, "Verdi");

    for (Map.Entry<Integer, String> entry : map.entrySet()) {
      System.out.println(entry);
      entry.setValue(null);
    }
    System.out.println(map);
    // {345211=null, 243001=null, 345122=null}
  }
}
```

La classe AbstractMap

In modo simile a AbstractSet

- Fornisce una implementazione scheletro per una mappa
- Necessita di un solo metodo da implementare: entrySet()
- Così facendo si ottiene una mappa iterabile e non modificabile
- Per fare modifiche è necessario ridefinire altri metodi..

Una semplice specializzazione di AbstractMap

```
package map;
import java.util.*;

public class CapitalsMap extends AbstractMap<String,String>{

  private final static Set<Map.Entry<String,String>> SET;

  static{ // costruisce il valore di SET una volta per tutte
    SET = new HashSet<>();
    SET.add(new AbstractMap.SimpleEntry<>("Italy","Rome"));
    SET.add(new AbstractMap.SimpleEntry<>("France","Paris"));
    SET.add(new AbstractMap.SimpleEntry<>("Germany","Berlin"));
  }

  public CapitalsMap(){}

  // Questo è l'unico metodo che è necessario implementare
  public Set<java.util.Map.Entry<String, String>> entrySet() {
    return SET;
  }

  public static void main(String[] args){
    CapitalsMap cmap = new CapitalsMap();
    System.out.println("Capital of Italy: "+cmap.get("Italy"));//Rome
    System.out.println("Capital of Spain: "+cmap.get("Spain"));//null
    System.out.println("All CapitalsMap: "+cmap);
  }
}
```

X

Outline

Inner Class – idea

Principali elementi

- Dentro una classe `Outer`, è possibile innestare la definizione di un'altra classe `InnerClass` (senza indicazione `static`!)
- `InnerClass` è vista come se fosse una proprietà non-statica di `Outer` al pari di altri campi o metodi
- L'effetto è che una istanza di `InnerClass` è sempre logicamente racchiusa in una istanza di `Outer` (enclosing instance), accessibile con la sintassi `Outer.this`

```
class Outer {
    ...
    class InnerClass { // Nota.. non è static!
        ...
        // ogni oggetto di InnerClass avrà un riferimento ad
        // un oggetto di Outer, denominato Outer.this
    }
}
```


Un semplice esempio

```
public class Outer {

  private int i;

  public Outer(int i){
    this.i=i;
  }

  public Inner createInner(){
    return new Inner();
    // oppure: return this.new Inner();
  }

  public class Inner {

    private int j = 0;

    public void update(){
      // si usa l'oggetto di outer..
      this.j = this.j + Outer.this.i;
    }

    public int getValue(){
      return this.j;
    }
  }
}
```

Uso di Inner e Outer

```
public class UseOuter {

  public static void main(String[] args) {
    Outer o = new Outer(5);
    Outer.Inner in = o.new Inner(); // o racchiude in
    System.out.println(in.getValue()); // 0
    in.update();
    in.update();
    System.out.println(in.getValue()); // 5

    Outer.Inner in2 = new Outer(10).createInner();
    in2.update();
    in2.update();
    System.out.println(in2.getValue()); // 20
  }
}
```

Enclosing instance – istanza esterna

Gli oggetti delle inner class

- Sono creati con espressioni:
  `<obj-outer>.new <classe-inner>(<args>)`
- (la parte `<obj-outer>` è omettibile quando sarebbe `this`)
- Possono accedere all'enclosing instance con notazione `<classe-outer>.this`

Motivazioni

- Tutte quelle relative alle classi innestate statiche, più..
- ...quando è necessario che ogni oggetto inner tenga un riferimento all'oggetto outer

Esempio

- La classe `Range` già vista usa una static nested class, che però ben usufruirebbe del riferimento all'oggetto di `Range` che l'ha generata

Una variante di Range

```
public class Range2 implements Iterable<Integer>{

    final private int start;
    final private int stop; // final è essenziale in questo caso
    public Range2(int start, int stop){
      .. // usuale implementazione
    }
    public java.util.Iterator<Integer> iterator(){
      return this.new Iterator();
    }

    private class Iterator implements java.util.Iterator<Integer>{

        private int current;

        public Iterator(){
          this.current = Range2.this.start;
        }

        public Integer next(){
          return this.current++;
        }
        public boolean hasNext(){
          return this.current <= Range2.this.stop;
        }
        public void remove(){}
    }
}
```


Outline

Classi locali – idea

Principali elementi

- Dentro un metodo di una classe `Outer`, è possibile innestare la definizione di un'altra classe `LocalClass` (senza indicazione `static`!)
- La `LocalClass` è a tutti gli effetti una inner class (e quindi ha enclosing instance)
- In più, la `LocalClass` "vede" anche le variabili nello scope del metodo in cui è definita, usabili solo se final

```
class Outer {
    ...
    void m(final int x){
        final String s=..;
        class LocalClass { // Nota.. non è static!
        ... // può usare Outer.this, s e x
        }
        LocalClass c=new LocalClass(...);
    }
}
```


Range tramite classe locale

```
public class Range3 implements Iterable<Integer>{

    final private int start;
    final private int stop;
    public Range3(int start, int stop){
      //.. usuale implementazione
    }

    public java.util.Iterator<Integer> iterator(){
      class Iterator implements java.util.Iterator<Integer>{

            private int current;

            public Iterator(){
              this.current = Range3.this.start;
            }
            public Integer next(){
              return this.current++;
            }
            public boolean hasNext(){
              return this.current <= Range3.this.stop;
            }
            public void remove(){}
        }
      return new Iterator();
    }
}
```

Classi locali – motivazioni

Perché usare una classe locale invece di una inner class

- Tale classe è necessaria solo dentro ad un metodo, e lì la si vuole confinare
- È eventualmente utile accedere anche alle variabili del metodo

Outline

Classi anonime – idea

Principali elementi

- Con una variante dell'istruzione `new`, è possibile innestare la definizione di un'altra classe senza indicarne il nome
- In tale definizione non possono comparire costruttori
- Viene creata al volo una classe locale, e da lì se ne crea un oggetto
- Tale oggetto, come per le classi locali, ha enclosing instance e "vede" anche le variabili final nello scope del metodo in cui è definita

```
1 class C {
2     ...
3     Object m(final int x){
4         return new Object(){
5             public String toString(){ return "Valgo "+x; }
6         }
7     }
8 }
```


Range tramite classe anonima – la soluzione ottimale

```
1 public class Range4 implements Iterable<Integer>{
2
3     final private int start;
4     final private int stop;
5
6     public Range4(int start, int stop){
7       this.start = start;
8       this.stop = stop;
9     }
10
11    public java.util.Iterator<Integer> iterator(){
12      return new java.util.Iterator<Integer>(){
13            // Non ci può essere costruttore!
14            private int current = start; // o anche Range.this.start
15
16            public Integer next(){
17              return this.current++;
18            }
19            public boolean hasNext(){
20              return this.current <= stop; // o anche Range.this.stop
21            }
22            public void remove(){}
23        };
24    }
25 }
```

Classi anonime – motivazioni

Perchè usare una classe anonima invece di una locale

- Se ne deve creare un solo oggetto, quindi è inutile anche solo nominarla
- È eventualmente utile accedere anche alle variabili del metodo
- Per implementare "al volo" una interfaccia

Altro esempio: classe anonima da Comparable

```
import java.util.*;

public class UseSort {

  public static void main(String[] args) {
    List<Integer> list = Arrays.asList(10,40,7,57,13,19,21,35);
    System.out.println(list);
    // classe anonima a partire da una interfaccia
    Collections.sort(list,new Comparator<Integer>(){
      public int compare(Integer a,Integer b){
        return a > b ? 1 : (a == b ? 0 : -1); //deboxing
      }
    });
    System.out.println(list);

    Collections.sort(list,new Comparator<Integer>(){
      public int compare(Integer a,Integer b){
        return a < b ? 1 : (a == b ? 0 : -1);
      }
    });
    System.out.println(list);
  }
}
```

Riassunto e linee guida

Inner class (e varianti)

Utili quando si vuole isolare un sotto-comportamento in una classe a sé, senza dichiararne una nuova che si affianchi alla liste di quelle fornite dal package, ma stia "dentro" una classs più importante

Se deve essere visibile alle altre classi

- Quasi sicuramente, una static nested class
- Raro usare la inner class, la sintassi `.new` è poco nota

Se deve essere invisibile da fuori

- Si sceglie uno dei quattro casi a seconda della visibilità che la inner class deve avere/dare
  - static nested class: solo parte statica
  - inner class: anche enclosing class, accessibile ovunque dall'outer
  - local class: anche argomenti/variabili, accessibile da un solo metodo
  - anonymous class: per creare un oggetto, senza un nuovo costruttore

Preview Java 8

Un pattern molto ricorrente

- Avere classi anonime usate per incapsulare metodi "funzionali" (senza stato)
- Java 8 introduce le lambda come notazione semplificata

Outline

Enumerazioni

Motivazioni

- in alcune situazioni occorre definire dei tipi che possono assumere solo un numero fissato e basso di possibili valori (o oggetti)
- Esempi:
  - le cifre da 0 a 9, le regioni d'Italia, il sesso di un individuo, i 6 pezzi negli scacchi

Possibili realizzazioni in Java

- usare degli `int` per codificarli (come in C): poco leggibile
- usare delle classi astratte, e una concreta per valore: prolisso

Enumerazioni: `enum { ...}`

- consentono di elencare i valori, associando ad ognuno un nome
- è possibile collegare un behaviour diverso ad ogni valore

Esempio classe Persona: 1/2

```java
import java.util.*;

public class Persona {
  private String nome;
  private String cognome;
  private String regione;

  public Persona(String nome, String cognome, String regione) {
    super();
    this.nome = nome;
    this.cognome = cognome;
    this.regione = regione;
  }

  public String toString() {
    return "[" + nome + "," + cognome + "," + regione + "]";
  }
```

Esempio classe Persona: 2/2

```java
  public boolean isIsolano(){
    // Confronto lento!!
    return (this.regione.equals("Sardegna") ||
            this.regione.equals("Sicilia"));
  }

  public static List<Persona>
          fromRegione(Collection<Persona> coll,String regione){
    ArrayList<Persona> list = new ArrayList<>();
    for (Persona persona: coll){
        // Confronto lento!!
      if (persona.regione.equals(regione)){
        list.add(persona);
      }
    }
    return list;
  }
}
```

UsePersona

```
package enums;

import java.util.ArrayList;
import java.util.List;

public class UsePersona {
  public static void main(String[] args){
    ArrayList<Persona> list = new ArrayList<>();
    list.add(new Persona("Mario","Rossi","Emilia-Romagna"));
    list.add(new Persona("Gino","Bianchi","Sicilia"));
    list.add(new Persona("Carlo","Verdi","EmiliaRomagna"));
    // Errore sul nome non intercettabile
    List<Persona> out = Persona.fromRegione(list,"Emilia-Romagna");
    System.out.println(list);
    // [[Mario,Rossi,Emilia-Romagna], [Gino,Bianchi,Sicilia],
    //    [Carlo,Verdi,EmiliaRomagna]]
    System.out.println(out);
    // [[Mario,Rossi,Emilia-Romagna]]
    for (Persona p: list){
      if (p.isIsolano()){
        System.out.println(p);
      }
    }
    // [Gino,Bianchi,Sicilia]
  }
}
```


Soluzione alternativa, Persona: 1/2

```
import java.util.*;

public class Persona {

    public static final int LOMBARDIA = 0;
    public static final int EMILIA_ROMAGNA = 1;
    public static final int SICILIA = 2;
    public static final int SARDEGNA = 3;
    ...

  private String nome;
  private String cognome;
  private int regione;

  public Persona(String nome, String cognome, int regione) {
    super();
    this.nome = nome;
    this.cognome = cognome;
    this.regione = regione;
  }

  private static String nomeRegione(int regione){
      switch (regione){
      case 0: return "Lombardia";
      case 1: return "Emilia-Romagna";
      ...
      }
  }
```

Soluzione alternativa Persona: 2/2

```
    public String toString() {
    return "[" + nome + "," + cognome + "," + nomeRegione(regione) + "]";
  }

  public boolean isIsolano(){
    // Confronto veloce!!
    return (this.regione == SARDEGNA ||
            this.regione == SICILIA);
  }

  public static List<Persona>
          fromRegione(Collection<Persona> coll,
                              String regione){
    ArrayList<Persona> list = new ArrayList<>();
    for (Persona persona: coll){
        // Confronto veloce!!
      if (persona.regione == regione){
        list.add(persona);
      }
    }
    return list;
  }
}
```

Soluzione alternativa UsePersona

```
package enums;

import java.util.ArrayList;
import java.util.List;

public class UsePersona {
  public static void main(String[] args){
    ArrayList<Persona> list = new ArrayList<>();
    list.add(new Persona("Mario","Rossi",Persona.EMILIA_ROMAGNA));
    list.add(new Persona("Gino","Bianchi",3));
    list.add(new Persona("Carlo","Verdi",Persona.LOMBARDIA));
    // Non si hanno errori sulle stringhe
    // Non si può prevenire l'uso diretto di interi..
    ...
  }
}
```

Discussione

Approccio a stringhe

- Penalizza molto le performance spazio-tempo
- Può comportare errori gravi per scorrette digitazioni
- Difficile intercettare gli errori

Approccio a interi – soluzione pre-enumerazioni

- Buone performance
- Può comportare comunque errori, anche se più difficilmente
- L'uso delle costante è un poco dispersivo

altri approcci: uso di classi diverse per ogni valore

- Impraticabile con un numero elevato di valori
- Comunque molto prolisso in termini di quantità di codice
- Previene gli errori che si possono commettere

enum Regione

```
public enum Regione {
  ABRUZZO, BASILICATA, CALABRIA, CAMPANIA, EMILIA_ROMAGNA,
  FRIULI_VENEZIA_GIULIA, LAZIO, LIGURIA, LOMBARDIA, MARCHE,
  MOLISE, PIEMONTE, PUGLIA, SARDEGNA, SICILIA, TOSCANA,
  TRENTINO_ALTO_ADIGE, UMBRIA, VALLE_D_AOSTA, VENETO;
}
```

```
import java.util.*;

public class UseEnum {
  public static void main(String[] args) {
    List<Regione> list = new ArrayList<>();

    list.add(Regione.LOMBARDIA);
    list.add(Regione.PIEMONTE);
    list.add(Regione.LIGURIA);

    for (Regione r: list){
      System.out.println(r);
    }
  }
}
```

Persona con uso della enum

```java
public class Persona {
  private String nome;
  private String cognome;
  private Regione regione;

  public Persona(String nome, String cognome, Regione regione) {
      ... // al solito
  }
  public String toString() { // Nota il toString() di Regione!
    return "[" + nome + "," + cognome + "," + regione + "]";
  }

  public boolean isIsolano(){
    return (regione.equals(Regione.SARDEGNA) ||
            regione.equals(Regione.SICILIA));
  }

  public static List<Persona> fromRegione(Collection<Persona> coll,
                                          Regione regione){
    ArrayList<Persona> list = new ArrayList<>();
    for (Persona persona: coll){
      if (persona.regione == regione){
        list.add(persona);
      }
    }
    return list;
  }
}
```


UsePersona con uso della enum

```java
import java.util.*;

public class UsePersona {
  public static void main(String[] args){
    ArrayList<Persona> list = new ArrayList<>();
    list.add(new Persona("Mario","Rossi",Regione.EMILIA_ROMAGNA));
    list.add(new Persona("Gino","Bianchi",Regione.SICILIA));
    list.add(new Persona("Carlo","Verdi",Regione.LOMBARDIA));
    List<Persona> out =
        Persona.fromRegione(list,Regione.EMILIA_ROMAGNA);
    System.out.println(list);
    // [[Mario,Rossi,EMILIA_ROMAGNA], [Gino,Bianchi,SICILIA],
    // [Carlo,Verdi,LOMBARDIA]]
    System.out.println(out);
    // [[Mario,Rossi,EMILIA_ROMAGNA]]
    for (Persona p: list){
      if (p.isIsolano()){
        System.out.println(p);
      }
    }
    // [Gino,Bianchi,SICILIA]
  }
}
```

UsePersona con import static

```java
package enums2;

import java.util.*;
import static enums2.Regione.*;
// Lo static import consente di evitare 'Regione.' ovunque sarebbe servito

public class UsePersona {
  public static void main(String[] args){
    ArrayList<Persona> list = new ArrayList<>();
    list.add(new Persona("Mario","Rossi",EMILIA_ROMAGNA));
    list.add(new Persona("Gino","Bianchi",SICILIA));
    list.add(new Persona("Carlo","Verdi",LOMBARDIA));
    List<Persona> out = Persona.fromRegione(list,EMILIA_ROMAGNA);
    System.out.println(list);
    // [[Mario,Rossi,EMILIA_ROMAGNA], [Gino,Bianchi,SICILIA],
    // [Carlo,Verdi,LOMBARDIA]]
    System.out.println(out);
    // [[Mario,Rossi,EMILIA_ROMAGNA]]
    for (Persona p: list){
      if (p.isIsolano()){
        System.out.println(p);
      }
    }
    // [Gino,Bianchi,SICILIA]
  }
}
```


Discussione

Cosa "nasconde sotto" una enum

È realizzata con una classe che incapsula un campo intero e un campo statico che tiene le stringhe dei nomi dei valori

Approccio a enum

- Performance piuttosto buone
- Impedisce completamente errori di programmazione
- Il codice aggiuntivo da produrre non è elevato

Funzionalità aggiuntive per le enum

- Ve ne sono varie, alcune delle quali presentate di seguito
- Si consiglia di non abusarne, per evitare di perdere i vantaggi di semplicità che le enum comportano

Metodi di default per ogni enum

```
package enums2;
import static enums2.Regione.*; // Consente di evitare 'Regione.'
import java.util.*;


public class UseRegione {
  public static void main(String[] args) {
    ArrayList<Regione> list = new ArrayList<>();
    // 4 modi di ottenere una Regione
    list.add(Regione.LOMBARDIA);
    list.add(SARDEGNA);
    list.add(Regione.valueOf("SICILIA"));
    list.add(Regione.values()[10]);

    for (Regione r: list){
      System.out.println("toString "+r); // LOMBARDIA,...,MOLISE
      System.out.println("ordinale "+r.ordinal()); // 8, 13, 14, 10
      System.out.println("nome "+r.name()); // LOMBARDIA,...,MOLISE
      System.out.println("---");
    }

    for (Regione r: Regione.values()){
      System.out.print(r+" "); // Stampa tutte le regioni
    }

  }
}
```


enum negli switch

```
package enums2;
import static enums2.Regione.*; // Consente di evitare 'Regione.'
import java.util.*;


public class UseRegione2 {
  public static void main(String[] args) {
    ArrayList<Regione> list = new ArrayList<>();
    // 4 modi di ottenere una Regione
    list.add(Regione.LOMBARDIA);
    list.add(SARDEGNA);
    list.add(Regione.valueOf("SICILIA"));
    list.add(Regione.values()[10]);

    // Le enum sono usabili negli switch
    for (Regione r: list){
      switch (r){
      case LOMBARDIA: System.out.println("Lombardia");
              break;
      case EMILIA_ROMAGNA: System.out.println("Emilia Romagna");
              break;
      default: System.out.println("Altre..");

      }
    }
  }
}
```

Metodi aggiuntivi nelle enum

```
public enum Regione {
  ABRUZZO, BASILICATA, CALABRIA, CAMPANIA, EMILIA_ROMAGNA,
  FRIULI_VENEZIA_GIULIA, LAZIO, LIGURIA, LOMBARDIA, MARCHE,
  MOLISE, PIEMONTE, PUGLIA, SARDEGNA, SICILIA, TOSCANA,
  TRENTINO_ALTO_ADIGE, UMBRIA, VALLE_D_AOSTA, VENETO;

  boolean isIsola(){
    return this == SARDEGNA || this == SICILIA;
  }
}

public class Persona {
  private String nome;
  private String cognome;
  private Regione regione;
    ...
  public boolean isIsolano(){
    return this.regione.isIsola();
  }
  ...
}
```

Metodi e campi aggiuntivi nelle enum: Zona

```
package enums4;
import java.util.*;

public enum Zona {
  NORD, CENTRO, SUD;

  List<Regione> getRegioni(){
    ArrayList<Regione> list=new ArrayList<>();
    for (Regione r: Regione.values()){
      if (r.getZona()==this){
        list.add(r);
      }
    }
    return list;
  }
}
```

Metodi e campi aggiuntivi nelle enum: Regione

```
package enums4;
import static enums4.Zona.*;

public enum Regione {
  ABRUZZO(CENTRO,"Abruzzo"),
  BASILICATA(SUD,"Basilicata"),
  CALABRIA(SUD,"Calabria"),
  ...
  VALLE_D_AOSTA(NORD,"Valle D'Aosta"),
  VENETO(NORD,"Veneto");

  private Zona z;
  private String actualName;

  private Regione(Zona z, String actualName){
    this.z=z;
    this.actualName=actualName;
  }

  public Zona getZona(){
    return this.z;
  }

  public String toString(){
    return this.actualName;
  }
}
```


Metodi e campi aggiuntivi nelle enum: UseZona

```
package enums4;
import static enums4.Zona.*; // Consente di evitare 'Regione.'


public class UseZona {
  public static void main(String[] args) {
    for (Regione r: NORD.getRegioni()){
      System.out.println("toString "+r);
      // Emilia Romagna,...,Veneto
      System.out.println("nome "+r.name());
      // EMILIA_ROMAGNA,...,VENETO
      System.out.println("---");
    }
  }
}
```

Meccanismi per le enum

Riassunto

- Esistono metodi istanza e statici disponibili per Enum
- Si possono aggiungere metodi
- Si possono aggiungere campi e costruttori

Riguardando la enum Regione

- È una classe standard, con l'indicazioni di alcuni oggetti predefiniti
- I 20 oggetti corrispondenti alle regioni italiane

Quindi

- È possibile intuirne la realizzazione interna
- E quindi capire meglio quando e come usarli

⇒ In caso in cui i valori sono "molti e sono noti", oppure..

⇒ Anche se i valori sono pochi, ma senza aggiungere troppi altri metodi..

Una realizzazione equivalente di Regione

```
package enums4;
import static enums4.Zona.*;

public class RegioneEquiv {
    public static final ABRUZZO = new Regione(CENTRO,"Abruzzo",0);
    public static final BASILICATA = new Regione(SUD,"Basilicata",1);
  ...
  public static final VENETO = new Regione(NORD,"Veneto",19);

  private static String[] strings = new String[]{
      "ABRUZZO", "BASILICATA",...,"VENETO"
  };
  private static RegioneEquiv[] values = new RegioneEquiv[]{
      ABRUZZO, BASILICATA,...,VENETO
  };

  private Zona z;
  private String name;
  private int ordinal;

  private Regione(Zona z, String actualName,int ordinal){
    ... // usuale implementazione
  }
  ...
  public static RegioneEquiv[] values(){ return values;}
  public int ordinal(){ return this.ordinal;}
  public string name(){ return strings[ordinal];}
}
```

enum innestate

Motivazione

- Anche le enum (statiche) possono essere innestate in una classe o interfaccia
- Questo è utile quando il loro uso è reputato essere confinato nel funzionamento della classe outer

In linea di principio..

È possibile usare anche enum inner e locali, ma la loro utilità non sembra tale da controbilanciare la complicazione che ne conseguirebbe, specialmente in termini di leggibilità del codice

Persona con enum innestata

```
1  public class Persona {
2
3    public static enum Regione {
4      ABRUZZO, BASILICATA, CALABRIA, CAMPANIA, EMILIA_ROMAGNA,
5      FRIULI_VENEZIA_GIULIA, LAZIO, LIGURIA, LOMBARDIA, MARCHE,
6      MOLISE, PIEMONTE, PUGLIA, SARDEGNA, SICILIA, TOSCANA,
7      TRENTINO_ALTO_ADIGE, UMBRIA, VALLE_D_AOSTA, VENETO;
8    }
9
10   private String nome;
11   private String cognome;
12   private Regione regione;
13
14   public Persona(String nome, String cognome, Regione regione) {
15     ... // implementazione usuale..
16   }
17
18   public String toString() {
19     return "[" + nome + "," + cognome + "," + regione + "]";
20   }
21
22   public boolean isIsolano(){
23     // nota la qualificazione Regione.
24     return (this.regione.equals(Regione.SARDEGNA) ||
25             this.regione.equals(Regione.SICILIA));
26   }
27 }
```

UsePersona con uso della enum innestata

```
package enums2bis;

import java.util.*;

public class UsePersona {
  public static void main(String[] args){
    ArrayList<Persona> list = new ArrayList<>();
    list.add(new Persona("Gino","Bianchi",Persona.Regione.SICILIA));
    list.add(new Persona("Carlo","Verdi",Persona.Regione.LOMBARDIA));
    System.out.println(list);
  }
}
```

```
package enums2bis;

import java.util.*;
import static enum2bis.Persona.Regione.*;

public class UsePersona2 {
  public static void main(String[] args){
    ArrayList<Persona> list = new ArrayList<>();
    list.add(new Persona("Gino","Bianchi",SICILIA));
    list.add(new Persona("Carlo","Verdi",LOMBARDIA));
    System.out.println(list);
  }
}
```