Appunti di Programmazione Java

Prof. Luca Campion

14 novembre 2025

Indice

Elenco delle figure

Listings

Capitolo 1

Classi, Oggetti, Ereditarietà e Package

1.1 Obiettivi di apprendimento

Al termine di questo capitolo sarai in grado di:

- Comprendere i concetti fondamentali della programmazione orientata agli oggetti e le loro applicazioni in Java.
- Dichiarare classi con attributi, costruttori e metodi.
- Utilizzare i diagrammi UML per rappresentare classi e le loro relazioni.
- Creare e manipolare array di oggetti.
- Implementare metodi statici e di utilità come `toString()`, `getter` e `setter`.
- Applicare l'ereditarietà per estendere classi esistenti e riusare il codice.
- Sfruttare il polimorfismo per gestire collezioni eterogenee e scrivere codice flessibile.
- Organizzare il codice in package per migliorare la modularità e la manutenibilità.

1.2 Introduzione alla Programmazione Orientata agli Oggetti (OOP) e a Java

La Programmazione Orientata agli Oggetti (OOP) è un paradigma di programmazione che ha rivoluzionato il modo in cui sviluppiamo software, fornendo un approccio più intuitivo e potente per modellare problemi complessi del mondo reale. In Java, l'OOP è al centro di quasi ogni aspetto del linguaggio.

1.2.1 Caratteristiche Fondamentali della OOP

L'OOP si basa su quattro pilastri principali:

1. **Astrazione**: La capacità di concentrarsi sugli aspetti essenziali di un oggetto, ignorando i dettagli irrilevanti. Permette di creare modelli semplificati di entità complesse. Ad

esempio, quando guidiamo un'auto, ci preoccupiamo del volante e dei pedali, non del funzionamento interno del motore.

2. **Incapsulamento**: Il meccanismo che lega insieme i dati (attributi) e i metodi che operano su quei dati, all'interno di una singola unità chiamata classe. Esso nasconde i dettagli implementativi e protegge i dati da accessi esterni non autorizzati. I modificatori di accesso (come `private`, `public`, `protected`) sono fondamentali per l'incapsulamento.

3. **Ereditarietà**: Permette a una nuova classe (sottoclasse) di acquisire le proprietà e i comportamenti di una classe esistente (superclasse). Favorisce il riuso del codice e stabilisce una relazione "è un" (is-a) tra le classi (es. "un'Auto è un Veicolo").

4. **Polimorfismo**: Significa "molte forme". Permette a oggetti di classi diverse, ma correlate tramite ereditarietà, di rispondere allo stesso messaggio (invocazione di metodo) in modi diversi, ciascuno secondo la propria implementazione. Questo rende il codice più generico e flessibile.

1.2.2 Perché Java è un Linguaggio Orientato agli Oggetti?

Java è stato progettato fin dall'inizio come un linguaggio orientato agli oggetti. Ogni cosa in Java (tranne i tipi primitivi) è un oggetto e ogni codice (tranne i costrutti speciali come i blocchi statici) risiede all'interno di una classe. Questa forte adesione all'OOP offre numerosi vantaggi:

- **Modularità**: Il codice è organizzato in moduli (classi) ben definiti.
- **Riusabilità**: Classi e metodi possono essere riutilizzati in diverse parti dell'applicazione o in altri progetti.
- **Manutenibilità**: Il codice è più facile da capire, debuggare e modificare.
- **Scalabilità**: Progetti complessi possono essere gestiti più efficacemente.
- **Sicurezza**: L'incapsulamento aiuta a proteggere i dati.

1.3 Classi, Oggetti e Attributi

Al cuore dell'OOP ci sono i concetti di classe e oggetto.

1.3.1 Classe: il Progetto

Una **classe** è un modello, un progetto o un prototipo che definisce le caratteristiche (attributi) e i comportamenti (metodi) di un tipo di oggetto. Non è un'entità fisica, ma una descrizione.

```
1 public class Persona {
2     // Attributi (variabili di istanza) - Rappresentano lo stato dell'
          oggetto
3     private String nome;
4     private String cognome;
5     private int eta;
6     private String codiceFiscale;
7 }
```

Nell'esempio sopra, `Persona` è una classe. `nome`, `cognome`, `eta` e `codiceFiscale` sono i suoi **attributi**. Il modificatore `private` implementa il principio di incapsulamento, rendendo questi attributi accessibili solo all'interno della classe stessa.

1.3.2 Oggetto: l'Istanza Concreta

Un **oggetto** è un'istanza concreta di una classe. Ogni oggetto ha il proprio insieme di attributi e può eseguire i metodi definiti nella sua classe. Gli oggetti vengono creati in memoria (heap) utilizzando l'operatore `new`.

```
public class TestOggetti {
    public static void main(String[] args) {
        // Dichiarazione di una variabile di riferimento 'p1' di tipo
            Persona
        Persona p1;

        // Creazione di un oggetto (istanza) della classe Persona
        // 'new Persona()' invoca il costruttore di default
        p1 = new Persona();

        // Ora p1 "contiene" un riferimento all'oggetto Persona in
            memoria
        // Possiamo anche fare tutto in una riga:
        Persona p2 = new Persona();
    }
}
```

In questo esempio, `p1` e `p2` sono riferimenti a due oggetti distinti della classe `Persona`. Ogni oggetto `Persona` avrà i propri valori per `nome`, `cognome`, `eta` e `codiceFiscale`.

1.4 Costruttori e Metodi

Le classi non sono complete senza costruttori per inizializzare gli oggetti e metodi per definirne il comportamento.

1.4.1 Costruttori: L'Inizializzazione degli Oggetti

Un **costruttore** è un metodo speciale che viene invocato automaticamente quando si crea un nuovo oggetto di una classe usando l'operatore `new`. Il suo scopo principale è inizializzare lo stato dell'oggetto.

- Deve avere lo **stesso nome** della classe.
- Non ha un **tipo di ritorno** (nemmeno `void`).
- Se non viene definito alcun costruttore, Java ne fornisce uno **di default** (senza parametri).

Costruttore con Parametri

Permette di inizializzare gli attributi con valori specifici al momento della creazione dell'oggetto.

```java
public class Persona {
    private String nome;
    private String cognome;
    private int eta;

    // Costruttore con parametri
    public Persona(String nome, String cognome, int eta) {
        this.nome = nome;         // 'this.nome' si riferisce all'
            attributo della classe
        this.cognome = cognome;  // 'nome' si riferisce al parametro
            del costruttore
        this.eta = eta;
    }
}
```

`this`: La parola chiave `this` è un riferimento all'istanza corrente dell'oggetto. È usata per distinguere gli attributi di istanza dai parametri locali che hanno lo stesso nome.

Costruttori di Default e Completo

I costruttori permettono di inizializzare gli oggetti della classe. Il **costruttore di default** fornisce valori predefiniti quando non vengono specificate informazioni, mentre il **costruttore completo** permette di impostare tutti i parametri della classe al momento della creazione dell'oggetto.

```java
public class Persona {
    private String nome;
    private String cognome;
    private int eta;

    // Costruttore completo
    public Persona(String nome, String cognome, int eta) {
        this.nome = nome;
        this.cognome = cognome;
        this.eta = eta;
    }

    // Costruttore di default che richiama quello completo
    public Persona() {
        this("Sconosciuto", "Sconosciuto", 0);
    }
}
```

1.4.2 Metodi: Il Comportamento degli Oggetti

I **metodi** sono blocchi di codice che definiscono il comportamento o le azioni che un oggetto può compiere. Possono accedere e modificare lo stato degli attributi dell'oggetto.

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

```
    private int eta;

    // Metodo per visualizzare le informazioni
    public void stampaInfo() {
        System.out.println(nome + " " + cognome + ", " + eta + " anni");
    }

    // Metodo per verificare se la persona è maggiorenne
    public boolean isMaggiorenne() {
        return eta >= 18;
    }
}
```

Esempio di Utilizzo della Classe Persona:

```
public class TestPersona {
    public static void main(String[] args) {
        // Creazione di oggetti usando diversi costruttori
        Persona p1 = new Persona("Mario", "Rossi", 25);
        Persona p2 = new Persona(); // Usa costruttore di default
        Persona p3 = new Persona("Laura", "Bianchi", 17);

        // Invocazione di metodi sugli oggetti
        p1.stampaInfo(); // Output: Mario Rossi, 25 anni

        if (p3.isMaggiorenne()) {
            // System.out.println(p3.nome + " è maggiorenne."); // Errore: nome è private
            System.out.println("Laura è maggiorenne."); // Versione corretta
        } else {
            // System.out.println(p3.cognome + " è minorenne."); // Errore: cognome è private
            System.out.println("Laura è minorenne."); // Versione corretta
        }
        // Nota: Qui si incorrerebbe in un errore di compilazione
        // perché nome e cognome sono privati. Vedremo come accedere
        // ad essi tramite getter e setter.
    }
}
```

1.4.3 Overloading dei Metodi (Sovraccarico)

L'**overloading** permette di definire più metodi con lo stesso nome all'interno della stessa classe, purché abbiano una **firma diversa**. La firma di un metodo è determinata dal suo nome e dalla lista dei tipi dei parametri, e dal loro ordine. Il tipo di ritorno non è parte della firma per l'overloading.

Regole dell'Overloading

- Stesso nome del metodo.

- Lista di parametri diversa (numero, tipo o ordine dei tipi).
- Il tipo di ritorno può essere lo stesso o diverso (non influisce sull'overloading).

Esempio: Overloading di Costruttori nella Classe Persona

Abbiamo già visto questo nella classe `Persona` con il costruttore con parametri e quello di default. Possiamo aggiungere un terzo costruttore:

```
public class Persona {
    private String nome;
    private String cognome;
    private int eta;

    // 1. Costruttore con 3 parametri
    public Persona(String nome, String cognome, int eta) {
        this.nome = nome;
        this.cognome = cognome;
        this.eta = eta;
    }

    // 2. Costruttore senza parametri (overloading)
    public Persona() {
        this("Sconosciuto", "Sconosciuto", 0); // Chiama il
            costruttore #1
    }

    // 3. Costruttore con 2 parametri (overloading)
    public Persona(String nome, String cognome) {
        this(nome, cognome, 0); // Chiama il costruttore #1,
            impostando età a 0
    }
}
```

`this()`: Un costruttore può chiamare un altro costruttore della stessa classe usando `this()`. Deve essere la prima istruzione del costruttore chiamante.

Esempio: Overloading di un Metodo `calcola`

```
public class Calcolatrice {

    // Somma due interi
    public int calcola(int a, int b) {
        return a + b;
    }

    // Somma tre interi (overloading: numero di parametri diverso)
    public int calcola(int a, int b, int c) {
        return a + b + c;
    }

    // Somma due double (overloading: tipo di parametri diverso)
    public double calcola(double a, double b) {
        return a + b;
```

```
16      }
17
18      // Esempio di utilizzo
19      public static void main(String[] args) {
20          Calcolatrice calc = new Calcolatrice();
21
22          System.out.println(calc.calcola(5, 3));          // Chiama
                calcola(int, int)
23          System.out.println(calc.calcola(5, 3, 2));       // Chiama
                calcola(int, int, int)
24          System.out.println(calc.calcola(5.5, 3.2));      // Chiama
                calcola(double, double)
25      }
26  }
```

1.5 UML: Rappresentare Classi e Relazioni

I diagrammi UML (Unified Modeling Language) sono uno strumento standard per visualizzare, specificare, costruire e documentare gli artefatti di un sistema software. I diagrammi di classe UML sono particolarmente utili per mostrare le classi, i loro attributi, i loro metodi e le relazioni tra di esse.

1.5.1 Sintassi Base del Diagramma di Classe UML

Un rettangolo diviso in tre sezioni rappresenta una classe:

- **Sezione superiore**: Nome della classe.
- **Sezione centrale**: Attributi della classe.
- **Sezione inferiore**: Metodi della classe.

Visibilità (Modificatori di Accesso) in UML:

- `+` (public): Accessibile ovunque.
- `-` (private): Accessibile solo all'interno della classe.
- `#` (protected): Accessibile all'interno della classe, nelle sottoclassi e nello stesso package.
- `~` (package-private/default): Accessibile solo nello stesso package.

Esempio: Diagramma UML della Classe `Persona`

Rappresentiamo la nostra classe `Persona` con i costruttori e i metodi visti finora.

1.6 Array di Oggetti

Gli array di oggetti ci permettono di gestire collezioni di istanze della stessa classe in modo efficiente. A differenza degli array di tipi primitivi, un array di oggetti contiene **riferimenti** agli oggetti, non gli oggetti stessi.

1.6.1 Dichiarazione e Inizializzazione di Array di Oggetti

```java
public class Studente {
    private String matricola;
    private String nome;
    private double media;

    public Studente(String matricola, String nome, double media) {
        this.matricola = matricola;
        this.nome = nome;
        this.media = media;
    }

    // Metodi getter (per ora, per accesso agli attributi)
    public String getMatricola() { return matricola; }
    public String getNome() { return nome; }
    public double getMedia() { return media; }

    public boolean isPromosso() {
        return media >= 6.0;
    }

    // Metodo toString per una rappresentazione testuale dell'oggetto
    @Override
    public String toString() {
        return "Studente [Matricola=" + matricola + ", Nome=" + nome +
            ", Media=" + media + "]";
    }
}
```

Creazione e Utilizzo di un Array di Studente:

```java
public class GestioneClasse {
    public static void main(String[] args) {
        // Dichiarazione e creazione di un array di 3 riferimenti a
            Studente
        Studente[] classe = new Studente[3];

        // Inizializzazione di ogni elemento dell'array con un nuovo
            oggetto Studente
        classe[0] = new Studente("A001", "Alice Rossi", 7.8);
        classe[1] = new Studente("A002", "Bob Bianchi", 5.2);
        classe[2] = new Studente("A003", "Carlo Verdi", 8.5);

        System.out.println("--- Elenco Studenti ---");
        // Iterazione con for tradizionale
        for (int i = 0; i < classe.length; i++) {
            System.out.println(classe[i].toString()); // Chiama il
                metodo toString() di ogni oggetto
        }

        System.out.println("\n--- Studenti Promossi ---");
        // Iterazione con for-each (più concisa per percorrere
            collezioni)
        for (Studente s : classe) {
            if (s.isPromosso()) {
```

```
                    System.out.println(s.getNome() + " è promosso!");
                }
            }
        }
}
```

Punti Chiave:

- Quando si dichiara ‘Studente[] classe = new Studente[3];‘, si crea un array di tre *riferimenti* a ‘Studente‘, ognuno inizializzato a ‘null‘.
- È necessario creare esplicitamente un oggetto ‘Studente‘ per ogni posizione dell’array (‘classe[0] = new Studente(...)‘). Altrimenti, tenteremo di accedere a un oggetto ‘null‘, causando una ‘NullPointerException‘.
- Possiamo accedere agli attributi e invocare i metodi di ogni oggetto nell’array utilizzando la notazione ‘classe[i].metodo()‘ o ‘s.metodo()‘.

1.7 Metodi Statici, Metodo `toString()`, Getter e Setter

Questa sezione approfondisce alcuni metodi e convenzioni importanti per la gestione delle classi.

1.7.1 Metodi Statici e Attributi Statici

Gli elementi **statici** (attributi o metodi) appartengono alla classe stessa, non a una specifica istanza (oggetto) della classe. Vengono caricati in memoria una sola volta quando la classe viene caricata e sono condivisi da tutte le istanze della classe.

- Si invocano direttamente sulla classe, non sull’oggetto: ‘NomeClasse.metodoStatico()‘ o ‘NomeClasse.attributoStatico‘.
- I metodi statici possono accedere solo ad altri membri statici della classe. Non possono accedere a membri non statici (di istanza) perché questi esistono solo in relazione a un oggetto.

Esempio: Classe `Contatore` con Membri Statici

```
public class Contatore {
    private static int conteggioGlobale = 0; // Attributo statico,
        condiviso da tutte le istanze
    private int idIstanza; // Attributo di istanza

    public Contatore() {
        conteggioGlobale++; // Incrementa il contatore globale ogni
            volta che un oggetto viene creato
        this.idIstanza = conteggioGlobale; // Assegna un ID univoco
            all'istanza
    }

    // Metodo statico: restituisce il conteggio totale degli oggetti
        creati
```

```
    public static int getConteggioGlobale() {
        return conteggioGlobale;
    }

    // Metodo di istanza: restituisce l'ID di questa specifica istanza
    public int getIdIstanza() {
        return idIstanza;
    }

    public static void main(String[] args) {
        System.out.println("Contatore globale iniziale: " + Contatore.
            getConteggioGlobale()); // 0

        Contatore c1 = new Contatore();
        Contatore c2 = new Contatore();
        Contatore c3 = new Contatore();

        System.out.println("ID c1: " + c1.getIdIstanza()); // 1
        System.out.println("ID c2: " + c2.getIdIstanza()); // 2
        System.out.println("ID c3: " + c3.getIdIstanza()); // 3

        System.out.println("Contatore globale finale: " + Contatore.
            getConteggioGlobale()); // 3
    }
}
```

1.7.2 Il Metodo `toString()`

Il metodo `toString()` è ereditato da tutte le classi Java dalla classe base `Object`. Il suo scopo è fornire una rappresentazione testuale dell'oggetto. È buona pratica fare l'override di questo metodo per restituire una stringa significativa per la nostra classe.

```
public class Prodotto {
    private String codice;
    private String descrizione;
    private double prezzo;

    public Prodotto(String codice, String descrizione, double prezzo)
        {
        this.codice = codice;
        this.descrizione = descrizione;
        this.prezzo = prezzo;
    }

    @Override // L'annotazione @Override è consigliata
    public String toString() {
        return "Prodotto [Codice=" + codice + ", Descrizione=" +
            descrizione + ", Prezzo=" + prezzo + "]";
    }

    public static void main(String[] args) {
        Prodotto p = new Prodotto("P001", "Laptop", 1200.0);
        System.out.println(p); // Chiama automaticamente p.toString()
        System.out.println("Dettagli: " + p); // Anche la
            concatenazione chiama toString()
```

```
    }
}
```

1.7.3 Getter (Accessors) e Setter (Mutators)

I **getter** e i **setter** sono metodi pubblici utilizzati per accedere e modificare gli attributi privati di una classe. Sono fondamentali per l'incapsulamento, poiché permettono di controllare come i dati vengono letti e scritti.

- **Getter**: Metodi che restituiscono il valore di un attributo. Convenzione: 'getNomeAttributo()'. Per attributi booleani, spesso 'isNomeAttributo()'.
- **Setter**: Metodi che impostano il valore di un attributo. Convenzione: 'setNomeAttributo(tipo valore)'. Possono includere logica di validazione.

Esempio: Getter e Setter nella Classe `Studente`

Aggiorniamo la classe `Studente` con getter e setter completi.

```java
public class Studente {
    private String matricola;
    private String nome;
    private double media;

    public Studente(String matricola, String nome, double media) {
        this.matricola = matricola;
        this.nome = nome;
        this.media = media;
    }

    // Getter
    public String getMatricola() {
        return matricola;
    }

    public String getNome() {
        return nome;
    }

    public double getMedia() {
        return media;
    }

    // Setter (con validazione per la media)
    public void setMatricola(String matricola) {
        this.matricola = matricola;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public void setMedia(double media) {
```

```
35          if (media >= 0.0 && media <= 10.0) { // Validazione
36              this.media = media;
37          } else {
38              System.out.println("Errore: Media non valida (deve essere
                    tra 0 e 10)");
39          }
40      }
41
42      public boolean isPromosso() {
43          return media >= 6.0;
44      }
45
46      @Override
47      public String toString() {
48          return "Studente [Matricola=" + matricola + ", Nome=" + nome +
                ", Media=" + media + "]";
49      }
50
51      public static void main(String[] args) {
52          Studente s1 = new Studente("B001", "Giulia Rossi", 7.0);
53          System.out.println(s1.getNome() + " ha media: " + s1.getMedia
                ()); // Accedo tramite getter
54
55          s1.setMedia(9.5); // Modifico tramite setter
56          System.out.println(s1.getNome() + " nuova media: " + s1.
                getMedia());
57
58          s1.setMedia(12.0); // Tentativo di impostare una media non
                valida
59          System.out.println(s1.getNome() + " media dopo tentativo non
                valido: " + s1.getMedia()); // La media rimane 9.5
60      }
61  }
```

Fino a questo punto, abbiamo gestito gli attributi della classe come ‘private‘ e i metodi (costruttori, di utilità, getter e setter) come ‘public‘. Questa è una best practice per garantire un forte incapsulamento.

1.8 Ereditarietà

L’**ereditarietà** è un pilastro fondamentale dell’OOP che permette di creare una gerarchia di classi. Una classe (sottoclasse o classe derivata) può ereditare attributi e metodi da un’altra classe (superclasse o classe base), stabilendo una relazione ”è un” (is-a).

1.8.1 La Parola Chiave `extends`

In Java, l’ereditarietà si implementa con la parola chiave `extends`.

```
1  // Superclasse
2  public class Veicolo {
3      protected String targa;
4      protected int annoImmatricolazione;
5      protected double velocitaMassima;
```

```
6
7       public Veicolo(String targa, int anno, double vMax) {
8           this.targa = targa;
9           this.annoImmatricolazione = anno;
10          this.velocitaMassima = vMax;
11      }
12
13      public void mostraInfo() {
14          System.out.println("Veicolo targa: " + targa);
15          System.out.println("Anno: " + annoImmatricolazione);
16          System.out.println("V.max: " + velocitaMassima + " km/h");
17      }
18
19      public double calcolaBollo() {
20          return 100.0; // Calcolo base del bollo
21      }
22  }
```

Modificatore `protected`: Permette agli attributi di essere accessibili all'interno della classe stessa e in tutte le sue sottoclassi, ma non da classi esterne non correlate. Questo è ideale per l'ereditarietà.

Diagramma UML di Ereditarietà

Questo diagramma mostra la relazione "è un" tra `Veicolo` (superclasse) e `Auto`, `Moto` (sottoclassi).

1.8.2 Sottoclasse `Auto`

```
1   public class Auto extends Veicolo {
2       private int numeroPosti;
3       private String alimentazione;
4
5       public Auto(String targa, int anno, double vMax, int posti, String
            alimentazione) {
6           super(targa, anno, vMax); // Chiama il costruttore della
                superclasse Veicolo
7           this.numeroPosti = posti;
8           this.alimentazione = alimentazione;
9       }
10
11      // Override del metodo della superclasse per aggiungere
            informazioni specifiche
12      @Override
13      public void mostraInfo() {
14          super.mostraInfo(); // Chiama il metodo mostraInfo() di
                Veicolo
15          System.out.println("Posti: " + numeroPosti);
16          System.out.println("Alimentazione: " + alimentazione);
17      }
18
19      // Override del metodo calcolaBollo() per implementare una logica
            specifica per le auto
20      @Override
```

```
21      public double calcolaBollo() {
22          double bolloBase = super.calcolaBollo(); // Ottiene il bollo
                base dalla superclasse
23          if (alimentazione.equalsIgnoreCase("elettrica")) {
24              return bolloBase * 0.5; // Auto elettriche hanno sconto
                    del 50%
25          }
26          return bolloBase;
27      }
28
29      // Metodo specifico della sottoclasse Auto
30      public boolean isFamiliare() {
31          return numeroPosti >= 5;
32      }
33  }
```

Parola chiave `super`:

- 'super(...)': Deve essere la prima istruzione in un costruttore di sottoclasse per invocare un costruttore della superclasse.
- 'super.metodo()': Permette di invocare un metodo della superclasse, utile quando si vuole estendere il comportamento piuttosto che sostituirlo completamente.

1.8.3 Override dei Metodi (Sovrascrittura)

L'**override** è il meccanismo attraverso il quale una sottoclasse fornisce la propria implementazione di un metodo che è già stato definito nella sua superclasse.

Regole dell'Override

- Stessa firma del metodo (nome, numero, tipo e ordine dei parametri).
- Il tipo di ritorno deve essere uguale o un sottotipo (covariante).
- Il modificatore di accesso non può essere più restrittivo di quello della superclasse.
- L'annotazione '@Override' è facoltativa ma altamente consigliata per prevenire errori.

Differenze tra Overload e Override

OVERLOAD (Sovraccarico)	**OVERRIDE** (Sovrascrittura)
Stessa classe	Superclasse e sottoclasse
Stesso nome, parametri diversi	Stessa firma (nome e parametri)
Risoluzione a tempo di compilazione (binding statico)	Risoluzione a tempo di esecuzione (binding dinamico)
Aumenta la flessibilità	Specializza il comportamento
Non richiede ereditarietà	Richiede ereditarietà

1.8.4 Sottoclasse `Moto`

```java
public class Moto extends Veicolo {
    private int cilindrata;
    private boolean hasBauletto;

    public Moto(String targa, int anno, double vMax, int cilindrata,
        boolean bauletto) {
        super(targa, anno, vMax);
        this.cilindrata = cilindrata;
        this.hasBauletto = bauletto;
    }

    @Override
    public void mostraInfo() {
        super.mostraInfo();
        System.out.println("Cilindrata: " + cilindrata + " cc");
        System.out.println("Bauletto: " + (hasBauletto ? "Sì" : "No"))
            ;
    }

    @Override
    public double calcolaBollo() {
        if (cilindrata <= 125) {
            return 50.0;
        } else if (cilindrata <= 500) {
            return 80.0;
        } else {
            return 120.0;
        }
    }
}
```

Qui il metodo ‘calcolaBollo()‘ della classe ‘Moto‘ sostituisce completamente la logica della superclasse, dimostrando un altro modo di utilizzare l’override.

1.9 Polimorfismo

Il **polimorfismo** (dal greco ”molte forme”) è la capacità di un’entità di assumere forme diverse. In OOP, significa che un riferimento a una superclasse può riferirsi a oggetti di qualsiasi sua sottoclasse. Questo permette di scrivere codice più generale e flessibile.

1.9.1 Esempio di Polimorfismo con i Veicoli

```java
public class GestioneParcoVeicoli {
    public static void main(String[] args) {
        // Un array di riferimenti di tipo Veicolo può contenere
            oggetti Auto e Moto
        Veicolo[] parcoVeicoli = new Veicolo[4];

        parcoVeicoli[0] = new Auto("AB123CD", 2020, 180.0, 5, "benzina
            ");
        parcoVeicoli[1] = new Moto("EF456GH", 2021, 200.0, 600, true);
```

```
        parcoVeicoli[2] = new Auto("IJ789KL", 2023, 150.0, 4, "elettrica");
        parcoVeicoli[3] = new Moto("MN012OP", 2019, 160.0, 125, false);

        System.out.println("=== Dettagli e Bolli dei Veicoli ===\n");

        double totaleBolli = 0;
        for (Veicolo v : parcoVeicoli) {
            v.mostraInfo(); // Chiama la versione specifica (Auto o Moto) a runtime
            double bollo = v.calcolaBollo(); // Chiama la versione specifica (Auto o Moto) a runtime
            System.out.println("Bollo da pagare: " + bollo + " euro");
            totaleBolli += bollo;
            System.out.println("--------------------");
        }

        System.out.println("\nTotale complessivo bolli: " + totaleBolli + " euro");

        // Uso di instanceof per distinguere i tipi reali (da usare con moderazione)
        System.out.println("\n--- Conteggio Veicoli ---");
        int autoCount = 0;
        int motoCount = 0;
        for (Veicolo v : parcoVeicoli) {
            if (v instanceof Auto) {
                autoCount++;
            } else if (v instanceof Moto) {
                motoCount++;
            }
        }
        System.out.println("Numero di Auto: " + autoCount);
        System.out.println("Numero di Moto: " + motoCount);
    }
}
```

Binding Dinamico (Late Binding): Quando un metodo sovrascritto viene chiamato su un riferimento di superclasse che punta a un oggetto di sottoclasse, la JVM determina a runtime quale versione del metodo deve essere eseguita, basandosi sul tipo effettivo dell'oggetto e non sul tipo del riferimento. Questo è il cuore del polimorfismo.

1.10 Package in Java

I **package** sono un meccanismo di raggruppamento per le classi, le interfacce e le altre entità Java. Servono a organizzare il codice in modo logico e gerarchico, evitando conflitti di nomi e controllando l'accesso. Possiamo pensarli come a delle cartelle all'interno del file system per i nostri file `.java` e `.class`.

1.10.1 Vantaggi dei Package

1. **Organizzazione**: Raggruppano classi correlate, rendendo il codice più gestibile.

2. **Prevenzione di conflitti di nomi**: Due classi con lo stesso nome possono esistere in package diversi (es. `java.util.Date` e `java.sql.Date`).

3. **Controllo degli accessi**: Forniscono un livello di visibilità (*package-private*) che limita l'accesso ai membri all'interno dello stesso package.

1.10.2 Dichiarazione e Naming Convention

La dichiarazione del package deve essere la **prima istruzione** non commento in un file sorgente Java.

```java
package it.scuola.gestionale.modelli;

public class Studente {
    // ...
}
```

Naming Convention: I nomi dei package seguono la convenzione del dominio internet inverso, tutto in minuscolo e separato da punti.

Esempio: `it.edu.miosito.nomeprogetto.modulo`.

1.10.3 Struttura delle Directory

La struttura delle directory del tuo progetto deve rispecchiare esattamente la gerarchia dei package. Se una classe è in `package it.scuola.gestionale.modelli;`, il suo file `.java` deve trovarsi in `project_root/it/scuola/gestionale/modelli/`.

1.10.4 L'istruzione `import`

Per utilizzare una classe di un altro package senza dover specificare il suo *fully qualified name* (nome completo del package + nome della classe), è necessario importarla.

Utilizziamo la direttiva `import` per questo scopo.

```java
package it.scuola.gestionale.applicazione;

// Importa una singola classe
import it.scuola.gestionale.modelli.Studente;
// Importa tutte le classi del package java.util (ma non i suoi sotto-
    package)
import java.util.ArrayList;

public class MainApp {
    public static void main(String[] args) {
        Studente s = new Studente("C001", "Luca Neri", 6.8);
        ArrayList<Studente> listaStudenti = new ArrayList<>();
        listaStudenti.add(s);
        System.out.println(listaStudenti);
    }
}
```

Import Statico: Permette di importare membri statici (metodi o campi) di una classe, consentendone l'uso diretto senza prefisso.

```
import static java.lang.Math.*; // Importa tutti i membri statici
    della classe Math

public class Calcoli {
    public static void main(String[] args) {
        double raggio = 2.0;
        double area = PI * pow(raggio, 2); // Uso PI e pow()
            direttamente
        System.out.println("Area: " + area);
    }
}
```

1.10.5 Package Predefiniti di Java

Java fornisce una ricca libreria standard organizzata in package, tra cui:

- `java.lang`: Classi fondamentali (es. `String`, `Math`, `System`).
  Importato automaticamente.
- `java.util`: Utilità e strutture dati (es. `ArrayList`, `HashMap`, `Scanner`).
- `java.io`: Classi per input/output (es. `File`, `BufferedReader`).
- `java.net`: Classi per la programmazione di rete.

1.11 Riepilogo

In questo capitolo abbiamo gettato le basi della programmazione orientata agli oggetti in Java. Abbiamo esplorato:

- I **pilastri dell'OOP**: Astrazione, Incapsulamento, Ereditarietà, Polimorfismo.
- La creazione di **classi e oggetti**: attributi, costruttori e metodi.
- L'**overloading** per fornire metodi con lo stesso nome ma firme diverse.
- L'uso dei **diagrammi UML** per modellare le classi.
- La gestione di **collezioni di oggetti** tramite array.
- I **membri statici** che appartengono alla classe, non all'istanza.
- L'importanza del metodo `toString()` e dei **getter/setter** per l'incapsulamento.
- L'**ereditarietà** con `extends` e `super` per riuso e specializzazione.
- L'**override** per ridefinire il comportamento nelle sottoclassi.
- Il **polimorfismo** e il binding dinamico per codice flessibile.

- L'organizzazione del codice in **package** per modularità e gestione dei nomi.

Questi concetti sono interconnessi e fondamentali per scrivere codice Java efficace, manutenibile e scalabile.