13
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. 2018/2019

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 e terminologia

Principali elementi

- Dentro una classe `A`, chiamata outer è possibile innestare la definizione di un'altra classe `B`, chiamata innestata (statica) – in inglese, static nested
- `B` viene quindi vista come se fosse una proprietà `statica` di `A` (richiamabile via `A`, come: tipo, per le `new` e le chiamate statiche)

```
1 // situazione di partenza
2 class A {...}
3 class B {...}
```

```
1 // modifica, usando le inner class
2 class A {
3     ...
4     static class B { .. }
5 }
```

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`

```
1 class Outer {
2     ...
3     static class A { .. static class C{..} ..}
4     static class B {..}
5     interface I {..} // static è implicito
6 }
```

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 proprietà `private`

```
1 class Outer {
2     ...
3     static class StaticNested {
4         ...
5     }
6 }
7 ..
8 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 solo come classi dello stesso 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, con 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
- due implicazioni:
  - schema di nome delle inner class
  - possibilità di accedere alle proprietà statiche

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)

```
1  public class Counter {
2
3    private int value; // o protected..
4
5    public Counter(int initialValue) {
6      this.value = initialValue;
7    }
8
9    public void increment() {
10     this.value++;
11   }
12
13   public int getValue() {
14     return this.value;
15   }
16
17   public static class Multi extends Counter{
18       ... // solito codice
19   }
20
21   public static class Bidirectional extends Counter{
22       ... // solito codice
23   }
24 }
```

Classe Counter e specializzazioni innestate (2/2)

```
1  public class Counter {
2
3    ...
4    // Codice della classe senza modifiche..
5    public static class Multi extends Counter{
6
7        public Multi(int initialValue){
8          super(initialValue);
9        }
10
11       public void multiIncrement(int n){
12         for (int i=0;i<n;i++){
13             this.increment();
14         }
15       }
16   }
17     ...
18   public static class Bidirectional extends Counter{
19       ... // solito codice
20   }
21 }
```

Uso di Counter e specializzazioni innestate

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

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)

```
1  public class Range implements Iterable<Integer>{
2
3      final private int start;
4      final private int stop;
5
6      public Range(final int start, final int stop){
7        this.start = start;
8        this.stop = stop;
9      }
10
11     public java.util.Iterator<Integer> iterator(){
12       return new Iterator(this.start,this.stop);
13     }
14
15     private static class Iterator
16               implements java.util.Iterator<Integer>{
17         ...
18     }
19 }
```

Classe Range e suo iteratore (2/2)

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

Uso di Range

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

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 "osservabile" come set di entry)

Riassunto classi innestate statiche

Principali aspetti

- Da fuori (se pubblica) vi si accede con nome `Outer.StaticNested`
- `Outer` e `StaticNested` sono co-locate: si vedono le proprietà `private`

Motivazione generale

- Voglio evitare la proliferazione di classi nel package
- Voglio sfruttare l'incapsulamento

Motivazione per il caso `public`

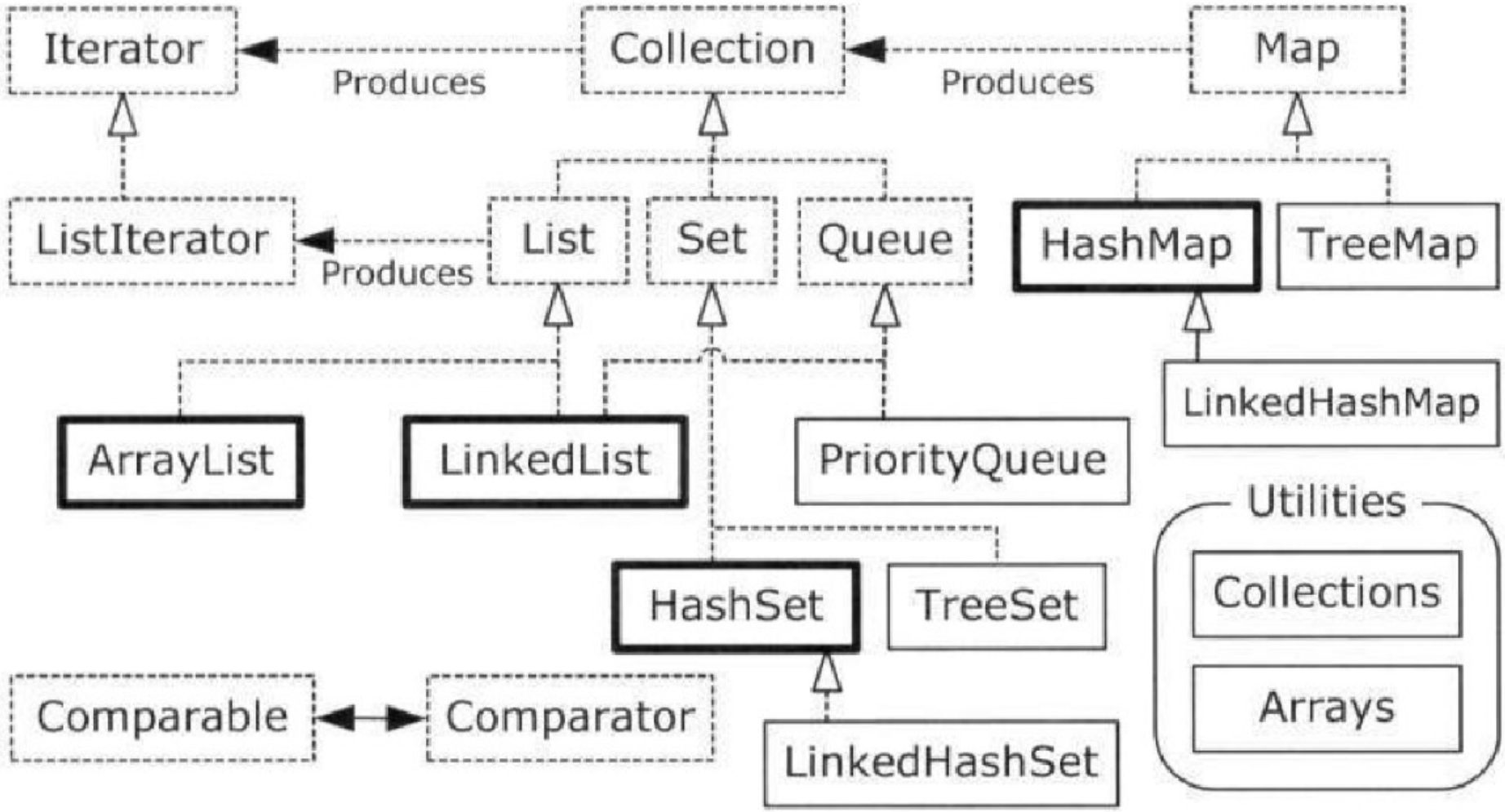
- Voglio enfatizzare i nomi `Out.C1`, `Out.C2`,..

Motivazione per il caso `private` – è il caso più frequente

- Voglio realizzare una classe a solo uso della outer, invisibile alle altre classi del package

Outline

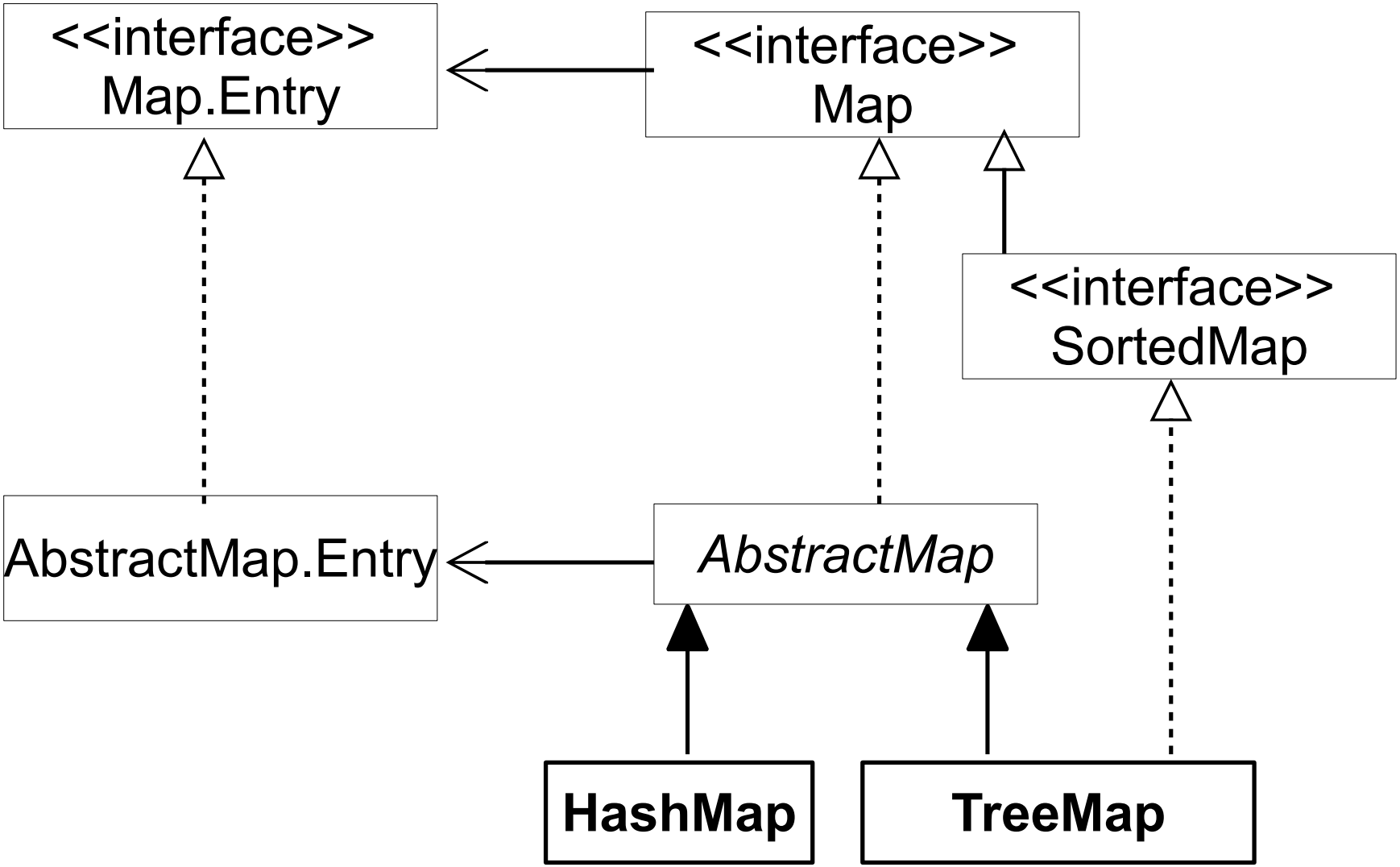
JCF – struttura semplificata



Map

```
1  public interface Map<K,V> {
2
3      // Query Operations
4      int size();
5      boolean isEmpty();
6      boolean containsKey(Object key);          // usa Object.equals
7      boolean containsValue(Object value);      // usa Object.equals
8      V get(Object key);                        // accesso a valore
9
10     // Modification Operations
11     V put(K key, V value);              // inserimento chiave-valore
12     V remove(Object key);               // rimozione chiave(-valore)
13
14     // Bulk Operations
15     void putAll(Map<? extends K, ? extends V> m);
16     void clear();                       // cancella tutti
17
18     // Views
19     Set<K> keySet();                        // set di valori
20     Collection<V> values();                 // collezione di chiavi
21     Set<Map.Entry<K, V>> entrySet();        // set di chiavi-valore
22
23     interface Entry<K,V> {...}              // public static implicito!
24 }
```

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

```
1  public interface Map<K,V> {
2
3      ...
4
5      Set<Map.Entry<K, V>> entrySet();
6
7      interface Entry<K,V> { // public e static implicite!
8
9          K getKey();
10         V getValue();
11         V setValue(V value);
12
13     }
14 }
```

Uso di Map.Entry

```java
public class UseMap2 {

  public static void main(String[] args) {

    // Al solito, uso una incarnazione, ma poi lavoro sull'interfaccia
    final 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 (final Map.Entry<Integer, String> entry : map.entrySet()) {
      System.out.println(entry.getClass());
      System.out.println(entry.getKey());
      System.out.println(entry.getValue());
      entry.setValue(entry.getValue()+"...");
    }
    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

```
1  public class CapitalsMap extends AbstractMap<String,String>{
2
3    private static final Set<Map.Entry<String,String>> set;
4
5    // inizializzatore statico..
6    // usato per inizializzare i campi statici
7    static{
8      // costruisce il valore di set
9      set = new HashSet<>();
10     set.add(new AbstractMap.SimpleEntry<>("Italy","Rome"));
11     set.add(new AbstractMap.SimpleEntry<>("France","Paris"));
12     set.add(new AbstractMap.SimpleEntry<>("Germany","Berlin"));
13   }
14
15   public CapitalsMap(){}
16
17   // Questo è l'unico metodo che è necessario implementare
18   public Set<java.util.Map.Entry<String, String>> entrySet() {
19     return set;
20   }
21
22 }
```

UseCapitalsMap

```
1  public class UseCapitalsMap{
2
3    public static void main(String[] args){
4      CapitalsMap cmap = new CapitalsMap();
5      System.out.println("Capital of Italy: "+cmap.get("Italy"));
6      System.out.println("Capital of Spain: "+cmap.get("Spain"));
7      System.out.println("All CapitalsMap: "+cmap);
8
9      // Iterazione "lenta" su una mappa
10     for (final String key: cmap.keySet()){
11       System.out.println("K,V: "+key+" "+cmap.get(key));
12     }
13
14     // Iterazione veloce su una mappa
15     for (final Map.Entry<String, String> entry: cmap.entrySet()){
16       System.out.println("E: "+entry+" "+entry.getKey()+" "+entry.getValue()
       );
17     }
18   }
19 }
```

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` ha sempre un riferimento ad una istanza di `Outer` (enclosing instance) che ne rappresenta il contesto, accessibile con la sintassi `Outer.this`, e ai suoi campi (privati)

```
1 class Outer {
2     ...
3     class InnerClass { // Nota.. non è static!
4         ...
5         // ogni oggetto di InnerClass avrà un riferimento ad
6         // un oggetto di Outer, denominato Outer.this
7     }
```

Un semplice esempio

```
1  public class Outer {
2
3    private int i;
4
5    public Outer(int i){
6      this.i=i;
7    }
8
9    public Inner createInner(){
10     return new Inner();
11     // oppure: return this.new Inner();
12   }
13
14   public class Inner {
15
16     private int j = 0;
17
18     public void update(){
19       // si usa l'oggetto di outer..
20       this.j = this.j + Outer.this.i;
21     }
22
23     public int getValue(){
24       return this.j;
25     }
26   }
27 }
```

Uso di Inner e Outer

```
1  public class UseOuter {
2
3    public static void main(String[] args) {
4      Outer o = new Outer(5);
5      Outer.Inner in = o.new Inner();
6      System.out.println(in.getValue()); // 0
7      in.update();
8      in.update();
9      System.out.println(in.getValue()); // 5
10
11     Outer.Inner in2 = new Outer(10).createInner();
12     in2.update();
13     in2.update();
14     System.out.println(in2.getValue()); // 20
15   }
16 }
```

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: quelle relative alle classi innestate statiche, più..

- ...quando è necessario che ogni oggetto inner tenga un riferimento all'oggetto outer
- pragmaticamente: usato quasi esclusivamente il caso `private`

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

```
1  public class Range2 implements Iterable<Integer> {
2
3    private final int start;
4    private final int stop;
5
6    public Range2(final int start, final int stop) {
7      this.start = start;
8      this.stop = stop;
9    }
10
11   public java.util.Iterator<Integer> iterator() {
12     return this.new Iterator();
13   }
14
15   private class Iterator implements java.util.Iterator<Integer> {
16
17     private int current;
18
19     public Iterator() {
20       this.current = Range2.this.start; // this.current = start
21     }
22
23     public Integer next() {
24       return this.current++;
25     }
26
27     public boolean hasNext() {
28       return this.current <= Range2.this.stop;
29     }
30
31     public void remove() {}
32   }
```

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

```
1  class Outer {
2      ...
3      void m(final int x){
4          final String s=..;
5          class LocalClass { // Nota.. non è static!
6          ... // può usare Outer.this, s e x
7          }
8          LocalClass c=new LocalClass(...);
9      }
10 }
```

Range tramite classe locale

```
1  public class Range3 implements Iterable<Integer>{
2
3      private final int start;
4      private final int stop;
5
6      public Range3(final int start, final int stop){
7        this.start = start;
8        this.stop = stop;
9      }
10
11     public java.util.Iterator<Integer> iterator(){
12       class Iterator implements java.util.Iterator<Integer>{
13
14             private int current;
15
16             public Iterator(){
17               this.current = Range3.this.start;
18             }
19
20             public Integer next(){
21               return this.current++;
22             }
23
24             public boolean hasNext(){
25               return this.current <= Range3.this.stop;
26             }
27
28             public void remove(){}
29         }
30       return new Iterator();
31     }
32 }
```

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

Pragmaticamente

- Mai viste usarle.. si usano invece le classi anonime..

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      private final int start;
4      private final int stop;
5
6      public Range4(final int start, final 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 Range4.this.start
15
16             public Integer next(){
17               return this.current++;
18             }
19             public boolean hasNext(){
20               return this.current <= stop; // o anche Range4.this.stop
21             }
22             public void remove(){}
23         }; // questo è il ; del return!!
24     }
25 }
```

Classi anonime – motivazioni

Perchè usare una classe anonima?

- Se ne deve creare un solo oggetto, quindi è inutile anche solo nominarla
- Si vuole evitare la proliferazione di classi
- Tipicamente: per implementare "al volo" una interfaccia

Altro esempio: classe anonima da Comparable

```
1  public class UseSort {
2
3    public static void main(String[] args) {
4      final List<Integer> list = Arrays.asList
       (10,40,7,57,13,19,21,35);
5      System.out.println(list);
6      // classe anonima a partire da una interfaccia
7      Collections.sort(list,new Comparator<Integer>(){
8        public int compare(Integer a,Integer b){
9          return a-b;
10       }
11     });
12     System.out.println(list);
13
14     Collections.sort(list,new Comparator<Integer>(){
15       public int compare(Integer a,Integer b){
16         return b-a;
17       }
18     });
19     System.out.println(list);
20   }
21 }
```

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 class più importante

Se deve essere visibile alle altre classi

- Quasi sicuramente, una static nested class

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"
- Java 8 introduce le lambda come notazione semplificata
- È il punto di partenza per la combinazione OO + funzionale

```
1  public class UseSortLambda {
2
3    public static void main(String[] args) {
4      final List<Integer> list = Arrays.asList
       (10,40,7,57,13,19,21,35);
5      System.out.println(list);
6      // classe anonima a partire da una interfaccia
7      Collections.sort(list,(a,b)->a-b);
8      System.out.println(list);
9
10     Collections.sort(list,(a,b)->b-a);
11     System.out.println(list);
12   }
13 }
```

Outline

Enumerazioni

Motivazioni

- in alcune situazioni occorre definire dei tipi che possono assumere solo un numero fissato e limitato di possibili valori
- Esempi:
  - le cifre da 0 a 9, le regioni d'Italia, il sesso di un individuo, i 6 pezzi negli scacchi, i giorni della settimana, le tipologie di camere di un hotel, le scuole di un ateneo, eccetera

Possibili realizzazioni in Java

- usare delle `String` rappresentando il loro nome: quasi assurdo
- 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 metodi e campi ad ogni "valore"

Esempio classe Persona: 1/2

```
1  public class Persona {
2    private final String nome;
3    private final String cognome;
4    private final String regione;
5
6    public Persona(String nome, String cognome, String regione) {
7      this.nome = nome;
8      this.cognome = cognome;
9      this.regione = regione;
10   }
11
12   public String toString() {
13     return "[" + nome + "," + cognome + "," + regione + "]";
14   }
```

Esempio classe Persona: 2/2

```
1
2   public boolean isIsolano() { // Confronto molto lento!!
3     return (this.regione.equals("Sardegna") ||
4         this.regione.equals("Sicilia"));
5   }
6
7   public static List<Persona> fromRegione(
8       final Collection<Persona> coll,
9       final String regione
10  ) { // note this possible way of formatting..
11    final List<Persona> list = new ArrayList<>();
12    for (final Persona persona : coll) {
13      if (persona.regione.equals(regione)) { // Confronto lento!!
14        list.add(persona);
15      }
16    }
17    return list;
18  }
19 }
```

UsePersona

```java
public class UsePersona {
  public static void main(String[] args){
    final 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
    final 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 (final Persona p: list){
      if (p.isIsolano()){
        System.out.println(p);
      }
    }
    // [Gino,Bianchi,Sicilia]
  }
}
```

Soluzione alternativa, Persona: 1/2

```
1  import java.util.*;
2
3  public class Persona {
4
5      public static final int LOMBARDIA = 0;
6      public static final int EMILIA_ROMAGNA = 1;
7      public static final int SICILIA = 2;
8      public static final int SARDEGNA = 3;
9      ...
10
11     private final String nome;
12     private final String cognome;
13     private final int regione;
14
15     public Persona(String nome, String cognome, int regione) {
16       this.nome = nome;
17       this.cognome = cognome;
18       this.regione = regione;
19     }
20
21     private static String nomeRegione(int regione){
22         switch (regione){
23           case 0: return "Lombardia";
24           case 1: return "Emilia-Romagna";
25           ...
26         }
27     }
```

Soluzione alternativa Persona: 2/2

```
1   public String toString() {
2     return "[" + nome + "," + cognome + "," + nomeRegione(regione) + "]";
3   }
4
5   public boolean isIsolano(){
6     // Confronto veloce!!
7     return (this.regione == SARDEGNA ||
8             this.regione == SICILIA);
9   }
10
11  public static List<Persona> fromRegione(Collection<Persona> coll,
12                                          String regione){
13    ArrayList<Persona> list = new ArrayList<>();
14    for (Persona persona: coll){
15        // Confronto veloce!!
16      if (persona.regione == regione){
17        list.add(persona);
18      }
19    }
20    return list;
21  }
22 }
```

Soluzione alternativa UsePersona

```java
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 ma cattiva leggibilità
- 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 molto elevato di valori
- Comunque molto prolisso in termini di quantità di codice
- Tuttavia:
  - Previene gli errori che si possono commettere
  - Consente facilmente di aggiungere nuovi elementi

enum Regione

```java
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;
}
```

```java
import java.util.*;

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

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

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

Persona con uso della enum

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

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

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

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

```

UsePersona con uso della enum

```java
import java.util.*;

public class UsePersona {
  public static void main(String[] args){
    final 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));
    final 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 (final Persona p: list){
      if (p.isIsolano()){
        System.out.println(p);
      }
    }
    // [Gino,Bianchi,SICILIA]
  }
}
```

UsePersona con import static

```java
import static it.unibo.apice.oop.p13advanced.enums.en2.Regione.*;

import java.util.*;

public class UsePersona2 {
  public static void main(String[] args){
    final 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));
    final 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 (final 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 estende `java.lang.Enum`, e incapsula un campo intero che tiene l'ID del valore, più un campo statico che tiene le stringhe dei nomi di tutti i valori

Approccio a `enum`

- Performance piuttosto buone
- Impedisce interamente errori di programmazione
- Il codice aggiuntivo da produrre non è elevato
- Non facilmente estendibile per aggiungere nuovi casi

Funzionalità aggiuntive per le `enum`

- Ve ne sono varie, alcune delle quali presentate di seguito

Metodi di default per ogni enum

```java
import static it.unibo.apice.oop.p13advanced.enums.en2.Regione.*;

import java.util.*;


public class UseRegione {
  public static void main(String[] args) {
    final 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 (final 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 (final Regione r: Regione.values()){
      System.out.print(r+" "); // Stampa tutte le regioni
    }

  }
}
```

enum negli switch

```
1  import static it.unibo.apice.oop.p13advanced.enums.en2.Regione.*;
2
3  import java.util.*;
4
5  public class UseRegione2 {
6    public static void main(String[] args) {
7      final ArrayList<Regione> list = new ArrayList<>();
8      // 4 modi di ottenere una Regione
9      list.add(Regione.LOMBARDIA);
10     list.add(SARDEGNA);
11     list.add(Regione.valueOf("SICILIA"));
12     list.add(Regione.values()[10]);
13
14     // Le enum sono usabili negli switch
15     for (final Regione r : list) {
16       switch (r) {
17       case LOMBARDIA:
18         System.out.println("Lombardia");
19         break;
20       case EMILIA_ROMAGNA:
21         System.out.println("Emilia Romagna");
22         break;
23       default:
24         System.out.println("Altre..");
25       }
26     }
27   }
28 }
```

Metodi aggiuntivi nelle enum: Zona

```
1  import java.util.*;
2
3  public enum Zona {
4    NORD, CENTRO, SUD;
5
6    public List<Regione> getRegioni(){
7      final ArrayList<Regione> list=new ArrayList<>();
8      for (final Regione r: Regione.values()){
9        if (r.getZona()==this){
10         list.add(r);
11       }
12     }
13     return list;
14   }
15 }
```

Metodi e campi aggiuntivi nelle enum: Regione (1/2)

```
1  import static it.unibo.apice.oop.p13advanced.enums.en3.Zona.*;
2
3  public enum Regione {
4    ABRUZZO(CENTRO,"Abruzzo"),
5    BASILICATA(SUD,"Basilicata"),
6    CALABRIA(SUD,"Calabria"),
7    CAMPANIA(SUD,"Campania"),
8    EMILIA_ROMAGNA(NORD, "Emilia Romagna"),
9    FRIULI_VENEZIA_GIULIA(NORD, "Friuli Venezia Giula"),
10   LAZIO(CENTRO, "Lazio"),
11   LIGURIA(NORD, "Liguria"),
12   LOMBARDIA(NORD, "Lombardia"),
13   MARCHE(CENTRO, "Marche"),
14   MOLISE(SUD, "Molise"),
15   PIEMONTE(NORD, "Piemonte"),
16   PUGLIA(SUD, "Puglia"),
17   SARDEGNA(SUD, "Sardegna"),
18   SICILIA(SUD, "Sicilia"),
19   TOSCANA(CENTRO, "Toscana"),
20   TRENTINO_ALTO_ADIGE(NORD, "Trentino Alto Adige"),
21   UMBRIA(CENTRO, "Umbria"),
22   VALLE_D_AOSTA(NORD,"Valle D'Aosta"),
23   VENETO(NORD,"Veneto");
24
25   private final Zona z;
26   private final String actualName;
27
28   private Regione(final Zona z, final String actualName){
29     this.z = z;
30     this.actualName = actualName;
31   }
32
```

Metodi e campi aggiuntivi nelle enum: Regione (2/2)

```
1    SICILIA(SUD, "Sicilia"),
2    TOSCANA(CENTRO, "Toscana"),
3    TRENTINO_ALTO_ADIGE(NORD, "Trentino Alto Adige"),
4    UMBRIA(CENTRO, "Umbria"),
5    VALLE_D_AOSTA(NORD,"Valle D'Aosta"),
6    VENETO(NORD,"Veneto");
7
8    private final Zona z;
9    private final String actualName;
10
11   private Regione(final Zona z, final String actualName){
12     this.z = z;
13     this.actualName = actualName;
14   }
15
16   public Zona getZona(){
17     return this.z;
18   }
19
20   public String toString(){
21     return this.actualName;
22   }
23 }
```

Metodi e campi aggiuntivi nelle enum: UseZona

```
1  import static it.unibo.apice.oop.p13advanced.enums.en3.Zona.*;
2
3  public class UseZona {
4    public static void main(String[] args) {
5      for (Regione r: NORD.getRegioni()){
6        System.out.println("toString "+r);
7        // Emilia Romagna,...,Veneto
8        System.out.println("nome "+r.name());
9        // EMILIA_ROMAGNA,...,VENETO
10       System.out.println("---");
11     }
12   }
13 }
```

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

$\Rightarrow$ In caso in cui i valori sono "molti e sono noti", oppure..

$\Rightarrow$ Anche se i valori sono pochi, ma senza aggiungere troppi altri metodi..

Una realizzazione equivalente di Regione

```
1  package enums4;
2  import static enums4.Zona.*;
3
4  public class RegioneEquiv {
5      public static final ABRUZZO = new Regione(CENTRO,"Abruzzo",0);
6      public static final BASILICATA = new Regione(SUD,"Basilicata",1);
7    ...
8    public static final VENETO = new Regione(NORD,"Veneto",19);
9
10   private static String[] strings = new String[]{
11       "ABRUZZO", "BASILICATA",...,"VENETO"
12   };
13   private static RegioneEquiv[] values = new RegioneEquiv[]{
14       ABRUZZO, BASILICATA,...,VENETO
15   };
16
17   private Zona z;
18   private String name;
19   private int ordinal;
20
21   private Regione(Zona z, String actualName,int ordinal){
22     ... // usuale implementazione
23   }
24   ...
25   public static RegioneEquiv[] values(){ return values;}
26   public int ordinal(){ return this.ordinal;}
27   public string name(){ return strings[ordinal];}
28 }
```

`enum` innestate

Motivazione

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

Esempio

- `enum Regione` potrebbe essere inserita dentro `Persona`
- `enum Zona` potrebbe essere inserita dentro `Regione`