1. Programmazione Orientata agli Oggetti

1.1 Concetti fondamentali

Paradigmi di programmazione

- **Paradigma procedurale**: organizza il programma come una sequenza di procedure (funzioni) che operano su dati
- **Paradigma orientato agli oggetti (OOP)**: organizza il programma come una collezione di oggetti che contengono dati e comportamenti

Astrazione

L'astrazione è il processo che permette di identificare le caratteristiche essenziali di un'entità ignorando i dettagli non rilevanti. In OOP, l'astrazione si realizza attraverso le classi.

Classe e Oggetto

- **Classe**: è un modello (template) che definisce attributi e comportamenti comuni a un gruppo di oggetti
- **Oggetto**: è un'istanza concreta di una classe

Esempio:

```
// Definizione di una classe
class Automobile {
    // Attributi (stato)
    String marca;
    String modello;
    int anno;
    double velocita;

    // Metodi (comportamento)
    void accelera() {
        velocita += 5;
    }

    void frena() {
        velocita -= 5;
        if (velocita < 0) velocita = 0;
    }
}

// Creazione di un oggetto (istanza)
Automobile miaAuto = new Automobile();
```

```
miaAuto.marca = "Fiat";
miaAuto.modello = "Panda";
miaAuto.anno = 2020;
```

1.2 UML (Unified Modeling Language)

UML è un linguaggio di modellazione standardizzato utilizzato per visualizzare la struttura di sistemi software.

Diagrammi delle classi

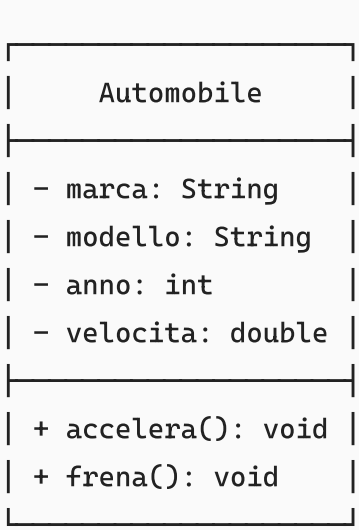
Rappresentano la struttura statica di un sistema mostrando:

- Le classi
- I loro attributi
- I loro metodi
- Le relazioni tra le classi

Simboli comuni:

- `+` pubblico
- `-` privato
- `#` protetto
- `_` sottolineato per elementi statici

Esempio di classe in UML:



2. Linguaggio Java

2.1 Caratteristiche di Java

- Linguaggio ad oggetti

- Portabile (write once, run anywhere)
- Robusto e sicuro
- Gestione automatica della memoria (garbage collection)

2.2 JVM e Class Loader

- **JVM (Java Virtual Machine)**: è un ambiente di esecuzione virtuale che permette l'esecuzione di bytecode Java
- **Bytecode**: codice intermedio generato dalla compilazione del codice sorgente Java
- **Class Loader**: componente della JVM che carica dinamicamente le classi Java

Processo di esecuzione:

1. Il codice sorgente (.java) viene compilato in bytecode (.class)
2. Il bytecode viene caricato in memoria dal Class Loader
3. La JVM esegue il bytecode

2.3 Tipi di Dati Fondamentali

Tipi primitivi

- **byte**: 8 bit, intero con segno (-128 a 127)
- **short**: 16 bit, intero con segno (-32.768 a 32.767)
- **int**: 32 bit, intero con segno (-2^31 a 2^31-1)
- **long**: 64 bit, intero con segno (-2^63 a 2^63-1)
- **float**: 32 bit, numero in virgola mobile
- **double**: 64 bit, numero in virgola mobile (precisione doppia)
- **boolean**: rappresenta valore logico (true/false)
- **char**: 16 bit, singolo carattere Unicode

Tipi di riferimento

- **Classi**: es. String, Scanner, ecc.
- **Interfacce**
- **Array**

```
// Esempi di dichiarazione e inizializzazione
int numero = 42;
double prezzo = 19.99;
char lettera = 'A';
boolean attivo = true;
String nome = "Mario";
```

2.4 Input/Output Base

Classe Scanner

```java
import java.util.Scanner;

public class Input {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Inserisci il tuo nome: ");
        String nome = scanner.nextLine();

        System.out.print("Inserisci la tua età: ");
        int eta = scanner.nextInt();

        System.out.println("Ciao " + nome + ", hai " + eta + " anni.");

        scanner.close();
    }
}
```

Differenze importanti:

- `next()`: legge fino al prossimo spazio bianco
- `nextLine()`: legge fino alla fine della linea
- Dopo aver usato `nextInt()`, `nextDouble()`, ecc., è necessario utilizzare un `nextLine()` aggiuntivo per consumare il carattere di fine linea

2.5 Classe Math

Fornisce metodi statici per operazioni matematiche comuni:

- `Math.abs(x)`: valore assoluto
- `Math.sqrt(x)`: radice quadrata
- `Math.pow(x, y)`: x elevato alla potenza y
- `Math.min(x, y)`: minimo tra x e y
- `Math.max(x, y)`: massimo tra x e y
- `Math.random()`: numero casuale tra 0.0 (incluso) e 1.0 (escluso)
- `Math.round(x)`: arrotonda all'intero più vicino
- `Math.floor(x)`: arrotonda per difetto
- `Math.ceil(x)`: arrotonda per eccesso

2.6 Errori di Arrotondamento

I numeri in virgola mobile possono presentare errori di arrotondamento:

```
double a = 0.1;
double b = 0.2;
double c = a + b;  // c sarà 0.30000000000000004, non esattamente 0.3
```

3. Strutture di Controllo

3.1 Selezione

If-else

```
if (condizione) {
    // blocco eseguito se la condizione è vera
} else if (altraCondizione) {
    // blocco eseguito se la prima condizione è falsa e la seconda è vera
} else {
    // blocco eseguito se tutte le condizioni sono false
}
```

Switch-case

```
switch (espressione) {
    case valore1:
        // codice eseguito se espressione == valore1
        break;
    case valore2:
        // codice eseguito se espressione == valore2
        break;
    default:
        // codice eseguito se nessun case corrisponde
}
```

Esempio di calcolatrice con switch:

```
import java.util.Scanner;

public class Calcolatrice {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Inserisci il primo numero: ");
        double num1 = scanner.nextDouble();

        System.out.print("Inserisci l'operazione (+, -, *, /): ");
        char operazione = scanner.next().charAt(0);

        System.out.print("Inserisci il secondo numero: ");
```

```
        double num2 = scanner.nextDouble();

        double risultato = 0;

        switch (operazione) {
            case '+':
                risultato = num1 + num2;
                break;
            case '-':
                risultato = num1 - num2;
                break;
            case '*':
                risultato = num1 * num2;
                break;
            case '/':
                if (num2 != 0) {
                    risultato = num1 / num2;
                } else {
                    System.out.println("Errore: divisione per zero!");
                    scanner.close();
                    return;
                }
                break;
            default:
                System.out.println("Operazione non valida!");
                scanner.close();
                return;
        }

        System.out.println("Risultato: " + risultato);
        scanner.close();
    }
}
```

3.2 Cicli

Ciclo for

```
for (inizializzazione; condizione; aggiornamento) {
    // blocco di codice da ripetere
}
```

Esempio:

```
// Stampa i numeri da 1 a 10
for (int i = 1; i <= 10; i++) {
```

```
    System.out.println(i);
}
```

Ciclo while

```
while (condizione) {
    // blocco di codice da ripetere finché la condizione è vera
}
```

Esempio:

```
// Stampa i numeri da 1 a 10
int i = 1;
while (i <= 10) {
    System.out.println(i);
    i++;
}
```

Ciclo do-while

```
do {
    // blocco di codice da eseguire almeno una volta
} while (condizione);
```

Esempio:

```
// Chiede un numero all'utente finché non inserisce un numero positivo
int numero;
do {
    System.out.print("Inserisci un numero positivo: ");
    numero = scanner.nextInt();
} while (numero <= 0);
```

4. Metodi in Java

4.1 Definizione e Firma

Sintassi di base:

```
modificatoreAccesso tipoRitorno nomeMetodo(parametri) {
    // corpo del metodo
    return valore; // se il tipo di ritorno non è void
}
```

Componenti della firma:

- **Modificatore di accesso**: public, private, protected o default (package-private)
- **Tipo di ritorno**: tipo di dato restituito dal metodo o void se non restituisce nulla
- **Nome del metodo**: identificatore che rispetta le convenzioni Java
- **Parametri**: lista di parametri (tipo e nome) separati da virgole

4.2 Invocazione dei Metodi

```
// Invocazione di un metodo statico
Math.abs(-10);

// Invocazione di un metodo su un oggetto
String nome = "Mario";
int lunghezza = nome.length();
```

4.3 Metodi Statici

I metodi statici appartengono alla classe e non all'istanza:

```
public class Matematica {
    // Metodo statico
    public static int somma(int a, int b) {
        return a + b;
    }

    // Metodo statico
    public static int max(int a, int b) {
        return (a > b) ? a : b;
    }
}

// Uso
int risultato = Matematica.somma(5, 3);
int massimo = Matematica.max(10, 7);
```

5. Classi e Oggetti

5.1 Definizione di Classi

```
public class Rettangolo {
    // Attributi (campi)
    private double base;
    private double altezza;

    // Costruttore senza parametri
```

```
    public Rettangolo() {
        base = 1.0;
        altezza = 1.0;
    }

    // Costruttore con parametri
    public Rettangolo(double base, double altezza) {
        this.base = base;
        this.altezza = altezza;
    }

    // Metodi
    public double calcolaArea() {
        return base * altezza;
    }

    public double calcolaPerimetro() {
        return 2 * (base + altezza);
    }

    // Getters e Setters
    public double getBase() {
        return base;
    }

    public void setBase(double base) {
        if (base > 0) {
            this.base = base;
        }
    }

    public double getAltezza() {
        return altezza;
    }

    public void setAltezza(double altezza) {
        if (altezza > 0) {
            this.altezza = altezza;
        }
    }

    // Override del metodo toString
    @Override
    public String toString() {
        return "Rettangolo [base=" + base + ", altezza=" + altezza + "]";
    }
}
```

5.2 Utilizzo della parola chiave this

La parola chiave `this` si riferisce all'istanza corrente della classe:

- Usata per distinguere tra attributi di classe e parametri con lo stesso nome
- Usata per richiamare altri costruttori della stessa classe
- Usata per passare l'oggetto corrente come parametro

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

    // Costruttore che utilizza this per distinguere tra attributi e
parametri
    public Persona(String nome, int eta) {
        this.nome = nome;
        this.eta = eta;
    }

    // Costruttore che richiama un altro costruttore
    public Persona(String nome) {
        this(nome, 0); // Richiama il costruttore con due parametri
    }
}
```

5.3 Classe tester

Una classe tester viene utilizzata per testare le funzionalità di un'altra classe:

```
public class RettangoloTester {
    public static void main(String[] args) {
        // Creazione di oggetti Rettangolo
        Rettangolo r1 = new Rettangolo();
        Rettangolo r2 = new Rettangolo(5.0, 3.0);

        // Test dei metodi
        System.out.println("Rettangolo 1: " + r1);
        System.out.println("Area: " + r1.calcolaArea());
        System.out.println("Perimetro: " + r1.calcolaPerimetro());

        System.out.println("Rettangolo 2: " + r2);
        System.out.println("Area: " + r2.calcolaArea());
        System.out.println("Perimetro: " + r2.calcolaPerimetro());

        // Modifica degli attributi
        r1.setBase(2.5);
        r1.setAltezza(1.5);

        System.out.println("Rettangolo 1 modificato: " + r1);
        System.out.println("Nuova area: " + r1.calcolaArea());
```

```
    }
}
```

5.4 JavaDoc

JavaDoc è uno strumento per la generazione di documentazione in formato HTML a partire dai commenti nel codice.

Sintassi dei commenti JavaDoc:

```
/**
 * Questa classe rappresenta un rettangolo geometrico.
 * @author Nome Cognome
 * @version 1.0
 */
public class Rettangolo {
    /**
     * La base del rettangolo.
     */
    private double base;

    /**
     * Calcola l'area del rettangolo.
     * @return l'area del rettangolo (base * altezza)
     */
    public double calcolaArea() {
        return base * altezza;
    }

    /**
     * Imposta la base del rettangolo.
     * @param base la nuova base (deve essere positiva)
     */
    public void setBase(double base) {
        if (base > 0) {
            this.base = base;
        }
    }
}
```

Generazione della documentazione:

```
javadoc -d docs Rettangolo.java
```

6. Array in Java

6.1 Concetti Base

Un array è una struttura dati che contiene elementi dello stesso tipo.

Dichiarazione e inizializzazione

```java
// Dichiarazione
int[] numeri;
String[] nomi;

// Inizializzazione
numeri = new int[5]; // Array di 5 interi
nomi = new String[3]; // Array di 3 stringhe

// Dichiarazione e inizializzazione combinata
int[] numeri = new int[5];
String[] nomi = new String[3];

// Inizializzazione con valori
int[] numeri = {10, 20, 30, 40, 50};
String[] nomi = {"Mario", "Luigi", "Peach"};
```

Accesso agli elementi

```java
int primo = numeri[0]; // Il primo elemento (indice 0)
numeri[2] = 35; // Modifica il terzo elemento (indice 2)
```

Proprietà length

```java
int lunghezza = numeri.length; // Restituisce la dimensione dell'array (5)
```

6.2 Operazioni Comuni

Scorrere un array

```java
// Con ciclo for
for (int i = 0; i < numeri.length; i++) {
    System.out.println(numeri[i]);
}

// Con for-each (disponibile da Java 5)
for (int numero : numeri) {
    System.out.println(numero);
}
```

Somma degli elementi

```
int somma = 0;
for (int numero : numeri) {
    somma += numero;
}
```

Trovare il valore massimo

```
int max = numeri[0];
for (int i = 1; i < numeri.length; i++) {
    if (numeri[i] > max) {
        max = numeri[i];
    }
}
```

6.3 Array Parzialmente Riempiti

Quando non tutti gli elementi dell'array sono utilizzati:

```
int[] numeri = new int[100]; // Dimensione massima
int contatore = 0; // Numero di elementi effettivamente utilizzati

// Aggiunta di un elemento
if (contatore < numeri.length) {
    numeri[contatore] = valore;
    contatore++;
}

// Rimozione dell'ultimo elemento
if (contatore > 0) {
    contatore--;
}
```

6.4 Array Bidimensionali

Gli array bidimensionali (matrici) sono array di array:

```
// Dichiarazione e inizializzazione
int[][] matrice = new int[3][4]; // 3 righe, 4 colonne

// Inizializzazione con valori
int[][] matrice = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};
```

```java
// Accesso agli elementi
int elemento = matrice[1][2]; // Elemento nella riga 1, colonna 2 (valore 7)

// Scorrere una matrice
for (int i = 0; i < matrice.length; i++) {
    for (int j = 0; j < matrice[i].length; j++) {
        System.out.print(matrice[i][j] + " ");
    }
    System.out.println();
}
```

6.5 Array di Oggetti

Gli array possono contenere oggetti di qualsiasi classe:

```java
// Array di oggetti Rettangolo
Rettangolo[] rettangoli = new Rettangolo[3];

// Inizializzazione degli elementi
rettangoli[0] = new Rettangolo(2.0, 3.0);
rettangoli[1] = new Rettangolo(4.0, 1.5);
rettangoli[2] = new Rettangolo(5.0, 2.0);

// Utilizzo
for (Rettangolo r : rettangoli) {
    System.out.println("Area: " + r.calcolaArea());
}
```

Esempio: ContoBancario e array di conti

```java
public class ContoBancario {
    private String intestatario;
    private double saldo;

    public ContoBancario(String intestatario, double saldoIniziale) {
        this.intestatario = intestatario;
        this.saldo = saldoIniziale;
    }

    public void deposita(double importo) {
        if (importo > 0) {
            saldo += importo;
        }
    }

    public boolean preleva(double importo) {
        if (importo > 0 && importo <= saldo) {
            saldo -= importo;
```

```
            return true;
        }
        return false;
    }

    public double getSaldo() {
        return saldo;
    }

    public String getIntestatario() {
        return intestatario;
    }

    @Override
    public String toString() {
        return "Conto di " + intestatario + ", saldo: " + saldo + " euro";
    }
}

// Utilizzo con array
ContoBancario[] conti = new ContoBancario[3];
conti[0] = new ContoBancario("Mario Rossi", 1000.0);
conti[1] = new ContoBancario("Laura Bianchi", 2500.0);
conti[2] = new ContoBancario("Paolo Verdi", 500.0);

// Calcolo del saldo totale
double saldoTotale = 0;
for (ContoBancario conto : conti) {
    saldoTotale += conto.getSaldo();
}
```

7. ArrayList

7.1 Introduzione agli ArrayList

ArrayList è una classe che implementa una lista dinamica (ridimensionabile automaticamente):

```
import java.util.ArrayList;

// Dichiarazione
ArrayList<String> nomi = new ArrayList<>(); // Java 7+
// Prima di Java 7: ArrayList<String> nomi = new ArrayList<String>();

// Aggiunta di elementi
nomi.add("Mario");
nomi.add("Luigi");
nomi.add("Peach");
```

```java
// Inserimento in una posizione specifica
nomi.add(1, "Wario"); // Inserisce "Wario" all'indice 1, spostando gli altri

// Accesso agli elementi
String nome = nomi.get(2); // "Luigi" ora è all'indice 2

// Modifica di un elemento
nomi.set(0, "Mario Bros"); // Sostituisce "Mario" con "Mario Bros"

// Rimozione di elementi
nomi.remove(3); // Rimuove "Peach"
nomi.remove("Wario"); // Rimuove la prima occorrenza di "Wario"

// Dimensione
int dimensione = nomi.size();

// Verificare se un elemento è presente
boolean contiene = nomi.contains("Luigi");

// Trovare l'indice di un elemento
int indice = nomi.indexOf("Luigi");

// Svuotare la lista
nomi.clear();

// Verificare se la lista è vuota
boolean vuota = nomi.isEmpty();
```

7.2 Differenze tra Array e ArrayList

Caratteristica	Array	ArrayList
Dimensione	Fissa (definita alla creazione)	Dinamica (si espande automaticamente)
Tipo	Può contenere tipi primitivi e oggetti	Solo oggetti (non tipi primitivi)
Sintassi	`tipo[] nome`	`ArrayList<Tipo> nome`
Accesso	Indice: `array[i]`	Metodo: `lista.get(i)`
Modifica	`array[i] = valore`	`lista.set(i, valore)`
Aggiunta	Non possibile (dimensione fissa)	`lista.add(elemento)`
Rimozione	Non possibile (dimensione fissa)	`lista.remove(i)` o `lista.remove(oggetto)`
Dimensione	Proprietà: `array.length`	Metodo: `lista.size()`

7.3 Esempi di Utilizzo

Contatto e Rubrica

```
public class Contatto {
    private String nome;
    private String cognome;
    private String telefono;

    public Contatto(String nome, String cognome, String telefono) {
        this.nome = nome;
        this.cognome = cognome;
        this.telefono = telefono;
    }

    // Getters e setters...

    @Override
    public String toString() {
        return nome + " " + cognome + " - " + telefono;
    }
}

public class Rubrica {
    private ArrayList<Contatto> contatti;

    public Rubrica() {
        contatti = new ArrayList<>();
    }

    public void aggiungiContatto(Contatto contatto) {
        contatti.add(contatto);
    }

    public boolean rimuoviContatto(String nome, String cognome) {
        for (int i = 0; i < contatti.size(); i++) {
            Contatto c = contatti.get(i);
            if (c.getNome().equals(nome) && c.getCognome().equals(cognome))
{
                contatti.remove(i);
                return true;
            }
        }
        return false;
    }

    public ArrayList<Contatto> cercaPerNome(String nome) {
        ArrayList<Contatto> risultati = new ArrayList<>();
        for (Contatto c : contatti) {
```

```
            if (c.getNome().equalsIgnoreCase(nome)) {
                risultati.add(c);
            }
        }
        return risultati;
    }

    public void stampaContatti() {
        for (Contatto c : contatti) {
            System.out.println(c);
        }
    }
}
```

8. Pacchetti in Java

8.1 Concetto di Pacchetto

I pacchetti sono contenitori che raggruppano classi correlate:

- Organizzano il codice
- Evitano conflitti di nomi
- Controllano l'accesso

8.2 Dichiarazione e Utilizzo

Dichiarazione di un pacchetto:

```
package com.esempio.geometria;

public class Rettangolo {
    // Implementazione...
}
```

Utilizzo di classi da pacchetti diversi:

```
// Import specifico
import com.esempio.geometria.Rettangolo;

// Import di tutte le classi di un pacchetto
import com.esempio.geometria.*;

// Utilizzo senza import (usando il nome completo)
com.esempio.geometria.Rettangolo r = new com.esempio.geometria.Rettangolo();
```

8.3 Pacchetti Standard

Java include diversi pacchetti standard:

- `java.lang`: classi fondamentali (importato automaticamente)
- `java.util`: collezioni, date, stringhe, ecc.
- `java.io`: input/output
- `java.net`: networking
- `java.awt` e `javax.swing`: interfacce grafiche

9. Ereditarietà e Polimorfismo

9.1 Concetto di Ereditarietà

L'ereditarietà è un meccanismo che permette a una classe di ereditare attributi e metodi da un'altra classe:

- **Superclasse** (o classe base): la classe da cui si eredita
- **Sottoclasse** (o classe derivata): la classe che eredita

```
// Superclasse
public class Veicolo {
    protected String marca;
    protected String modello;
    protected int anno;

    public Veicolo(String marca, String modello, int anno) {
        this.marca = marca;
        this.modello = modello;
        this.anno = anno;
    }

    public void avvia() {
        System.out.println("Il veicolo è stato avviato");
    }

    // Getters e setters...
}

// Sottoclasse
public class Automobile extends Veicolo {
    private int numeroPorte;

    public Automobile(String marca, String modello, int anno, int
numeroPorte) {
        super(marca, modello, anno); // Chiama il costruttore della
superclasse
        this.numeroPorte = numeroPorte;
    }
```

```
    // Override di un metodo della superclasse
    @Override
    public void avvia() {
        System.out.println("L'automobile è stata avviata");
    }

    // Metodo specifico della sottoclasse
    public void suonaClacson() {
        System.out.println("Beep beep!");
    }

    // Getters e setters...
}
```

9.2 Vantaggi dell'Ereditarietà

- **Riutilizzo del codice**: evita di scrivere lo stesso codice più volte
- **Estensibilità**: permette di aggiungere funzionalità alle classi esistenti
- **Manutenibilità**: facilita la gestione del codice

9.3 Polimorfismo

Il polimorfismo permette di trattare oggetti di diverse classi attraverso un'interfaccia comune:

```
// Utilizzo del polimorfismo
Veicolo v1 = new Veicolo("Ford", "Transit", 2020);
Veicolo v2 = new Automobile("Fiat", "Panda", 2019, 5); // Un'automobile è
anche un veicolo

v1.avvia(); // Stampa "Il veicolo è stato avviato"
v2.avvia(); // Stampa "L'automobile è stata avviata" (metodo sovrascritto)

// v2.suonaClacson(); // Errore: il tipo di riferimento è Veicolo, non
Automobile

// Downcasting (con controllo)
if (v2 instanceof Automobile) {
    Automobile a = (Automobile) v2;
    a.suonaClacson(); // OK
}
```

9.4 Overriding dei Metodi

L'overriding permette a una sottoclasse di fornire un'implementazione specifica di un metodo già definito nella superclasse:

- Il metodo deve avere la stessa firma (nome, parametri, tipo di ritorno)
- Si usa l'annotazione `@Override` per chiarezza e per controllo del compilatore
- È possibile chiamare l'implementazione della superclasse usando `super.nomeMetodo()`

```
@Override
public void avvia() {
    super.avvia(); // Chiama il metodo della superclasse
    System.out.println("Modalità specifiche dell'automobile attivate");
}
```

9.5 Metodo equals() di Object

Ogni classe in Java eredita il metodo `equals()` dalla classe Object. Di default, confronta i riferimenti:

```
public class Punto2D {
    private double x;
    private double y;

    public Punto2D(double x, double y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public boolean equals(Object obj) {
        // Controllo se l'oggetto è se stesso
        if (this == obj) return true;

        // Controllo se l'oggetto è null o di tipo diverso
        if (obj == null || getClass() != obj.getClass()) return false;

        // Cast a Punto2D
        Punto2D altro = (Punto2D) obj;

        // Confronto degli attributi
        return x == altro.x && y == altro.y;
    }
}
```

9.6 Operatore instanceof

L'operatore `instanceof` verifica se un oggetto è istanza di una classe:

```
if (obj instanceof Punto2D) {
    Punto2D punto = (Punto2D) obj;
```

```
    // ...
}
```

9.7 Relazioni tra Classi

Ereditarietà ("is-a")

Un'automobile **è un** veicolo.

```
public class Automobile extends Veicolo {
    // ...
}
```

Composizione ("has-a" stretta)

Un'automobile **ha un** motore. Il motore non può esistere senza l'automobile.

```
public class Automobile {
    private Motore motore; // Composizione

    public Automobile() {
        motore = new Motore(); // Il motore viene creato con l'automobile
    }

    // Quando l'automobile viene distrutta, il motore viene distrutto
}
```

Aggregazione ("has-a" debole)

Un'università **ha** studenti. Gli studenti possono esistere senza l'università.

```
public class Universita {
    private ArrayList<Studente> studenti; // Aggregazione

    public Universita() {
        studenti = new ArrayList<>();
    }

    public void aggiungiStudente(Studente s) {
        studenti.add(s);
    }

    // Gli studenti esistono indipendentemente dall'università
}
```

9.8 Esempi di Gerarchia di Classi

Esempio 1: Punto2D, Punto3D, Pixel

```
public class Punto2D {
    protected double x;
    protected double y;

    public Punto2D(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double distanzaDaOrigine() {
        return Math.sqrt(x*x + y*y);
    }

    public double distanzaDa(Punto2D altro) {
        double dx = x - altro.x;
        double dy = y - altro.y;
        return Math.sqrt(dx*dx + dy*dy);
    }

    // Getters e setters...
}

public class Punto3D extends Punto2D {
    private double z;

    public Punto3D(double x, double y, double z) {
        super(x, y);
        this.z = z;
    }

    @Override
    public double distanzaDaOrigine() {
        return Math.sqrt(x*x + y*y + z*z);
    }

    public double distanzaDa(Punto3D altro) {
        double dx = x - altro.x;
        double dy = y - altro.y;
        double dz = z - altro.z;
        return Math.sqrt(dx*dx + dy*dy + dz*dz);
    }

    // Getters e setters...
}

public class Pixel extends Punto2D {
    private String colore;
```

```
    public Pixel(double x, double y, String colore) {
        super(x, y);
        this.colore = colore;
    }

    public String getColore() {
        return colore;
    }

    public void setColore(String colore) {
        this.colore = colore;
    }
}
```

Esempio 2: Veicolo, Automobile, Bicicletta, Camion, Moto

```
public class Veicolo {
    protected String marca;
    protected String modello;
    protected int anno;

    public Veicolo(String marca, String modello, int anno) {
        this.marca = marca;
        this.modello = modello;
        this.anno = anno;
    }

    public void avvia() {
        System.out.println("Il veicolo è avviato.");
    }

    // Getters e setters...
}

public class Automobile extends Veicolo {
    private int numeroPorte;
    private boolean automatica;

    public Automobile(String marca, String modello, int anno, int
numeroPorte, boolean automatica) {
        super(marca, modello, anno);
        this.numeroPorte = numeroPorte;
        this.automatica = automatica;
    }

    @Override
    public void avvia() {
```

```
        System.out.println("L'automobile è avviata. Motore acceso.");
    }

    public void suonaClacson() {
        System.out.println("Beep beep!");
    }

    // Getters e setters...
}

public class Bicicletta extends Veicolo {
    private int numeroMarce;

    public Bicicletta(String marca, String modello, int anno, int
numeroMarce) {
        super(marca, modello, anno);
        this.numeroMarce = numeroMarce;
    }

    @Override
    public void avvia() {
        System.out.println("La bicicletta è pronta. Inizia a pedalare!");
    }

    public void cambiaMarcia(int marcia) {
        if (marcia > 0 && marcia <= numeroMarce) {
            System.out.println("Cambiata marcia: " + marcia);
        }
    }

    // Getters e setters...
}
```

10. Gestione delle Eccezioni

10.1 Concetto di Eccezione

Un'eccezione è un evento che interrompe il normale flusso di esecuzione di un programma. Rappresenta una condizione di errore o un evento inatteso.

Tipi di eccezioni in Java:

- **Checked exceptions**: devono essere gestite o dichiarate
- **Unchecked exceptions** (runtime exceptions): non devono necessariamente essere gestite
- **Errors**: problemi gravi che generalmente non possono essere gestiti

10.2 Gerarchia delle Eccezioni

```
Throwable
├── Error
│   ├── OutOfMemoryError
│   ├── StackOverflowError
│   └── ...
└── Exception
    ├── IOException (checked)
    ├── SQLException (checked)
    └── RuntimeException (unchecked)
        ├── ArithmeticException
        ├── NullPointerException
        ├── IndexOutOfBoundsException
        └── ...
```

10.3 Gestione delle Eccezioni

Try-Catch-Finally

```java
try {
    // Codice che potrebbe generare un'eccezione
    int risultato = 10 / 0; // Genererà un'ArithmeticException
} catch (ArithmeticException e) {
    // Gestione dell'eccezione
    System.out.println("Errore aritmetico: " + e.getMessage());
} finally {
    // Codice che viene eseguito sempre, indipendentemente dalle eccezioni
    System.out.println("Operazione completata.");
}
```

Try con più Catch

```java
try {
    int[] numeri = new int[5];
    numeri[10] = 100; // IndexOutOfBoundsException

    String s = null;
    s.length(); // NullPointerException
} catch (IndexOutOfBoundsException e) {
    System.out.println("Errore di indice: " + e.getMessage());
} catch (NullPointerException e) {
    System.out.println("Riferimento null: " + e.getMessage());
} catch (Exception e) {
    // Cattura qualsiasi altra eccezione
```

```
    System.out.println("Errore generico: " + e.getMessage());
}
```

Try-with-Resources

Introdotto in Java 7, chiude automaticamente le risorse:

```
try (Scanner scanner = new Scanner(System.in);
     FileWriter writer = new FileWriter("output.txt")) {

    String input = scanner.nextLine();
    writer.write(input);

} catch (IOException e) {
    System.out.println("Errore di I/O: " + e.getMessage());
}
// Scanner e FileWriter vengono chiusi automaticamente
```

10.4 Throw e Throws

Lanciare un'eccezione

```
public void deposita(double importo) {
    if (importo <= 0) {
        throw new IllegalArgumentException("L'importo deve essere
positivo");
    }

    saldo += importo;
}
```

Dichiarare un'eccezione

```
public void salvaFile(String contenuto, String percorso) throws IOException
{
    FileWriter writer = new FileWriter(percorso);
    writer.write(contenuto);
    writer.close();
}
```

10.5 Eccezioni Personalizzate

```
// Eccezione personalizzata
public class SaldoInsufficienteException extends Exception {
    private double saldo;
```

```
    private double importo;

    public SaldoInsufficienteException(String messaggio, double saldo,
double importo) {
        super(messaggio);
        this.saldo = saldo;
        this.importo = importo;
    }

    public double getSaldo() {
        return saldo;
    }

    public double getImporto() {
        return importo;
    }

    public double getDifferenza() {
        return importo - saldo;
    }
}

// Utilizzo
public class ContoBancario {
    private String intestatario;
    private double saldo;

    // ...

    public void preleva(double importo) throws SaldoInsufficienteException {
        if (importo <= 0) {
            throw new IllegalArgumentException("L'importo deve essere
positivo");
        }

        if (importo > saldo) {
            throw new SaldoInsufficienteException(
                "Saldo insufficiente per prelevare " + importo + " euro",
                saldo,
                importo
            );
        }

        saldo -= importo;
    }
}

// Gestione
try {
    conto.preleva(1000);
```

```
    System.out.println("Prelievo effettuato");
} catch (SaldoInsufficienteException e) {
    System.out.println(e.getMessage());
    System.out.println("Saldo attuale: " + e.getSaldo() + " euro");
    System.out.println("Mancano " + e.getDifferenza() + " euro");
}
```

10.6 File ed Eccezioni

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class LetturaFile {
    public static void main(String[] args) {
        try {
            // Apertura del file
            File file = new File("dati.txt");
            Scanner scanner = new Scanner(file);

            // Lettura del file
            while (scanner.hasNextLine()) {
                String riga = scanner.nextLine();
                System.out.println(riga);
            }

            // Chiusura del file
            scanner.close();
        } catch (FileNotFoundException e) {
            System.out.println("File non trovato: " + e.getMessage());
        }
    }
}
```