Objektorientierte Programmierung mit Java

Die **objektorientierte Programmierung (OOP)** ist ein zentrales Paradigma der modernen Softwareentwicklung. Java, als eine der bekanntesten Programmiersprachen, ist von Grund auf objektorientiert. Dieser Artikel gibt eine Einführung in die Grundlagen der OOP mit Java und zeigt, wie diese Konzepte in der Praxis angewendet werden.

Grundlagen der Objektorientierung

Die objektorientierte Programmierung basiert auf vier grundlegenden Prinzipien:

- **Kapselung**: Daten und Methoden werden in Klassen zusammengefasst. Die interne Implementierung ist vor dem Zugriff von außen geschützt.
- **Vererbung**: Klassen können Eigenschaften und Methoden von anderen Klassen übernehmen.
- **Polymorphismus**: Objekte können in verschiedenen Formen auftreten, und Methoden können überladen oder überschrieben werden.
- **Abstraktion**: Wichtige Eigenschaften eines Objekts werden hervorgehoben, unwichtige Details ausgeblendet.

Zusätzlich gibt es wichtige Konzepte wie **Assoziation**, **Aggregation** und **Komposition**, die die Beziehungen zwischen Klassen beschreiben.

Klassen und Objekte

In Java ist eine **Klasse** eine Blaupause für Objekte. Ein **Objekt** ist eine Instanz dieser Klasse.

Beispiel für eine einfache Klasse:

```
public class Auto {
    // Attribute (Eigenschaften)
    String marke;
    String modell;
    int baujahr;

    // Konstruktor
    public Auto(String marke, String modell, int baujahr) {
        this.marke = marke;
        this.modell = modell;
        this.baujahr = baujahr;
    }

    // Methode
    public void starten() {
        System.out.println(marke + " " + modell + " startet.");
    }
```

```
}
```

Erstellen eines Objekts:

```
public class Main {
    public static void main(String[] args) {
        Auto meinAuto = new Auto("VW", "Golf", 2020);
        meinAuto.starten();
    }
}
```

Kapselung

Die **Kapselung** schützt die Daten einer Klasse, indem sie als privat deklariert werden und nur über Methoden (Getter und Setter) zugänglich sind.

Beispiel:

```
public class Konto {
    private double saldo;

    public Konto(double anfangsSaldo) {
        this.saldo = anfangsSaldo;
    }

    public double getSaldo() {
        return saldo;
    }

    public void einzahlen(double betrag) {
        if (betrag > 0) {
            saldo += betrag;
        }
    }

    public void abheben(double betrag) {
        if (betrag > 0 && saldo >= betrag) {
            saldo -= betrag;
        }
    }
}
```

Vererbung

Vererbung erlaubt es, eine neue Klasse zu erstellen, die von einer bestehenden Klasse erbt.

Beispiel:

```
// Basisklasse
public class Tier {
    public void geraeuschMachen() {
        System.out.println("Das Tier macht ein Geräusch.");
    }
}

// Abgeleitete Klasse
public class Hund extends Tier {
    @Override
    public void geraeuschMachen() {
        System.out.println("Der Hund bellt.");
    }
}

public class Main {
    public static void main(String[] args) {
        Tier meinTier = new Hund();
        meinTier.geraeuschMachen();  // Ausgabe: Der Hund bellt.
    }
}
```

Polymorphismus

Polymorphismus bedeutet, dass eine Methode unterschiedliche Implementierungen haben kann, je nachdem, welches Objekt sie aufruft.

* **Methodenüberladung (Overloading)**: Mehrere Methoden mit demselben Namen, aber unterschiedlichen Parameterlisten. * **Methodenüberschreibung (Overriding)**: Eine Methode der Basisklasse wird in der abgeleiteten Klasse mit einer neuen Implementierung versehen.

Beispiel für Überschreiben:

```
public class Katze extends Tier {
    @Override
    public void geraeuschMachen() {
        System.out.println("Die Katze miaut.");
    }
}

public class Main {
    public static void main(String[] args) {
        Tier meinTier1 = new Hund();
        Tier meinTier2 = new Katze();

        meinTier1.geraeuschMachen();  // Der Hund bellt.
        meinTier2.geraeuschMachen();  // Die Katze miaut.
```

```
    }
}
```

Beispiel für Überladen:

```
public class Rechner {
    public int addieren(int a, int b) {
        return a + b;
    }

    public double addieren(double a, double b) {
        return a + b;
    }
}
```

Abstraktion

Mit **Abstraktion** werden nur die notwendigen Details dargestellt, während komplexe Implementierungen verborgen bleiben.

Beispiel für eine abstrakte Klasse:

```
public abstract class Fahrzeug {
    public abstract void fahren();
}

public class Fahrrad extends Fahrzeug {
    @Override
    public void fahren() {
        System.out.println("Das Fahrrad wird gefahren.");
    }
}

public class Auto extends Fahrzeug {
    @Override
    public void fahren() {
        System.out.println("Das Auto wird gefahren.");
    }
}

public class Main {
    public static void main(String[] args) {
        Fahrzeug meinFahrzeug = new Fahrrad();
        meinFahrzeug.fahren();  // Das Fahrrad wird gefahren.
    }
}
```

Beispiel für ein Interface:

Ein **Interface** definiert einen Vertrag, den die implementierenden Klassen erfüllen müssen. Alle Methoden in einem Interface sind implizit abstrakt und müssen von den implementierenden Klassen überschrieben werden.

```
public interface Fahrbar {
    void fahren();
}

public class Motorrad implements Fahrbar {
    @Override
    public void fahren() {
        System.out.println("Das Motorrad fährt.");
    }
}

public class Auto implements Fahrbar {
    @Override
    public void fahren() {
        System.out.println("Das Auto fährt.");
    }
}

public class Main {
    public static void main(String[] args) {
        Fahrbar fahrzeug1 = new Motorrad();
        Fahrbar fahrzeug2 = new Auto();

        fahrzeug1.fahren();  // Das Motorrad fährt.
        fahrzeug2.fahren();  // Das Auto fährt.
    }
}
```

Assoziation

Assoziation beschreibt die Beziehung zwischen zwei Klassen. Es gibt verschiedene Formen:

* **Einfache Assoziation**: Eine Klasse kennt eine andere. * **Aggregation**: Eine “Hat-ein”-Beziehung, bei der das Teil unabhängig vom Ganzen existieren kann. * **Komposition**: Eine stärkere Form der Aggregation, bei der das Teil nicht ohne das Ganze existieren kann.

Beispiel für Assoziation:

```
public class Fahrer {
    String name;

    public Fahrer(String name) {
        this.name = name;
    }
}
```

```
public class Auto {
    String modell;
    Fahrer fahrer;  // Assoziation

    public Auto(String modell, Fahrer fahrer) {
        this.modell = modell;
        this.fahrer = fahrer;
    }
}
```

Fazit

Die objektorientierte Programmierung mit Java bietet eine strukturierte und wiederverwendbare Art der Softwareentwicklung. Durch das Verständnis von Klassen, Objekten, Kapselung, Vererbung, Polymorphismus, Abstraktion und Assoziation können komplexe Programme effektiv und effizient entwickelt werden.

Weitere Übung und praktische Beispiele helfen dabei, diese Konzepte zu vertiefen und im Berufsalltag anzuwenden.

Weiterführende Literatur

- **“Java ist auch eine Insel” von Christian Ullenboom** – Ein umfassendes Nachschlagewerk für Java.
- **“Head First Java” von Kathy Sierra und Bert Bates** – Ein praxisnahes Buch für Einsteiger: https://openbook.rheinwerk-verlag.de/javainsel/
- **Oracle Java Documentation**: https://docs.oracle.com/de/java/
- **w3schools“”: https://www.w3schools.com/java/java_oop.asp