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. 2022/2023

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 un membro `statico` 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 mutuamente i membri `private`
  - ossia il significato completo di `private` è:
  - "privato a livello della outerclass più esterna" (vista come unità di design)

```
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 si reputa sconveniente usare sorgenti diversi

Almeno tre motivazioni (non necessariamente contemporanee)

- Evitare il proliferare di classi in un package, specialmente quando poche di queste debbano essere pubbliche
- Migliorare l'incapsulamento, con un meccanismo per consentire un accesso locale anche a membri `private`
- Migliorare la leggibilità, inserendo classi là dove serve (con nomi qualificati, quindi più espressivi)

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 ai membri statici

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 i membri `private`

Motivazione generale

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

Motivazione per il caso `public`

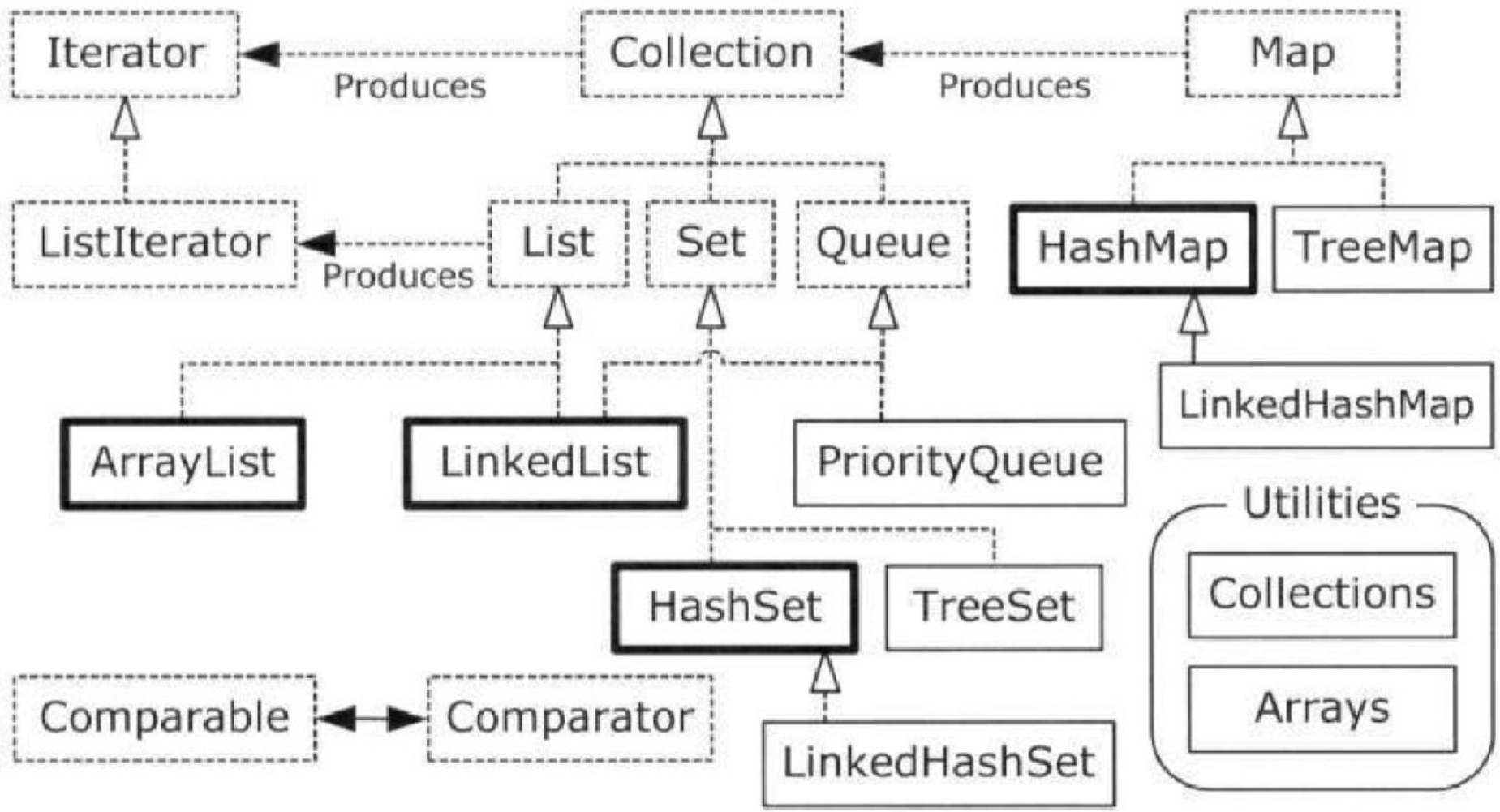
- Voglio enfatizzare i nomi `Out.C1`, `Out.C2`,..

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

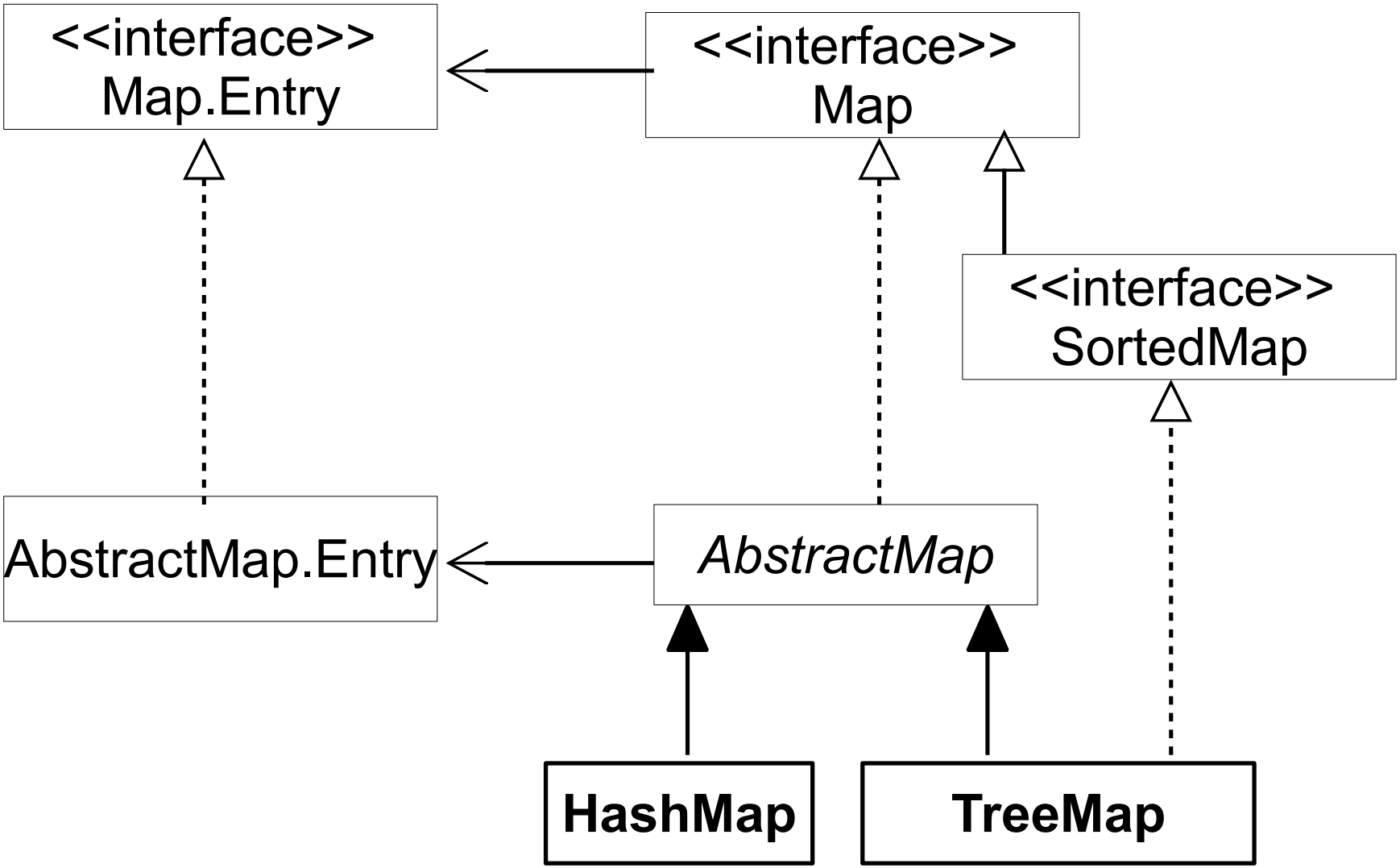
JCF – struttura semplificata



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 chiavi
20     Collection<V> values();                 // collezione di valori
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 UseMap {

  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}
  }
}
```

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 un membro non-statico di `Outer` al pari di altri campi o metodi, quindi da richiamare (ad esempio quando si fa una `new`) su una istanza di `Outer`, chiamata "enclosing instance"
- L'effetto è che una istanza di `InnerClass` ha sempre un riferimento ad una enclosing instance accessibile con la sintassi `Outer.this`, che ne rappresenta il contesto

```
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     }
8 }
```

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, o se "di fatto finali"

```
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 (o di fatto finali) 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 lista di quelle fornite dal package, ma stia "dentro" una classe 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 lambda expressions

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, che non cambierà in futuro
- 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: astrazione errata
- usare degli `int` per codificarli (come in C): di basso livello
- usare una classe (e singolo oggetto) per elemento: prolisso

Enumerazioni: `enum { ...}`

- consentono di elencare i valori, associando ad ognuno un nome
- è possibile collegare metodi e campi ad ogni "valore"

Esempio classe Person

```java
public class Person {
  private final String name;
  private final String surname;
  private final String region;

  public Person(String name, String surname, String region) {
    this.name = name;
    this.surname = surname;
    this.region = region;
  }

  public boolean isIslander() { // Confronto lento e incline a errori!!
    return (this.region.equals("Sardegna") ||
        this.region.equals("Sicilia"));
  }

  public boolean fromRegion(String region){
    return this.region.equals(region);
  }

  public String toString() {
    return "Person [" + name + ", " + surname + ", " + region + "]";  }
}
```

UsePerson

```java
import java.util.*;

public class UsePerson {
  public static void main(String[] args){
    final List<Person> list = new ArrayList<>();
    list.add(new Person("Mario","Rossi","Emilia-Romagna"));
    list.add(new Person("Gino","Bianchi","Sicilia"));
    list.add(new Person("Carlo","Verdi","EmiliaRomagna"));
    // Errore sul nome non intercettabile
    System.out.println(list.get(0).fromRegion("Emilia Romagna"));

    // Prestazioni problematiche su collezioni lunghe
    for (final Person p: list){
      if (p.isIslander()){
        System.out.println(p);
      }
    }
  }
}
```

Soluzione alternativa (con int), Person

```java
public class Person {
  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; // and so on...

  private final String name;
  private final String surname;
  private final int region;

  public Person(String name, String surname, int region) {
    this.name = name;
    this.surname = surname;
    this.region = region;
  }

  public boolean isIslander() { // Confronto molto veloce!!
    return (this.region == SICILIA || this.region == SARDEGNA);
  }

  public boolean fromRegion(int region){ // Servirebbe controllo sull'input
    return this.region == region;
  }

  public String getRegionName(){
    switch(this.region){
      case 0: return "Lombardia";
      case 1: return "Emilia-Romagna";
      //... and so on
      default: return "?";
    }
  }
}
```

Soluzione alternativa (con int), UsePerson

```
1  import java.util.*;
2
3  public class UsePerson {
4    public static void main(String[] args){
5      final List<Person> list = new ArrayList<>();
6      list.add(new Person("Mario","Rossi",Person.EMILIA_ROMAGNA));
7      list.add(new Person("Gino","Bianchi",Person.SICILIA));
8      list.add(new Person("Carlo","Verdi",0));
9      list.add(new Person("Carlo","Verdi",-23)); // ??
10
11     // Errore sul nome da intercettare via controllo su interi
12     System.out.println(list.get(0).fromRegion(-23));
13
14     // Prestazioni OK!
15     for (final Person p: list){
16       if (p.isIslander()){
17         System.out.println(p);
18       }
19     }
20   }
21 }
```

Soluzione con polimorfismo

```
1  interface Region {
2      String getName();
3  }
4
5  class Sicilia implements Region {
6      public String getName() {
7          return "Sicilia";
8      }
9      // tweak equals to equate all objects of the same class
10     public boolean equals(Object obj) {
11         return obj instanceof Sicilia;
12     }
13 }
14
15 class Sardegna implements Region {
16     public String getName() {
17         return "Sardegna";
18     }
19     // tweak equals to equate all objects of the same class
20     public boolean equals(Object obj) {
21         return obj instanceof Sardegna;
22     }
23 }
24 //... and so on
```

Soluzione con polimorfismo

```
1  class Person {
2      private String name;
3      private String surname;
4      private Region region;
5
6      public Person(String name, String surname, Region region) {
7          this.name = name;
8          this.surname = surname;
9          this.region = region;
10     }
11
12     public boolean isIslander() {
13         return this.region == new Sicilia() || this.region == new Sardegna();
14     }
15     // and so on...
16 }
17
18 public class UseRegionWithPolymorphism {
19     public static void main(String[] args) {
20         Person p = new Person("mario", "rossi", new Sicilia());
21         // and so on...
22     }
23 }
```

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

Approccio a polimorfismo: uso di classi diverse per ogni valore

- Difficilmente praticabile con un numero molto elevato di valori
- Prolisso, anche in termine di generazione di oggetti
- Tuttavia è sicuro e estendibile!
  - Previene gli errori che si possono commettere
  - Consente facilmente di aggiungere nuovi elementi

Soluzione pre enumerazioni: costanti e costruttore privato

```java
public class Region {

  public static final Region MARCHE = new Region(0,"Marche");
  public static final Region VENETO = new Region(1,"Veneto");
  public static final Region LOMBARDIA = new Region(2,"Lombardia");
  //... and so on

  public static final Region[] VALUES =
      new Region[] {MARCHE, VENETO, LOMBARDIA};

  private final int id;
  private final String name;

  private Region(int id, String name) {
    this.id = id;
    this.name = name;
  }

  public int getId() {
    return this.id;
  }

  public String getName() {
    return this.name;
  }

  public String toString() {
    return "Regione [id=" + id + ", name=" + name + "]";
```

UseRegion

```java
import java.util.Arrays;

public class UseRegion {

  public static void main(String[] args) {
    // nella variabile regione, si possono usare solo 3 casi
    Region region = Region.MARCHE;

    System.out.println(region);
    // si ottengono gli array dei valori possibile
    System.out.println(Arrays.toString(Region.VALUES));
    // è possibile accedere alla "prossima regione"
    System.out.println(Region.VALUES[region.getId()+1]);
  }
}
```

enum in Java

Un nuovo tipo di dato

- Simile ad una classe
- Realizza l'approccio a costanti e costruttore privato
- Ottime performance, l'oggetto è già disponibile
- Impedisce interamente errori di programmazione
- Il codice aggiuntivo da produrre non è elevato

Unica precauzione

- Andrebbero usate per insiemi di valori che difficilmente cambieranno in futuro
- Difficile modificare il codice successivamente

enum Region

```java
public enum Region {
  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<Region> list = new ArrayList<>();

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

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

Person con uso della enum

```
1  public class Person {
2    private final String name;
3    private final String surname;
4    private final Region region;
5
6    public Person(String name, String surname, Region region) {
7      this.name = name;
8      this.surname = surname;
9      this.region = region;
10   }
11
12   public boolean isIslander() {
13     return this.region == Region.SICILIA || this.region == Region.SARDEGNA;
14   }
15
16   public boolean fromRegion(Region region) {
17     return this.region == region;
18   }
19
20   public String getRegionName() {
21     return this.region.name();
22   }
23 }
```

UsePerson con uso della enum

```java
import java.util.*;
import static it.unibo.apice.oop.p13advanced.enums.enumeration.Region.*;

public class UsePerson {
  public static void main(String[] args){
    final List<Person> list = new ArrayList<>();
    list.add(new Person("Mario","Rossi",Region.EMILIA_ROMAGNA));
    list.add(new Person("Gino","Bianchi",Region.SICILIA));
    list.add(new Person("Carlo","Verdi",Region.LOMBARDIA));

    // Impossibile commettere errori (tranne null...)
    System.out.println(list.get(0).fromRegion(ABRUZZO));
    // uso dell'import static

    // Prestazioni OK!
    for (final Person p: list){
      if (p.isIslander()){
        System.out.println(p);
      }
    }
  }
}
```

Metodi di default per ogni enum

```java
import java.util.*;


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

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

  }
}
```

Metodi aggiuntivi nelle enum

```java
public enum Region {
  ABRUZZO("Abruzzo"),
  BASILICATA("Basilicata"),
  CALABRIA("Calabria"),
  CAMPANIA("Campania"),
  EMILIA_ROMAGNA("Emilia Romagna"),
  FRIULI_VENEZIA_GIULIA("Friuli Venezia Giula"),
  LAZIO("Lazio"),
  LIGURIA("Liguria"),
  LOMBARDIA("Lombardia"),
  MARCHE("Marche"),
  MOLISE("Molise"),
  PIEMONTE("Piemonte"),
  PUGLIA("Puglia"),
  SARDEGNA("Sardegna"),
  SICILIA("Sicilia"),
  TOSCANA("Toscana"),
  TRENTINO_ALTO_ADIGE("Trentino Alto Adige"),
  UMBRIA("Umbria"),
  VALLE_D_AOSTA("Valle D'Aosta"),
  VENETO("Veneto");

  private final String actualName;

  private Region(final String actualName){
    this.actualName = actualName;
  }

  public String getName(){
    return this.actualName;
  }

```

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..

`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
- . . . oppure quando il concetto dipende da quello della classe outer