



**SAPIENZA**  
UNIVERSITÀ DI ROMA

Università "Sapienza" di Roma

Facoltà di Informatica

**Corso:** Metodologie Di Programmazione

# **LA PROGRAMMAZIONE JAVA**

**Author:** Giovanni Cinieri

**Reviewed by:** Alessia Cassetta

Giugno 2024

# Indice

<b>1</b>	<b>Concetti Base</b>	<b>2</b>
1.1	L'Object Oriented Programming . . . . .	2
1.2	Tipi di Dati . . . . .	3
1.3	Gli Identificatori . . . . .	3
1.4	Le Classi . . . . .	4
1.5	I Metodi . . . . .	5
1.6	Gli Oggetti . . . . .	6
1.7	I Cicli . . . . .	7
<b>2</b>	<b>2.0 I modificatori di accesso</b>	<b>8</b>
2.1	Public, Private, Protected e Default . . . . .	8
2.2	Static e Final . . . . .	9
	2.2.1 Final . . . . .	9
	2.2.2 Static . . . . .	10
<b>3</b>	<b>Ereditarietà</b>	<b>12</b>

# 1 Concetti Base

## 1.1 L'Object Oriented Programming

L'Object Oriented Programming (OOP) è un paradigma di programmazione che si basa sul concetto di "oggetto". Un oggetto è un'istanza di una classe, che a sua volta è un modello astratto di un concetto. Le classi sono organizzate in gerarchie, in cui una classe può ereditare i metodi e gli attributi di un'altra classe. Questo permette di creare codice più modulare, riutilizzabile e facile da mantenere.

L'OOP si concentra sulla rappresentazione del mondo reale nel codice, dove gli oggetti sono entità che hanno attributi (dati) e metodi (comportamenti) correlati. Gli oggetti interagiscono tra loro comunicando attraverso metodi e scambiando dati tra loro.

I quattro principi fondamentali dell'OOP sono:

1. **Incapsulamento:** ossia la capacità di raggruppare dati e metodi correlati in una singola entità chiamata classe. Inoltre, permette di nascondere i dettagli di implementazione di un oggetto e mostrare solo le funzionalità pubbliche.
2. **Ereditarietà:** Creare nuove classi basate su classi esistenti. La classe derivata (o sottoclasse) eredita gli attributi e i metodi della classe genitore (o superclasse) e può estenderli o modificarli per adattarli alle proprie esigenze.
3. **Polimorfismo:** Consentire a un oggetto di comportarsi in modo diverso in base al contesto.
4. **Astrazione:** la capacità di astrarre i dettagli complessi di un oggetto e fornire un'interfaccia semplificata per utilizzarlo. L'astrazione consente di focalizzarsi sull'essenza dell'oggetto e nascondere la complessità dei dettagli di implementazione.

Java è un linguaggio di programmazione orientato agli oggetti che supporta pienamente l'OOP. Tutti i concetti fondamentali dell'OOP, come classi, oggetti, incapsulamento, ereditarietà, polimorfismo e astrazione, sono supportati in Java.

Vedremo ora alcuni dei concetti di base di Java che sono fondamentali per la programmazione orientata agli oggetti.

## 1.2 Tipi di Dati

### Tipi di dati

Sono le diverse categorie di valori che una variabile può memorizzare. I tipi di dati determinano la natura e la rappresentazione dei valori che possono essere assegnati a una variabile, nonché le operazioni che possono essere eseguite su di essi.

Ecco i tipi di dati fondamentali in Java:

- **Tipi primitivi:** rappresentano i tipi di dati di base e sono supportati direttamente dal linguaggio. I tipi primitivi includono `int`, `double`, `boolean`, `char`, ecc.
- **Tipi di riferimento:** rappresentano oggetti complessi e sono creati utilizzando le classi. I tipi di riferimento includono `String`, `ArrayList`, `Scanner`, ecc.

I tipi di dati primitivi sono passati per valore, mentre i tipi di dati di riferimento sono passati per riferimento. Ciò significa che quando si assegna un valore di tipo primitivo a una variabile, viene copiato il valore effettivo, mentre quando si assegna un valore di tipo di riferimento a una variabile, viene copiato il riferimento all'oggetto.

## 1.3 Gli Identificatori

### Identificatori

E' un nome utilizzato per identificare una variabile, una costante, una funzione, una classe o qualsiasi altra entità nel codice.

Ecco alcune regole per la definizione degli identificatori in Java:

1. Gli identificatori possono essere composti da lettere, numeri e il carattere di sottolineatura `_`. Devono iniziare con una lettera o il carattere di sottolineatura.
2. Gli identificatori sono sensibili al caso, quindi `myVariable` e `myvariable` sono considerati identificatori distinti.
3. Non è possibile utilizzare parole chiave riservate come identificatori. Ad esempio, non è possibile utilizzare `int`, `class`, `if`, ecc. come nomi di variabili o funzioni.
4. Gli identificatori possono avere qualsiasi lunghezza, ma è buona pratica utilizzare nomi significativi e descrittivi che aiutino a capire lo scopo dell'entità identificata.
5. Gli identificatori non possono contenere spazi o caratteri speciali come `!`, `@`, `#`.
6. È possibile utilizzare il camel case o l'underscore per separare le parole negli identificatori. Ad esempio, `nomeUtente`, `numero_telefono`, ecc.

Ecco alcuni esempi di identificatori validi:

```
1  int numero;  
2  String nomeCompleto;  
3  double tassoInteresse;  
4  final int LIMITE_MAX = 100;  
5  void calcolaSomma() {  
6      // corpo del metodo  
7  }
```

In questi esempi, `numero`, `nomeCompleto`, `tassoInteresse` sono identificatori di variabili, `LIMITE_MAX` è un identificatore di costante, e `calcolaSomma` è un identificatore di metodo.

## 1.4 Le Classi

### Le Classi

E' una struttura fondamentale per l'organizzazione e l'astrazione del codice. È un modello o un blueprint che definisce le caratteristiche e il comportamento di un oggetto.

Una classe rappresenta un concetto o un'entità del mondo reale e contiene dati (attributi) e comportamenti (metodi) correlati a quella specifica entità. Ad esempio, si potrebbe avere una classe "Persona" che ha attributi come nome, età e indirizzo, e metodi come "saluta" o "cammina".

Le classi in Java forniscono una struttura per creare oggetti, che sono le istanze specifiche di una classe. Ad esempio, utilizzando la classe "Persona", si possono creare oggetti come "persona1" o "persona2" che hanno i loro valori unici per i campi dati come nome ed età.

Le classi sono fondamentali nel paradigma di programmazione orientata agli oggetti (OOP) in Java. Consentono l'incapsulamento dei dati e del comportamento correlato, la modularità del codice e la possibilità di creare gerarchie di classi tramite l'ereditarietà.

Ecco un esempio di definizione di una classe in Java:

```
1  public class Persona {  
2      // attributi  
3      String nome;  
4      int eta;  
5  
6      // metodi  
7      void saluta() {  
8          System.out.println("Ciao, mi chiamo " + nome);  
9      }  
10 }
```

## 1.5 I Metodi

### Metodo

E' un blocco di codice che definisce un comportamento specifico e può essere richiamato (o chiamato) da altre parti del programma per eseguire determinate operazioni. I metodi consentono di organizzare il codice in unità più piccole e modulari, rendendo il programma più strutturato e facile da leggere, comprendere e mantenere.

Un metodo in Java è definito all'interno di una classe e ha una firma che specifica il suo nome, i suoi parametri di input (se presenti) e il suo tipo di ritorno (se produce un risultato). La sintassi generale per definire un metodo in Java è la seguente:

```
1  <modificatore_di_accesso> <tipo_di_ritorno> <nomeDelMetodo> (<  
    parametri_di_input>) {  
2      // istruzioni da eseguire  
3  }
```

Dove:

- **modificatore\_di\_accesso**: specifica il livello di accesso del metodo (**public**, **private**, **protected**, **package-private**).
- **tipo\_di\_ritorno**: specifica il tipo di dato restituito dal metodo. Se il metodo non restituisce alcun valore, il tipo di ritorno è **void**.
- **nomeDelMetodo**: specifica il nome del metodo.
- **parametri\_di\_input**: specifica i parametri di input del metodo, separati da virgole. Ogni parametro è costituito dal tipo di dato e dal nome del parametro.
- **corpo\_del\_metodo**: contiene le istruzioni che definiscono il comportamento del metodo.

Ecco un esempio di definizione di un metodo in Java:

```
1  public int somma(int a, int b) {  
2      int risultato = a + b;  
3      return risultato;  
4  }
```

In questo esempio, il metodo si chiama **somma**, accetta due parametri di tipo **int** chiamati **a** e **b**, esegue l'operazione di somma tra i due parametri e restituisce il risultato come valore di ritorno di tipo **int**. Il modificatore di accesso **public** indica che il metodo può essere richiamato da altre classi.

## 1.6 Gli Oggetti

### Oggetto

In Java, un **oggetto** è un'istanza (esecuzione specifica) di una classe. Un oggetto rappresenta un'entità con caratteristiche (attributi) e comportamenti (metodi) specifici. È una struttura dati che incapsula lo stato e il comportamento correlato.

Un oggetto viene creato a partire da una classe utilizzando la parola chiave **new**. La classe definisce la struttura e il comportamento dell'oggetto, mentre l'oggetto effettivo esiste in memoria durante l'esecuzione del programma.

Ad esempio, supponiamo di avere la seguente classe **Persona** che rappresenta una persona con attributi come nome e età:

```
1 public class Persona {
2     private String nome;
3     private int eta;
4     public Persona(String nome, int eta) {
5         this.nome = nome;
6         this.eta = eta;
7     }
8     public void saluta() {
9         System.out.println("Ciao, sono "+nome+" e ho "+eta);
10    }
11 }
```

Possiamo creare istanze di oggetti Persona come segue:

```
1 public class Test {
2     public static void main(String[] args) {
3         Persona persona1 = new Persona("Mario", 30);
4         Persona persona2 = new Persona("Anna", 25);
5         persona1.saluta(); // Output: Ciao, sono Mario e ho 30 anni.
6         persona2.saluta(); // Output: Ciao, sono Anna e ho 25 anni.
7     }
8 }
```

Nell'esempio sopra, **persona1** e **persona2** sono oggetti Persona che sono stati creati istanziando la classe Persona utilizzando il costruttore e specificando i valori dei parametri ( **nome** e **età** ). Ogni oggetto ha i suoi dati (nome e età) e può eseguire il metodo **saluta()** per stampare un messaggio di saluto personalizzato.

## 1.7 I Cicli

### Ciclo

Il termine ciclo si riferisce a una struttura di controllo che consente di eseguire ripetutamente un blocco di istruzioni fino a quando una determinata condizione è soddisfatta o fino a quando una certa operazione è completata.

Un ciclo è utilizzato per automatizzare l'esecuzione ripetuta di un blocco di codice, consentendo di scrivere in modo più efficiente e conciso il codice che deve essere eseguito più volte. Ciò riduce la duplicazione del codice e semplifica la gestione delle iterazioni.

In Java, ci sono quattro tipi di cicli principali:

1. Ciclo **for**: è comunemente utilizzato quando si conosce in anticipo il numero di iterazioni che Appunti Java9 devono essere eseguite. Ha una sintassi definita, composta da tre parti: l'inizializzazione, la condizione di continuazione e l'aggiornamento. Ecco la sua struttura generale:

```
1   for (inizializzazione; condizione; aggiornamento) {  
2       // corpo del ciclo  
3   }
```

2. Ciclo **while**: viene utilizzato quando la condizione di iterazione non è nota in anticipo e deve essere verificata all'inizio di ogni iterazione. Ha una sintassi semplice:

```
1   while (condizione) {  
2       // corpo del ciclo  
3   }
```

3. Ciclo **do-while**: è simile al ciclo **while**, ma la condizione viene verificata alla fine di ogni iterazione. Ciò significa che il blocco di istruzioni viene eseguito almeno una volta, anche se la condizione è falsa. La sua sintassi è la seguente:

```
1   do {  
2       // corpo del ciclo  
3   } while (condizione);
```

4. Ciclo **for-each**: viene utilizzato per iterare attraverso gli elementi di una collezione (come array, ArrayList, LinkedList, etc.) senza dover gestire gli indici manualmente. La sua sintassi è la seguente:

```
1   for (tipo elemento : collezione) {  
2       // corpo del ciclo  
3   }
```



## 2 2.0 I modificatori di accesso

### 2.1 Public, Private, Protected e Default

#### Modificatori di accesso

I modificatori di accesso in Java sono parole chiave che specificano l'accessibilità di classi, metodi, attributi e costruttori. Ci sono quattro tipi: public, private, protected e default (package-protected), ognuno con un diverso livello di accesso.

- **public**: indica che l'elemento è accessibile da qualsiasi parte del programma, incluso da altre classi e pacchetti.

```
1 public class MyClass {
2     public int publicVariable;
3     public void publicMethod() {
4         // Codice del metodo
5     }
6 }
```

- **private**: l'elemento è accessibile solo all'interno della stessa classe. Non è accessibile da altre classi o pacchetti.

```
1 public class MyClass {
2     public int publicVariable;
3     public void publicMethod() {
4         // Codice del metodo
5     }
6 }
```

- **protected**: l'elemento è accessibile all'interno della stessa classe, dalle sottoclassi (anche se si trovano in pacchetti diversi) e dagli altri membri del pacchetto.

```
1 public class MyClass {
2     protected int protectedVariable;
3     protected void protectedMethod() {
4         // Codice del metodo
5     }
6 }
```

- **default (package-protected)**: se non viene specificato alcun modificatore di accesso, si applica il livello di accesso di default. L'elemento è accessibile solo all'interno dello stesso pacchetto.

```
1 class MyClass {
2     int defaultVariable;
3     void defaultMethod() {
4         // Codice del metodo
5     }
6 }
```

## 2.2 Static e Final

### 2.2.1 Final

**final**: Quando viene applicato a una variabile, indica che il suo valore non può essere modificato una volta assegnato. Quando viene applicato a un metodo, indica che il metodo non può essere sovrascritto dalle sottoclassi. Quando viene applicato a una classe, indica che la classe non può essere estesa da altre classi.

**Esempio:**

```
1 public class MyClass {
2     final int finalVariable = 10; //variabile costante
3     final void finalMethod() {
4         // Tentativo di modificare la variabile costante
5         // myConstant = 20; // Errore di compilazione
6     }
7 }
```

**Esempio:**

```
1 public class ParentClass {
2     public final void finalMethod() {
3         // Implementazione del metodo
4     }
5 }
6
7 public class ChildClass extends ParentClass {
8     // Tentativo di sovrascrivere il metodo final
9     // public void finalMethod() {} // Errore di compilazione
10 }
```

**Esempio:**

```
1 public final class FinalClass {
2     // Implementazione della classe
3 }
4
5 // Tentativo di estendere una classe finale
6 // public class ChildClass extends FinalClass {} // Errore di
   compilazione
```

### 2.2.2 Static

**static:** quando viene applicato a una variabile, indica che la variabile appartiene alla classe anziché a un'istanza specifica della classe. Quando viene applicato a un metodo, indica che il metodo può essere chiamato senza creare un'istanza della classe.

**Esempio:**

```
1 public class MyClass {
2     static int staticVariable = 5; // Variabile statica
3     static void staticMethod() {
4         int myLocalVariable = myStaticVariable + 10; // Accesso
5     }
6 }
```

**Esempio:**

```
1 public class MyClass {
2     static void staticMethod() {
3         System.out.println("Blocco Statico Eseguito");
4     }
5     public static void main(String[] args) {
6         MyClass.staticMethod(); // Chiamata al metodo statico se
7     }
8 }
```

**Esempio:**

```
1 public class MyClass {
2     static int myStaticVariable;
3     static {
4         // Inizializzazione della variabile statica nel blocco s
5         myStaticVariable = 10;
6         System.out.println("Blocco statico eseguito");
7     }
8     public static void main(String[] args) {
9         System.out.println(MyClass.myStaticVariable); // Output
10    }
11 }
```

Principalmente static si usa in un metodo quando a quel metodo non serve accedere a nessun attributo della classe a cui appartiene. Si può utilizzare anche negli attributi, quando si vuole che quell'attributo venga condiviso tra tutte le istanze.

L'utilizzo di "final" e "static" può apportare restrizioni e comportamenti specifici alle variabili, ai metodi e alle classi, offrendo maggiore controllo e flessibilità nella progettazione e nell'utilizzo del codice.

È possibile utilizzare i modificatori "static" e "final" insieme. Quando un membro viene dichiarato come "static final", significa che è una costante condivisa tra tutte le istanze della classe e il suo valore non può essere modificato una volta assegnato.

### Esempio:

```
1 public class Cerchio {
2     static final double PI_GRECO = 3.14159; // Costante statica
3     private double raggio;
4     public Cerchio(double raggio) {
5         this.raggio = raggio;
6     }
7     public double calcolaArea() {
8         return PI_GRECO * raggio * raggio;
9     }
10    public static void main(String[] args) {
11        Cerchio cerchio1 = new Cerchio(5.0);
12        Cerchio cerchio2 = new Cerchio(3.0);
13        System.out.println("Area del cerchio1: " + cerchio1.
14        calcolaSomma())
15        System.out.println("Area del cerchio2: " + cerchio2.
16        calcolaArea())
17        System.out.println("Valore di PI_GRECO: " + Cerchio.PI_GRECO)
18    }
19 }
```

### 3 Ereditarietà