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

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:

```
    int numero;
    String nomeCompleto;
    double tassoInteresse;
    final int LIMITE_MAX = 100;
    void calcolaSomma() {
        // corpo del metodo
    }
```

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:

```
    public class Persona {
        // attributi
        String nome;
        int eta;

        // metodi
        void saluta() {
            System.out.println("Ciao, mi chiamo " + nome);
        }
    }
```

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:

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

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:

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

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:

```
public class Persona {
    private String nome;
    private int eta;
    public Persona(String nome, int eta) {
        this.nome = nome;
        this.eta = eta;
    }
    public void saluta() {
        System.out.println("Ciao, sono "+nome+" e ho "+eta);
    }
}
```

Possiamo creare istanze di oggetti Persona come segue:

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

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 Java9devono essere eseguite. Ha una sintassi definita, composta da tre parti: l'inizializzazione, la condizione di continuazione e l'aggiornamento. Ecco la sua struttura generale:

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

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:

```
    while (condizione) {
        // corpo del ciclo
    }
```

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:

```
    do {
        // corpo del ciclo
    } 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:

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

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.

```
    public class MyClass {
        public int publicVariable;
        public void publicMethod() {
            // Codice del metodo
        }
    }
```

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

```
    public class MyClass {
        public int publicVariable;
        public void publicMethod() {
            // Codice del metodo
        }
    }
```

- `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.

```
    public class MyClass {
        protected int protectedVariable;
        protected void protectedMethod() {
            // Codice del metodo
        }
    }
```

- `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.

```
    class MyClass {
        int defaultVariable;
        void defaultMethod() {
            // Codice del metodo
        }
    }
```

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:

```
    public class MyClass {
        final int finalVariable = 10; //variabile costante
        final void finalMethod() {
            // Tentativo di modificare la variabile costante
            // myConstant = 20; // Errore di compilazione
        }
    }
```

Esempio:

```
    public class ParentClass {
        public final void finalMethod() {
            // Implementazione del metodo
        }
    }

    public class ChildClass extends ParentClass {
        // Tentativo di sovrascrivere il metodo final
        // public void finalMethod() {} // Errore di compilazione
    }
```

Esempio:

```
    public final class FinalClass {
        // Implementazione della classe
    }
    // Tentativo di estendere una classe finale
    public class ChildClass extends FinalClass {} // Errore di
   compilazione
```

Esempio:

```
    public class MyClass {
        final int myFinalVariable;
        public MyClass() {
            // Inizializzazione della variabile finale nel costruttore
            myFinalVariable = 10;
        }
    }
```

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:

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

Esempio:

```
    public class MyClass {
        static void staticMethod() {
            System.out.println("Blocco Statico Eseguito");
        }
        public static void main(String[] args) {
            MyClass.staticMethod(); // Chiamata al metodo statico se
        }
    }
```

Esempio:

```
    public class MyClass {
        static int myStaticVariable;
        static {
            // Inizializzazione della variabile statica nel blocco s
            myStaticVariable = 10;
            System.out.println("Blocco statico eseguito");
        }
        public static void main(String[] args) {
            System.out.println(MyClass.myStaticVariable); // Output
        }
    }
```

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.

3 Ereditarietà

Ereditarietà

L'ereditarietà è un concetto fondamentale della programmazione orientata agli oggetti che consente di creare nuove classi (classi figlie o sottoclassi) basate su classi esistenti (classi genitore o superclassi). La classe figlia eredita tutti gli attributi e i metodi della classe genitore, consentendo di estendere e specializzare il comportamento della classe genitore.

Si dice che una classe A è sottoclasse di B quando:

- A eredita da B sia il suo stato che il suo comportamento. Quindi un'istanza della classe A è utilizzabile in ogni parte del codice in cui sia possibile utilizzare una istanza della classe B.
- Un elemento della classe A è anche un elemento della classe B.
- Gli elementi della classe A si comportano come elementi della classe B.

Ecco un esempio di ereditarietà in Java:

```
class Veicolo {
    protected String marca;
    protected String modello;
    public Veicolo(String marca, String modello) {
        this.marca = marca;
        this.modello = modello;
    }
    public void accelera() {
        System.out.println("Il veicolo accelera");
    }
    public void frena() {
        System.out.println("Il veicolo frena");
    }
}

class Auto extends Veicolo {
    private int numeroPorte;

    public Auto(String marca, String modello, int numeroPorte) {
        super(marca, modello);
        this.numeroPorte = numeroPorte;
    }
    public void apriPorte() {
        System.out.println("Apri le porte dell'auto");
    }
}

```

```
28      public class EreditarietaEsempio {
29          public static void main(String[] args) {
30              Auto auto = new Auto("Fiat", "Panda", 5);
31              auto.accelera(); // Eredita il metodo dalla classe Veicolo
32              auto.apriPorte(); // Metodo specifico della classe Auto
33          }
34      }
```

In questo esempio, la classe `Veicolo` è la classe genitore o superclasse, e la classe `Auto` è la classe figlia o sottoclasse. La classe `Auto` eredita gli attributi `marca` e `modello` , nonché i metodi `accellera()` e `frena()` dalla classe Veicolo . La classe `Auto` ha anche il suo metodo specifico `apriPorte()` . Possiamo creare un'istanza della classe `Auto` e chiamare i suoi metodi, compresi quelli ereditati dalla classe `Veicolo` .

L'ereditarietà consente di creare una gerarchia di classi, con classi più specializzate che ereditano le caratteristiche di classi più generiche. Questo promuove il riutilizzo del codice, l'estensibilità e la modularità del programma.

3.1 Super e This

Super e This

In Java, super e this sono parole chiave utilizzate per fare riferimento a classi e oggetti specifici durante la programmazione orientata agli oggetti

1. `super`: La parola chiave super viene utilizzata per fare riferimento alla classe genitore o superclasse di una classe. Può essere utilizzata in diversi contesti:
   - Per richiamare il costruttore della classe genitore all'interno del costruttore della classe figlio.
   - Per richiamare i metodi della classe genitore se sono stati sovrascritti nella classe figlio.
   - Per fare riferimento ai membri della classe genitore nello stesso modo in cui si farebbe con qualsiasi altro membro della classe figlio.
2. `this`: La parola chiave this viene utilizzata per fare riferimento all'oggetto corrente all'interno di una classe. Può essere utilizzata in diversi contesti:
   - Per fare riferimento alle variabili di istanza della classe corrente.
   - Per richiamare i costruttori della stessa classe (overload del costruttore).
   - Per passare l'istanza corrente come argomento a un metodo di un'altra classe.
   - Per restituire l'istanza corrente da un metodo (ad esempio, in un metodo di fabbrica).

L'utilizzo di `super` e `this` consente di distinguere tra variabili e metodi locali e quelli delle superclassi o delle istanze correnti. Ad esempio, se una classe ha un membro con lo stesso nome di una variabile locale o di un parametro di un metodo, `this` può essere utilizzato per fare riferimento al membro della classe, mentre `super` può essere utilizzato per fare riferimento al membro della superclasse.

Ecco un esempio di utilizzo di `super` e `this`:

```
class Veicolo {
    String marca;
    public Veicolo(String marca) {
        this.marca = marca;
    }
    Appunti Java18public void stampaMarca() {
        System.out.println("Marca: " + marca);
    }
}

class Auto extends Veicolo {
    int anno;
    public Auto(String marca, int anno) {
        super(marca);
        this.anno = anno;
    }
    @Override
    public void stampaMarca() {
        super.stampaMarca();
        System.out.println("Anno: " + anno);
    }
}

public class Test {
    public static void main(String[] args) {
        Auto auto = new Auto("BMW", 2021);
        auto.stampaMarca();
    }
}
```

In questo esempio, la classe `Veicolo` è la superclasse di `Auto` . All'interno del costruttore di `Auto` , `super(marca)` viene utilizzato per richiamare il costruttore della superclasse `Veicolo` e impostare il valore della variabile `marca` . Inoltre, all'interno del metodo `stampaMarca()` di `Auto` , `super.stampaMarca()` viene utilizzato per richiamare il metodo `stampaMarca()` della superclasse e successivamente viene stampato l'anno specifico dell'auto.

4 Polimorfismo

Polimorfismo

Il polimorfismo è un concetto chiave della programmazione orientata agli oggetti che consente a un oggetto di apparire in diversi modi o forme. In Java, il polimorfismo può essere realizzato attraverso l'ereditarietà e l'implementazione di metodi polimorfici.

In altre parole, il polimorfismo consente a un oggetto di assumere più forme. Si può utilizzare un oggetto di una classe specifica come se fosse un oggetto di una delle sue classi genitore o di una delle sue interfacce implementate. Questo offre una maggiore flessibilità e modularità nel progettare e scrivere il codice.

Il polimorfismo in Java può essere realizzato attraverso l'ereditarietà e l'implementazione di interfacce. Quando un oggetto viene trattato come un'istanza della classe genitore o dell'interfaccia, può accedere solo ai membri e ai metodi definiti nella classe genitore o nell'interfaccia. Tuttavia, l'implementazione specifica dei metodi viene determinata dinamicamente in base al tipo reale dell'oggetto.

4.1 Overloading