

# Java SQL

## Seconda parte: JAVA Fondamenti

Giorno 11: 14.05.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer



# Storia di Java

*Da Sun Microsystems a Oggi*

## Origini (Anni '90)

Sviluppato da Sun Microsystems. Originariamente chiamato "Oak", poi rinominato in Java nel 1995.

## Java 1.0 (1996)

Rilascio della prima versione pubblica, promuovendo il concetto di "Write Once, Run Anywhere".

## Acquisizione da Oracle (2010)

Sun Microsystems è stata acquisita da Oracle Corporation, portando nuove direzioni nello sviluppo di Java.

*Java è uno dei linguaggi più utilizzati al mondo.*

# Evoluzione di Java: Le versioni principali

*Un Cammino di Innovazione Continua*

**Java 1.0 (1996):** Il debutto di Java con il concetto di "Write Once, Run Anywhere".

**Java 2 (1998 - 2006):** Introduzione di numerose API, inclusi Swing e Java Collections Framework.

**Java 5 (2004):** Introduzione di nuove funzionalità come generics, annotations, autoboxing/unboxing.

**Java 7 (2011):** Aggiunta del file system NIO.2, switch su stringhe, miglioramenti della gestione delle eccezioni.

**Java 8 (2014):** Rivoluzionaria per l'introduzione delle espressioni lambda, dell'API Stream e dell'Optional class.

**Java 9 (2017):** Introduzione del sistema di moduli (Project Jigsaw) e miglioramenti all'API.

**Java 11 (2018):** Rilascio come LTS (Long Term Support) con miglioramenti nelle API e nuove funzionalità.

**Java 17 (2021):** Altra versione LTS con miglioramenti in termini di performance, sicurezza e modernità del linguaggio come Record e sealed Class.

# Caratteristiche Principali di Java

*Perché Java?*

- Portatile e indipendente dalla piattaforma
- Fortemente orientato agli oggetti
- Semplice, sicuro e robusto
- Supporta il multithreading

*Java è uno dei linguaggi più utilizzati al mondo.*

# Caratteristiche Principali di Java

*Portabilità e Orientamento agli Oggetti*

## Portabilità

*Write Once, Run Anywhere* - Java è progettato per funzionare su qualsiasi piattaforma senza la necessità di riscrivere il codice.

## Object Oriented

Tutto in Java è un oggetto o una **classe**, promuovendo un design modulare e riutilizzabile e orientato agli oggetti

*Java: Portatile e Modularmente Progettato*

# Caratteristiche Uniche di Java

## *Sicurezza e Multithreading*

### **Sicurezza**

Java offre un robusto sistema di sicurezza, compreso il controllo degli accessi e il sandboxing delle applet.

### **Multithreading**

Supporto nativo per il multithreading, permettendo l'esecuzione di più thread o processi in parallelo.

*Java: Sicuro e Efficiente con il Multithreading*

# Usi di Java nel mondo reale

*Perché Java?*

- **Sviluppo di applicazioni web e server**
- **Applicazioni Android**
- **Applicazioni enterprise**
- **Applicazioni scientifiche e di ricerca**
- **Sviluppo su sistemi legacy**

*Java trova applicazione in una grande varietà di contesti.*

# Usi di Java nel mondo reale

*Sviluppo Web e Applicazioni Enterprise*

## Sviluppo Web

Utilizzo nei server back-end, servlet, JSP, framework come Spring e Hibernate.

## Applicazioni Enterprise

Java è ampiamente usato per sviluppare applicazioni aziendali robuste, scalabili e sicure.

*Java: Colonna Portante dello Sviluppo Web e delle Applicazioni Enterprise*

# Usi di Java nel mondo reale

## *Sviluppo Mobile e Applicazioni Scientifiche*

### **Sviluppo Mobile (Android)**

Java è stato il linguaggio primario per lo sviluppo di app Android fino all'avvento di Kotlin.

### **Applicazioni Scientifiche e di Ricerca**

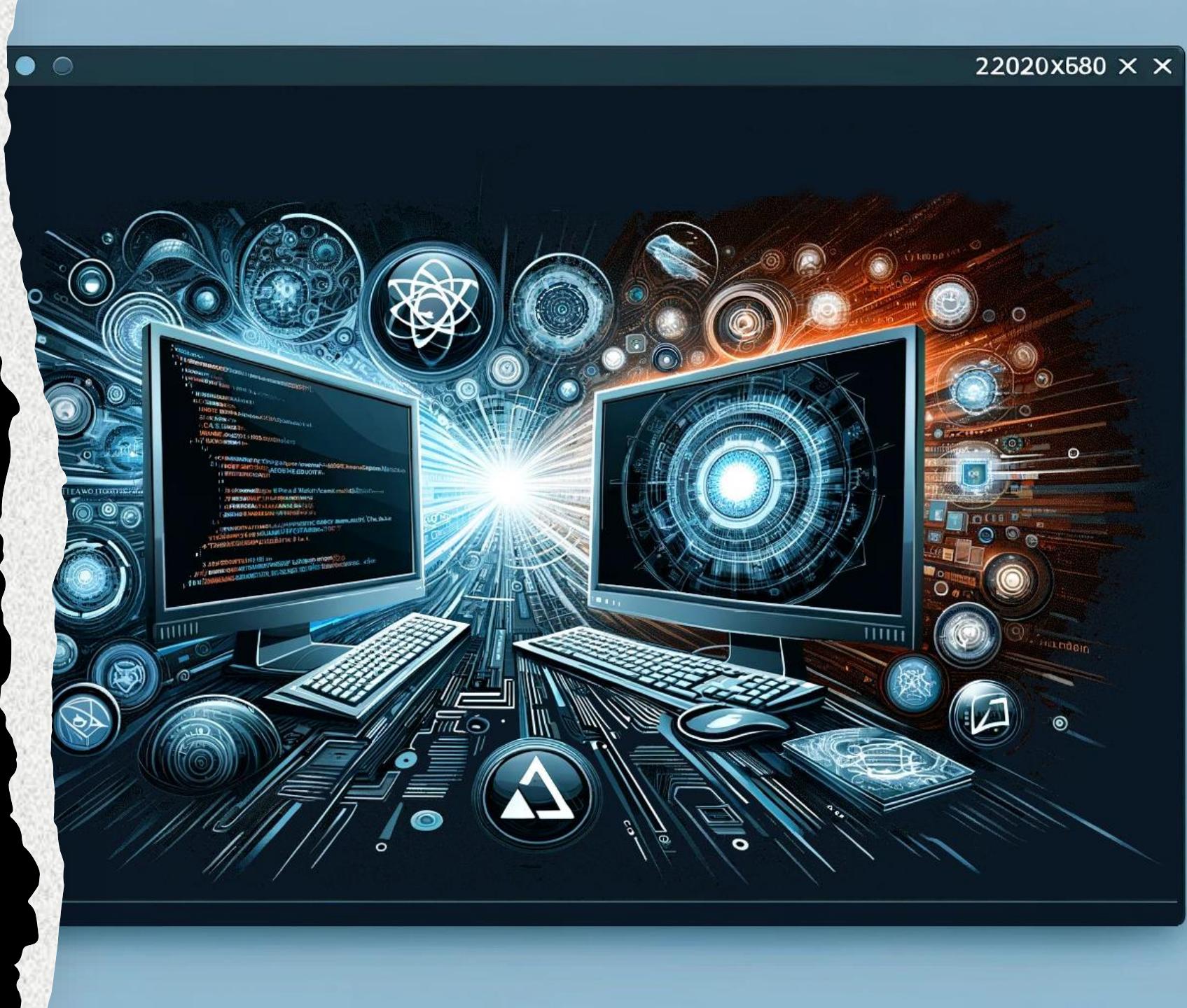
Utilizzato per la sua affidabilità e capacità di gestire grandi volumi di dati.

### **Sistemi Legacy**

Fondamentale nel mantenimento e nell'aggiornamento di sistemi legacy in molte aziende.

*Java: Da Protagonista in Android e Scienza a Custode dei Sistemi Legacy*

# Ambiente di SVILUPPO



# Configurare l'Ambiente di Sviluppo Java

22020x680 × ×

## *Primi Passi Pratici*

- **Installazione del JDK (Java Development Kit)**
- **Scelta e installazione di un IDE ( Eclipse, IntelliJ IDEA, VS Code, ...)**
- **Creazione del primo progetto Java**

*Java trova applicazione in una grande varietà di contesti.*

# Configurare l'Ambiente Java con Eclipse

220x680 × ×

## *Passaggi per l'Installazione e la Configurazione*

### **Installazione JDK (Java Development Kit)**

- Scarica l'ultima versione del JDK da Oracle o AdoptOpenJDK.
- Segui le istruzioni di installazione per il tuo sistema operativo.

### **Verifica dell'Installazione JDK**

- Apri il prompt dei comandi e digita `java -version` per confermare l'installazione.

### **Scarica e Installa Eclipse**

- Visita il sito Eclipse Downloads e scarica l'installer di Eclipse IDE per sviluppatori Java.
- Esegui l'installer e seleziona "Eclipse IDE for Java Developers".

### **Configurazione di Eclipse**

- Al primo avvio, imposta la directory di lavoro (workspace).
- Configura il JDK in Eclipse andando su 'Window' > 'Preferences' > 'Java' > 'Installed JREs'.

# *Il primo programma in JAVA*

22020x680 × ×

## *Hello World*

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello, World!");  
    }  
}
```

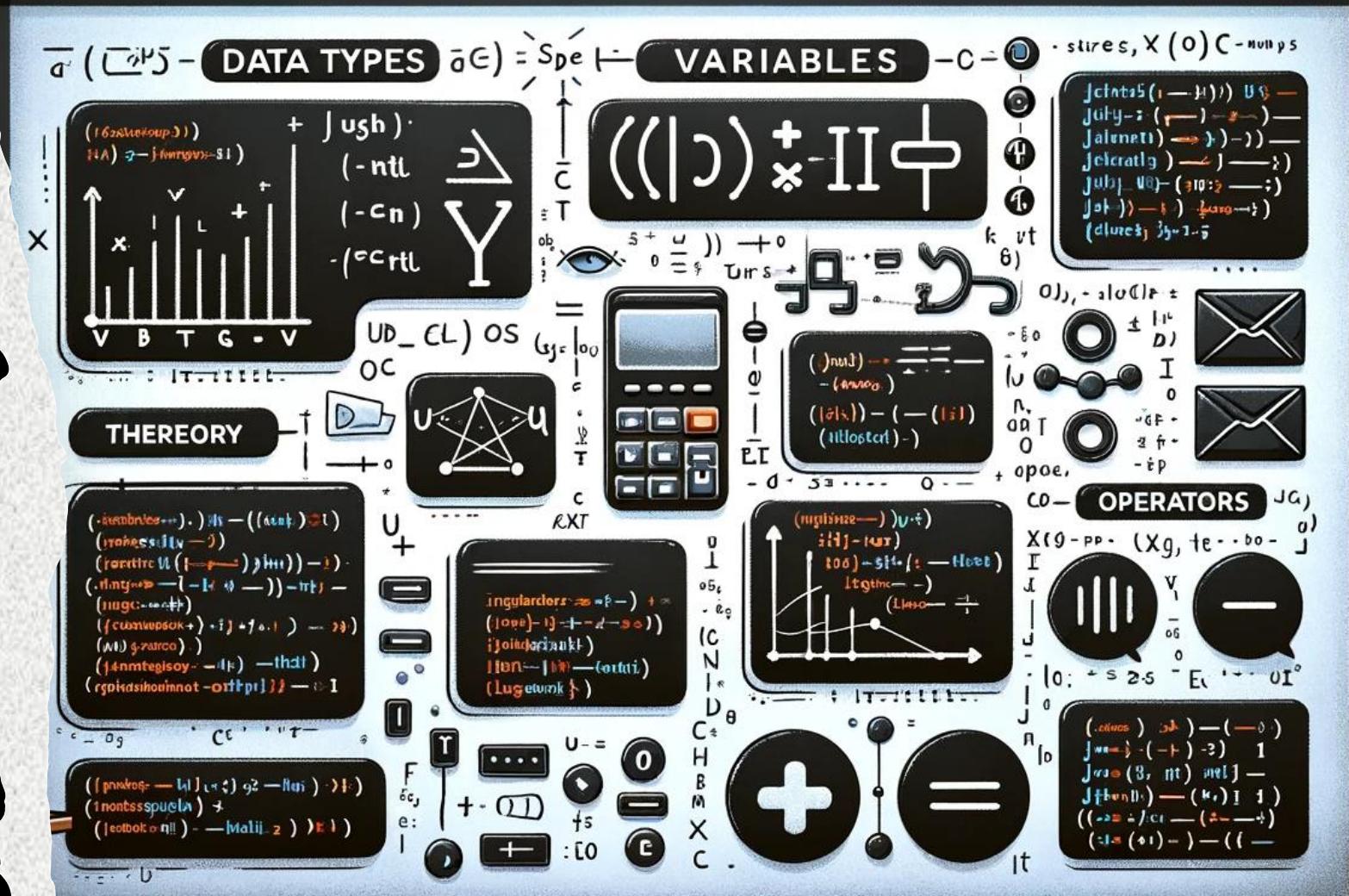
# Case text

22020x680 × ×

## In che modo scrivere i deversi identificatori

- **lowercase:** ([project, artifact, group\\_id, keyword java](#)) tutto minuscolo, tutto attaccato: *corsojava, helloworld,...*
- **UPPERCASE:** tutto maiuscolo, tutto attaccato: *CORSOJAVA, HELLOWORLD,...*
- **UPPER\_SNAKE\_CASE:** ([costanti JAVA](#)) tutto maiuscolo con \_ (underscore) come separatore: *CORSO\_JAVA, HELLO\_WORLD*
- **snake\_case:** ([database: campi, tavelle](#)) tutto minuscolo con \_ (underscore) come separatore: *corso\_java, hello\_world*
- **kebab-case:** ([url, uri](#)) tutto minuscolo utilizzando - (trattino) come separatore: *corso-java, hello-world*
- **PascalCase:** ([nomi di Classi Java](#)) Ogni parola che inizia con la maiuscola: *CorsoJava, HelloWorld*
- **camelCase:** ([identificatori: variabili, metodi, oggetti](#)) : Come pascal case ma con SOLO LA prima lettera SEMPRE MINUSCOLA: *corsoJava, helloWorld*

# Tipi di Dati Variabili Operatori



# Tipi di Dati Primitivi

## *La Base della Programmazione in Java*

### **int (4 byte)**

Usato per valori interi. Esempio: **int eta = 30;**

### **double (8 byte)**

Per numeri in virgola mobile, specialmente per calcoli precisi. Es: **double coeff= 31.5373133;**

### **char (2 byte)**

Rappresenta singoli caratteri. Esempio: **char lettera = 'A';**

### **boolean (1 byte)**

Per valori veri o falsi, utili in condizioni e cicli. Esempio: **boolean isOnline = true;**

### **Altri Tipi Primitivi: byte, short, long, float**

ciascuno con un uso specifico a seconda della precisione e della dimensione dei dati richiesti.

*Conoscere i tipi di dati primitivi è essenziale per scrivere codice efficace e efficiente in Java.*

# Dichiarazione di Variabili

## Nomi e Tipi

**Sintassi di Base:** *tipo nomeVariabile = valore;*

## Esempi:

- int numero = 5;
- double temperatura = 36.5;
- char gruppo = 'B';
- boolean isQualified = false;

## Naming convention

Usa il **camelCase**, inizia con lettere, evita numeri e caratteri speciali all'inizio.

## Importanza di Nomi Significativi

Scegli nomi che descrivono il significato della variabile per migliorare la leggibilità del codice.

*Una variabile ben nominata e dichiarata è un passo verso codice chiaro e manutenibile*

# Operatori Aritmetici

## Calcoli e Operazioni Matematiche

### Operatori Principali:

- Addizione +
- Sottrazione -
- Moltiplicazione \*
- Divisione /
- Modulo % (resto della divisione)

### Esempi di Utilizzo:

- *int somma = 5 + 5;*
- *double risultato = 20.0 / 3.0;*

### Precedenza degli Operatori:

*Parentesi > Moltiplicazione/Divisione > Addizione/Sottrazione.*

### Uso del Modulo:

Utile per determinare se un numero è pari o dispari, per cicli, ecc.

# Operatori Logici

## Fondamenti della Logica Booleana

### Operatori Logici Principali:

- **AND (&&)**

Ritorna true se entrambe le espressioni booleane sono vere. Esempio: *condizione1 && condizione2*.

- **OR (||)**

Ritorna true se almeno una delle espressioni booleane è vera. Esempio: *condizione1 || condizione2*.

- **NOT (!)**

Inverte il valore di una espressione boolean. *!condizione* diventa *true* se condiz. è *false*, e viceversa.

### Combinazione di Operatori:

- Gli operatori logici possono essere combinati per formare espressioni complesse.

Esempio: *if ((a > b) && (a > 0)) {...}*.

### Precedenza degli Operatori:

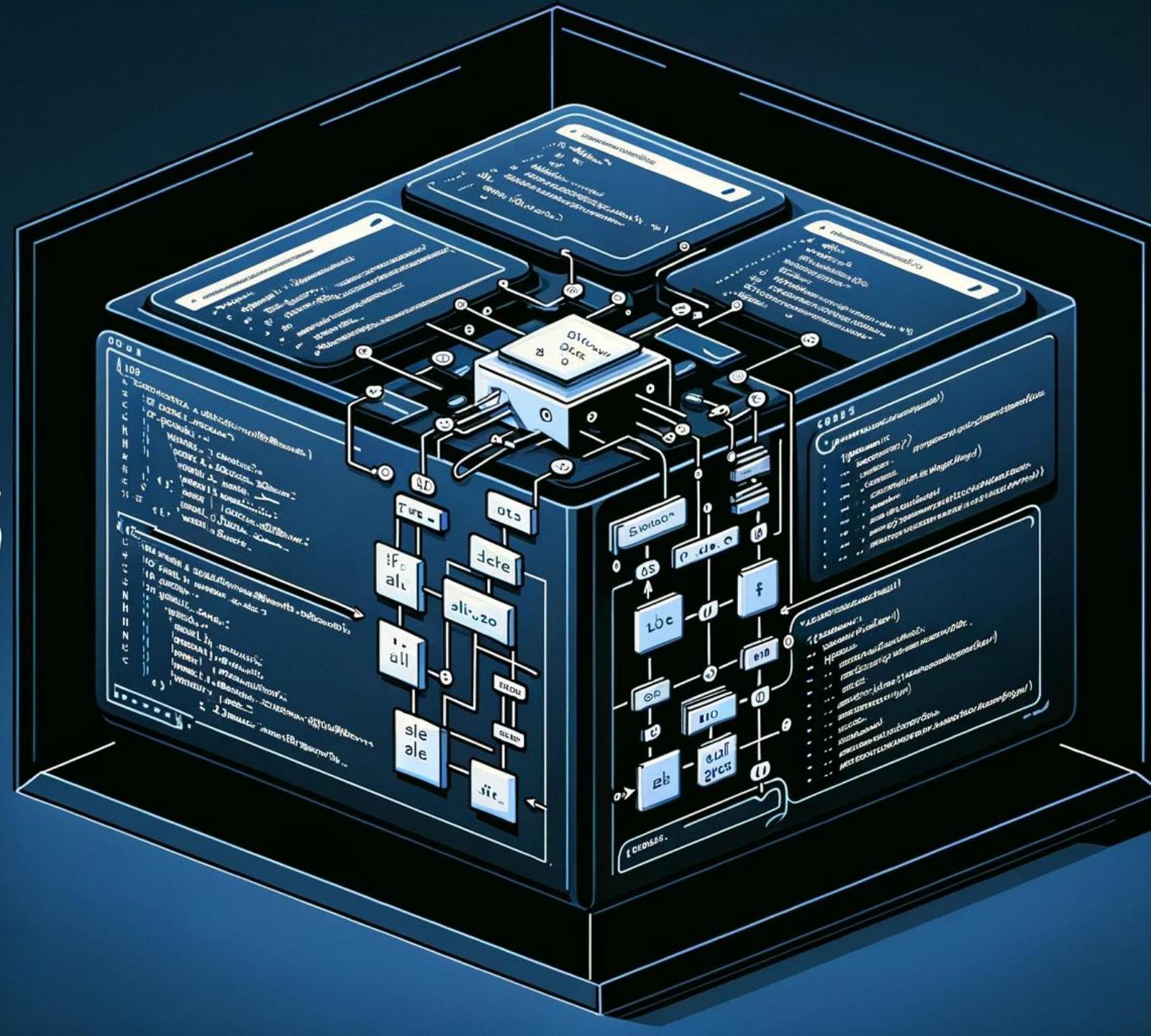
- L'ordine di valutazione è !, poi **&&**, e infine **||**.

• Usare parentesi per : *if (!(condizione1 || condizione2) && condizione3) {...}*.

### Short-Circuiting:

- **&&** e **||** eseguono il "short-circuit". Se la prima condizione di **&&** è false, la seconda non viene valutata.  
Analogamente, se la prima condizione di **||** è true, la seconda non viene valutata.

# Costrutti condizionali



# Istruzione Condizionale *if*

*Prendere Decisioni nel Codice*

**Sintassi dell'istruzione if:**

```
if (condizione) { ... }  
else { ... }
```

**Esempio:** verifica che l'utente sia maggiorenne

[..]

```
System.out.println("Inserisci la tua età:");  
int eta = scanner.nextInt();  
  
if (eta >= 18) {  
    System.out.println("Accesso consentito. Sei maggiorenne.");  
}  
else {  
    System.out.println("Accesso negato. Non sei maggiorenne.");  
}
```

# L'istruzione *else if*

*Gestione di multiple condizioni*

**Sintassi dell'istruzione if:**

```
if (condizione) { ... }
else if { ... }
else { ... }
```

**Esempio:** verifica grado in base al punteggio

```
int punteggio = 75;
if (punteggio >= 90) {
    System.out.println("Grado: A");
} else if (punteggio >= 80) {
    System.out.println("Grado: B");
} else if (punteggio >= 70) {
    System.out.println("Grado: C");
} else {
    System.out.println("Grado: D o inferiore");
}
```

# L'istruzione switch in dettaglio

*Uso dell'istruzione switch*

**Sintassi dell'istruzione if:**

```
switch (variabile) {  
    case X1:  
        ... break;  
    case X2:  
        ... ;  
    ...  
    default:  
}
```

**Applicazioni:**



# Esempio con switch

Verifica l'esito dell'esame in base al grado

```
char grado = 'B';
switch (grado) {
    case 'A':
        System.out.println("Eccellente");
        break;
    case 'B':
    case 'C':
        System.out.println("Ben fatto");
        break;
    case 'D':
        System.out.println("Passato");
        break;
    case 'F':
        System.out.println("Meglio provare di nuovo");
        break;
    default:
        System.out.println("Grado non valido");
}
```

# L'istruzione switch

*Un'alternativa a più if-else*

## ***Quando Usare switch***

- **Per Semplicità e Leggibilità:** Quando si hanno molteplici condizioni basate sullo stesso valore o espressione. switch rende il codice più pulito e facile da leggere rispetto a molteplici if-else.
- **Per Valori Specifici:** switch è ideale quando si devono confrontare molte condizioni con valori specifici, come numeri o stringhe.
- **Per Sostituire Catene di if-else:** Se il codice contiene una lunga serie di if-else che confrontano lo stesso valore, switch può essere una soluzione più elegante e gestibile.

## ***Perché Usare switch:***

- **Efficienza:** In alcuni casi, switch può essere più efficiente delle corrispondenti istruzioni if-else, specialmente con un grande numero di casi.
- **Migliore Organizzazione del Codice:** Aiuta a organizzare il codice in maniera strutturata, rendendo più semplice vedere tutti i possibili percorsi e risultati.
- **Facilità di Aggiunta di Nuovi Casi:** Aggiungere nuovi casi in un switch è generalmente più semplice e richiede meno modifiche al codice rispetto all'aggiunta di nuovi if-else.

# **Switch come espressione**

*Ritorno di valori in uno stile moderno*

## **switch come Espressione:**

- A partire da Java 12, switch può essere utilizzato come un'espressione che restituisce un valore.
- Ciò permette di assegnare il risultato di un switch direttamente a una variabile.

## **Uso di yield:**

- All'interno di un blocco **case**, **yield** restituisce un valore dalla espressione **switch**.
- Sostituisce il tradizionale uso di break con un valore da restituire.

# Input da tastiera con Scanner

*Leggere l'Input da Console*

## Cosa è Scanner

Scanner è una classe in Java utilizzata per ottenere l'input dell'utente.

Fa parte del pacchetto `java.util` e fornisce metodi per leggere tipi di dati diversi (stringhe, numeri, ecc.) dalla console.

**Esempio di inizializzazione:** `Scanner input = new Scanner(System.in);`

## Metodi della Classe Scanner:

- `next()`: Legge il prossimo token di input come stringa. Utile per ottenere singole parole.
- `nextLine()`: Legge una riga intera di input. Utile per ottenere frasi o stringhe contenenti spazi.
- `nextInt()`, `nextDouble()`: Leggono il prossimo token di input come intero e come decimale

## Perché Usare Scanner:

Permette agli sviluppatori di creare applicazioni interattive che possono ricevere input direttamente dall'utente.

# Java SQL

## Seconda parte: JAVA Fondamenti

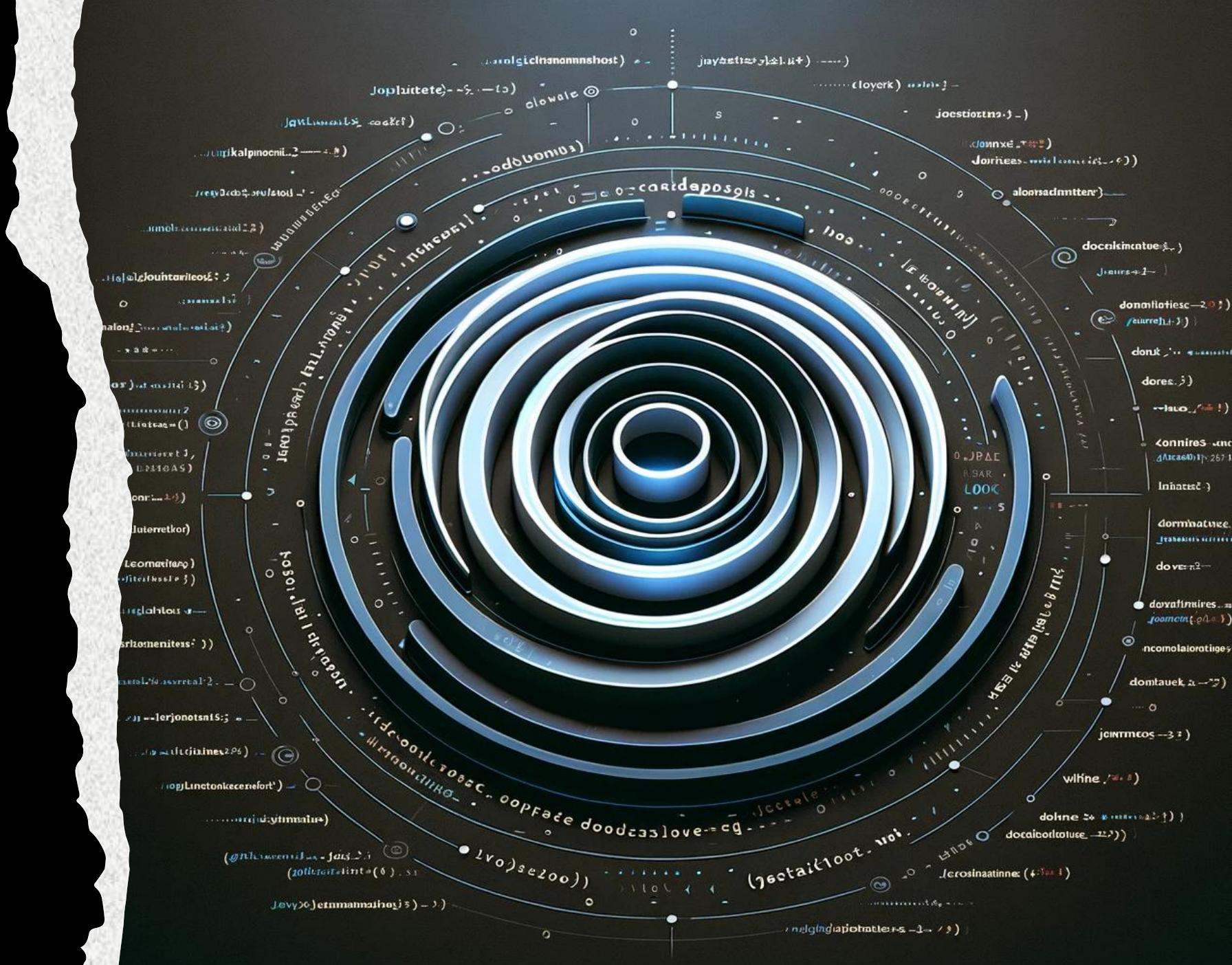
Giorno 12: 15.05.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer



# Cicli



# Il ciclo **for**

*Iterazione Controllata in Java*

**Sintassi del Ciclo for:**

*for (**inizializzazione**; **condizione**; **incremento**) { ... }*

**Esempi di Utilizzo:**

Iterazione su array, generazione di sequenze numeriche, ecc.

**Variazioni:**

Ciclo for tradizionale, ciclo "for-each" per array e collezioni.

# Il ciclo **while** e **do while**

*Iterazione Basata su Condizione*

**Sintassi del ciclo *while*:**

*while (condizione) { ... }*

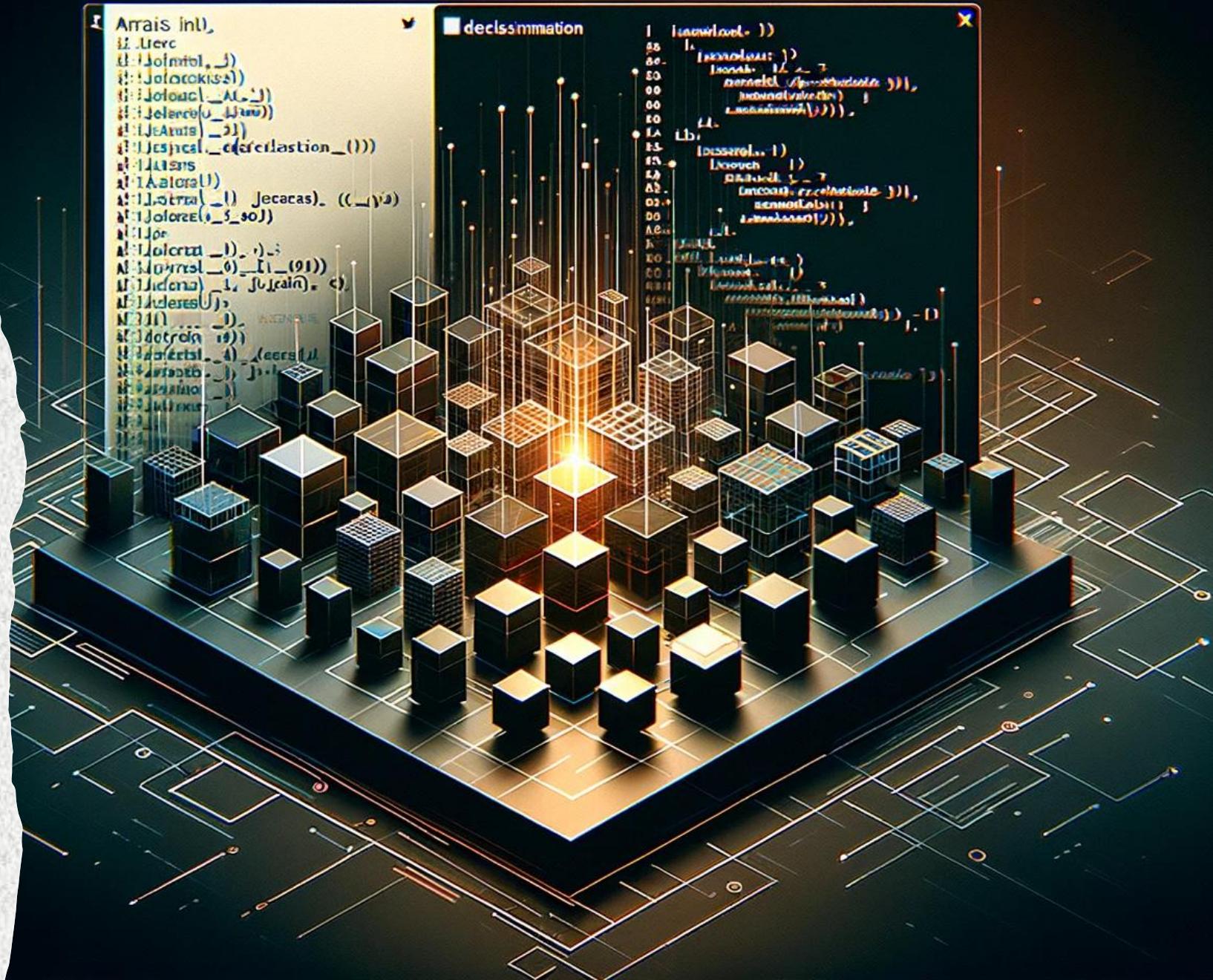
**Sintassi del ciclo *do-while*:**

*do { ... } while (condizione);*

**Quando usare *while* vs *do-while*:**

- **while:** per attese condizionate, iterazioni su input dinamici.
- **do-while:** Cicli che richiedono almeno un'esecuzione del blocco di codice.

# Array di tipo base



# Introduzione agli array

*Strutture di dati fondamentali*

## Definizione

Un array è una struttura di dati che contiene un numero **fisso** di valori dello stesso tipo.

## Caratteristiche

- **Dimensione Fissa:** La dimensione di un array è definita al momento della sua creazione e non può essere modificata.
- **Accesso ad Indice:** Gli elementi in un array sono accessibili tramite un indice numerico.

## Dichiarazione e Inizializzazione

- **Dichiarazione:** `int[] mioArray;`
- **Inizializzazione**

`mioArray = new int[10];`

`int[] mioArray = {1, 2, 3};`

# Lavorare con gli elementi dell'array

## Accesso e Modifica

### Accesso agli Elementi:

Gli elementi di un array possono essere letti o modificati utilizzando l'indice.

- **Lettura:** `int primoElemento = mioArray[0];`
- **Modifica:** `mioArray[0] = 100;`
- **Zero based:** il primo elemento dell'array ha indice 0 sempre.

**Iterazione su Array:** Uso del ciclo for o for-each per scorrere tutti gli elementi.

- Esempio con **for**: `for (int i = 0; i < mioArray.length; i++) { ... }`
- Esempio con **for-each**: `for (int elemento : mioArray) { ... }`

*L'accesso e la modifica degli elementi di un array sono operazioni fondamentali che permettono un controllo dettagliato sui dati collezionati*

# Array multidimensionale

*Oltre la Singola Dimensione*

## Definizione

Un array multidimensionale è un array di array. Comunemente usato per rappresentare matrici o griglie.

## Dichiarazione e Inizializzazione:

- **Dichiarazione:** `int[][] matrice;`
- **Inizializzazione:** `matrice = new int[3][3];` o `int[][] matrice = {{1, 2}, {3, 4}};`

## Accesso agli Elementi: Utilizzo di indici multipli.

- `int elemento = matrice[0][1];`

*Gli array multidimensionali ampliano le possibilità di rappresentazione dei dati, permettendo di lavorare con concetti come tavole e griglie in modo intuitivo*

# Limitazioni e alternative

## Oltre la Singola Dimensione

### Limitazioni:

- **Dimensione Fissa:** Una volta creato, la dimensione dell'array non può essere modificata.
- **Tipo Omogeneo:** Tutti gli elementi devono essere dello stesso tipo.

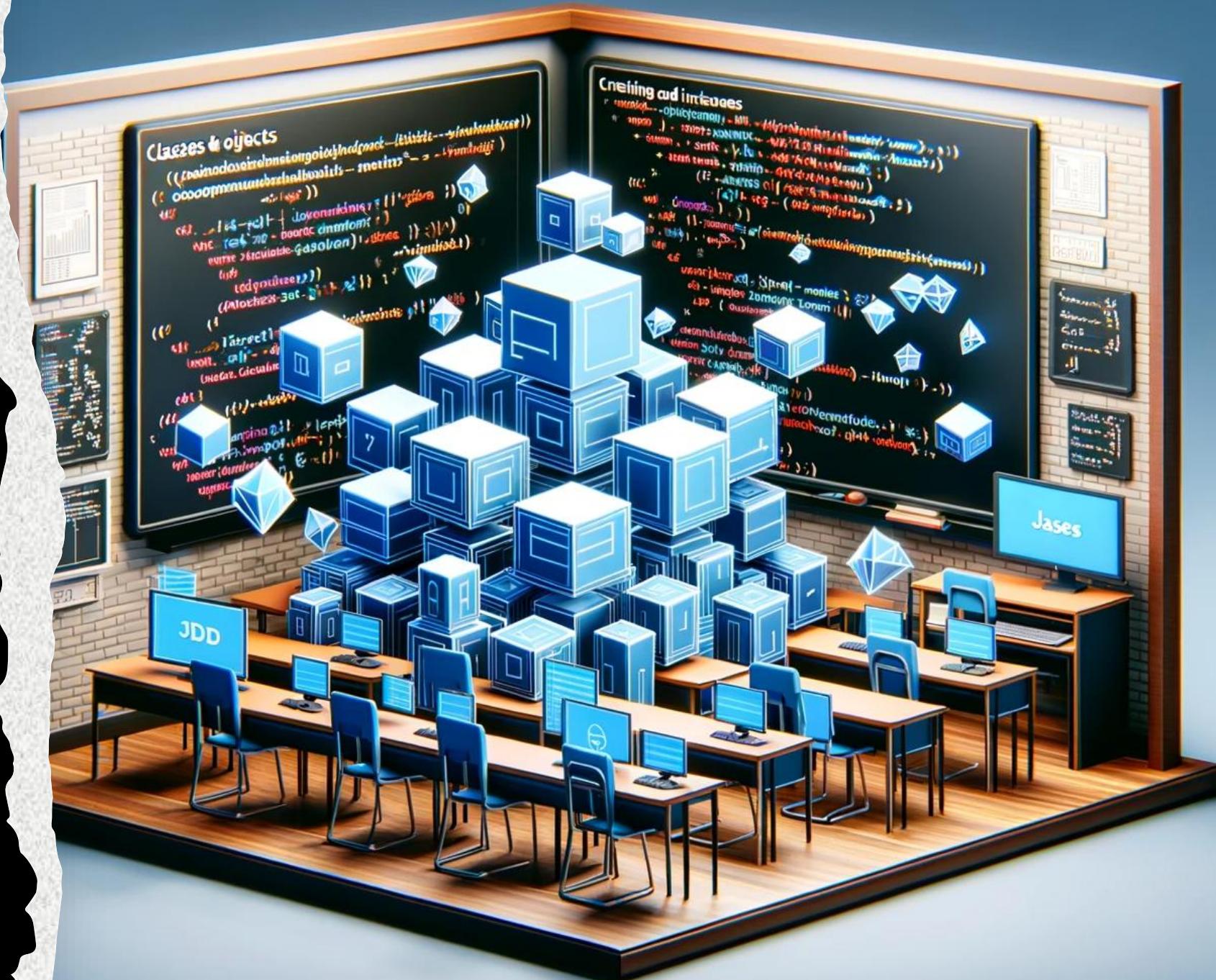
### Alternative agli Array:

- **ArrayList:** Dimensione dinamica, permette l'aggiunta e la rimozione degli elementi.
- **LinkedList:** Migliore in termini di prestazioni per l'inserimento e la rimozione frequenti.

### Uso: (*Vedi ultima parte del corso Java base*)

*Gli array multidimensionali ampliano le possibilità di rappresentazione dei dati, permettendo di lavorare con concetti come tavole e griglie in modo intuitivo*

# Classi ed Oggetti



# Classi e oggetti in Java

*Fondamenti della Programmazione Orientata agli Oggetti (OOP)*

## Classe

Un modello o un'astrazione che definisce le caratteristiche (stato) e i comportamenti (metodi) degli oggetti.

## Oggetto

Una istanza concreta di una classe con uno stato unico.

*Le classi e gli oggetti, basi della programmazione OOP, rendono Java un linguaggio potente e flessibile*

# Struttura di una Classe in Java

## Campi, Metodi e Costruttori

### Campi (o Attributi):

Variabili che detengono lo stato di un oggetto.

Esempio: `public String nome;`

### Metodi:

Funzioni che definiscono il comportamento di un oggetto.

Esempio: `public void mostraNome() { System.out.println(nome); }`

### Costruttori:

Utilizzati per inizializzare nuovi oggetti.

Esempio: `public Studente(String nome) { this.nome = nome; }`

### Access Modifiers:

Controllano la visibilità: `public, private, protected`.

*Le classi e gli oggetti, basi della programmazione OOP, rendono Java un linguaggio potente e flessibile*

# *Creazione di una Classe Semplice*

22020x680 × ×

## *Esempio Pratico*

```
public class Studente {  
    public String nome;  
    public int eta;  
    Studente(String nome, int eta) {  
        this.nome = nome;  
        this.eta = eta;  
    }  
  
    public void mostraInfo() {  
        System.out.println("Nome: " + this.nome + ", Età: " + this.eta);  
    }  
}
```

# *Creazione e Utilizzo di Oggetti*

22020x680 × ×

## *Istanziare Oggetti dalla Classe*

```
public class TestStudente {  
    public static void main(String[] args) {  
        Studente s1 = new Studente("Mario Rossi", 20);  
        s1.mostraInfo();  
  
        Studente s2 = new Studente("Luigi Bianchi", 22);  
        s2.mostraInfo();  
    }  
}
```

# Incapsulamento



# Incapsulamento: Protezione dei Dati

*Sicurezza e Integrità dei Dati*

## Definizione di Incapsulamento:

- Incapsulamento è una pratica di nascondimento dei dettagli di implementazione.
- Controllo dell'accesso ai dati di un oggetto tramite metodi getter e setter.

## Vantaggi dell'Incapsulamento:

- Protegge l'integrità dell'oggetto impedendo accessi esterni non autorizzati ai suoi campi.
- Fornisce flessibilità per modificare l'implementazione interna senza impattare gli utenti dell'oggetto.

## Implementazione in Java

- Dichiarazione dei campi come private.
- Fornire metodi pubblici get e set per accedere e modificare i campi.

*L'incapsulamento garantisce la sicurezza dei dati e la modularità del codice*

# Getter e Setter: Strumenti di Incapsulamento

*Gestire l'Accesso ai Dati*

## Ruolo dei Getter e Setter

- Metodi getter e setter consentono di leggere e modificare i valori dei campi privati di una classe.
- Mantengono l'integrità dei dati, permettendo di validare gli input prima di modificarli.

## Esempi di Implementazione

- Esempio di un setter che controlla la validità dell'input prima di assegnarlo a un campo.
- Esempio di un getter che restituisce una copia di un oggetto complesso per prevenire modifiche indesiderate.

## Principi da Seguire

- Non esporre i dettagli interni tramite getter e setter a meno che sia necessario.
- Utilizzare questi metodi per implementare controlli e logiche di business.

# Java SQL

## Seconda parte: JAVA Fondamenti

Giorno 13: 16.05.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer



# Proprietà e metodi static

*Condividere metodi e proprietà fra tutte le istanze*

- I metodi e le proprietà statiche sono utili quando si desidera che una proprietà o un metodo sia condiviso tra tutte le istanze di una classe, senza dover creare un oggetto per accedervi
- Va premessa la keyword **static** alla proprietà o al metodo
- Un metodo statico può accedere solo a variabili statiche e può chiamare solo altri metodi statici della stessa classe o istanziare altre classi.

```
// proprietà statica
public class Studente {
    public static String nomeScuola;
    //... codice
}

// accesso alla proprietà
Studente.nomeScuola = 'ITI Vittorio Emanuele'
```

```
// proprietà statica
public class Studente {
    public static String getScuola() {
        //... codice
    }

    // accesso alla proprietà
    Studente.getScuola()
```

# Overload di Metodi e Costruttori in Java

*Diverse versioni dello stesso metodo*

- L'overload di metodi e costruttori avviene definendo nuovamente il metodo o il costruttore ma con differenti parametri (numero, tipo, e ordine).
- Si definisce **Firma**, la definizione del metodo con i parametri e il valore restituito
- La restituzione di tipo diverso non è sufficiente per differenziare due metodi.
- L'overload dei costruttori fornisce diverse modalità di inizializzazione degli oggetti

```
// overload di metodo

public double add (double a, double b) {
    return a + b;
}

public int add (int a, int b) {
    return a + b;
}
```

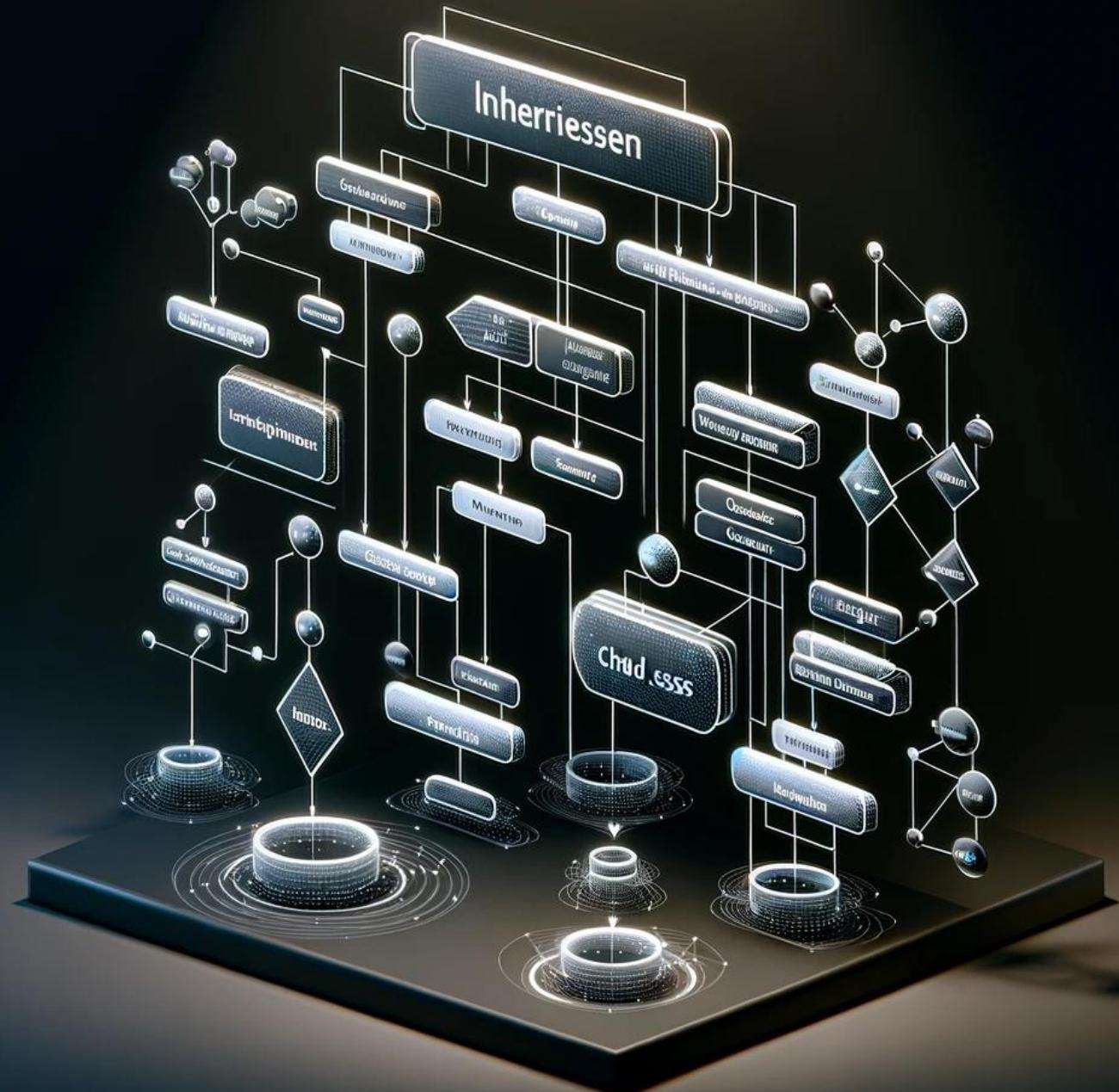
```
public class Person {      // overload di costruttore
    private String name;
    private int age;

    public Person() {
        this.name = "Unknown";
        this.age = 0;
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

*L'overload permette diverse modalità di esecuzione e/o inizializzazione a seconda dei parametri forniti*

# Ereditarietà



# Implementazione dell'Ereditarietà

## *Creazione di Sottoclassi*

## **Creazione di Sottoclassi:**

- Una sottoclassa estende la superclasse usando la parola chiave ***extends***.
  - >> Eredita automaticamente TUTTI i metodi e gli attributi pubblici e protetti. <<

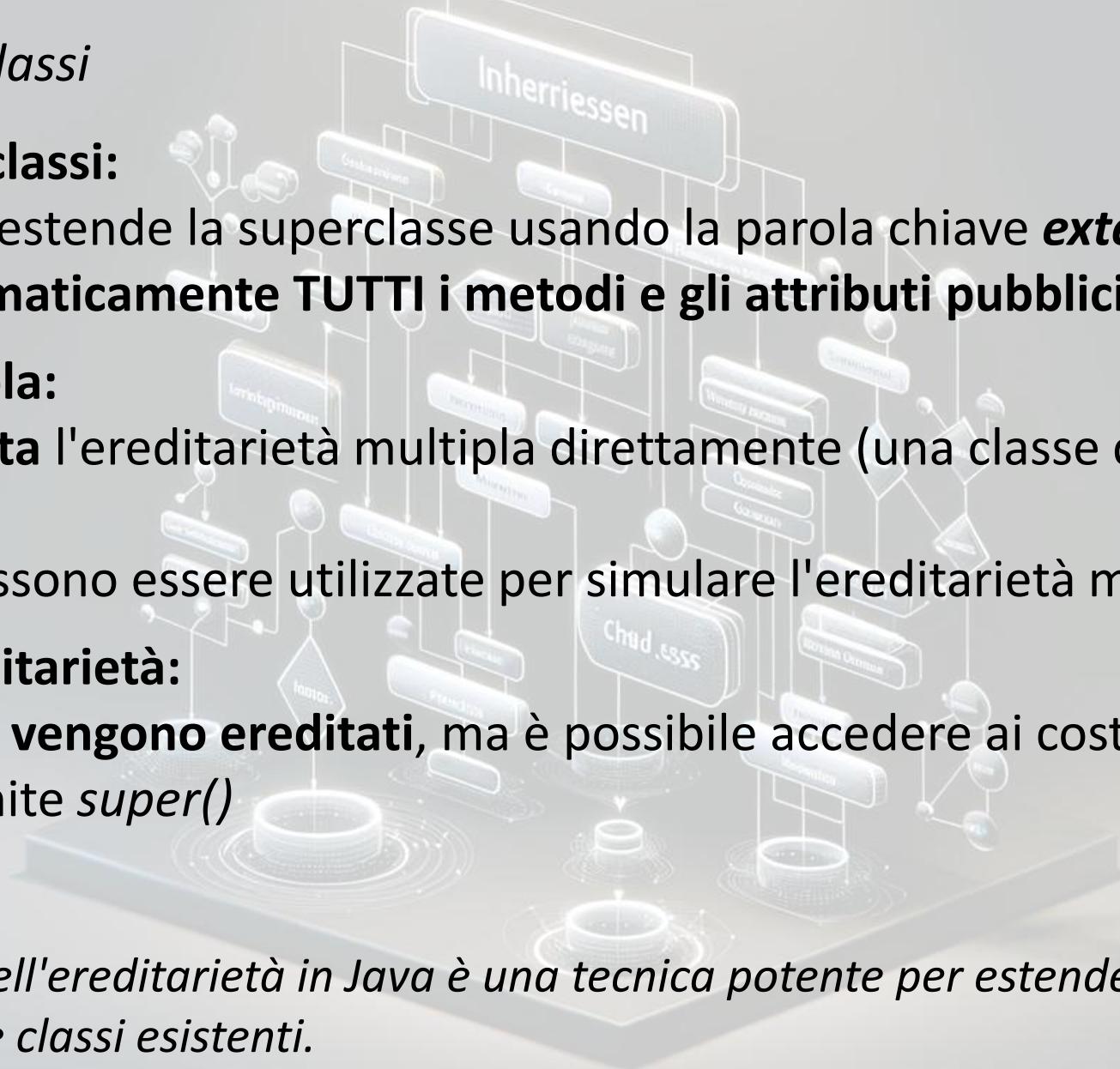
## Ereditarietà Multipla:

- Java **non supporta** l'ereditarietà multipla direttamente (una classe che estende più di una classe).
  - Le **interfacce** possono essere utilizzate per simulare l'ereditarietà multipla.

## **Costruttori ed Ereditarietà:**

- I **costruttori non vengono ereditati**, ma è possibile accedere ai costruttori della superclasse tramite *super()*

*L'implementazione dell'ereditarietà in Java è una tecnica potente per estendere e personalizzare il comportamento delle classi esistenti.*



# Override dei metodi

*Personalizzazione del Comportamento Ereditato*

## Concetto di Override

- Override permette di definire una nuova implementazione di un metodo ereditato nella sottoclasse.
- Essenziale per modificare o estendere il comportamento ereditato

## Regole e Buone Pratiche

- Utilizzare l'annotazione `@Override`.
- Il metodo sovrascritto deve avere **la stessa firma** del metodo nella superclasse.
- Overriding non va confuso con Overloading

*L'implementazione dell'ereditarietà in Java è una tecnica potente per estendere e personalizzare il comportamento delle classi esistenti.*

# Uso di *super()*

## Navigare nella Gerarchia di Ereditarietà

### Definizione di *super()*

- *super()* è una parola chiave usata per riferirsi direttamente al costruttore della superclasse.
- Viene utilizzata per invocare costruttori della superclasse all'interno di una sottoclasse.

### Uso di *super()* nei Costruttori

- Invocare il costruttore della superclasse quando una sottoclasse viene istanziata.
- **È il primo comando in un costruttore di una sottoclasse.**
- Esempio: *super(parametri)*; chiama il costruttore della superclasse con i parametri specificati.

### Uso di *super* nei Metodi

- Utilizzato per fare riferimento a metodi sovrascritti nella superclasse.
- Esempio: *super.metodoSuperclasse()*; invoca un metodo definito nella superclasse.

### Regole e Buone Pratiche

- il costruttore, se non dichiarata esplicitamente una chiamata a *super(...)*, **chiama implicitamente *super()***, quindi la superclasse DEVE avere il costruttore vuoto ();
- Dichiare esplicitamente un costruttore senza argomenti quindi è una pratica comune per evitare la duplicazione del codice e mantenere la coerenza nella gerarchia delle classi.

# Overriding vs Overloading

*Due Concetti Fondamentali nella Programmazione OOP*

## Overriding (Sovrascrittura):

- Si verifica quando una **sottoclasse** fornisce una specifica implementazione di un metodo già definito nella sua **superclasse**.
- Utilizzato per modificare o migliorare il comportamento esistente.

### Regole:

- **Stessa firma del metodo** (nome, parametri e valore restituito), **classi diverse**.
- Utilizzo dell'annotazione `@Override`.

## Overloading (Sovraccarico):

- Si verifica quando due o più metodi nella **stessa classe** hanno lo stesso nome ma numero o tipo dei parametri diversi (**firme diverse del metodo**).
- Utilizzato per aggiungere più modi per invocare un metodo, basandosi su differenti tipi di input.

**Regole: Parametri diversi del metodo**

# Java SQL

## Seconda parte: JAVA Fondamenti

Giorni 14 e 15: 19-20.05.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer



# Composizione

*Una tecnica di progettazione orientata agli oggetti.*

## Caratteristiche

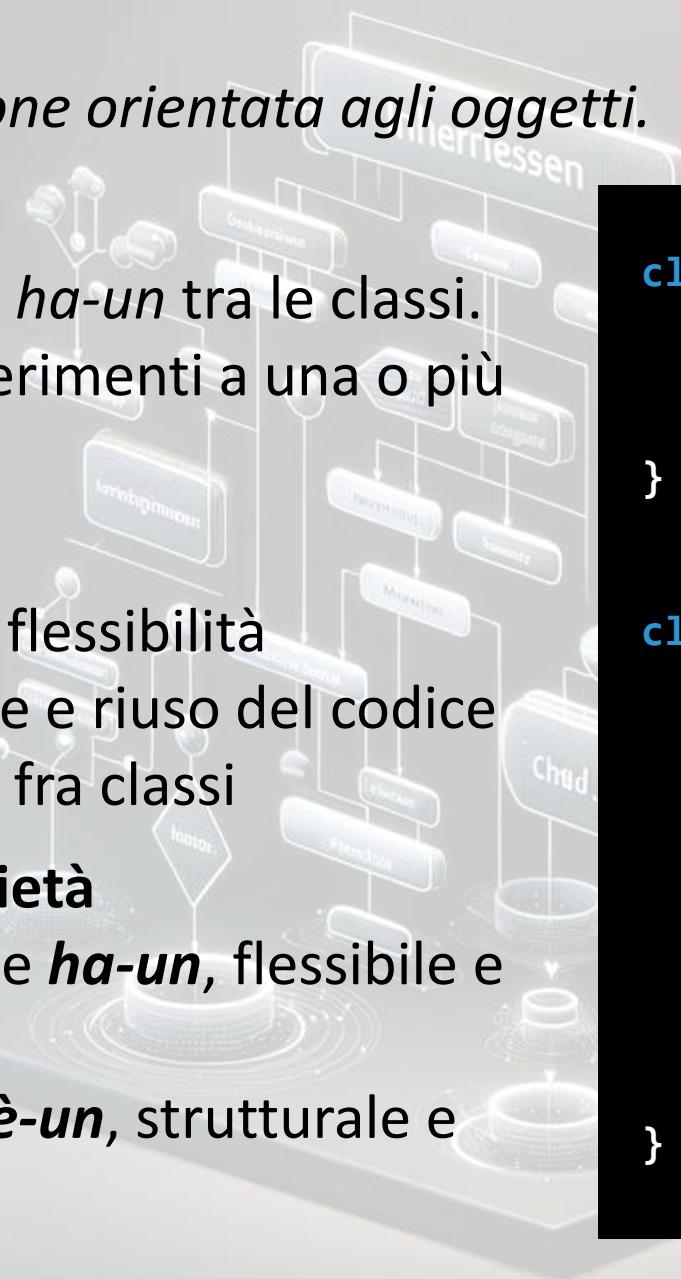
- Definisce una relazione *ha-un* tra le classi.
- Una classe contiene riferimenti a una o più istanze di altre classi.

## Vantaggi

- Maggiore modularità e flessibilità
- Facilità di manutenzione e riuso del codice
- Accoppiamento ridotto fra classi

## Composizione vs Ereditarietà

- Composizione: relazione *ha-un*, flessibile e modulare..
- Ereditarietà: relazione *è-un*, strutturale e rigida.



```
class Motore {  
    public void avvia() {  
        System.out.println("Motore avviato");  
    }  
  
class Auto {  
    private Motore motore;  
    Auto() {  
        this.motore = new Motore();  
    }  
  
    public void avviaAuto() {  
        motore.avvia();  
        System.out.println("Auto avviata");  
    }  
}
```

# Polimorfismo



# Introduzione al Polimorfismo

*Versatilità e Flessibilità nella Programmazione OOP*

## Definizione di Polimorfismo

Capacità di un'entità (ad es., una variabile di tipo superclasse) di **riferirsi agli oggetti di più di una sottoclasse** (*o interfaccia implementata*).

## Polimorfismo di runtime (o tardivo)

il tipo di oggetto (istanza) determina quale implementazione di metodo viene eseguita a runtime (late binding).

## Polimorfismo e Metodi:

- Il polimorfismo si basa sull'override dei metodi.
- Permette di trattare oggetti di differenti sottoclassi in modo uniforme.

*Il polimorfismo è il cuore della programmazione orientata agli oggetti, rendendo il nostro codice più modulare, flessibile e mantenibile*

# Valorizzazione e Casting in Java: Classi e Sottoclassi

*Come valorizzare oggetti di classi diverse e best practices*

La valorizzazione e il casting tra classi e sottoclassi sono concetti fondamentali che sfruttano l'ereditarietà e il polimorfismo per scrivere codice flessibile e riutilizzabile.

**Assegnazione Sottoclasse a SuperClasse (nessun casting necessario)**

```
Veicolo mioVeicolo = new Automobile("Ferrari", "F8", 2);
```

Questo tipo di assegnazione sfrutta il polimorfismo, permettendo di trattare oggetti di sottoclassi come se fossero della superclasse.

**Assegnazione SuperClasse a Sottoclasse (casting esplicito)**

```
Veicolo veicoloGenerico = new Veicolo("Toyota", "Corolla");
Automobile miaAutomobile = (Automobile) veicoloGenerico;
```

Questo tipo di assegnazione porta a ***ClassCastException*** se l'oggetto non è effettivamente di tipo **Automobile**

*Assegnazione sicura e polimorfismo e uso del casting necessario*

# Valorizzazione e Casting in Java: *instanceof*

*Best Practice per il casting*

Prima di eseguire un casting da superclasse a sottoclasse, è consigliabile verificare il tipo dell'oggetto con *instanceof*

```
if (veicoloGenerico instanceof Automobile) {  
    Automobile miaAutomobile = (Automobile) veicoloGenerico;  
}
```

Questo evita errori a runtime come ClassCastException, garantendo che il casting sia sicuro.

Se trovi necessario fare frequentemente casting da superclasse a sottoclassi, considera di rivedere il design del tuo software.

*Un uso eccessivo di casting può nascondere problemi di design, come una cattiva distribuzione delle responsabilità tra classi*

# Interfacce in Java

*Definizione di Contratti in OOP*

## Definizione di Interfaccia

- Una interfaccia in Java è una struttura che può contenere solo costanti e definizioni di metodi astratti (senza implementazione).
- Fornisce un "contratto" che le classi implementanti devono seguire.

## Caratteristiche delle Interfacce

- Tutti i metodi in un'interfaccia sono implicitamente astratti e pubblici.\*

## Utilizzo delle Interfacce

- Le interfacce sono usate per definire capacità comuni che possono essere condivise da diverse classi.

*\*(Le interfacce possono contenere metodi default e statici da java 8 in poi, vedi parte avanzata del corso)*

*Le interfacce forniscono un mezzo per definire contratti formali, promuovendo una programmazione flessibile e modulare*

# Classi Astratte in Java

*Base per Ereditarietà e Polimorfismo*

## Definizione di Classe Astratta

- Una classe astratta è una classe che non può essere istanziata e che può contenere sia metodi astratti sia metodi con implementazione..

## Caratteristiche delle Classi Astratte

- Possono contenere variabili di stato e metodi concreti.
- Forniscono una base comune per diverse sottoclassi.

## Utilizzo delle Classi Astratte

- Le Classi Astratte sono utilizzate quando diverse classi hanno metodi con implementazioni simili o comuni.

*Le classi astratte agiscono come superclassi parziali, fornendo un framework che altre classi possono estendere e personalizzare*

# Interfacce vs Classi Astratte

*Scegliere la Struttura Giusta*

## Somiglianze

- Entrambe non possono essere istanziate direttamente.
- Servono come un tipo di riferimento per polimorfismo.

## Interfacce

- Non hanno variabili di stato (solo costanti statiche finali).
- Tutti i metodi sono astratti, (*salvo metodi default e statici*).
- Si usano quando diverse classi devono seguire lo stesso *contratto* ma hanno implementazioni completamente diverse

## Classi Astratte

- Possono avere variabili di stato, metodi concreti e/o astratti.
- Forniscono un'implementazione parziale (compresa di costruttori/e)
- Si usano Quando diverse classi hanno un'implementazione di base comune ma anche comportamenti personalizzati.

# Java SQL

## Seconda parte: JAVA Fondamenti

Giorno 16: 21.05.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer



# Java Best Practice



# Principi Fondamentali nella Scrittura del Codice Java

*Chiarezza e Manutenibilità*

## Nomenclatura Intuitiva:

- Usare nomi significativi e descrittivi per classi, metodi e variabili.
- Seguire le convenzioni di Java: metodi e variabili con **camelCase**, classi con **ProperCase**

## Consistenza del Codice:

- Essere consistenti nello stile, nella formattazione e nelle convenzioni di nomenclatura.
- Utilizzare strumenti di formattazione automatica e linee guida per il team.

## Commenti e Documentazione:

- Commentare il codice quando necessario per chiarire la logica complessa.
- Usare Javadoc per documentare le classi e i metodi pubblici.

# Modellazione Efficiente degli Oggetti

*Progettare per Flessibilità e Estensibilità*

## **Principio della Responsabilità Singola:**

- Ogni classe dovrebbe avere una sola responsabilità o scopo.
- Evitare classi "onnipotenti" che gestiscono troppe funzionalità.

## **Uso di Classi Astratte e Interfacce:**

- Sfruttare classi astratte e interfacce per definire modelli comuni e contratti.
- Promuovere il riutilizzo del codice e la flessibilità con l'ereditarietà e il polimorfismo.

## **Incapsulamento e Astrazione:**

- Proteggere i dati interni e concentrarsi sulle interfacce con l'incapsulamento.
- Utilizzare l'astrazione per nascondere la complessità e mostrare solo le funzionalità necessarie.

# Getter e Setter: Strumenti di Incapsulamento

*Gestire l'Accesso ai Dati*

## Ruolo dei Getter e Setter

- Metodi getter e setter consentono di leggere e modificare i valori dei campi privati di una classe.
- Mantengono l'integrità dei dati, permettendo di validare gli input prima di modificarli.

## Esempi di Implementazione

- Esempio di un setter che controlla la validità dell'input prima di assegnarlo a un campo.
- Esempio di un getter che restituisce una copia di un oggetto complesso per prevenire modifiche indesiderate.

## Principi da Seguire

- Non esporre i dettagli interni tramite getter e setter a meno che sia necessario.
- Utilizzare questi metodi per implementare controlli e logiche di business.

# Ottimizzazione e Refactoring

*Migliorare e Mantenere il Codice nel Tempo*

## Refactoring Regolare:

- Rivedere e migliorare continuamente il codice per aumentarne la qualità e l'efficienza.
- Identificare i *code smells* e affrontarli sistematicamente.

## Ottimizzazione delle Prestazioni:

- Monitorare e ottimizzare le prestazioni dove necessario.
- Evitare ottimizzazioni premature; concentrarsi prima sulla chiarezza e sulla manutenibilità.

# Principi SOLID per una Codifica OOP Efficace

## *Fondamenti di OOP Avanzato*

### **Single Responsibility**

Una classe dovrebbe avere una sola ragione per cambiare.

### **Open/Closed**

Le entità software dovrebbero essere aperte all'estensione, ma chiuse alle modifiche

### **Liskov Substitution**

Le sottoclassi dovrebbero essere sostituibili con le loro superclassi.

### **Interface Segregation**

Creare interfacce specifiche piuttosto che uniche interfacce generali.

### **Dependency Inversion**

Dipendere da astrazioni, non da implementazioni concrete.

*I principi SOLID forniscono una guida per costruire software ben progettato, mantenibile e scalabile.*

# Principio di Single Responsibility (SRP)

*Unica Responsabilità per Classe*

## Definizione di SRP:

Ogni classe dovrebbe avere solo una ragione per cambiare, ovvero una singola responsabilità.

## Importanza di SRP

- Semplifica la manutenzione del codice.
- Riduce l'impattabilità delle modifiche e facilita la comprensione della classe.

## Applicazione Pratica

- Evitare classi che gestiscono molteplici funzioni.
- Separazione delle responsabilità in classi più piccole e focalizzate.

*SRP promuove la modularità e la manutenibilità, rendendo il codice più flessibile ai cambiamenti.*

# Principio Open/Closed (OCP)

*Aperto all'Estensione, Chiuso alle Modifiche*

## Definizione di OCP

Le entità software (classi, moduli, funzioni, ecc.) dovrebbero essere aperte per l'estensione, ma chiuse per la modifica.

## Realizzazione di OCP

- Estendere il comportamento esistente tramite ereditarietà e composizione piuttosto che modificare il codice esistente.
- Uso di interfacce e classi astratte per permettere variazioni nei comportamenti.

## Benefici

- Promuove la stabilità del codice e la flessibilità nell'aggiungere funzionalità.

*Adottando OCP, il codice diventa più robusto e adattabile ai cambiamenti futuri*

# Principio di Liskov Substitution (LSP)

*Sostituibilità delle Sottoclassi*

## Definizione di LSP:

Le sottoclassi dovrebbero essere sostituibili con le loro superclassi senza influire sul comportamento del programma.

## Significato di LSP

- Le sottoclassi non dovrebbero sovrascrivere i metodi esistenti in modo da alterarne il comportamento atteso.
- Ogni sottoclasse dovrebbe aderire ai contratti e alle aspettative della superclasse.

## Esempi e Considerazioni

- Prevenzione di sovrascritture che rompono la funzionalità della superclasse.
- Esempi di violazioni e conformità a LSP (Quadrato/Rettangolo e Dispositivi di Output)

*LSP assicura che l'ereditarietà sia usata in modo da mantenere l'integrità del design originale.*

# Principio di Interface Segregation (ISP)

## *Specializzazione delle Interfacce*

### **Definizione di ISP:**

Le classi non dovrebbero essere costretti a dipendere da interfacce che non utilizzano.

### **Significato di LSP**

- Creare interfacce specifiche e mirate piuttosto che un'unica interfaccia generale.
- Separare le interfacce in base alle funzionalità per evitare dipendenze inutili.

### **Esempi e Considerazioni**

- Riduce l'accoppiamento e migliora la coesione del codice.
- Migliora la flessibilità e la facilità di manutenzione.

*ISP promuove la creazione di interfacce focalizzate, aumentando la modularità del software.*

# Principio di Dependancy Inversion (DIP)

*Inversione delle dipendenze*

**Definizione di DIP:**

Il DIP afferma che i moduli di alto livello non devono dipendere da quelli di basso livello; entrambi devono dipendere da astrazioni.

**Significato di DIP:**

Promuove software flessibile e manutenibile, disaccoppiando le dipendenze tra componenti.

**Esempio:** Un modulo di report che usa un'interfaccia DataAccess piuttosto che un'implementazione diretta, facilitando la sostituzione delle fonti di dati.

*DIP riduce l'accoppiamento, aumenta la modularità e la testabilità.*

# Java React

## Seconda parte: JAVA Fondamenti

Giorno 17: 22.05.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer



# Introduzione alle Eccezioni

## *Il concetto di eccezione in Java*

In Java, le eccezioni sono eventi che disturbano il normale flusso di esecuzione di un programma.

Si distinguono principalmente in due categorie: eccezioni controllate e non controllate.

**Le eccezioni controllate** devono essere gestite nel codice attraverso la clausola *try-catch* o dichiarate nel metodo tramite la keyword *throws*.

**Le eccezioni non controllate**, invece, includono errori e runtime exceptions che possono essere lanciati durante l'esecuzione del programma senza essere necessariamente catturati o dichiarati.

*Introduzione al sistema di gestione delle eccezioni in Java.*

# Gestione delle Eccezioni

*Try, Catch, Finally*

## Il blocco try /catch

permette di definire un codice che potrebbe lanciare un'eccezione, mentre il blocco catch viene utilizzato per catturare e gestire l'eccezione.

## Il blocco finally

è opzionale, viene eseguito dopo il **try** e **catch**, indipendentemente dal fatto che un'eccezione sia stata lanciata o meno, utilizzato tipicamente per eseguire operazioni di pulizia, come la chiusura di connessioni o il rilascio di risorse.

*Strutture fondamentali per una robusta gestione delle eccezioni.*

# Eccezioni Personalizzate

*Definire eccezioni specifiche*

Java permette di definire eccezioni personalizzate estendendo la classe

***Exception*** (controllate) o ***RuntimeException*** (non controllate).

Questo consente di creare tipi di eccezione **specifici** per le esigenze del proprio programma, migliorando la leggibilità del codice e fornendo errori più specifici ai consumatori del codice.

*Creare eccezioni su misura per le specifiche esigenze del programma.*

# File I/O: Lettura

## *Leggere da file*

Java fornisce diverse classi nel package `java.io` per leggere da file.

**FileReader** e **BufferedReader** sono comunemente usati per leggere file di testo, permettendo di leggere dati carattere per carattere o linea per linea.

L'uso di `BufferedReader` in combinazione con `FileReader` aumenta l'efficienza della lettura attraverso il buffering dei caratteri.

*Imparare a leggere contenuti da file in Java.*

# File I/O: Scrittura

*Scrivere su file*

Per scrivere su file, Java utilizza classi come **FileWriter** e **BufferedWriter**.

FileWriter permette di scrivere testo su file in modo semplice, mentre l'uso di BufferedWriter in combinazione con FileWriter può migliorare le prestazioni grazie al buffering.

*Tecniche per scrivere dati su file in Java.*

# File I/O: Gestione Avanzata

## *File e Directory*

Le classi **File**, **FileInputStream**, **FileOutputStream**, e le API **NIO** forniscono funzionalità avanzate per la gestione di file e directory, permettendo operazioni come:

- la creazione,
- la lettura,
- la scrittura e
- la cancellazione di file,
- gestione delle directory e la verifica delle proprietà dei file.

*Gestire file e directory con Java per applicazioni più complesse.*

# Java React

## Seconda parte: JAVA Fondamenti

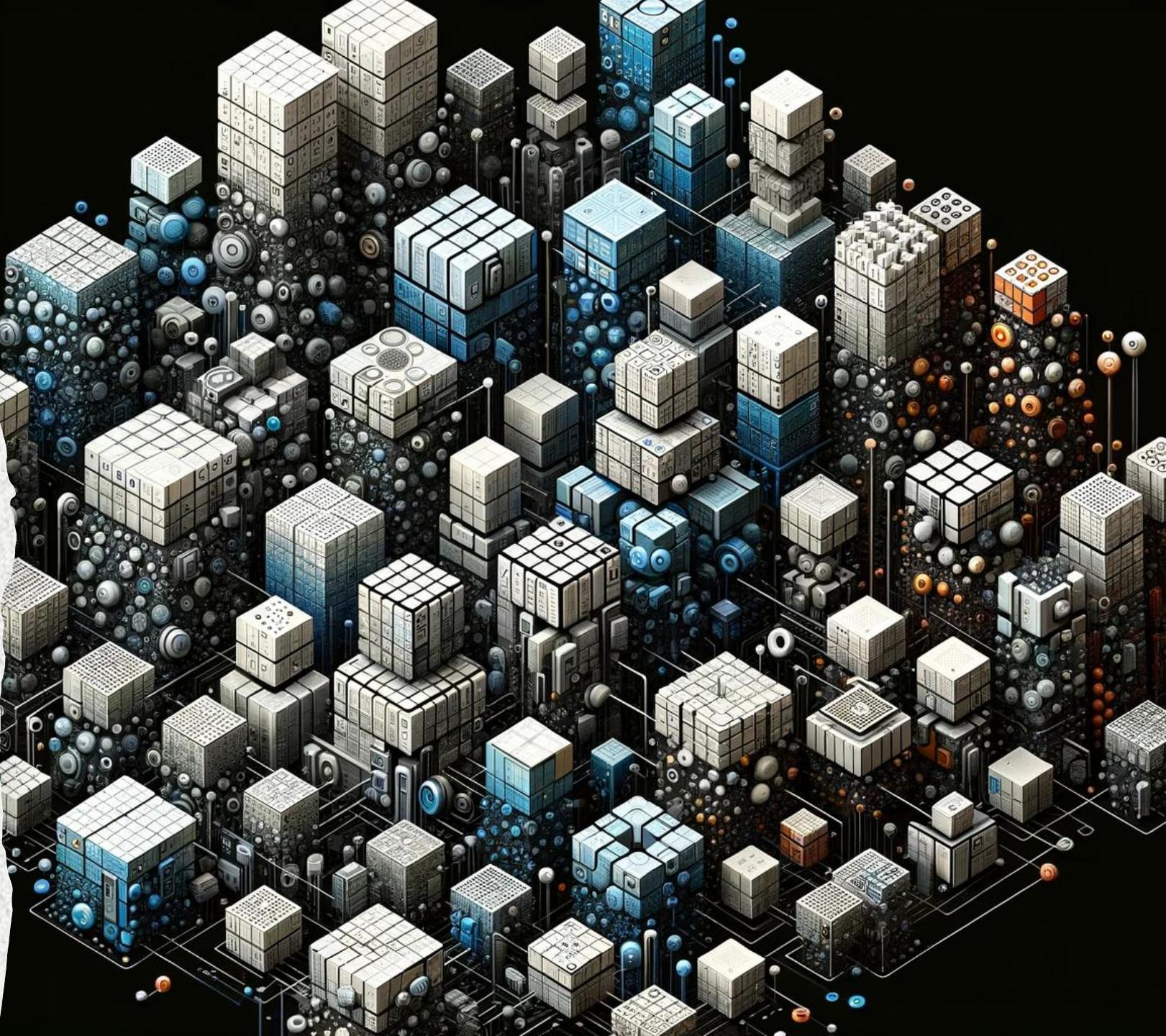
Giorno 17: 22.05.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer



# Generics (cenni)



# Generics in Java

## *Introduzione*

I **generics** introducono la tipizzazione generica in Java, permettendo di scrivere codice più sicuro e riutilizzabile.

Consentono di specificare, al momento della dichiarazione di una classe, interfaccia o metodo, quale tipo di dati (**classe o interfaccia**) essi possono accettare.

Questo approccio **riduce la necessità di casting** e diminuisce il rischio di errori a runtime.

*Generics: Migliorano la sicurezza e la riutilizzabilità del codice.*

# Perché Usare i Generics

## Vantaggi

- Riduzione degli errori a runtime mediante controlli di tipo più forti
- Eliminazione del casting esplicito, rendendo il codice più leggibile e meno soggetto a errori
- Maggiore riutilizzabilità del codice: lo stesso codice può essere utilizzato con diversi tipi di dati.

*Generics: Aumentano la sicurezza e la leggibilità.*

# Sintassi dei Generics

*Come si usano*

La sintassi dei generics utilizza le parentesi angolari (`< >`) per specificare il tipo di parametro.

Ad esempio, `List<String>` indica una lista che può contenere solo stringhe. Questo concetto può essere applicato a classi, interfacce e metodi.

Esempio:

```
// T è un tipo di Classe, non un tipo primitivo
public class Box<T> {
    private T t;
}
```

*Generics: Sintassi e utilizzo molto semplici.*

# Esempio Pratico: ArrayList

## Vantaggi

Gli **ArrayList** sono un esempio comune dell'uso dei generics.

Senza generics, un ArrayList può contenere qualsiasi tipo di oggetto, ma con i generics, puoi specificare il tipo di elementi che l'ArrayList può contenere,

ad esempio **ArrayList<String>** o **ArrayList<Integer>** Questo garantisce che solo il tipo specificato possa essere inserito nella lista.

*ArrayList è uno degli usi più comuni di Generics:*

# Collections



# Le Collezioni in Java

## *Panoramica e importanza*

Le collezioni in Java rappresentano un framework unificato che fornisce un'architettura per memorizzare e manipolare **gruppi di oggetti**.

Esse offrono **strutture dati dinamiche**, a differenza degli array che hanno dimensione fissa. Questo framework supporta diverse operazioni come la ricerca, l'ordinamento, l'inserimento, la manipolazione e l'eliminazione degli elementi.

Le principali interfacce sono ***List*, *Set*, *Map*, e *Queue***, ognuna con più implementazioni specifiche che offrono diverse proprietà, come **l'ordinamento** degli elementi, **l'unicità** e **l'ordine di inserimento**.

Comprendere le collezioni è fondamentale per lo sviluppo efficace di applicazioni Java, poiché permettono di gestire dati complessi in modo flessibile ed efficiente.

*Fondamenti delle collezioni per la gestione avanzata dei dati.*

# Introduzione a List

*Lista di elementi*

L'interfaccia ***List*** in Java rappresenta una **sequenza ordinata di elementi**, dove **ogni elemento ha un indice** basato sulla sua posizione.

A differenza dei set, permette elementi duplicati e offre operazioni per l'inserimento, la rimozione, e l'accesso agli elementi in base alla loro posizione.

Le implementazioni principali includono ***ArrayList***, ***LinkedList***, e ***Vector***, ciascuna con caratteristiche e casi d'uso specifici.

*Comprendere l'interfaccia List e le sue implementazioni.*

# List: ArrayList

*Array dinamico*

**ArrayList** è un'implementazione dell'interfaccia List basata su un **array ridimensionabile**.

Offre accesso casuale rapido agli elementi tramite indici, rendendolo efficiente per operazioni di **lettura frequenti**.

Tuttavia, l'aggiunta e la rimozione di elementi possono essere costose a causa della necessità di ridimensionare e copiare l'array.

Esempio di utilizzo: gestione di un elenco di prodotti in un carrello della spesa, dove l'accesso e l'aggiornamento frequenti sono comuni.

*Utilizzo di ArrayList per accesso rapido agli elementi.*

# List: LinkedList

*Lista doppiamente collegata*

**LinkedList** è un'implementazione dell'interfaccia List basata su una **lista doppiamente collegata**.

Supporta l'**inserimento e la rimozione efficienti** degli elementi, poiché queste operazioni non richiedono il ridimensionamento dell'array. Tuttavia, l'accesso casuale agli elementi è meno efficiente rispetto ad ArrayList.

Esempio di utilizzo: applicazioni che richiedono inserimenti e rimozioni frequenti di elementi, come la gestione di una coda di attività.

*LinkedList per inserimenti e rimozioni efficienti.*

# List: Vector

*Array dinamico sincronizzato*

**Vector** è simile ad `ArrayList`, ma con due differenze chiave: è **sincronizzato**, e quindi thread-safe, e ha capacità che si espandono in modo **incrementale**.

Queste caratteristiche lo rendono adatto per applicazioni **multithreading** che richiedono accesso concorrente ai dati.

Tuttavia, la sincronizzazione introduce un overhead che può influire sulle prestazioni.

Esempio di utilizzo: gestione di un elenco condiviso di messaggi in una applicazione di chat multiutente.

*Vector per ambienti multithreading.*

# Introduzione a Set

*Lista unica di elementi*

L'interfaccia **Set** in Java rappresenta una collezione che **non ammette elementi duplicati**. È ideale per la gestione di insiemi di dati unici, come elenchi di identificativi o valori dove l'unicità è fondamentale.

A differenza della List, Set non mantiene un ordine specifico degli elementi.

Le implementazioni comuni includono **HashSet**, **LinkedHashSet**, e **TreeSet**, ciascuna con peculiarità specifiche come l'ordine di inserimento o l'ordinamento degli elementi.

*Capire le basi dell'interfaccia Set.*

# HashSet

*Gestione di elementi unici*

**HashSet** è l'implementazione più comune del Set interface basata su una hash table.

È la scelta migliore per collezioni che richiedono **operazioni veloci** di aggiunta, rimozione, e verifica dell'esistenza di un elemento, **senza** mantenere un **ordine** specifico degli elementi.

Esempio di utilizzo: gestire un elenco di email uniche dove l'ordine non è rilevante.

*Utilizzo di HashSet per garantire l'unicità degli elementi.*

# Set: LinkedHashSet

*Ordine di inserimento*

**LinkedHashSet** estende HashSet, mantenendo una **lista doppiamente collegata** attraverso tutti i suoi elementi.

Questo mantiene l'ordine di inserimento degli elementi, rendendolo ideale per situazioni in cui l'ordine conta ma sono richiesti anche valori unici.

Esempio di utilizzo: mantenere un registro delle visite di pagine web, dove ogni pagina è unica ma l'ordine di visita è importante.

*LinkedHashSet per unire unicità e ordine di inserimento.*

# Set: TreeSet

*Elementi ordinati naturalmente*

**TreeSet** è implementato basandosi su una **struttura ad albero** (Red-Black tree), che ordina gli elementi automaticamente.

È perfetto per quando è necessario mantenere un **ordinamento naturale** o custom degli elementi (ad esempio, ordinati alfabeticamente o per data).

Esempio di utilizzo: un insieme ordinato di date per eventi futuri, dove è importante sia l'unicità che l'ordine cronologico.

*TreeSet per mantenere gli elementi ordinati.*

# Introduzione a Map

*Associazione chiave-valore*

L'interfaccia **Map** in Java rappresenta una struttura dati che **associa chiavi univoche a valori**.

È utilizzata per memorizzare e gestire coppie di dati in modo che ogni chiave mappi a esattamente un valore.

Le **chiavi sono uniche** nella mappa, mentre i valori possono essere duplicati.

Implementazioni principali includono **HashMap**, **LinkedHashMap**, e **TreeMap**, offrendo diverse prestazioni e caratteristiche come l'ordine di inserimento o l'ordinamento delle chiavi.

*Comprendere l'uso dell'interfaccia Map.*

# Map: HashMap

*Associazione chiave-valore*

**HashMap** è una delle implementazioni più comuni della Map interface basata su hash table.

Offre **prestazioni elevate** per l'aggiunta, la rimozione e la ricerca di valori basati su una chiave.

Non mantiene alcun ordine delle chiavi o dei valori.

Esempio di utilizzo: un catalogo di prodotti dove ogni prodotto ha un ID unico (chiave) e specifiche dettagliate (valore).

*HashMap per gestire dati chiave-valore.*

# Map: LinkedHashMap

*Ordine di inserimento per chiavi*

**LinkedHashMap** estende HashMap, aggiungendo la capacità di mantenere le **chiavi in ordine di inserimento**.

Questo è utile per applicazioni come cache LRU (Least Recently Used), dove oltre all'accesso rapido, l'ordine di inserimento è cruciale.

Esempio di utilizzo: cache di pagine web visitate, mantenendo l'ordine di visita.

*LinkedHashMap combina velocità e ordine di inserimento.*

# Map:TreeMap

*Chiavi ordinate*

**TreeMap** implementa la Map interface usando un **albero rosso-nero** (RB Tree), mantenendo le chiavi ordinate secondo il loro ordine naturale o tramite un comparatore fornito.

È ideale quando è necessario un **ordinamento specifico delle chiavi**.

Esempio di utilizzo: un dizionario dove ogni parola (chiave) ha una definizione (valore) e le parole sono ordinate alfabeticamente.

*TreeMap per chiavi ordinate.*

# Introduzione a Queue

*Gestione sequenziale degli elementi*

L'interfaccia **Queue** in Java rappresenta una collezione usata per tenere in coda gli elementi prima del loro elaborazione.

Segue in genere un principio FIFO (First-In-First-Out), ma ci sono eccezioni come le code a priorità.

È utile in scenari dove è **importante l'ordine di elaborazione**, come nell'elaborazione dei task o nella gestione dei messaggi.

Implementazioni comuni includono **LinkedList**, che implementa anche List, e **PriorityQueue**, che ordina gli elementi secondo la loro priorità.

*Introduzione alla gestione delle code in Java.*

# Queue: PriorityQueue

*Elementi ordinati per priorità*

**PriorityQueue** è una coda basata su priorità che ordina i suoi elementi secondo il loro ordine naturale o un comparatore.

Non è **thread-safe** e non garantisce di mantenere un ordine FIFO per elementi con uguale priorità.

Esempio di utilizzo: gestione di task da eseguire dove alcuni task hanno priorità più alta e devono essere eseguiti per primi.

*PriorityQueue per gestire priorità.*

# Java React

## Seconda parte: JAVA Fondamenti

Giorno 18: 23.05.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer



# Serializzazione



# Serializzazione

*Salvare stati degli oggetti*

La serializzazione in Java permette di convertire un oggetto in una sequenza di byte che può essere salvata su un file o trasferita attraverso la rete.

Questo processo è facilitato dall'implementazione dell'interfaccia **Serializable** negli oggetti che si vogliono serializzare.

*Conservare e trasferire lo stato degli oggetti Java.*

# Deserializzazione

*Ricreare oggetti da byte*

La deserializzazione è il processo inverso della serializzazione e consente di ricreare un oggetto a partire dalla sua rappresentazione binaria.

Questo processo è critico per recuperare lo stato degli oggetti salvati o trasferiti.

*Ripristinare oggetti Java dalla loro forma serializzata.*

# Serializzazione Personalizzata

*Controllare il processo di serializzazione*

Java permette di personalizzare il processo di serializzazione e deserializzazione attraverso l'implementazione dei metodi **writeObject** e **readObject**.

Questo può essere utile per gestire casi particolari, come la gestione di transient fields o la serializzazione di dipendenze complesse.

*Personalizzare la serializzazione per soddisfare esigenze specifiche.*

# Uso di JSON in Java

*Introduzione a JSON*

**JSON** (JavaScript Object Notation) è un formato leggero di scambio dati, ideale per la trasmissione di dati tra server e applicazioni web.

È facilmente leggibile e scrivibile dagli umani e facilmente analizzabile e generabile dalle macchine.

Per lavorare con JSON in Java, si utilizzano librerie popolari come **Gson** e **Jackson**.

*Introduzione all'interoperabilità dei dati con JSON in Java.*

# Perché JSON?

## *Vantaggi dell'uso di JSON*

JSON è diventato lo standard de facto per lo scambio di dati su internet per diversi motivi: è testuale e leggero, con una sintassi semplice; è facilmente leggibile da umani e macchine; è completamente indipendente dal linguaggio, il che lo rende ideale in ambienti di sviluppo eterogenei; supporta strutture dati complesse, inclusi oggetti e array.

Queste caratteristiche rendono JSON una scelta eccellente per la serializzazione dei dati in Java e per la comunicazione tra client e server.

*Capire i benefici di JSON nell'ecosistema di sviluppo moderno.*

# Serializzazione con Gson

*Da Java a JSON*

La serializzazione è il processo di conversione di un oggetto Java in una stringa JSON. Questo è utile quando si hanno oggetti Java che si desidera trasmettere a un client web o salvare in un formato leggibile e trasportabile.

La libreria **Gson**, che va importata all'interno della classe, permette di serializzare oggetti Java con una singola linea di codice.

```
Gson gson = new Gson();
String json = gson.toJson(myObject);
System.out.println(json);
```

*Tecniche di serializzazione di oggetti Java in stringhe JSON con Gson.*

# Deserializzazione con Gson

*Da JSON a Java*

La deserializzazione è il processo inverso, convertendo una stringa JSON in un oggetto Java.

Per deserializzare una stringa JSON in un oggetto Java usando Gson occorre definire una classe Java con campi che corrispondono alle chiavi JSON e poi utilizzare un metodo dell'oggetto di tipo Gson per deserializzarlo:

```
String json = "{\"id\":101,\"name\":\"John Doe\",\"email\":\"john.doe@example.com\"}";  
  
Gson gson = new Gson();  
MyObject myObject = gson.fromJson(json,myObject);  
System.out.println(myObject.getName());
```

*Convertire stringhe JSON in oggetti Java con Gson.*

# Serializzazione con Jackson

## *Serializzazione e Deserializzazione*

Jackson è un'altra popolare libreria Java utilizzata per serializzare oggetti Java in JSON e deserializzare JSON in oggetti Java.

La libreria **Jackson**, che va importata all'interno della classe, permette di **serializzare** e deserializzare oggetti Java utilizzando l'oggetto **ObjectMapper** :

```
ObjectMapper mapper = new ObjectMapper();

MyClass myObject = new MyClass(1, "Jackson Example");
try {
    String json = mapper.writeValueAsString(myObject);
    System.out.println(json);
} catch (JsonProcessingException JsonProcessingException e) {
    e.printStackTrace();
}
```

*Utilizzare Jackson per una manipolazione avanzata di JSON in Java.*

# Deserializzazione con Jackson

## *Serializzazione e Deserializzazione*

Jackson è un'altra popolare libreria Java utilizzata per serializzare oggetti Java in JSON e deserializzare JSON in oggetti Java.

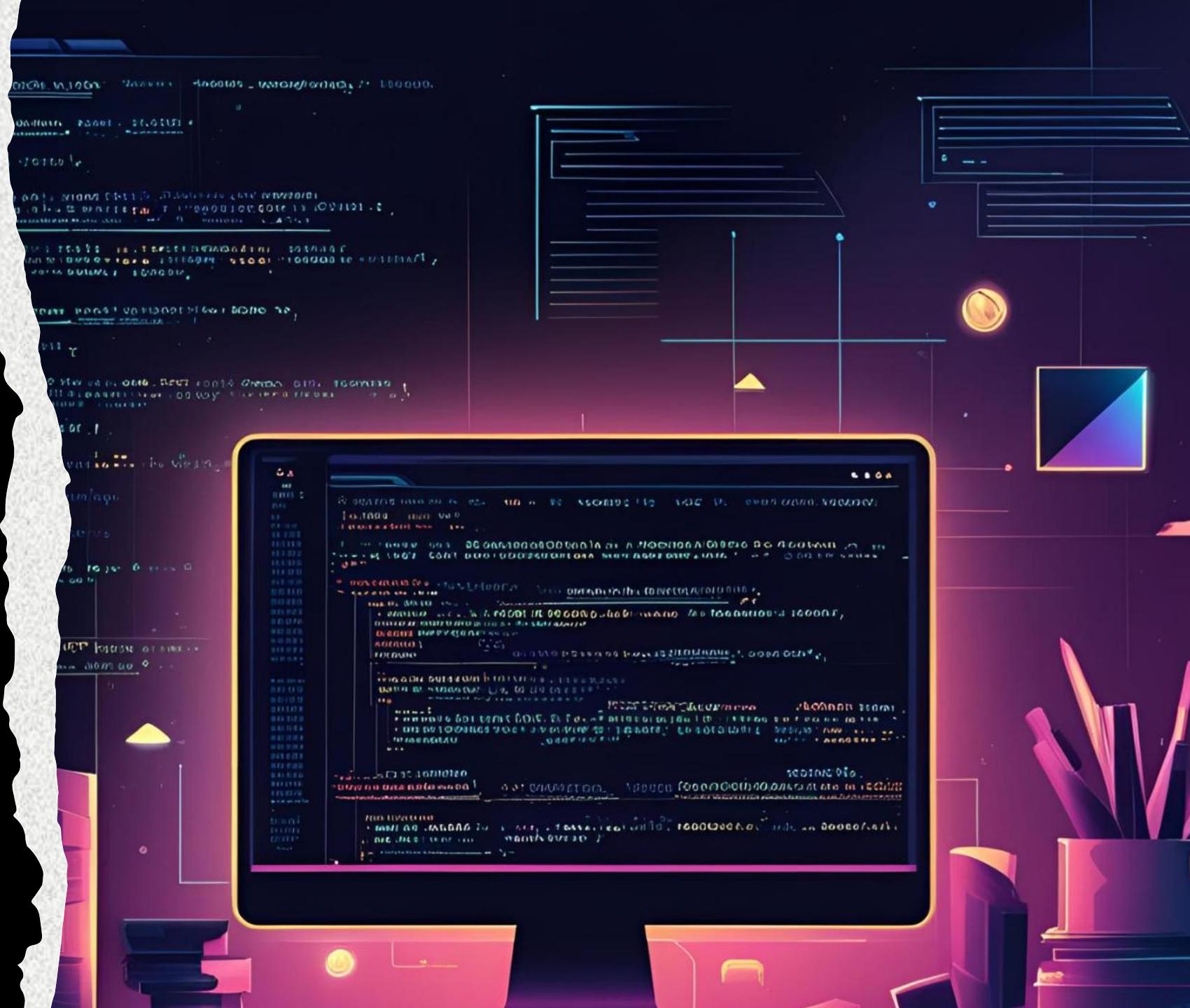
La libreria **Jackson**, che va importata all'interno della classe, permette di serializzare e **deserializzare** oggetti Java utilizzando l'oggetto **ObjectMapper** :

```
ObjectMapper mapper = new ObjectMapper();

String json = "{\"id\":1,\"name\":\"Jackson Example\"}";
try {
    MyClass myObject = mapper.readValue(json, MyClass.class);
    System.out.println(myObject.getName());
} catch (JsonProcessingException e) {
    e.printStackTrace();
}
```

*Utilizzare Jackson per una manipolazione avanzata di JSON in Java.*

# Test unitari con JUnit



# Test Unitario

*Cos'è e perché è utile*

Un **test unitario** è un metodo che esegue una piccola parte del codice – tipicamente una funzione o metodo – in modo **isolato e controllato**.

L'obiettivo è verificare che con un certo input, il metodo produca l'output corretto.

I test unitari sono **ripetibili, automatici, veloci** e documentano il comportamento del software.

JUnit fornisce le annotazioni, le classi e le funzioni per scrivere test unitari efficaci.

Usare i test unitari aiuta ad aumentare la **qualità** e la **manutenibilità** del **software**, ed è una buona pratica professionale.

*Un test unitario verifica il comportamento di un singolo metodo in modo preciso e ripetibile.*

# Cos'è JUnit

*Il framework di test automatici per Java*

**JUnit** è un framework open-source che consente di **testare automaticamente** il comportamento del codice Java.

Con **JUnit** è possibile scrivere piccoli metodi che testano altri metodi del programma, verificando se producono i risultati attesi.

Questo tipo di test è detto **test unitario** perché riguarda unità minime di logica, come i singoli metodi.

**JUnit** permette di eseguire questi test rapidamente, mostrare risultati chiari, e identificare eventuali errori o regressioni nel codice.

*JUnit ci aiuta a trovare gli errori nel codice prima ancora di eseguire l'applicazione completa.*

# Classe da testare

*Esempio: una classe Dispositivo*

Prima di scrivere un test, è necessario avere una classe da testare.

Nell'esempio seguente, la classe Dispositivo ha due metodi: **accendi()** e **spegni()**,

e un attributo booleano **acceso**.

Sarà oggetto dei nostri test **unitari**.

```
public class Dispositivo {  
    public boolean acceso = false;  
  
    public void accendi() {  
        acceso = true;  
    }  
  
    public void spegni() {  
        acceso = false;  
    }  
}
```

*Per ogni classe logica si possono (e si dovrebbero) scrivere dei test unitari.*

# Struttura di un test JUnit

## Annotazioni e asserzioni

Un **metodo di test** in JUnit viene definito tramite l'annotazione **@Test**.

All'interno del metodo, si usano le **asserzioni** (assert) per verificare che i risultati siano corretti.

- Se le asserzioni falliscono, il test viene marcato come fallito.
- Le asserzioni più comuni sono **assertEquals**, **assertTrue**, **assertFalse**.

Tutti i test sono eseguiti **automaticamente** da JUnit.

```
@Test
void testAccensione() {
    Dispositivo d = new Dispositivo();
    d.accendi();
    assertTrue(d.acceso);
}
```

La struttura base di un test JUnit è semplice ma molto potente e leggibile.

# Come usare JUnit in Eclipse

## *Configurazione in Eclipse*

In Eclipse, puoi usare JUnit anche senza Maven.

È sufficiente:

1. Fare clic destro sul progetto > *Build Path* > *Add Libraries* > *JUnit*
2. Scegliere **JUnit 5**
3. Creare una nuova classe di test con *New* > *JUnit Test Case*
4. Aggiungere i metodi di test usando l'annotazione **@Test**
5. Eseguire i test con *Run As* > *JUnit Test*.

*Eclipse integra JUnit nativamente: bastano pochi clic per iniziare a testare il proprio codice.*