Java SQL

Terza parte:

JAVA Avanzato

Giorno 18: 02.04.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer

Java
Design Pattern

Introduzione ai Design Pattern

Cosa sono i Design Pattern

I design pattern sono **soluzioni progettuali standardizzate a problemi comuni** che si presentano frequentemente durante lo sviluppo di software OOP.
Non sono porzioni di codice pronte all'uso, ma piuttosto schemi architetturali riutilizzabili che **descrivono come organizzare classi e oggetti** per risolvere in modo efficiente determinate situazioni.

Questi pattern favoriscono la **manutenibilità, la riusabilità e la flessibilità** del codice, promuovendo una progettazione più modulare e pulita.
Inoltre, forniscono un **linguaggio comune tra sviluppator**i, facilitando la comunicazione e la comprensione delle soluzioni implementate.

I design pattern migliorano la qualità del codice e facilitano lo sviluppo.

Design Pattern: Storia e tipologie

Le tipologie di design pattern

I design pattern sono stati formalizzati nel **1994** nel libro *Design Patterns: Elements of Reusable Object-Oriented Software*, scritto da Erich **Gamma**, Richard **Helm**, Ralph **Johnson** e John **Vlissides**, noti come la **Gang of Four (GoF)**.

In questo testo vengono classificati **23 pattern fondamentali**, suddivisi in tre categorie principali:

- **Pattern Creazionali** Gestiscono la creazione degli oggetti.
- **Pattern Strutturali** Definiscono come comporre classi e oggetti.
- **Pattern Comportamentali** Descrivono come gli oggetti interagiscono e comunicano tra loro

La Gang Of Four ha definito 23 design pattern

Design Pattern Creazionali

Caratteristiche e Principali Pattern Creazionali

I pattern creazionali sono focalizzati sul processo di **creazione degli oggetti**. Questi pattern **rendono il sistema indipendente** dal modo in cui gli oggetti vengono creati, composti e rappresentati. I principali pattern creazionali sono:

- **Singleton**: Garantisce che una classe abbia una sola istanza.
- **Factory Method**: Definisce un metodo per creare un oggetto, lasciando alle sottoclassi la decisione di quale classe istanziare.
- **Abstract Factory**: Fornisce un'interfaccia per creare famiglie di oggetti correlati o dipendenti.
- **Builder**: Separa la costruzione di un oggetto complesso dalla sua rappresentazione.
- **Prototype**: Crea nuovi oggetti copiando un prototipo esistente.

I pattern creazionali migliorano la flessibilità e la separazione delle responsabilità nel codice.

Design Pattern Strutturali

Caratteristiche e Principali Pattern Strutturali

I pattern strutturali riguardano la **composizione delle classi e degli oggetti** per formare strutture più grandi. Questi **pattern gestiscono le relazioni tra le entità** per essere flessibili ed efficienti. Principali pattern strutturali:

- **Adapter**: Permette a interfacce incompatibili di lavorare insieme.
- **Bridge**: Separare l'astrazione dall'implementazione.
- ***Composite***: Compone oggetti in strutture a forma di albero per rappresentare gerarchie parte-tutto.
- ***Decorator***: Aggiunge responsabilità aggiuntive a un oggetto in modo dinamico.
- **Façade**: Fornisce un'interfaccia semplificata a un sistema complesso.
- ***Flyweight***: Usa la condivisione per supportare grandi numeri di oggetti leggeri
- **Proxy**: Fornisce un surrogato per controllare l'accesso a un altro oggetto.

I pattern strutturali migliorano la composizione e la gestione delle dipendenze nel codice.

Design Pattern Comportamentali 1/2

Caratteristiche e Principali Pattern Comportamentali

I pattern comportamentali sono focalizzati sulle **interazioni tra oggetti** e sulla **assegnazione delle responsabilità**.

Questi pattern descrivono come gli oggetti collaborano e comunicano tra loro.

Principali pattern comportamentali (1/2):

- **Chain of Responsibility**: Evita di accoppiare il mittente di una richiesta al suo destinatario.
- **Command**: Incapsula una richiesta come un oggetto.
- **Interpreter**: Definisce una rappresentazione grammaticale per una lingua e un interprete.
- **Iterator**: Fornisce un modo per accedere agli elementi di un aggregato.
- ***Mediator***: Incapsula il modo in cui un insieme di oggetti interagisce.
- ***Memento***: Permette di catturare e ripristinare lo stato interno di un oggetto.

I pattern comportamentali migliorano la modularità e la gestione delle interazioni nel codice

Design Pattern Comportamentali 2/2

Caratteristiche e Principali Pattern Comportamentali

I pattern comportamentali sono focalizzati sulle interazioni tra oggetti e sull'**assegnazione delle responsabilità**.

Questi pattern descrivono come gli oggetti collaborano e comunicano tra loro.

Principali pattern comportamentali (2/2):

- **Observer**: Definisce una dipendenza uno-a-molti tra oggetti.
- **State**: Permette a un oggetto di alterare il comportamento quando il suo stato cambia.
- **Strategy**: Definisce una famiglia di algoritmi
- **Template Method**: Definisce lo *skeleton* di un algoritmo, differendo alcuni passaggi alle sottoclassi.
- ***Visitor***: Permette di definire nuove operazioni senza cambiare le classi degli elementi su cui opera.

I pattern comportamentali migliorano la modularità e la gestione delle interazioni nel codice

Creazionali - Singleton

Singleton

Il pattern Singleton garantisce che una classe abbia **una sola istanza** e fornisce un punto di accesso globale a questa istanza. Viene utilizzato quando è necessario un unico punto di controllo per una risorsa condivisa.

Caratteristiche principali:

- **Unica istanza**
- **Accesso globale**
- **Controllo centralizzato**

Esempio di utilizzo:

Gestione della configurazione dell'applicazione.

```
public class Singleton {
    private static Singleton instance;
    private Singleton() {}  // privato
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Il pattern Singleton è utile quando un'unica istanza di una classe deve avere accesso a una risorsa condivisa.

Creazionali - Factory Method

Factory Method

Il pattern Factory Method definisce un'interfaccia (o classe astratta) per creare un oggetto, ma lascia alle sottoclassi la decisione di quale classe istanziare.
Viene utilizzato per **delegare la creazione degli oggetti a sottoclassi specifiche**.
Caratteristiche principali:

- Definisce un **metodo di creazione**
- **Delegazione alle sottoclassi**
- Permette di **cambiare la classe istanziata**

Esempio di utilizzo: Creazione di diversi tipi di documenti.

Il pattern Factory Method è utile per la creazione di oggetti senza specificare la classe concreta.

Creazionali - Factory Method : Esempio

Esempio di utilizzo: Creazione di diversi tipi di documenti.

```
public abstract class Document {
    public abstract void open();
}
public class WordDocument extends Document {
    public void open() {
        System.out.println("Opening Word document");
    }
}
public class DocumentFactory {
    public static Document createDocument(String type) {
        if (type.equals("word")) {
            return new WordDocument();
        }
        // Altri tipi di documenti
        return null;
    }
}
```

Creazionali - Abstract Factory

Abstract Factory

Il pattern Abstract Factory fornisce un'interfaccia per creare famiglie di oggetti correlati o dipendenti senza specificare le loro classi concrete.
Viene utilizzato **per garantire che gli oggetti creati siano compatibili tra loro**.

Caratteristiche principali:

- Interfaccia per **famiglie di oggetti**
- **Classi** concrete **nascoste**
- Garanzia di **compatibilità**

Esempio di utilizzo: Creazione di interfacce grafiche per diversi sistemi operativi.

Il pattern Abstract Factory è utile per creare famiglie di oggetti correlati che devono essere utilizzati insieme.

Creazionali - Abstract Factory : Esempio

Abstract Factory

Esempio di utilizzo: Creazione di interfacce grafiche per diversi sistemi operativi.

```
public interface GUIFactory {
    Button createButton();
    Checkbox createCheckbox();
}

public class WinFactory implements GUIFactory {
    public Button createButton() {
        return new WinButton();
    }
    public Checkbox createCheckbox() {
        return new WinCheckbox();
    }
}
```

```
public class MacFactory implements GUIFactory {
    public Button createButton() {
        return new MacButton();
    }
    public Checkbox createCheckbox() {
        return new MacCheckbox();
    }
}
```

Differenza tra Abstract Factory e Factory Method

Confronto tra Abstract Factory e Factory Method

Sia il pattern Abstract Factory che il pattern Factory Method sono utilizzati per la creazione di oggetti, ma hanno scopi e approcci diversi.

Abstract Factory:

- Fornisce un'interfaccia per creare famiglie di oggetti correlati.
- Nasconde le classi concrete.
- Garantisce che gli oggetti creati siano compatibili tra loro.

Factory Method:

- Definisce un metodo per **creare un oggetto specifico**.
- **Delegazione della creazione** alle sottoclassi.
- Permette di **cambiare la classe istanziata** senza modificare il codice client.

In sintesi,

Abstract Factory è più adatto quando si devono **creare famiglie di oggetti** correlati

Factory Method è più utile per delegare la creazione di **singoli oggetti** a sottoclassi.

Creazionali - Builder

Builder

Il pattern Builder separa la costruzione di un oggetto complesso dalla sua rappresentazione, in modo che lo stesso processo di costruzione possa creare rappresentazioni diverse.
Viene utilizzato per **creare oggetti complessi** passo dopo passo.

Caratteristiche principali:

- **Separazione** della **costruzione** dalla **rappresentazione**
- Creazione **passo dopo passo**
- **Rappresentazioni diverse**

Esempio di utilizzo: Costruzione di oggetti complessi come case o pasti.

Il pattern Builder è utile per la creazione di oggetti complessi che richiedono una configurazione passo-passo.

Creazionali - Builder: Esempio

```
public class House {
    protected String foundation;
    protected String structure;
    protected String roof;
    protected String interior;
    public House(String foundation, String
        structure, String roof, String interior) {
        this.foundation = foundation;
        this.structure = structure;
        this.roof = roof;
        this.interior = interior;
    }
}

public class HouseBuilder {
    private String foundation;
    private String structure;
    private String roof;
    private String interior;
    // continua...
```

```
    // ...segue
    public HouseBuilder setFoundation(String foundation) {
        this.foundation = foundation;
        return this;
    }
    public HouseBuilder setStructure(String structure) {
        this.structure = structure;
        return this;
    }
    public HouseBuilder setRoof(String roof) {
        this.roof = roof;
        return this;
    }
    public HouseBuilder setInterior(String interior) {
        this.interior = interior;
        return this;
    }
    public House build() {
        return new House(foundation, structure, roof, interior);
    }
}
```

Creazionali - Prototype

Prototype

Il pattern Prototype specifica i tipi di oggetti da creare utilizzando un'istanza-prototipo e crea nuovi oggetti copiando questo prototipo.
Viene utilizzato **quando il costo di creazione di un nuovo oggetto è proibitivo**.

Caratteristiche principali:

- Utilizzo di un **prototipo**
- Creazione di **copie**
- **Riduzione del costo** di creazione

Esempio di utilizzo: Clonazione di oggetti costosi da creare.

Il pattern Prototype è utile quando è necessario creare copie di oggetti costosi da istanziare.

Creazionali - Prototype : Esempio

Esempio di utilizzo: Clonazione di oggetti costosi da creare.

```
public abstract class Shape implements Cloneable {
    private String id;
    protected String type;
    abstract void draw();
    public String getType() {
        return type;
    }
    // .. getters and setters
    public Object clone() {
        Object clone = null;
        try {
            clone = super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return clone;
    }
}
```

```
public class Circle extends Shape {

    public Circle() {
        type = "Circle";   //protected
    }

    public void draw() {
        System.out.println("Circle.Draw");
    }
}
```

Java SQL

Terza parte:
JAVA Avanzato

Giorno 19: 03.04.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer

Strutturali - Adapter

Adapter

Il pattern Adapter permette a **interfacce incompatibili di lavorare insieme** convertendo l'interfaccia di una classe in un'altra interfaccia che il client si aspetta. È utilizzato quando le classi esistenti non funzionano con altre classi a causa di interfacce incompatibili.

Caratteristiche principali:

- **Conversione delle interfacce**
- Permette la **collaborazione** tra **classi incompatibili**
- Implementazione sia come **class adapter** che come **object adapter**

Esempi: Librerie di terze parti, Interfacce legacy

Il pattern Adapter facilita la collaborazione tra classi con interfacce incompatibili.

Strutturali - Adapter : Esempio

Adapter

Esempio:

```
// Class Adapter
class Adaptee {
    public void specificRequest() {
        System.out.println("Specific request");
    }
}

interface Target {
    void request();
}

class Adapter extends Adaptee implements Target {
    public void request() {
        specificRequest();
    }
}
```

```
// Utilizzo
public class Main {
    public static void main(String[] args) {
        Target target = new Adapter();
        target.request();
    }
}
```

Strutturali - Bridge

Bridge

Il pattern Bridge separa l'astrazione dall'implementazione, permettendo a entrambe di variare indipendentemente. È utilizzato quando è necessario **disaccoppiare l'interfaccia di un'astrazion**e dalla sua implementazione concreta.

Caratteristiche principali:

- **Separazione** tra **astrazione** e **implementazione**
- Aumenta la **flessibilità** e **l'estensibilità**
- **Decoupling** delle classi

Esempi: Sistemi grafici, Stack/heap

Il pattern Bridge aumenta la flessibilità separando l'astrazione dall'implementazione.

Strutturali - Bridge : Esempio

Esempio:

```
interface Implementor {
    void operationImpl();
}

class ConcreteImplementorA implements Implementor {
    public void operationImpl() {
        System.out.println("Implementation A");
    }
}

class Abstraction {
    protected Implementor implementor;
    public Abstraction(Implementor implementor) {
        this.implementor = implementor;
    }
    public void operation() {
        implementor.operationImpl();
    }
}
```

```
// Utilizzo
public class Main {
    public static void main(String[] args) {
        Implementor implementor = new
ConcreteImplementorA();
        Abstraction abstraction = new
Abstraction(implementor);
        abstraction.operation();
    }
}"
```

Strutturali - Façade

Façade

Il pattern Façade fornisce **un'interfaccia unificata a un insieme di interfacce in un sottosistema**, rendendo più facile l'uso del sistema. È utilizzato per fornire un'interfaccia semplice a un sistema complesso.

Caratteristiche principali:

- **Interfaccia semplificata**
- **Riduce la complessità**
- Fornisce un **punto di accesso unico**

Esempi: API Sistemi complessi

Il pattern Façade semplifica l'uso di sistemi complessi fornendo un'interfaccia unificata.

Strutturali - Façade : Esempio

Esempio:

```
class Subsystem1 {
    public void operation1() {
        System.out.println("Subsystem1 operation");
    }
}

class Subsystem2 {
    public void operation2() {
        System.out.println("Subsystem2 operation");
    }
}
```

```
class Facade {
    private Subsystem1 subsystem1 = new Subsystem1();
    private Subsystem2 subsystem2 = new Subsystem2();

    public void operation() {
        subsystem1.operation1();
        subsystem2.operation2();
    }
}

// Utilizzo
public class Main {
    public static void main(String[] args) {
        Facade facade = new Facade();
        facade.operation();
    }
}
```

Strutturali - Proxy

Proxy

Il pattern Proxy fornisce un surrogato o un sostituto per un altro oggetto per controllare l'accesso ad esso. È utilizzato per fornire una **interfaccia a qualcosa che è costoso o complesso da creare**.

Caratteristiche principali:

- **Controllo dell'accesso**
- **Riduzione del carico**
- **Proxy virtuali**, remoti, di protezione

Esempi: Lazy initialization, Accesso remoto

Il pattern Proxy controlla l'accesso a un oggetto, riducendo il carico e gestendo l'inizializzazione.

Design Pattern Strutturali

Esempio:

```
interface RealSubject {
    void request();
}


class RealSubjectImpl implements RealSubject {
    public void request() {
        System.out.println("Real subject request");
    }
}
```

```
class Proxy implements RealSubject {
    private RealSubject realSubject;

    public void request() {
        if (realSubject == null) {
            realSubject = new RealSubjectImpl();
        }
        realSubject.request();
    }
}
// Utilizzo
public class Main {
    public static void main(String[] args) {
        RealSubject proxy = new Proxy();
        proxy.request();
    }
}
```

Java SQL

Terza parte:
JAVA Avanzato

Giorno 20: 04.04.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer

Comportamentali : Chain of Risponsibility

Chain of Responsibility

Il pattern Chain of Responsibility evita di accoppiare il mittente di una richiesta al suo destinatario, dando a **più oggetti la possibilità di gestire la richiesta**.
Gli oggetti sono collegati in una catena e la richiesta scorre lungo la catena fino a quando un oggetto non la gestisce.

Caratteristiche principali:

- **Disaccoppiamento** del mittente dal destinatario
- Maggiore **flessibilità** nel gestire le **richieste**
- **Implementazione dinamica** delle responsabilità

Esempi: Gestione delle richieste in un sistema di supporto

Questo pattern permette una gestione flessibile delle richieste passando la richiesta lungo una catena di handler

Comportamentali - Chain of Resp. : Esempio

Esempio:

```
abstract class Handler {
    protected Handler successor;
    public void setSuccessor(Handler successor) {
        this.successor = successor;
    }
    public abstract void handleRequest(int request);
}

class ConcreteHandler1 extends Handler {
    public void handleRequest(int request) {
        if (request < 10) {
            System.out.println("Handled by
ConcreteHandler1");
        } else if (successor != null) {
            successor.handleRequest(request);
        }
    }
}
```

```
class ConcreteHandler2 extends Handler {
    public void handleRequest(int request) {
        if (request >= 10) {
            System.out.println("Handled by
ConcreteHandler2");
        } else if (successor != null) {
            successor.handleRequest(request);
        }
    }
}
public class Main {
    public static void main(String[] args) {
        Handler handler1 = new ConcreteHandler1();
        Handler handler2 = new ConcreteHandler2();
        handler1.setSuccessor(handler2);
        handler1.handleRequest(5);
        handler1.handleRequest(15);
    }
}
```

Comportamentali - Command

Command

Il pattern Command **incapsula una richiesta come un oggetto**, permettendo di parametrizzare i client con richieste diverse, mettere in coda richieste o registrare richieste. Questo pattern disaccoppia il mittente della richiesta dal destinatario.

Caratteristiche principali:

- **Incapsulamento** delle **richieste**
- **Decoupling** del **mittente** dal **destinatario**
- Supporto per l'**undo** delle operazioni

Esempi: Sistemi di menu e pulsanti Sistemi di transazioni

Il pattern Command incapsula le richieste come oggetti, permettendo di parametrizzare i client e supportare l'undo delle operazioni.

Comportamentali - Command : Esempio

Esempio:

```
interface Command {
    void execute();
}

class ConcreteCommand implements Command {
    private Receiver receiver;
    public ConcreteCommand(Receiver receiver) {
        this.receiver = receiver;
    }
    public void execute() {
        receiver.action();
    }
}

class Receiver {
    public void action() {
        System.out.println("Receiver action executed");
    }
}
```

```
class Invoker {
    private Command command;
    public void setCommand(Command command) {
        this.command = command;
    }
    public void executeCommand() {
        command.execute();
    }
}

public class Main {
    public static void main(String[] args) {
        Receiver receiver = new Receiver();
        Command command = new
ConcreteCommand(receiver);
        Invoker invoker = new Invoker();
        invoker.setCommand(command);
        invoker.executeCommand();
    }
}
```

Comportamentali - State

State

Il pattern State permette a un **oggetto** di **alterare il suo comportamento quando il suo stato interno cambia**. L'oggetto appare come se cambiasse classe.

Caratteristiche principali:

- Permette il **cambiamento dinamico del comportamento**
- Incapsula **gli stati come oggetti**
- Migliora la **manutenibilità** e l'estensibilità

Esempi: Macchine a stati finiti, Sistemi di gestione dei processi

Il pattern State permette a un oggetto di alterare il suo comportamento dinamicamente quando il suo stato interno cambia.

Comportamentali - State : Esempio

```
interface State {
    void handle(Context context);
}

class ConcreteStateA implements State {
    public void handle(Context context) {
        System.out.println("State A");
        context.setState(new ConcreteStateB());
    }
}

class ConcreteStateB implements State {
    public void handle(Context context) {
        System.out.println("State B");
        context.setState(new ConcreteStateA());
    }
}
```

```
class Context {
    private State state;
    public Context(State state) {
        this.state = state;
    }
    public void setState(State state) {
        this.state = state;
    }
    public void request() {
        state.handle(this);
    }
}

public class Main {
    public static void main(String[] args) {
        Context context = new Context(new ConcreteStateA());
        context.request();
        context.request();
    }
}
```

Comportamentali - Strategy

Strategy

Il pattern Strategy **definisce una famiglia di algoritmi**, incapsula ciascuno di essi **e li rende intercambiabili.** Permette agli algoritmi di variare indipendentemente dal client che li utilizza.

Caratteristiche principali:

- **Incapsulamento** degli **algoritmi**
- **Intercambiabilità** degli **algoritmi**
- **Separazione** del **comportamento** dal **contesto**

Esempi: Algoritmi di ordinamento, Strategie di compressione

Il pattern Strategy permette di intercambiare facilmente algoritmi diversi senza modificare il client.

Comportamentali - Strategy : Esempio

```
interface Strategy {
    int execute(int a, int b);
}

class AddStrategy implements Strategy
{
    public int execute(int a, int b) {
        return a + b;
    }
}

class MultiplyStrategy implements
Strategy {
    public int execute(int a, int b) {
        return a * b;
    }
}
}
```

```
class Context {
    private Strategy strategy;
    public void setStrategy(Strategy strategy) {
        this.strategy = strategy;
    }
    public int executeStrategy(int a, int b) {
        return strategy.execute(a, b);
    }
}

public class Main {
    public static void main(String[] args) {
        Context context = new Context();
        context.setStrategy(new AddStrategy());
        System.out.println(
                "Add: "+context.executeStrategy(5, 3));
        context.setStrategy(new MultiplyStrategy());
        System.out.println(
                "Multiply:" + context.executeStrategy(5, 3));
    }
}
```

Comportamentali - Template Method

Template Method

Il pattern Template Method **definisce lo *skeleton* di un algoritmo** in un'operazione, differendo alcuni passaggi alle sottoclassi. Permette alle sottoclassi di ridefinire certi passi di un algoritmo senza cambiare la struttura dell'algoritmo.

Caratteristiche principali:

- Definizione di un **algoritmo** a livello **astratto**
- Permette alle **sottoclassi** di **specificare certi passi**
- Riutilizzo del **codice comune**

Esempi: Algoritmi di ordinamento, Flussi di lavoro

Il pattern Template Method permette di definire un algoritmo a livello astratto, differendo certi passi alle sottoclassi.

Comportamentali - Templ. Method : Esempio

```
abstract class AbstractClass {
    public final void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
    }
    protected abstract void primitiveOperation1();
    protected abstract void primitiveOperation2();
    private void concreteOperation() {
        System.out.println("Concrete operation");
    }
}
```

```
class ConcreteClass extends AbstractClass {
    protected void primitiveOperation1() {
        System.out.println("Primitive operation 1");
    }
    protected void primitiveOperation2() {
        System.out.println("Primitive operation 2");
    }
}


public class Main {
    public static void main(String[] args) {
        AbstractClass instance = new ConcreteClass();
        instance.templateMethod();
    }
}"
```

Java SQL

Terza parte:

JAVA Avanzato

Giorno 21: 07.04.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer

Java
Threads

Introduzione alla Programmazione Concorrente

Concetti base di concorrenza

La programmazione concorrente permette di eseguire più operazioni contemporaneamente, migliorando l'efficienza e la reattività delle applicazioni.

Questo è particolarmente utile in applicazioni che devono gestire operazioni di I/O, calcoli complessi o interazioni con l'utente.

La concorrenza introduce la necessità di gestire l'accesso a risorse condivise per evitare condizioni di race, deadlock e altri problemi comuni.

La programmazione concorrente è fondamentale per migliorare l'efficienza delle applicazioni moderne.

Processi vs. Thread

Differenze tra processi e thread

- I **processi** sono istanze di un programma in esecuzione, con il proprio spazio di indirizzamento e memoria non condivisa.

- I **thread** sono unità di esecuzione più piccole all'interno di un processo, condividono lo stesso spazio di indirizzamento e possono comunicare più facilmente tra loro rispetto ai processi.

L'uso dei thread permette un'interazione più efficiente all'interno dello stesso processo, rendendo possibile l'esecuzione parallela di più task.

Comprendere le differenze tra processi e thread è cruciale per sfruttare appieno i vantaggi della programmazione concorrente.

Creazione di Thread in Java : Thread

Estendere la classe Thread

In Java, è possibile creare un thread estendendo la classe **Thread**. Questo metodo richiede la sovrascrittura del metodo **run()**, che contiene il codice che il thread eseguirà. Una volta creato, il thread può essere avviato chiamando il metodo **start()**, che a sua volta invoca **run()** in un nuovo thread di esecuzione.

```
public class MyThread extends Thread {

    @Override
    public void run() {
        System.out.println("Thread is running");
    }

}
```

```
public static void main(String[] args) {

    MyThread t1 = new MyThread();

    t1.start();

}
```

Creare un thread estendendo la classe Thread è uno dei metodi fondamentali per la programmazione concorrente in Java.

Esecuzione di Thread

Metodo start() vs. Metodo run()

Il metodo **start**() è utilizzato per avviare un nuovo thread di esecuzione. Quando viene chiamato **start**(), il metodo **run**() del thread viene invocato in un nuovo thread di esecuzione. Se si chiama direttamente **run**(), il metodo viene eseguito nello stesso thread di chiamata, non in un nuovo thread.
È importante utilizzare **start**() per garantire che il codice venga eseguito in un thread separato.

```
public class TestThread {
    public static void main(String[] args) {
        Thread t = new Thread(() -> System.out.println("Thread is running"));
        t.start(); // Avvia un nuovo thread
        // t.run(); // Esegue run() nello stesso thread
    }
}
```

Utilizzare start() per avviare nuovi thread e garantire l'esecuzione parallela.

Creazione di Thread in Java : Runnable

Implementare l'interfaccia Runnable

Un altro modo per creare un thread in Java è implementare l'interfaccia **Runnable**. Questo approccio separa il comportamento del thread dalla sua implementazione, permettendo una maggiore flessibilità. La classe che implementa Runnable deve sovrascrivere il metodo **run()** e un'istanza di questa classe deve essere passata al costruttore di un oggetto Thread, che può essere poi avviato con start().

```
public class MyRunnable implements Runnable {

    public void run() {

        System.out.println("Thread is running");

    }
```

```
public static void main(String[] args) {

        MyRunnable runnable = new MyRunnable();

        Thread t1 = new Thread(runnable);
        t1.start();

    }
```

Implementare Runnable è un metodo versatile per creare thread, permettendo di separare il comportamento del thread dalla sua implementazione.

Metodi di Controllo dei Thread

sleep(), join(), yield(), interrupt()

Java fornisce diversi metodi per controllare il comportamento dei thread:

- ***sleep(long millis):*** mette il thread in stato di inattività per un tempo specificato in millisecondi.
- ***join()***: attende il completamento di un altro thread.
- ***yield():*** suggerisce al thread scheduler di dare la possibilità di eseguire altri thread.
- ***interrupt():*** interrompe un thread, sollevando una InterruptedException se il thread è in stato di attesa, sonno o bloccato.

Usare questi metodi per gestire l'esecuzione dei thread e sincronizzarne il comportamento.

Metodi di Controllo dei Thread

sleep(), join(), yield(), interrupt()

```
public class ThreadControl {
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            try {
                Thread.sleep(500);
                System.out.println("Thread 1 finished");
            } catch (InterruptedException e) {
                System.out.println("Thread 1 interrupted");
            }
        });
        t1.start();
        t1.join();
        System.out.println("Main thread finished");
    }
}
```

Espressioni Lambda in Java

Introduzione alle Espressioni Lambda

Le espressioni lambda sono una caratteristica introdotta in Java 8 che permette di trattare la funzionalità come un metodo a parametro unico o una funzione anonima.

Le espressioni lambda sono utilizzate principalmente per definire il comportamento in modo conciso e leggere il codice.

Sintassi di base delle Lambda:

(parametri) -> espressione (parametri) -> { blocco di codice }

```
public class LambdaExample {
    public static void main(String[] args) {
        // Espressione lambda per implementare un'interfaccia Runnable
        Runnable r = () -> System.out.println("Esecuzione di un thread con lambda");
        new Thread(r).start();
    }
}
```

Le espressioni lambda rendono il codice più conciso e leggibile, facilitando l'uso delle interfacce funzionali.

Sincronizzazione e Problemi di Concorrenza

Race Conditions e Uso del Blocco synchronized

Le race conditions si verificano quando **più thread accedono e modificano una risorsa condivisa** simultaneamente, portando a risultati inattesi.

Per prevenire queste situazioni, Java offre il blocco **synchronized**, che assicura che solo un thread alla volta possa eseguire un blocco di codice critico.

Questo aiuta a mantenere la consistenza dei dati e a prevenire accessi concorrenti non sicuri.

Usare synchronized per prevenire race conditions e garantire la sicurezza dei dati in ambienti concorrenti.

Sincronizzazione e Problemi di Concorrenza : Esempio

Esempio dell'uso di syncronized

```
class Counter {
    private int count = 0;
    public synchronized void increment() {
        count++;
    }
    public int getCount() {
        return count;
    }
}
public class SyncExample {
    public static void main(String[] args) {
        Counter counter = new Counter();
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

       // continua...
```

```
        // ...segue
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Count: " + counter.getCount());
    }
}
```

Problemi Comuni nella Programmazione Concorrente

Deadlock, Starvation, Livelock

I problemi comuni nella programmazione concorrente includono:

- **Deadlock**: due o più thread sono bloccati permanentemente in attesa l'uno dell'altro per risorse che non verranno mai rilasciate.
- **Starvation**: un thread non riesce mai a ottenere le risorse necessarie perché altri thread le monopolizzano.
- **Livelock**: due o più thread continuano a cambiare stato in risposta alle azioni dell'altro, senza fare progressi effettivi.

Riconoscere e gestire questi problemi è fondamentale per scrivere programmi concorrenti robusti.

Sincronizzazione Avanzata

Blocchi sincronizzati e Classi di Sincronizzazione

Oltre al blocco synchronized, Java offre altre classi per la sincronizzazione avanzata:

- **ReentrantLock**: una versione più flessibile del blocco synchronized, che permette di tentare di acquisire il blocco con ***tryLock***() e di supportare blocchi con timeout.
- **ReadWriteLock** : permette di avere un lock per la lettura e uno per la scrittura, migliorando le performance in scenari con letture frequenti.

```
import java.util.concurrent.locks.ReentrantLock;
public class AdvancedSync {
    private final ReentrantLock lock =
                        new ReentrantLock();
    public void doSomething() {
        lock.lock();
        try {
            // codice critico
        } finally {
            lock.unlock();
        }
    }
}
```

Utilizzare le classi di sincronizzazione avanzate per una gestione più flessibile e performante dei blocchi.

Esempio Pratico: Sincronizzazione

Codice di esempio per l'uso di ReentrantLock

L'uso di ReentrantLock permette di implementare una sincronizzazione più fine rispetto al blocco synchronized, fornendo metodi aggiuntivi per il controllo del lock.

```
import java.util.concurrent.locks.ReentrantLock;
public class ReentrantLockExample {
    private final ReentrantLock lock = new ReentrantLock();
    private int count = 0;
    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock();
        }
    }
    public int getCount() {
        return count;
    }
```

```
public static void main(String[] args)
                            throws InterruptedException {
        ReentrantLockExample example =
                        new ReentrantLockExample();
        Thread t1 = new Thread(example::increment);
        Thread t2 = new Thread(example::increment);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("Count: " + example.getCount());
    }
}
```

Questo esempio mostra come usare ReentrantLock per sincronizzare l'accesso a una risorsa condivisa.

Libreria di Concorrenza di Java

Introduzione alle java.util.concurrent

La libreria **java.util.concurrent** fornisce classi e interfacce per gestire la concorrenza in modo più efficace e sicuro.

Include:

- **Executor**: un'interfaccia che rappresenta un oggetto in grado di eseguire task.
- **ExecutorService**: un'interfaccia che aggiunge funzionalità di gestione della terminazione dei task.
- **Future**: rappresenta il risultato di un task asincrono.
- **CountDownLatch, CyclicBarrier**: strumenti per sincronizzare più thread.

*Utilizzare le classi di **java.util.concurrent** per migliorare la gestione della concorrenza nelle applicazioni Java.*

Gestione Avanzata dei Thread

Thread pool e Fork/Join framework

Un thread pool è un gruppo di thread pre-creati che possono essere riutilizzati per eseguire più task, migliorando le performance e la gestione delle risorse.
Il Fork/Join framework è progettato per suddividere i compiti complessi in sottocompiti più piccoli, utilizzando il parallelismo.

- **Executors.newFixedThreadPool(int n)**: crea un thread pool con un numero fisso di thread.
- **ForkJoinPool**: gestisce la suddivisione e il ri-accorpamento dei task.

Utilizzare i thread pool per gestire efficientemente l'esecuzione dei task e migliorare le performance delle applicazioni.

Gestione Avanzata dei Thread

Esempio di Thread Pool

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadPoolExample {

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(2);
        for (int i = 0; i < 5; i++) {
            executor.submit(() -> {
                System.out.println("Task eseguito da " + Thread.currentThread().getName());
            });
        }
        executor.shutdown();
    }

}
```

Java SQL

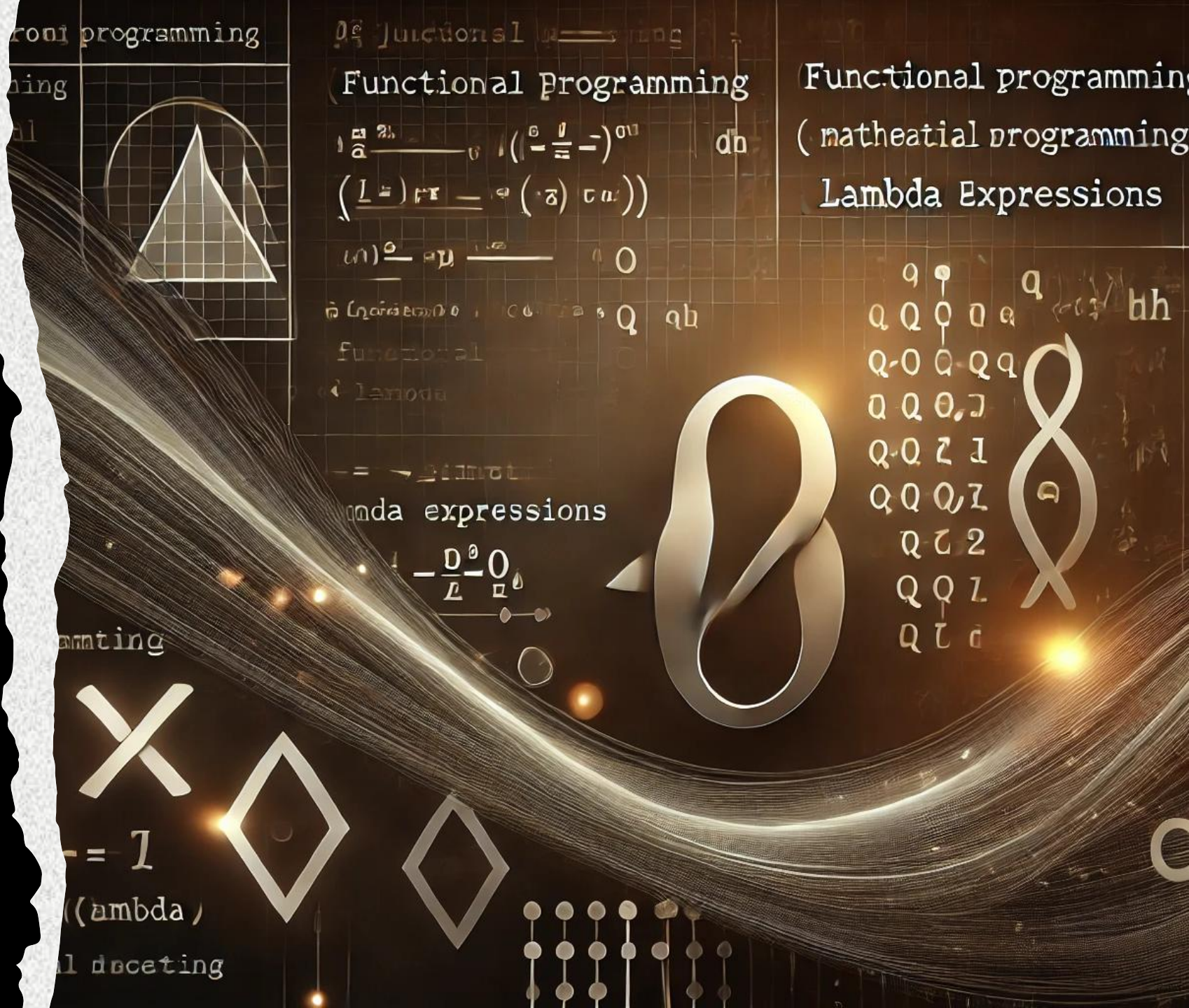
Terza parte:
JAVA Avanzato

Giorno 22: 08.04.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer

Programmazione **Funzionale**



Introduzione alla Programmazione Funzionale

Concetti Fondamentali

La programmazione funzionale è un paradigma di programmazione che tratta il calcolo come la valutazione di funzioni matematiche e evita lo stato e i dati mutabili.

In Java, la programmazione funzionale è supportata a partire dalla versione 8, che introduce le **lambda expressions** e le **interfacce funzionali**.

Questi concetti permettono di scrivere codice più conciso e leggibile, riducendo al contempo la complessità e i possibili errori legati alla gestione dello stato.

La programmazione funzionale offre nuovi strumenti per affrontare problemi complessi in modo molto più efficiente e pulito.

Differenze tra Programmazione Imperativa e Funzionale

Confronto dei Paradigmi

La programmazione imperativa si concentra su come raggiungere un risultato attraverso una serie di istruzioni che mutano lo stato del programma.
Esempi di linguaggi imperativi includono C, C++ e Java (prima della versione 8). Al contrario, la programmazione funzionale si concentra sul **cosa calcolare**, utilizzando **funzioni pure che non hanno effetti collaterali**.
Questo porta a un codice che è spesso più facile da comprendere, testare e manutenere.
In Java, la programmazione funzionale è facilitata dalle **lambda expressions** e dallo **Stream API**.

Capire le differenze tra i paradigmi aiuta a scegliere l'approccio migliore per ogni problema specifico.

Funzioni come Cittadini di Prima Classe

Concetto di Funzioni di Prima Classe

Nella programmazione funzionale, le funzioni sono trattate come cittadini di prima classe, il che significa che **possono essere assegnate a variabili**, passate come argomenti ad altre funzioni, e ritornate da funzioni.
Questo concetto permette di creare codice altamente modulare e riutilizzabile.

In Java, le lambda expressions e le interfacce funzionali permettono di utilizzare le funzioni come cittadini di prima classe, aprendo nuove possibilità per il design del software.

Le funzioni come cittadini di prima classe rendono il codice più flessibile e potente.

Interfacce Funzionali

Uso delle Interfacce Funzionali in Java

Le interfacce funzionali **sono interfacce che contengono esattamente un solo metodo astratto**. In Java, sono la base per le *lambda expressions*.

Esempi di interfacce funzionali predefinite includono:

Predicate<T>, Function<T, R>, Consumer<T> e Supplier<T>

Queste interfacce permettono di utilizzare metodi come argomenti e risultati di altre funzioni, facilitando la programmazione funzionale.

Una singola annotazione **@FunctionalInterface** è utilizzata per dichiarare un'interfaccia funzionale.

Le interfacce funzionali sono fondamentali per utilizzare le lambda expressions in Java.

Interfacce Funzionali : Esempio

Uso delle Interfacce Funzionali in Java

```
@FunctionalInterface
public interface MyFunctionalInterface {
    void execute();
}

MyFunctionalInterface func = () -> {
     System.out.println("Hello Functional Interface!");
}

func.execute();
```

Le interfacce funzionali sono fondamentali per utilizzare le lambda expressions in Java.

Lambda Expressions - Sintassi e Utilizzo

Introduzione alle Lambda Expressions

- Le lambda expressions sono una caratteristica introdotta in Java 8 che permette di trattare le funzioni come valori di prima classe.
- La sintassi delle lambda expressions è concisa e permette di scrivere codice che sarebbe altrimenti verboso e ripetitivo.
- La forma generale di una lambda expression è ***(argomenti) -> {corpo}.***
- Le lambda expressions possono essere utilizzate ovunque sia **richiesta una interfaccia funzionale**, rendendo il codice più leggibile e meno soggetto a errori.

```
(String s) -> System.out.println(s);
Runnable r = () -> System.out.println("Running in a thread");
r.run();
```

Le lambda expressions semplificano il codice rendendolo più conciso e leggibile.

Ambito delle Lambda Expressions

Scope e Limiti delle Lambda

Le **lambda expressions** in Java hanno **accesso** alle variabili **finali** o **effettivamente finali** definite nel loro ambito circostante.

Tuttavia, non possono modificare le variabili locali di questo ambito.

Questo limita l'uso delle lambda expressions ma **garantisce che siano sicure da usare in contesti paralleli**.

Queste restrizioni influenzano il design del codice e sul modi di lavorare con esse.

```
final int factor = 2;
List<Integer> numbers = Arrays.asList(1, 2, 3, 4);

List<Integer> doubled = numbers.stream().map(n -> n * factor)
                                        .collect(Collectors.toList());
System.out.println(doubled); // Output: [2, 4, 6, 8]
```

Capire l'ambito delle lambda expressions è cruciale per utilizzarle correttamente e in modo sicuro.

Esempi Pratici di Lambda Expressions

Utilizzo delle Lambda in Situazioni Reali

Le lambda expressions possono essere utilizzate in molte situazioni reali per semplificare il codice. Ad esempio, possono essere utilizzate per iterare su collezioni, filtrare dati, mappare valori, e ridurre collezioni a un singolo valore. Ecco alcuni esempi pratici per comprendere come le lambda expressions possono migliorare la leggibilità e la manutenzione del codice.

```
List<String> names =
                Arrays.asList("Alice", "Bob", "Charlie");

names.forEach(name -> System.out.println(name));

List<String> filteredNames =
                names.stream()
                .filter(name -> name.startsWith("A"))
                .collect(Collectors.toList());
```

```
List<Integer> lengths = names.stream()
                            .map(name -> name.length())
                            .collect(Collectors.toList());

int totalLength = names.stream()
                       .mapToInt(name -> name.length())
                       .sum();
```

Gli esempi pratici aiutano a comprendere l'uso delle lambda expressions nel codice quotidiano.

Stream API e Lambda Expressions

Utilizzo dello Stream API

Lo Stream API di Java è uno strumento potente per lavorare con **collezioni di dati in modo funzionale.**

Consente operazioni come filtro, mappatura e riduzione in modo conciso e leggibile utilizzando lambda expressions.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.stream()
     .filter(name -> name.startsWith("A"))
     .map(String::toUpperCase)
     .forEach(System.out::println); // Output: ALICE
```

Lo Stream API insieme alle lambda expressions permette di lavorare con i dati in modo più efficiente.

Pipeline di Stream

Costruzione di Pipeline di Stream

Una pipeline di stream consiste in una **sequenza di operazioni** che vengono eseguite su un flusso di dati.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

List<String> result = names.stream()
                           .filter(name -> name.length() > 3)
                           .map(String::toUpperCase)
                           .collect(Collectors.toList());

System.out.println(result); // Output: [ALICE, CHARLIE]
```

Queste operazioni possono essere:

- **intermedie**: *filter, map,...*
- **terminali**: *collect, forEach,...*

Le pipeline di stream permettono di trasformare e processare i dati in modo dichiarativo e conciso.

Una pipeline è fatta da zero o più operazioni intermedie e **una** operazione **terminale**

Costruire pipeline di stream aiuta a gestire e trasformare i dati in modo più leggibile e manutenibile.

Parallel Streams

Utilizzo dei Parallel Streams

I parallel streams permettono di eseguire operazioni di **stream in parallelo,** sfruttando le capacità multicore dei moderni processori.
Questo può migliorare significativamente le performance per operazioni intensive sui dati. Tuttavia, l'uso dei parallel streams richiede attenzione per evitare **problemi di concorrenza** e garantire la correttezza del risultato.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

int sum = numbers.parallelStream()
                 .filter(n -> n % 2 == 0)
                 .mapToInt(Integer::intValue)
                 .sum();

System.out.println(sum); // Output: 30
```

I parallel streams possono migliorare le performance, ma vanno usati con attenzione.