Definizioni da sapere sulla programmazione Java:

Java è un linguaggio di programmazione ad alto livello, orientato agli oggetti, multi-piattaforma e sicuro. Si basa su una sintassi simile a quella del linguaggio C++ ed è utilizzato per sviluppare applicazioni desktop, applicazioni web, applicazioni mobile e molti altri progetti.

Variabili:

Le variabili in Java sono usate per memorizzare i dati; Per dichiarare una variabile, è necessario specificare il tipo di dati che verrà memorizzato nella variabile e un nome per la variabile.

Variabile= possiamo definirla come un contenitore/scatola a cui assegniamo un valore.

Variabile dichiarata= diamo alla variabile solo un nome, senza attribuire un valore.

Variabile inizializzata= dichiariamo + attribuiamo ad essa un valore, in futuro modificabile.

Variabile/Costante= rendiamo una variabile inizializzata una costante. Il suo valore non potrà più essere modificato, da qui il nome costante.

I campi di una classe Java sono (le variabili/attributi) dichiarate all'interno della classe. Questi campi possono essere utilizzati per memorizzare dati che sono specifici della classe stessa. Ad esempio, se si sta creando una classe "Automobile", i campi potrebbero includere il modello dell'auto, il suo anno di fabbricazione, il colore, il prezzo, il numero di porte, ecc.

I campi sono dichiarati all'interno della classe, ma al di fuori dei metodi. Possono essere di diversi tipi, come interi, stringhe, booleani, array, oggetti e così via. Quando si dichiara un campo, si specifica il suo tipo di dati e il nome del campo.

Ecco un esempio di come dichiarare campi in una classe Java:

```
public class Automobile {
    String modello;
    int anno;
    String colore;
    double prezzo;
    int numeroPorte;
}
```

In questo esempio, abbiamo dichiarato cinque campi all'interno della classe "Automobile". Il campo "modello" è una stringa, il campo "anno" è un intero, il campo "colore" è una stringa, il campo "prezzo" è un double e il campo "numeroPorte" è un intero.

È possibile accedere ai campi all'interno dei metodi della classe utilizzando il nome del campo. Ad esempio, per impostare il campo "modello" su "Fiat 500", **si può utilizzare il seguente codice**:

```
public class Automobile {

      String modello;

      int anno;
```

```
    String colore;

double prezzo;

int numeroPorte;


        public void setModello(String modello) {
            this.modello = modello;
        }
    }

    Automobile auto = new Automobile();
auto.setModello("Fiat 500");
```

In questo esempio, abbiamo creato un metodo "setModello" che imposta il campo "modello" sulla stringa passata come argomento. Nota che all'interno del metodo abbiamo utilizzato "this.modello" per riferirci al campo "modello" della classe "Automobile".

È importante notare che i campi di una classe Java possono avere diversi livelli di accessibilità, come pubblico, privato, protetto o pacchetto (default). Questo influisce su quale parte del codice può accedere al campo. Ad esempio, se un campo è dichiarato come privato, solo i metodi all'interno della stessa classe possono accedervi. Se un campo è dichiarato come pubblico, qualsiasi parte del codice può accedervi.

In generale, è una buona pratica dichiarare i campi come privati e fornire metodi pubblici per accedere e modificare i valori dei campi. Questo è noto come incapsulamento e aiuta a garantire che il codice sia più sicuro e manutenibile.

Dati Primitivi e Reference differenze:

- **Primitivi**= dati forniti da java standard (iniziano con la minuscola).

- **Reference**= tipi di dati complessi quasi tutti creati da noi (iniziano tutti con la maiuscola) e si differenziano anche dagli attributi disponibili o metodi.

Dati primitivi in Java

I dati primitivi sono i tipi di dati fondamentali in Java. Sono chiamati "primitivi" perché non sono oggetti, e quindi non hanno metodi o campi. I dati primitivi sono immagazzinati direttamente nello stack della memoria e vengono manipolati direttamente dal processore, il che li rende molto efficienti.

I dati primitivi includono:

- **byte:** un numero intero a 8 bit, che può essere compreso tra -128 e 127

- **short:** un numero intero a 16 bit, che può essere compreso tra -32,768 e 32,767
- **int:** un numero intero a 32 bit, che può essere compreso tra -2,147,483,648 e 2,147,483,647
- **long:** un numero intero a 64 bit, che può essere compreso tra -9,223,372,036,854,775,808 e 9,223,372,036,854,775,807
- **float:** un numero in virgola mobile a 32 bit
- **double**: un numero in virgola mobile a 64 bit
- **boolean**: un valore booleano che può essere true o false
- **char**: un singolo carattere Unicode a 16 bit

Ecco un esempio di utilizzo di dati primitivi in Java:

```
int numero = 10;

double pi = 3.14159;

boolean valoreLogico = true;

char carattere = 'A';
```

In questo esempio, abbiamo dichiarato quattro variabili che memorizzano rispettivamente un numero intero, un numero in virgola mobile, un valore booleano e un carattere.

Reference/riferimenti (oggetti) in Java:

I riferimenti (o oggetti) in Java sono usati per rappresentare tipi di dati più complessi, come stringhe, array e oggetti personalizzati. A differenza dei dati primitivi, i riferimenti sono oggetti che hanno metodi e campi. I riferimenti sono memorizzati nello heap della memoria e vengono manipolati indirettamente tramite un puntatore che viene memorizzato nello stack.

I dati reference includono:

- **String** |istanza della classe String: oggetto che incapsula una sequenza di caratteri "ciao sono io" | variabile [referenza].
- **array** |contenitore di dati con lunghezza prefissata, creazione di elementi tutti dello stesso tipo.
- **Oggetti** | l'istanza particolare di una certa classe, e esso può possedere (o esporre) alcuni metodi.

Ecco un esempio di utilizzo di un riferimento in Java:

```
String nome = "Mario";
```

In questo esempio, abbiamo dichiarato una variabile nome di tipo String. La stringa "Mario" è un oggetto di tipo String, e la variabile nome contiene un riferimento all'oggetto String memorizzato nello heap. La variabile nome non contiene la stringa stessa, ma solo il riferimento all'oggetto che contiene la stringa.

Inoltre, i riferimenti possono essere usati per creare oggetti personalizzati. Ad esempio:

```java
public class Automobile {
    String modello;
    int anno;

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

Automobile auto = new Automobile("Fiat 500", 2023);
```

Scanner:

La classe Scanner in Java è una classe di utilità che consente di leggere dati di input da diverse fonti, come la console, un file o una stringa. La classe Scanner è definita nel package java.util e fornisce molti metodi per leggere dati di input in diversi formati.

import java.util.Scanner;
ha lo scopo di importare nel programma la classe Scanner utilizzata per facilitare lo stream di input. **User input** ---> inserimento dati dall'utente:

Scanner scanner =new Scanner(System.in);
crea un'oggetto di classe scanner chiamato in che può disporre di metodi
come nextInt() o nextLine() in grado di accettare dati in input da tastiera.

Mentre per l'output può essere usata la tradizionale istruzione System.out.println();
chiediamo all'utente cosa inserire tramite la stampa.

String nome = scanner nex.line();
diamo la possibilità all'utente di inserire i dati richiesti.

Ecco un esempio di utilizzo della classe Scanner per leggere un intero dalla console:

```java
import java.util.Scanner;

public class LeggiIntero {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Inserisci un intero: ");
        int numero = input.nextInt();
        System.out.println("Hai inserito il numero " + numero);
    }
}
```

In questo esempio, abbiamo creato un oggetto Scanner chiamato input che legge i dati di input dalla console (**System.in**). Successivamente, abbiamo utilizzato il metodo **nextInt**() per leggere un numero intero dalla console. Infine, abbiamo stampato il numero intero inserito dall'utente sulla console.

La classe Scanner fornisce molti altri metodi per leggere dati di input in diversi formati, come **next**(), **nextLine**(), **nextDouble**(), **nextBoolean**(), ecc.

Ad esempio, ecco un esempio di utilizzo del metodo nextLine() per leggere una riga di testo dalla console:

```
import java.util.Scanner;

public class LeggiTesto {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Inserisci una riga di testo: ");
        String testo = input.nextLine();
        System.out.println("Hai inserito il testo: " + testo);
    }
}
```

In questo esempio, abbiamo utilizzato il metodo **nextLine**() per leggere una riga di testo dalla console, e abbiamo stampato il testo inserito dall'utente sulla console.

In generale, la classe Scanner è molto utile per leggere dati di input da diverse fonti, come la console, un file o una stringa.

Ecco un esempio di utilizzo della classe Scanner per leggere i dati di input da un file:

```
    import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
    public class LeggiFile {
        public static void main(String[] args) throws
FileNotFoundException {
            File file = new File("input.txt");
            Scanner input = new Scanner(file);
            while (input.hasNextLine()) {
                String linea = input.nextLine();
                System.out.println(linea);
            }
            input.close();
        }
    }
```

In questo esempio, abbiamo creato un oggetto File che rappresenta il file **input.txt**, e abbiamo creato un oggetto Scanner che legge i dati di input dal file. **Successivamente**, abbiamo utilizzato il metodo **hasNextLine**() per verificare se ci sono altre linee nel file, e il metodo **nextLine**() per leggere le linee del file finché non ci sono altre linee disponibili. Infine, abbiamo chiuso l'oggetto Scanner.

Operatori Aritmetici= [+ - / * %]:

In Java, gli operatori aritmetici vengono utilizzati per eseguire operazioni matematiche su numeri. Ecco una lista degli operatori aritmetici in Java:

Operatore + = esegue le somme.
Operatore – = esegue le sottrazioni.
Operatore * = esegue le moltiplicazioni.
Operatore / = esegue le divisioni.
Operatore % = esegue i resti della divisione| Il modulo resto si può utilizzare anche per trovare se un numero è pari da resto 0 – dispari se da resto 1.
Inoltre si possono utilizzare in modo Shorthand= **esempio**: a += b | **è uguale ad**: a = a + b
Ancora gli operatori + + e – – = si possono utilizzare come incremento o decremento.

Ecco alcuni esempi di utilizzo degli operatori aritmetici in Java:

```
int a = 10;

int b = 3;

int somma = a + b; // 13

int differenza = a - b; // 7

int prodotto = a * b; // 30

int divisione = a / b; // 3

int resto = a % b; // 1
```

In questo esempio, abbiamo dichiarato due variabili **a** e **b** di tipo **int** e abbiamo **assegnato** loro i valori **10** e **3** rispettivamente. Successivamente, abbiamo utilizzato gli operatori aritmetici per eseguire le **operazioni** matematiche desiderate e abbiamo assegnato i **risultati** alle variabili **somma**, **differenza**, **prodotto**, **divisione** e **resto**.

È possibile utilizzare anche gli operatori aritmetici con variabili di tipo double.

Ecco un esempio:

```
double c = 5.5;

double d = 2.5;

double somma = c + d; // 8.0

double differenza = c - d; // 3.0

double prodotto = c * d; // 13.75
```

```
double divisione = c / d; // 2.2

double resto = c % d; // 0.5
```

In questo esempio, abbiamo dichiarato due variabili c e d di tipo double e abbiamo **assegnato** loro i valori **5.5** e **2.5** rispettivamente. Successivamente, abbiamo utilizzato gli operatori aritmetici per eseguire le **operazioni** matematiche desiderate e abbiamo assegnato i **risultati** alle variabili **somma**, **differenza**, **prodotto**, **divisione** e **resto**.

Ricorda che quando si utilizzano operatori aritmetici con variabili di tipo **int** e **double**, il **risultato** è di **tipo int** se **tutti** gli **operandi** sono di **tipo int**, **altrimenti** è di tipo **double**. **Se si desidera ottenere** un **risultato** di **tipo double**, è possibile **utilizzare la conversione esplicita**, come mostrato di seguito:

```
int a = 10;

int b = 3;

double divisione = (double) a / b; // 3.3333333333333335
```

In questo esempio, abbiamo **utilizzato** la **conversione** esplicita per **convertire** la variabile **a** di **tipo int** in **double prima di eseguire la divisione**, in modo da **ottenere** un **risultato** di tipo **double**.

Operatori di Comparazione:

In Java, gli **operatori di comparazione** vengono **utilizzati** per **confrontare** due **valori** e **restituire** un valore **booleano** (**true** o **false**) a seconda del risultato del confronto. Ecco una lista degli operatori di comparazione in Java:

- **==** (uguale a)
- **!=** (diverso da)
- **<** (minore di)
- **>** (maggiore di)
- **<=** (minore o uguale a)
- **>=** (maggiore o uguale a)

Ecco alcuni esempi di utilizzo degli operatori di comparazione in Java:

```
int a = 5;

int b = 3;

boolean uguale = a == b; // false

boolean diverso = a != b; // true

boolean minore = a < b; // false
```

```
boolean maggiore = a > b; // true

boolean minoreUguale = a <= b; // false

boolean maggioreUguale = a >= b; // true
```

In questo esempio, abbiamo **dichiarato** due variabili **a** e **b** di tipo **int** e abbiamo **assegnato** loro i **valori 5** e **3** rispettivamente. Successivamente, abbiamo **utilizzato** gli **operatori** di **comparazione** per **confrontare** i **valori** delle due variabili e abbiamo assegnato i **risultati** alle variabili **uguale**, **diverso**, **minore**, **maggiore**, **minoreUguale** e **maggioreUguale**.

Ricorda che quando si confrontano valori di tipo double, è necessario prestare attenzione alla precisione, a causa delle limitazioni della rappresentazione in virgola mobile dei numeri.

Ad esempio, il seguente confronto può restituire un risultato inaspettato:

```
double c = 0.1 + 0.2;

double d = 0.3;

boolean uguale = c == d; // false
```

In questo esempio, abbiamo **dichiarato** due variabili **c** e **d** di tipo **double**. La variabile **c** viene **inizializzata** con la somma di **0.1** e **0.2**, che **in teoria dovrebbe essere uguale a 0.3**. Tuttavia, a **causa** della **rappresentazione in virgola mobile**, il **valore** di c è **leggermente diverso da 0.3**, il che fa sì che il **confronto** con d **restituisca false**.

Per evitare problemi di precisione, è possibile utilizzare una tolleranza durante il confronto, come mostrato di seguito:

```
double c = 0.1 + 0.2;

double d = 0.3;

boolean uguale = Math.abs(c - d) < 1e-10; // true
```

In questo esempio, abbiamo **utilizzato** il metodo **Math.abs**() per **ottenere** il **valore assoluto** della **differenza** tra **c** e **d**, e abbiamo **confrontato** questo **valore** con una **tolleranza** di **1e-10** (**10 elevato alla meno 10**), che **rappresenta** una **piccola frazione di errore accettabile**.

Operatori Logici:

L'uso degli operatori logici può far sì che in un inserimento di dati si possano valutare più condizioni e in base agli operatori che utilizzeremo potranno o dovranno essere:

l'operatore **&&** (AND logico) **restituisce true** solo se **entrambe** le condizioni sono **true**.

L'operatore **||** (OR logico) **restituisce true** se almeno **una** delle condizioni è **true**.

L'operatore **!** (NOT logico) **nega** il **valore** di una **condizione booleana**, **ovvero restituisce true** se la **condizione** è **false**, e **viceversa**.

AND (&&) --> equivale ad e cioè anche (tutte le condizioni devono coincidere vere).
OR (||) --> equivale ad oppure (le condizioni vanno valutate distintamente).
NOT (!) --> equivale a diverso (le condizioni devono essere diverse).

Ecco alcuni esempi di utilizzo degli operatori logici in Java:

```java
int a = 5;

int b = 3;

boolean condizione1 = a > b && a < 10; // true

boolean condizione2 = a == b || a > 10; // false

boolean condizione3 = !(a == b); // true
```

In questo esempio, abbiamo **dichiarato** due variabili **a** e **b** di tipo **int** e abbiamo **assegnato** loro i valori **5** e **3** rispettivamente. Successivamente, abbiamo utilizzato gli **operatori logici** per **combinare** o **negare** le **condizioni booleane** e abbiamo **assegnato** i **risultati** alle variabili **condizione1**, **condizione2** e **condizione3**.

È possibile combinare più operatori logici per creare espressioni booleane più complesse.

Ad esempio:

```java
int x = 5;

int y = 10;

int z = 15;

boolean condizione = x > y || (y < z && z < x); // true
```

In questo esempio, abbiamo **dichiarato** tre variabili **x**, **y** e **z** di tipo **int** e abbiamo **assegnato** loro i valori **5**, **10** e **15** rispettivamente. Successivamente, abbiamo **utilizzato** gli **operatori logici** per

combinare le **condizioni booleane** e abbiamo **assegnato** il **risultato** alla **variabile condizione**. L'espressione booleana **x > y || (y < z && z < x)** restituisce **true** perché **almeno una** delle **due condizioni** è **true**: la prima condizione **x > y** è **false**, ma la **seconda** condizione (**y < z && z < x**) è **true perché y < z è true e z < x è false**.

Math Class:

La classe Math è a disposizione nella libreria di Java, **è una classe finale, perciò non possono essere estese nuove classi, inoltre non possono essere create istanze e tutti i suoi metodi sono statici.**

In Java, la classe **Math** fornisce metodi statici per eseguire operazioni matematiche comuni come l'arrotondamento, il calcolo della radice quadrata, il calcolo del valore assoluto, il calcolo del seno, coseno e tangente di un angolo, e così via.

Ecco alcuni esempi di utilizzo della classe Math:

alcuni metodi:

```
double x = 4.5;

double y = -2.7;

double result1 = Math.ceil(x); // result1 = 5.0

double result2 = Math.floor(x); // result2 = 4.0

double result3 = Math.abs(y); // result3 = 2.7

double result4 = Math.sqrt(x); // result4 = 2.1213203435596424

double result5 = Math.sin(Math.PI / 2); // result5 = 1.0

double result6 = Math.max(x, y); // result6 = 4.5

double result7 = Math.min(x, y); // result7 = -2.7
```

In questo esempio, abbiamo **dichiarato** due variabili **x** e **y** di tipo **double** e abbiamo **assegnato** loro i valori **4.5** e **-2.7** rispettivamente. Successivamente, abbiamo **utilizzato alcuni metodi** della classe **Math** per **eseguire operazioni matematiche** su queste variabili:

- Il metodo **Math.ceil**(**x**) **restituisce** il più piccolo intero maggiore o uguale a x, ovvero 5.0.
- Il metodo **Math.floor**(**x**) **restituisce** il più grande intero minore o uguale a x, ovvero 4.0.
- Il metodo **Math.abs**(**y**) **restituisce** il valore assoluto di y, ovvero 2.7.
- Il metodo **Math.sqrt**(**x**) **restituisce** la radice quadrata di x, ovvero 2.1213203435596424.

- Il metodo **Math.sin**(**Math.PI / 2**) **calcola** il seno di un angolo di 90 gradi (espresso in radianti) e restituisce 1.0.
- Il metodo **Math.max**(**x**, **y**) **restituisce** il valore massimo tra x e y, ovvero 4.5.
- Il metodo **Math.min**(**x**, **y**) **restituisce** il valore minimo tra x e y, ovvero -2.7.

Di seguito, ecco una lista dei metodi disponibili nella classe Math e un esempio di utilizzo per ciascuno di essi:

1. **Math.abs(x)** - restituisce il valore assoluto di x.

```
double x = -4.5;

double absX = Math.abs(x); // absX = 4.5
```

2. **Math.ceil(x)** - restituisce il più piccolo intero maggiore o uguale a x.

```
double x = 4.5;

double ceilX = Math.ceil(x); // ceilX = 5.0
```

3. **Math.floor(x**) - restituisce il più grande intero minore o uguale a x.

```
double x = 4.5;

double floorX = Math.floor(x); // floorX = 4.0
```

4. **Math.max(x, y)** - restituisce il valore massimo tra x e y.

```
double x = 4.5;

double y = 3.2;

double maxXY = Math.max(x, y); // maxXY = 4.5
```

5. **Math.min(x, y)** - restituisce il valore minimo tra x e y.

```
double x = 4.5;
```

```
double y = 3.2;

double minXY = Math.min(x, y); // minXY = 3.2
```

6. **Math.pow(x, y)** - restituisce x elevato alla potenza di y.

```
double x = 2.0;

double y = 3.0;

double powXY = Math.pow(x, y); // powXY = 8.0
```

7. **Math.sqrt(x)** - restituisce la radice quadrata di x.

```
double x = 9.0;

double sqrtX = Math.sqrt(x); // sqrtX = 3.0
```

8. **Math.random()** - restituisce un numero casuale tra 0 e 1.

```
double randomNum = Math.random(); // valore casuale compreso tra 0.0 e
1.0
```

9. **Math.sin(x)** - restituisce il seno di x (in radianti).

```
double x = Math.PI / 2.0;

double sinX = Math.sin(x); // sinX = 1.0
```

10. **Math.cos(x)** - restituisce il coseno di x (in radianti).

```
double x = Math.PI / 2.0;

double cosX = Math.cos(x); // cosX = 6.123233995736766E-17
```

11. System.out.println(**Math.PI**); Calcola il P Grco (¶)

Ci sono molti altri metodi disponibili nella classe Math per eseguire altre operazioni matematiche. Per utilizzare questi metodi, è necessario **importare la classe Math all'inizio del file Java**:

```
import java.lang.Math;
```

Dopo aver importato la classe Math, è possibile utilizzare i suoi metodi come mostrato negli esempi sopra.

Condizioni con IF:

L'istruzione IF è una **struttura condizionale del linguaggio Java**. if (condizione) istruzione1 [else istruzione2] ; Se le istruzioni sono blocchi di istruzioni, devo inserirle tra parentesi graffe.

- **if** (vuol dire se, e esprime una condizione che deve essere soddisfatta)
- **else if** (seconda condizione in caso la prima non dovesse essere soddisfatta il programma controlla la seconda condizione all'interno dell'else if).
- **else** (Ultima condizione che possiamo porre, se le prime due risultassero false passa a quest'ultima).

In Java, l'istruzione IF (che significa "se" in italiano) viene utilizzata per eseguire un blocco di codice solo se una determinata condizione è vera. Ecco la sintassi generale di un'istruzione IF:

```
if (condizione) {

// blocco di codice da eseguire se la condizione è vera

}
```

La "condizione" nell'istruzione IF può essere qualsiasi espressione che restituisce un valore booleano (true o false). Ad esempio, potrebbe essere una semplice variabile booleana, una serie di condizioni logiche combinate con gli operatori AND (&&) o OR (||), una chiamata a un metodo che restituisce un valore booleano, o qualsiasi altra cosa che restituisca un valore booleano.

Ecco un esempio di utilizzo dell'istruzione IF in Java:

```
int x = 10;

if (x > 5) {

System.out.println("x è maggiore di 5");

}
```

In questo esempio, **l'istruzione IF controlla** se la variabile "**x**" è **maggiore** di **5**. Se la condizione è vera, il **blocco di codice** all'interno delle **parentesi** graffe verrà **eseguito** e la **stringa** "**x** è **maggiore** di **5**" verrà **stampata** sulla console.

Puoi anche utilizzare un'istruzione ELSE con l'istruzione IF per specificare un blocco di codice da eseguire se la condizione è falsa.

Ecco un esempio:

```
int y = 2;

if (y > 5) {

System.out.println("y è maggiore di 5");

} else {

System.out.println("y non è maggiore di 5");

}
```

In questo esempio, **l'istruzione IF controlla** se la variabile "**y**" è **maggiore** di **5**. Se la **condizione è vera**, il **blocco** di **codice** all'interno delle **prime parentesi** graffe **verrà eseguito** e la stringa "y è maggiore di 5" verrà stampata sulla console. Se la **condizione** è **falsa**, il **blocco** di **codice** all'interno delle **seconde parentesi** graffe **verrà eseguito** e la stringa "y non è maggiore di 5" verrà stampata sulla console.

Ecco un esempio pratico in cui le condizioni if, else-if e else vengono combinate insieme per determinare se un numero intero è positivo, negativo o zero:

```
import java.util.Scanner;

public class Condizioni {
```

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

      Scanner input = new Scanner(System.in);

      System.out.print("Inserisci un numero intero: ");

      int numero = input.nextInt();

      if (numero > 0) {

      System.out.println(numero + " è un numero positivo");

      } else if (numero < 0) {

      System.out.println(numero + " è un numero negativo");

      } else {

      System.out.println("Il numero inserito è zero");

    }

  }

}
```

In questo esempio, viene chiesto all'utente di inserire un numero intero. Quindi, la condizione if viene utilizzata per determinare se il numero è maggiore di zero. Se questa condizione è vera, viene stampato un messaggio che indica che il numero è positivo.

Se la condizione if non è vera, viene eseguita la condizione else-if, che controlla se il numero è minore di zero. Se questa condizione è vera, viene stampato un messaggio che indica che il numero è negativo.

Infine, se nessuna delle due condizioni precedenti è vera, viene eseguita la condizione else, che indica che il numero inserito è zero e viene stampato un messaggio appropriato.

- if annidati (if dentro l'if):

Un if annidato è una struttura in cui una condizione if è annidata all'interno di un'altra condizione if. Ciò consente di eseguire un'azione diversa in base a diverse combinazioni di condizioni.

Ecco un esempio pratico di if annidato che verifica se un numero è divisibile per 2 e per 3:

```
import java.util.Scanner;

public class IfAnnidato {
```

```
   public static void main(String[] args) {
     Scanner input = new Scanner(System.in);
     System.out.print("Inserisci un numero intero: ");
     int numero = input.nextInt();
     if (numero % 2 == 0) {
      if (numero % 3 == 0) {
      System.out.println(numero + " è divisibile per 2 e per 3");
      } else {
      System.out.println(numero + " è divisibile per 2 ma non per 3");
      }
   } else {
    System.out.println(numero + " non è divisibile per 2");
  }
 }
}
```

In questo esempio, viene chiesto all'utente di inserire un numero intero. Quindi, viene utilizzato un if annidato per determinare se il numero è divisibile per 2 e per 3.

La condizione if esterna verifica se il numero è divisibile per 2, utilizzando l'operatore modulo (%), che restituisce il resto della divisione tra il numero e 2. Se il resto è 0, significa che il numero è divisibile per 2, quindi viene eseguita la condizione if interna.

La condizione if interna verifica se il numero è anche divisibile per 3, utilizzando l'operatore modulo (%). Se il resto è 0, significa che il numero è divisibile sia per 2 che per 3, quindi viene stampato un messaggio che indica che il numero è divisibile per entrambi.

Se la condizione if interna non è vera, viene eseguita la condizione else, che indica che il numero è divisibile solo per 2 e viene stampato un messaggio appropriato.

Se la condizione if esterna non è vera, viene eseguita la condizione else, che indica che il numero non è divisibile per 2 e viene stampato un messaggio appropriato.

Operatore Ternario - ternary operator:

(ESEMPIO: **String numeroMagico= 3 < 10 ? "Corretto" : " Non correto!"**;)
Il primo numero 3 in questo caso, sarà la condizione per true o false Quindi dopo ? se si verifica la condizione vera--- dopo : quando si verifica quella falsa.

L'operatore ternario in Java è un'alternativa concisa alla struttura if-else per effettuare il controllo di una condizione e assegnare un valore in base a essa.

L'operatore ternario ha la seguente sintassi:

```
espressione_booleana ? valore_vero : valore_falso;
```

Se l'espressione booleana è vera, viene restituito il valore vero, altrimenti viene restituito il valore falso.

Ecco un esempio pratico in cui viene utilizzato l'operatore ternario per determinare se un numero è pari o dispari:

```
import java.util.Scanner;

  public class OperatoreTernario {

    public static void main(String[] args) {

      Scanner input = new Scanner(System.in);

      System.out.print("Inserisci un numero intero: ");

      int numero = input.nextInt();

      String risultato = (numero % 2 == 0) ? "pari" : "dispari";

      System.out.println(numero + " è un numero " + risultato);

    }
}
```

In questo esempio, viene chiesto all'utente di inserire un numero intero. Quindi, viene utilizzato l'operatore ternario per determinare se il numero è pari o dispari.

L'espressione booleana verifica se il numero è divisibile per 2, utilizzando l'operatore modulo (%). Se il resto è 0, significa che il numero è pari, quindi viene restituito il valore "pari" e assegnato alla variabile risultato.

Se la condizione non è vera, ovvero se il numero è dispari, viene restituito il valore "dispari" e assegnato alla variabile risultato.

Infine, viene stampato un messaggio che indica se il numero è pari o dispari, utilizzando il valore restituito dall'operatore ternario.

Switch:

L'istruzione SWITCH CASE è una struttura condizionale del linguaggio Java che permette di eseguire diversi blocchi di istruzioni, a seconda del valore dell'espressione di controllo.

Crea un insieme di condizioni, naturalmente al suo interno potremmo annidare degli if e altri come cicli for, while e do-while, o farlo ciclare all'interno di un do-while ad esempio.

La condizione switch in Java è una struttura di controllo del flusso che consente di selezionare un blocco di codice da eseguire tra diverse alternative in base al valore di una variabile o di un'espressione.

La sintassi di base di una dichiarazione switch è la seguente:

```
   switch (espressione) {
   case valore1:
        istruzioni1;
        break;
   case valore2:
        istruzioni2;
        break;
   case valore3:
        istruzioni3;
        break;
   default:
        istruzioni default;
}
```

Consente di scegliere tra diverse opzioni basate su un'espressione o una variabile specificata. In questo modo, è possibile evitare l'uso di più istruzioni "if-else" nidificate.

Ecco un esempio di codice che utilizza la condizione switch in Java:

```
int day = 3;
String dayString;
   switch (day) {
      case 1:
         dayString = "Lunedì";
         break;
      case 2:
         dayString = "Martedì";
         break;
      case 3:
         dayString = "Mercoledì";
         break;
      case 4:
         dayString = "Giovedì";
         break;
      case 5:
         dayString = "Venerdì";
         break;
      case 6:
         dayString = "Sabato";
         break;
      case 7:
         dayString = "Domenica";
```

```
        break;
    default:
        dayString = "Valore non valido";
        break;
}
    System.out.println(dayString);
```

In questo esempio, la variabile "day" viene assegnata al valore intero 3. La condizione switch prende in input questa variabile e sceglie una delle possibili opzioni di case in base al valore di "day".

Nel nostro esempio, il valore 3 corrisponde al caso "Mercoledì". Quindi, la variabile "dayString" viene assegnata a "Mercoledì".

Se il valore di "day" non corrisponde a nessuno dei casi specificati, viene eseguito il codice del caso "default", in questo caso "Valore non valido".

È importante notare che, per ogni caso, è necessario utilizzare l'istruzione "break" per uscire dallo switch dopo che il codice del caso è stato eseguito. Senza l'istruzione "break", lo switch continuerà ad eseguire il codice dei successivi casi, anche se non corrispondono alla variabile di input.

La condizione switch in Java può anche essere utilizzata con tipi di dati diversi dall'intero, come le stringhe o le enumerazioni.

Ecco un altro esempio di codice che utilizza la condizione switch con una stringa:

```
String color = "rosso";
String result;
    switch (color) {
        case "rosso":
            result = "La tua macchina è rossa.";
            break;
        case "blu":
            result = "La tua macchina è blu.";
            break;
```

```
        case "verde":
            result = "La tua macchina è verde.";
            break;
        default:
            result = "Non conosco il colore della tua macchina.";
            break;
    }
    System.out.println(result);
```

In questo caso, la variabile "color" viene assegnata alla stringa "rosso". La condizione switch prende in input questa stringa e sceglie una delle possibili opzioni di case in base al valore di "color".

Nel nostro esempio, il valore "rosso" corrisponde al caso "rosso". Quindi, la variabile "result" viene assegnata a "La tua macchina è rossa".

Come puoi vedere, la condizione switch in Java è un modo utile per gestire diverse opzioni basate

Ciclo For:

Il ciclo for in Java è un'istruzione di controllo del flusso di esecuzione del programma che consente di ripetere un blocco di codice un certo numero di volte.

Ecco un esempio di codice che utilizza il ciclo for in Java:

```
    for (int i = 1; i <= 5; i++) {
        System.out.println("Il valore di i è: " + i);
}
```

In questo esempio, il ciclo for viene utilizzato per stampare i valori di "i" da 1 a 5.

La sintassi del ciclo for è la seguente:

```
for (inizializzazione; condizione; espressione di aggiornamento) {

// blocco di codice da ripetere

}
```

L'inizializzazione viene eseguita solo una volta all'inizio del ciclo e viene utilizzata per inizializzare una variabile di controllo. Nel nostro esempio, la variabile di controllo è "i", che viene inizializzata a 1.

La condizione viene valutata all'inizio di ogni iterazione del ciclo. Se la condizione è vera, il blocco di codice all'interno del ciclo viene eseguito. Nel nostro esempio, la condizione è "i <= 5", che significa "ripeti il ciclo finché il valore di i è minore o uguale a 5".

L'espressione di aggiornamento viene eseguita alla fine di ogni iterazione del ciclo e viene utilizzata per aggiornare la variabile di controllo. Nel nostro esempio, l'espressione di aggiornamento è "i++", che significa "aumenta il valore di i di 1".

Il blocco di codice all'interno del ciclo viene ripetuto fino a quando la condizione non diventa falsa.

Oltre a utilizzare una variabile di controllo numerica, il ciclo for in Java può essere utilizzato anche con altri tipi di dati, come le stringhe o le matrici.

Ecco un altro esempio di codice che utilizza il ciclo for con una matrice:

```
    int[][] matrice = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    for (int riga = 0; riga < 3; riga++) {
       for (int colonna = 0; colonna < 3; colonna++) {
           System.out.print(matrice[riga][colonna] + " ");
     }
     System.out.println();
  }
```

Il codice crea una matrice 3x3 chiamata "matrice" e utilizza il ciclo for in Java per scorrere attraverso ogni elemento della matrice e stamparlo a video.

Nel ciclo for esterno, la variabile "riga" viene utilizzata per scorrere attraverso le righe della matrice. La condizione nel ciclo for esterno è "riga < 3", il che significa che il ciclo for esterno

continuerà ad eseguire fino a quando la variabile "riga" è minore di 3. In altre parole, ci sono tre righe nella matrice e la variabile "riga" viene utilizzata per scorrere attraverso tutte e tre.

All'interno del ciclo for esterno, c'è un altro ciclo for che utilizza la variabile "colonna" per scorrere attraverso le colonne della matrice. La condizione del ciclo for interno è "colonna < 3", il che significa che il ciclo for interno continuerà ad eseguire fino a quando la variabile "colonna" è minore di 3. In altre parole, ci sono tre colonne nella matrice e la variabile "colonna" viene utilizzata per scorrere attraverso tutte e tre.

All'interno del ciclo for interno, viene utilizzata l'istruzione "System.out.print()" per stampare l'elemento corrente della matrice sulla console. L'elemento corrente della matrice viene ottenuto utilizzando le variabili "riga" e "colonna" come indici nella matrice. Ad esempio, l'elemento nella prima riga e nella prima colonna della matrice sarebbe "matrice[0][0]".

Dopo aver stampato tutti gli elementi della riga corrente, viene utilizzata l'istruzione "System.out.println()" per andare a capo e passare alla riga successiva.

Alla fine dell'esecuzione del ciclo for esterno, l'output a video sarà il seguente:

```
1 2 3

4 5 6

7 8 9
```

Come puoi vedere, tutti gli elementi della matrice sono stati stampati in ordine. Questo è solo un esempio di come utilizzare il ciclo for in Java per iterare su una matrice e accedere ai suoi elementi.

Ecco un esempio di codice che utilizza il ciclo for in Java per creare un quadrato vuoto con degli asterischi:

```
int lato = 7;

  for (int riga = 1; riga <= lato; riga++) {

    for (int colonna = 1; colonna <= lato; colonna++) {

      if (riga == 1 || riga == lato || colonna == 1 || colonna == lato) {

          System.out.print("*");

      } else {

      System.out.print(" ");
```

```
    }

  }

  System.out.println();

}
```

In questo esempio, il ciclo for viene utilizzato per iterare sulle righe e sulle colonne del quadrato.

Il valore della variabile "lato" viene utilizzato per determinare la dimensione del quadrato.

All'interno del ciclo for, viene utilizzata un'istruzione condizionale "if" per determinare se la posizione corrente nel quadrato deve essere un asterisco o uno spazio vuoto.

Se la riga o la colonna corrente è la prima o l'ultima, o se la riga corrente è la stessa della colonna corrente, l'istruzione "if" stampa un asterisco. In caso contrario, viene stampato uno spazio vuoto.

Il metodo "println" viene utilizzato alla fine di ogni riga per andare a capo e passare alla riga successiva.

Ecco l'output che verresti a vedere a video:

```
*******

*     *

*     *

*     *

*     *

*     *

*******
```

Come puoi vedere, il quadrato ha solo i lati visibili, con il centro vuoto.

Ciclo for each:

Il ciclo for-each in Java viene utilizzato per iterare attraverso una collezione di oggetti o attraverso un array, fornendo un'alternativa più semplice e leggibile rispetto al ciclo for tradizionale.

La sintassi del ciclo for-each in Java è la seguente:

```
for (tipo nomeVariabile : collezione) {
// istruzioni da eseguire per ogni elemento della collezione
}
```

La variabile "tipo" è il tipo di dati degli elementi nella collezione, **mentre** "nomeVariabile" è il nome della variabile che viene utilizzata per riferirsi all'elemento corrente della collezione.

Ad esempio, se abbiamo un array di interi chiamato "numeri", **possiamo utilizzare il ciclo for-each per scorrere tutti gli elementi dell'array e stamparli a video in questo modo:**

```
int[] numeri = {1, 2, 3, 4, 5};
   for (int numero : numeri) {
      System.out.println(numero);
 }
```

In questo caso, la variabile "numero" viene utilizzata per riferirsi all'elemento corrente dell'array. Il ciclo for-each itera attraverso ogni elemento dell'array, assegnando l'elemento corrente alla variabile "numero" e stampandolo a video con l'istruzione "System.out.println(numero)".

Il ciclo for-each può essere utilizzato anche con altre collezioni, come ad esempio una lista o un insieme di oggetti. Ad esempio, se abbiamo una lista di stringhe chiamata "nomi", possiamo utilizzare il ciclo for-each per scorrere tutti gli elementi della lista e stamparli a video **in questo modo:**

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

In questo caso, la variabile "nome" viene utilizzata per riferirsi all'elemento corrente della lista. Il ciclo for-each itera attraverso ogni elemento della lista, assegnando l'elemento corrente alla variabile "nome" e stampandolo a video con l'istruzione "System.out.println(nome)".

Il ciclo for-each può anche essere utilizzato con oggetti personalizzati, come ad esempio un array di oggetti "Persona". In questo caso, dovremmo assicurarci di definire un metodo "toString()" nella classe "Persona" in modo che possiamo stampare correttamente ogni oggetto durante l'iterazione.
Ad esempio:

```
Persona[] persone = {new Persona("Mario", 35), new Persona("Luigi", 32),
new Persona("Peach", 28)};

   for (Persona persona : persone) {

      System.out.println(persona.toString());

}
```

In questo caso, la variabile "persona" viene utilizzata per riferirsi all'oggetto "Persona" corrente nell'array. Il ciclo for-each itera attraverso ogni oggetto nell'array, assegnando l'oggetto corrente alla variabile "persona" e stampandolo a video con l'istruzione "System.out.println(persona.toString())".

Quadrato vuoto con for each:

non è possibile creare un quadrato vuoto nel centro utilizzando solo un ciclo for-each in Java. Il ciclo for-each è utilizzato principalmente per iterare attraverso le collezioni e gli array, ma non permette di modificare gli elementi all'interno di una collezione o di un array.

Per creare un quadrato vuoto nel centro con degli asterischi, è necessario utilizzare un ciclo for tradizionale come quello che ho mostrato in un esempio precedente. Tuttavia, posso mostrarti come utilizzare un ciclo for-each per stampare tutti gli elementi di un array bidimensionale contenente gli asterischi del quadrato.

Ecco un esempio:

```
char[][] quadrato = {
        {'*', '*', '*', '*', '*', '*'},
        {'*', ' ', ' ', ' ', ' ', '*'},
        {'*', ' ', ' ', ' ', ' ', '*'},
        {'*', ' ', ' ', ' ', ' ', '*'},
        {'*', ' ', ' ', ' ', ' ', '*'},
        {'*', '*', '*', '*', '*', '*'}
```

```
    };
        for (char[] riga : quadrato) {
            for (char carattere : riga) {
                System.out.print(carattere + " ");
            }
            System.out.println();
    }
```

In questo esempio, ho creato un array bidimensionale chiamato "quadrato" che contiene gli asterischi necessari per creare un quadrato vuoto nel centro.

Il ciclo for-each esterno scorre attraverso ogni riga dell'array "quadrato".

Il ciclo for-each interno scorre attraverso ogni carattere all'interno di ogni riga.

Per stampare ogni carattere a video, utilizziamo l'istruzione "System.out.print(carattere + " ")" all'interno del ciclo for-each interno.

L'output di questo programma sarà:

```
* * * * * *
*           *
*           *
*           *
*           *
* * * * * *
```

Come puoi vedere, questo esempio non crea un quadrato vuoto nel centro, ma utilizza il ciclo for-each per stampare tutti gli elementi dell'array bidimensionale contenente gli asterischi del quadrato.

Ciclo While e Do-while:

Il ciclo while e il ciclo do-while sono due costrutti di controllo del flusso in Java che consentono di eseguire ripetutamente un blocco di codice fino a quando una determinata condizione viene soddisfatta.

Il ciclo while esegue il blocco di codice finché la condizione è vera.

La sintassi del ciclo while è la seguente:

```
while (condizione) {

// blocco di codice da eseguire finché la condizione è vera

}
```

Ecco un esempio pratico del ciclo while:

```
int i = 1;

    while (i <= 5) {

    System.out.println("Valore di i: " + i);

    i++;

}
```

In questo esempio, dichiariamo una variabile "i" inizializzata a 1. Il ciclo while esegue il blocco di codice finché "i" è minore o uguale a 5. Durante ogni iterazione del ciclo, stampiamo il valore corrente di "i" a video e incrementiamo "i" di 1.

L'output di questo programma sarà:

```
Valore di i: 1

Valore di i: 2

Valore di i: 3

Valore di i: 4

Valore di i: 5
```

Il ciclo do-while è simile al ciclo while, ma garantisce che il blocco di codice venga eseguito almeno una volta, indipendentemente dal fatto che la condizione sia vera o falsa.

La sintassi del ciclo do-while è la seguente:

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

Ecco un esempio pratico del ciclo do-while:

```
int j = 6;
do {
System.out.println("Valore di j: " + j);
j++;
} while (j <= 5);
```

In questo esempio, dichiariamo una variabile "j" inizializzata a 6. Il blocco di codice all'interno del ciclo do-while viene eseguito almeno una volta, poiché la condizione "j <= 5" non è vera. Durante ogni iterazione del ciclo, stampiamo il valore corrente di "j" a video e incrementiamo "j" di 1. Il ciclo termina quando "j" diventa maggiore di 5.

L'output di questo programma sarà:

```
Valore di j: 6
```

Come puoi vedere, il blocco di codice all'interno del ciclo do-while viene eseguito almeno una volta, indipendentemente dal fatto che la condizione sia vera o falsa.

Array:

Un array in Java è un contenitore che permette di gestire una sequenza di lunghezza fissa di elementi tutti del medesimo tipo. Il numero di elementi in un array, detto lunghezza dell'array, deve essere dichiarato al momento della sua allocazione e non può essere cambiato.

Gli array si possono definire come una collezione di dati, quando in sostanza abbiamo tante variabili come ad esempio il calendario dei voti scolastici divisi per materia, apposto di creare per ogni voto una variabile, con un array avremo una variabile unica che li contiene tutti.

Se volessimo recuperare il loro valori dovremmo partire a contare da 0 per la prima cella e così via in modo crescente, al contrario se volessimo partire dalla fine dovremmo ricordarci di contare un numero in meno della lunghezza totale.

La prima cella dell'array conterrà l'indirizzo per poterlo identificare all'interno della memoria della macchina.

Esempio: int [] votiFisica = {8, 9, 7, 7, 8, 9, 9, 6}

In Java, un array è una struttura di dati che consente di contenere un insieme di valori dello stesso tipo. L'array è un oggetto che può essere utilizzato per organizzare e manipolare una grande quantità di dati in modo efficiente.

Per creare un array in Java, è necessario specificare il tipo di dati che verranno memorizzati nell'array e la dimensione dell'array. La dimensione dell'array rappresenta il numero di elementi che l'array può contenere.

Ecco un esempio di come creare un array di interi in Java:

```
int[] numeri = new int[5];
```

In questo esempio, viene creato un array di interi denominato "numeri" con una dimensione di 5 elementi. Ogni elemento dell'array è inizializzato con il valore predefinito di 0.

Per accedere ai singoli elementi dell'array, è possibile utilizzare un indice che rappresenta la posizione dell'elemento nell'array. L'indice dell'array inizia sempre da 0. Ad esempio, per accedere al terzo elemento dell'array "numeri", **possiamo utilizzare la seguente sintassi:**

```
int terzoNumero = numeri[2];
```

In questo caso, la variabile "terzoNumero" contiene il valore del terzo elemento dell'array "numeri".

È possibile anche inizializzare un array durante la sua creazione. Ad esempio, se vogliamo creare un array di stringhe con i nomi di alcune città italiane**, possiamo utilizzare la seguente sintassi:**

```
String[] cittaItaliane = {"Roma", "Milano", "Napoli", "Firenze",
"Torino"};
```

In questo caso, l'array "cittaItaliane" viene creato con la dimensione necessaria per contenere le cinque stringhe specificate all'interno delle parentesi graffe.

L'array in Java è un'utile struttura dati che può essere utilizzata in molti contesti. Ad esempio, può essere utilizzato per implementare algoritmi di ordinamento, per memorizzare dati di input da un utente, per rappresentare una griglia di valori, per gestire i dati dei database, e molto altro ancora.

Alcuni metodi da utilizzare con gli array:

1. **length:** questo metodo ti permette di ottenere la lunghezza dell'array, ovvero il numero di elementi al suo interno.
   Ad esempio:

```
int[] numeri = {1, 2, 3, 4, 5};

int lunghezzaArray = numeri.length; // restituisce 5
```

In questo esempio, abbiamo creato un array di interi con cinque elementi. Abbiamo quindi utilizzato il metodo length per ottenere la lunghezza dell'array e assegnarla alla variabile lunghezzaArray. La variabile lunghezzaArray conterrà quindi il valore 5, poiché ci sono cinque elementi nell'array.

2. **sort:** questo metodo ti permette di ordinare gli elementi dell'array in ordine crescente o decrescente.
   Ad esempio:

```
int[] numeri = {5, 2, 4, 1, 3};

Arrays.sort(numeri); // ordina gli elementi in ordine crescente

System.out.println(Arrays.toString(numeri)); // [1, 2, 3, 4, 5]
```

In questo esempio, abbiamo creato un array di interi non ordinati. Abbiamo quindi utilizzato il metodo sort per ordinarli in ordine crescente. Dopo aver ordinato l'array, abbiamo utilizzato il metodo toString per visualizzare i suoi elementi. L'output del programma sarà [1, 2, 3, 4, 5].

3. **copyOf:** questo metodo ti permette di creare una copia dell'array con una lunghezza specificata.
   Ad esempio:

```
int[] numeri = {1, 2, 3, 4, 5};

int[] numeriCopia = Arrays.copyOf(numeri, 3); // crea una copia con i
primi tre elementi

System.out.println(Arrays.toString(numeriCopia)); // [1, 2, 3]
```

In questo esempio, abbiamo creato un array di interi con cinque elementi. Abbiamo quindi utilizzato il metodo copyOf per creare una copia dell'array con i primi tre elementi. Dopo aver creato la copia, abbiamo utilizzato il metodo toString per visualizzarla. L'output del programma sarà [1, 2, 3], poiché la copia contiene solo i primi tre elementi dell'array originale.

4. **binarySearch:** questo metodo ti permette di cercare un elemento specifico all'interno dell'array. Tuttavia, l'array deve essere ordinato in precedenza per poter utilizzare questo metodo correttamente.
   Ad esempio:

```
int[] numeri = {1, 2, 3, 4, 5};

int posizione = Arrays.binarySearch(numeri, 3); // cerca il valore 3 e restituisce la posizione 2
```

In questo esempio, abbiamo creato un array di interi ordinati. Abbiamo quindi utilizzato il metodo binarySearch per cercare il valore 3 all'interno dell'array. Il metodo binarySearch restituisce la posizione dell'elemento cercato se lo trova, altrimenti restituisce un valore negativo che indica che l'elemento non è presente nell'array. Nel nostro caso, poiché il valore 3 è presente nell'array, la variabile posizione conterrà il valore 2.

5. **fill:** questo metodo ti permette di riempire l'array con un valore specifico.
   Ad esempio:

```
int[] numeri = new int[5];

Arrays.fill(numeri, 0); // riempie l'array con il valore 0

System.out.println(Arrays.toString(numeri)); // [0, 0, 0, 0, 0]
```

In questo esempio, abbiamo creato un nuovo array di interi con una lunghezza di 5 elementi. Abbiamo quindi utilizzato il metodo fill per riempire l'array con il valore 0. Dopo aver riempito l'array, abbiamo utilizzato il metodo toString per visualizzare i suoi elementi. L'output del programma sarà [0, 0, 0, 0, 0].

Esempi di come utilizzare i cicli for insieme ai metodi degli array:

1. **Utilizzo del ciclo for-each:**

```
int[] numeri = {1, 2, 3, 4, 5};

    for (int numero : numeri) {

        System.out.print(numero" ");
```

```
}
```

In questo esempio, abbiamo creato un array di interi con cinque elementi. Abbiamo quindi utilizzato il ciclo for-each per iterare attraverso gli elementi dell'array e stamparli uno per uno. Durante ogni iterazione del ciclo, la variabile numero assume il valore dell'elemento corrente dell'array.

L'output del programma sarà:

```
1 2 3 4 5
```

La stampa ritornerà i numeri in orizzontale con uno spazio tra uno e l'altro.

2. **Utilizzo del ciclo for con contatore**:

```
int[] numeri = {1, 2, 3, 4, 5};

for (int i = 0; i < numeri.length; i++) {

System.out.print(numeri[i] + " ");

}
```

In questo esempio, abbiamo creato un array di interi con cinque elementi. Abbiamo quindi utilizzato il ciclo for con contatore per iterare attraverso gli elementi dell'array e stamparli uno per uno. Durante ogni iterazione del ciclo, la variabile i assume il valore del contatore corrente, che parte da 0 e termina alla lunghezza dell'array meno 1.

L'output del programma sarà lo stesso dell'esempio precedente:

```
1 2 3 4 5
```

La stampa ritornerà i numeri in orizzontale con uno spazio tra uno e l'altro.

3. **Utilizzo del ciclo for-each per calcolare la somma degli elementi dell'array:**

```
int[] numeri = {1, 2, 3, 4, 5};

int somma = 0;

for (int numero : numeri) {

somma += numero;

}

System.out.println(somma);
```

In questo esempio, abbiamo creato un array di interi con cinque elementi. Abbiamo quindi utilizzato il ciclo for-each per iterare attraverso gli elementi dell'array e calcolare la somma di tutti i numeri. Durante ogni iterazione del ciclo, la variabile numero assume il valore dell'elemento corrente dell'array, che viene poi aggiunto alla variabile somma. Dopo aver completato l'iterazione attraverso tutti gli elementi dell'array, la variabile somma conterrà il valore della somma totale degli elementi.

L'output del programma sarà:

```
15
```

4. **Utilizzo di un ciclo for per trovare il valore massimo in un array:**

```
int[] numeri = {1, 5, 3, 4, 2};

int massimo = numeri[0];

for (int i = 1; i < numeri.length; i++) {

if (numeri[i] > massimo) {

massimo = numeri[i]; // aggiorna il massimo se l'elemento corrente è maggiore

}

}

System.out.println("Il valore massimo è: " + massimo); // Il valore massimo è: 5
```

In questo esempio, abbiamo creato un array di interi non ordinati. Abbiamo quindi utilizzato un ciclo for per trovare il valore massimo all'interno dell'array. Abbiamo inizializzato la variabile massimo con il primo elemento dell'array e poi confrontato ogni elemento successivo con il valore corrente di massimo. Se l'elemento corrente è maggiore di massimo, aggiorniamo la variabile massimo con il valore dell'elemento corrente.

Alla fine del ciclo for, la variabile massimo conterrà il valore massimo nell'array. Abbiamo quindi utilizzato il metodo println per stampare il valore massimo sullo schermo. In questo caso, l'output del programma sarà "Il valore massimo è: 5".

5. **Utilizzo di un ciclo for per calcolare la somma degli elementi di un array:**

```
int[] numeri = {1, 2, 3, 4, 5};

int somma = 0;
```

```
for (int i = 0; i < numeri.length; i++) {

somma += numeri[i]; // aggiungi l'elemento corrente alla somma

}

System.out.println("La somma degli elementi è: " + somma); // La somma degli elementi è: 15
```

In questo esempio, abbiamo creato un array di interi e utilizzato un ciclo for per calcolare la somma di tutti gli elementi nell'array. Abbiamo inizializzato la variabile somma a zero all'inizio del ciclo, e poi aggiunto l'elemento corrente ad essa in ogni iterazione del ciclo.

6. **Utilizzo di un ciclo for per calcolare la media degli elementi di un array:**

```
int[] numeri = {1, 2, 3, 4, 5};

double media = 0;

for (int i = 0; i < numeri.length; i++) {

media += numeri[i];

}

media /= numeri.length; // dividi la somma per il numero di elementi

System.out.println("La media degli elementi è: " + media); // La media degli elementi è: 3.0
```

In questo esempio, abbiamo utilizzato un ciclo for per calcolare la somma di tutti gli elementi nell'array, come nel primo esempio. Tuttavia, invece di stampare la somma, abbiamo diviso la somma per il numero di elementi nell'array per ottenere la media. Abbiamo inizializzato la variabile media come un valore double a zero, poiché il risultato potrebbe essere un valore decimale.

7. **Utilizzo di un ciclo for per copiare gli elementi di un array in un altro array:**

```
int[] numeri1 = {1, 2, 3, 4, 5};

int[] numeri2 = new int[numeri1.length];

for (int i = 0; i < numeri1.length; i++) {

numeri2[i] = numeri1[i]; // copia l'elemento corrente in numeri2

}
```

```
System.out.println("numeri1: " + Arrays.toString(numeri1)); // numeri1: [1, 2, 3, 4, 5]

System.out.println("numeri2: " + Arrays.toString(numeri2)); // numeri2: [1, 2, 3, 4, 5]
```

In questo esempio, abbiamo creato un array numeri1 di interi e un altro array numeri2 vuoto con la stessa lunghezza di numeri1. Abbiamo quindi utilizzato un ciclo for per copiare tutti gli elementi di numeri1 in numeri2. In ogni iterazione del ciclo, abbiamo copiato l'elemento corrente da numeri1 in numeri2. Alla fine del ciclo, entrambi gli array avranno gli stessi valori.

Nota che in questo esempio abbiamo utilizzato il metodo toString della classe Arrays per stampare gli array sullo schermo in modo leggibile**.**

Inoltre, abbiamo utilizzato un ciclo per tradizionale per iterare attraverso gli elementi degli array e copiare gli elementi corrispondenti in numeri2.

L'output del programma è il seguente:

```
numeri1: [1, 2, 3, 4, 5]

numeri2: [1, 2, 3, 4, 5]
```

Array 2D:

Gli array bidimensionali o matrici sono array di array.

A differenza degli array classici, possono essere trattati con delle righe e delle colonne, l'array principale il primo fungerà da riga e il secondo contenuto nel primo fungerà da colonne.

Anche in questi si parte a contare da 0 sia per righe che per le colonne.

Un array 2D in Java è una matrice, ovvero una struttura di dati bidimensionale che consente di organizzare e manipolare una quantità di dati in due dimensioni. In una matrice, i dati sono organizzati in righe e colonne. Ogni elemento della matrice è identificato dalla sua posizione, indicata da due indici: uno per la riga e uno per la colonna.

Per creare un array 2D in Java, è necessario specificare il tipo di dati che verranno memorizzati nell'array, il numero di righe e il numero di colonne.

La sintassi per creare un array 2D in Java è la seguente:

```
tipoDati[][] nomeArray = new tipoDati[numeroRighe][numeroColonne];
```

Ad esempio, per creare una matrice di interi 3x3, è possibile utilizzare la seguente sintassi:

```
int[][] matrice = new int[3][3];
```

In questo caso, la matrice "matrice" viene creata con tre righe e tre colonne, ognuna delle quali è inizializzata con il valore predefinito di 0.

Per accedere ai singoli elementi della matrice, è possibile utilizzare due indici: uno per la riga e uno per la colonna. Ad esempio, per accedere all'elemento nella seconda riga e nella terza colonna della matrice "matrice", **possiamo utilizzare la seguente sintassi:**

```
int valore = matrice[1][2];
```

In questo caso, la matrice "nomi" viene creata con due righe e tre colonne, e ogni elemento della matrice viene inizializzato con una stringa specificata all'interno delle parentesi graffe.

L'array 2D in Java è utile per rappresentare dati strutturati in modo bidimensionale, come ad esempio una tabella o una griglia di valori. Può essere utilizzato per rappresentare immagini, per memorizzare dati di una tabella di database, per implementare algoritmi di elaborazione di immagini e molto altro ancora.

Metodi da utilizzare con array 2D (bidimensionali):

Per creare un array bidimensionale, devi prima dichiarare il tipo dell'array (ad esempio int, double, String, ecc.) e quindi specificare le dimensioni dell'array (cioè il numero di righe e il numero di colonne).

1) **Creazione di un array 2D vuoto:**

```
int[][] array2D = new int[3][4];

System.out.println(Arrays.deepToString(array2D));
```

In questo punto abbiamo creato un array 2D vuoto con 3 righe e 4 colonne utilizzando il costruttore di default. Quindi abbiamo utilizzato il metodo 'deepToStringArrays per stampare l'array sullo schermo in modo leggibile.

Output:

```
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

2) **Inizializzazione di un array bidimensionale:**

Puoi inizializzare un array bidimensionale in modo simile a un array unidimensionale, specificando i valori degli elementi all'interno delle parentesi graffe.
Ad esempio:

```
int[][] array2D = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12}};

System.out.println(Arrays.deepToString(array2D));
```

In questo punto abbiamo creato un array 2D vuoto con 3 righe e 4 colonne utilizzando il costruttore di default.

Quindi abbiamo utilizzato il metodo 'deepToStringArrays per stampare l'array sullo schermo in modo leggibile.

Output:

```
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

3) **Accesso agli elementi di un array bidimensionale:**
Per accedere agli elementi di un array bidimensionale, devi specificare l'indice della riga e l'indice della colonna dell'elemento desiderato.
Ad esempio:

```
int[][] array2D = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12}};

int elemento = array2D[2][1]; // ottiene l'elemento nella seconda riga e
terza colonna (ovvero il valore 8)

System.out.println(elemento); // stampa 8
```

In questo punto abbiamo acceduto all'elemento nella terza riga e nella seconda colonna dell'array 2D e lo abbiamo salvato nella variabile 'elemento.

Output:

```
8
```

4) **Modifica di un elemento specifico dell'array 2D:**

```
int[][] array2D = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12}};

array2D[2][1] = 99;

System.out.println(Arrays.deepToString(array2D));
```

In questo punto abbiamo modificato l'elemento nella terza riga e nella seconda colonna dell'array 2D e abbiamo assegnato il valore 99 all'elemento.

Quindi abbiamo utilizzato il metodo 'deepToString per stampare l'array sullo schermo in modo leggibile.

Output:

```
[[1, 2, 3], [4, 5, 6], [7, 99, 9], [10, 11, 12]]
```

5) **Calcolo della somma degli elementi di un array bidimensionale:**

```
int[][] matrice = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

int somma = 0;

      for (int riga = 0; riga < matrice.length; riga++) {

          for (int colonna = 0; colonna < matrice[riga].length;
colonna++) {

            somma += matrice[riga][colonna];

    }

 }

   System.out.println(somma); // stampa "45"
```

La matrice è stata definita utilizzando il seguente codice:

int[][] matrice = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

Questa istruzione crea una matrice 3x3 contenente i numeri da 1 a 9. La matrice è rappresentata come un array di array, dove ogni riga è un array separato. Ad esempio, la prima riga della matrice (1, 2, 3) è rappresentata dall'array matrice[0].

Calcolo della somma:

La somma di tutti i valori nella matrice viene calcolata utilizzando un doppio ciclo for annidato. Il primo ciclo for scorre le righe della matrice, mentre il secondo ciclo for scorre le colonne di ogni riga. Per ogni valore trovato nella matrice, la somma viene aggiornata con la seguente **istruzione:**

somma += matrice[riga][colonna];

Questa istruzione aggiunge il valore dell'elemento corrente alla variabile somma**.**

Stampa del risultato:

Dopo che il ciclo for ha completato il calcolo della somma, il risultato viene stampato a video utilizzando l'istruzione: **[T:System.out.println]**

Questo codice stampa il valore della variabile somma, che nel nostro esempio è 45.

Prodotto:

L'output di questo programma sarà "45", in quanto la somma di tutti i numeri nella matrice è 1+2+3+4+5+6+7+8+9 = 45.

Output:

```
45
```

6) **Ricerca del valore massimo in un array bidimensionale:**

```
public class ArrayBidimensionale {

   public static void main(String[] args) {

        int[][] array = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

        int max = array[0][0];

        for (int i = 0; i < array.length; i++) {

          for (int j = 0; j < array[i].length; j++) {

             if (array[i][j] > max) {

                  max = array[i][j];

             }

          }

      }

      System.out.println("Il valore massimo nell'array è: " + max);

  }

}
```

In questo programma, abbiamo una matrice 3x3 (o array bidimensionale) contenente i numeri da 1 a 9. Il nostro obiettivo è quello di cercare il valore massimo in questa matrice.

Il processo per trovare il valore massimo è il seguente:

1. **Definizione della matrice bidimensionale**

La matrice è stata definita utilizzando il seguente codice**:**

```
int[][] array = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

2. **Inizializzazione del valore massimo:**

   Abbiamo inizializzato la variabile max con il primo valore dell'array (array[0][0]):

```
int max = array[0][0];
```

3. **Scansione dell'array:**

   Abbiamo eseguito un ciclo for annidato per scorrere tutti i valori dell'array:

```
for (int i = 0; i < array.length; i++) {
   for (int j = 0; j < array[i].length; j++) {
   // codice di ricerca del massimo
  }
}
```

4. **Confronto con il valore massimo corrente:**

   Per ogni elemento dell'array, abbiamo confrontato il valore corrente con il valore massimo corrente. Se il valore corrente è maggiore del valore massimo corrente, abbiamo aggiornato il valore massimo:

```
if (array[i][j] > max) {
max = array[i][j];
}
```

5. **Stampa del risultato:**

   Dopo che il ciclo for ha completato la ricerca del valore massimo, abbiamo stampato il risultato:

```
System.out.println("Il valore massimo nell'array è: " + max);
```

Prodotto:

L'output di questo programma sarà "Il valore massimo nell'array è: 9", in quanto 9 è il valore massimo presente nella matrice.

Output:

```
Il valore massimo nell'array è: 9
```

7) **Iterazione attraverso un array bidimensionale:**
Per iterare attraverso un array bidimensionale, puoi utilizzare un ciclo for nidificato, in cui il primo ciclo scorre le righe e il secondo ciclo scorre le colonne.
Ad esempio:

```
int[][] array2d = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

for (int riga = 0; riga < array2d.length; riga++) {

for (int colonna = 0; colonna < array2d[riga].length; colonna++) {

int elemento = array2d[riga][colonna];

System.out.print(elemento + " ");

}

System.out.println(); // va a capo dopo ogni riga

}
```

1. **Definizione della matrice bidimensionale:**

   La matrice bidimensionale è stata definita utilizzando il seguente codice:

```
int[][] array2d = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

2. **Scansione dell'array:**

   Abbiamo eseguito un ciclo for annidato per scorrere tutti gli elementi dell'array:

```
for (int riga = 0; riga < array2d.length; riga++) {
   for (int colonna = 0; colonna < array2d[riga].length; colonna++) {
        int elemento = array2d[riga][colonna];
        // codice per l'elaborazione dell'elemento
   }
}
```

3. **Accesso all'elemento corrente:**

   Per accedere all'elemento corrente nell'array bidimensionale, abbiamo utilizzato la notazione a doppia parentesi quadre:

```
int elemento = array2d[riga][colonna];
```

4. **Elaborazione dell'elemento:**

   Dopo aver acceduto all'elemento corrente, abbiamo eseguito il codice necessario per elaborare l'elemento. Nel nostro caso, abbiamo semplicemente stampato il valore dell'elemento:

```
System.out.print(elemento + " ");
```

5. **Stampa dell'output:**

   Dopo che il ciclo for ha completato la scansione dell'array, abbiamo stampato l'output utilizzando il seguente codice:

```
System.out.println(); // va a capo dopo ogni riga
```

   Questo codice serve per andare a capo dopo ogni riga stampata. L'output di questo programma sarà il seguente:

Output:

```
1 2 3

4 5 6

7 8 9
```

In questo output, abbiamo stampato la matrice bidimensionale con uno spazio tra ogni elemento e andando a capo dopo ogni riga. Questo è stato possibile grazie all'iterazione attraverso l'array bidimensionale utilizzando il ciclo for nidificato.

Metodi delle stringhe:

Metodo .equals → ci ritornerà un risultato di tipo boleano

quello che troviamo scritto in .equals "....." dovrà coincidere perfettamente con quello scritto nella stringa stabile anche le lettere maiuscole o minuscole ci darebbero errore in stampa.

```
String s1 = "ciao";

String s2 = "ciao";

String s3 = "Ciao";

boolean result1 = s1.equals(s2); // restituisce true

boolean result2 = s1.equals(s3); // restituisce false
```

Metodo **.equalsIgnoreCase** → ignora maiuscole o minuscole

Per ovviare al problema delle lettere maiuscole o minuscole, basterà aggiungere equals.IgnoreCase e successivamente mandare a stampa.

```
String s1 = "ciao";

String s2 = "CIAO";

boolean result = s1.equalsIgnoreCase(s2); // restituisce true
```

Metodo **.length** → ritornerà la lunghezza della stringa= quindi useremo una variabile int perché numero intero.

```
String s = "ciao";

int length = s.length(); // restituisce 4
```

Metodo **.charAt** → Ci comunica in quale posizione si trova un determinato carattere= variabile char perché definisce un solo carattere.

```
String s = "ciao";

char firstChar = s.charAt(0); // restituisce 'c'
```

Metodo **.indexOf** → Ci restituirà l'indice = useremo la variabile int perché numero intero.

```
String s = "ciao mondo";

int index = s.indexOf("mon"); // restituisce 5
```

Metodo **.isEmpty** → Ci riporterà se la stringa è vuota = useremo variabile booleana perché può solo essere vero o falso.

```
String s1 = "";

String s2 = "ciao";

boolean result1 = s1.isEmpty(); // restituisce true

boolean result2 = s2.isEmpty(); // restituisce false
```

Metodo .toUpperCase → Ci restituirà quello che c'è nel case della stringa = toUpperCase (**maiuscolo**).

```
String s = "ciao mondo";

String uppercase = s.toUpperCase(); // restituisce "CIAO MONDO"
```

Metodo **.toLowerCase** → Ci restituirà quello che c'è nel case della stringa = toLowerCase (**minuscolo**).

```
String s = "CIAO MONDO";

String lowercase = s.toLowerCase(); // restituisce "ciao mondo"
```

Per entrambi, utilizzeremo variabile stringa perché composto da una stringa "testo".

Metodo .trim → questo metodo ci permette di rimuovere gli spazi davanti e dopo una stringa.

```
String s = " ciao mondo ";

String trimmed = s.trim(); // restituisce "ciao mondo"
```

Metodo .replace → ci permette di sostituire dei caratteri all'interno della stringa.

```
String s = "ciao mondo";

String replaced = s.replace("o", "a"); // restituisce "caia manda"
```

Wrapper Class:

In varie situazioni, può essere comodo poter trattare i tipi primitivi come oggetti.
Una classe wrapper (involucro) incapsula una variabile di un tipo primitivo.
In qualche modo “trasforma” un tipo primitivo in un oggetto equivalente.

La differenza tra dati primitivi Es: int numero= 1; non ci permettono di utilizzare dei metodi || Invece reference come le Stringhe Es: String x = "Alessandro"; ci permetterà di utilizzare dei metodi.

Le wrapper class saranno però molto più lente delle primitive (PERCHÉ TRATTATE COME PRIMITIVE MA IMPACCHETTATE IN REFERENCE (Autoboxing CHE DARA QUINDI LA POSSIBILITÀ DI AVERE ACCESSO AI METODI, PER POI SPACCHETTARLE (Unboxing) PER RENDERE IL VALORE PRIMITIVO CHE SAREBBE IN FINE QUELLO REALE).

Nella tabella sottostante sono riportati i tipi primitivi e le relative classi wrapper**:**
Integer: incapsula
Double: incapsula un valore in virgola mobile a doppia precisione.
Float: incapsula un valore in virgola mobile a singola precisione.
Long: incapsula un valore intero lungo.
Short: incapsula un valore intero breve.
Byte: incapsula un valore di tipo byte.
Boolean: incapsula
Character: incapsula un singolo carattere Unicode.

Classi wrapper - Funzionamento :

- Ogni classe wrapper ha come stato semplicemente un attributo del tipo che incapsula: Integer avrà un attributo di tipo int, Double un attributo di tipo double e così via.
- Le classi wrapper sono state costruite per essere immutabili: assumono un valore al momento della creazione e non lo cambiano mai più.
- Quindi per ogni classe esiste un costruttore che prende come parametro una variabile del tipo incapsulato e lo memorizza nello stato.
- Esiste poi un metodo selettore che consente di leggere in modo protetto il valore dello stato.

Classi wrapper – Funzionamento:

- Ogni classe wrapper definisce metodi per estrarre il valore della variabile incapsulata e viceversa.
  Per estrarre il valore incapsulato:
- **Integer** fornisce il metodo intValue()
- **Double** fornisce il metodo doubleValue()
- **Boolean** fornisce il metodo booleanValue()
- **Character** fornisce il metodo charValue() §
- ...
  Per creare un oggetto da un valore primitivo:
- **Integer** i = new Integer(valore int)
- **Double** d = new Double(valore double)
- **Boolean** b = new Boolean(valore boolean)
- **Character** c = new Character(valore char)
- ...
-

Ecco alcuni esempi pratici di come utilizzare le wrapper class in Java:

1. **Numero intero La classe Integer è la wrapper class per il tipo di dato primitivo int.**
   Ecco alcuni esempi di utilizzo:
   a) **Conversione di un int in un oggetto Integer:**

```
int x = 5;

Integer y = Integer.valueOf(x); // conversione di un int in un oggetto
Integer
```

b) Conversione di una stringa in un oggetto Integer:

```
String str = "10";

Integer num = Integer.valueOf(str); // conversione di una stringa in un
oggetto Integer
```

c) Esecuzione di operazioni matematiche con gli oggetti Integer:

```
Integer a = 5;

Integer b = 10;

Integer c = a + b; // esecuzione di una somma tra oggetti Integer
```

2. **Doppio La classe Double è la Wrapper class per il tipo di dato primitivo double.**
   Ecco alcuni esempi di utilizzo:

a) Conversione di un double in un oggetto Double:

```
double x = 5.5;

Double y = Double.valueOf(x); // conversione di un double in un oggetto
Double
```

b) Conversione di una stringa in un oggetto Double:

```
String str = "10.5";

Double num = Double.valueOf(str); // conversione di una stringa in un
oggetto Double
```

c) Esecuzione di operazioni matematiche con gli oggetti Double:

```
Double a = 5.5;

Double b = 10.5;

Double c = a + b; // esecuzione di una somma tra oggetti Double
```

3. **Booleano La classe:**
   Boolean è la wrapper class per il tipo di dato primitivo boolean.
   Ecco alcuni esempi di utilizzo:

   a) **Conversione di un boolean in un oggetto Boolean:**

```
boolean x = true;

Boolean y = Boolean.valueOf(x); // conversione di un boolean in un
oggetto Boolean
```

   b) **Conversione di una stringa in un oggetto Boolean:**

```
String str = "true";

Boolean bool = Boolean.valueOf(str); // conversione di una stringa in un
oggetto Boolean
```

   c) **Conversione di una stringa in un oggetto Boolean:**

```
String str = "true";

Boolean bool = Boolean.valueOf(str); // conversione di una stringa in un
oggetto Boolean
```

Array List e Array List 2D:

A Differenza degli array classici a cui viene fissata una dimensione che non è modificabile negli ArrayList invece si può modificare la dimensione= aggiungere, modificare o rimuovere gli elementi degli stessi.

Questi ArrayList accetteranno solo però dei dati di tipo reference (Wrapper).

ANCHE LORO CONTANO LE CELLE PARTENDO DA 0.

In Java, un ArrayList è una classe che implementa l'interfaccia List e utilizza un array per memorizzare gli elementi. Tuttavia, a differenza degli array standard, gli ArrayList possono essere ridimensionati dinamicamente in modo da poter aggiungere o rimuovere elementi in modo efficiente senza dover preallocare memoria.

I. **Ecco un esempio di come creare un ArrayList di interi in Java:**

```
import java.util.ArrayList;

public class Main {
```

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

ArrayList<Integer> numeri = new ArrayList<Integer>();

numeri.add(10);

numeri.add(20);

numeri.add(30);

System.out.println(numeri);

}

}
```

In questo esempio, abbiamo creato un ArrayList chiamato "numeri" che contiene interi. **Abbiamo** quindi aggiunto tre numeri all'ArrayList utilizzando il metodo "add()", che aggiunge l'elemento specificato alla fine dell'ArrayList.

II. **Per accedere agli elementi dell'ArrayList, possiamo utilizzare il metodo "get()", che restituisce l'elemento all'indice specificato:**

```
int primoNumero = numeri.get(0);

System.out.println(primoNumero); // Output: 10
```

III. **Possiamo anche rimuovere elementi dall'ArrayList utilizzando il metodo "remove()", che rimuove l'elemento specificato dall'ArrayList:**

```
numeri.remove(1);

System.out.println(numeri); // Output: [10, 30]
```

IV. **Un altro utilizzo comune degli ArrayList in Java è l'iterazione attraverso gli elementi dell'ArrayList utilizzando un ciclo "for-each":**

```
for (int numero : numeri) {

System.out.println(numero);

}
```

Questo ciclo for-each scorre tutti gli elementi nell'ArrayList "numeri" e stampa ciascun elemento sulla console.

Gli ArrayList sono estremamente utili perché consentono di memorizzare una collezione di oggetti di qualsiasi tipo in modo dinamico.

V. **Ecco un esempio pratico di come creare un ArrayList di oggetti String e aggiungere o rimuovere elementi dall'ArrayList:**

```
public class EsempioArrayList {

    public static void main(String[] args) {

        // creiamo un nuovo ArrayList di oggetti String

        ArrayList<String> elencoNomi = new ArrayList<>();

        // aggiungiamo alcuni nomi all'ArrayList

        elencoNomi.add("Mario");

        elencoNomi.add("Luigi");

        elencoNomi.add("Principessa Peach");

        elencoNomi.add("Yoshi");

        // stampiamo il numero di elementi presenti nell'ArrayList

        System.out.println("Numero di elementi presenti nell'ArrayList: "

        + elencoNomi.size());

        // stampiamo tutti i nomi presenti nell'ArrayList

        System.out.println("Nomi presenti nell'ArrayList: " +

        elencoNomi);

        // rimuoviamo il nome "Principessa Peach" dall'ArrayList

        elencoNomi.remove("Principessa Peach");

        // stampiamo tutti i nomi presenti nell'ArrayList dopo la

        rimozione

        System.out.println("Nomi presenti nell'ArrayList dopo la

        rimozione: " + elencoNomi);

        // rimuoviamo tutti gli elementi dall'ArrayList
```

```
        elencoNomi.clear();

        // stampiamo il numero di elementi presenti nell'ArrayList dopo la
        rimozione

        System.out.println("Numero di elementi presenti nell'ArrayList
        dopo la rimozione: " + elencoNomi.size());

    }

}
```

Questo programma crea un nuovo ArrayList di oggetti String chiamato , aggiunge alcuni nomi all'ArrayList, stampa il numero di elementi presenti nell'ArrayList, stampa tutti i nomi presenti nell'ArrayList, rimuove il nome "Principessa Peach" dall'ArrayList, stampa tutti i nomi presenti nell'ArrayList dopo la rimozione, rimuove tutti gli elementi dall'ArrayList e infine stampa il numero di elementi presenti nell'ArrayList dopo la rimozione.

Questo è solo un esempio di come gli ArrayList possono essere utilizzati in Java.elencoNomi.

VI. Esempio di utilizzo del metodo 'size()

Il metodo 'size restituisce il numero di elementi presenti nell'ArrayList. Vediamo un esempio di come utilizzarlo:

```
public class EsempioSizeArrayList {

    public static void main(String[] args) {

        ArrayList<String> lista = new ArrayList<>();

        lista.add("Uno");

        lista.add("Due");

        lista.add("Tre");

        int numeroElementi = lista.size();

        System.out.println("Il numero di elementi presenti nell'ArrayList
        è: " + numeroElementi);

    }

}
```

In questo esempio, abbiamo creato un ArrayList di stringhe e abbiamo aggiunto 3 elementi utilizzando il metodo add(). Successivamente, abbiamo utilizzato il metodo 'sizesize() per determinare il numero di elementi nell'ArrayList. Infine, abbiamo stampato il numero di elementi a console con un messaggio esplicativo.

Prodotto:

```
Il numero di elementi presenti nell'ArrayList è: 3
```

VII. Esempio di utilizzo del metodo 'clear()

Esempio:

```
public class EsempioClearArrayList {

    public static void main(String[] args) {

        ArrayList<String> lista = new ArrayList<>();

        lista.add("Uno");

        lista.add("Due");

        lista.add("Tre");

        System.out.println("Lista prima della rimozione: " + lista);

        lista.clear();

        System.out.println("Lista dopo la rimozione: " + lista);

    }

 }
```

In questo esempio, abbiamo creato un ArrayList di stringhe e abbiamo aggiunto 3 elementi utilizzando il metodo 'addtoString() implicito di ArrayList. Quindi abbiamo utilizzato il metodo 'clearclear() per rimuovere tutti gli elementi dall'ArrayList. Infine, abbiamo stampato l'ArrayList dopo la rimozione utilizzando nuovamente il metodo 'toStringtoString().

Prodotto:

Nella prima stampa, l'output mostra l'ArrayList originale con i suoi 3 elementi:

```
Lista prima della rimozione: [Uno, Due, Tre]
```

Successivamente, abbiamo utilizzato il metodo 'clear per rimuovere tutti gli elementi dall'ArrayList. Quando stampiamo l'ArrayList dopo la rimozione, vediamo che l'ArrayList è vuoto:

```
Lista dopo la rimozione: []
```

Metodi:

Chiamati metodi o funzioni= di base i metodi sono delle funzioni, dei blocchi di codice che ci permettono di scrivere del codice e richiamarlo ogni volta che ne abbiamo bisogno.
Questo sarà possibile solo inserendo il nome che avremmo assegnato al codice in questione.
Sono chiamati così perché ogni funzione che scriviamo dentro una classe "public class Main" viene definita metodo (Questo richiama il costrutto di java della programmazione a oggetti).

I parametri= sono dei dati in entrata, dei dati che noi passiamo volta per volta alla funzione.

Override:

Mentre l'override sarebbe un esempio di polimorfismo dinamico

Nella programmazione ad oggetti, override funzionalità che utilizza il polimorfismo di runtime, è l'operazione di riscrittura di un metodo ereditato.
Nella programmazione ad oggetti assume notevole importanza la possibilità di creare classi a partire da classi già esistenti (ereditarietà).
Il polimorfismo dinamico (runtime) è il polimorfismo esistente in fase di esecuzione. Qui, il compilatore Java non capisce quale metodo viene chiamato al momento della compilazione. Solo JVM decide quale metodo viene chiamato in fase di esecuzione. L'overloading del metodo e l'override del metodo utilizzando i metodi di istanza sono gli esempi di polimorfismo dinamico.

Overload di Metodi:

L'overloading del metodo sarebbe un esempio di polimorfismo statico.

Attraverso l'Overload si possono creare metodi completamente identici ma con parametri differenti, questo è imperativo se usassimo gli stessi parametri risulterebbe un errore.
In sostanza gli overloaded methods= un metodo con lo stesso nome ripetuto più volte la cui signature (firma) ciò che caratterizza il metodo, la firma (signature) del metodo che è la combinazione di parametri dovrà essere diversa categoricamente in tutti, oppure cambia il tipo di dato, o ancora la combinazione di tipi di dato differenti Es: double e int - int e float e così via.

Il polimorfismo statico (tempo di compilazione) è il polimorfismo esibito al momento della compilazione. Qui, il compilatore Java sa quale metodo viene chiamato. Overloading del metodo e override del metodo utilizzando metodi statici; override del metodo utilizzando metodi privati o finali sono esempi di statici polimorfismo

Che cosa è il polimorfismo in Java?

Il termine polimorfismo, dal greco πολυμορφοσ (polymorfos, "avere molte forme"), nell'ambito dei linguaggi di programmazione si riferisce in generale alla possibilità data ad una determinata espressione di assumere valori diversi in relazione ai tipi di dato a cui viene applicata.

Introduzione OPP (programmazione ad oggetti):

Programmazione ad oggetti o (OOP= Object Oriented Programing) è una programmazione che si prefissa di suddividere un programma secondo diversi tipi di oggetti.
Creiamo delle entità che esistono nella vita reale e le andiamo a trattare come degli oggetti in programmazione---> Esempio: sulla nostra scrivania possiamo avere:
un mouse, schermi, tastiera, microfono ecc. ognuno di questo è un oggetto e ogni oggetto, ha delle caratteristiche specifiche e delle azioni che può eseguire.
Ogni cosa che esiste nel nostro mondo reale, comprese persone e animali possiamo definirle oggetti in programmazione.
Altro esempio: io oggetto (programmazione) persona fisica e avrò degli attributi:
nome, cognome, età, genere, altezza, peso, capelli ecc.

É potrò compiere anche delle azioni, gestite attraverso i metodi e funzioni da usare come oggetto persona :
mangiare, giocare, guidare, dormire, respirare, pensare ecc.

La sostanza della programmazione ad oggetti è:
prendere cose che esistono nella vita reale e trasformarle in oggetti nella programmazione, questo ci aiuterà a gestire meglio determinate cose, azioni e situazioni

Classi Java:

- Una classe è uno dei concetti fondamentali di OOP.
- Una classe è un modello o un progetto per la creazione di oggetti.
- Una classe non consuma memoria.
- Una classe può essere istanziata più volte.
- Una classe fa una, e solo una, cosa.

Una classe è uno dei concetti fondamentali di OOP. Una classe è un insieme di istruzioni necessarie per costruire un tipo specifico di oggetto. Possiamo pensare a una classe come a un modello, un progetto o una ricetta che ci dice come creare oggetti di quella classe.

La creazione di un oggetto di quella classe è un processo chiamato istanziazione e di solito viene eseguito tramite la parola chiave new.

Possiamo istanziare tutti gli oggetti che vogliamo. Una definizione di classe non consuma memoria salvata come file sul disco rigido. Una delle migliori pratiche che una classe dovrebbe seguire è il principio di responsabilità singola (SRP): una classe dovrebbe essere progettata e scritta per fare una, e solo una, cosa.

Java è un linguaggio orientato agli oggetti:

- Come definire classi e oggetti in Java?
- Classe: codice che definisce un tipo concreto di oggetto, con proprietà e comportamenti in un unico file

- Oggetto: istanza, esemplare della classe, entità che dispone di alcune proprietà e comportamenti propri, come gli oggetti della realtà
- In Java quasi tutto è un oggetto, ci sono solo due eccezioni:
    - i tipi di dato semplici (tipi primitivi) e
    - gli array (un oggetto trattato in modo particolare)
- Le classi, in quanto tipi di dato strutturati, prevedono usi e regole più complessi rispetto ai tipi semplici

Le classi definiscono:

- I dati (detti campi o attributi)
- Le azioni (metodi, comportamenti) che agiscono sui dati

Possono essere definite:

- Dal programmatore (p.es. Automobile, Topo, Studente, ...)
- Dall'ambiente Java (p.es. String, System, Scanner, ...)

La "gestione" di una classe avviene mediante:

- Definizione della classe
- Instanziazione di Oggetti della classe

Struttura di una classe

```
package model;

public class Persona {

    //proprietà private - vedi incapsulamento
    private String nome;
    private String cognome;
    private Integer eta;

    //metodo costruttore
    public Persona (String nome, String cognome, Integer eta)  {
    this.nome = nome;
    this.cognome = cognome;
    this.eta = eta;
    }

    //per gestire le proprietà vedi metodi getters and setters
    //metodi...

    @Override
    public String toString () {
    return this.nome + " " + this.cognome + " " +  this.eta;
    }
}
```

Le classi in Java:

- Il primo passo per definire una classe in Java è creare un file che deve chiamarsi esattamente come la classe e con estensione .java
- Java permette di definire solo una classe per ogni file
- Una classe in Java è formata da:
- Attributi: (o campi/proprietà) che immagazzinano alcune informazioni sull'oggetto. Definiscono lo stato dell'oggetto
- Costruttore: metodo che si utilizza per inizializzare un oggetto
- Metodi: sono utilizzati per modificare o consultare lo stato di un oggetto. Sono equivalenti alle funzioni o procedure di altri linguaggi di programmazione

Classi e documentazione:

- Java è dotato di una libreria di classi "pronte all'uso" che coprono molte esigenze
- Usare classi già definite da altri è la norma (principio DRY)
- La libreria Java standard è accompagnata da documentazione che illustra lo scopo e l'utilizzo di ciascuna classe presente
- Dalla versione 9 di Java la libreria è stata divisa in moduli

Costruttore e il this:

Costruttore:

è quel metodo di una classe il cui compito è proprio quello di creare nuove istanze, oltre ad essere il punto del programma in cui un nuovo elemento (quindi una nuova identità) viene creato ed è reso disponibile per l'interazione con il resto del sistema.

Il costruttore riassunto in modo metaforico è creare una pagina bianca con degli attributi all'interno ad esempio di una classe chiamata Persona, ma con la possibilità di poter inserire noi i parametri di questi attributi ad esempio creando più persone.
Come si intuisce a differenza del metodo richiamato senza avere il costruttore alle spalle, dove sì, possiamo creare molteplici persone, ma con dei parametri fissi e preimpostati per ogni nuova persona creata, può essere utile ma non sempre.
Potremmo dover o voler inserire dei parametri diversi dei: nome, cognome ed età ad ogni creazione di una persona nuova e questo lo faremo per mezzo del costruttore.

This:

This usata all'interno di un metodo è un riferimento all'oggetto che ha invocato il metodo.

Nel linguaggio Java si usa il riferimento this. davanti a un oggetto o una variabile per evitare problemi di ambiguità tra attributi e metodi o costruttori che hanno lo stesso identificativo.

Scoop delle variabili:

lo possiamo concepire come il raggio d'azione in cui la variabile è disponibile nel codice, cioè:

In un programma possiamo trovare delle variabili, al di fuori di un metodo (come gli attributi o campi) oppure quelle all'interno di un metodo, questa differenza di creazione delle variabili fa sì che il loro scope sia diverso.

Scope Globale e Locale:

Scope= raggio d'azione in cui le variabili possono essere chiamate e lette, si dividono in:

Globale= variabile fuori dal metodo ma all'interno della classe, viene definita locale perché è richiamabile in tutta la classe quindi in qualsiasi metodo.

Locale= all'interno di un metodo, viene definita locale perché esiste solo in quel metodo, non abbiamo infatti accesso al di fuori del metodo.

Si può ovviare a questo problema creando un altro metodo passando il secondo metodo nel primo per poi richiamare la variabile del primo nel secondo cambiando il valore.

Overload del costruttore:

Possiamo dichiarare all'interno di una classe molti tipi di costruttori sovraccarichi, aventi stesso nome, ma che accettano parametri diversi e che quindi possono aiutarci ad inizializzare un oggetto a seconda delle nostre esigenze. Infatti i costruttori sovraccarichi risultano una validissima scelta nella creazione di apposite librerie che permettono la creazione di un oggetto nei più svariati modi.

Attraverso l'Overload si possono creare costruttori ai quali poi passare parametri limitati o differenti.
Naturalmente non possono creare due costruttori identici, genererebbe un errore!
In sostanza gli overloaded constructors = un costruttore con lo stesso nome ripetuto più volte nel quale poter mettere solo quello di cui abbiamo bisogno.

Metodo toString:

Se facessimo una stampa di oggetto quello che ci andrebbe a video sarebbe all'incirca questo:

Persona@6d03e736 = che è la collocazione a livello di memoria, come fosse l'indirizzo in cui è salvato il nostro oggetto.

Se andassimo ad aggiungere il metodo toString che tutti gli oggetti ereditano di default, non cambierebbe nulla ancora.
Per far sì che a video passino i dati corretti che noi vorremmo effettivamente vedere:

occorre andare nella classe desiderata e creare il metodo toString che passi i dati in modo corretto.

In sostanza il metodo toString codifica in modo leggibile il contenuto dell'oggetto.

In Java, il metodo 'toStringObject e viene ereditato da tutte le classi Java. Il metodo 'toStringtoString() viene utilizzato per ottenere una rappresentazione in formato stringa dell'oggetto.

Per utilizzare il metodo 'toString:

```
public class Persona {
      private String nome;
      private int età;
      public Persona(String nome, int età) {
      this.nome = nome;
      this.età = età;
   }
      @Override
      public String toString() {
      return "Nome: " + nome + ", Età: " + età;
    }
  }
  public class EsempioToString {
     public static void main(String[] args) {
        Persona p = new Persona("Mario Rossi", 30);
        String personaStringa = p.toString();
```

```
        System.out.println(personaStringa);

    }

}
```

In questo esempio, abbiamo creato una classe 'Personanome ed 'etàetà e il metodo 'toStringtoString() che restituisce una stringa contenente i valori dei campi 'nomenome ed età.

Nel metodo 'main della classe EsempioToString, abbiamo creato un oggetto 'PersonaPersona e abbiamo chiamato il metodo toString() su quell'oggetto per ottenere una rappresentazione in formato stringa dell'oggetto.

Infine, abbiamo stampato la rappresentazione in formato stringa a console utilizzando il metodo 'printlnprintln().

Prodotto:

```
Nome: Mario Rossi, Età: 30
```

Come puoi vedere, la chiamata al metodo 'toString sull'oggetto 'Persona restituisce una stringa con il nome e l'età della persona, come definito nel metodo 'toStringtoString() della classe 'PersonaPersona.

- # Eccezioni Java :

In Java ci sono tre tipi di eccezioni:

Eccezioni verificate (checked exceptions): sono le eccezioni che devono essere dichiarate o gestite all'interno del codice, altrimenti il codice non compila. Sono tipicamente associate a situazioni che potrebbero verificarsi durante l'esecuzione del programma, ma che possono essere gestite senza che l'applicazione termini in modo anomalo.

Esempi di eccezioni verificate includono: IOException, SQLException, ClassNotFoundException, InterruptedException.

Eccezioni non verificate (unchecked exceptions): sono le eccezioni che non richiedono la dichiarazione o la gestione all'interno del codice. Si verificano in genere in situazioni impreviste e difficilmente recuperabili, come ad esempio la divisione per zero, la dereferenziazione di un puntatore nullo, l'accesso ad un indice non valido di un array.

Esempi di eccezioni non verificate includono: NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException.

Errori (errors): sono eccezioni gravi che indicano un errore irrecuperabile che può causare la terminazione dell'applicazione.

Ad esempio, un errore OutOfMemoryError indica che l'applicazione ha esaurito la memoria disponibile per l'esecuzione. Gli errori non devono essere gestiti o dichiarati all'interno del codice.

Eccezioni verificate sono quelle che estendono la classe Exception in Java. Qui di seguito troverai alcuni esempi di eccezioni verificate:

1. **IOException**: questa eccezione viene sollevata quando si verificano problemi di input/output, ad esempio quando un file non può essere aperto o chiuso correttamente.

```
import java.io.*;

 public class IOExceptionExample {

    public static void main(String[] args) {

        try {

          FileReader file = new FileReader("file.txt");

          file.close();

        } catch (IOException e) {

        System.out.println("IOException caught: " + e.getMessage());

    }

   }

}
```

L'eccezione IOException è una delle eccezioni verificate in Java e viene lanciata quando si verifica un errore di input/output durante la lettura o la scrittura di un file. Ad esempio, se si tenta di aprire un file che non esiste o di scrivere su un file che non ha i permessi di scrittura, verrà lanciata l'eccezione IOException.

Nell'esempio di codice, abbiamo definito una classe IOExceptionExample con un metodo main. All'interno del metodo main, abbiamo utilizzato un blocco try-catch per gestire eventuali eccezioni IOException che potrebbero essere lanciate dalla lettura del file.

Nello specifico, abbiamo creato un oggetto FileReader per leggere il contenuto di un file chiamato "file.txt". Se la lettura del file ha successo, chiudiamo il file con il metodo close(). Tuttavia, se si verifica un'eccezione IOException durante la lettura del file, verrà catturata dal blocco catch e verrà stampato un messaggio di errore che indica la descrizione dell'errore (e.getMessage()).

In generale, quando si lavora con le eccezioni verificate, è necessario gestirle adeguatamente all'interno del codice, ad esempio con l'uso di un blocco try-catch come fatto in questo esempio. Ciò garantisce che il codice sia robusto e in grado di gestire eventuali errori che potrebbero verificarsi durante l'esecuzione del programma.

2. **SQLException**: questa eccezione viene sollevata quando si verifica un problema con un'operazione SQL, ad esempio quando una query fallisce.

```
import java.sql.*;

  public class SQLExceptionExample {

    public static void main(String[] args) {

      Connection con = null;

      Statement stmt = null;

      try {

        Class.forName("com.mysql.jdbc.Driver");

        con = DriverManager.getConnection(

        "jdbc:mysql://localhost:3306/mydatabase","root","");

        stmt = con.createStatement();

        String sql = "SELECT * FROM mytable";

        ResultSet rs = stmt.executeQuery(sql);

        while (rs.next()) {

       // do something with the results

      }

      rs.close();

    } catch (SQLException e) {

      System.out.println("SQLException caught: " + e.getMessage());

    } catch (ClassNotFoundException e) {

      System.out.println("ClassNotFoundException caught: " +

      e.getMessage());
```

```
    } finally {
      try {
        if (stmt != null) stmt.close();
      } catch (SQLException e) {
        System.out.println("SQLException caught: " + e.getMessage());
    }
    try {
      if (con != null) con.close();
    } catch (SQLException e) {
      System.out.println("SQLException caught: " + e.getMessage());
   }
  }
 }
}
```

L'eccezione SQLException è un tipo di eccezione verificata in Java che viene lanciata quando si verifica un errore durante l'esecuzione di una query sul database SQL. Ad esempio, se si cerca di eseguire una query su una tabella che non esiste o si tenta di eseguire una query non valida, verrà lanciata l'eccezione SQLException.

Innanzitutto, il codice utilizza la libreria JDBC (Java Database Connectivity) per accedere ad un database MySQL.

Nel blocco try viene eseguita una serie di operazioni:

1) **La riga Class.forName**("com.mysql.jdbc.Driver"); carica la classe del driver JDBC necessario per la connessione al database MySQL.

2) **La riga con** = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydatabase","root",""); si connette al database specificato nell'URL "jdbc:mysql://localhost:3306/mydatabase" (in questo caso, il database si chiama mydatabase e si trova sulla stessa macchina del programma Java) utilizzando il nome utente root e la password vuota.

3) **La riga stmt** = con.createStatement(); crea un'istanza di Statement, che viene utilizzata per eseguire le query SQL sul database.

4) **La riga String sql** = "SELECT * FROM mytable"; definisce una query SQL di selezione che recupera tutte le righe dalla tabella mytable.

5) **La riga ResultSet rs** = stmt.executeQuery(sql); esegue la query SQL e restituisce un oggetto ResultSet contenente i risultati della query.

6) **Nel blocco while** (rs.next()), il programma cicla attraverso tutte le righe restituite dalla query e fa qualcosa con i dati (in questo caso, il commento // do something with the results indica che qui si dovrebbe inserire il codice per elaborare i dati).

7) **Infine**, la riga rs.close(); chiude il ResultSet.

Se si verifica un'eccezione di tipo SQLException, il programma gestisce l'eccezione nel blocco catch (SQLException e) e stampa il messaggio di errore associato all'eccezione utilizzando il metodo getMessage().

Se si verifica un'eccezione di tipo ClassNotFoundException, il programma gestisce l'eccezione nel blocco catch (ClassNotFoundException e) e stampa il messaggio di errore associato all'eccezione utilizzando il metodo getMessage().

In ogni caso, dopo l'esecuzione del blocco try, il programma esegue il blocco finally, che garantisce che le risorse aperte (ovvero le connessioni al database e gli oggetti Statement) vengano sempre chiuse correttamente, anche in caso di errore durante l'esecuzione. In particolare, nel blocco finally il programma chiude prima lo statement (stmt) e poi la connessione (con), utilizzando il metodo close() di ciascun oggetto. Anche in questo caso, se si verifica un'eccezione durante la chiusura, il programma gestisce l'eccezione nel blocco catch (SQLException e) e stampa il messaggio di errore associato all'eccezione utilizzando il metodo getMessage().

3. **FileNotFoundException**: questa eccezione viene sollevata quando si cerca di accedere a un file che non esiste.

```
import java.io.*;

   public class FileNotFoundExceptionExample {

     public static void main(String[] args) {

      try {

      FileReader file = new FileReader("missingfile.txt");
```

```
      file.close();
    } catch (FileNotFoundException e) {
      System.out.println("FileNotFoundException caught: " +
      e.getMessage());
    } catch (IOException e) {
      System.out.println("IOException caught: " + e.getMessage());
  }
 }
}
```

Il codice inizia importando il package java.io, che contiene le classi e le interfacce per la gestione dei flussi di input/output, tra cui la classe FileReader.

La classe FileNotFoundExceptionExample contiene un metodo main che rappresenta il punto di ingresso del programma.

All'interno del blocco try, viene creato un nuovo oggetto FileReader che tenta di aprire il file missingfile.txt. Tuttavia, poiché il file non esiste, viene sollevata un'eccezione di tipo FileNotFoundException. Questa eccezione viene catturata dal blocco catch associato e il messaggio di errore viene stampato sulla console.

Il secondo blocco catch gestisce eventuali eccezioni di tipo IOException che potrebbero essere sollevate da FileReader durante la lettura del file. Poiché l'eccezione sollevata in questo caso è di tipo FileNotFoundException, il blocco catch associato a IOException viene ignorato.

Infine, il programma termina.

L'utilizzo di un blocco try-catch permette di gestire le eccezioni che potrebbero verificarsi durante l'esecuzione del codice e di prevenire l'arresto anomalo del programma. In questo caso, la gestione dell'eccezione consente di avvisare l'utente del problema e di garantire che il programma termini in modo controllato.

4. **ClassNotFoundException**: questa eccezione viene sollevata quando non si trova una classe a cui si sta cercando di accedere.

```
public class ClassNotFoundExceptionExample {
```

```
  public static void main(String[] args) {
     try {
     Class.forName("com.mysql.jdbc.Driver");
   } catch (ClassNotFoundException e) {
     System.out.println("ClassNotFoundException caught: " +
     e.getMessage());
  }
 }
}
```

Questa classe Java mostra come gestire un'eccezione di tipo ClassNotFoundException.

La classe cerca di caricare la classe "com.mysql.jdbc.Driver" utilizzando il metodo statico forName della classe Class, che restituisce l'oggetto Class corrispondente al nome della classe specificato come parametro. Tuttavia, se la classe richiesta non è presente nel classpath, viene sollevata un'eccezione di tipo ClassNotFoundException.

Il codice all'interno del blocco try-catch cattura l'eccezione e stampa il messaggio di errore associato all'eccezione utilizzando il metodo getMessage().

In questo caso, se la classe "com.mysql.jdbc.Driver" non è presente nel classpath, verrà sollevata un'eccezione di tipo ClassNotFoundException e il messaggio di errore verrà stampato sulla console. **Altrimenti**, il programma proseguirà senza alcun errore.

5. **InterruptedException**: questa eccezione viene sollevata quando un thread viene interrotto mentre è in attesa o in esecuzione.

```
public class InterruptedExceptionExample {
  public static void main(String[] args) {
    Thread t = new Thread(new Runnable() {
      public void run() {
        try {
          Thread.sleep(1000);
```

```
        } catch (InterruptedException e) {
          System.out.println("InterruptedException caught: " +
          e.getMessage());
        }
      }
    });
    t.start();
    t.interrupt();
  }
}
```

Questo è un esempio di come gestire l'eccezione InterruptedException. L'eccezione viene generata quando un thread in pausa viene interrotto da un altro thread.

Nel codice, viene creato un nuovo thread t che esegue un metodo run(). All'interno del metodo run(), il thread viene messo in pausa per 1000 millisecondi (ovvero 1 secondo) utilizzando il metodo Thread.sleep(1000).

Tuttavia, il thread principale interrompe il thread t utilizzando il metodo t.interrupt(). Questo genera un'eccezione InterruptedException che viene gestita all'interno del blocco catch.

All'interno del blocco catch, viene stampato il messaggio "InterruptedException caught: " seguito dal messaggio di errore specifico restituito dall'eccezione. In questo caso, il messaggio di errore sarà "sleep interrupted" perché il thread è stato interrotto mentre era in pausa.

In generale, l'eccezione InterruptedException viene sollevata quando un thread viene interrotto mentre è in attesa di un blocco synchronized, di un monitor o in attesa di un'operazione I/O. In questo caso, la gestione dell'eccezione prevede la possibilità di terminare correttamente il thread in modo che non causi problemi al sistema.

Eccezioni non Verificate:

Eccezioni non verificate in Java con dei commenti nel codice per spiegare il loro significato:

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

```
    int[] arr = {1, 2, 3};
    System.out.println(arr[3]); // ArrayIndexOutOfBoundsException: 3
    int x = 5 / 0; // ArithmeticException: divisione per zero
    String str = null;
    System.out.println(str.length()); // NullPointerException:
    dereferenziazione di un puntatore nullo
  }
}
```

In questo esempio, abbiamo creato una classe UncheckedExceptionsExample con un metodo main. All'interno del metodo main, abbiamo definito un array di interi arr con tre elementi e poi abbiamo cercato di accedere al quarto elemento (l'indice 3). Questo causerà l'eccezione ArrayIndexOutOfBoundsException, poiché l'array ha solo tre elementi e quindi non esiste un quarto elemento.

Inoltre, abbiamo diviso il numero intero 5 per 0, il che è impossibile e causerà l'eccezione ArithmeticException poiché la divisione per zero è un'operazione non valida.

Infine, abbiamo definito una stringa str come null e abbiamo cercato di accedere alla sua lunghezza. Ciò causerà l'eccezione NullPointerException, poiché la stringa è null e non ha una lunghezza valida.

In tutti e tre i casi, queste eccezioni non verificate si verificano in situazioni impreviste e difficilmente recuperabili, come abbiamo già detto in precedenza. Pertanto, non sono necessarie dichiarazioni o gestione esplicita nel codice.

Error:

Esempi di errori in Java con dei commenti nel codice per spiegare il loro significato:

```
public class ErrorsExample {
  public static void main(String[] args) {
    int[] arr = new int[Integer.MAX_VALUE]; // OutOfMemoryError: spazio
    heap
    Thread.currentThread().stop(); // ThreadDeath
```

```
  }

}
```

In questo esempio, abbiamo creato una classe ErrorsExample con un metodo main. All'interno del metodo main, abbiamo definito un array di interi arr con una dimensione enorme, pari al valore massimo dell'intero in Java (Integer.MAX_VALUE). Questo causerà l'errore OutOfMemoryError, poiché l'array richiederebbe una quantità di memoria superiore a quella disponibile.

Inoltre, abbiamo chiamato il metodo stop sul thread corrente. Questo causerà l'errore ThreadDeath, poiché stop è un metodo deprecato e non dovrebbe essere utilizzato. Inoltre, chiamare questo metodo causerà la terminazione del thread corrente in modo non corretto.

In entrambi i casi, questi errori indicano situazioni gravi e irrecuperabili che potrebbero causare la terminazione dell'applicazione. Pertanto, non devono essere gestiti o dichiarati all'interno del codice, ma dovrebbero essere evitati attraverso la progettazione appropriata dell'applicazione.

```
package dal;

import java.sql.SQLException;

import java.util.List;

import model.Serie;

public interface SerieDao { //sono tutte final

String TABELLA ="serietv";

String ADD ="insert into" + TABELLA + " () VALUES ()";

String GET_ONE ="select * from " + TABELLA + " where id =?";

String GET_ALL ="select * from " + TABELLA;

String UPDATE ="undate " + TABELLA + " set ";

String DELETE ="delete from " + TABELLA + " where id =?\";";
```

```
void addSerie(Serie s) throws SQLException;

void updateSerie(Serie s) throws SQLException;

void deleteSerie(Serie s) throws SQLException;

void deleteSerieBYId(int id) throws SQLException;

Serie getSerieByID (int id) throws SQLException;

List <Serie> getSerie() throws SQLException;

}
```