



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	3
1.1	L'Object Oriented Programming	3
1.2	Tipi di Dati	4
1.3	Gli Identificatori	4
1.4	Le Classi	5
1.5	I Metodi	6
1.6	Gli Oggetti	7
1.7	I Cicli	8
2	2.0 I modificatori di accesso	9
2.1	Public, Private, Protected e Default	9
2.2	Static e Final	10
2.2.1	Final	10
2.2.2	Static	11
3	Ereditarietà	12
3.1	Super e This	13
4	Polimorfismo	15
4.1	Overloading	15
4.2	Overriding	16
5	Interfacce	18
5.1	Abstract	18
5.1.1	Metodo Astratto	19
5.1.2	Classe Astratta	19
6	Eccezioni	20
6.1	Try-Catch	20
6.2	Eccezioni Non Controllate	20
7	Strutture dati	22
7.1	ArrayList	22
7.2	Array vs ArrayList	23
7.3	Linked List	24
7.4	Code	25
7.5	Stack (Pila)	26
8	I figli dell'Interfaccia Set	27
8.1	HashSet	28
8.1.1	ArrayList vs HashSet	29

8.2	TreeSet	30
8.2.1	HashSet vs TreeSet	31
9	I figli dell'Interfaccia Map	32
9.1	TreeMap	33
9.1.1	TreeSet vs TreeMap	34
10	I principi SOLID	35
11	Java Conventions	36

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

Esempio:

```
1 public class MyClass {
2     final int myFinalVariable;
3     public MyClass() {
4         // Inizializzazione della variabile finale nel costruttore
5         myFinalVariable = 10;
6     }
7 }
```

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.

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:

```
1  class Veicolo {
2      protected String marca;
3      protected String modello;
4      public Veicolo(String marca, String modello) {
5          this.marca = marca;
6          this.modello = modello;
7      }
8      public void accelera() {
9          System.out.println("Il veicolo accelera");
10     }
11     public void frena() {
12         System.out.println("Il veicolo frena");
13     }
14 }
15
16 class Auto extends Veicolo {
17     private int numeroPorte;
18
19     public Auto(String marca, String modello, int numeroPorte) {
20         super(marca, modello);
21         this.numeroPorte = numeroPorte;
22     }
23     public void apriPorte() {
24         System.out.println("Apri le porte dell'auto");
25     }
26 }
27
```

```

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 **accelera()** 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`:

```
1  class Veicolo {
2      String marca;
3      public Veicolo(String marca) {
4          this.marca = marca;
5      }
6      Appunti Java18public void stampaMarca() {
7          System.out.println("Marca: " + marca);
8      }
9  }
10
11  class Auto extends Veicolo {
12      int anno;
13      public Auto(String marca, int anno) {
14          super(marca);
15          this.anno = anno;
16      }
17      @Override
18      public void stampaMarca() {
19          super.stampaMarca();
20          System.out.println("Anno: " + anno);
21      }
22  }
23
24  public class Test {
25      public static void main(String[] args) {
26          Auto auto = new Auto("BMW", 2021);
27          auto.stampaMarca();
28      }
29  }
```

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 è 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 sotto-classi. 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 codice 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:

```
1  import java.util.ArrayList;
2  class Persona {
3      private String nome;
4      private int eta;
5      public Persona(String nome, int eta) {
6          this.nome = nome;
7          this.eta = eta;
8      }
9      public String getNome() {
10         return nome;
11     }
12     public int getEta() {
13         return eta;
14     }
15 }
16 public class EsempioArrayListOggetti {
17     public static void main(String[] args) {
18         // Creazione di un ArrayList di oggetti Persona
19         ArrayList<Persona> persone = new ArrayList<>();
20         // Creazione di oggetti Persona e aggiunta all'ArrayList
21         Persona persona1 = new Persona("Mario", 25);
22         persone.add(persona1);
23         // Accesso agli oggetti Persona nell'ArrayList
24         Persona primaPersona = persone.get(0);
25         System.out.println(primaPersona.getNome()); // Stampa: "
26         System.out.println(primaPersona.getEta()); // Stampa: 25
27         // Iterazione sugli oggetti Persona nell'ArrayList
28         for (Persona persona : persone) {
29             System.out.println(persona.getNome()) + " - " + perso
30         }
31         persone.remove(persona1); // Rimozione di un oggetto
32         System.out.println(persone.size()); // Stampa: 0
33     }
34 }
```

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:

```
1  import java.util.LinkedList;
2  public class LinkedListExample {
3      public static void main(String[] args) {
4          // Creazione di una LinkedList
5          LinkedList<String> linkedList = new LinkedList<>();
6          // Aggiunta di elementi alla LinkedList
7          linkedList.add("Elemento 1");
8          linkedList.add("Elemento 2");
9          linkedList.add("Elemento 3");
10         // Accesso agli elementi della LinkedList
11         String primoElemento = linkedList.get(0);
12         System.out.println("Primo elemento: " + primoElemento);
13         // Iterazione attraverso gli elementi della LinkedList
14         for (String elemento : linkedList) {
15             System.out.println(elemento);
16         }
17         // Rimozione di un elemento dalla LinkedList
18         linkedList.remove(1);
19         System.out.println("Elemento rimosso.");
20         // Stampa della dimensione della LinkedList
21         System.out.println("Dimensione della LinkedList: " + linkedList.size());
22     }
23 }
```

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

Coda

È un'interfaccia che rappresenta una coda, ovvero una struttura dati in cui gli elementi vengono inseriti alla fine e rimossi dall'inizio.

In Java, una Queue è un'interfaccia che rappresenta una struttura dati di tipo coda, che segue il principio FIFO (First-In-First-Out). Ciò significa che l'elemento che viene inserito per primo nella coda è anche il primo ad essere rimosso.

Ecco un esempio di utilizzo della Queue in Java utilizzando la classe LinkedList come implementazione:

```
1  import java.util.Queue;
2  import java.util.LinkedList;
3  public class Main {
4      public static void main(String[] args) {
5          Queue<String> queue = new LinkedList<>();
6          // Aggiunta degli elementi alla coda
7          queue.add("Elemento 1");
8          queue.add("Elemento 2");
9          queue.add("Elemento 3");
10         // Stampa della coda
11         System.out.println("Contenuto della coda: " + queue);
12         // Rimozione dell'elemento in testa alla coda
13         String elemento = queue.poll();
14         System.out.println("Elemento rimosso dalla coda: " + elemento);
15     };
16     // Stampa della coda dopo la rimozione
17     System.out.println("Contenuto della coda dopo la rimozione: "
18 + queue);
19 }
```

In questo esempio, abbiamo creato una Queue utilizzando la classe LinkedList come implementazione. Abbiamo aggiunto tre elementi alla coda utilizzando il metodo `add()`. Successivamente, stampiamo il contenuto della coda utilizzando la sintassi `System.out.println(queue)`. Quindi, utilizziamo il metodo `poll()` per rimuovere l'elemento in testa alla coda e lo assegniamo a una variabile chiamata `elemento`. Infine, stampiamo il contenuto della coda dopo la rimozione.

L'output del programma sarà:

```
1   Contenuto della coda: [Elemento 1, Elemento 2, Elemento 3]
2   Elemento rimosso dalla coda: Elemento 1
3   Contenuto della coda dopo la rimozione: [Elemento 2, Elemento 3]
```

L'implementazione `LinkedList` è una scelta comune per implementare una `Queue` in Java poiché fornisce un'efficace gestione della coda, consentendo l'aggiunta e la rimozione degli elementi in modo efficiente.

Alcune delle implementazioni più comuni di `Queue` sono:

- **ArrayDeque:** È una classe che implementa l'interfaccia `Deque` (Double Ended Queue) ed è basata su un array dinamico. Supporta operazioni di inserimento e rimozione efficienti sia all'inizio che alla fine della coda.
- **PriorityQueue:** È una classe che implementa l'interfaccia `Queue` utilizzando un heap binario. Gli elementi all'interno di `PriorityQueue` vengono ordinati in base al loro ordine naturale o utilizzando un comparatore specificato.

Sia `LinkedList` che `Queue` sono ampiamente utilizzate per gestire collezioni di dati in Java. `LinkedList` è adatta quando è necessario un accesso efficiente agli elementi tramite l'indice, oltre alle operazioni di inserimento e rimozione in diverse posizioni. `Queue`, d'altra parte, si concentra sulle operazioni di inserimento e rimozione agli estremi della coda, seguendo l'approccio "First-In-First-Out" (FIFO).

7.5 Stack (Pila)

Stack

In Java, lo stack è una struttura dati che segue il principio "last in, first out" (LIFO). Ciò significa che l'elemento più recentemente inserito nello stack è il primo ad essere rimosso.

Java fornisce un'implementazione dello stack attraverso la classe `java.util.stack`. La classe `Stack` estende la classe `Vector` e fornisce metodi specifici per gestire lo stack, come `push()` per inserire un elemento, `pop()` per rimuovere l'elemento superiore, `peek()` per accedere all'elemento superiore senza rimuoverlo e altri metodi utili per la gestione dello stack.

Ecco un esempio di utilizzo dello stack in Java:

```
1  import java.util.Stack;
2  public class StackExample {
3      public static void main(String[] args) {
4          Stack<String> stack = new Stack<>();
5          // Inserimento di elementi nello stack
6          stack.push("Elemento 1");
7          stack.push("Elemento 2");
8          stack.push("Elemento 3");
9          // Stampa dello stack
10         System.out.println("Contenuto dello stack: " + stack);
11         // Rimozione dell'elemento superiore dello stack
12         String elemento = stack.pop();
13         System.out.println("Elemento rimosso dallo stack: " +
14         elemento);
15         // Stampa dello stack dopo la rimozione
16         System.out.println("Contenuto dello stack dopo la
17         rimozione: " + stack);
18         // Accesso all'elemento superiore dello stack senza
19         rimuoverlo
20         String elementoSuperiore = stack.peek();
21         System.out.println("Elemento superiore dello stack: " +
22         elementoSuperiore);
23     }
24 }
```

Nell'esempio sopra, abbiamo creato uno stack utilizzando la classe `Stack` e abbiamo inserito alcuni elementi nello stack utilizzando il metodo `push()`. Successivamente, abbiamo utilizzato il metodo `peek()` per accedere all'elemento superiore dello stack senza rimuoverlo e il metodo `pop()` per rimuovere l'elemento superiore. Infine, abbiamo stampato gli elementi rimanenti nello stack utilizzando il metodo `toString()` implicito.

8 I figli dell'Interfaccia Set

In Java, un Set è un'interfaccia che rappresenta una collezione di elementi unici, senza duplicati.

Alcune caratteristiche importanti dei Set includono:

- Un Set non può contenere elementi duplicati. Ogni elemento nel Set è unico e presente al massimo una volta.
- Un Set non mantiene un ordine specifico degli elementi. Non puoi fare affidamento sul fatto che gli elementi vengano memorizzati o restituiti in un particolare ordine.
- I metodi principali forniti da un'implementazione di Set includono l'aggiunta di un elemento (`add`), la rimozione di un elemento (`remove`), la verifica dell'appartenenza di un elemento (`contains`), il conteggio degli elementi (`size`) e l'iterazione sugli elementi.

8.1 HashSet

HashSet

Un HashSet in Java è una classe che implementa l'interfaccia Set e utilizza una struttura di dati chiamata "hash table" per memorizzare e gestire un insieme di elementi. Un HashSet non consente elementi duplicati e non garantisce un ordine specifico degli elementi.

Ecco un esempio di utilizzo di HashSet:

```
1  import java.util.HashSet;
2  public class HashSetExample {
3      public static void main(String[] args) {
4          // Creazione di un HashSet di stringhe
5          HashSet<String> set = new HashSet<>();
6          // Aggiunta di elementi al set
7          set.add("Apple");
8          set.add("Banana");
9          set.add("Orange");
10         set.add("Grapes");
11         // Stampa del set
12         System.out.println("HashSet: " + set);
13         boolean containsApple = set.contains("Apple");
14         System.out.println("Contains Apple: " + containsApple);
15         // Rimozione di un elemento dal set
16         set.remove("Orange");
17         // Stampa del set aggiornato
18         System.out.println("Updated HashSet: " + set);
19         // Iterazione attraverso gli elementi del set
20         System.out.println("Iterating over HashSet:");
21         for (String fruit : set) {
22             System.out.println(fruit);
23         }
24         // Dimensione del set
25         int size = set.size();
26         System.out.println("Size of HashSet: " + size);
27         boolean isEmpty = set.isEmpty();
28         System.out.println("Is HashSet empty? " + isEmpty);
29         // Rimozione di tutti gli elementi dal set
30         set.clear();
31         System.out.println("HashSet after clear: " + set);
32     }
33 }
```

HashSet mantiene gli elementi unici e non mantiene un ordine specifico.

8.1.1 ArrayList vs HashSet

ARRAYLIST	HASHSET
ArrayList mantiene l'ordine di inserimento degli elementi.	HashSet non garantisce un ordine specifico
ArrayList consente duplicati, il che significa che puoi avere più elementi con lo stesso valore.	HashSet, non consente duplicati, quindi ogni elemento nell'HashSet deve essere unico.
ArrayList consente l'accesso agli elementi utilizzando un indice specifico.	per accedere agli elementi in un HashSet, bisogna utilizzare metodi come contains() o iterare sull'HashSet.
ArrayList è implementato come un array ordinato, quindi alcune operazioni come l'inserimento o la rimozione possono richiedere spostamenti di elementi all'interno dell'array	HashSet utilizza la struttura dati hash table per archiviare gli elementi, il che offre un'accesso più rapido in base al valore hash degli elementi.
Internamente, gli ArrayList utilizzano un array per archiviare i dati.	HashSet in Java utilizza internamente una HashMap per implementare la sua struttura dati.
<p>Un ArrayList può contenere valori null. Puoi inserire valori null all'interno di un ArrayList e recuperarli. Un HashSet può contenere al massimo un valore null. Se si prova a inserire più valori null, verrà accettato solo il primo valore null e i successivi.</p> <p>Di seguito un esempio con codice:</p> <pre>1 ArrayList<String> list = new A 2 list.add("Hello"); 3 list.add(null); 4 list.add("World"); 5 System.out.println(list); 6</pre>	<p>Un HashSet può contenere al massimo un valore null. Se si prova a inserire più valori null, verrà accettato solo il primo valore null e i successivi.</p> <pre>1 HashSet<String> set = new HashSet 2 <>(); 3 set.add("Hello"); 4 set.add(null); 5 set.add("World"); 6 set.add(null); 7 System.out.println(set);</pre>

8.2 TreeSet

TreeSet

Un TreeSet in Java è una classe che implementa l'interfaccia Set e fornisce un'implementazione basata su un albero bilanciato (di solito un albero rosso-nero) per memorizzare e gestire un insieme di elementi. A differenza di HashSet, un TreeSet ordina gli elementi in base all'ordine naturale o tramite un comparatore personalizzato.

Ecco un esempio di utilizzo di TreeSet:

```
1  import java.util.TreeSet;
2  public class TreeSetExample {
3      public static void main(String[] args) {
4          // Creazione di un TreeSet di stringhe
5          TreeSet<String> set = new TreeSet<>();
6          // Aggiunta di elementi al set
7          set.add("Apple");
8          set.add("Banana");
9          set.add("Orange");
10         set.add("Grapes");
11         // Stampa del set
12         System.out.println("TreeSet: " + set);
13         boolean containsApple = set.contains("Apple");
14         System.out.println("Contains Apple: " + containsApple);
15         // Rimozione di un elemento dal set
16         set.remove("Orange");
17         // Stampa del set aggiornato
18         System.out.println("Updated TreeSet: " + set);
19         // Iterazione attraverso gli elementi del set
20         System.out.println("Iterating over TreeSet:");
21         for (String fruit : set) {
22             System.out.println(fruit);
23         }
24         // Dimensione del set
25         int size = set.size();
26         System.out.println("Size of TreeSet: " + size);
27         boolean isEmpty = set.isEmpty();
28         System.out.println("Is TreeSet empty? " + isEmpty);
29         // Rimozione di tutti gli elementi dal set
30         set.clear();
31         System.out.println("TreeSet after clear: " + set);
32     }
33 }
```

In questo esempio, viene creato un `TreeSet` di interi chiamato "set". Vengono aggiunti alcuni numeri al set utilizzando il metodo `add()`, viene verificata la presenza di un numero con `contains()`, viene rimosso un numero con `remove()`, viene iterato attraverso gli elementi del set con un loop for-each e vengono mostrate alcune operazioni comuni come la dimensione del set, la verifica se è vuoto e la rimozione di tutti gli elementi con `clear()`.

8.2.1 HashSet vs TreeSet

HashSet	TreeSet
HashSet non mantiene un ordine specifico degli elementi. Gli elementi sono organizzati in base al loro valore hash.	TreeSet mantiene gli elementi ordinati in base al loro ordine naturale (seguendo l'ordinamento naturale degli elementi).
HashSet utilizza una struttura dati hash table internamente per archiviare gli elementi.	TreeSet utilizza una struttura dati di albero bilanciato (generalmente un albero rosso-nero) per archiviare gli elementi.
HashSet offre prestazioni migliori per operazioni come l'inserimento, la rimozione e la ricerca, con complessità di tempo media $O(1)$.	TreeSet richiede una complessità di tempo $O(\log n)$ per operazioni come l'inserimento, la rimozione e la ricerca, poiché gli elementi devono essere mantenuti ordinati all'interno dell'albero.
HashSet non fornisce un accesso diretto agli elementi tramite indici, poiché non mantiene un ordine specifico. Per accedere agli elementi in HashSet, bisogna utilizzare metodi come <code>contains()</code> o iterare sull'insieme.	TreeSet fornisce un accesso diretto agli elementi tramite iteratori, consentendo la navigazione in base all'ordine degli elementi all'interno dell'albero.

9 I figli dell'Interfaccia Map

HashMap

Gli HashMap sono una classe che implementa l'interfaccia Map e sono delle strutture dati che memorizzano gli elementi in coppie chiave-valore. È basato sul concetto di tabella hash, dove ogni elemento viene associato a una chiave univoca (come i dizionari in python). L'HashMap permette di accedere rapidamente agli elementi utilizzando la chiave.

Ecco un esempio di utilizzo di HashMap:

```
1  import java.util.HashMap;
2  public class HashMapExample {
3      public static void main(String[] args) {
4          // Creazione di un HashMap di stringhe
5          HashMap<String, String> map = new HashMap<>();
6          // Aggiunta di coppie chiave-valore all'HashMap
7          map.put("Italy", "Rome");
8          map.put("France", "Paris");
9          map.put("Germany", "Berlin");
10         // Stampa dell'HashMap
11         System.out.println("HashMap: " + map);
12         // Accesso al valore tramite la chiave
13         String capital = map.get("Italy");
14         System.out.println("Capital of Italy: " + capital);
15         // Rimozione di una coppia chiave-valore dall'HashMap
16         map.remove("France");
17         // Stampa dell'HashMap aggiornato
18         System.out.println("Updated HashMap: " + map);
19         // Iterazione attraverso le coppie chiave-valore dell'
20         HashMap
21         System.out.println("Iterating over HashMap:");
22         for (String country : map.keySet()) {
23             String capitalCity = map.get(country);
24             System.out.println(country + " - " + capitalCity);
25         }
26         // Dimensione dell'HashMap
27         int size = map.size();
28         System.out.println("Size of HashMap: " + size);
29         boolean isEmpty = map.isEmpty();
30         System.out.println("Is HashMap empty? " + isEmpty);
31         // Rimozione di tutte le coppie chiave-valore dall'HashMap
32         map.clear();
33         System.out.println("HashMap after clear: " + map);
34     }
}
```

9.1 TreeMap

TreeMap

TreeMap è una classe nella libreria Java che implementa l'interfaccia Map e fornisce un'implementazione ordinata dei dati in base alla chiave. A differenza di HashMap, TreeMap mantiene un ordine naturale delle chiavi utilizzando una struttura di albero bilanciato (solitamente un albero rosso-nero)

Le caratteristiche principali di TreeMap sono:

1. Ordine delle chiavi: Le chiavi all'interno di un TreeMap sono automaticamente ordinate in base al loro valore naturale o all'ordine specificato tramite un comparatore personalizzato. Ciò significa che è possibile iterare sugli elementi del TreeMap in un ordine definito.
2. Implementazione basata su albero: TreeMap utilizza una struttura di albero bilanciato per archiviare i dati. Ciò garantisce un'efficienza di ricerca e un'operazione di inserimento/rimozione di $O(\log n)$, rendendo TreeMap ideale per grandi quantità di dati o per applicazioni in cui è richiesta un'alta efficienza nelle operazioni di ricerca.
3. Duplicati: A differenza di HashSet, TreeMap consente la presenza di chiavi duplicate. Tuttavia, i valori associati alle chiavi duplicate non possono essere duplicati. Se si tenta di inserire una chiave duplicata con un valore diverso, il valore associato alla chiave verrà sovrascritto.
4. Iterazione ordinata: Grazie alla natura ordinata di TreeMap, è possibile iterare sugli elementi in base all'ordine delle chiavi. Ciò rende più facile l'elaborazione dei dati in un ordine specifico.

Ecco un esempio di utilizzo di TreeMap:

```
1 import java.util.TreeMap;
2 public class TreeMapExample {
3     public static void main(String[] args) {
4         // Creazione di un TreeMap di stringhe
5         TreeMap<String, String> map = new TreeMap<>();
6         // Aggiunta di coppie chiave-valore al TreeMap
7         map.put("Italy", "Rome");
8         map.put("France", "Paris");
9         map.put("Germany", "Berlin");
10        // Stampa del TreeMap
11        System.out.println("TreeMap: " + map);
12        // Accesso al valore tramite la chiave
13        String capital = map.get("Italy");
14        System.out.println("Capital of Italy: " + capital);
```

```

15         // Rimozione di una coppia chiave-valore dal TreeMap
16         map.remove("France");
17         // Stampa del TreeMap aggiornato
18         System.out.println("Updated TreeMap: " + map);
19         // Iterazione attraverso le coppie chiave-valore del
TreeMap
20         System.out.println("Iterating over TreeMap:");
21         for (String country : map.keySet()) {
22             String capitalCity = map.get(country);
23             System.out.println(country + " - " + capitalCity);
24         }
25         // Dimensione del TreeMap
26         int size = map.size();
27         System.out.println("Size of TreeMap: " + size);
28         boolean isEmpty = map.isEmpty();
29         System.out.println("Is TreeMap empty? " + isEmpty);
30         // Rimozione di tutte le coppie chiave-valore dal TreeMap
31         map.clear();
32         System.out.println("TreeMap after clear: " + map);
33     }
34 }

```

9.1.1 TreeSet vs TreeMap

TreeSet	TreeMap
TreeSet è un'implementazione di Set, che significa che memorizza solo le chiavi senza alcun valore associato.	TreeMap è un'implementazione di Map, che significa che memorizza coppie chiave-valore.
Utilizza una struttura di albero (solitamente un albero rosso-nero) per mantenere le chiavi in un ordine ordinato.	Utilizza una struttura di albero (solitamente un albero rosso-nero) per mantenere le coppie chiave-valore in un ordine ordinato in base alle chiavi.
Non consente chiavi duplicate. Se si tenta di inserire una chiave duplicata, verrà ignorata.	Non consente chiavi duplicate. Se si tenta di inserire una chiave duplicata, il valore associato verrà sovrascritto.
Offre un'efficienza di ricerca e un'operazione di inserimento/rimozione di $O(\log n)$.	Offre un'efficienza di ricerca e un'operazione di inserimento/rimozione di $O(\log n)$.
TreeSet fornisce un accesso diretto agli elementi tramite iteratori, consentendo la navigazione in base all'ordine degli elementi.	È possibile accedere ai valori utilizzando le chiavi.
Ideale per archiviare un insieme di elementi unici ordinati.	Ideale per archiviare dati associati a una chiave univoca ordinata.

10 I principi SOLID

SOLID

SOLID è un acronimo che rappresenta cinque principi di progettazione orientata agli oggetti che aiutano a scrivere codice più pulito, manutenibile e scalabile.

1. **S - Single Responsibility Principle (Principio di Responsabilità Singola):** Una classe dovrebbe avere una sola responsabilità. Ciò significa che una classe dovrebbe essere incaricata di un'unica responsabilità all'interno del sistema e non dovrebbe essere affetta da molteplici motivi per essere modificata. Ad esempio, se abbiamo una classe "Employee" che gestisce la logica dell'impiegato e contemporaneamente si occupa dell'invio delle email, potremmo violare il principio di responsabilità singola. Invece, sarebbe meglio separare le funzionalità dell'invio delle email in una classe separata, come ad esempio una classe "EmailSender"
2. **O - Open-Closed Principle (Principio di Estensione/Apertura e Chiusura):** Le entità del software dovrebbero essere aperte all'estensione ma chiuse alle modifiche. Ciò significa che dovremmo poter estendere il comportamento di un'entità senza modificarne il codice sorgente esistente. Ad esempio, consideriamo una classe "Shape" che ha un metodo "calcolaArea()". Se volessimo aggiungere una nuova forma, come un "Triangle", non dovremmo modificare la classe "Shape" esistente. Al contrario, dovremmo estendere la classe "Shape" con una nuova classe "Triangle" e implementare il metodo "calcolaArea()" specifico per il triangolo.
3. **L - Liskov Substitution Principle (Principio di Sostituzione di Liskov):** Gli oggetti di una classe derivata devono essere sostituibili dagli oggetti della classe base senza alterare la correttezza del programma. In altre parole, una classe derivata dovrebbe essere in grado di essere utilizzata al posto della classe base senza causare problemi nel funzionamento del software. Ad esempio, consideriamo una gerarchia di classi "Animal" con una classe base "Animal" e classi derivate come "Dog" e "Cat". Se il codice accetta un oggetto di tipo "Animal", dovrebbe funzionare correttamente anche con un oggetto di tipo "Dog" o "Cat" senza causare comportamenti indesiderati.
4. **I - Interface Segregation Principle (Principio di Segregazione delle Interfacce):** I clienti non dovrebbero essere costretti ad implementare interfacce che non utilizzano. È meglio avere molte interfacce specifiche, piuttosto che una singola interfaccia generica. Invece di creare un'interfaccia generica con un gran numero di metodi, è preferibile creare diverse interfacce più specifiche. Ad esempio, consideriamo una classe "Printer" che implementa un'interfaccia "Machine". Se la classe "Printer" ha solo bisogno di metodi per stampare, non è necessario che implementi metodi non correlati come "fax()" o "scan()". Invece, potremmo creare un'interfaccia "Printer" specifica solo per i metodi di stampa.

5. **D - Dependency Inversion Principle (Principio di Inversione delle Dipendenze):** Gli oggetti dovrebbero dipendere da astrazioni, non da implementazioni concrete. I moduli di alto livello non dovrebbero dipendere direttamente dai moduli di basso livello, entrambi dovrebbero dipendere da astrazioni. Ad esempio, invece di avere una classe "Order" che dipende direttamente da una classe "PaymentGateway" specifica, possiamo invertire la dipendenza e far dipendere la classe "Order" da un'interfaccia "PaymentProcessor". In questo modo, possiamo facilmente sostituire l'implementazione concreta del "PaymentProcessor" senza dover modificare la classe "Order".

Questi principi SOLID promuovono la scrittura di codice modularizzato, disaccoppiato e flessibile, facilitando la manutenzione, l'estendibilità e la riusabilità del software.

11 Java Conventions

Le Java Code Conventions (Convenzioni di codifica Java) sono linee guida stabilite da Oracle per la scrittura del codice Java in modo uniforme e leggibile. Le convenzioni sono un insieme di regole e raccomandazioni che coprono vari aspetti del codice, come la formattazione, la denominazione delle variabili e dei metodi, l'uso dei commenti e altro ancora. L'obiettivo principale delle Java Code Conventions è quello di migliorare la leggibilità del codice e la comprensione del programma da parte di altri sviluppatori. Aderire a queste convenzioni facilita la collaborazione nel team di sviluppo e favorisce la manutenibilità del codice nel lungo termine.

1. **Formattazione:** Le convenzioni di formattazione riguardano l'uso di indentazioni, spazi bianchi, parentesi graffe e altro per migliorare la leggibilità del codice. Ad esempio, l'uso di 4 spazi per l'indentazione, l'uso di parentesi graffe separate e l'allineamento corretto dei blocchi di codice.
2. **Denominazione:** Le convenzioni di denominazione riguardano la scelta di nomi significativi per le variabili, i metodi, le classi e altre entità del codice. Ad esempio, l'uso di nomi descrittivi che riflettono il significato e lo scopo dell'elemento.
3. **Commenti:** Le convenzioni per i commenti riguardano l'uso di commenti per spiegare il codice e fornire informazioni utili. Ad esempio, l'uso di commenti Javadoc per documentare le classi e i metodi pubblici.
4. **Organizzazione del codice:** Le convenzioni per l'organizzazione del codice riguardano la suddivisione del codice in blocchi logici, l'ordine dei metodi e delle dichiarazioni, e altre considerazioni per migliorare la leggibilità e la manutenibilità del codice.
5. **Costanti:** Le costanti, cioè le variabili dichiarate come `final` e tutte in maiuscolo, vengono scritte in maiuscolo con parole separate da underscore. Ad esempio: `MAX_VALUE`, `DEFAULT_TIMEOUT`.

6. **Import:** Le dichiarazioni di importazione devono essere ordinate in modo logico e divise in blocchi separati per le importazioni del pacchetto di base, le importazioni specifiche del pacchetto e le importazioni statiche.
7. **Uso delle parentesi:** Le parentesi dovrebbero essere utilizzate in modo coerente e seguendo le convenzioni. Ad esempio, l'uso delle parentesi anche quando non sono strettamente necessarie, per migliorare la leggibilità e la chiarezza del codice.