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

4.1 Overloading

Overloading

L'overloading, o **sovraccarico dei metodi**, è una caratteristica del linguaggio di programmazione Java che consente di definire più metodi con lo stesso nome ma con diversi parametri. Questi metodi possono avere lo stesso nome ma devono avere firme di parametro diverse, il che significa che il numero, il tipo o l'ordine dei parametri devono essere diversi.

L'overloading è utile quando si desidera fornire diverse versioni di un metodo che eseguono operazioni simili ma con input diversi. Ciò rende il codice più flessibile e leggibile, consentendo di utilizzare lo stesso nome di metodo per operazioni simili su tipi di dati diversi o con diverse combinazioni di parametri.

Nell'esempio che vedremo, la classe `Calcolatrice` ha tre metodi `somma` sovraccaricati. Il primo metodo somma accetta due interi e restituisce un intero, il secondo metodo `somma` accetta due double e restituisce un double, e il terzo metodo `somma` accetta tre interi e restituisce un intero. All'interno del metodo `main` , possiamo chiamare i metodi `somma` con diversi tipi di parametri e ottenere risultati diversi in base alla firma dei metodi. Questo è possibile grazie all'overloading dei metodi, che ci consente di utilizzare lo stesso nome di metodo per operazioni simili ma con parametri diversi.

```
1   public class Calcolatrice {
2       public int somma(int a, int b) {
3           return a + b;
4       }
5       public double somma(double a, double b) {
6           return a + b;
7       }
8
9       public int somma(int a, int b, int c) {
10          return a + b + c;
11      }
12      public static void main(String[] args) {
13          Calcolatrice calcolatrice = new Calcolatrice();
14          int risultato1 = calcolatrice.somma(3, 5);
15          System.out.println("Risultato 1: " + risultato1);
16          double risultato2 = calcolatrice.somma(2.5, 4.7);
17          System.out.println("Risultato 2: " + risultato2);
18          int risultato3 = calcolatrice.somma(1, 2, 3);
19          System.out.println("Risultato 3: " + risultato3);
20      }
21  }
```

4.2 Overriding

Overriding

L'overriding è un concetto al polimorfismo e all'ereditarietà che si applica ai metodi nelle classi derivate. Consiste nell'implementare un metodo nella classe figlia con la stessa firma (nome e parametri) di un metodo definito nella classe genitore, in modo che il comportamento del metodo possa essere personalizzato nella classe figlia.

Quando una classe figlia eredita un metodo dalla classe genitore, può scegliere di sovrascriverlo fornendo una nuova implementazione che sostituisce quella della classe genitore. Ciò permette alla classe figlia di definire il comportamento specifico desiderato per quel metodo.

Nell'esempio che mostreremo, la classe `Vaicolo` ha un metodo `accellera` che viene ereditato sia dalla classe `Auto` che dalla classe `Moto` . Tuttavia, entrambe le classi figlie decidono di sovrascrivere il metodo `accellera` fornendo un'implementazione specifica per ciascuna di esse. Quando chiamiamo il metodo `accellera` sugli oggetti `Auto` e `Moto` , viene eseguita l'implementazione specifica definita nelle rispettive classi figlie. Ciò significa che l'output sarà diverso a seconda del tipo di oggetto su cui viene chiamato il metodo.

L'overriding dei metodi ci consente di personalizzare il comportamento dei metodi nelle classi figlie, consentendo una maggiore flessibilità e specializzazione del codice.

```
1    public class Veicolo {
2        public void accelera() {
3            System.out.println("Il veicolo accelera.");
4            }
5        }
6        public class Moto extends Veicolo {
7        @Override
8            public void accelera() {
9                System.out.println("La moto accelera velocemente.");
10           }
11       }
12
13       public class Main {
14           public static void main(String[] args) {
15           Veicolo veicolo1 = new Veicolo();
16           veicolo1.accelera();
17           Moto moto = new Moto();
18           moto.accelera();
19       }
20   }
```

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.

5 Interfacce

Interfaccia

Un'interfaccia è una collezione di metodi astratti (senza implementazione) che definiscono un contratto o un set di comportamenti che una classe deve implementare. È un'entità chiave nell'approccio di programmazione orientata agli oggetti, poiché consente la definizione di contratti comuni e l'implementazione di polimorfismo.

1. Un'interfaccia viene definita utilizzando la parola chiave interface.
2. I metodi all'interno di un'interfaccia sono implicitamente astratti e non contengono un'implementazione concreta.
3. Le interfacce possono contenere anche costanti (variabili final e static) e metodi di default (metodi con un'implementazione predefinita).
4. Una classe può implementare più interfacce, fornendo un'implementazione per tutti i metodi definiti nelle interfacce.
5. Le interfacce vengono implementate utilizzando la parola chiave implements.
6. Un'interfaccia può estendere altre interfacce, consentendo la creazione di gerarchie di interfacce.
7. Le interfacce consentono la creazione di tipi di dati astratti, che possono essere utilizzati per definire comportamenti comuni tra diverse classi.

Le interfacce in Java servono a definire un "contratto" che le classi possono implementare. Le classi che implementano un'interfaccia devono fornire un'implementazione per tutti i metodi dichiarati nell'interfaccia, questo significa che la classe deve implementare tutti i metodi dichiarati nell'interfaccia e garantire che essi siano conformi alle specifiche definite dall'interfaccia.

5.1 Abstract

In Java, la parola chiave **abstract** viene utilizzata per dichiarare classi e metodi astratti. Il concetto di "astratto" indica che la classe o il metodo ha un'implementazione incompleta o non specifica.

5.1.1 Metodo Astratto

Un metodo astratto, dichiarato con la parola chiave `abstract` , non ha un'implementazione specifica nella classe in cui è dichiarato. La sua firma (nome, parametri e tipo di ritorno) è definita, ma l'implementazione effettiva viene fornita solo dalle sottoclassi concrete che ereditano il metodo astratto. Le sottoclassi devono fornire l'implementazione del metodo astratto utilizzando l'annotazione `@Override` per indicare che si tratta di un'implementazione del metodo ereditato.

5.1.2 Classe Astratta

Quando una classe viene dichiarata come astratta con la parola chiave "abstract", significa che non può essere istanziata direttamente, ma può essere utilizzata solo come classe base per le sue sottoclassi concrete. Una classe astratta può contenere metodi concreti, ma può anche contenere metodi astratti che non hanno un'implementazione definita nella classe astratta stessa. Le sottoclassi concrete devono fornire l'implementazione dei metodi astratti ereditati.

Di seguito c'è un esempio in cui vengono utilizzati una classe e un metodo astratti:

```
1     public abstract class Veicolo {
2         public abstract void accelera();
3         public void frena() {
4             System.out.println("Il veicolo sta frenando");
5         }
6     }
7     public class Auto extends Veicolo {
8         @Override
9         public void accelera() {
10            System.out.println("L'auto sta accelerando");
11        }
12    }
13    public class Moto extends Veicolo {
14        @Override
15        public void accelera() {
16            System.out.println("La moto sta accelerando");
17        }
18    }
19    public class AbstractExample {
20        public static void main(String[] args) {
21            Veicolo veicolo1 = new Auto();
22            Veicolo veicolo2 = new Moto();
23            veicolo1.accelera();
24            veicolo2.accelera();
25            veicolo1.frena();
26            veicolo2.frena();
27        }
28    }
```

6 Eccezioni

Eccezioni

La **gestione delle eccezioni** in Java è un modo per affrontare e gestire gli errori che possono verificarsi durante l'esecuzione di un programma in modo controllato. Quando un'eccezione si verifica, può interrompere il normale flusso del programma. La gestione delle eccezioni consente di catturare queste eccezioni e fornire un modo per gestirle senza interrompere il programma completamente.

6.1 Try-Catch

Appunti Java286.1 Eccezioni Controllate (Try-Catch) Le eccezioni controllate sono eccezioni che vengono individuate durante la compilazione e che lo sviluppatore é obbligato a gestire, pena la morte (e la non compilazione).

Nel gestire le eccezioni, è possibile utilizzare due meccanismi principali: l'uso delle istruzioni `try-catch`. L'istruzione `try-catch` consente di delimitare un blocco di codice che potrebbe generare un'eccezione, e di catturare e gestire l'eccezione se si verifica. Può essere utilizzata nel seguente modo:

```
1    try {
2        // Blocco di codice che potrebbe generare un'eccezione
3    } catch (TipoEccezione1 e1) {
4        // Gestione specifica dell'eccezione di TipoEccezione1
5    } catch (TipoEccezione2 e2) {
6        // Gestione specifica dell'eccezione di TipoEccezione2
7    } finally {
8        // Blocco di codice eseguito sempre, anche se viene generata
9    }
```

Nel blocco `try` , viene inserito il codice che potrebbe generare un'eccezione. Se un'eccezione viene sollevata, il controllo passa al blocco `catch` corrispondente, che gestisce l'eccezione specifica. È possibile avere più clausole `catch` per gestire diversi tipi di eccezioni.

6.2 Eccezioni Non Controllate

Le eccezioni non controllate, anche conosciute `RuntimeException`, sono eccezioni che non richiedono una gestione esplicita da parte del programmatore perché si verificano solo in alcune interazioni da parte dell'utente. A differenza delle eccezioni controllate, che devono essere dichiarate nella firma del metodo o gestite utilizzando `try-catch`, le eccezioni non controllate non richiedono queste misure preventive.

Le eccezioni non controllate derivano dalla classe `RuntimeException` o dalle sue sottoclassi. Alcuni esempi comuni di eccezioni non controllate includono `NullPointerException` , `ArrayIndexOutOfBoundsException` , `ArithmeticException` e `IllegalArgumentException`.

Poiché le eccezioni non controllate non richiedono una gestione esplicita, possono propagarsi attraverso i livelli di chiamata dei metodi fino a quando non vengono catturate o raggiungono il punto in cui il programma viene terminato a causa di un'eccezione non gestita. Ciò può rendere il Appunti Java29codice più conciso e leggibile, ma richiede una maggiore attenzione nella gestione delle situazioni in cui queste eccezioni possono verificarsi.

Ecco un esempio di codice che solleva un'eccezione non controllata:

```
1    public class EsempioEccezioneNonControllata {
2        public static void main(String[] args) {
3            int[] numeri = {1, 2, 3};
4            int risultato = numeri[4];
5            System.out.println("Risultato: " + risultato);
6        }
7    }
```

Nell'esempio sopra, l'accesso all'indice `4` nell'array `numeri` provocherà un'eccezione `ArrayIndexOutOfBoundsException` perché l'array ha solo tre elementi. Questa eccezione è un esempio di eccezione non controllata e verrà sollevata durante l'esecuzione del programma. A meno che non venga gestita con un blocco `try-catch` , il programma terminerà con un'eccezione non gestita.

Le eccezioni non controllate vengono spesso utilizzate per segnalare errori di programmazione, come errori di logica o violazioni delle precondizioni. Tuttavia, è sempre consigliabile gestire esplicitamente le eccezioni non controllate se si desidera fornire un'elaborazione specifica per tali situazioni o se si desidera terminare il programma in modo controllato in caso di eccezione non gestita.

7 Strutture dati

7.1 ArrayList

ArrayList

Gli `ArrayList` sono molto simili agli array tradizionali, ma hanno la caratteristica di poter crescere o diminuire dinamicamente la loro dimensione in base alle necessità del programma. Ciò significa che non è necessario specificare la dimensione dell' `ArrayList` al momento della creazione, a differenza degli array tradizionali in cui la dimensione deve essere definita in anticipo.

Vediamo un esempio:

```
import java.util.ArrayList;
class Persona {
    private String nome;
    private int eta;
    public Persona(String nome, int eta) {
        this.nome = nome;
        this.eta = eta;
    }
    public String getNome() {
        return nome;
    }
    public int getEta() {
        return eta;
    }
}
public class EsempioArrayListOggetti {
    public static void main(String[] args) {
        // Creazione di un ArrayList di oggetti Persona
        ArrayList<Persona> persone = new ArrayList<>();
        // Creazione di oggetti Persona e aggiunta all'ArrayList
        Persona persona1 = new Persona("Mario", 25);
        persone.add(persona1);
        // Accesso agli oggetti Persona nell'ArrayList
        Persona primaPersona = persone.get(0);
        System.out.println(primaPersona.getNome()); // Stampa: "
        System.out.println(primaPersona.getEta()); // Stampa: 25
        // Iterazione sugli oggetti Persona nell'ArrayList
        for (Persona persona : persone) {
            System.out.println(persona.getNome()) + " - " + perso
        }
        persone.remove(persona1); // Rimozione di un oggetto
        System.out.println(persone.size()); // Stampa: 0
    }
}
```

7.2 Array vs ArrayList

Array	ArrayList
Gli array hanno una dimensione fissa che viene specificata durante la loro creazione e non può essere modificata successivamente.	Gli ArrayList hanno una dimensione dinamica e possono essere ridimensionati dinamicamente aggiungendo o rimuovendo elementi.
Gli array possono contenere elementi di un singolo tipo di dato, che viene specificato durante la loro creazione	Gli ArrayList possono contenere elementi di qualsiasi tipo di dato, poiché utilizzano una rappresentazione generica attraverso il concetto di tipo parametrico.
Gli array, d'altra parte, hanno un insieme limitato di funzionalità integrate e richiedono spesso l'implementazione manuale di algoritmi per operazioni come l'inserimento, la rimozione o la ricerca di elementi.	Gli ArrayList forniscono un set di metodi e funzionalità integrate che semplificano la gestione e la manipolazione dei dati. Questi metodi includono l'aggiunta di elementi, la rimozione di elementi, la ricerca di elementi, la verifica della presenza di un elemento e molto altro.
Gli array forniscono un controllo diretto sugli indici degli elementi, consentendo un accesso rapido e diretto a un elemento specifico attraverso la sua posizione nell'array	Gli ArrayList richiedono l'utilizzo dei loro metodi per accedere agli elementi, poiché non forniscono un accesso diretto agli indici.
Gli array sono tipi di dato primitivi e vengono memorizzati come blocchi continui di memoria.	Gli ArrayList in Java non possono essere direttamente di tipo primitivo come int, char, boolean, etc. Gli ArrayList sono basati sul concetto di tipo parametrico o generico, il che significa che possono contenere solo oggetti di classi o tipi di riferimento. Tuttavia, Java fornisce classi wrapper per i tipi primitivi, come Integer, Character, Boolean, etc., che possono essere utilizzate per rappresentare i tipi primitivi come oggetti. Quindi, puoi utilizzare gli ArrayList di questi wrapper per ottenere una funzionalità simile agli array di tipi primitivi.

7.3 Linked List

Linked List

Le Linked List (liste concatenate) sono strutture dati dinamiche utilizzate per memorizzare una sequenza di elementi. In Java, le Linked List sono implementate dalla classe LinkedList che fa parte della libreria standard.

Appunti Java33A differenza degli array, le Linked List non utilizzano un blocco di memoria continuo per memorizzare i loro elementi. Invece, ogni elemento della lista, chiamato nodo, contiene un valore e un riferimento al nodo successivo nella sequenza. Questo collegamento tra i nodi consente di attraversare la lista in modo efficiente.

Esempio:

```
    import java.util.LinkedList;
    public class LinkedListExample {
        public static void main(String[] args) {
        // Creazione di una LinkedList
        LinkedList<String> linkedList = new LinkedList<>();
        // Aggiunta di elementi alla LinkedList
        linkedList.add("Elemento 1");
        linkedList.add("Elemento 2");
        linkedList.add("Elemento 3");
        // Accesso agli elementi della LinkedList
        String primoElemento = linkedList.get(0);
        System.out.println("Primo elemento: " + primoElemento);
        // Iterazione attraverso gli elementi della LinkedList
        for (String elemento : linkedList) {
            System.out.println(elemento);
        }
        // Rimozione di un elemento dalla LinkedList
        linkedList.remove(1);
        System.out.println("Elemento rimosso.");
        // Stampa della dimensione della LinkedList
        System.out.println("Dimensione della LinkedList: " + lin
        }
    }
```

A differenza degli Array le Linked List sono implementate come una lista concatenata, in cui ogni elemento contiene un riferimento al successivo. Le operazioni di inserimento e rimozione sono più efficienti rispetto ad ArrayList. Non richiedono il ridimensionamento dell'array, ma richiedono il collegamento dei nodi all'interno della lista. Non offre un accesso diretto agli elementi tramite l'indice. Per accedere a un elemento specifico, è necessario attraversare la lista partendo dall'inizio o dalla fine. L'accesso casuale è meno efficiente. Richiede più memoria rispetto ad ArrayList, poiché ogni elemento deve memorizzare un riferimento al successivo.

7.4 Code