15. Meccanismi Avanzati: Classi Innestate ed Enumerazioni

Questo capitolo approfondisce alcuni meccanismi avanzati del linguaggio Java: le **classi innestate** (nested classes) nelle loro varie forme e le **enumerazioni (enum)**. Questi costrutti permettono di organizzare il codice in modo più granulare, migliorare l'incapsulamento e definire tipi di dati con un numero fisso e limitato di valori.

9.1. Classi Innestate (Nested Classes)

Le classi innestate sono **classi definite all'interno di un'altra classe**. Java offre diverse tipologie di classi innestate, ognuna con caratteristiche e casi d'uso specifici.

9.1.1. Classi Innestate Statiche (`static nested classes`)

Una classe innestata statica è una classe definita all'interno di un'altra classe (chiamata classe "outer") e dichiarata con il modificatore `static`.

- **Idea e Terminologia**:
  - `static class B { ... }` definita all'interno di `class A { ... }`.
  - `B` viene vista come una proprietà statica di `A`.
  - L'accesso a `B` avviene tramite la sintassi `A.B` (come un membro statico).
- **Possibilità di Innestamento**:
  - Anche un'interfaccia può fungere da classe "outer".
  - Si possono innestare anche interfacce.
  - Il nesting può essere multiplo e/o multilivello (es. `Outer.A.C`).
- **Accesso**:
  - L'accesso da fuori la classe `Outer` avviene con `Outer.StaticNested` (se `StaticNested` è pubblica).
  - Da dentro `Outer`, si può accedere direttamente con `StaticNested`.
  - Le classi esterna e interna si "vedono" a vicenda anche le proprietà `private`.
- **Motivazioni per l'uso delle classi innestate statiche**:
  1. **Evitare la proliferazione di classi**: Raggruppare classi strettamente correlate all'interno di una singola unità logica, specialmente quando solo la classe "outer" deve essere pubblica.
  2. **Migliorare l'incapsulamento**: Consentire un accesso locale (anche a proprietà private) tra la classe outer e la classe innestata, senza esporre la classe innestata all'esterno del package.
  3. **Migliorare la leggibilità**: Posizionare le definizioni delle classi dove sono logicamente più pertinenti, utilizzando nomi qualificati più espressivi (es. `Counter.Multi`).
- **Casi d'uso comuni**:
  - **Specializzazioni come classi innestate**: Quando una classe base ha alcune specializzazioni frequenti e ovvie che possono essere raggruppate al suo interno (es. `Counter.Multi`, `Counter.Bidirectional`).

```
public class Counter {
    private 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 { // Classe innestata statica
        public Multi(int initialValue) { super(initialValue); }
        public void multiIncrement(int n) { for (int i = 0; i < n; i++) { this.increment(); } }
```

```
    }
}
```

- **Necessità di una classe separata ai fini di ereditarietà**: Quando un sotto-comportamento deve implementare una data interfaccia o estendere una data classe, e questa classe non deve essere visibile dall'esterno (spesso `private static class`). Un esempio classico è un `Iterator` interno a una collezione.

```
public class Range implements Iterable<Integer> {
    private final int start;
    private final int stop;
    public Range(final int start, final int stop) { this.start = start; this.stop = stop; }

    @Override
    public java.util.Iterator<Integer> iterator() {
        // Restituisce un'istanza della classe innestata statica Iterator
        return new Iterator(this.start, this.stop);
    }

    // Classe innestata statica privata che implementa Iterator
    private static class Iterator implements java.util.Iterator<Integer> {
        private int current;
        private final int stop;
        public Iterator(final int start, final 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() { /* non implementato */ }
    }
}
// Uso: for (final int i : new Range(5, 12)) { System.out.println(i); }
```

- **Necessità di comporre una o più classi diverse**: Quando ogni classe realizza un sotto-comportamento e non è utilizzabile indipendentemente dalla classe "outer", spesso per suddividere lo stato dell'oggetto. Un esempio è `Map.Entry` all'interno dell'interfaccia `Map` nel Java Collections Framework.

9.1.2. Inner Class (Non Statiche)

Una "inner class" è una classe innestata **senza** il modificatore `static`.

- **Idea**: Una istanza di una inner class ha sempre un riferimento implicito a un'istanza della sua classe `Outer` (chiamata "enclosing instance" o istanza esterna). Questo riferimento è accessibile con la sintassi `Outer.this`.
- **Creazione**: Gli oggetti delle inner class sono creati con espressioni come `<obj-outer>.new <InnerClass>(<args>)`. La parte `<obj-outer>` è omettibile quando sarebbe `this`.
- **Accesso**: L'inner class può accedere a tutti i membri (anche privati) dell'istanza esterna.
- **Motivazioni**: Usate quando è necessario che ogni oggetto inner mantenga un riferimento all'oggetto outer. Pragmaticamente, sono usate quasi esclusivamente come `private`.

Esempio di `Inner Class`:

```
public class Outer {
    private int i; // Campo della classe Outer

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

    public Inner createInner() {
        return new Inner(); // Equivale a this.new Inner();
```

```
    }

    // Inner class (non statica)
    public class Inner {
        private int j = 0; // Campo della inner class

        public void update() {
            // La inner class accede al campo 'i' dell'istanza Outer.this
            this.j = this.j + Outer.this.i;
        }

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

public class UseOuter {
    public static void main(String[] args) {
        Outer o = new Outer(5);
        Outer.Inner in = o.new Inner(); // Creazione di Inner associata a 'o'
        System.out.println(in.getValue()); // Output: 0
        in.update(); // j = 0 + 5 = 5
        in.update(); // j = 5 + 5 = 10
        System.out.println(in.getValue()); // Output: 10

        Outer.Inner in2 = new Outer(10).createInner(); // Nuova istanza Outer, nuova Inner
        in2.update(); // j = 0 + 10 = 10
        in2.update(); // j = 10 + 10 = 20
        System.out.println(in2.getValue()); // Output: 20
    }
}
```

Un esempio classico è un iteratore che ha bisogno di accedere allo stato non statico della collezione che sta iterando (come `Range2` che usa `Range2.this.start` e `Range2.this.stop`).

9.1.3. Classi Locali (`local classes`)

Una classe locale è una classe definita all'interno di un metodo.

- **Idea**: È a tutti gli effetti una inner class (quindi ha un'istanza esterna implicita). In più, una classe locale "vede" anche le variabili nello scope del metodo in cui è definita, ma solo se queste variabili sono `final` o "di fatto final" (cioè, il loro valore non cambia dopo l'inizializzazione).
- **Motivazioni**: Usata quando una classe è necessaria solo all'interno di un singolo metodo e si vuole confinarla lì.
- **Uso Pratico**: Raramente usate direttamente in pratica, spesso sostituite dalle classi anonime o dalle lambda expressions.

9.1.4. Classi Anonime (`anonymous classes`)

Una classe anonima è una **classe locale senza nome**, definita e istanziata "al volo" in un'unica espressione.

- **Idea**: Si crea una classe locale senza indicarne il nome, e se ne crea subito un oggetto. Non possono contenere costruttori.
- **Caratteristiche**: Come le classi locali, hanno un'istanza esterna implicita e "vedono" le variabili `final` (o di fatto final) nello scope del metodo in cui sono definite.
- **Motivazioni**:
    1. **Oggetto singolo**: Se si deve creare un solo oggetto di quella classe, è inutile darle un nome.

2. **Evitare proliferazione di classi**: Non aggiungono un nuovo file `.java` al progetto.
3. **Implementazione "al volo"**: Tipicamente usate per implementare un'interfaccia o estendere una classe astratta in modo conciso.

Esempio di `Anonymous Class` (per `Iterator`):

```
public class Range4 implements Iterable<Integer> {
    private final int start;
    private final int stop;
    public Range4(final int start, final int stop) {
        this.start = start;
        this.stop = stop;
    }

    @Override
    public Iterator<Integer> iterator() {
        // Implementazione di Iterator tramite classe anonima
        return new java.util.Iterator<Integer>() {
            // Non ci può essere costruttore!
            private int current = start; // Accesso diretto a 'start' della classe esterna

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

            @Override
            public boolean hasNext() {
                return this.current <= stop; // Accesso diretto a 'stop' della classe esterna
            }

            @Override
            public void remove() { }
        }; // Questo è il punto e virgola del return!
    }
}
```

Altro esempio (con `Comparator`):

```
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class UseSort {
    public static void main(String[] args) {
        final List<Integer> list = Arrays.asList(10, 40, 7, 57, 13, 19, 21, 35);
        System.out.println(list);

        // Ordinamento crescente usando una classe anonima per Comparator
        Collections.sort(list, new Comparator<Integer>() {
            @Override
            public int compare(Integer a, Integer b) {
                return Integer.compare(a, b);
            }
        });
        System.out.println(list); // Output: [7, 10, 13, 19, 21, 35, 40, 57]
```

```
        // Ordinamento decrescente usando un'altra classe anonima
        Collections.sort(list, new Comparator<Integer>() {
            @Override
            public int compare(Integer a, Integer b) {
                return Integer.compare(b, a); // Inverte l'ordine
            }
        });
        System.out.println(list); // Output: [57, 40, 35, 21, 19, 13, 10, 7]
    }
}
```

- *Provenienza*: 19-advanced-mechanisms-nesting_slides.pdf, Pagina 42

9.1.5. Riassunto e Linee Guida per le Classi Innestate

La scelta del tipo di classe innestata dipende dalla visibilità e dall'accesso ai membri della classe esterna che sono necessari:

Tipo di Classe Innestata	Visibilità	Accesso all'istanza esterna	Accesso a variabili locali del metodo	Motivazione/Uso Tipico
Static Nested Class	Dipende dal modificatore (`public` , `private` , etc.)	**No** (solo membri statici della outer)	**No**	Raggruppare classi strettamente correlate, senza dipendenza da istanza outer.
Inner Class	Dipende dal modificatore	**Sì** (`Outer.this`)	**No**	Quando la inner class ha bisogno di accedere ai membri non statici dell'istanza outer.
Local Class	Confinata al metodo	**Sì** (`Outer.this`)	**Sì** (se `final` o "di fatto final")	Quando la classe è necessaria solo all'interno di un metodo specifico.
Anonymous Class	Confinata al metodo	**Sì** (`Outer.this`)	**Sì** (se `final` o "di fatto final")	Per implementare "al volo" un'interfaccia o estendere una classe astratta, creando un singolo oggetto.

- *Provenienza*: 19-advanced-mechanisms-nesting_slides.pdf, Pagina 41 (tabella riassuntiva adattata e integrata).

9.2. Enumerazioni (Enums)

Le enumerazioni (o `enum`) in Java sono un tipo speciale di classe che consente di definire un insieme fisso e limitato di valori costanti.

9.2.1. Motivazioni e Problemi delle Soluzioni Precedenti

In situazioni in cui un tipo di dato può assumere solo un numero predefinito di valori (es. giorni della settimana, regioni d'Italia, pezzi degli scacchi), le soluzioni precedenti a Java 5 presentavano diversi problemi:

- **Uso di Stringhe**:
  - **Problemi**: Lento (confronti di stringhe), soggetto a errori di digitazione (`"Emilia-Romagna"` vs `"Emilia Romagna"`), difficile da intercettare a compile-time.
- **Uso di Interi (`int`)**:
  - **Problemi**: Scarsa leggibilità (cosa significa `regione = 1` ?), non impedisce l'uso di interi non validi (`regione = 99`), le costanti possono essere dispersive
- **Uso di Classi Astratte/Concrete per Valore**:
  - **Problemi**: Molto prolisso (troppo codice per ogni valore), impraticabile con un numero elevato di valori.

9.2.2. Introduzione alle `enum` in Java

Le `enum` in Java (introdotte in Java 5) offrono una soluzione robusta ed elegante a questi problemi.

- **Concetto**: Consentono di elencare i valori, associando a ognuno un nome. Internamente, ogni valore `enum` è un'istanza di una classe speciale che estende `java.lang.Enum` .

- **Vantaggi**:
  - **Sicurezza del tipo**: Impediscono errori di programmazione (non si possono usare valori non definiti).
  - **Leggibilità**: I nomi dei valori sono auto-esplicativi (es. `Regione.LOMBARDIA`).
  - **Performance**: Gli oggetti `enum` sono già disponibili (sono costanti statiche `final`), quindi non c'è overhead di creazione. I confronti avvengono per riferimento (`==`), che è molto veloce.
  - **Flessibilità**: È possibile collegare metodi e campi a ogni "valore" dell'enum.
- **Precauzione**: Andrebbero usate per insiemi di valori che difficilmente cambieranno in futuro, poiché modificarle successivamente può essere più complesso.

Esempio Base di `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;
}

public class UseEnum {
    public static void main(String[] args) {
        final java.util.List<Regione> list = new java.util.ArrayList<>();
        list.add(Regione.LOMBARDIA);
        list.add(Regione.PIEMONTE);
        list.add(Regione.EMILIA_ROMAGNA);
        for (final Regione r : list) {
            System.out.println(r.toString()); // Stampa il nome della regione
        }
    }
}
```

9.2.3. `enum` con Campi, Costruttori e Metodi

Le enumerazioni possono essere molto più potenti di semplici liste di costanti. Possono avere campi, un costruttore (sempre implicito `private`) e metodi, proprio come una classe normale.

Esempio: `Regione` con `Zona` e nome esteso

```
// Definizione dell'enum Zona
public enum Zona {
    NORD, CENTRO, SUD;

    // Metodo per ottenere le regioni di una data zona
    public List<Regione> getRegioni() {
        final ArrayList<Regione> list = new ArrayList<>();
        for (final Regione r : Regione.values()) {
            if (r.getZona() == this) { // Confronto per riferimento (veloce)
                list.add(r);
            }
        }
        return list;
    }
}

// Definizione dell'enum Regione con campi e costruttore
public enum Regione {
    ABRUZZO(Zona.CENTRO, "Abruzzo"),
```

```
    BASILICATA(Zona.SUD, "Basilicata"),
    // ... altre regioni
    EMILIA_ROMAGNA(Zona.NORD, "Emilia Romagna"),
    // ...
    VENETO(Zona.NORD, "Veneto");

    private final Zona z; // Campo per la zona geografica
    private final String actualName; // Campo per il nome esteso

    // Costruttore (implicito private)
    private Regione(final Zona z, final String actualName) {
        this.z = z;
        this.actualName = actualName;
    }

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

    @Override
    public String toString() {
        return this.actualName; // Sovrascrive toString per un output più leggibile
    }
}

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

9.2.4. Metodi di Default per ogni `enum`

Ogni `enum` in Java eredita automaticamente alcuni metodi dalla classe `java.lang.Enum`:

- `String name()`: Restituisce il nome esatto della costante enum (es. `EMILIA_ROMAGNA`).
- `int ordinal()`: Restituisce la posizione ordinale della costante enum (partendo da 0).
- `static E valueOf(String name)`: Restituisce la costante enum con il nome specificato.
- `static E[] values()`: Restituisce un array contenente tutte le costanti enum in ordine di dichiarazione.
- `String toString()`: Di default restituisce `name()`, ma può essere sovrascritto (come nell'esempio `Regione`).
- *Provenienza*: 20-advanced-mechanisms-enum_slides.pdf, Pagina 19-20

9.2.5. `enum` negli `switch` Statement

Le enumerazioni possono essere utilizzate direttamente negli statement `switch`, migliorando la leggibilità e la sicurezza del codice.

```
import static it.unibo.advancedmechanisms.enums.en2.Regione.*; // Esempio di import static
import java.util.*;

public class UseRegione2 {
    public static void main(String[] args) {
        final ArrayList<Regione> list = new ArrayList<>();
```

```
        list.add(LOMBARDIA); // Usando import static
        list.add(SARDEGNA);
        list.add(Regione.valueOf("SICILIA")); // Ottenere un enum da stringa
        list.add(Regione.values()[10]); // Ottenere un enum per ordinale (sconsigliato per leggibilità)

        for (final Regione r : list) {
            switch (r) { // Uso dell'enum nello switch
                case LOMBARDIA:
                    System.out.println("Sei in Lombardia!");
                    break;
                case EMILIA_ROMAGNA:
                    System.out.println("Sei in Emilia Romagna!");
                    break;
                case SICILIA:
                case SARDEGNA:
                    System.out.println("Sei in un'isola!");
                    break;
                default:
                    System.out.println("Altra regione.");
            }
        }
    }
}
```

9.2.6. `enum` Innestate

Anche le `enum` possono essere innestate all'interno di classi, interfacce o altre `enum` (e sono implicitamente statiche). Questo è utile quando il loro uso è strettamente confinato al funzionamento della classe "outer".

Esempio: L'enum `Regione` potrebbe essere innestata dentro una classe `Persona`, o l'enum `Zona` dentro `Regione.`