Introduzione al Paradigma Orientato agli Oggetti (OO)

Il **Paradigma Orientato agli Oggetti (OO)** è un modello di programmazione che organizza il software attorno a oggetti, piuttosto che funzioni o logiche procedurali. Questo paradigma è alla base di molti linguaggi moderni, tra cui Java, ed è progettato per migliorare la modularità, la riusabilità e la manutenibilità del codice.

1. Concetti Fondamentali del Paradigma OO

- **Oggetti:** Rappresentano entità del mondo reale o concetti astratti con stato (dati) e comportamento (metodi).
- **Classi:** Sono modelli o blueprint da cui vengono creati gli oggetti.
- **Incapsulamento:** Consente di nascondere i dettagli interni degli oggetti e fornire accesso controllato tramite metodi pubblici.
- **Ereditarietà:** Permette di creare nuove classi basate su classi esistenti, favorendo il riuso del codice.
- **Polimorfismo:** Consente di trattare oggetti di tipi diversi attraverso un'interfaccia comune.
- **Astrazione:** Nasconde i dettagli di implementazione e mostra solo le funzionalità essenziali.

—

Java come Linguaggio Puramente Orientato agli Oggetti

Java è considerato un linguaggio **puramente orientato agli oggetti**, poiché quasi ogni elemento del linguaggio ruota attorno al concetto di oggetti e classi.

1. Caratteristiche di Java

- **Tutto è un Oggetto:** Anche concetti primitivi possono essere trattati come oggetti utilizzando le classi wrapper (`Integer`, `Double`, ecc.).
- **Modularità:** Il codice è organizzato in classi e pacchetti.
- **Ereditarietà e Polimorfismo:** Sono nativamente supportati e sfruttati in molte librerie standard.
- **Incapsulamento:** Supportato attraverso modificatori di accesso (`private`, `protected`, `public`).
- **Gestione della Memoria:** La memoria è gestita automaticamente tramite il Garbage Collector.

2. Eccezioni alla Purezza OO

Java include tipi primitivi (`int`, `double`, ecc.) per motivi di performance. Tuttavia, tramite le classi wrapper, è possibile trattarli come oggetti.

Gli Oggetti

Un **oggetto** è un'entità che rappresenta uno stato (dati) e un comportamento (metodi). Gli oggetti sono le unità fondamentali in Java.

1. Caratteristiche degli Oggetti

- **Stato:** Rappresentato da variabili di istanza (o campi) dell'oggetto.
- **Comportamento:** Rappresentato dai metodi dell'oggetto.
- **Identità:** Ogni oggetto ha un identificatore univoco in memoria.

2. Creazione di un Oggetto

Gli oggetti vengono creati utilizzando la parola chiave `new`.

Esempio

```
class Automobile {
    String colore;
    int velocità;

    void avvia() {
        System.out.println("Automobile avviata.");
    }
}

Automobile auto = new Automobile();
auto.colore = "Rosso";
auto.velocità = 120;
auto.avvia();
```

Classi ed Oggetti

Classi e **Oggetti** sono i concetti chiave del paradigma OO.

1. Definizione di Classe

Una classe è un modello o schema che definisce i dati (variabili) e i comportamenti (metodi) degli oggetti.

Sintassi Generale

```
class NomeClasse {
    // Variabili di istanza
    // Costruttori
    // Metodi
}
```

Esempio

```
class Studente {
    String nome;
    int età;

    void saluta() {
        System.out.println("Ciao, mi chiamo " + nome + ".");
    }
}
```

2. Creazione di Oggetti dalla Classe

Gli oggetti sono istanze della classe. Ogni oggetto ha una copia indipendente dei dati della classe.

Esempio

```
Studente studente1 = new Studente();
studente1.nome = "Mario";
studente1.età = 21;

Studente studente2 = new Studente();
studente2.nome = "Luigi";
studente2.età = 22;

studente1.saluta(); // Output: Ciao, mi chiamo Mario.
studente2.saluta(); // Output: Ciao, mi chiamo Luigi.
```

3. Costruttori

Un costruttore è un metodo speciale utilizzato per inizializzare gli oggetti di una classe. Ha lo stesso nome della classe e non ha un tipo di ritorno.

Esempio

```
class Studente {
    String nome;
    int età;

    // Costruttore
    Studente(String nome, int età) {
```

```
        this.nome = nome;
        this.età = età;
    }
}

// Utilizzo del costruttore
Studente studente = new Studente("Mario", 21);
System.out.println("Nome: " + studente.nome); // Output: Nome: Mario
```

Programmazione Orientata agli Oggetti in Java

Cos'è un Oggetto?

Un **oggetto** è un'entità software che combina uno *stato* (dati) e un *comportamento* (metodi). Gli oggetti sono spesso usati per modellare entità reali o concetti astratti. In Java, lo stato di un oggetto è rappresentato da *variabili di istanza*, mentre i comportamenti sono definiti da *metodi*.

Incapsulamento dei dati

L'oggetto nasconde i dettagli interni e fornisce metodi pubblici per accedere e modificare il suo stato. Questo garantisce:

- Sicurezza del codice.
- Modularità e riutilizzabilità.

Esempio

Un oggetto `Automobile` può avere:

- **Attributi:** colore, modello, velocità.
- **Metodi:** accelerare, frenare.

Cos'è una Classe?

Una **classe** è un modello o un prototipo da cui vengono creati gli oggetti. Definisce lo schema dello stato e del comportamento che gli oggetti della classe avranno.

Componenti principali di una classe

- **Variabili:** rappresentano lo stato della classe.
- **Metodi:** definiscono il comportamento della classe.
- **Costruttori:** metodi speciali usati per inizializzare gli oggetti.

Esempio

```
public class Automobile {
    String colore;
    String modello;

    void accelerare() {
        System.out.println("L'auto sta accelerando");
    }
}
```

Cos'è l'Ereditarietà?

L'**ereditarietà** consente di creare una nuova classe basata su una classe esistente, riutilizzando attributi e metodi della classe base (superclasse). Questo permette di estendere e personalizzare il comportamento senza dover riscrivere tutto il codice.

Caratteristiche principali

- La parola chiave `extends` è usata per indicare che una classe eredita da un'altra.
- Il *method overriding* consente di ridefinire i metodi della superclasse nella sottoclasse.

Esempio

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

class Automobile extends Veicolo {
    @Override
    void avvia() {
        System.out.println("L'automobile è avviata");
    }
}
```

Cos'è un'Interfaccia?

Un'**interfaccia** è un contratto tra una classe e il mondo esterno. Specifica un insieme di metodi che una classe deve implementare.

Caratteristiche principali

- Una classe può implementare più interfacce.
- Gli attributi in un'interfaccia sono implicitamente `public`, `static` e `final`.

Esempio

```
interface Veicolo {
    void avvia();
}

class Bicicletta implements Veicolo {
    public void avvia() {
        System.out.println("La bicicletta è pronta");
    }
}
```

Cos'è un Package?

Un **package** è uno spazio dei nomi usato per organizzare classi e interfacce in modo logico. Aiuta a evitare conflitti di nome e migliora la gestione di progetti complessi.

Benefici

- Permette di raggruppare classi correlate.
- Migliora la modularità e la riusabilità del codice.
- Facilita la protezione delle classi con accessi personalizzati.

Creazione di un Package

```
package veicoli;

public class Automobile {
    // definizione della classe
}
```

1)Introduzione alle Variabili in Java

Le **variabili** in Java rappresentano lo stato di un oggetto e sono memorizzate nei campi (*fields*). Sono uno degli elementi fondamentali di ogni programma Java e consentono di gestire dati e stato nel corso dell'esecuzione.

Tipologie di Variabili

Il linguaggio Java definisce quattro principali tipi di variabili:

1. Variabili di Istanza (Instance Variables o No Satic Fields)

- Dichiarate senza la parola chiave `static`.
- Ogni istanza di una classe possiede una propria copia di queste variabili.

- Lo stato di un oggetto è specifico per ogni istanza, quindi i valori delle variabili di istanza possono variare tra oggetti diversi.

Esempio:

```
class Bicicletta {
    int velocità;
    int marcia;
}
```

2. Variabili di Classe (Class Variables o Static Fields)

- Dichiarate utilizzando la parola chiave `static`.
- Esiste una sola copia di queste variabili per tutta la classe, indipendentemente dal numero di istanze create.
- Possono essere dichiarate `final` per indicare che il loro valore non può essere modificato.

Esempio:

```
class Bicicletta {
    static int numeroIngranaggi = 6;
}
```

3. Variabili Locali (Local Variables)

- Dichiarate all'interno di metodi, costruttori o blocchi.
- Visibili e utilizzabili solo nel contesto in cui sono dichiarate.
- Devono essere inizializzate prima dell'uso, poiché non hanno valori predefiniti.

Esempio:

```
void esempioMetodo() {
    int contatore = 0;
    System.out.println(contatore);
}
```

4. Parametri (Parameters)

- Variabili passate ai metodi, ai costruttori o agli handler di eccezioni.
- Sono sempre considerate *variabili* e non *fields*.

Esempio:

```
void impostaVelocità(int nuovaVelocità) {
    velocità = nuovaVelocità;
}
```

Regole di Denominazione delle Variabili

La denominazione delle variabili in Java segue regole rigide per mantenere il codice leggibile e organizzato:

Regole di base

- **Case-sensitive:** i nomi delle variabili distinguono tra maiuscole e minuscole (ad esempio, `Velocità` e `velocità` sono considerati diversi).
- **Caratteri ammessi:** i nomi possono iniziare con una lettera, il simbolo `$` o il carattere _. Dopo il primo carattere, possono contenere lettere, numeri, `$` e _.
- **Parole riservate:** i nomi non possono coincidere con le parole chiave di Java, come `class`, `int`, `void`.

Buone pratiche

- Usare nomi significativi e descrittivi (ad esempio, `gearRatio` invece di `g`).
- Seguire lo stile *camelCase* per i nomi delle variabili:
  - La prima parola in minuscolo.
  - Parole successive con la prima lettera maiuscola.
- Evitare abbreviazioni criptiche, optando per nomi completi e leggibili.

Considerazioni Pratiche

- Le variabili `static` sono utili per rappresentare informazioni condivise tra tutte le istanze di una classe.
- Le variabili locali sono ideali per dati temporanei o specifici di un metodo.
- I parametri consentono di passare informazioni ai metodi o ai costruttori in modo chiaro ed efficace.

2)Espressioni, Istruzioni e Blocchi in Java

Introduzione

In Java, le espressioni, le istruzioni e i blocchi rappresentano i costrutti fondamentali per definire il comportamento di un programma. Questa guida illustra i concetti di base e le loro applicazioni.

Espressioni

Un'**espressione** è una combinazione di variabili, operatori e invocazioni di metodi costruita secondo la sintassi del linguaggio. Le espressioni producono sempre un valore.

Esempi di Espressioni

```
int cadence = 0;                // Espressione di assegnazione
anArray[0] = 100;               // Accesso a un array e assegnazione
System.out.println(cadence);    // Invocazione di metodo
int result = 1 + 2;             // Operazione aritmetica
if (value1 == value2) { ... }  // Confronto logico
```

Caratteristiche delle Espressioni

- **Tipo di dato del risultato:** Il valore prodotto da un'espressione dipende dagli elementi usati. Ad esempio:
  - `cadence = 0` restituisce un `int`.
  - `value1 == value2` restituisce un `boolean`.
- **Espressioni composte:** Si possono creare espressioni più complesse combinando più espressioni:

```
int risultato = (a + b) * c;
```

Ordine di Valutazione

- Gli operatori con maggiore precedenza vengono valutati prima.
- Le parentesi possono essere utilizzate per specificare l'ordine di valutazione:

```
(x + y) / 100  // Chiarezza nella valutazione
```

Istruzioni

Un'**istruzione** (*statement*) rappresenta un'unità completa di esecuzione. Le istruzioni terminano con un punto e virgola (`;`).

Tipologie di Istruzioni

1. **Istruzioni di espressione:**

```
aValue = 8933.234;              // Assegnazione
aValue++;                       // Incremento
System.out.println("Hello!");   // Invocazione di metodo
Bicycle myBike = new Bicycle(); // Creazione di oggetti
```

2. **Istruzioni di dichiarazione:** Dichiarano variabili e opzionalmente le inizializzano:

```
double aValue = 8933.234;
```

3. **Istruzioni di controllo del flusso:** Regolano l'ordine di esecuzione delle istruzioni:

```
if (aValue > 10) {
    System.out.println("Valore maggiore di 10");
}
```

Blocchi

Un **blocco** è un gruppo di zero o più istruzioni racchiuse tra parentesi graffe {}. I blocchi possono essere utilizzati ovunque sia ammessa una singola istruzione.

Esempio di Blocco

```
class BlockDemo {
    public static void main(String[] args) {
        boolean condition = true;
        if (condition) { // Inizio del blocco 1
            System.out.println("Condizione vera.");
        } // Fine del blocco 1
        else { // Inizio del blocco 2
            System.out.println("Condizione falsa.");
        } // Fine del blocco 2
    }
}
```

Caratteristiche dei Blocchi

- **Struttura nidificata:** I blocchi possono contenere altri blocchi.
- **Scope delle variabili:** Le variabili dichiarate all'interno di un blocco esistono solo all'interno di quel blocco.

Pratiche Consigliate

- Usa parentesi per chiarire la precedenza delle operazioni.
- Raggruppa istruzioni correlate in blocchi per migliorare la leggibilità.
- Mantieni le espressioni semplici e comprensibili.

3)Creazione di Oggetti in Java

Introduzione

In Java, una classe funge da blueprint per creare oggetti. Gli oggetti sono istanze delle classi e vengono creati seguendo un processo che coinvolge dichiarazione, istanziazione e inizializzazione.

Passaggi per Creare un Oggetto

La creazione di un oggetto in Java avviene in tre passaggi:

1. Dichiarazione

Si dichiara una variabile che farà riferimento a un oggetto. Questo passaggio associa un nome a un tipo di oggetto.

```
Point originOne;
```

2. Istanziazione

Si utilizza l'operatore `new` per allocare memoria per l'oggetto e creare un'istanza della classe.

```
originOne = new Point(23, 94);
```

3. Inizializzazione

L'operatore `new` chiama un costruttore della classe, che assegna valori iniziali alle proprietà dell'oggetto.

```
Point originOne = new Point(23, 94);
```

Costruttori

Un **costruttore** è un metodo speciale che ha lo stesso nome della classe e nessun tipo di ritorno. Serve a inizializzare un nuovo oggetto. Ogni classe ha almeno un costruttore, che può essere esplicitamente definito o fornito dal compilatore come costruttore di default senza argomenti.

Esempio di una Classe con Costruttori

```
public class Point {
    public int x = 0;
    public int y = 0;

    // Costruttore
    public Point(int a, int b) {
```

```
        x = a;
        y = b;
    }
}
```

Costruttori con Argomenti

I costruttori possono accettare parametri per personalizzare l'inizializzazione.

```
Point originOne = new Point(23, 94);
```

Esempio Completo

La classe `Rectangle` mostra come utilizzare costruttori con diversi argomenti per creare oggetti con stati differenti.

```
public class Rectangle {
    public int width = 0;
    public int height = 0;
    public Point origin;

    // Diversi costruttori
    public Rectangle() {
        origin = new Point(0, 0);
    }

    public Rectangle(Point p) {
        origin = p;
    }

    public Rectangle(int w, int h) {
        origin = new Point(0, 0);
        width = w;
        height = h;
    }

    public Rectangle(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
    }

    // Metodo per calcolare l'area
    public int getArea() {
        return width * height;
    }
}
```

Tipologie di Costruttori

- **Costruttore senza Argomenti (Default Constructor):** Fornito automaticamente dal compilatore se non esistono altri costruttori.

  ```
  Rectangle rect = new Rectangle();
  ```

- **Costruttore con Argomenti:** Consente di inizializzare i valori dell'oggetto durante la creazione.

  ```
  Rectangle rectOne = new Rectangle(originOne, 100, 200);
  ```

- **Costruttore Sovraccaricato:** Una classe può avere più costruttori con firme diverse (numero e tipo di parametri differenti).

  ```
  Rectangle rectTwo = new Rectangle(50, 100);
  ```

Note Importanti

- **Operatore `new`:** Crea l'oggetto e restituisce un riferimento alla memoria allocata.
- **Molteplici Riferimenti:** Più variabili possono fare riferimento allo stesso oggetto.

  ```
  Point p1 = new Point(0, 0);
  Point p2 = p1;  // p2 fa riferimento allo stesso oggetto di p1
  ```

Eliminazione degli Oggetti in Java

Introduzione

In Java, l'eliminazione degli oggetti è gestita automaticamente dal **Garbage Collector** (GC), che libera la memoria occupata dagli oggetti non più raggiungibili.

Garbage Collector

- Gli oggetti sono considerati **non raggiungibili** quando nessuna variabile fa riferimento a essi.
- Il `GC` si attiva automaticamente durante l'esecuzione del programma e libera la memoria occupata da questi oggetti.
- È possibile richiedere l'esecuzione del `GC` tramite:

  ```
  System.gc();
  ```

  Tuttavia, non è garantito che venga eseguito immediatamente.

Gestione delle Risorse

Per risorse esterne (es. file o socket), è preferibile usare blocchi `try-with-resources` per chiudere automaticamente le risorse:

```
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
    String linea = br.readLine();
}
```

Esempio Pratico

```
public class GarbageCollectionExample {
    public static void main(String[] args) {
        Point p1 = new Point(10, 20);
        p1 = null; // Oggetto non raggiungibile
        System.gc(); // Richiesta al Garbage Collector
        System.out.println("Fine del programma");
    }
}
```

Conclusione

La gestione della memoria in Java è automatica, ma è fondamentale rilasciare manualmente le risorse esterne per evitare problemi di memoria.

4)Controllo dell'Accesso

Il controllo dell'accesso in Java definisce la visibilità di classi, metodi e variabili tramite i **modificatori di accesso**.

Modificatori di Accesso

- **public:** Visibile ovunque nel programma.
- **protected:** Visibile nel pacchetto e nelle sottoclassi.
- **default:** Visibile solo nel pacchetto (nessun modificatore esplicito).
- **private:** Visibile solo all'interno della classe.

Esempio

```
public class Automobile {
    private String modello;
    protected int velocità;
    public void avvia() {
        System.out.println("Automobile avviata");
    }
}
```

Riuso di Classi

Il riutilizzo del codice in Java può essere ottenuto tramite:

1. Ereditarietà

Permette a una classe (sottoclasse) di estendere un'altra classe (superclasse).

```
class Veicolo {
    void avvia() {
        System.out.println("Veicolo avviato");
    }
}

class Automobile extends Veicolo {
    @Override
    void avvia() {
        super.avvia();
        System.out.println("Automobile avviata");
    }
}
```

2. Composizione

Consiste nell'includere oggetti di altre classi come membri.

```
class Motore {
    void avvia() {
        System.out.println("Motore avviato");
    }
}

class Automobile {
    private Motore motore = new Motore();
    void avvia() {
        motore.avvia();
        System.out.println("Automobile avviata");
    }
}
```

Polimorfismo

Il polimorfismo consente a un oggetto di assumere diverse forme.

1. Polimorfismo Statico (Overloading)

Permette di definire metodi con lo stesso nome ma firme diverse.

```
class Calcolatrice {
    int somma(int a, int b) {
        return a + b;
    }

    double somma(double a, double b) {
        return a + b;
    }
}
```

2. Polimorfismo Dinamico (Overriding)

Permette a una sottoclasse di ridefinire i metodi della superclasse.

```
class Veicolo {
    void avvia() {
        System.out.println("Veicolo avviato");
    }
}

class Automobile extends Veicolo {
    @Override
    void avvia() {
        System.out.println("Automobile avviata");
    }
}
```

Interfacce

Un'interfaccia è un contratto che definisce un insieme di metodi che una classe deve implementare.

Definizione e Implementazione

```
interface Veicolo {
    void avvia();
}

class Automobile implements Veicolo {
    @Override
    public void avvia() {
        System.out.println("Automobile avviata");
    }
}
```

Interfacce Funzionali

Interfacce con un solo metodo astratto, utilizzabili con espressioni lambda.

```
interface Calcolatore {
    int calcola(int a, int b);
}

public class Main {
    public static void main(String[] args) {
        Calcolatore somma = (a, b) -> a + b;
        System.out.println(somma.calcola(5, 3));
    }
}
```

Classi Interne

Le classi interne sono dichiarate all'interno di altre classi e permettono di organizzare il codice logicamente.

1. Classi Interne Non Statiche

Hanno accesso ai membri della classe esterna.

```
class Automobile {
    private String modello = "BMW";

    class Motore {
        void mostraModello() {
            System.out.println("Modello: " + modello);
        }
    }
}
```

2. Classi Interne Statiche

Non possono accedere direttamente ai membri dell'istanza della classe esterna.

```
class Automobile {
    static class Motore {
        void avvia() {
            System.out.println("Motore avviato");
        }
    }
}
```

3. Classi Locali

Definite all'interno di un metodo o di un blocco.

```
class Automobile {
    void avvia() {
        class Motore {
```

```
            void avviaMotore() {
                System.out.println("Motore avviato");
            }
        }
        Motore motore = new Motore();
        motore.avviaMotore();
    }
}
```

4. Classi Anonime

Utili per implementazioni rapide di interfacce o classi.

```
Veicolo auto = new Veicolo() {
    @Override
    public void avvia() {
        System.out.println("Automobile anonima avviata");
    }
};
auto.avvia();
```

5)Organizzazione del Codice

In Java, il codice è organizzato in maniera logica per migliorare la modularità, la leggibilità e la riutilizzabilità. L'organizzazione segue una gerarchia ben definita basata su pacchetti e classi.

1. Pacchetti

Un **pacchetto** (`package`) è un namespace che organizza classi e interfacce correlate in moduli logici. Serve a:

- Evitare conflitti di nomi tra classi.
- Raggruppare classi con funzionalità simili.
- Controllare l'accesso ai membri.

Definire un Pacchetto

Per dichiarare un pacchetto, utilizza la parola chiave `package` all'inizio del file sorgente:

```
package veicoli;
```

Tutte le classi in questo file appartengono al pacchetto `veicoli`.

Importare Classi da un Pacchetto

Puoi accedere a una classe in un altro pacchetto usando `import`:

```
import veicoli.Automobile;
```

Puoi anche importare tutte le classi di un pacchetto:

```
import veicoli.*;
```

2. Convenzioni sui Nomi

Seguire le convenzioni di denominazione rende il codice più leggibile e standardizzato:

- **Pacchetti:** Scritti in minuscolo (`com.azienda.modulo`).
- **Classi e Interfacce:** Inizia con una lettera maiuscola (`Automobile`).
- **Metodi e Variabili:** Inizia con una lettera minuscola e utilizza lo stile `camelCase` (`avviaMotore`).
- **Costanti:** Scrivi tutto maiuscolo separato da _ (`MAX_VELOCITY`).

3. Struttura dei File

Ogni file Java deve contenere una singola classe pubblica. Il nome del file deve corrispondere al nome della classe:

```
public class Automobile {
    // Definizione della classe
}
```

Il file deve essere salvato come `Automobile.java`.

4. Progetto Multi-Pacchetto

Un progetto complesso è suddiviso in più pacchetti. Esempio:

```
src/
  veicoli/
      Automobile.java
      Motore.java
  utility/
      Calcolatore.java
  Main.java
```

5. Esempio Completo

```
// File: veicoli/Automobile.java
package veicoli;

public class Automobile {
    public void avvia() {
        System.out.println("Automobile avviata");
    }
}

// File: Main.java
import veicoli.Automobile;

public class Main {
    public static void main(String[] args) {
        Automobile auto = new Automobile();
        auto.avvia();
    }
}
```

Compilazione ed esecuzione:

```
javac veicoli/Automobile.java Main.java
java Main
```

6)Strutture Dati: Holding Your Objects, Arrays, Containers in Depth

Le strutture dati in Java sono progettate per gestire e organizzare oggetti in modo efficace. Questa sezione esplora tre aspetti fondamentali: la gestione degli oggetti, gli array e i contenitori.

1. Holding Your Objects

La gestione degli oggetti (*Holding Your Objects*) riguarda il modo in cui Java organizza e conserva gli oggetti in memoria per il loro utilizzo successivo. Le strutture principali per tenere traccia degli oggetti sono:

- **Array:** Una struttura di dati fissa in termini di dimensioni che contiene elementi dello stesso tipo.
- **Contenitori:** Strutture dinamiche e flessibili offerte dal framework delle collezioni.

Uso degli Array per Conservare Oggetti

Gli array sono il modo più semplice per conservare oggetti:

```
String[] nomi = {"Mario", "Luigi", "Peach"};
System.out.println(nomi[0]); // Output: Mario
```

- Sono altamente efficienti in termini di memoria.
- La loro dimensione è fissa al momento della creazione.

Uso dei Contenitori

I contenitori offrono maggiore flessibilità rispetto agli array, consentendo:

- Dimensioni dinamiche (es. `ArrayList`).
- Operazioni avanzate come ordinamento e ricerca.

Esempio:

```
List<String> nomi = new ArrayList<>();
nomi.add("Mario");
nomi.add("Luigi");
System.out.println(nomi.get(0)); // Output: Mario
```

2. Arrays

Gli array sono strutture di dati fondamentali in Java, utilizzati per conservare collezioni di oggetti o dati primitivi.

Creazione e Uso degli Array

Un array è definito specificando il tipo di dato e la sua dimensione:

```
int[] numeri = new int[5]; // Array di interi con 5 elementi
numeri[0] = 42;            // Assegna il valore 42 al primo elemento
```

Caratteristiche degli Array

- Gli elementi sono indicizzati (indice parte da 0).
- Il tipo degli elementi è omogeneo.
- La dimensione è fissa e non può essere modificata dopo la creazione.

Array Multidimensionali

Gli array possono avere più dimensioni per rappresentare tabelle o matrici:

```
int[][] matrice = new int[3][3];
matrice[0][0] = 1;
System.out.println(matrice[0][0]); // Output: 1
```

Limitazioni degli Array

- Dimensioni fisse.
- Operazioni limitate (es. non è possibile cercare o ordinare direttamente).
- Per gestire grandi quantità di dati, è preferibile usare contenitori.

3. Containers in Depth

I contenitori sono strutture dati avanzate che offrono maggiore flessibilità rispetto agli array.

Tipologie di Contenitori

- **List:** Collezione ordinata che consente duplicati (`ArrayList`, `LinkedList`).
- **Set:** Collezione che non consente duplicati (`HashSet`, `TreeSet`).
- **Map:** Collezione di coppie chiave-valore (`HashMap`, `TreeMap`).

Esempi di Contenitori

Lista (ArrayList):

```
List<Integer> numeri = new ArrayList<>();
numeri.add(10);
numeri.add(20);
System.out.println(numeri); // Output: [10, 20]
```

Set (HashSet):

```
Set<String> nomi = new HashSet<>();
nomi.add("Mario");
nomi.add("Luigi");
nomi.add("Mario"); // Ignorato, perché duplicato
System.out.println(nomi); // Output: [Mario, Luigi]
```

Mappa (HashMap):

```
Map<String, Integer> età = new HashMap<>();
età.put("Mario", 30);
età.put("Luigi", 25);
System.out.println(età.get("Mario")); // Output: 30
```

Vantaggi dei Contenitori

- Gestione dinamica della memoria.
- Operazioni avanzate come ricerca, ordinamento e filtraggio.
- Supporto per iterazione semplificata.

Iterazione sui Contenitori

L'iterazione su un contenitore è resa semplice grazie ai cicli `for-each` e agli iteratori:

```
List<String> nomi = Arrays.asList("Mario", "Luigi", "Peach");
for (String nome : nomi) {
    System.out.println(nome);
}
```

Conclusione

Le strutture dati in Java offrono opzioni flessibili e potenti per conservare e gestire oggetti. Gli array sono semplici ed efficienti, ma i contenitori come `ArrayList`, `HashSet` e `HashMap` offrono maggiore versatilità per applicazioni più complesse.

Java Collections Framework

Il **Java Collections Framework** è un insieme di classi e interfacce che forniscono un'architettura standard per gestire e manipolare gruppi di oggetti. Questo framework offre una struttura ben definita per organizzare i dati, migliorare l'efficienza e supportare operazioni comuni come ricerca, ordinamento e iterazione.

1. Introduzione

Le collezioni in Java sono strutture dati che consentono di gestire gruppi di oggetti. Offrono diversi vantaggi:

- **Facilità di utilizzo:** semplificano la gestione di grandi quantità di dati.
- **Efficienza:** ottimizzano operazioni comuni come ricerca e ordinamento.
- **Componenti principali:**
  - **Interfacce:** Definiscono il comportamento standard delle collezioni.
  - **Implementazioni:** Forniscono implementazioni specifiche delle interfacce (es. `ArrayList`, `HashSet`).
  - **Algoritmi:** Operazioni comuni come ordinamento e ricerca.

2. Interfacce

Le interfacce rappresentano il cuore del *Java Collections Framework*. Definiscono comportamenti standard che tutte le collezioni devono seguire.

- **Collection:** Interfaccia radice che definisce le operazioni di base su gruppi di oggetti.
- **List:** Una collezione ordinata che consente duplicati (es. `ArrayList`, `LinkedList`).
- **Set:** Una collezione che non consente duplicati (es. `HashSet`, `TreeSet`).
- **Queue:** Una collezione ordinata secondo il principio FIFO (First In, First Out).
- **Map:** Collezione di coppie chiave-valore, dove ogni chiave è unica (es. `HashMap`, `TreeMap`).

3. Operazioni Aggregate

Le operazioni aggregate consentono di iterare sulle collezioni in modo efficiente, utilizzando `Stream` e metodi funzionali.

```
List<String> nomi = Arrays.asList("Mario", "Luigi", "Peach");
nomi.stream().filter(n -> n.startsWith("M")).forEach(System.out::println);
```

4. Implementazioni

Java fornisce implementazioni generali delle interfacce di collezioni:

- **ArrayList:** Lista basata su array, utile per accesso rapido.
- **LinkedList:** Lista collegata, ottimale per inserimenti/rimozioni.
- **HashSet:** Collezione non ordinata che non consente duplicati.
- **TreeSet:** Collezione ordinata che non consente duplicati.
- **HashMap:** Mappa non ordinata di coppie chiave-valore.
- **TreeMap:** Mappa ordinata di coppie chiave-valore.

5. Algoritmi

Il framework include algoritmi generici per lavorare con le collezioni:

- **Ordinamento:** `Collections.sort()`.
- **Ricerca:** `Collections.binarySearch()`.
- **Manipolazione:** Operazioni come `shuffle`, `reverse`.

  Esempio:

```
List<Integer> numeri = Arrays.asList(3, 1, 4, 1, 5);
Collections.sort(numeri);
System.out.println(numeri); // Output: [1, 1, 3, 4, 5]
```

6. Implementazioni Personalizzate

È possibile creare implementazioni personalizzate estendendo classi astratte come `AbstractList` o `AbstractMap`.

7. Interoperabilità

Le collezioni moderne interoperano con API più vecchie per garantire la compatibilità retroattiva e l'integrazione con codice legacy.

Conclusione

Il **Java Collections Framework** è uno strumento essenziale per lo sviluppo in Java, fornendo un'architettura potente e flessibile per lavorare con gruppi di dati. Le sue caratteristiche principali — interfacce, implementazioni, algoritmi e interoperabilità — lo rendono uno standard per la gestione dei dati.

7)Lezioni sulle Eccezioni in Java

Java utilizza il meccanismo delle **eccezioni** per gestire errori ed eventi eccezionali durante l'esecuzione di un programma. Questa sezione descrive come e quando utilizzare le eccezioni.

1. Che Cos'è un'Eccezione?

Un'**eccezione** è un evento che si verifica durante l'esecuzione di un programma e interrompe il normale flusso di istruzioni. Le eccezioni possono essere generate da errori di sistema, da errori logici del programma o da eventi eccezionali previsti.

Esempio:

```
int a = 10, b = 0;
int risultato = a / b; // Genera un'eccezione: ArithmeticException
```

2. Gestione delle Eccezioni

Java offre strumenti per catturare e gestire le eccezioni, garantendo che il programma non si arresti in modo anomalo.

Blocco `try-catch`

Il blocco `try-catch` consente di gestire le eccezioni.

```
try {
    int a = 10, b = 0;
    int risultato = a / b;
} catch (ArithmeticException e) {
    System.out.println("Errore: divisione per zero.");
}
```

Blocco `finally`

Il blocco `finally` viene eseguito indipendentemente dal fatto che un'eccezione sia stata catturata o meno.

```
try {
    int[] numeri = {1, 2};
    System.out.println(numeri[5]);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Indice non valido.");
} finally {
    System.out.println("Operazione completata.");
}
```

Eccezioni Chained

Le eccezioni chained consentono di collegare un'eccezione a un'altra, fornendo più contesto sugli errori.

```
Throwable eccezionePrincipale = new Exception("Errore principale");
Throwable causa = new IllegalArgumentException("Causa specifica");
eccezionePrincipale.initCause(causa);
throw eccezionePrincipale;
```

3. Lanciare le Eccezioni

Per generare un'eccezione, si utilizza la parola chiave `throw` seguita da un oggetto di tipo `Throwable` o una sua sottoclasse.

Esempio:

```
void validaInput(int numero) throws IllegalArgumentException {
    if (numero < 0) {
        throw new IllegalArgumentException("Il numero deve essere positivo.");
    }
}
```

4. Il Blocco `try-with-resources`

Il blocco `try-with-resources` è progettato per gestire oggetti che devono essere chiusi, come file o connessioni.

```
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
    String linea = br.readLine();
    System.out.println(linea);
} catch (IOException e) {
    System.out.println("Errore nella lettura del file.");
}
```

Questo assicura che il `BufferedReader` venga chiuso automaticamente.

5. Eccezioni Non Controllate (Unchecked Exceptions)

Le **eccezioni non controllate** sono sottoclassi di `RuntimeException` e non richiedono una gestione esplicita tramite `try-catch` o `throws`.

- **Esempi:**
  - `NullPointerException`
  - `ArrayIndexOutOfBoundsException`
- Possono essere evitate tramite una buona progettazione del codice.

6. Vantaggi delle Eccezioni

L'utilizzo delle eccezioni in Java offre diversi vantaggi:

- **Separazione della logica normale dal trattamento degli errori:** Il codice principale è più leggibile.
- **Riusabilità:** Le eccezioni possono essere lanciate e catturate in punti diversi del programma.
- **Chiarezza:** Gli errori vengono comunicati in modo più chiaro tramite messaggi specifici.

Conclusione

Le eccezioni rappresentano un potente strumento in Java per gestire errori ed eventi eccezionali. Utilizzarle correttamente migliora la robustezza e la manutenibilità del codice.

8)Type Information (Informazioni sui Tipi)

Java offre strumenti per ottenere informazioni sui tipi di oggetti durante l'esecuzione del programma. Questo processo è noto come **Reflection** o **Run-Time Type Identification (RTTI)**.

1. RTTI (Identificazione del Tipo a Runtime)

L'RTTI consente di determinare il tipo effettivo di un oggetto durante l'esecuzione. Si utilizza principalmente l'operatore `instanceof` per verificare l'appartenenza a un tipo specifico.

Esempio

```
Animal animale = new Cane();
if (animale instanceof Cane) {
    System.out.println("L'animale è un cane.");
}
```

2. Reflection

La **Reflection** è un meccanismo avanzato che consente di ottenere informazioni sulle classi, sui metodi e sui campi a runtime. Si utilizza la classe `Class` e le sue API.

Esempio

```
Class<?> c = String.class;
System.out.println("Nome della classe: " + c.getName());

Method[] metodi = c.getMethods();
for (Method metodo : metodi) {
    System.out.println("Metodo: " + metodo.getName());
}
```

3. Vantaggi e Svantaggi

- **Vantaggi:**
  - Permette di costruire applicazioni flessibili e dinamiche.
  - Utile per strumenti come ORM, framework di test e librerie di iniezione delle dipendenze.
- **Svantaggi:**

- Può ridurre le performance.
- Introduce complessità e rischi di sicurezza.

Strings (Stringhe)

Le stringhe in Java sono una delle strutture più utilizzate per rappresentare e manipolare sequenze di caratteri. La classe `String` è immutabile e fa parte del package `java.lang`.

1. Creazione di Stringhe

Le stringhe possono essere create in diversi modi:

- Usando i letterali:

```
String saluto = "Ciao!";
```

- Usando il costruttore:

```
String saluto = new String("Ciao!");
```

2. Metodi Principali della Classe `String`

La classe `String` offre numerosi metodi per la manipolazione di stringhe:

- `length()`: Restituisce la lunghezza della stringa.
- `charAt(int index)`: Restituisce il carattere all'indice specificato.
- `substring(int start, int end)`: Restituisce una sottostringa.
- `toUpperCase()`: Converte la stringa in maiuscolo.
- `concat(String s)`: Concatena due stringhe.

Esempio

```
String frase = "Benvenuto";
System.out.println("Lunghezza: " + frase.length());
System.out.println("Carattere all'indice 3: " + frase.charAt(3));
System.out.println("Sottostringa: " + frase.substring(0, 5));
System.out.println("Maiuscolo: " + frase.toUpperCase());
```

3. Stringhe Immutabili

Le stringhe in Java sono immutabili, il che significa che non possono essere modificate dopo la creazione. Ogni operazione di modifica crea un nuovo oggetto.

Esempio

```
String s1 = "Ciao";
String s2 = s1.concat(" Mondo");
System.out.println(s1); // Output: Ciao
System.out.println(s2); // Output: Ciao Mondo
```

4. `StringBuilder` e `StringBuffer`

Per manipolazioni intensive di stringhe, si utilizzano `StringBuilder` o `StringBuffer`, che sono mutabili.

- **StringBuilder:** Non è thread-safe, ma più veloce.
- **StringBuffer:** Thread-safe, ma più lento.

Esempio

```
StringBuilder sb = new StringBuilder("Ciao");
sb.append(" Mondo");
System.out.println(sb.toString()); // Output: Ciao Mondo
```

9)Generics (Generici)

I **Generics** in Java consentono di creare classi, interfacce e metodi parametrizzati, migliorando la sicurezza del tipo e riducendo la necessità di cast espliciti. I generics sono una parte essenziale del linguaggio e vengono utilizzati principalmente nel framework delle collezioni.

1. Vantaggi dei Generics

- **Sicurezza del tipo:** Gli errori di tipo vengono rilevati a tempo di compilazione.
- **Riusabilità:** Le classi e i metodi generici possono essere utilizzati con diversi tipi di dati.
- **Riduzione dei cast espliciti:** Non è necessario effettuare conversioni esplicite.

2. Creazione di Classi Generiche

Una classe generica utilizza un **parametro di tipo** per indicare il tipo di dati che manipolerà.

Sintassi Generale

```
class NomeClasse<T> {
    private T valore;

    public NomeClasse(T valore) {
        this.valore = valore;
```

```
    }

    public T getValore() {
        return valore;
    }
}
```

Esempio

```
class Box<T> {
    private T contenuto;

    public Box(T contenuto) {
        this.contenuto = contenuto;
    }

    public T getContenuto() {
        return contenuto;
    }
}

// Utilizzo
Box<String> scatola = new Box<>("Ciao");
System.out.println(scatola.getContenuto()); // Output: Ciao
```

3. Metodi Generici

I metodi generici possono essere definiti in qualsiasi classe, anche in quelle non generiche.

Sintassi Generale

```
public <T> void nomeMetodo(T parametro) {
    // Corpo del metodo
}
```

Esempio

```
public static <T> void stampaArray(T[] array) {
    for (T elemento : array) {
        System.out.println(elemento);
    }
}

// Utilizzo
String[] nomi = {"Mario", "Luigi", "Peach"};
stampaArray(nomi);
```

4. Generics con Collezioni

I generics sono ampiamente utilizzati nel framework delle collezioni per garantire la sicurezza del tipo.

```
List<String> lista = new ArrayList<>();
lista.add("Mario");
lista.add("Luigi");

// Errore a tempo di compilazione
// lista.add(123);
```

5. Wildcard

Le **wildcard** (`?`) consentono maggiore flessibilità nel lavorare con generics.

- `? extends T`: Consente di leggere oggetti di tipo `T` o sottotipi.
- `? super T`: Consente di scrivere oggetti di tipo `T` o supertipi.
- `?`: Accetta qualsiasi tipo.

Esempio

```
List<? extends Number> numeri = Arrays.asList(1, 2.5, 3);
for (Number numero : numeri) {
    System.out.println(numero);
}
```

Generics e Arrays

Gli **array** e i generics presentano alcune limitazioni in Java a causa della natura del tipo runtime degli array.

1. Limitazioni degli Array con Generics

Gli array in Java non possono essere direttamente generici a causa della cancellazione del tipo (*type erasure*) in fase di runtime. Questo porta a:

- Impossibilità di creare array di tipi generici.
- Errori di tipo non rilevati a tempo di compilazione.

Esempio di Errore

```
// Questo codice genera un errore a tempo di compilazione
List<String>[] arrayDiListe = new ArrayList<String>[10];
```

2. Soluzione: Uso di Cast e Collezioni

Per aggirare questa limitazione, è possibile utilizzare collezioni come `ArrayList` invece di array generici.

```
List<String> lista = new ArrayList<>();
lista.add("Mario");
```

3. Uso degli Array con Wildcard

Gli array possono essere dichiarati con wildcard per una maggiore flessibilità.

```
List<?>[] arrayGenerico = new ArrayList<?>[10];
```

4. Consigli Pratici

- Evita di mescolare generics e array quando possibile.
- Utilizza strutture come `ArrayList` per una maggiore sicurezza del tipo.

Tipi Enumerati (Enumerated Types)

I **tipi enumerati** (`enum`) in Java consentono di definire un insieme finito di costanti simboliche. Sono particolarmente utili per rappresentare valori predefiniti, migliorando la leggibilità e la sicurezza del codice.

1. Creazione di un Tipo Enumerato

Un tipo enumerato viene definito utilizzando la parola chiave `enum`. Le costanti sono scritte in maiuscolo per convenzione.

Esempio

```
public enum Giorno {
    LUNEDI, MARTEDI, MERCOLEDI, GIOVEDI, VENERDI, SABATO, DOMENICA
}
```

2. Uso dei Tipi Enumerati

Le costanti di un tipo enumerato possono essere utilizzate direttamente, come mostrato nell'esempio seguente.

Esempio

```
Giorno oggi = Giorno.LUNEDI;

switch (oggi) {
    case LUNEDI:
        System.out.println("Inizio settimana!");
        break;
```

```
    case VENERDI:
        System.out.println("Quasi weekend!");
        break;
    default:
        System.out.println("Un giorno qualsiasi.");
}
```

3. Metodi Associati agli `enum`

Gli `enum` in Java includono diversi metodi utili:

- `values()`: Restituisce un array contenente tutte le costanti.
- `valueOf(String name)`: Converte una stringa nel valore corrispondente dell'enum.
- `name()`: Restituisce il nome della costante.
- `ordinal()`: Restituisce l'indice della costante, partendo da 0.

Esempio

```
for (Giorno giorno : Giorno.values()) {
    System.out.println(giorno + " è il giorno numero " + giorno.ordinal());
}
```

4. Aggiunta di Metodi e Campi Personalizzati

Gli `enum` possono includere campi, metodi e costruttori per estendere le loro funzionalità.

Esempio

```
public enum Stagione {
    INVERNO("Freddo"), PRIMAVERA("Mite"), ESTATE("Caldo"), AUTUNNO("Fresco");

    private String descrizione;

    // Costruttore
    Stagione(String descrizione) {
        this.descrizione = descrizione;
    }

    public String getDescrizione() {
        return descrizione;
    }
}

// Utilizzo
Stagione stagione = Stagione.ESTATE;
System.out.println("L'estate è " + stagione.getDescrizione()); // Output: L'estate è
```

5. Confronto tra Costanti Enumerate

Le costanti di un `enum` possono essere confrontate utilizzando l'operatore `==` o il metodo `compareTo()`.

Esempio

```
Stagione attuale = Stagione.INVERNO;

if (attuale == Stagione.INVERNO) {
    System.out.println("Fa freddo!");
}

int confronto = Stagione.ESTATE.compareTo(Stagione.PRIMAVERA);
System.out.println(confronto); // Output: 2 (ESTATE viene dopo PRIMAVERA)
```

6. Benefici dei Tipi Enumerati

- **Leggibilità:** I tipi enumerati migliorano la leggibilità del codice rispetto a costanti numeriche o stringhe.
- **Sicurezza:** Forniscono un controllo statico sui valori validi.
- **Funzionalità Estese:** Consentono l'aggiunta di metodi personalizzati.

7. Limitazioni

- Non possono essere estesi poiché gli `enum` implicano l'ereditarietà da `java.lang.Enum`.
- Non supportano l'overloading degli operatori.

Conclusione

I `tipi enumerati` offrono un modo strutturato per rappresentare insiemi finiti di valori, migliorando la leggibilità e la sicurezza del codice. La possibilità di aggiungere metodi e campi personalizzati li rende particolarmente versatili.