# Prinzipien der Programmierung

Daniel Merkle

Wintersemester 2024

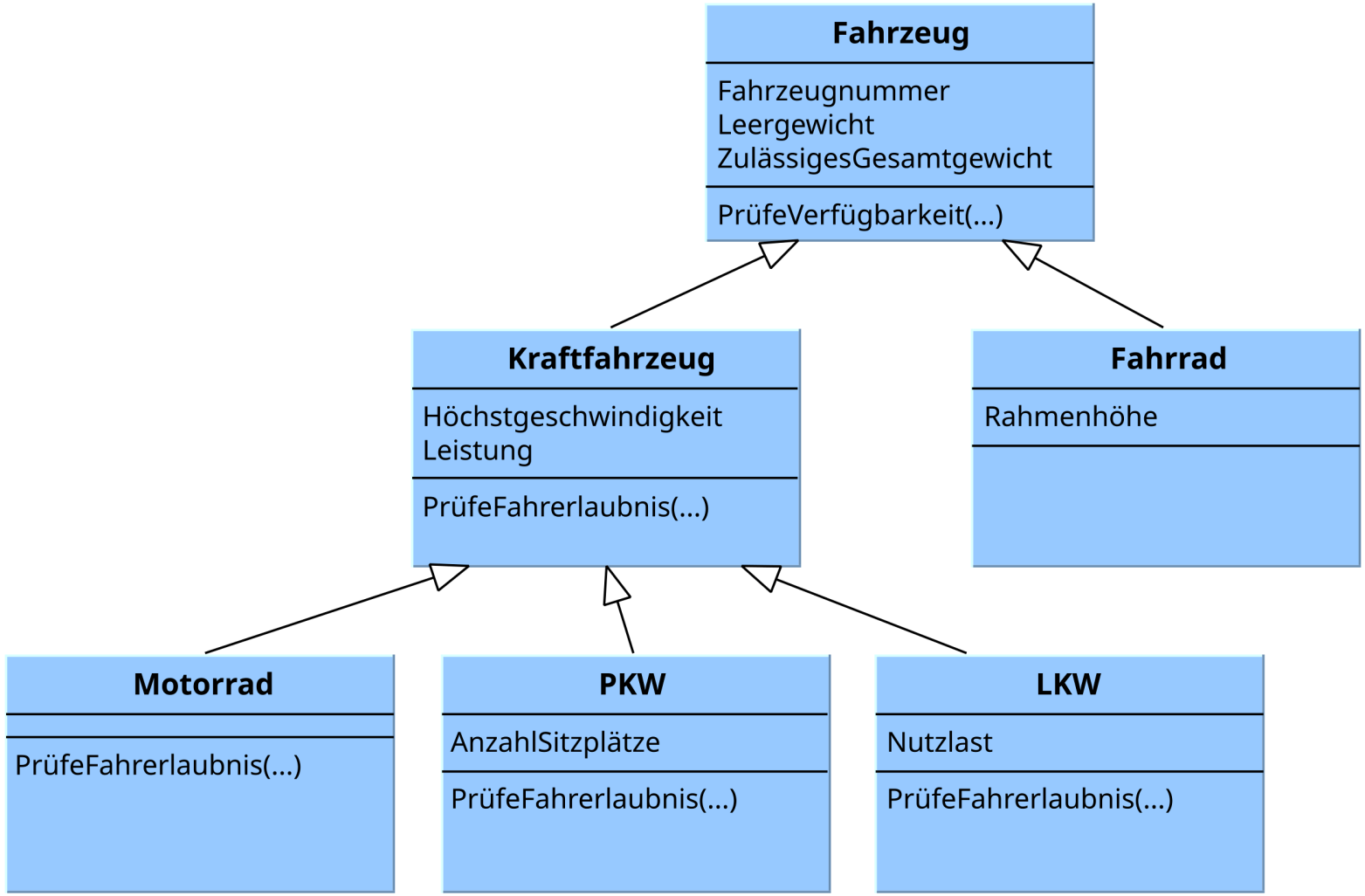
(Teil 09)

# Vererbung in Java

# Lernziele

- Sie wissen, dass in Java jede Klasse von `Object` erbt.
- Sie verstehen, warum jedes Objekt die Methoden `toString`, `equals` und `hashCode` besitzt.
- Sie können Klassen erstellen, die einige Eigenschaften von anderen Klassen erben.
- Sie kennen den Unterschied zwischen Oberklasse und Unterklasse.
- Sie wissen, wie Polymorphie funktioniert und wann Vererbung sinnvoll ist.

# Klassendiagramm (Beispiel, wikipedia)



# Beispiel: Fahrzeugverleih

Das Beispiel zeigt einen Ausschnitt aus der Unterstützung eines Fahrzeugverleihs.

- **Basisklasse `Fahrzeug`:**
  - Attribute:
    - `Fahrzeugnummer`
    - `Leergewicht`
    - `ZulässigesGesamtgewicht`
- **Spezialisierungen:**
  - `Kraftfahrzeug`
    - Zusätzliche Attribute:
      - `Höchstgeschwindigkeit`
      - `Leistung`
  - `Fahrrad`
    - Ergänzt weitere Attribute

# Vererbung von Attributen und Methoden

- **Objekte der Klasse `Kraftfahrzeug`:**
  - Enthalten alle Attribute von `Fahrzeug`
  - Zusätzlich:
    - `Höchstgeschwindigkeit`
    - `Leistung`
- **Geerbte Methoden:**
  - `PruefeVerfuegbarkeit`
    - Überprüft Verfügbarkeit anhand der `Fahrzeugnummer`
    - Wird von allen Spezialisierungen genutzt
  - Neue Methoden in `Fahrzeug` werden automatisch vererbt, z. B. `HatNavigationssystem`

## Methode `PruefeFahrerlaubnis` in `Kraftfahrzeug`

- **Zweck:**
  - Prüft, ob ein Fahrzeug mit einer bestimmten Fahrerlaubnis geführt werden darf
- **Berücksichtigt:**
  - Länderspezifische Fahrerlaubnisse
  - Betriebsland des Fahrzeugs
- **Implementierung:**
  - Basierend auf:
    - `ZulässigesGesamtgewicht`
    - `Höchstgeschwindigkeit`
    - `Leistung`
  - In `Kraftfahrzeug` teilweise möglich
  - In Spezialisierungen (`Motorrad`, `PKW`, `LKW`) oft überschrieben
    - Zusätzliche Kriterien, z. B. Anzahl der Sitzplätze

# Beispielcode: Vererbung in Java

# Basisklasse `Fahrzeug`

```java
public class Fahrzeug {
    private String fahrzeugnummer;
    private double leergewicht;
    private double zulässigesGesamtgewicht;

    public Fahrzeug(String fahrzeugnummer, double leergewicht, double zulässigesGesamtgewicht) {
        this.fahrzeugnummer = fahrzeugnummer;
        this.leergewicht = leergewicht;
        this.zulässigesGesamtgewicht = zulässigesGesamtgewicht;
    }

    public boolean pruefeVerfuegbarkeit() {
        // Beispielhafte Implementierung
        // Verfügbarkeit anhand der Fahrzeugnummer prüfen
        return true; // Hier könnte ein Datenbankzugriff erfolgen
    }

    // Getter- und Setter-Methoden
    public String getFahrzeugnummer() {
        return fahrzeugnummer;
    }

    public void setFahrzeugnummer(String fahrzeugnummer) {
        this.fahrzeugnummer = fahrzeugnummer;
    }

    public double getLeergewicht() {
        return leergewicht;
    }
```

# Spezialisierung `Kraftfahrzeug`

```java
public class Kraftfahrzeug extends Fahrzeug {
    private double höchstgeschwindigkeit;
    private double leistung;

    public Kraftfahrzeug(String fahrzeugnummer, double leergewicht, double zulässigesGesamtgewicht,
                         double höchstgeschwindigkeit, double leistung) {
        super(fahrzeugnummer, leergewicht, zulässigesGesamtgewicht);
        this.höchstgeschwindigkeit = höchstgeschwindigkeit;
        this.leistung = leistung;
    }

    public boolean pruefeFahrerlaubnis(Fahrerlaubnis fahrerlaubnis) {
        // Allgemeine Implementierung für Kraftfahrzeuge
        // Könnte länderspezifische Logik enthalten
        return true; // Platzhalter für tatsächliche Logik
    }

    // Getter- und Setter-Methoden
    public double getHöchstgeschwindigkeit() {
        return höchstgeschwindigkeit;
    }

    public void setHöchstgeschwindigkeit(double höchstgeschwindigkeit) {
        this.höchstgeschwindigkeit = höchstgeschwindigkeit;
    }

    public double getLeistung() {
        return leistung;
    }
```

# Spezialisierung `PKW` mit Methodenüberschreibung

```java
public class PKW extends Kraftfahrzeug {
    private int anzahlSitzplätze;

    public PKW(String fahrzeugnummer, double leergewicht, double zulässigesGesamtgewicht,
               double höchstgeschwindigkeit, double leistung, int anzahlSitzplätze) {
        super(fahrzeugnummer, leergewicht, zulässigesGesamtgewicht, höchstgeschwindigkeit, leistung);
        this.anzahlSitzplätze = anzahlSitzplätze;
    }

    @Override
    public boolean pruefeFahrerlaubnis(Fahrerlaubnis fahrerlaubnis) {
        // Spezifische Implementierung für PKW
        // Berücksichtigt z.B. die Anzahl der Sitzplätze und die Fahrerlaubnisklasse
        if ("B".equals(fahrerlaubnis.getKlasse()) && anzahlSitzplätze <= 9) {
            return true;
        } else {
            return false;
        }
    }

    // Getter- und Setter-Methoden
    public int getAnzahlSitzplätze() {
        return anzahlSitzplätze;
    }

    public void setAnzahlSitzplätze(int anzahlSitzplätze) {
        this.anzahlSitzplätze = anzahlSitzplätze;
    }
}
```

# Klasse `Fahrerlaubnis` (sehr vereinfacht)

```java
public class Fahrerlaubnis {
    private String land;
    private String klasse;

    public Fahrerlaubnis(String land, String klasse) {
        this.land = land;
        this.klasse = klasse;
    }

    // Getter- und Setter-Methoden
    public String getLand() {
        return land;
    }

    public void setLand(String land) {
        this.land = land;
    }

    public String getKlasse() {
        return klasse;
    }

    public void setKlasse(String klasse) {
        this.klasse = klasse;
    }
}
```

# Anwendung im Programm

```java
public class Main {
    public static void main(String[] args) {
        // Instanziierung eines PKW-Objekts
        PKW pkw = new PKW("ABC123", 1500, 2000, 220, 110, 5);

        // Erzeugung eines Fahrerlaubnis-Objekts
        Fahrerlaubnis fahrerlaubnis = new Fahrerlaubnis("DE", "B");

        // Prüfung der Verfügbarkeit
        if (pkw.pruefeVerfuegbarkeit()) {
            // Prüfung der Fahrerlaubnis
            boolean erlaubt = pkw.pruefeFahrerlaubnis(fahrerlaubnis);
            if (erlaubt) {
                System.out.println("Fahrerlaubnis ist gültig für dieses Fahrzeug.");
            } else {
                System.out.println("Fahrerlaubnis ist nicht gültig für dieses Fahrzeug.");
            }
        } else {
            System.out.println("Fahrzeug ist nicht verfügbar.");
        }
    }
}
```

## Zusammenfassung

- **Vererbung** ermöglicht die Wiederverwendung von Attributen und Methoden.
- **Basisklasse** `Fahrzeug` definiert gemeinsame Eigenschaften.
- **Unterklassen** wie `Kraftfahrzeug` und `PKW` erweitern oder spezialisieren die Funktionalität.
- **Methodenüberschreibung** erlaubt es Unterklassen, spezifische Implementierungen bereitzustellen.
- **Polymorphismus**: Methodenaufrufe richten sich nach dem tatsächlichen Objekttyp.

# Hinweise zum Code

- **Basisklasse** `Fahrzeug` enthält grundlegende Attribute und die Methode `pruefeVerfuegbarkeit()`, die von allen Unterklassen geerbt wird.
- `Kraftfahrzeug` erweitert `Fahrzeug` um Attribute wie `höchstgeschwindigkeit` und `leistung` sowie die Methode `pruefeFahrerlaubnis()`.
- `PKW` überschreibt die Methode `pruefeFahrerlaubnis()`, um spezifische Anforderungen zu berücksichtigen (z. B. Anzahl der Sitzplätze).
- `Fahrerlaubnis` ist eine separate Klasse, die Informationen über die Fahrerlaubnis enthält.
- Illustriert die Konzepte der **Vererbung**, **Methodenüberschreibung** und des **Polymorphismus** anhand eines praktischen Beispiels aus dem Bereich Fahrzeugverleih.

# Back to square one

# Was ist Vererbung?

Vererbung ermöglicht es, eine neue Klasse zu erstellen, die die Eigenschaften einer bestehenden Klasse erbt.

```
class SubClass extends SuperClass {
    // neue Eigenschaften und Methoden
}
```

- Oberklasse (SuperClass): Die ursprüngliche Klasse
- Unterklasse (SubClass): Die Klasse, die erbt

## Vererbung in Java: Das "extends"-Schlüsselwort

- **Vererbung** ermöglicht es einer Klasse, die Eigenschaften (Methoden und Variablen) einer anderen Klasse zu übernehmen.
- Die Klasse, die erbt, wird als **Unterklasse** bezeichnet.
- Die Klasse, von der geerbt wird, wird als **Oberklasse** bezeichnet.
- Verwenden Sie das Schlüsselwort `extends`, um die Vererbung zu implementieren.

```
public class MyClass extends SuperClass {
    // MyClass erbt alle Methoden und Variablen von SuperClass
}
```

## Vererbungshierarchie von `ArrayList`

- Jede Klasse in Java erbt letztlich von `Object`.
- Beispiel: Die Vererbungshierarchie von `ArrayList`:

```
java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.ArrayList<E>
```

- Jede Klasse erbt die Eigenschaften aller übergeordneten Klassen in der Hierarchie.

## Erben von `AbstractList` und `AbstractCollection`

- `ArrayList` erbt von `AbstractList`, welches wiederum von `AbstractCollection` erbt.
- Dadurch hat `ArrayList` Zugriff auf Methoden und Variablen aus **allen** übergeordneten Klassen:

```
ArrayList<String> list = new ArrayList<>();
list.add("Hello");
list.isEmpty();   // Methode aus AbstractCollection
list.size();      // Methode aus AbstractList
list.toString();  // Methode aus Object
```

## Vererbung von `Object`

- **Jede Klasse in Java** erbt implizit von der Klasse `Object`.
- Dies bedeutet, dass alle Klassen Zugriff auf Methoden wie `toString()`, `equals()`, und `hashCode()` haben, die in der `Object`-Klasse definiert sind:

```
ArrayList<String> list = new ArrayList<>();
System.out.println(list.toString());  // Methode aus Object
System.out.println(list.equals(new ArrayList<>()));  // equals-Methode
```

```
public class MyClass {
    // MyClass erbt automatisch von Object
}
```

## Direktes und Indirektes Erben

- Eine Klasse kann (in Java) nur **direkt** von **einer** anderen Klasse erben (Single Inheritance).
- Jedoch erbt eine Klasse **indirekt** alle Eigenschaften ihrer übergeordneten Klassen in der Vererbungshierarchie.

```
public class MyList extends AbstractList<String> {
    // MyList erbt direkt von AbstractList
    // und indirekt von AbstractCollection und Object
}
```

## Vorteile der Vererbung

- **Wiederverwendbarkeit**: Gemeinsam genutzter Code muss nicht neu geschrieben werden.
- **Erweiterbarkeit**: Neue Klassen können durch Vererbung erweitert werden.
- **Konsistenz**: Alle Klassen in Java haben Zugriff auf grundlegende Methoden wie `toString()` und `equals()` durch die Vererbung von `Object`.

# Beispiel: Autoteile

- Beispiel eines Automobilproduktionssystems
- `Part` als Oberklasse

```
public class Part {
    private String identifier;
    private String manufacturer;
    private String description;

    // Konstruktor und Methoden
}
```

# Beispiel: Motor erbt von Part

```
public class Engine extends Part {
    private String engineType;

    public Engine(String engineType, String identifier, String manufacturer, String description) {
        super(identifier, manufacturer, description);
        this.engineType = engineType;
    }

    // Methoden
}
```

- `Engine` erbt alle Eigenschaften von `Part`.
- Zusätzliche Eigenschaften, z.B. der Motortyp, werden ergänzt.

# Konstruktoren und `super`

- Der Konstruktor der Unterklasse kann den Konstruktor der Oberklasse aufrufen.
- Dies erfolgt mit dem Schlüsselwort `super`.

```
public Engine(String engineType, String identifier, String manufacturer, String description) {
    super(identifier, manufacturer, description);
    this.engineType = engineType;
}
```

# Polymorphie in Java

- Polymorphie bedeutet unter anderem, dass die tatsächliche Methode zur Laufzeit bestimmt wird.
- Beispiel:

```
Part part = new Engine("combustion", "hz", "volkswagen", "VW GOLF 1L 86-91");
System.out.println(part.getIdentifier());  // Aufruf der Methode aus Part
```

# Polymorphie in Java

# Was ist Polymorphie?

- **Polymorphie** bedeutet "Vielgestaltigkeit".
- Ermöglicht, dass eine Oberklassen-Referenz auf Unterklassen-Objekte verweist.
- Methodenaufrufe können zur **Laufzeit (Runtime)** anhand des Objekttyps aufgelöst werden.
- Überladen zur **Kompilierzeit (Compile-Time)** wird manchmal mit **ad-hoc Polymorphie** gleichgesetzt.
- ein wenig genauer...

## Arten der Polymorphie

- **Methodenüberladung** (Compile-Time)
  - Gleicher Methodenname, unterschiedliche Parameter.
- **Methodenüberschreibung** (Runtime)
  - Unterklasse überschreibt Methode der Oberklasse.
- Man beachte: Polymorphie ist ein sehr weitreichendes Thema, es gibt viele Arten und Unterteilungen. Wir halten es einfach.

# Beispiel: Methodenüberschreibung

```
class Fahrzeug {
    void starte() {
        System.out.println("Fahrzeug startet");
    }
}

class PKW extends Fahrzeug {
    @Override
    void starte() {
        System.out.println("PKW startet");
    }
}
```

# Polymorpher Methodenaufruf

```
Fahrzeug meinFahrzeug = new PKW();
meinFahrzeug.starte(); // Ausgabe: "PKW startet"
```

- Obwohl `meinFahrzeug` vom Typ `Fahrzeug` ist, wird `PKW.starte()` aufgerufen.

# Virtuelle Methoden in Java

- **Virtuelle Methoden** sind Methoden, deren Implementierung zur **Laufzeit** anhand des Objekttyps aufgelöst wird.
- Java verwendet standardmäßig **dynamisches Binden** (Late Binding) für alle **nicht-statischen Methoden**.
- Ziel: Polymorphie ermöglichen und flexibles Verhalten je nach Objekttyp bereitstellen.

```java
class Fahrzeug {
    void starte() {
        System.out.println("Fahrzeug startet");
    }
}

class PKW extends Fahrzeug {
    @Override
    void starte() {
        System.out.println("PKW startet");
    }
}

public class Main {
    public static void main(String[] args) {
        Fahrzeug meinFahrzeug = new PKW();
        meinFahrzeug.starte(); // Ausgabe: "PKW startet"
    }
}
```

# Dynamisches Binden: Was ist das?

- **Dynamisches/Spätes Binden** (Late Binding):
  - Die Methode wird basierend auf dem **tatsächlichen Objekttyp** zur **Laufzeit** aufgerufen.
  - Standardverhalten für alle **nicht-statischen Methoden** in Java.
- **Vorteile**:
  - **Polymorphie**: Erlaubt die Verwendung eines Basistyps für verschiedene Objekttypen.
  - **Flexibilität**: Subklassen können Verhalten überschreiben, ohne dass der Code, der die Basisklasse verwendet, geändert werden muss.

# Dynamisches Binden: Beispiel (nochmal)

```
class Fahrzeug {
    void starte() {
        System.out.println("Fahrzeug startet");
    }
}

class PKW extends Fahrzeug {
    @Override
    void starte() {
        System.out.println("PKW startet");
    }
}

public class Main {
    public static void main(String[] args) {
        Fahrzeug meinFahrzeug = new PKW();
        meinFahrzeug.starte(); // Ausgabe: "PKW startet"
    }
}
```

# Spätes / Dynamisches Binden: Erklärung des Beispiels

- **Referenztyp**: `Fahrzeug`
  - Die Referenz `meinFahrzeug` ist vom Typ der Basisklasse `Fahrzeug`.
- **Objekttyp**: `PKW`
  - Das tatsächliche Objekt, auf das `meinFahrzeug` zeigt, ist ein `PKW`.
- **Laufzeitentscheidung**:
  - Beim Aufruf von `meinFahrzeug.starte()` entscheidet die JVM zur **Laufzeit**, dass die Methode `starte()` von `PKW` ausgeführt wird.
- **Ergebnis**: Ausgabe: `"PKW startet"`

## Vorteile der Polymorphie

- **Flexibilität** im Code
- **Erweiterbarkeit** durch einfache Integration neuer Klassen
- **Wartbarkeit** durch Nutzung allgemeiner Schnittstellen

# Zusammenfassung

- Polymorphie ermöglicht dynamisches Verhalten in Java.
- Kernkonzept der objektorientierten Programmierung.
- Erreicht durch Vererbung und Methodenüberschreibung.

# Zugriffsmodifikatoren

- `private` : Nur innerhalb der Klasse sichtbar
- `protected` : Für Unterklassen und innerhalb des Pakets sichtbar
- `public` : Überall sichtbar

```
protected String identifier;
```

# Wann ist Vererbung sinnvoll?

- Vererbung ist sinnvoll, wenn die Unterklasse eine Spezialisierung der Oberklasse ist.
- Beispiel: Ein Motor ist ein spezielles Teil.
- Vererbung sollte nicht verwendet werden, wenn Objekte nur eine lose Beziehung haben.

```
class Car {
    private Engine engine;  // Ein Auto hat einen Motor, erbt aber nicht von Engine.
}
```

# Programmierübung: ABC

- Erstellen Sie die Klassen `A`, `B`, `C`, wobei jede Klasse von der vorherigen erbt.
- Jede Klasse sollte eine Methode haben, die einen Buchstaben ausgibt.
- Beispiel:

```
A a = new A();
B b = new B();
C c = new C();

a.a();
b.b();
c.c();
```

```
A
B
C
```

# Falsche Verwendung von Vererbung

# Single Responsibility Principle

- Jede Klasse sollte nur einen Grund zur Änderung haben.
- Missachten dieses Prinzips führt zu schwer wartbarem Code.
- Vererbung darf keine zusätzlichen Verantwortlichkeiten hinzufügen.

Single Responsibility Principle

# Falsche Vererbung: Customer und Order

```
public class Customer {
    private String name;
    private String address;

    public Customer(String name, String address) {
        this.name = name;
        this.address = address;
    }

    public String getName() {
        return name;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }
}
```

# Falsche Vererbung: Customer und Order

```
public class Order extends Customer {
    private String product;
    private String count;

    public Order(String product, String count, String name, String address) {
        super(name, address);
        this.product = product;
        this.count = count;
    }

    public String getProduct() {
        return product;
    }

    public String getCount() {
        return count;
    }

    public String postalAddress() {
        return this.getName() + "\n" + this.getAddress();
    }
}
```

- **Problem**: `Order` erbt von `Customer`, was semantisch falsch ist.
- Eine Bestellung ist kein spezieller Fall eines Kunden, daher sollte keine Vererbung verwendet werden.

# Warum ist dies problematisch?

- **Verletzung des Single Responsibility Prinzips**:
  - `Order` sollte nur für Bestelldaten verantwortlich sein, ist aber auch für Kundendaten zuständig.
- **Wartungsprobleme**:
  - Wenn sich die Adresse eines Kunden ändert, müssten alle Bestellungen, die diesen Kunden betreffen, manuell aktualisiert werden.

```
Order order = new Order("Laptop", "2", "Alice", "Main St 123");
order.setAddress("New Address 456");  // Muss in allen Bestellungen geändert werden
```

- Der Code wird unübersichtlich und schwer wartbar.

# Bessere Lösung: Kapselung

```
public class Order {
    private Customer customer;
    private String product;
    private String count;

    public Order(Customer customer, String product, String count) {
        this.customer = customer;
        this.product = product;
        this.count = count;
    }

    public String getProduct() {
        return product;
    }

    public String getCount() {
        return count;
    }

    public String postalAddress() {
        return this.customer.getName() + "\n" + this.customer.getAddress();
    }
}
```

- **Vorteil**: Änderungen an den Kundendaten betreffen nur das `Customer`-Objekt.
- Das `Order`-Objekt kapselt nur die Bestellinformationen.

```
Customer customer = new Customer("Alice", "Main St 123");
Order order = new Order(customer, "Laptop", "2");
customer.setAddress("New Address 456");
System.out.println(order.postalAddress());
```

# Fazit: Vererbung vs. Kapselung

- Verwenden Sie Vererbung, wenn eine "ist-ein"-Beziehung besteht:
  - **Ein Motor ist ein Teil**.
- Verwenden Sie Kapselung, wenn eine "hat-ein"-Beziehung besteht:
  - **Eine Bestellung hat einen Kunden**.

```
public class Car {
    private Engine engine;  // Ein Auto hat einen Motor
}
```

- Vermeiden Sie es, Vererbung für Beziehungen zu verwenden, die keine klare "ist-ein"-Beziehung darstellen.

# Zusammenfassung: Vererbung richtig verwenden

- Vererbung sollte sparsam und bewusst eingesetzt werden.
- Kapselung ist oft eine bessere Alternative, wenn die Beziehung zwischen Klassen keine klare "ist-ein"-Beziehung ist.
- Halten Sie sich an das Single Responsibility Principle, um sauberen und wartbaren Code zu schreiben.

# Single Responsibility Principle (SRP)

- **Definition (korrekt)**:
  „Es sollte nie mehr als einen Grund geben, eine Klasse zu ändern.“
  – Robert C. Martin
- Das Prinzip bezieht sich auf **Gründe für Änderungen**, nicht auf die **Aufgaben** einer Klasse.
- Ziel: Klassen sollten eine **klare Verantwortung** haben, um Wartbarkeit und Modularität zu verbessern.

# Fehlerhafte Interpretation des SRP

- **Falsche Annahme**:
  "Eine Klasse sollte genau eine fest definierte Aufgabe erfüllen."
- **Problem**:
  - Führt zu einer übermäßigen Aufteilung von Klassen.
  - Der Code wird fragmentiert und schwer wartbar.

**Beispiel (fehlerhaft):**

```
class UserManager {
    void createUser(String username) {
        log("User created: " + username);
        notifyAdmin("New user created: " + username);
    }

    void log(String message) { ... }
    void notifyAdmin(String message) { ... }
}
```

- Falsche Anwendung von SRP: Jede Methode könnte als "eigene Aufgabe" gesehen werden, was dazu führen könnte, dass die Klasse in viele kleine Klassen aufgeteilt wird.
- Ergebnis: UserManager, Logger, AdminNotifier sind zu klein und schwer wartbar, da ihre Aufgaben stark zusammenhängen.
- Oben (ohne Änderung): Logik und Benachrichtigungen sind in der gleichen Klasse enthalten.

# Korrekte Interpretation des SRP

- **Korrekte Aussage**:
  "Es sollte nie mehr als einen Grund geben, eine Klasse zu ändern."
- Fokus: **Gründe für Änderungen**, nicht **Aufgaben**.
- Jede Klasse sollte eine einzige Verantwortlichkeit haben, die unabhängig änderbar ist.

**Beispiel (korrekt):**

```
class UserManager {
    private Logger logger;
    private AdminNotifier notifier;

    void createUser(String username) {
        logger.log("User created: " + username);
        notifier.notify("New user created: " + username);
    }
}

class Logger { void log(String message) { ... } }
class AdminNotifier { void notify(String message) { ... } }
```

# Gründe für Änderungen im SRP

- Ein **Grund für eine Änderung** entspricht einer spezifischen **Verantwortlichkeit** oder einem **Aspekt**, der sich ändern könnte.
- Jede Verantwortlichkeit sollte in einer eigenen Klasse gekapselt werden.
- Ziel: Änderungen in einer Klasse sollen **keine Auswirkungen** auf andere Klassen haben.

# Ursprüngliches Beispiel: Gründe für Änderungen

### Ursprünglicher Code:

```
class UserManager {
    void createUser(String username) {
        System.out.println("User created: " + username);
        log("User created: " + username);
        notifyAdmin("New user created: " + username);
    }

    void log(String message) { ... }
    void notifyAdmin(String message) { ... }
}
```

### Mögliche Gründe für Änderungen:

1. **Änderungen an der Benutzerverwaltung**:
    - z. B. Hinzufügen von Validierung oder Datenbankintegration.
2. **Änderungen am Logging**:
    - Wechsel von `System.out` zu einem Framework wie Log4j.
3. **Änderungen an der Benachrichtigungslogik**:
    - Senden einer E-Mail statt einer Konsolennachricht.

## Verbesserter Code:

```
class UserManager {
    private final Logger logger;
    private final AdminNotifier notifier;

    public UserManager(Logger logger, AdminNotifier notifier) {
        this.logger = logger;
        this.notifier = notifier;
    }

    public void createUser(String username) {
        System.out.println("User created: " + username);
        logger.log("User created: " + username);
        notifier.notify("New user created: " + username);
    }
}

class Logger {
    void log(String message) {
        System.out.println("Log: " + message);
    }
}

class AdminNotifier {
    void notify(String message) {
        System.out.println("Notify Admin: " + message);
    }
}
```

- Jede Klasse hat eine **klar definierte Verantwortung**:
  - `UserManager`: Benutzerverwaltung.
  - `Logger`: Logging.
  - `AdminNotifier`: Benachrichtigung.

# Beispiel: Grund für Änderungen - Logging

- **Problem**: Umstellung von `System.out` auf ein Logging-Framework wie Log4j.
- Änderung betrifft nur die `Logger`-Klasse:

```
class Logger {
    void log(String message) {
        // Neue Implementierung mit Log4j
        System.out.println("Log: " + message); // Ersetzen durch Framework-Call
    }
}
```

**Vorteile:**

- **Modularität**: Nur die `Logger`-Klasse muss angepasst werden.
- **Keine Auswirkungen**: `UserManager` und `AdminNotifier` bleiben unverändert.

- Ein **Grund für eine Änderung** entspricht einer Verantwortlichkeit.
- **SRP korrekt angewandt**:
  - Jede Klasse ist **modular** und hat nur **einen Grund für Änderungen**.
  - Änderungen in einem Bereich (z. B. Logging) beeinflussen andere Bereiche nicht.

# Vererbung und Verletzung des SRP

- **Problem**: Vererbung kann das **Single Responsibility Principle (SRP)** verletzen, wenn:
    i. Eine Unterklasse Methoden oder Verantwortlichkeiten erbt, die nicht zu ihrer spezifischen Aufgabe gehören.
    ii. Die Unterklasse dadurch unnötig anfällig für Änderungen wird.
- **Beispiel**: Eine Unterklasse erbt Funktionen, die sie nicht benötigt.

# Beispiel: Unpassende Verantwortlichkeiten

**Code:**

```
class Document {
    void print() {
        System.out.println("Printing document...");
    }

    void saveToDatabase() {
        System.out.println("Saving document to database...");
    }
}

class PDFDocument extends Document {
    // PDF-Dokumente sollen nicht in der Datenbank gespeichert werden
}
```

- **Problem**:
  - Wir nehmen an PDF-Dokumente sollen nicht in der Datenbank gespeichert werden. `PDFDocument` erbt die Methode `saveToDatabase()`, obwohl das Speichern in der Datenbank für PDF-Dokumente irrelevant ist.
  - Änderungen an `saveToDatabase()` betreffen `PDFDocument`, obwohl diese Methode nicht genutzt wird.

# Einführung in Abstrakte Klassen

# Was ist eine abstrakte Klasse?

- Eine abstrakte Klasse dient als Grundlage für andere Klassen.
- Sie kann normale Methoden enthalten, aber auch **abstrakte Methoden**.
- **Wichtig**: Sie können keine Instanz einer abstrakten Klasse erstellen!

```
public abstract class Animal {
    public abstract void makeSound();
}
```

- In diesem Beispiel ist `Animal` eine abstrakte Klasse.
- Die Methode `makeSound()` ist abstrakt – sie hat keine Implementierung.

# Warum abstrakte Klassen?

- Abstrakte Klassen definieren ein **Konzept**, das von mehreren Klassen geteilt wird.
- Sie erlauben es uns, **gemeinsame Funktionalität** zu definieren, ohne vollständige Implementierungen bereitzustellen.

```
public abstract class Shape {
    public abstract double area();
    public abstract double perimeter();
}
```

- **Beispiel**: Eine Form ( `Shape` ) ist ein Konzept, aber keine spezifische Form.
- Subklassen wie `Circle` oder `Rectangle` erben von `Shape` .

# Ein einfaches Beispiel: Tiere

```
public abstract class Animal {
    public abstract void makeSound();

    public void sleep() {
        System.out.println("Zzzz...");
    }
}
```

- `Animal` ist eine abstrakte Klasse.
- Die Methode `makeSound()` wird von den Unterklassen implementiert.
- `sleep()` ist eine normale Methode, die für alle Tiere gilt.

# Unterklassen von Animal

```java
public class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Bark");
    }
}
```

```java
public class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow");
    }
}
```

- `Dog` und `Cat` erben von `Animal` und implementieren `makeSound()`.
- Sie haben jeweils ihre eigene Version der Methode.

```java
Animal dog = new Dog();
Animal cat = new Cat();
dog.makeSound();  // Ausgabe: Bark
cat.makeSound();  // Ausgabe: Meow
```

# Erweiterung des Beispiels: Form (Shape)

```
public abstract class Shape {
    public abstract double area();
    public abstract double perimeter();
}
```

- `Shape` ist eine abstrakte Klasse, die die Berechnung von Fläche und Umfang erzwingt.

```
public class Circle extends Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double area() {
        return Math.PI * radius * radius;
    }

    @Override
    public double perimeter() {
        return 2 * Math.PI * radius;
    }
}
```

# Erweiterung des Beispiels: Form (Shape)

- `Circle` implementiert die abstrakten Methoden `area()` und `perimeter()`.

```
public class Rectangle extends Shape {
    private double width;
    private double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public double area() {
        return width * height;
    }

    @Override
    public double perimeter() {
        return 2 * (width + height);
    }
}
```

- `Rectangle` implementiert ebenfalls die abstrakten Methoden `area()` und `perimeter()`.

# Nutzung von Shape

```
Shape circle = new Circle(5);
Shape rectangle = new Rectangle(4, 6);

System.out.println("Circle area: " + circle.area());
System.out.println("Rectangle area: " + rectangle.area());
```

```
Circle area: 78.54
Rectangle area: 24
```

- **Wichtig**: Wir können keine Instanz von `Shape` erstellen, aber von `Circle` und `Rectangle`, da diese die abstrakten Methoden implementieren.

# Zusammenfassung: Abstrakte Klassen

- Abstrakte Klassen definieren **gemeinsame Funktionalität**.
- Sie enthalten abstrakte Methoden, die von den Unterklassen implementiert werden müssen.
- Man kann keine Instanzen von abstrakten Klassen erstellen – nur von den konkreten Unterklassen.

```
Animal animal = new Animal();  // Nicht erlaubt!
Animal dog = new Dog();        // Erlaubt
```

- Abstrakte Klassen helfen dabei, **gemeinsame Strukturen** zu definieren und die Implementierung der Details den Unterklassen zu überlassen.

# Programmierübung: Warehouse mit Historie

- **Warehouse** speichert Produktmengen.
- **ProductWarehouse** erbt von Warehouse und fügt einen Produktnamen hinzu.
- **ProductWarehouseWithHistory** erweitert dies mit einer Änderungshistorie.

```
ProductWarehouseWithHistory juice = new ProductWarehouseWithHistory("Juice", 1000.0, 1000.0);
juice.addToWarehouse(50.0);
System.out.println(juice.history());
```

[1000.0, 1050.0]

# Zusammenfassung: Vererbung und Abstrakte Klassen

- Vererbung sollte mit Bedacht eingesetzt werden, um das Single Responsibility Principle nicht zu verletzen.
- Kapselung bei "hat-ein" Beziehungen ("hat-ein" vs. "ist-ein").
- Abstrakte Klassen ermöglichen gemeinsame Funktionalität und flexible Erweiterbarkeit.
- Vererbung ermöglicht es, Code wiederzuverwenden und Objekte zu spezialisieren.
- Das Schlüsselwort `super` wird verwendet, um Konstruktoren und Methoden der Oberklasse aufzurufen.
- Polymorphie ermöglicht es, dass die tatsächliche Methode zur Laufzeit bestimmt wird.

# Interfaces in Java

# Lernziele

- Verstehen des Konzepts von Interfaces und Implementieren eigener Interfaces in Klassen.
- Nutzung von Interfaces als Variablentypen, Methodenparameter und Methodenrückgabewerte.
- Kennenlernen von Java-bereitgestellten Interfaces.

# Was ist ein Interface?

- Ein Interface definiert **Verhalten** durch Methodennamen und Rückgabewerte.
- Es enthält keine Implementierung der Methoden, nur deren Signaturen.
- Ein Interface wird durch das Schlüsselwort `interface` definiert.

```
public interface Readable {
    String read();
}
```

- Beispiel: Das Interface `Readable` definiert die Methode `read()`, die einen `String` zurückgibt.

# Wie wird ein Interface implementiert?

- Eine Klasse implementiert ein Interface mit dem Schlüsselwort `implements`.
- Die Klasse **muss** alle Methoden des Interfaces implementieren.
- Beispiel:

```
public class TextMessage implements Readable {
    private String sender;
    private String content;

    public TextMessage(String sender, String content) {
        this.sender = sender;
        this.content = content;
    }

    public String read() {
        return this.content;
    }
}
```

# Ein Interface ist ein Vertrag

- Eine Klasse, die ein Interface implementiert, **unterzeichnet einen Vertrag**.
- Dieser Vertrag verpflichtet zur Implementierung aller Methoden des Interfaces.
- Das Interface legt nur Methodennamen, Parameter und Rückgabewerte fest – die Implementierung bleibt der Klasse überlassen.

# Mehrere Klassen können ein Interface implementieren

- Beispiel: `Ebook` implementiert ebenfalls das `Readable`-Interface.

```
public class Ebook implements Readable {
    private ArrayList<String> pages;
    private int currentPage;

    public Ebook(ArrayList<String> pages) {
        this.pages = pages;
        this.currentPage = 0;
    }

    public String read() {
        String page = pages.get(currentPage);
        currentPage = (currentPage + 1) % pages.size();
        return page;
    }
}
```

# Verwendung von Interfaces als Variablentyp

- Ein Interface kann auch als Typ verwendet werden, z.B. als Variablentyp oder Rückgabewert einer Methode.
- Beispiel:

```
Readable message = new TextMessage("Anna", "Hallo!");
Readable ebook = new Ebook(pages);
```

- **Vorteil**: Eine Variable des Typs `Readable` kann jedes Objekt speichern, das dieses Interface implementiert.

# Interface als Methodenparameter

- Interfaces sind besonders nützlich als Methodentypen.
- Beispiel: Eine Methode, die ein `Readable`-Objekt druckt.

```
public class Printer {
    public void print(Readable readable) {
        System.out.println(readable.read());
    }
}
```

```
Printer printer = new Printer();
printer.print(new TextMessage("Anna", "Hallo!"));
printer.print(new Ebook(pages));
```

## Beispiel: Interface als Methodenparameter

```java
public interface Packable {
    double weight();
}
```

```java
public class Book implements Packable {
    private String title;
    private double weight;

    public Book(String title, double weight) {
        this.title = title;
        this.weight = weight;
    }

    @Override
    public double weight() {
        return this.weight;
    }
}
```

## Methodenparameter mit Packable

```
public class Packer {
    public void pack(Packable item) {
        System.out.println("Packing item with weight: " + item.weight());
    }
}
```

- Die Methode `pack` akzeptiert jedes Objekt, das das `Packable`-Interface implementiert.
- Dies ermöglicht es, unterschiedliche Klassen, die `Packable` implementieren, als Parameter zu verwenden.

## Verwendung der Methode

```
Book book = new Book("Clean Code", 1.2);
Packer packer = new Packer();

packer.pack(book);  // Packing item with weight: 1.2
```

- `Book` implementiert `Packable`, daher kann es an die Methode `pack` übergeben werden.
- Durch die Verwendung eines Interfaces als Parameter kann dieselbe Methode mit verschiedenen Implementierungen von `Packable` verwendet werden.

## Vorteile der Interface-Verwendung

- **Flexibilität:** Jede Klasse, die `Packable` implementiert, kann der Methode übergeben werden.
- **Lose Kopplung:** Methoden müssen keine konkreten Klassen kennen.
- **Erweiterbarkeit:** Neue Implementierungen von `Packable` können hinzugefügt werden, ohne dass der Code der Methode geändert werden muss.

# Programmierübung: TacoBoxes

- Implementiere die Klassen `TripleTacoBox` und `CustomTacoBox`, die beide das Interface `TacoBox` implementieren.

```java
public interface TacoBox {
    int tacosRemaining();
    void eat();
}
```

- `TripleTacoBox` startet mit 3 Tacos, `CustomTacoBox` hat eine benutzerdefinierte Anzahl von Tacos.
- Methoden: `tacosRemaining()` und `eat()`.

## Verwendung von `Interface` als Rückgabewert

- Im folgenden Beispiel wird das `Packable`-Interface als Rückgabewert verwendet.
- Die Klasse `Factory` erstellt verschiedene Objekte, die das `Packable`-Interface implementieren.

# Übung (aus 09-06)

Erstellen Sie Klassen `Book` und `CD`, die das `Packable` Interface implementieren.

```
public static void main(String[] args) {
    Book book1 = new Book("Fyodor Dostoevsky", "Crime and Punishment", 2);
    Book book2 = new Book("Robert Martin", "Clean Code", 1);
    Book book3 = new Book("Kent Beck", "Test Driven Development", 0.5);

    CD cd1 = new CD("Pink Floyd", "Dark Side of the Moon", 1973);
    CD cd2 = new CD("Wigwam", "Nuclear Nightclub", 1975);
    CD cd3 = new CD("Rendezvous Park", "Closer to Being Here", 2012);

    System.out.println(book1);
    System.out.println(book2);
    System.out.println(book3);
    System.out.println(cd1);
    System.out.println(cd2);
    System.out.println(cd3);
}
```

## Die `Factory`-Klasse

- Die `Factory`-Klasse stellt eine Methode zur Verfügung, die ein `Packable`-Objekt zurückgibt.
- Abhängig von einer Zufallszahl erzeugt die Methode ein neues `Book` oder eine `CD`.

```
import java.util.Random;

public class Factory {
    public Packable produceNew() {
        Random random = new Random();
        int number = random.nextInt(4);

        if (number == 0) {
            return new Book("Fyodor Dostoevsky", "Crime and Punishment", 2.0);
        } else if (number == 1) {
            return new CD("Pink Floyd", "Dark Side of the Moon", 1973);
        } else if (number == 2) {
            return new Book("Robert Martin", "Clean Code", 1.5);
        } else {
            return new CD("The Beatles", "Abbey Road", 1969);
        }
    }
}
```

## Beispiel für die Verwendung der `Factory`

- Die Methode `produceNew` gibt immer ein `Packable`-Objekt zurück.
- Dies ermöglicht es, verschiedene Typen von `Packable`-Objekten flexibel zu erzeugen.

```
public class Main {
    public static void main(String[] args) {
        Factory factory = new Factory();

        for (int i = 0; i < 5; i++) {
            Packable item = factory.produceNew();
            System.out.println("Created item: " + item);
            System.out.println("Item weight: " + item.weight() + " kg");
        }
    }
}
```

## Ausgabe des Factory-Beispiels

- Beispielausgabe für die Erstellung von `Packable`-Objekten mit der Factory:

```
Created item: Dark Side of the Moon by Pink Floyd (1973)
Item weight: 0.1 kg
Created item: Crime and Punishment by Fyodor Dostoevsky
Item weight: 2.0 kg
Created item: Clean Code by Robert Martin
Item weight: 1.5 kg
Created item: Abbey Road by The Beatles (1969)
Item weight: 0.1 kg
Created item: Crime and Punishment by Fyodor Dostoevsky
Item weight: 2.0 kg
```

## Vorteile von Interfaces als Rückgabewert

- **Flexibilität**: Die Methode kann unterschiedliche Objekte zurückgeben, solange sie das gleiche Interface implementieren.
- **Erweiterbarkeit**: Neue Klassen können später hinzugefügt werden, ohne den bestehenden Code zu verändern.
- **Polymorphismus**: Der Benutzer muss nicht wissen, welche konkrete Klasse zurückgegeben wird – sie arbeitet mit dem Interface-Typ.

# Abstrakte Klassen vs. Interfaces

- Beide ermöglichen Polymorphie und die Definition gemeinsamer Strukturen.
- Unterschiedliche Anwendungsfälle:
  - **Abstrakte Klassen**: Für enge Beziehungen und Code-Wiederverwendung.
  - **Interfaces**: Für flexible Verhaltensverträge.

# Gemeinsamkeiten

- **Gemeinsamer Zweck**:
  - Struktur für Klassen definieren.
  - Polymorphie ermöglichen.
- Ziel: Vereinfachung von Code und Wiederverwendbarkeit.

# Unterschiede

| Eigenschaft | Abstrakte Klasse | Interface |
| --- | --- | --- |
| **Vererbung** | Eine Klasse kann nur **eine** abstrakte Klasse erben. | Eine Klasse kann **mehrere** Interfaces implementieren. |
| **Methoden** | Kann abstrakte **und** konkrete Methoden enthalten. | Nur abstrakte Methoden (ab Java 8 auch Default-/Static-Methoden). |
| **Felder** | Kann **Instanzvariablen** enthalten. | Kann nur **Konstanten** ( `public static final` ) enthalten. |
| **Konstruktoren** | Kann Konstruktoren haben. | Keine Konstruktoren möglich. |
| **Zweck** | Gemeinsame Logik und Hierarchie definieren. | Flexible Verträge ohne Hierarchie. |

# Typische Anwendungsfälle: Abstrakte Klassen

- **Natürliche Hierarchie**:
  - Beispiel: `Animal` → `Mammal` → `Dog`.
- **Code-Wiederverwendung**:
  - Gemeinsame Funktionen in der Basisklasse definieren.
- **Konstruktorlogik**:
  - Wenn Subklassen einen Konstruktor benötigen.

# Typische Anwendungsfälle: Interfaces

- **Mehrfachvererbung**:
  - Beispiel: Eine Klasse `Car` kann `Vehicle` und `ElectricPowered` implementieren.
- **Unabhängige Verträge**:
  - Klassen, die keine gemeinsame Basisklasse haben, aber eine ähnliche Funktionalität bieten sollen.

# Interfaces: List, Map, Set, und Collection in Java

# List-Interface

- Das `List`-Interface speichert **geordnete** Elemente, bei denen Duplikate erlaubt sind.
- `ArrayList` ist eine häufig verwendete Implementierung des `List`-Interfaces.
- Beispiel:

```
List<String> list = new ArrayList<>();
list.add("one");
list.add("one");
list.add("two");

for (String element: list) {
    System.out.println(element);
}
```

- Ausgabe:

```
one
one
two
```

# Set-Interface

- Ein `Set` speichert **einzigartige** Elemente.
- `HashSet` ist eine Implementierung des `Set`-Interfaces.
- Beispiel:

```
Set<String> set = new HashSet<>();
set.add("one");
set.add("one");
set.add("two");

for (String element: set) {
    System.out.println(element);
}
```

- Ausgabe:

```
one
two
```

# Map-Interface

- Eine `Map` speichert **Schlüssel-Wert-Paare**, bei denen der Schlüssel eindeutig ist.
- Beispiel:

```
Map<String, String> map = new HashMap<>();
map.put("A", "Apple");
map.put("B", "Banana");

for (String key: map.keySet()) {
    System.out.println(key + ": " + map.get(key));
}
```

- Ausgabe:

```
A: Apple
B: Banana
```

# Collection-Interface

- Das `Collection`-Interface beschreibt die grundlegenden Funktionalitäten für **Sammlungen** von Objekten.
- Sowohl `List` als auch `Set` **implementieren** das `Collection`-Interface. (Achtung: was heißt hier "implementieren"?)
- Beispiel:

```
Collection<String> collection = new ArrayList<>();
collection.add("one");
collection.add("two");

for (String element : collection) {
    System.out.println(element);
}
```

- Ausgabe:

```
one
two
```

# Vererbung von Interfaces

```
java.lang.Iterable
    ↳ java.util.Collection
        ↳ java.util.List
        ↳ java.util.Set
            ↳ java.util.SortedSet
                ↳ java.util.NavigableSet
java.util.Map
    ↳ java.util.SortedMap
        ↳ java.util.NavigableMap
```

# Vererbung von Interfaces

```java
public interface Collection<E> extends Iterable<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    boolean add(E e);
    boolean remove(Object o);
    void clear();
    // und weitere Methoden...
}
```

```java
public interface Set<E> extends Collection<E> {
    // Keine zusätzlichen Methoden, nur die Garantie:
    // - Keine Duplikate
    // - Optional sortierte Elemente (bei Unterinterfaces wie SortedSet)
}
```

# Vergleich: List, Map, Set und Collection

| Feature | List (ArrayList) | Map (HashMap) | Set (HashSet) | Collection |
|---|---|---|---|---|
| **Speicherstruktur** | Geordnete Sammlung von Elementen | Schlüssel-Wert-Paare | Nur eindeutige Elemente | Allgemeine Sammlung |
| **Zugriff auf Daten** | Zugriff per Index | Zugriff über Schlüssel | Direkt über das Element | Direkt über das Element |
| **Einzigartigkeit** | Duplikate erlaubt | Schlüssel eindeutig | Elemente sind eindeutig | Duplikate können erlaubt sein |
| **Einfachheit** | Reihenfolge wird beibehalten | Muss Schlüssel für jedes Element haben | Kein Schlüssel erforderlich | Variiert je nach Implementierung |
| **Speichereffizienz** | Variiert je nach Implementierung | Zusätzlicher Speicher für Schlüssel und Werte | Geringerer Speicherverbrauch | Variiert je nach Implementierung |
| **Spezifische Methoden** | Methoden für Indexoperationen | Methoden für Schlüssel und Werte | Methoden für Elemente (add, remove) | Grundlegende Methoden wie `add`, `remove` |
| **Typische Anwendung** | Geordnete Liste von Elementen | Assoziation von Schlüsseln mit Werten | Speichern von eindeutigen Elementen | Generische Sammlung von Elementen |

**Erläuterung:**

- **List**: Gut für geordnete Sammlungen, bei denen Duplikate erlaubt sind.
- **Map**: Ideal für die Zuordnung von Schlüsseln zu Werten.
- **Set**: Praktisch, wenn Duplikate ausgeschlossen werden sollen.
- **Collection**: Generisches Oberinterface für Sammlungen von Elementen, die flexibel implementiert werden können.

# Zusammenfassung

- **Interfaces** definieren Verhalten, ohne die konkrete Implementierung festzulegen.
  - Ermöglichen es, gemeinsame Funktionalität für unterschiedliche Klassen bereitzustellen.
  - Ein Interface kann als **Typ für Variablen, Parameter und Rückgabewerte** verwendet werden.
  - Interfaces definieren **Verhalten**, nicht die Implementierung.
  - Sie ermöglichen es, **flexiblen Code** zu schreiben, der leicht erweiterbar ist.
- **List**, **Map**, **Set**, und **Collection** sind wichtige Java-Interfaces für das Arbeiten mit Datenstrukturen:
  - **List**: Speichert **geordnete** Elemente, bei denen **Duplikate** erlaubt sind.
  - **Map**: Speichert **Schlüssel-Wert-Paare**, bei denen der **Schlüssel eindeutig** ist.
  - **Set**: Speichert **einzigartige** (unique) Elemente.
  - **Collection**: Oberinterface für **allgemeine Sammlungen** von Elementen.

# Vergleich: `List` und `Map`

- **`List`**: Repräsentiert eine geordnete Sammlung, bei der Elemente durch Indizes zugänglich sind.
- **`Map`**: Arbeitet mit Schlüssel-Wert-Paaren (und ist keine Unterschnittstelle von `Collection`.)

# Gemeinsame Methoden von `List` und `Map`

| Methodensignatur | Beschreibung |
| --- | --- |
| `int size()` | Gibt die Anzahl der Elemente zurück. |
| `boolean isEmpty()` | Überprüft, ob die Struktur leer ist. |
| `void clear()` | Entfernt alle Elemente. |
| `boolean equals(Object o)` | Vergleicht die Struktur mit einem anderen Objekt auf Gleichheit. |
| `int hashCode()` | Gibt den Hash-Code der Struktur zurück. |

# Methoden exklusiv für `List`

| Methodensignatur | Beschreibung |
| --- | --- |
| `E get(int index)` | Gibt das Element an der angegebenen Position zurück. |
| `E set(int index, E element)` | Ersetzt das Element an der angegebenen Position. |
| `void add(int index, E element)` | Fügt ein Element an der angegebenen Position ein. |
| `E remove(int index)` | Entfernt das Element an der angegebenen Position. |
| `int indexOf(Object o)` | Gibt den Index der ersten Vorkommen eines Elements zurück. |
| `int lastIndexOf(Object o)` | Gibt den Index des letzten Vorkommens eines Elements zurück. |
| `ListIterator<E> listIterator()` | Gibt einen Iterator für die Liste zurück. |
| `List<E> subList(int fromIndex, int toIndex)` | Gibt eine Teilliste zurück. |

# Methoden exklusiv für `Map`

| Methodensignatur | Beschreibung |
| --- | --- |
| `V put(K key, V value)` | Fügt ein Schlüssel-Wert-Paar hinzu oder ersetzt den vorhandenen Wert. |
| `V get(Object key)` | Gibt den Wert zurück, der dem Schlüssel zugeordnet ist. |
| `V remove(Object key)` | Entfernt das Schlüssel-Wert-Paar für den angegebenen Schlüssel. |
| `boolean containsKey(Object key)` | Überprüft, ob ein Schlüssel in der Map enthalten ist. |
| `boolean containsValue(Object value)` | Überprüft, ob ein Wert in der Map enthalten ist. |
| `Set<K> keySet()` | Gibt eine Menge aller Schlüssel zurück. |
| `Collection<V> values()` | Gibt eine Sammlung aller Werte zurück. |
| `Set<Map.Entry<K, V>> entrySet()` | Gibt eine Menge aller Schlüssel-Wert-Paare zurück. |
| `V putIfAbsent(K key, V value)` | Fügt ein Paar hinzu, falls der Schlüssel noch nicht vorhanden ist. |
| `void forEach(BiConsumer<? super K, ? super V> action)` | Führt eine Aktion für jedes Schlüssel-Wert-Paar aus. |

# Fazit: `List` vs. `Map`

- **Gemeinsamkeiten**:
  - Beide bieten grundlegende Methoden wie `size()`, `isEmpty()`, `clear()`.
  - Beide sind Teil der **Collections Framework**.
- **Unterschiede**:
  - **`List`**: Geordnete Sammlung, Zugriff über Indizes.
  - **`Map`**: Arbeitet mit Schlüssel-Wert-Paaren, keine geordnete Datenstruktur.
- **Wann verwenden?**:
  - Verwende `List`, wenn die Reihenfolge wichtig ist und Du mit Indexen arbeitest.
  - Verwende `Map`, wenn Du Daten über Schlüssel effizient speichern und abrufen möchtest.

# Vergleich: `List` und `Set`

- `List` : Repräsentiert eine geordnete Sammlung, bei der Elemente durch Indizes zugänglich sind.
- `Set` : Repräsentiert eine ungeordnete Sammlung ohne doppelte Elemente.

# Gemeinsame Methoden von `List` und `Set`

| Methodensignatur | Beschreibung |
|---|---|
| `int size()` | Gibt die Anzahl der Elemente zurück. |
| `boolean isEmpty()` | Überprüft, ob die Sammlung leer ist. |
| `boolean contains(Object o)` | Überprüft, ob ein bestimmtes Element enthalten ist. |
| `boolean add(E e)` | Fügt ein Element hinzu (bei `Set` keine doppelten Werte erlaubt). |
| `boolean remove(Object o)` | Entfernt ein bestimmtes Element, falls es existiert. |
| `Iterator<E> iterator()` | Gibt einen Iterator über die Elemente der Sammlung zurück. |
| `boolean containsAll(Collection<?> c)` | Überprüft, ob alle Elemente einer anderen Sammlung enthalten sind. |
| `boolean addAll(Collection<? extends E> c)` | Fügt alle Elemente einer anderen Sammlung hinzu. |
| `boolean removeAll(Collection<?> c)` | Entfernt alle Elemente, die in der angegebenen Sammlung enthalten sind. |
| `boolean retainAll(Collection<?> c)` | Behält nur die Elemente, die auch in der angegebenen Sammlung enthalten sind. |
| `void clear()` | Entfernt alle Elemente aus der Sammlung. |

# Methoden exklusiv für `List`

| Methodensignatur | Beschreibung |
| --- | --- |
| `E get(int index)` | Gibt das Element an der angegebenen Position zurück. |
| `E set(int index, E element)` | Ersetzt das Element an der angegebenen Position. |
| `void add(int index, E element)` | Fügt ein Element an der angegebenen Position ein. |
| `E remove(int index)` | Entfernt das Element an der angegebenen Position. |
| `int indexOf(Object o)` | Gibt den Index der ersten Vorkommen eines Elements zurück. |
| `int lastIndexOf(Object o)` | Gibt den Index des letzten Vorkommens eines Elements zurück. |
| `ListIterator<E> listIterator()` | Gibt einen Iterator für die Liste zurück. |
| `List<E> subList(int fromIndex, int toIndex)` | Gibt eine Teilliste zurück. |

# Methoden exklusiv für `Set`

`Set` bietet keine Methoden, die nicht bereits in `Collection` definiert sind. Es garantiert jedoch:

- Keine doppelten Elemente.
- Keine geordnete Struktur (bei Implementierungen wie `HashSet`).

# Fazit: `List` vs. `Set`

- **Gemeinsamkeiten**:
  - Beide erben von `Collection` und bieten grundlegende Methoden wie `add()`, `remove()`, `size()`.
  - Beide repräsentieren Sammlungen von Elementen.
- **Unterschiede**:
  - `List`:
    - Elemente haben eine feste Reihenfolge.
    - Zugriff über Indizes.
    - Doppelte Werte erlaubt.
  - `Set`:
    - Keine doppelten Elemente.
    - Keine Garantie für Reihenfolge (außer bei `LinkedHashSet` oder `TreeSet`).
- **Wann verwenden?**:
  - Verwende `List`, wenn die Reihenfolge wichtig ist oder du über Indizes zugreifen möchtest.
  - Verwende `Set`, wenn keine Duplikate erlaubt sind und die Reihenfolge irrelevant ist.

## Buchspeicherung: List, Map, Set und Collection

- Wir haben eine Bibliothek, in der wir Bücher nach ISBN, Titel und Autor speichern.
- Wir verwenden vier unterschiedliche Ansätze:
  - **List**: Eine geordnete Sammlung von Büchern, bei der Duplikate erlaubt sind.
  - **Map**: ISBN als Schlüssel, Buch als Wert.
  - **Set**: Menge von Büchern, keine Duplikate.
  - **Collection**: Allgemeine Sammlung.

# Klassenstruktur: Buch

```
class Book {
    private String isbn;
    private String title;
    private String author;

    public Book(String isbn, String title, String author) {
        this.isbn = isbn;
        this.title = title;
        this.author = author;
    }

    public String getIsbn() {
        return isbn;
    }

    public String getTitle() {
        return title;
    }

    public String getAuthor() {
        return author;
    }

    @Override
    public String toString() {
        return title + " by " + author + " (ISBN: " + isbn + ")";
    }

    @Override
    public boolean equals(Object o) {
```

## Verwendung einer `List`

- Eine geordnete Sammlung von Büchern, bei der Duplikate erlaubt sind.

```
List<Book> bookList = new ArrayList<>();

bookList.add(new Book("978-3-16-148410-0", "The Catcher in the Rye", "J.D. Salinger"));
bookList.add(new Book("978-0-7432-7356-5", "The Great Gatsby", "F. Scott Fitzgerald"));
bookList.add(new Book("978-0-452-28423-4", "1984", "George Orwell"));
bookList.add(new Book("978-0-7432-7356-5", "The Great Gatsby", "F. Scott Fitzgerald")); // Duplikat erlaubt

for (Book book : bookList) {
    System.out.println(book);
}
```

## Vorteile einer `List`

- **Reihenfolge der Elemente** wird beibehalten.
- **Duplikate** sind erlaubt.
- **Einfacher Zugriff** auf Bücher über Indexpositionen.

## Verwendung einer `Map`

- Wir verwenden die ISBN als Schlüssel und das Buch als Wert.
- Ermöglicht schnellen Zugriff auf Bücher durch die ISBN.

```
Map<String, Book> bookMap = new HashMap<>();

bookMap.put("978-3-16-148410-0", new Book("978-3-16-148410-0", "The Catcher in the Rye", "J.D. Salinger"));
bookMap.put("978-0-7432-7356-5", new Book("978-0-7432-7356-5", "The Great Gatsby", "F. Scott Fitzgerald"));
bookMap.put("978-0-452-28423-4", new Book("978-0-452-28423-4", "1984", "George Orwell"));

for (String isbn : bookMap.keySet()) {
    System.out.println(bookMap.get(isbn));
}
```

## Vorteile einer `Map`

- **Schneller Zugriff** auf Bücher durch den eindeutigen Schlüssel (ISBN).
- **Keine Duplikate** der Schlüssel (ISBN).
- **Komplexere Suche** erforderlich, wenn man nach Autor oder Titel sucht.

## Verwendung eines `Set`

- Speichern einer Menge von Büchern, wobei keine Duplikate erlaubt sind.

```
Set<Book> bookSet = new HashSet<>();

bookSet.add(new Book("978-3-16-148410-0", "The Catcher in the Rye", "J.D. Salinger"));
bookSet.add(new Book("978-0-7432-7356-5", "The Great Gatsby", "F. Scott Fitzgerald"));
bookSet.add(new Book("978-0-452-28423-4", "1984", "George Orwell"));
bookSet.add(new Book("978-3-16-148410-0", "The Catcher in the Rye", "J.D. Salinger")); // Duplikat wird nicht hinzug

for (Book book : bookSet) {
    System.out.println(book);
}
```

## Vorteile eines `Set`

- **Keine Duplikate** erlaubt (Bücher werden durch ISBN verglichen).
- **Einfacher Zugriff** auf alle Bücher, aber keine Schlüssel wie in einer `Map`.
- Gut für Szenarien, in denen **jedes Element einzigartig** sein muss.

## Verwendung einer `Collection`

- Eine allgemeine Sammlung, die flexibel ist und Bücher aufnehmen kann.

```
Collection<Book> bookCollection = new ArrayList<>();

bookCollection.add(new Book("978-3-16-148410-0", "The Catcher in the Rye", "J.D. Salinger"));
bookCollection.add(new Book("978-0-7432-7356-5", "The Great Gatsby", "F. Scott Fitzgerald"));
bookCollection.add(new Book("978-0-452-28423-4", "1984", "George Orwell"));

for (Book book : bookCollection) {
    System.out.println(book);
}
```

## "Vorteile" einer `Collection`

- **Flexibilität**: Kann als Liste, Set oder Queue verwendet werden.
- **Reihenfolge** ? wird beibehalten (nur im Beispiel, nicht immer - aber in Listen wie `ArrayList`).
- **Duplikate** ? sind erlaubt (nur im Beispiel, nicht immer)

# Map? HashMap? Und Mehr?

## Weitere Map Implementierungen

| Implementierung | Beschreibung | Merkmale | Anwendungsfall |
|---|---|---|---|
| **HashMap** | Eine unsortierte, hash-basierte Implementierung. | - Bietet konstante Zeit für `put`, `get`, `remove`.<br>- Keine Garantie für die Reihenfolge. | Verwenden Sie diese, wenn schnelle Zugriffe wichtig sind und die Reihenfolge keine Rolle spielt. |
| **LinkedHashMap** | Eine Erweiterung von `HashMap`, die die Einfügereihenfolge beibehält. | - Beibehaltung der Einfügereihenfolge.<br>- Etwas langsamer als `HashMap`. | Verwenden Sie diese, wenn Sie die Einfügereihenfolge beibehalten möchten. |
| **TreeMap** | Eine `Map`, die basierend auf den Schlüsseln sortiert ist. | - Implementiert das `NavigableMap`-Interface.<br>- Bietet log(n)-Zeit für Operationen.<br>- Sortiert nach natürlicher Ordnung oder durch einen Comparator. | Verwenden Sie diese, wenn Sie eine sortierte `Map` benötigen. |
| **WeakHashMap** | Eine speicherfreundliche `Map`, deren Einträge entfernt werden können, wenn keine starken Referenzen auf die Schlüssel bestehen. | - Verwendet schwache Referenzen für Schlüssel.<br>- Nützlich für Caching und speicherfreundliche Anwendungen. | Verwenden Sie diese, wenn Sie Einträge automatisch entfernen möchten, sobald keine starken Referenzen auf den Schlüssel bestehen. |
| **IdentityHashMap** | Eine `Map`, die die Identität (`==`) und nicht `equals` für den Vergleich von Schlüsseln verwendet. | - Vergleicht Schlüssel anhand von Identität (Referenzgleichheit).<br>- Verwendet nicht die `equals`-Methode. | Verwenden Sie diese, wenn Sie Wert auf Identitätsvergleich der Schlüssel legen, z. B. in speziellen Szenarien wie Caching oder Serialisierung. |
| **ConcurrentHashMap** | Eine threadsichere Variante von `HashMap`, die eine bessere Leistung in Mehrfach-Threading-Umgebungen bietet. | - Threadsicher ohne explizite Synchronisierung.<br>- Bietet hohe Leistung in parallelen Umgebungen. | Verwenden Sie diese, wenn Sie mit mehreren Threads arbeiten und gleichzeitiger Zugriff erforderlich ist. |

## Polymorphismus in Java

- Der Begriff wird als wissenschaftlicher Fachbegriff verwendet: beschreibt die Eigenschaft eines Systems (z. B. eines Programms), Polymorphie zu unterstützen.
- Polymorphismus ermöglicht es, dass ein Objekt in mehreren Formen auftritt.
- Ein Objekt kann durch seinen eigenen Typ, die Typen seiner Elternklassen und die von ihm implementierten Interfaces dargestellt werden.
- Dies ermöglicht eine flexiblere Programmierung, indem Methoden allgemeiner gestaltet werden können.

## Beispiel: Objekt-Polymorphismus

```
String text = "Hello!";
Object obj = text; // funktioniert, da String von Object erbt

String newText = obj; // Fehler: Object kann nicht direkt zu String konvertiert werden
```

- Eine Variable vom Typ `String` kann auch als `Object` -Typ dargestellt werden.
- Eine Umkehrung der Zuweisung funktioniert jedoch nicht direkt, da `Object` nicht vom Typ `String` ist.

## Polymorphismus durch Vererbung

```java
public class Printer {
    public void printManyTimes(Object object, int times) {
        for (int i = 0; i < times; i++) {
            System.out.println(object);
        }
    }
}
```

```java
Printer printer = new Printer();
String message = "Hello!";
List<String> list = Arrays.asList("Item1", "Item2");

printer.printManyTimes(message, 3);
printer.printManyTimes(list, 2);
```

- Die Methode `printManyTimes` akzeptiert jedes Objekt als Parameter, weil sie den `Object`-Typ verwendet.
- `String` und `List` können übergeben werden, da sie von `Object` erben.

## Polymorphismus durch Interfaces

```
public void printCharacters(CharSequence sequence) {
    for (int i = 0; i < sequence.length(); i++) {
        System.out.print(sequence.charAt(i));
    }
}
```

- Die Methode `printCharacters` akzeptiert Objekte, die die `CharSequence`-Schnittstelle implementieren.
- Dies umfasst z. B. `String` und `StringBuilder`.

## Polymorphismus: Vorteile

- **Vielseitigkeit:** Objekte können durch viele ihrer Typen dargestellt werden.
- **Code-Wiederverwendbarkeit:** Einfache Implementierungen durch gemeinsame Oberklassen oder Interfaces.
- **Erweiterbarkeit:** Neue Klassen können hinzugefügt werden, ohne den bestehenden Code anzupassen.

## Formal(er)e Einführung in Polymorphismus

- **Polymorphismus** kommt aus dem Griechischen:
  - "poly" = "viele"
  - "morph" = "Formen"
- In der Theorie bedeutet es "viele Formen annehmen können".
- Verwendet, um zu beschreiben, wie ein Konzept oder eine Funktionalität in verschiedenen Kontexten mehrere Bedeutungen oder Darstellungen annehmen kann.

## Ursprung und Bedeutung

- Ursprünglich aus der Biologie, um Lebewesen zu beschreiben, die verschiedene Erscheinungsformen haben.
- In der Mathematik und Informatik wird es verwendet, um die Fähigkeit eines Systems zu beschreiben, durch einen einzigen Operator oder eine einzige Funktion verschiedene Datentypen zu verarbeiten.

## Arten von Polymorphismus

- **Ad-hoc-Polymorphismus**
  - Verschiedene Funktionen oder Operatoren können sich abhängig vom Typ der Argumente unterschiedlich verhalten.
  - Beispiel: Überladen von Funktionen oder Operatoren (Compile-Zeit!).
- **Parametrischer Polymorphismus**
  - Eine Funktion oder ein Datenstrukturentyp kann für beliebige Typen verwendet werden.
  - Beispiel: Generische Programmierung, Templates.
  - `ArrayList<Strng>`, `ArrayList<Integer>`, ...

## Parametrischer Polymorphismus (Detail)

- Ein Mechanismus, der es erlaubt, Funktionen oder Datenstrukturen unabhängig von konkreten Datentypen zu definieren.
- **Beispiel**: Eine Liste kann unabhängig vom Typ ihrer Elemente definiert werden:
  - `List<Integer>`
  - `List<String>`
- Typische Anwendungen:
  - Typgenerische Datenstrukturen (Listen, Bäume, etc.).
  - Funktionen, die auf mehreren Typen operieren, z. B. Sortieralgorithmen.

## Ad-hoc-Polymorphismus (Detail)

- Bezieht sich auf das spezifische Verhalten einer Funktion abhängig von der Art ihrer Argumente.
- **Überladung**: Dieselbe Funktion kann mehrere Signaturen haben.
- **Operatorüberladung**: Ein Operator kann für verschiedene Typen unterschiedlich funktionieren.

**Beispiel**:

- `+` kann sowohl für Ganzzahlen als auch für Gleitkommazahlen definiert werden:
  - `1 + 1` ergibt `2`
  - `1.5 + 2.5` ergibt `4.0`

## Subtyp-Polymorphismus

- Ein weiteres, in objektorientierten Sprachen übliches, Beispiel von Polymorphismus ist der **Subtyp-Polymorphismus**.
- Ein Objekt eines bestimmten Typs kann als Objekt eines seiner Supertypen behandelt werden.
- Dies ermöglicht **Vererbung**, wodurch Objekte einer Unterklasse auf die gleiche Weise wie Objekte einer Oberklasse verwendet werden können.

## Polymorphismus in der Programmierung

- Polymorphismus ermöglicht die Entwicklung von **flexiblem und erweiterbarem Code**.
- **Wiederverwendbarkeit** von Funktionen oder Klassen ohne Rücksicht auf spezifische Typen.
- **Einheitliche Schnittstellen** für unterschiedliche Typen: Ein Operator/Funktion kann auf verschiedene Typen angewendet werden.

## Zusammenfassung

- Polymorphismus bietet die Möglichkeit, Funktionen und Typen **universell** zu gestalten.
- **Ad-hoc-Polymorphismus**: Methoden oder Operatoren, die abhängig vom Typ unterschiedliche Implementierungen haben.
- **Parametrischer Polymorphismus**: Typen und Funktionen, die für beliebige Typen verwendet werden können.
- **Subtyp-Polymorphismus**: Objekte von Unterklassen können als Supertypen behandelt werden.
- ...

## Prinzipien der Programmierung: Vererbung

- **Vererbung** erlaubt es, Gemeinsamkeiten in einer Oberklasse zu definieren und von ihr zu erben.
- Erfüllt das Prinzip **Don't Repeat Yourself (DRY)**: Vermeidung von Code-Duplikation.

**Beispiel**:

```
public class Animal {
    public void eat() {
        System.out.println("Eating...");
    }
}

public class Dog extends Animal {
    public void bark() {
        System.out.println("Barking...");
    }
}

Dog dog = new Dog();
dog.eat();  // Von Animal geerbt
dog.bark(); // In Dog definiert
```

## Prinzipien der Programmierung: Abstraktion (durch Interface)

- **Interfaces** erlauben die Definition von Verhalten ohne konkrete Implementierung.
- Unterstützt das Prinzip der **Abstraktion**: Trennung von **was** ein Objekt tut und **wie** es das tut.

**Beispiel**:

```
public interface Flyable {
    void fly();
}

public class Bird implements Flyable {
    public void fly() {
        System.out.println("Bird is flying...");
    }
}

Flyable bird = new Bird();
bird.fly();  // Abstraktes Verhalten, von der konkreten Implementierung getrennt
```

## Prinzipien der Programmierung: Polymorphismus

- **Polymorphismus** ermöglicht es, Code für verschiedene Objekttypen wiederzuverwenden.
- Erfüllt das **Open/Closed-Prinzip**: Softwarekomponenten sollen offen für Erweiterungen, aber geschlossen für Modifikationen sein.

**Beispiel**:

```
public class Animal {
    public void makeSound() {
        System.out.println("Some sound...");
    }
}

public class Dog extends Animal {
    public void makeSound() {
        System.out.println("Bark!");
    }
}

public class Cat extends Animal {
    public void makeSound() {
        System.out.println("Meow!");
    }
}

Animal dog = new Dog();
Animal cat = new Cat();

dog.makeSound();  // Bark!
cat.makeSound();  // Meow!
```

## Vererbung und Interface: Single Responsibility

- **Vererbung** und **Interfaces** erlauben die Einhaltung des **Single Responsibility Principle (SRP)**.
- Eine Klasse sollte nur eine klar definierte Verantwortung tragen.

**Beispiel**:

```java
public interface Saveable {
    void save();
}

public class User implements Saveable {
    private String name;
    public User(String name) {
        this.name = name;
    }

    public void save() {
        System.out.println("Saving user: " + name);
    }
}

Saveable user = new User("Alice");
user.save();  // Fokus auf eine einzige Verantwortlichkeit
```

## Polymorphismus und Erweiterbarkeit

- **Polymorphismus** und **Interfaces** erleichtern die Erweiterung des Codes ohne Änderungen an bestehenden Klassen.
- Entspricht dem Prinzip der **Modularität** und **Erweiterbarkeit**.

**Beispiel**:

```java
public interface PaymentMethod {
    void pay(double amount);
}

public class CreditCardPayment implements PaymentMethod {
    public void pay(double amount) {
        System.out.println("Paid with credit card: " + amount);
    }
}

public class PayPalPayment implements PaymentMethod {
    public void pay(double amount) {
        System.out.println("Paid with PayPal: " + amount);
    }
}

// Einfach neue Bezahlmethoden hinzufügen, ohne vorhandenen Code zu ändern
PaymentMethod payment = new CreditCardPayment();
payment.pay(100.0);
```

## Zusammenfassung: Prinzipien und Konzepte

- **Vererbung**: DRY-Prinzip, Wiederverwendbarkeit.
- **Interfaces**: Abstraktion, Trennung von Verhalten und Implementierung.
- **Polymorphismus**: Flexibilität, Open/Closed-Prinzip, Erweiterbarkeit.

Durch die richtige Anwendung dieser Konzepte lassen sich wartbare, erweiterbare und saubere Systeme erstellen.

Quiz: link