Prinzipien der Programmierung

Daniel Merkle

Wintersemester 2024

(Teil 05)

Lernziele (Objektorientierte Programmierung)

- Die Konzepte von **Klasse** und **Objekt** wiederholen.
- Verstehen, wie man Programme mit Objekten umsetzt.
- Erkennen, dass Objekte Programme verständlicher machen können.

Worum geht es bei der objektorientierten Programmierung?

Beispiel: Eine Uhr

- Uhr mit drei Zeigern: **Stunden**, **Minuten**, **Sekunden**.
- Sekundenzeiger bewegt sich jede Sekunde, Minutenzeiger alle 60 Sekunden, Stundenzeiger alle 60 Minuten.
- Format: `Stunden: Minuten: Sekunden`.

```
int hours = 0;
int minutes = 0;
int seconds = 0;
```

Eine Uhr

```
int hours = 0;
int minutes = 0;
int seconds = 0;

while (true) {
    // 1. Printing the time
    if (hours < 10) {
        System.out.print("0");
    }
    System.out.print(hours);

    System.out.print(":");

    if (minutes < 10) {
        System.out.print("0");
    }
    System.out.print(minutes);

    System.out.print(":");

    if (seconds < 10) {
        System.out.print("0");
    }
    System.out.print(seconds);
    System.out.println();

    // 2. The second hand's progress
    seconds = seconds + 1;

    // 3. The other hand's progress when necessary
    if (seconds > 59) {
        minutes = minutes + 1;
        seconds = 0;

        if (minutes > 59) {
```

Codeverständlichkeit?

Codeverständlichkeit

- Schwierigkeit beim Verstehen von Code mit einfachen `int`-Variablen
- Beispiel: Uhr aus drei `int`-Variablen
- Code sollte lesbar und verständlich sein
- Wichtigkeit von klarer Code-Struktur

Zitat von Kent Beck

> **„Jeder Dummkopf kann Code schreiben, den ein Computer versteht. Gute Programmierende schreiben Code, den Menschen verstehen können.“**

Uhrzeiger als Klasse implementieren

```java
public class ClockHand {
    private int value;
    private int limit;

    public ClockHand(int limit) {
        this.limit = limit;
        this.value = 0;
    }

    public void advance() {
        this.value = this.value + 1;

        if (this.value >= this.limit) {
            this.value = 0;
        }
    }

    public int value() {
        return this.value;
    }

    public String toString() {
        if (this.value < 10) {
            return "0" + this.value;
        }

        return "" + this.value;
    }
}
```

Verbesserte Uhr mit `ClockHand`-Klasse

- Erstellung der `ClockHand`-Klasse macht die Uhr verständlicher
  - Einfachere Ausgabe der Uhrzeiger
  - Fortschritt der Zeiger ist in der Klasse verborgen

Automatisches Zurücksetzen der Zeiger

- Zeiger kehren automatisch zum Anfang zurück durch obere Grenzvariablen
- Unterschied zur Verwendung von `int`-Variablen:
    - Bei `int`-Variablen:
        - Prüfung, ob Wert die obere Grenze überschreitet
        - Bei Überschreitung: Wert auf Null setzen und nächsten Zeiger inkrementieren

Fortschreiten der Zeigerobjekte

- Minutenzeiger schreitet voran, wenn Sekundenzeiger Null ist
- Stundenzeiger schreitet voran, wenn Minutenzeiger Null ist
- **ClockHand**-Klasse kapselt die Logik eines Uhrzeigers.
- Methoden zur Fortschritt- und Anzeige-Steuerung.

Zusammenspiel von Uhrzeigern

```
ClockHand hours = new ClockHand(24);
ClockHand minutes = new ClockHand(60);
ClockHand seconds = new ClockHand(60);
```

- **Objekte** der ClockHand-Klasse repräsentieren Stunden, Minuten und Sekunden.

```
while (true) {
    System.out.println(hours + ":" + minutes + ":" + seconds);
    seconds.advance();
    if (seconds.value() == 0) {
        minutes.advance();
        if (minutes.value() == 0) {
            hours.advance();
        }
    }
}
```

Objektorientierte Programmierung

- Isoliert Konzepte in eigene Einheiten
- Erstellung von **Abstraktionen**

Warum eigene Klassen erstellen?

- **Details verbergen** (Abstraktion)
  - Beispiel: Rotation eines Zeigers innerhalb der Klasse
- **Einfachere Nutzung** durch klar benannte Methoden
  - Methode `advance()` statt if-Anweisung und Zuweisung
- **Wiederverwendbarkeit** in anderen Programmen
  - Klasse könnte z. B. `CounterLimitedFromTop` heißen
- **Flexibilität**
  - Implementierungsdetails können bei Bedarf geändert werden

Trennung von Konzepten

- **Uhr** besteht aus drei Zeigern (Sekunden, Minuten, Stunden)
- **Uhr** als eigenes Konzept
  - Erstellung einer eigenen Klasse `Clock`
  - Zeiger werden innerhalb der Klasse verborgen

Erstellung der `Clock`-Klasse

- **Verborgenheit** der Zeiger innerhalb der Uhr
- **Klarheit** durch Abstraktion der Uhr als Konzept
- **Weiterer Schritt** in der objektorientierten Programmierung

Abstraktion durch Klassen

- Objektorientierte Programmierung isoliert Konzepte in Klassen.
- Eine Klasse enthält **Daten** (Variablen) und **Verhalten** (Methoden).
- Versteckt Implementierungsdetails und bietet einfache Schnittstellen.

Beispiel: `ClockHand.advance()`

Eine Uhr (unter Verwendung von OOP)

```java
public class Clock {
    private ClockHand hours;
    private ClockHand minutes;
    private ClockHand seconds;

    public Clock() {
        this.hours = new ClockHand(24);
        this.minutes = new ClockHand(60);
        this.seconds = new ClockHand(60);
    }

    public void advance() {
        this.seconds.advance();

        if (this.seconds.value() == 0) {
            this.minutes.advance();

            if (this.minutes.value() == 0) {
                this.hours.advance();
            }
```

Eine Uhr (unter Verwendung von OOP)

```java
Clock clock = new Clock();

while (true) {
    System.out.println(clock);
    clock.advance();
}
```

Vergleich mit ursprünglicher Lösung!

Objektorientierte Programmierung: Die große Idee

- Programme bestehen aus **kleinen**, **klar abgegrenzten Objekten**, die zusammenarbeiten.
- Objekte kapseln Zustände und Verhalten und interagieren durch Methoden.

Objekt

- **Objekt**: Eine unabhängige Einheit mit Daten (Instanzvariablen) und Verhalten (Methoden)
- Objekte können Konzepte aus dem Problemfeld beschreiben oder Interaktionen koordinieren
- Interagieren durch **Methodenaufrufe**:
  - Informationen anfordern
  - Anweisungen geben

Eigenschaften von Objekten

- Klar definierte Grenzen und Verhaltensweisen
- Kennt nur notwendige Objekte zur Erfüllung seiner Aufgabe
- Verbirgt interne Vorgänge
  - Zugriff über klar definierte Methoden
- Unabhängig von nicht relevanten Objekten

Klassen und Objekte

- **Klasse**: Bauplan zur Erstellung von Objekten
  - Definiert Variablen und Methoden
- **Objekt** wird über Konstruktor erstellt
- Beispiel: Klasse `Person`
  - Eigenschaften: Name, Alter, Gewicht, Größe
  - Methoden: BMI, maximale Herzfrequenz berechnen

Beispielklasse `Person`

```java
public class Person {
    private String name;
    private int age;
    private double weight;
    private double height;

    public Person(String name, int age, double weight, double height) {
        this.name = name;
        this.age = age;
        this.weight = weight;
        this.height = height;
    }

    public double bodyMassIndex() {
        return this.weight / (this.height * this.height);
    }

    public double maximumHeartRate() {
        return 206.3 - (0.711 * this.age);
    }

    public String toString() {
        return this.name + ", BMI: " + this.bodyMassIndex()
            + ", maximale Herzfrequenz: " + this.maximumHeartRate();
    }
}
```

Nutzung der Klasse `Person`

```java
Scanner reader = new Scanner(System.in);
System.out.println("What's your name?");
String name = reader.nextLine();
System.out.println("What's your age?");
int age = Integer.valueOf(reader.nextLine());
System.out.println("What's your weight?");
double weight = Double.valueOf(reader.nextLine());
System.out.println("What's your height?");
double height = Double.valueOf(reader.nextLine());

Person person = new Person(name, age, weight, height);
System.out.println(person);
```

Beispielausgabe:

```
What's your name?
Napoleone Buonaparte
What's your age?
51
What's your weight?
80
What's your height?
1.70
Napoleone Buonaparte, BMI: 27.68166089965398, maximale Herzfrequenz: 170.03900000000002
```

Definition einer Klasse

- **Klasse** definiert Objekttypen
  - Enthält Instanzvariablen (Daten)
  - Konstruktor(en) (Erstellung)
  - Methoden (Verhalten)
- Beispiel: Klasse `Rectangle`

Beispielklasse `Rectangle`

```java
public class Rectangle {

    // Instanzvariablen
    private int width;
    private int height;

    // Konstruktor
    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    // Methoden
    public void widen() {
        this.width = this.width + 1;
    }

    public void narrow() {
        if (this.width > 0) {
            this.width = this.width - 1;
```

Nutzung der Klasse `Rectangle`

```java
Rectangle first = new Rectangle(40, 80);
Rectangle rectangle = new Rectangle(10, 10);
System.out.println(first);
System.out.println(rectangle);

first.narrow();
System.out.println(first);
System.out.println(first.surfaceArea());
```

Beispielausgabe:

```
(40, 80)
(10, 10)
(39, 80)
3920
```

Methoden und Rückgabewerte

- Methoden können:
  - **Keinen Wert zurückgeben** (`void`)
  - **Einen Wert zurückgeben** (z.B. `int` , `double`)
- `toString()` -Methode gibt String zur Darstellung des Objekts zurück
- Objekte werden mit `new` und dem Konstruktor erstellt

Lernziele (Code-Wiederholungen entfernen, Methoden- und Konstruktorüberladung)

- Verstehen des Begriffs **Überladung**.
- Mehrere **Konstruktoren** für eine Klasse erstellen.
- Mehrere **Methoden** mit demselben Namen in einer Klasse erstellen.

Die `Person`-Klasse

```java
public class Person {
    private String name;
    private int age;
    private int height;
    private int weight;

    public Person(String name) {
        this.name = name;
        this.age = 0;
        this.weight = 0;
        this.height = 0;
    }

    public void printPerson() {
        System.out.println(this.name + " is " + this.age + " years old");
    }
}
```

- Jeder Person wird bei der Erstellung automatisch **0 Jahre** zugewiesen.
- Der **Konstruktor** setzt `age` auf 0.

Konstruktorüberladung

```java
public Person(String name) {
    this.name = name;
    this.age = 0;
}

public Person(String name, int age) {
    this.name = name;
    this.age = age;
}
```

- Wir können **mehrere Konstruktoren** für eine Klasse erstellen.
- Überladung ermöglicht es, verschiedene **Parameter** zu akzeptieren.

```java
Person paul = new Person("Paul", 24);
Person ada = new Person("Ada");
```

Paul is 24 years old.

Ada is 0 years old.

Konstruktorüberladung verbessern

```java
public Person(String name) {
    this(name, 0);
}

public Person(String name, int age) {
    this.name = name;
    this.age = age;
}
```

- Der erste Konstruktor ruft den zweiten Konstruktor mit `this` auf.
- `this(name, 0)` ersetzt den Code durch den Aufruf des zweiten Konstruktors.

NB! Der Konstruktoraufruf innerhalb eines Konstrukturs muss der **erste Befehl** sein.

Beispiel eines syntaktischen Fehlers

```
public class MyClass {
    public MyClass() {
        System.out.println("Hallo"); // Fehler: Muss nach `this(...)` oder `supe
        this("Test");
    }

    public MyClass(String text) {
        System.out.println(text);
    }
}
```

Methodenüberladung

Methoden können wie Konstruktoren überladen werden:

```
public void growOlder() {
    this.age++;
}

public void growOlder(int years) {
    this.age += years;
}
```

```
Person paul = new Person("Paul", 24);
paul.growOlder();    // Alter um 1 erhöhen
paul.growOlder(10);  // Alter um 10 erhöhen
```

Paul is 24 years old.

Paul is 25 years old.

Paul is 35 years old.

Aufruf einer überladenen Methode aus einer anderen

```
public void growOlder() {
    this.growOlder(1);
}

public void growOlder(int years) {
    this.age += years;
}
```

- Die **parameterlose** Methode `growOlder()` ruft die überladene Version auf.

Quiz Überladen von Konstruktoren `Pet`

Quiz Überladen von Konstruktoren `Counter`

Zusammenfassung

- **Konstruktorüberladung** ermöglicht es, mehrere Konstruktoren mit unterschiedlichen Parametern zu erstellen.
- **Methodenüberladung** erlaubt die Erstellung mehrerer Methoden mit demselben Namen, aber unterschiedlichen Parametern.
- Überladene Methoden und Konstruktoren verbessern die **Flexibilität** und **Wiederverwendbarkeit** des Codes.

Lernziele (Primitive und Referenzvariablen)

(nochmals!)

- Was sind **primitive** und **Referenzvariablen**?
- Welche **primitiven Typen** gibt es in Java?
- Was sind die **Unterschiede** zwischen primitiven und Referenzvariablen bei Zuweisungen und Methodenaufrufen?

Primitive und Referenzvariablen

- Primitive Variablen speichern **Werte** direkt.
- Referenzvariablen speichern **Referenzen** auf Objekte.

Beispiele:

```
int value = 10;
System.out.println(value); // Ausgabe: 10

Name luke = new Name("Luke");
System.out.println(luke); // Ausgabe: Name@4aa298b7
```

Primitive: Direkte Wertausgabe.

Referenz: Ausgabe zeigt Objekttyp und Speicherort.

Referenzvariablen und toString

Ohne `toString()` wird die **Referenz** angezeigt:

```
Name luke = new Name("Luke");
System.out.println(luke); // Name@4aa298b7
```

Mit `toString()` (in der Klasse `Name`):

```
public String toString() {
    return this.name;
}

Name luke = new Name("Luke");
System.out.println(luke); // Ausgabe: Luke
```

- `toString()` definiert, wie ein Objekt als **String** ausgegeben wird.

Primitive Typen in Java

- **boolean**: true/false
- **byte**: 8-Bit-Ganzzahl (-128 bis 127)
- **char**: 16-Bit-Zeichen
- **short**: 16-Bit-Ganzzahl (-32,768 bis 32,767)
- **int**: 32-Bit-Ganzzahl
- **long**: 64-Bit-Ganzzahl
- **float**: 32-Bit-Gleitkommazahl
- **double**: 64-Bit-Gleitkommazahl

Beispiele:

```
boolean truthValue = false;
int integer = 42;
double floatingPointNumber = 4.2;
```

Wertzuweisungen bei primitiven Variablen

```
int first = 10;
int second = first;
int third = second;
System.out.println(first + " " + second + " " + third); // 10 10 10

second = 5;
System.out.println(first + " " + second + " " + third); // 10 5 10
```

- Zuweisungen **kopieren** den Wert der Variablen.
- Änderungen in einer Variablen betreffen nicht die anderen.

Primitive Variablen als Methodenparameter

```
public static void main(String[] args) {
    int number = 1;
    call(number);
    System.out.println("Number still: " + number);
}

public static void call(int number) {
    System.out.println("Number in the beginning: " + number);
    number += 1;
    System.out.println("Number in the end: " + number);
}
```

Ausgabe:

```
Number in the beginning: 1
Number in the end: 2
Number still: 1
```

- Der Wert wird **kopiert**, Änderungen in der Methode betreffen die Originalvariable nicht.

Code Visualization, primitive, 3-primitive-and-reference-variables, `Example`

Quiz, modifying object variable, 3-primitive

Referenzvariablen

```java
Name leevi = new Name("Leevi");
```

- Eine **Referenz** auf das Objekt wird gespeichert.
- Der Wert der Referenz zeigt auf den Speicherort des Objekts.
- Variablenname referenziert den Ort, nicht den Wert selbst.

Wieder mal : Klasse `Person`

```
public class Person {
    private String name;
    private int birthYear;

    public Person(String name) {
        this.name = name;
        this.birthYear = 1970;
    }

    public int getBirthYear() {
        return this.birthYear;
    }

    public void setBirthYear(int birthYear) {
        this.birthYear = birthYear;
    }

    public String toString() {
        return this.name + " (" + this.birthYear + ")";
    }
```

Beispiel: Referenzvariablen und Methoden

```java
public class Example {
    public static void main(String[] args) {
        Person first = new Person("First");

        System.out.println(first);
        older(first);
        System.out.println(first);

        Person second = first;
        older(second);

        System.out.println(first);
    }

    public static void older(Person person) {
        person.setBirthYear(person.getBirthYear() + 1);
    }
}
```

Ausgabe:

```
First (1970)
First (1971)
First (1972)
```

Unterschiede in Methodenaufrufen

```
Person second = first;
youthen(second);
System.out.println(first); // First (1972)
```

- **Referenzvariablen** verweisen auf dasselbe Objekt.
- Methoden können den Zustand des **Originalobjekts** ändern, da die Referenz kopiert wird.

Java

```
 1 public class Example {
 2     public static void main(Strin
 3         Person first = new Person
 4 
 5         System.out.println(first)
 6         youthen(first);
 7         System.out.println(first)
 8 
 9         Person second = first;
10         youthen(second);
11 
12         System.out.println(first)
13     }
14 
15     public static void youthen(Pe
16         person.setBirthYear(perso
17     }
18 }
```

Edit Code & Get AI Help

line that just executed

next line to execute

< Prev Next >

Done running (29 steps)

Visualized with pythontutor.com

Print output (drag lower right corner to resize)

```
First (1970)
First (1971)
First (1972)
```

Frames

main:13

first

second

Objects

Person instance

name	"First"
birthYear	1972

Zusammenfassung

- **Primitive Variablen** speichern Werte direkt.
- **Referenzvariablen** speichern Verweise auf Objekte.
- Primitive Werte werden bei Zuweisungen und Methodenaufrufen **kopiert**.
- Referenzwerte zeigen auf das **gleiche Objekt**, daher können Methoden das Originalobjekt ändern.

Arbiträre Größe und Präzision in Java

Java unterstützt "Arbitrary Size" und "Arbitrary Precision" für Ganzzahlen und Dezimalzahlen durch spezielle Klassen wie `BigInteger` und `BigDecimal`.

Was bedeutet "Arbitrary Size" und "Arbitrary Precision"?

- **Arbitrary Size**: Zahlen ohne Begrenzung auf bestimmte Bitgröße, beschränkt nur durch den verfügbaren Speicher.
- **Arbitrary Precision**: Berechnungen mit maximaler Präzision ohne Rundungsfehler, z.B. für Finanz- und wissenschaftliche Anwendungen.

Die Klassen `BigInteger` und `BigDecimal`

`BigInteger`

- Ermöglicht die Arbeit mit ganzen Zahlen beliebiger Größe.
- Beispiel: Sehr große Zahlen wie Faktorialwerte oder kryptografische Berechnungen.

`BigDecimal`

- Ermöglicht präzise Berechnungen mit Dezimalzahlen.
- Beispiel: Finanzberechnungen, bei denen Rundungsfehler vermieden werden müssen.

Beispiel: Große Ganzzahlen mit `BigInteger`

```java
import java.math.BigInteger;

public class BigIntegerExample {
    public static void main(String[] args) {
        BigInteger bigNumber = new BigInteger("123456789012345678901234567890");
        BigInteger anotherBigNumber = new BigInteger("9876543210987654
        BigInteger result = bigNumber.multiply(anotherBigNumber);
        System.out.println("Ergebnis: " + result);
    }
}
```

Erklärung: Mit `BigInteger` können wir Werte multiplizieren, die weit über den Bereich von `int` oder `long` hinausgehen.

Beispiel: Hohe Präzision mit `BigDecimal`

```java
import java.math.BigDecimal;

public class BigDecimalExample {
    public static void main(String[] args) {
        BigDecimal wert1 = new BigDecimal("1234567890.12345678901234567890");
        BigDecimal wert2 = new BigDecimal("9876543210.98765432109876543210");
        BigDecimal ergebnis = wert1.add(wert2);
        System.out.println("Ergebnis: " + ergebnis);
    }
}
```

Erklärung: `BigDecimal` stellt sicher, dass wir eine präzise Addition von Dezimalzahlen ohne Rundungsfehler durchführen können.

Methoden für `BigInteger` und `BigDecimal`

- `BigInteger`:
  - `add()`, `subtract()`, `multiply()`, `divide()`
  - Bitmanipulation und Vergleich
- `BigDecimal`:
  - `add()`, `subtract()`, `multiply()`, `divide()`
  - Rundung und Präzision anpassbar

Anwendungsfälle

- **Kryptografie**: Berechnung extrem großer Primzahlenprodukte mit `BigInteger`.
- **Finanzwesen**: Präzise Dezimalberechnungen ohne Rundungsfehler mit `BigDecimal`.
- **Wissenschaftliche Berechnungen**: Arbiträr große und präzise Berechnungen, z.B. für physikalische Simulationen oder statistisch korrektes Sampling mit Boltzmann-Sampling
- **Effizienz?**

Lernziele (Objekte und Referenzen)

- Kenntnisse über Klassen und Objekte auffrischen.
- Verstehen, was eine `null`-Referenz ist und wie **NullPointerExceptions** entstehen.
- Ein Objekt als Methodenparameter verwenden und ein Objekt zurückgeben.
- Die Methode `equals` implementieren, um Objektgleichheit zu prüfen.

Da ist sie wieder: Klasse `Person`

```java
public class Person {
    private String name;
    private int age;
    private int weight;
    private int height;

    public Person(String name) {
        this(name, 0, 0, 0);
    }

    public Person(String name, int age, int height, int weight) {
        this.name = name;
        this.age = age;
        this.weight = weight;
        this.height = height;
    }

    public void growOlder() {
        this.age++;
    }
}
```

Zuweisung von Referenzvariablen

- **Kopieren von Referenzen**:

```java
Person ball = joan;
```

  - `ball` erhält die Referenz von `joan`
  - Beide Variablen verweisen auf dasselbe Objekt

- **Auswirkung**:
  - Änderungen über eine Referenz sind über die andere sichtbar

Beispiel: Gemeinsame Referenz

```
Person joan = new Person("Joan Ball");
System.out.println(joan);

Person ball = joan;
ball.growOlder();
ball.growOlder();

System.out.println(joan);
```

Ausgabe:

```
Joan Ball, age 0 years
Joan Ball, age 2 years
```

- `joan` und `ball` verweisen auf dasselbe Objekt
- Methode `growOlder()` ändert den Zustand des Objekts

Zuweisung eines neuen Objekts

- **Neue Referenzzuweisung**:

```java
joan = new Person("Joan B.");
System.out.println(joan);
```

- **Ergebnis**:
  - `joan` verweist jetzt auf ein neues Objekt
  - `ball` verweist weiterhin auf das ursprüngliche Objekt

Ausgabe:

```
Joan B., age 0 years
```

Null-Referenzen und `NullPointerException`

- **`null`-Referenz**:
  - Variable, die auf kein Objekt verweist
  - Wird durch Zuweisung von `null` gesetzt
- **Gefahr**:
  - Aufruf von Methoden auf einer `null`-Referenz führt zu `NullPointerException`

Beispiel: `NullPointerException`

```
Person joan = new Person("Joan Ball");
joan = null;

System.out.println(joan);    // Ausgabe: null
joan.growOlder();            // Führt zu NullPointerException
```

- **Fehlermeldung**:

  ```
  Exception in thread "main" java.lang.NullPointerException
  ```

- **Lösung**:
  - Überprüfen, ob die Referenz `null` ist, bevor Methoden aufgerufen werden

Garbage Collection

- **Unreferenzierte Objekte**:
  - Objekte ohne Referenz gelten als Müll
  - Beispiel:

    ```
    ball = null;
    ```

- **Automatische Speicherbereinigung**:
  - Java's Garbage Collector entfernt solche Objekte
  - Speicher wird freigegeben

Objekte als Methodenparameter

- **Referenzübergabe**:
  - Parameter erhalten Kopie der Referenz
  - Änderungen innerhalb der Methode beeinflussen das Originalobjekt
- **Beispiel**:

```java
public boolean allowedToRide(Person person) {
    return person.getHeight() >= this.lowestHeight;
}
```

Beispiel: Freizeitpark-Fahrgeschäft

```
Person matt = new Person("Matt", 180);
AmusementParkRide ride = new AmusementParkRide("Water track", 140);

if (ride.allowedToRide(matt)) {
    System.out.println(matt.getName() + " may enter the ride");
} else {
    System.out.println(matt.getName() + " may not enter the ride");
}
```

Ausgabe:

```
Matt may enter the ride
```

Benutzung

```java
Person matt = new Person("Matt");
matt.setWeight(86);
matt.setHeight(180);

Person jasper = new Person("Jasper");
jasper.setWeight(34);
jasper.setHeight(132);

AmusementParkRide waterTrack = new AmusementParkRide("Water track", 140);

if (waterTrack.allowedToRide(matt)) {
    System.out.println(matt.getName() + " may enter the ride");
} else {
    System.out.println(matt.getName() + " may not enter the ride");
}

if (waterTrack.allowedToRide(jasper)) {
    System.out.println(jasper.getName() + " may enter the ride");
} else {
    System.out.println(jasper.getName() + " may not enter the ride");
}

System.out.println(waterTrack);
```

Ausgabe

```
Matt may enter the ride
Jasper may not enter the ride
Water track, minimum height: 140
```

Personen im Fahrgeschäft zählen

```java
public class AmusementParkRide {
    private String name;
    private int lowestHeight;
    private int visitors;

    public AmusementParkRide(String name, int lowestHeight) {
        this.name = name;
        this.lowestHeight = lowestHeight;
        this.visitors = 0;
    }

    public boolean allowedToRide(Person person) {
        if (person.getHeight() < this.lowestHeight) {
            return false;
        }

        this.visitors++;
        return true;
    }
```

Benutzung

```java
Person matt = new Person("Matt");
matt.setWeight(86);
matt.setHeight(180);

Person jasper = new Person("Jasper");
jasper.setWeight(34);
jasper.setHeight(132);

AmusementParkRide waterTrack = new AmusementParkRide("Water track", 140);

if (waterTrack.allowedToRide(matt)) {
    System.out.println(matt.getName() + " may enter the ride");
} else {
    System.out.println(matt.getName() + " may not enter the ride");
}

if (waterTrack.allowedToRide(jasper)) {
    System.out.println(jasper.getName() + " may enter the ride");
} else {
    System.out.println(jasper.getName() + " may not enter the ride");
}

System.out.println(waterTrack);
```

Man beachte : dieser Code ist unverändert!

Ausgabe

```
Matt may enter the ride
Jasper may not enter the ride
Water track, minimum height: 140, visitors: 1
```

Objekte als Objektvariablen

- **Komposition**:
  - Objekte können andere Objekte als Instanzvariablen enthalten
- **Beispiel**:

```
public class Person {
    private String name;
    private SimpleDate birthday;
}
```

  - `Person` enthält ein `SimpleDate`-Objekt als Geburtstag

Beispiel: Person mit Geburtstag

```java
SimpleDate date = new SimpleDate(1, 1, 1990);
Person john = new Person("John Doe", date);

System.out.println(john);
```

Ausgabe:

```
John Doe, born on 1.1.1990
```

- `Person` enthält Referenz auf `SimpleDate`
- Zusammengesetzte Objekte ermöglichen komplexe Datenstrukturen

Parameterübergabe in Java und C++

Pass by Value vs. Pass by Reference

Parameterübergabe in Java

- **Java übergibt immer "by Value"**
  - Primitive Datentypen: Wert wird kopiert
  - Objektreferenzen: Referenz wird kopiert
- **Wichtig:** Es wird eine **Kopie der Referenz** übergeben, nicht die Referenz selbst

Beispiel in Java: Primitive Datentypen

```java
public class Main {
    public static void main(String[] args) {
        int zahl = 10;
        increment(zahl);
        System.out.println("Zahl nach increment: " + zahl); // Ausgabe: 10
    }

    public static void increment(int num) {
        num++;
    }
}
```

- **Erklärung:**
  - `num` ist eine Kopie von `zahl`
  - Änderungen an `num` beeinflussen `zahl` nicht

Beispiel in Java: Objektreferenzen

```java
public class Main {
    public static void main(String[] args) {
        int[] array = {1, 2, 3};
        modifyArray(array);
        System.out.println("Erstes Element: " + array[0]); // Ausgabe: 99
    }

    public static void modifyArray(int[] arr) {
        arr[0] = 99;
    }
}
```

- **Erklärung:**
  - `arr` ist eine Kopie der Referenz von `array`
  - Beide Referenzen zeigen auf dasselbe Array
  - Änderungen am Objekt (Array) sind sichtbar

Neuzuweisung von Referenzen in Java

```
public class Main {
    public static void main(String[] args) {
        int[] array = {1, 2, 3};
        reassignArray(array);
        System.out.println("Länge des Arrays: " + array.length); // Ausgabe: 3
    }

    public static void reassignArray(int[] arr) {
        arr = new int[]{4, 5, 6, 7};
    }
}
```

- **Erklärung:**
  - Neuzuweisung von `arr` beeinflusst nicht `array`
  - `arr` zeigt nun auf ein neues Array
  - `array` in `main` bleibt unverändert

Parameterübergabe in C++

- **C++ unterstützt sowohl:**
  - **Pass by Value** (Standard)
  - **Pass by Reference** (mit `&`)
- **Entwickler können wählen**, wie Parameter übergeben werden

Beispiel in C++: Pass by Value

```
#include <iostream>
void increment(int num) {
    num++;
}

int main() {
    int zahl = 10;
    increment(zahl);
    std::cout << "Zahl nach increment: " << zahl << std::endl; // Ausgabe: 10
    return 0;
}
```

- **Erklärung:**
  - `num` ist eine Kopie von `zahl`
  - Änderungen an `num` beeinflussen `zahl` nicht

Beispiel in C++: Pass by Reference

```cpp
#include <iostream>
void increment(int& num) {
    num++;
}

int main() {
    int zahl = 10;
    increment(zahl);
    std::cout << "Zahl nach increment: " << zahl << std::endl; // Ausgabe: 11
    return 0;
}
```

- **Erklärung:**
  - `num` ist eine Referenz auf `zahl`
  - Änderungen an `num` beeinflussen direkt `zahl`

Beispiel in C++: Zeiger (Pointer)

```
#include <iostream>
void increment(int* num) {
    (*num)++;
}

int main() {
    int zahl = 10;
    increment(&zahl);
    std::cout << "Zahl nach increment: " << zahl << std::endl; // Ausgabe: 11
    return 0;
}
```

- **Erklärung:**
  - `num` ist ein Zeiger auf `zahl`
  - Dereferenzierung mit `*num` ermöglicht Zugriff auf den Wert

Vergleich Java vs. C++

	Java	C++
Primitives	Immer Pass by Value	Standardmäßig Pass by Value
Objekte	Referenz wird by Value übergeben	Nativ Pass by Value oder Reference
Referenzen	Keine echten Referenzen	Echte Pass by Reference möglich

Praktische Implikationen

- **In Java:**
  - Seien Sie vorsichtig bei Methoden, die Objekte manipulieren
  - Nutzen Sie Immutable Objects, um Seiteneffekte zu vermeiden
- **In C++:**
  - Wählen Sie bewusst zwischen Wert- und Referenzübergabe
  - Beachten Sie die Speicherverwaltung bei Zeigern

Zusammenfassung

- **Parameterübergabe ist sprachabhängig**
- **Verstehen der Konzepte ist entscheidend**
  - Für korrektes Programmverhalten
  - Zur Vermeidung von Fehlern
- **Bewusste Anwendung der Übergabemethoden**
  - Führt zu robustem und wartbarem Code

Vergleich der Gleichheit von Objekten (`equals`)

- **Standardverhalten von `equals`**:
  - Vergleicht Referenzen (Speicheradressen), nicht Inhalte
- **Problem**:
  - Objekte mit identischem Inhalt gelten als ungleich
- **Lösung**:
  - Überschreiben der `equals`-Methode, um Inhalte zu vergleichen

Überschreiben der `equals`-Methode

Beispiel für `SimpleDate`:

```
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (!(obj instanceof SimpleDate)) {
        return false;
    }
    SimpleDate other = (SimpleDate) obj;
    return this.day == other.day &&
           this.month == other.month &&
           this.year == other.year;
}
```

- **Vergleich der Instanzvariablen**
- Ermöglicht inhaltlichen Vergleich
- Beachte: Argument vom Typ `Object`.

Verwendung der überschriebenen `equals`-Methode

```
SimpleDate date1 = new SimpleDate(1, 1, 2000);
SimpleDate date2 = new SimpleDate(1, 1, 2000);

System.out.println(date1.equals(date2));  // Ausgabe: true
```

- Objekte werden nun basierend auf ihren Werten als gleich erkannt
- Ohne die überschriebene `equals` Methode wäre die Ausgabe `false`

`equals`-Methode und Listen

- **Listen verwenden `equals`**:
  - Methoden wie `contains` nutzen `equals`, um Objekte zu vergleichen
- **Beispiel**:

```
ArrayList<SimpleDate> dates = new ArrayList<>();
dates.add(date1);

System.out.println(dates.contains(date2));  // Ausgabe: true
```

- Ohne korrektes `equals` würde `contains` `false` zurückgeben

Objekte als Rückgabewerte von Methoden

- **Methoden können Objekte zurückgeben**:
  - Ermöglicht die Erstellung neuer Objekte innerhalb von Methoden
- **Beispiel**:

```java
public SimpleDate afterNumberOfDays(int days) {
    // Berechnung des neuen Datums
    SimpleDate newDate = new SimpleDate(...);
    return newDate;
}
```

- Das ursprüngliche Objekt bleibt unverändert

Beispiel: Berechnung zukünftiger Daten

```
SimpleDate today = new SimpleDate(13, 2, 2020);
SimpleDate futureDate = today.afterNumberOfDays(7);

System.out.println("Heute: " + today);
System.out.println("In einer Woche: " + futureDate);
```

Ausgabe:

```
Heute: 13.2.2020
In einer Woche: 20.2.2020
```

Zusammenfassung

- **Referenzen**:
  - Variablen speichern Referenzen auf Objekte
  - Zuweisung kopiert Referenzen, nicht Objekte
- **`null`-Referenzen**:
  - Können zu `NullPointerException` führen
  - Immer prüfen, bevor Methoden aufgerufen werden
- **`equals`-Methode**:
  - Überschreiben, um Objekte inhaltlich zu vergleichen
  - Wichtig für das Arbeiten mit Collections

Abstraktion und Modularität

- **Abstraktion** ist eines der wichtigsten Konzepte der Programmierung.
- Sie erlaubt uns, komplexe Systeme in überschaubare Einheiten zu zerlegen.

Beispiel:

- Eine Klasse kapselt Details und bietet eine Schnittstelle für die Interaktion.

```
Person person = new Person("John");
person.growOlder();
```

- Die interne Logik der Methode `growOlder` bleibt verborgen.

Kapselung und Informationsverbergen

- **Kapselung** schützt Daten und kontrolliert den Zugriff auf sie.
- Programmierer:innen müssen nicht den gesamten inneren Zustand eines Objekts kennen, sondern nur dessen öffentliche Schnittstellen.

Beispiel:

- Getter und Setter kontrollieren den Zugang zu Objektvariablen.

```
public String getName() {
    return this.name;
}
```

Wiederverwendbarkeit und Modularität

- Durch das Erstellen kleiner, spezialisierter Module wird **Wiederverwendbarkeit** erreicht.
- Module sind in sich geschlossen und können in verschiedenen Kontexten verwendet werden.

Beispiel:

- Eine Klasse `Car` kann in unterschiedlichen Projekten verwendet werden, da sie unabhängig ist.

```
Car myCar = new Car("Toyota", "Corolla");
```

Vererbung und Polymorphie

- **kommt im Detail später in der Vorlesung**
- **Vererbung** ermöglicht das Wiederverwenden und Erweitern bestehender Funktionalitäten.
- **Polymorphie** erlaubt das Verarbeiten verschiedener Objekttypen über eine gemeinsame Schnittstelle.

Beispiel:

```
class Dog extends Animal { }
Animal myDog = new Dog();
```

- Methodenaufrufe passen sich dem Objekttyp an.

Wartbarkeit und Skalierbarkeit

- Der Entwurf von Software sollte **wartbar** und **skalierbar** sein.
- Änderungen in einem Modul dürfen keine weitreichenden Auswirkungen auf andere Teile des Programms haben.

Beispiel:

- Eine korrekt entworfene Klasse kann geändert werden, ohne dass das gesamte System angepasst werden muss.

```java
public void setName(String name) {
    this.name = name;
}
```

Prinzipien der Effizienz und Ressourcenverwaltung

- Die Effizienz eines Programms wird durch intelligente **Ressourcenverwaltung** gesteigert.
- Speicher, Rechenleistung und Zeit sollten durch geeignete Algorithmen optimal genutzt werden.

Beispiel:

- Eine **Speicherverwaltung** kann sicherstellen, dass veraltete Objekte gelöscht werden (Garbage Collection in Java).

Fazit: Prinzipien der Programmierung

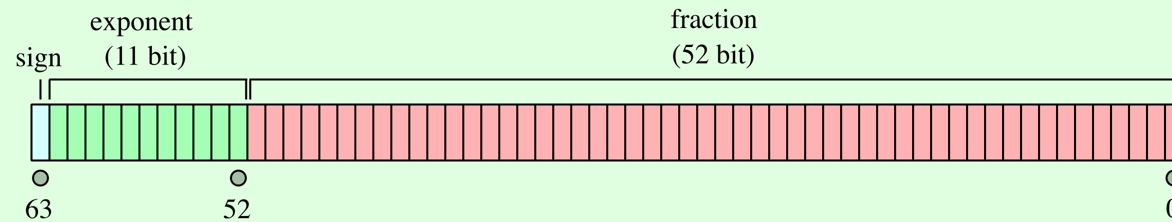
- **Abstraktion, Kapselung, Wiederverwendbarkeit, Vererbung** und **Polymorphie** sind zentrale Prinzipien, die zu robustem und erweiterbarem Code führen.
- Diese Prinzipien fördern die **Lesbarkeit, Wartbarkeit** und **Skalierbarkeit** von Software.

IEEE 754 Darstellung von Gleitkommazahlen

- Java verwendet das **IEEE 754** Standardformat für Gleitkommazahlen.
- **32-Bit** `float` und **64-Bit** `double` Datentypen.
- Darstellung besteht aus:
  - **Vorzeichenbit**
  - **Exponentenfeld**
  - **Mantissenfeld**

Aufbau einer `double`-Zahl (64 Bit)

- **1 Bit** für das Vorzeichen
- **11 Bits** für den Exponenten
- **52 Bits** für die Mantisse



Beispiel: Grenzen von `double` in Java

- **Maximaler Wert**: `Double.MAX_VALUE` $\approx 1.7976931348623157 \times 10^{308}$
- **Minimaler positiver Wert**: `Double.MIN_VALUE` $\approx 4.9 \times 10^{-324}$

```
System.out.println("Max Value: " + Double.MAX_VALUE);
System.out.println("Min Positive Value: " + Double.MIN_VALUE);
```

Limitationen bei Gleitkommazahlen

- **Rundungsfehler**: Nicht alle Dezimalzahlen können exakt dargestellt werden.

```
double a = 0.1 + 0.2;
System.out.println(a); // Ausgabe: 0.30000000000000004
```

- **Vergleich von Gleitkommazahlen**: Direkter Vergleich kann fehlschlagen.

```
if (a == 0.3) {
    // Wird möglicherweise nicht ausgeführt
}
```

Umgang mit Rundungsfehlern

- **Epsilon-Wert** verwenden:

```
double epsilon = 1e-9;
if (Math.abs(a - 0.3) < epsilon) {
    // Werte gelten als gleich
}
```

- **BigDecimal** für präzise Berechnungen:

```
BigDecimal x = new BigDecimal("0.1");
BigDecimal y = new BigDecimal("0.2");
BigDecimal sum = x.add(y);
System.out.println(sum); // Ausgabe: 0.3
```

Behandlung von `null` in anderen Sprachen: C

- In **C** gibt es keinen `null`-Wert wie in Java.
- Zeiger können den Wert `NULL` haben, was bedeutet, dass sie auf nichts zeigen.
- **Dereferenzieren** eines `NULL`-Zeigers führt zu undefiniertem Verhalten und möglicherweise zu Abstürzen.

```c
int *ptr = NULL;
// Zugriff auf *ptr ist unsicher und sollte vermieden werden
```

Behandlung von `null` in anderen Sprachen: Python

- In **Python** gibt es das Objekt `None`, das das Fehlen eines Wertes darstellt.
- Methoden können `None` zurückgeben, um anzuigen, dass kein gültiges Objekt vorhanden ist.
- Überprüfungen auf `None` sind üblich:

```
result = some_function()
if result is not None:
    # Verarbeitung
```

- **AttributeError** tritt auf, wenn man versucht, auf Attribute von `None` zuzugreifen.

Vermeidung von `null`-Problemen in modernen Sprachen

- **Optionale Typen** in Sprachen wie **Kotlin** oder **Swift**:
  - Unterscheidung zwischen nullable und non-nullable Typen.
  - Compiler erzwingt Überprüfungen auf `null`.

```
var name: String? = null
if (name != null) {
    println(name.length)
}
```

Verbindungen zur funktionalen Programmierung

- **Immutabilität**: Objekte werden nicht verändert, sondern es werden neue erzeugt.
  - In den Beispielen wurden Methoden gezeigt, die neue Objekte zurückgeben.
- **First-Class Funktionen**: In funktionalen Sprachen sind Funktionen Werte erster Klasse (first-class citizens), d.h., Funktionen können von anderen Funktionen als Resultat zurückgegeben werden.
  - Java unterstützt seit Version 8 **Lambdas** und **Streams**.
  - Sind in Java Funktionen first-class citizens? Teilweise! (Sie sind immer über funktionale Schnittstellen (wie Function, Consumer, etc.) verpackt)

```java
List<String> names = Arrays.asList("Anna", "Bob", "Charlie");
names.stream()
     .filter(name -> name.startsWith("C"))
     .forEach(System.out::println);
```

Funktionen als Parameter

- In funktionaler Programmierung werden Funktionen häufig als Parameter übergeben.
- Java ermöglicht dies durch **Functional Interfaces**.

```
public void processNumbers(List<Integer> numbers, Function<Integer, Integer>
    for (Integer number : numbers) {
        System.out.println(func.apply(number));
    }
}
```

Rekursion statt Iteration

- Funktionale Programmierung bevorzugt **Rekursion** über Schleifen.
- Beispiel: Fakultät berechnen.

```java
public int factorial(int n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Funktionalen Programmierung mit/in Java

- **Objektorientierte Programmierung** und **funktionale Programmierung** können kombiniert werden.
- Java bietet zunehmend funktionale Programmierkonzepte.
- **Unveränderliche Objekte**, **reine Funktionen** und **höhere Ordnungsfunktionen** sind Beispiele für Überschneidungen.

Garbage Collection

- **Garbage Collection** ist ein automatischer Speicherbereinigungsprozess in Java.
- Er befreit den Speicher von Objekten, die nicht mehr verwendet werden.
- Entwickelnde müssen sich nicht manuell um die Speicherfreigabe kümmern, wie in Sprachen wie C oder C++.

Funktionsweise der Garbage Collection

- **Trigger**: Ein Objekt wird zur Garbage Collection markiert, wenn keine Referenzen mehr darauf existieren.
- **Unklarer Zeitpunkt**: Es ist nicht genau vorhersehbar, wann die Garbage Collection tatsächlich ausgeführt wird.
- Die Java Virtual Machine (JVM) entscheidet basierend auf Speicherbedarf und internen Algorithmen.

Auswirkungen auf die Programmierung

- **Null-Referenzen**: Durch Setzen von Referenzen auf `null` kann man Objekte für die Garbage Collection freigeben.

```
Person person = new Person("Alice");
person = null; // Objekt kann jetzt gesammelt werden
```

- **Best Practices**:
  - Ressourcen wie Streams oder Datenbankverbindungen sollten explizit geschlossen werden.
  - Vermeiden von Speicherlecks durch unbeabsichtigte Referenzen.

Einfache immutable Klasse

```java
public final class Person {
    private final String name;
    private final int age;

    // Konstruktor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getter für die Felder
    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

`final` in Java

Was ist eine `final`-Variable?

- Eine Variable, die **nach der Initialisierung nicht mehr verändert werden kann**.
- Gilt sowohl für primitive Datentypen als auch Referenztypen.

Verhalten:

1. Primitive Datentypen

- Wert kann **nicht erneut zugewiesen** werden.

```
final int x = 10;
x = 20; // Compilerfehler!
```

2. Referenztypen

- **Referenz kann nicht geändert** werden, aber das Objekt **kann modifiziert** werden.

```
final List<String> list = new ArrayList<>();
list.add("Hallo"); // OK
list = new ArrayList<>(); // Compilerfehler!
```

Beispiel: `final` und Konstanten

```
public class Beispiel {
    public static final double PI = 3.14159; // Konstante
    public static void main(String[] args) {
        System.out.println(PI);
        // PI = 3.14; // Compilerfehler!
    }
}
```

Hinweis:

- `final` garantiert Immutabilität der **Referenz**, aber nicht immer des Objekts!

Immutable Klasse mit veränderbaren Feldern

```java
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public final class Company {
    private final String name;
    private final List<String> employees;

    // Konstruktor
    public Company(String name, List<String> employees) {
        this.name = name;
        // Defensivkopie der Liste (**defensive copying**)
        this.employees = new ArrayList<>(employees);
    }

    public String getName() {
        return name;
    }

    // Unmodifiable List zurückgeben
```

Warum ist das wichtig?

1. **Thread-Sicherheit**:
    - Immutable Objekte sind von Natur aus **thread-sicher**, da sie nach ihrer Erstellung nicht mehr verändert werden.
2. **Vorhersagbarkeit**:
    - Einmal erstellte Objekte bleiben **stabil**, was es einfacher macht, sie zu verwenden, da sich ihr Zustand nicht ändert.
3. **Fehlervermeidung**:
    - Unbeabsichtigte **Zustandsänderungen** können verhindert werden, indem der interne Zustand nach der Initialisierung nicht mehr veränderbar ist.

Diskussion: Warum ist es nicht trivial?

- **Mehrere Schritte erforderlich**:
  - Alle Felder müssen `private` und `final` sein.
  - Keine Setter-Methoden erlauben.
  - Bei **mutierbaren Feldern** (wie Listen, Arrays, etc.) muss eine **defensive Kopie** im Konstruktor erstellt werden und beim Zugriff ebenfalls eine **unveränderliche Version** zurückgegeben werden.
- **Versteckte Komplexität**:
  - Objekte mit verschachtelten mutierbaren Feldern (z.B. Listen von Listen) benötigen **mehrere Ebenen** der Absicherung.
  - Das Kopieren großer Datenstrukturen kann **Speicher und Performance kosten**.
  - Die Vermeidung von **Zugriffen während der Konstruktion** (z.B. `this` im Konstruktor weitergeben) erfordert zusätzliche Aufmerksamkeit.
- **Fazit**: Während Immutability Vorteile wie **Thread-Sicherheit** und **Vorhersagbarkeit** bietet, erfordert es eine **sorgfältige Implementierung**, um sicherzustellen, dass das Objekt wirklich unveränderlich bleibt.

Ende Teil 05