La programmazione orientata agli oggetti (OOP) è un paradigma di programmazione che si basa sul concetto di "oggetti", che possono contenere dati sotto forma di campi, spesso noti come attributi o proprietà, e codice sotto forma di procedure, spesso note come metodi. Gli oggetti sono istanze di classi, che possono essere pensate come dei modelli per la creazione di oggetti. I concetti fondamentali della OOP includono:

### **1. Classi e Oggetti**

* **Classe**: Una classe è una struttura che definisce proprietà (attributi) e comportamenti (metodi) che gli oggetti creati da quella classe avranno. Può essere vista come una blueprint o un template.
  + **Esempio**: class Cane, che può avere attributi come razza, colore, età e metodi come abbaia(), corre().
* **Oggetto**: Un'istanza di una classe. È un'entità autonoma con uno stato e un comportamento definiti dalla classe da cui è stato creato.
  + **Esempio**: Fido potrebbe essere un oggetto della classe Cane.

### **2. Incapsulamento**

* **Incapsulamento**: Il processo di nascondere i dettagli interni di un oggetto e di fornire un'interfaccia pubblica attraverso cui interagire con l'oggetto. Questo aiuta a proteggere l'integrità dei dati e impedisce modifiche indesiderate.
  + **Esempio**: Attributi privati in una classe accessibili solo attraverso metodi pubblici (getter e setter).

### **3. Ereditarietà**

* **Ereditarietà**: Un meccanismo che permette di creare una nuova classe (classe derivata o sottoclasse) basata su una classe esistente (classe base o superclasse). La classe derivata eredita attributi e metodi dalla classe base, ma può anche aggiungere o modificare funzionalità.
  + **Esempio**: class CaneDomestico che eredita da class Cane ma aggiunge un metodo faFeste().

### **4. Polimorfismo**

* **Polimorfismo**: La capacità di usare un'interfaccia comune per oggetti di diverse classi. Permette di trattare oggetti di classi diverse in modo uniforme se condividono la stessa interfaccia o classe base.
  + **Esempio**: Se Cane e Gatto ereditano entrambi da una classe base Animale, un metodo che accetta un parametro di tipo Animale può lavorare sia con oggetti Cane sia con oggetti Gatto.

### **5. Astrazione**

* **Astrazione**: Il concetto di nascondere i dettagli complessi di un sistema e mostrare solo le funzionalità essenziali. Permette di concentrarsi su ciò che un oggetto fa piuttosto che su come lo fa.
  + **Esempio**: Una classe Veicolo che ha un metodo astratto muovi(). Le sottoclassi come Auto e Bicicletta forniscono implementazioni specifiche di muovi().

Le classi astratte e le interfacce sono concetti fondamentali nella programmazione orientata agli oggetti in Java. Entrambe consentono di definire contratti che altre classi devono seguire, ma hanno scopi e caratteristiche leggermente diversi.

### **Classi Astratte in Java**

Una classe astratta è una classe che non può essere istanziata direttamente, ma può contenere metodi concreti (cioè con implementazione) e/o metodi astratti (senza implementazione). Gli oggetti vengono creati dalle sue sottoclassi, che implementano i metodi astratti definiti nella classe astratta.

* **Quando usarle**:
  + **Quando si vuole fornire una base comune**: Se diverse classi condividono una parte significativa dell'implementazione, è utile creare una classe astratta che contenga questa implementazione comune.
  + **Per definire un contratto**: Quando si desidera definire un insieme di metodi (alcuni con implementazione predefinita e alcuni astratti) che le classi derivate devono implementare.

Esempio di classe astratta in Java:

java

Copia codice

abstract class Figura {

// Metodo concreto

public void descrizione() {

System.out.println("Sono una figura.");

}

// Metodo astratto

public abstract double area();

}

class Rettangolo extends Figura {

private double base;

private double altezza;

public Rettangolo(double base, double altezza) {

this.base = base;

this.altezza = altezza;

}

@Override

public double area() {

return base \* altezza;

}

}

### **Interfacce in Java**

Un'interfaccia in Java è un contratto che definisce un insieme di metodi (e costanti) che una classe deve implementare. A differenza delle classi astratte, le interfacce non possono contenere implementazioni dei metodi; solo le loro firme (senza corpo del metodo) sono definite.

* **Quando usarle**:
  + **Per definire un contratto più rigoroso**: Le interfacce permettono di definire un insieme di metodi che devono essere implementati da diverse classi, indipendentemente dalla gerarchia delle classi.
  + **Per supportare il polimorfismo**: Poiché una classe può implementare più interfacce, l'utilizzo delle interfacce favorisce la flessibilità e il riutilizzo del codice.

Esempio di interfaccia in Java:

java

Copia codice

interface Volante {

void vola();

}

class Uccello implements Volante {

@Override

public void vola() {

System.out.println("L'uccello sta volando.");

}

}

### **Quando conviene usarle**

* **Classi Astratte**:
  + Quando si desidera fornire una base comune per un gruppo di classi.
  + Quando si vuole definire una parte dell'implementazione che può essere condivisa tra diverse sottoclassi.
  + Quando si desidera evitare la duplicazione del codice.
* **Interfacce**:
  + Quando si desidera definire un contratto che più classi possono implementare in modo indipendente.
  + Quando si vuole supportare il polimorfismo senza vincolarsi alla gerarchia delle classi.
  + Quando si vogliono realizzare design che favoriscono la composizione rispetto all'ereditarietà.

In generale, è comune combinare l'uso di classi astratte e interfacce nei progetti Java per ottenere un equilibrio tra ereditarietà e composizione. Le interfacce spesso definiscono comportamenti, mentre le classi astratte forniscono implementazioni di base.

Le classi wrapper in Java sono delle classi che rappresentano i tipi primitivi come oggetti. Questo permette di trattare i tipi primitivi (come int, char, boolean, etc.) come se fossero oggetti. Le classi wrapper si trovano nel pacchetto java.lang e ognuna corrisponde a un tipo primitivo:

* Byte per byte
* Short per short
* Integer per int
* Long per long
* Float per float
* Double per double
* Character per char
* Boolean per boolean

## **Funzionalità Principali**

**Conversione tra tipi primitivi e oggetti**: Le classi wrapper permettono di convertire un valore primitivo in un oggetto e viceversa. Questo è utile quando si lavora con collezioni come ArrayList, che possono contenere solo oggetti.  
java  
Copia codice  
int num = 5;

Integer wrappedNum = Integer.valueOf(num); // Conversione da int a Integer

int unwrappedNum = wrappedNum.intValue(); // Conversione da Integer a int

**Utilizzo nelle collezioni**: Poiché le collezioni Java come ArrayList possono contenere solo oggetti, le classi wrapper permettono di memorizzare tipi primitivi all'interno di queste strutture dati.  
java  
Copia codice  
ArrayList<Integer> list = new ArrayList<>();

list.add(10); // Autoboxing: il valore int viene automaticamente convertito in un Integer

int value = list.get(0); // Unboxing: il valore Integer viene automaticamente convertito in un int

**Metodi di utilità**: Le classi wrapper offrono metodi di utilità per operazioni comuni, come la conversione di stringhe in numeri e viceversa.  
java  
Copia codice  
String numberString = "123";

int number = Integer.parseInt(numberString); // Converte la stringa in un int

String stringValue = Integer.toString(number); // Converte l'int in una stringa

**Costanti e Valori Limite**: Le classi wrapper forniscono costanti e metodi per ottenere i valori limite dei tipi primitivi.  
java  
Copia codice  
int maxInt = Integer.MAX\_VALUE;

int minInt = Integer.MIN\_VALUE;

**Autoboxing e Unboxing**: Java supporta automaticamente la conversione tra tipi primitivi e le loro classi wrapper. Questo processo è chiamato autoboxing (conversione automatica da tipo primitivo a wrapper) e unboxing (conversione automatica da wrapper a tipo primitivo).  
java  
Copia codice  
Integer wrappedInt = 5; // Autoboxing

int primitiveInt = wrappedInt; // Unboxing

## **Utilità Principali**

* **Compatibilità con le API**: Molte API di Java richiedono oggetti e non tipi primitivi. Le classi wrapper consentono di utilizzare i tipi primitivi con queste API.
* **Collezioni e Generics**: Le collezioni e i generics in Java lavorano solo con oggetti, non con tipi primitivi. Le classi wrapper permettono di utilizzare tipi primitivi con collezioni come List, Set e Map.
* **Metodi di Utilità**: Le classi wrapper offrono numerosi metodi di utilità per conversioni e operazioni sui dati.
* **Valori Null**: A differenza dei tipi primitivi, gli oggetti wrapper possono essere null, il che può essere utile in certi scenari, ad esempio per rappresentare valori mancanti in una collezione.

In sintesi, le classi wrapper in Java sono fondamentali per integrare i tipi primitivi nelle strutture e nelle API che richiedono oggetti, offrendo al contempo funzionalità aggiuntive per la manipolazione e la conversione dei dati.

In Java, il valore di default delle variabili dipende dal tipo di dato. Di seguito sono elencati i valori di default per i vari tipi di variabili:

1. **Tipi primitivi**:
   * byte: 0
   * short: 0
   * int: 0
   * long: 0L
   * float: 0.0f
   * double: 0.0d
   * char: '\u0000' (null character)
   * boolean: false
2. **Oggetti e array**:
   * Oggetti (inclusi array e stringhe): null
   * Array di tipi primitivi: valori di default corrispondenti ai tipi primitivi
   * Array di oggetti: null per ogni elemento dell'array

### **Esempio Pratico**

Quando dichiari una variabile di istanza (non locale) senza inizializzarla, Java assegna automaticamente il valore di default. Ecco un esempio:

java

Copia codice

public class DefaultValues {

// Variabili di istanza

byte byteValue;

short shortValue;

int intValue;

long longValue;

float floatValue;

double doubleValue;

char charValue;

boolean booleanValue;

String stringValue;

public void printValues() {

System.out.println("byte: " + byteValue);

System.out.println("short: " + shortValue);

System.out.println("int: " + intValue);

System.out.println("long: " + longValue);

System.out.println("float: " + floatValue);

System.out.println("double: " + doubleValue);

System.out.println("char: [" + charValue + "]");

System.out.println("boolean: " + booleanValue);

System.out.println("String: " + stringValue);

}

public static void main(String[] args) {

DefaultValues defaultValues = new DefaultValues();

defaultValues.printValues();

}

}

### **Output:**

plaintext

Copia codice

byte: 0

short: 0

int: 0

long: 0

float: 0.0

double: 0.0

char: [ ]

boolean: false

String: null

### **Note Importanti**

* Le variabili locali (dichiarate all'interno di un metodo) **non** hanno un valore di default e devono essere inizializzate prima di essere utilizzate. Se tenti di usare una variabile locale non inizializzata, otterrai un errore di compilazione.
* Le variabili di istanza (dichiarate all'interno di una classe ma al di fuori di qualsiasi metodo) e le variabili di classe (dichiarate con il modificatore static) vengono inizializzate ai loro valori di default se non vengono esplicitamente inizializzate.

I valori di default sono utili per prevenire errori quando le variabili vengono utilizzate prima di essere inizializzate esplicitamente, ma è sempre una buona pratica inizializzare le variabili in modo esplicito.

In Java, le classi possono essere **istanziabili** o **non istanziabili** a seconda di come sono progettate e del loro scopo.

### **Classi Istanziabili**

Le classi istanziabili sono quelle che possono essere utilizzate per creare oggetti. In altre parole, puoi usare il costruttore di una classe istanziabile per creare un'istanza (un oggetto) di quella classe. Ecco un esempio semplice:

class Persona {

String nome;

int eta;

Persona(String nome, int eta) {

this.nome = nome;

this.eta = eta;

}

void saluta() {

System.out.println("Ciao, mi chiamo " + nome);

}

}

public class Main {

public static void main(String[] args) {

Persona p = new Persona("Mario", 30);

p.saluta();

}

}

In questo esempio, la classe Persona è istanziabile perché possiamo creare un oggetto di tipo Persona usando il suo costruttore.

### **Classi Non Istanziabili**

Le classi non istanziabili sono progettate in modo tale da non poter creare oggetti direttamente da esse. Questo è spesso ottenuto rendendo il costruttore privato. Le classi non istanziabili sono spesso usate per fornire metodi e variabili statiche, come utility class o costanti. Un esempio comune è la classe Math in Java:

public final class Math {

private Math() {

// Costruttore privato per impedire l'istanza della classe

}

public static int add(int a, int b) {

return a + b;

}

}

In questo esempio, la classe Math ha un costruttore privato, quindi non può essere istanziata. Puoi solo chiamare i suoi metodi statici:

public class Main {

public static void main(String[] args) {

int risultato = Math.add(5, 3);

System.out.println(risultato); // Output: 8

}

}

### 

### 

### 

### **Differenze Principali**

1. **Creazione di Oggetti**:
   * **Classi Istanziabili**: Puoi creare oggetti di queste classi usando il costruttore.
   * **Classi Non Istanziabili**: Non puoi creare oggetti direttamente da queste classi, spesso a causa di un costruttore privato.
2. **Uso dei Metodi**:
   * **Classi Istanziabili**: I metodi sono generalmente chiamati sugli oggetti creati.
   * **Classi Non Istanziabili**: I metodi sono spesso statici e sono chiamati usando il nome della classe.
3. **Scopo**:
   * **Classi Istanziabili**: Utilizzate per rappresentare oggetti con stato e comportamento.
   * **Classi Non Istanziabili**: Utilizzate per fornire metodi di utilità e costanti che non dipendono dallo stato di un oggetto.

In Java, le annotazioni @Override e @Overload (anche se questa non esiste come tale) servono a scopi diversi e hanno usi specifici. Inoltre, ci sono altre annotazioni comuni e utili. Vediamo cosa sono e il loro scopo:

### **@Override**

#### **Cosa è:**

@Override è un'annotazione che si usa per indicare che un metodo di una classe sta sovrascrivendo un metodo con lo stesso nome in una superclasse o interfaccia.

#### **Scopo:**

* Garantire che un metodo sta effettivamente sovrascrivendo un metodo nella superclasse. Se non esiste un metodo con la stessa firma nella superclasse, il compilatore genererà un errore.
* Migliorare la leggibilità del codice e facilitare la manutenzione.

#### **Esempio:**

java

Copia codice

class Animale {

void suono() {

System.out.println("L'animale fa un suono");

}

}

class Cane extends Animale {

@Override

void suono() {

System.out.println("Il cane abbaia");

}

}

### **@Overload**

#### **Cosa è:**

@Overload non è un'annotazione esistente in Java. Tuttavia, il concetto di **sovraccarico** (overloading) dei metodi esiste.

#### **Scopo:**

* Permettere a una classe di avere più metodi con lo stesso nome ma con diverse firme (differenti parametri). Il compilatore decide quale metodo chiamare basandosi sui parametri passati.

#### **Esempio:**

class Calcolatrice {

int add(int a, int b) {

return a + b;

}

double add(double a, double b) {

return a + b;

}

}

La programmazione imperativa è un paradigma di programmazione in cui si specifica una sequenza di istruzioni che il computer deve eseguire per raggiungere un certo risultato. Si concentra su come un programma opera, descrivendo i cambiamenti di stato del sistema attraverso istruzioni di assegnazione, iterazione e controllo di flusso.

Le tre principali categorie di programmazione imperativa sono:

1. **Programmazione Procedurale**:
   * Si basa sulla divisione del programma in procedure o funzioni.
   * Le procedure sono blocchi di codice riutilizzabili che eseguono un compito specifico.
   * Esempi: C, Pascal.
2. **Programmazione a Oggetti (OOP)**:
   * Organizza il codice in oggetti che contengono dati (attributi) e funzioni (metodi).
   * Promuove l'incapsulamento, l'ereditarietà e il polimorfismo.
   * Esempi: Java, C++, Python.
3. **Programmazione Parallela**:
   * Consente l'esecuzione simultanea di più istruzioni o processi.
   * Utilizza concetti come thread, processi e sincronizzazione per migliorare l'efficienza e le prestazioni.
   * Esempi: OpenMP, MPI.