Handout: Abstrakte Klassen in Java

Definition

Eine **abstrakte Klasse** in Java ist eine Klasse, die mit dem Schlüsselwort `abstract` deklariert wird. Sie kann Methoden enthalten, die entweder:

- **abstrakt** sind (keine Implementierung, nur Methodensignatur), oder
- **konkret** sind (mit Implementierung).

Abstrakte Klassen können nicht direkt instanziiert werden. Sie dienen als **Basisklassen**, die von anderen Klassen geerbt werden müssen.

Merkmale einer abstrakten Klasse

1. Kann sowohl **abstrakte Methoden** (ohne Implementierung) als auch **konkrete Methoden** (mit Implementierung) enthalten.
2. Kann **Konstruktoren**, **Variablen** und **statische Methoden** enthalten.
3. Vererbt gemeinsame Funktionalitäten an Unterklassen, zwingt aber die Unterklassen zur Implementierung spezifischer Methoden.
4. Kann nicht direkt instanziiert werden.

Vorteile abstrakter Klassen

- **Wiederverwendbarkeit:** Gemeinsame Logik kann in der abstrakten Klasse implementiert und von Unterklassen geerbt werden.
- **Strukturierung:** Erzeugt eine klare Struktur für die Vererbungshierarchie.
- **Flexibilität:** Unterklassen können spezifische Implementierungen der abstrakten Methoden bereitstellen.
- **Teilweise Implementierung:** Ermöglicht die Kombination aus vorgefertigtem Code und Methoden, die von Unterklassen implementiert werden müssen.

Beispiel: Abstrakte Klasse mit Autos

1. Abstrakte Klasse `Auto`

```
package com.fahrzeuge;

public abstract class Auto {
    protected String marke; // Geschützt: zugänglich in Unterklassen
    protected int baujahr;

    public Auto(String marke, int baujahr) {
        this.marke = marke;
        this.baujahr = baujahr;
```

```
    }

    // Abstrakte Methode: Muss in Unterklassen implementiert werden
    public abstract void antriebStarten();

    // Konkrete Methode: Kann direkt von Unterklassen geerbt werden
    public void anzeigen() {
        System.out.println("Marke: " + marke + ", Baujahr: " + baujahr);
    }
}
```

---

2. Konkrete Unterklassen

Klasse `ElektroAuto`

```
package com.fahrzeuge;

public class ElektroAuto extends Auto {
    private int batteriekapazitaet; // In kWh

    public ElektroAuto(String marke, int baujahr, int batteriekapazitaet) {
        super(marke, baujahr);
        this.batteriekapazitaet = batteriekapazitaet;
    }

    @Override
    public void antriebStarten() {
        System.out.println(marke + " startet den Elektromotor mit einer
Batteriekapazität von " + batteriekapazitaet + " kWh.");
    }
}
```

Klasse `VerbrennungsAuto`

```
package com.fahrzeuge;

public class VerbrennungsAuto extends Auto {
    private String kraftstofftyp; // z. B. Benzin oder Diesel

    public VerbrennungsAuto(String marke, int baujahr, String kraftstofftyp) {
        super(marke, baujahr);
        this.kraftstofftyp = kraftstofftyp;
    }

    @Override
    public void antriebStarten() {
        System.out.println(marke + " startet den Verbrennungsmotor mit " +
```

```
kraftstofftyp + ".");
    }
}
```

---

3. Hauptklasse

Datei: com/main/Main.java

```
package com.main;

import com.fahrzeuge.*;

public class Main {
    public static void main(String[] args) {
        Auto elektroAuto = new ElektroAuto("Tesla", 2022, 75);
        Auto verbrennungsAuto = new VerbrennungsAuto("BMW", 2020, "Benzin");

        elektroAuto.anzeigen();
        elektroAuto.antriebStarten();

        verbrennungsAuto.anzeigen();
        verbrennungsAuto.antriebStarten();
    }
}
```

Diagramm der Vererbungshierarchie

```
Abstrakte Klasse: Auto
    |
    +--- Konkrete Klasse: ElektroAuto
    |         - Eigenschaft: batteriekapazitaet (int)
    |         - Methode: antriebStarten() [implementiert]
    |
    +--- Konkrete Klasse: VerbrennungsAuto
              - Eigenschaft: kraftstofftyp (String)
              - Methode: antriebStarten() [implementiert]
```

Zusammenfassung

- Abstrakte Klassen sind ideal, um eine gemeinsame Basis für mehrere verwandte Klassen zu schaffen.
- **Abstrakte Methoden** definieren Verhalten, das von Unterklassen implementiert werden muss.
- **Konkrete Methoden** ermöglichen die Wiederverwendung von Code in allen Unterklassen.
- Das Beispiel mit Autos zeigt, wie Sie eine klare Struktur und Wiederverwendbarkeit in Ihrem Code erreichen können.