4. Incapsulamento, Interfacce ed Ereditarietà

Questo capitolo approfondisce i principi chiave della programmazione orientata agli oggetti che consentono di costruire sistemi software robusti, flessibili e manutenibili: l'incapsulamento, le interfacce per la definizione di contratti e l'ereditarietà per il riuso e la specializzazione del codice.

4.1 Incapsulamento e Information Hiding

L'incapsulamento è uno dei pilastri della programmazione orientata agli oggetti. Non è solo un meccanismo sintattico, ma un principio di buona progettazione che mira a migliorare la robustezza e la manutenibilità del software.

4.1.1 Principi di Buona Progettazione

L'analisi dell'OOP in Java finora ha mostrato i meccanismi base (tipi primitivi, operatori, cicli, classi, oggetti, costruttori, campi, metodi, codice statico, controllo d'accesso). Tuttavia, per realizzare un **buon sistema software** che sia facilmente manutenibile (estendibile, flessibile, leggibile) e che soddisfi i requisiti, è necessario applicare principi di buona progettazione

4.1.2 Incapsulamento in Java

L'incapsulamento in Java si riferisce al **raggruppamento di dati (campi) e dei metodi** che operano su quei dati all'interno di una singola unità, la **classe**. Il suo obiettivo principale è limitare il più possibile le dipendenze con chi usa la classe, riducendo l'impatto delle modifiche che si rendono via via necessarie

Questo si ottiene attraverso il principio di **Information Hiding** (occultamento delle informazioni), che consiste nel nascondere i dettagli interni di implementazione di una classe e esporre solo un'interfaccia pubblica ben definita

- **Vantaggi dell'Incapsulamento**:
  - **Protezione dei dati**: Impedisce accessi diretti e non controllati ai dati interni di un oggetto, prevenendo stati inconsistenti.
  - **Manutenibilità**: Le modifiche all'implementazione interna di una classe non influenzano il codice esterno che la utilizza, purché l'interfaccia pubblica rimanga invariata.
  - **Modularità**: Promuove la creazione di moduli indipendenti e riutilizzabili.
  - **Flessibilità**: Permette di cambiare l'implementazione interna senza dover modificare il codice client.

4.1.3 Controllo d'Accesso e Incapsulamento

Come visto nel Capitolo 3, i modificatori di accesso (`public`, `protected`, `default`, `private`) sono gli strumenti sintattici in Java per implementare l'incapsulamento.

La metodologia standard di progettazione per l'incapsulamento prevede:

- **Campi `private`**: Tutti i campi di istanza di una classe dovrebbero essere dichiarati `private`. Questo significa che possono essere acceduti o modificati solo dai metodi all'interno della stessa classe
- **Metodi `public` (Getter e Setter)**: Se è necessario che il codice esterno acceda o modifichi lo stato di un oggetto, si forniscono metodi `public` specifici:
  - **Getter (Accessors)**: Metodi che restituiscono il valore di un campo privato (es. `public double getIntensity()`)
  - **Setter (Mutators)**: Metodi che impostano il valore di un campo privato. Possono includere logica di validazione per assicurare che il valore sia valido (es. `public void setIntensity(double intensity)`)

```
class Lamp {
   private double intensity; // Campo privato, non accessibile direttamente dall'esterno
   private boolean switchedOn; // Campo privato

   public void switchOn() { // Metodo pubblico per accendere la lampada
      this.switchedOn = true;
   }

   public void switchOff() { // Metodo pubblico per spegnere la lampada
```

```
    this.switchedOn = false;
}

public boolean isSwitchedOn() { // Getter pubblico per lo stato di accensione
    return this.switchedOn;
}

public void setIntensity(double intensity) { // Setter pubblico con validazione
    if (intensity >= 0.0 && intensity <= 1.0) {
        this.intensity = intensity;
    } else {
        System.out.println("Intensità non valida. Deve essere tra 0.0 e 1.0.");
    }
}

public double getIntensity() { // Getter pubblico per l'intensità
    return this.intensity;
}
}
```

Questo esempio mostra come lo stato interno (`intensity`, `switchedOn`) sia protetto, e l'interazione con l'oggetto `Lamp` avvenga solo tramite i suoi metodi pubblici.

4.1.4 Il Metodo `toString()`

Una convenzione importante in Java è che ogni classe dovrebbe definire un metodo `toString()`

- **Scopo**: Questo metodo dovrebbe restituire una rappresentazione in stringa dell'oggetto, utile per il debugging, il logging o la visualizzazione all'utente.
- **Incapsulamento della Presentazione**: Definendo `toString()`, si incapsula anche la funzionalità di presentazione dell'oggetto.
- **Uso automatico**: Il metodo `toString()` viene automaticamente chiamato quando si usa l'operatore `+` per concatenare stringhe a oggetti, o quando un oggetto viene passato a `System.out.println()`.

```
public class LampString {
    private double intensity;
    private boolean switchedOn;

    // ... costruttori e altri metodi ...

    @Override // Annotazione che indica l'override di un metodo della superclasse
    public String toString() {
        return "Lampada [Accesa: " + switchedOn + ", Intensità: " + intensity + "]";
    }

    public static void main(String[] s) {
        LampString l = new LampString();
        l.switchOn();
        l.setIntensity(0.5);
        // ... altre operazioni ...
        System.out.println(l.toString()); // Chiamata esplicita
        System.out.println("Stato della lampada: " + l); // Chiamata implicita di toString()
    }
}
```

4.2 Interfacce e Polimorfismo

Le interfacce sono un concetto fondamentale in Java per definire contratti e abilitare il polimorfismo, separando la definizione di un comportamento dalla sua implementazione.

4.2.1 Tipi di Composizione e Riuso

L'incapsulamento fornisce i meccanismi per progettare bene le classi, limitando le dipendenze. Tuttavia, le dipendenze tra classi sono inevitabili e anzi, sono un prerequisito per fare di un gruppo di classi un sistema. Le dipendenze sono anche una manifestazione di un effettivo "riuso"

Esistono due tipi principali di composizione:

1. **Composizione "usa un" (Use-a)**:
    - Una classe utilizza i servizi di un'altra classe.
    - Si realizza tipicamente attraverso l'invocazione di metodi su un oggetto di un'altra classe.
    - *Esempio*: Una classe `Car` "usa un" oggetto `Engine` chiamando il suo metodo `start()`.
2. **Composizione "ha un" (Has-a)**:
    - Una classe contiene un oggetto di un'altra classe come suo campo.
    - Questo è un modo per costruire oggetti complessi a partire da oggetti più semplici.
    - *Esempio*: Una classe `Car` "ha un" `Engine` (il campo `engine` nella classe `Car`).

4.2.2 Notazione UML (Unified Modeling Language)

UML è un linguaggio standardizzato per la modellazione visuale dei sistemi software. Aiuta a visualizzare le **relazioni tra classi e oggetti.**

- **Associazione**: Rappresenta una relazione generica tra due classi
- **Aggregazione**: Un tipo speciale di associazione "ha un" dove una classe è un "tutto" e l'altra è una "parte", ma la parte può esistere indipendentemente dal tutto
- **Composizione (UML)**: Un tipo più forte di aggregazione, dove la "parte" non può esistere senza il "tutto". Se il "tutto" viene distrutto, anche la "parte" viene distrutta

4.2.3 Interfacce in Java

Un'**interfaccia** in Java è un contratto che definisce un **insieme di metodi** che una classe deve implementare. Un'interfaccia non contiene l'implementazione di questi metodi, ma solo la loro **firma** (nome, parametri, tipo di ritorno)

- **Dichiarazione**: Si usa la parola chiave `interface`.
- **Membri**:
    - Fino a Java 8, le interfacce potevano contenere solo:
        - **Costanti pubbliche statiche e finali** (implicitamente `public static final`).
        - **Metodi astratti pubblici** (implicitamente `public abstract`).
    - Da Java 8 in poi, le interfacce possono anche contenere:
        - **Metodi** `default`: Metodi con un'implementazione di default.
        - **Metodi** `static`: Metodi statici con implementazione.
    - Da Java 9, possono contenere anche **metodi** `private` e **metodi** `private static` (per logica interna all'interfaccia).
- **Implementazione**: Una classe che vuole aderire a un'interfaccia usa la parola chiave `implements` e deve fornire un'implementazione per tutti i metodi astratti dell'interfaccia. Una classe può implementare più interfacce

```
interface Switchable {
    void switchOn();
    void switchOff();
    boolean isSwitchedOn();
}
```

```
class Lamp implements Switchable {
    private boolean switchedOn;

    @Override
    public void switchOn() {
        this.switchedOn = true;
        System.out.println("Lampada accesa.");
    }

    @Override
    public void switchOff() {
        this.switchedOn = false;
        System.out.println("Lampada spenta.");
    }

    @Override
    public boolean isSwitchedOn() {
        return this.switchedOn;
    }
}
```

4.2.4 Polimorfismo con le Interfacce

Il polimorfismo ("molte forme") è la capacità di un **oggetto** di assumere **diverse forme**. Con le interfacce, il polimorfismo si manifesta nella capacità di **trattare oggetti di classi diverse in modo uniforme, purché implementino la stessa interfaccia**

- **Principio di Sostituibilità di Liskov**: Un sottotipo deve essere sostituibile per il suo tipo base senza alterare la correttezza del programma. Le interfacce garantiscono questo principio.
- **Vantaggi**:
    - **Flessibilità**: Il codice può operare su un'interfaccia, senza conoscere i dettagli della classe concreta che la implementa.
    - **Estendibilità**: Nuove classi possono implementare l'interfaccia e essere utilizzate dal codice esistente senza modifiche.
    - **Decoupling**: Riduce l'accoppiamento tra le classi, rendendo il sistema più modulare.

```
public class UseSwitchable {
    public static void main(String[] args) {
        Switchable device1 = new Lamp(); // Un oggetto Lamp trattato come Switchable
        device1.switchOn();

        // Potrei avere un'altra classe che implementa Switchable, ad esempio un Fan
        // Switchable device2 = new Fan();
        // device2.switchOn();

        // Posso creare un array di Switchable e iterare su di essi
        Switchable[] devices = new Switchable[2];
        devices[0] = new Lamp();
        // devices[1] = new Fan(); // Se avessimo la classe Fan

        for (Switchable device : devices) {
            if (device != null) {
                device.switchOn(); // Chiamo switchOn() su diversi tipi di oggetti
            }
        }
```

```
    }
}
```

4.2.5 Esempi di Interfacce nelle Librerie Java

Le librerie standard di Java sono ricche di interfacce che definiscono contratti comuni, come:

- `java.lang.CharSequence`: Contratto per oggetti che rappresentano "sequenze di caratteri" leggibili (implementata da `String`, `StringBuffer`).
- `java.lang.Appendable`: Contratto per oggetti su cui "appendere" (aggiungere in coda) sequenze di caratteri (implementata da `StringBuffer`).
- `java.io.Serializable`: Un'interfaccia "tag" (vuota, senza metodi) che indica che un oggetto può essere serializzato (convertito in un flusso di byte per essere memorizzato o trasmesso).

4.3 Ereditarietà

L'ereditarietà è un meccanismo di **riuso del codice** che consente di definire una nuova classe specializzandone una esistente.

4.3.1 Meccanismo dell'Ereditarietà (`extends`)

L'ereditarietà permette a una classe (la **sottoclasse** o **classe derivata**) di "ereditare" i campi e i metodi da un'altra classe (la **superclasse** o **classe base**). Questo promuove il riuso di codice già scritto e testato.

- **Relazione "È UN" (Is-A)**: L'ereditarietà modella una relazione "è un tipo di". Ad esempio, un `Cane` "è un tipo di" `Animale`.
- **Parola chiave `extends`**: In Java, si usa la parola chiave `extends` per indicare che una classe eredita da un'altra.
- **Membri ereditati**: La sottoclasse eredita tutti i campi e i metodi della superclasse. I membri `private` sono ereditati ma non sono direttamente visibili o accessibili dalla sottoclasse (da `09-inheritance_slides.pdf`, Pag. 4).

Esempio (da `09-inheritance_slides.pdf`, Pag. 6):

```
class Animal {
    private String name;

    public Animal(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void eat() {
        System.out.println(name + " sta mangiando.");
    }
}

class Dog extends Animal { // Dog eredita da Animal
    public Dog(String name) {
        super(name); // Chiama il costruttore della superclasse
    }

    public void bark() {
        System.out.println(getName() + " sta abbaiando: Bau bau!");
    }
}
```

4.3.2 Livello d'Accesso `protected`

Il modificatore `protected` (introdotto nel Capitolo 3) è particolarmente rilevante nell'ereditarietà.

- I membri `protected` sono accessibili all'interno dello stesso package e da tutte le sottoclassi, anche se si trovano in package diversi.
- Questo permette alle sottoclassi di accedere a membri della superclasse che non sono `public`, ma che sono comunque parte della sua API interna pensata per l'estensione.

4.3.3 Overriding dei Metodi

L'**overriding** (o sovrascrittura) è la capacità di una **sottoclasse** di fornire una propria **implementazione** per un metodo che è già definito nella sua superclasse

- **Regola**: La firma del metodo (nome, numero e tipo di parametri) deve essere esattamente la stessa del metodo nella superclasse. Il tipo di ritorno può essere lo stesso o un sottotipo (covarianza del tipo di ritorno).
- **Annotazione** `@Override`: È buona prassi annotare i metodi sovrascritti con `@Override`. Questo non è obbligatorio, ma il compilatore genererà un errore se il metodo non fa effettivamente override, migliorando la leggibilità e prevenendo errori
- **Parola chiave** `super`: All'interno di un metodo sovrascritto, è possibile invocare l'implementazione del metodo della superclasse usando la parola chiave `super` (es. `super.metodo()`)

```
class Shape {
    public void draw() {
        System.out.println("Disegno una forma generica.");
    }
}

class Circle extends Shape {
    @Override // Indica che questo metodo sovrascrive un metodo della superclasse
    public void draw() {
        System.out.println("Disegno un cerchio.");
    }
}
```

4.3.4 Gestione dei Costruttori e Chiamate `super()`

Quando si crea un'istanza di una sottoclasse, viene sempre invocato prima il costruttore della superclasse.

- **Chiamata** `super()`: La prima istruzione di un costruttore di una sottoclasse deve essere una chiamata esplicita al costruttore della superclasse usando `super()`. Se non si effettua una chiamata esplicita, Java inserisce automaticamente una chiamata implicita a `super()` senza argomenti.
- **Motivazione**: Questo garantisce che la parte della superclasse dell'oggetto venga inizializzata correttamente prima che la sottoclasse aggiunga la propria inizializzazione.

Esempio:

```
class Person {
    private String name;

    public Person(String name) {
        this.name = name;
        System.out.println("Costruttore Person: " + name);
    }
}

class Student extends Person {
    private int studentId;

    public Student(String name, int studentId) {
        super(name); // Deve essere la prima istruzione, chiama il costruttore di Person
        this.studentId = studentId;
        System.out.println("Costruttore Student: " + name + ", ID: " + studentId);
```

```
    }
}
```

4.3.5 Il Modificatore `final` su Classi e Metodi

La parola chiave `final` ha significati diversi a seconda del contesto:

- `final` **su un campo**: Il campo diventa una **costante** e il suo valore non può essere modificato dopo l'inizializzazione.
- `final` **su un metodo**: Il metodo **non può essere sovrascritto** da nessuna sottoclasse. Questo è utile per garantire che un comportamento critico non venga alterato.
- `final` **su una classe**: La classe **non può essere estesa**, cioè non può avere sottoclassi. Questo si usa per classi che non sono progettate per l'ereditarietà (es. `String` in Java).

4.3.6 La Classe `Object`

In Java, **tutte le classi ereditano implicitamente dalla classe** `java.lang.Object`. Ciò significa che ogni oggetto in Java ha accesso ai metodi definiti in `Object`.

Alcuni metodi importanti della classe `Object` che spesso vengono sovrascritti includono

- `toString()`: Restituisce una rappresentazione in stringa dell'oggetto (già visto).
- `equals(Object obj)`: Utilizzato per confrontare due oggetti per l'uguaglianza semantica. L'implementazione di default confronta i riferimenti (come `==`), che spesso non è il comportamento desiderato per l'uguaglianza di contenuto.
- `hashCode()`: Restituisce un codice hash per l'oggetto, utilizzato principalmente nelle collezioni basate su hash (es. `HashMap`, `HashSet`). Se si sovrascrive `equals()`, è **obbligatorio** sovrascrivere anche `hashCode()` per mantenere il contratto generale di `Object`.
- `clone()`: Per creare una copia di un oggetto.
- `notify()`, `wait()`: Usati nella gestione dei thread per la sincronizzazione.