Prinzipien der Programmierung

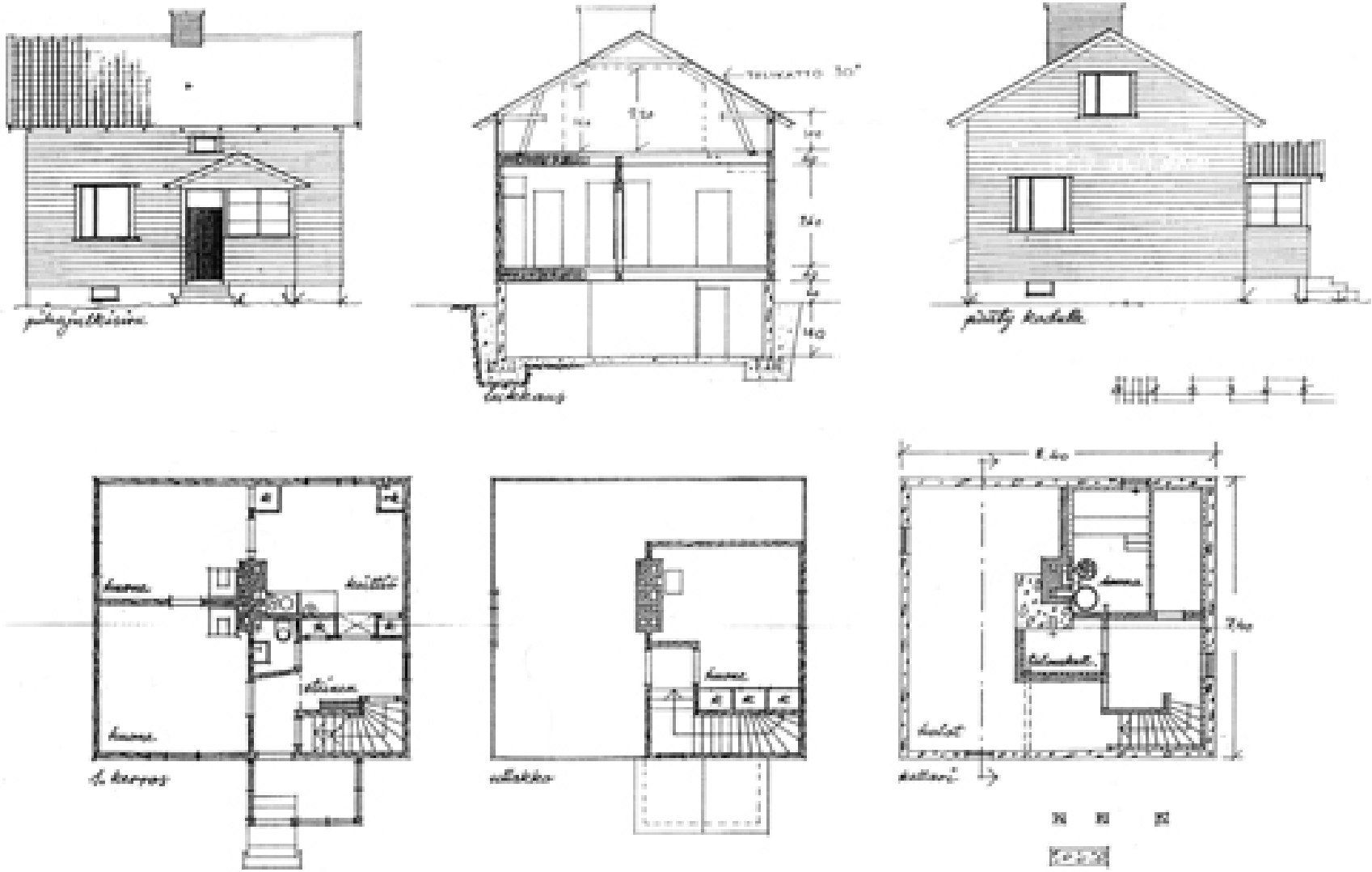
Daniel Merkle

Wintersemester 2024

(Teil 04)

Einführung in die objektorientierte Programmierung

- Was sind Klassen und Objekte?
- Methoden und Instanzvariablen
- Konstruktoren
- Erstellen eigener Klassen und Objekte



Klasse

Objekt

Lernziele

- Klasse, Objekt, Konstruktor, Objektmethoden und Objektvariablen.
- Verstehen, dass eine Klasse Methoden eines Objekts definiert und Werte von Instanzvariablen objektspezifisch sind.
- Wissen, wie man Klassen und Objekte erstellt und verwendet.

Objektorientierte Programmierung

- Abstraktion von Problemen durch **Objekte**.
- Programme bestehen aus kleinen, kooperativen **Objekten**.
- Konzepte eines Problembereichs werden in **separate Einheiten** isoliert.

Klassen und Objekte

- Eine **Klasse** definiert Attribute (Instanzvariablen) und Methoden eines Objekts.
- Ein **Objekt** ist eine Instanz einer Klasse mit spezifischen Attributwerten.
- Objekte werden mit einem **Konstruktor** erstellt.

Beispiel: Konto

```
Account artosAccount = new Account("Arto's account", 100.00);
System.out.println(artosAccount.balance());
```

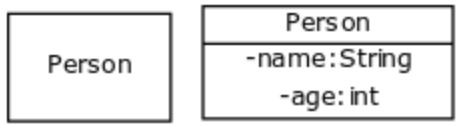
- Klasse `Account`: Repräsentiert ein Bankkonto
- Methoden: `withdraw`, `deposit`, `balance`

Erstellen von Klassen

- Klassen definieren den **Zustand** (Instanzvariablen) und das **Verhalten** (Methoden) von Objekten.
- Klasse `Person`:

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

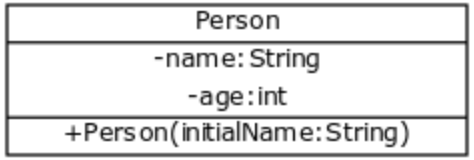
- Instanzvariablen: `name`, `age`



Konstruktoren

- Konstruktoren initialisieren Objekte mit spezifischen Werten.

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



Das Schlüsselwort `this`

- `this` verweist auf das aktuelle Objekt, das die Methode aufruft.
- Es wird verwendet, um auf die **Instanzvariablen** und **Methoden** des aktuellen Objekts zuzugreifen.

Beispiel: Verwendung von `this` im Konstruktor

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

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

- **`this.name`** bezieht sich auf die Instanzvariable `name` der Klasse.
- Der Parameter `name` wird an den Konstruktor übergeben, und mit **`this.name = name`** wird der Parameterwert der Instanzvariablen zugewiesen.

Unterschied ohne `this`

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

    public Person(String name) {
        name = name;  // Kein Bezug zur Instanzvariable
        this.age = 0;
    }
}
```

- Ohne **`this`** wird der Parameter **`name`** nur auf sich selbst zugewiesen.
- Die Instanzvariable **`name`** bleibt unverändert.

Konstruktor: Präsentation in Intellij IDEA

Übung: Ihr erstes Konto

- Erstelle ein Konto mit einem Startguthaben und führe Ein- und Auszahlungen durch.

```
Account artosAccount = new Account("Arto's account", 100.00);
artosAccount.withdraw(20);
System.out.println(artosAccount.balance());
```

Methoden für ein Objekt definieren

- Wie definiert man Methoden in einer Klasse?
- Wie nutzt man Instanzvariablen in Methoden?
- Beispielmethoden in einer Klasse `Person`

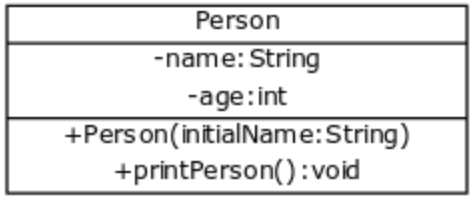
Methode `printPerson`

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

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

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

- Methode gibt den Namen und das Alter der Person aus.



Methode aufrufen

```java
public class Main {
    public static void main(String[] args) {
        Person ada = new Person("Ada");
        ada.printPerson();
    }
}
```

- Die Methode `printPerson()` wird auf dem Objekt `ada` aufgerufen.
- Ausgabe: "Ada, age 0 years"

Quiz Fehler beim Methodenaufruf `Student`, 1-intro

Übung: Whistle

- Erstelle die Klasse `Whistle` mit einem Attribut `sound`.
- Definiere die Methode `sound()`:

```
Whistle duckWhistle = new Whistle("Kvaak");
duckWhistle.sound();
```

- Ausgabe: "Kvaak"

Instanzvariablen in Methoden ändern

- Hinzufügen der Methode `growOlder()`:

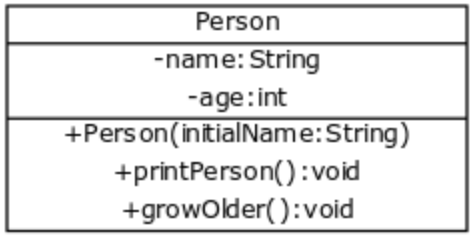
```
public class Person {
    private String name;
    private int age;

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

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

    // growOlder() Methode wurde hinzugefügt
    public void growOlder() {
        this.age = this.age + 1;
    }
}
```

- Ändert den Zustand des Objekts, indem das Alter erhöht wird.



Bedingte Anweisungen in Methoden

- `growOlder()` Methode mit einer Bedingung:

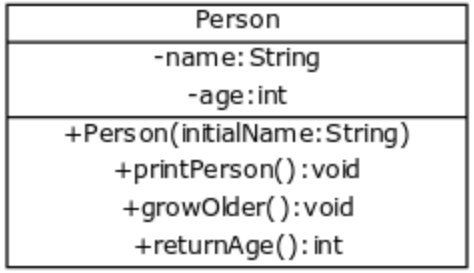
```
public void growOlder() {
    if (this.age < 30) {
        this.age = this.age + 1;
    }
}
```

- Verhindert, dass das Alter größer als 30 wird.

Die Methode `returnAge()` in der Klasse `Person`

```java
public int returnAge() {
    return this.age;
}
```

- Diese Methode gibt das Alter der Person zurück.



Objektzustand vor und nach Methode

```
ada.growOlder();
ada.printPerson();  // Ausgabe: Ada, age 1 years
```

- Das Alter von Ada wurde durch `growOlder()` geändert.

Beispiel: Hinzufügen von Getter- und Setter-Methoden

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

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

    // Getter für name
    public String getName() {
        return this.name;
    }

    // Setter für name
    public void setName(String name) {
        this.name = name;
    }

    // Getter für age
    public int getAge() {
        return this.age;
    }

    // Setter für age
    public void setAge(int age) {
        if (age >= 0) {
            this.age = age;  // Das Alter kann nicht negativ sein
        }
```

Kapselung

- **Kapselung** ist ein grundlegendes Prinzip der objektorientierten Programmierung (OOP).
- Es bedeutet, dass die **Daten eines Objekts privat** gehalten werden und nur über **öffentliche Methoden** darauf zugegriffen wird.
- Dies schützt die Daten vor ungewolltem Zugriff und Modifikation.

Getter-Methoden

- Eine **Getter-Methode** wird verwendet, um den Wert einer privaten Instanzvariablen zurückzugeben.

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

- Getter bieten **nur Lesezugriff** auf die privaten Variablen des Objekts.

Setter-Methoden

- Eine **Setter-Methode** wird verwendet, um den Wert einer privaten Instanzvariablen zu ändern.

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

- Setter bieten **kontrollierten Schreibzugriff**, oft mit Validierung (z.B. kein negatives Alter).

Beispiel: Verwendung von Getter und Setter

```
public class Main {
    public static void main(String[] args) {
        Person p1 = new Person("Anna");
        p1.setAge(25);  // Setzt das Alter auf 25
        System.out.println(p1.getName() + " is " + p1.getAge() + " years old.");
    }
}
```

- **Setter** setzt den Namen und das Alter, **Getter** gibt die Werte zurück.
- Ausgabe: "Anna is 25 years old."

Kapselung und Datenintegrität

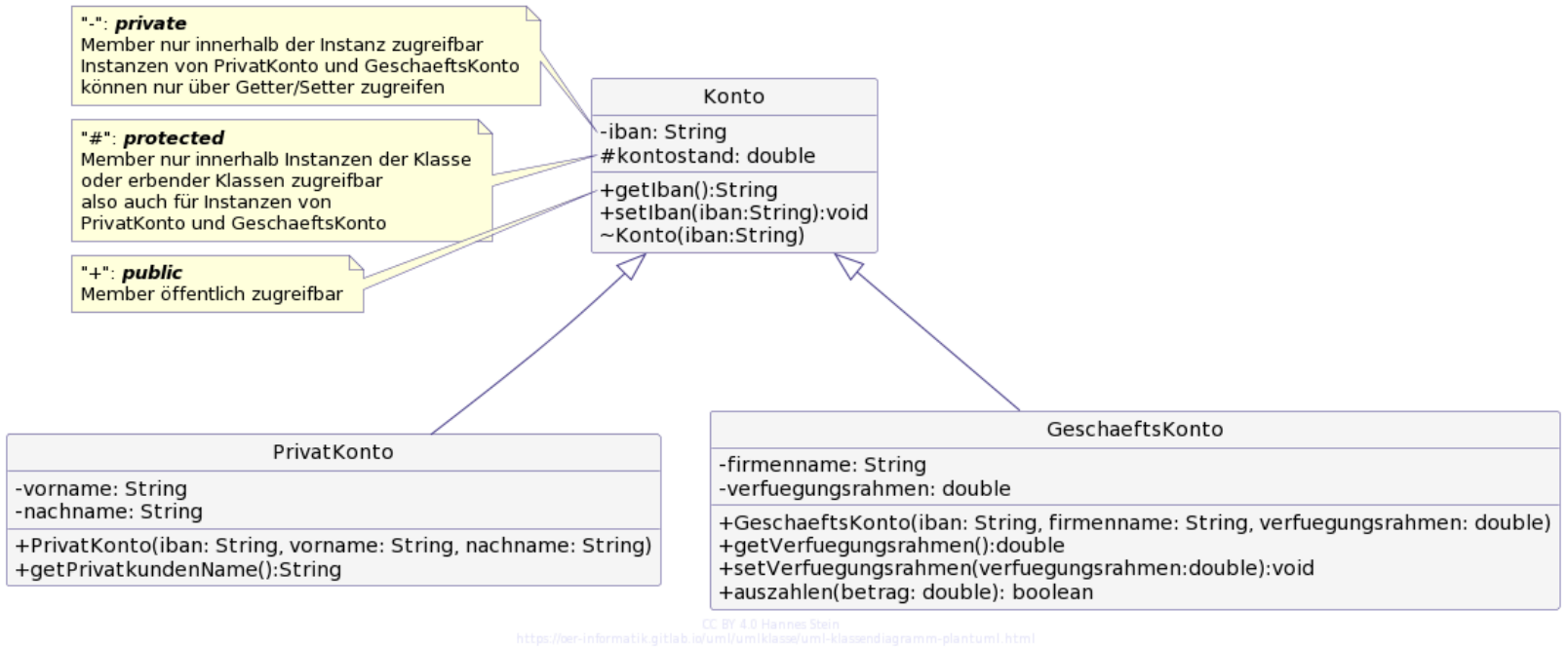
- Durch die Verwendung von **Getter- und Setter-Methoden** wird sichergestellt, dass:
  - Instanzvariablen nur **kontrolliert** geändert werden können.
  - **Fehlerhafte oder ungültige Werte** (z.B. negatives Alter) verhindert werden.

```
public void setAge(int age) {
    if (age >= 0) {
        this.age = age;
    }
}
```

- In diesem Beispiel wird sichergestellt, dass das Alter nur auf gültige Werte gesetzt wird.

Zusammenfassung: Getter/Setter/Kapselung

- **Kapselung** schützt die internen Daten eines Objekts und macht sie nur über **öffentliche Methoden** zugänglich.
- **Getter- und Setter-Methoden** bieten sicheren und kontrollierten Zugriff auf die Instanzvariablen.



Klassen-diagramme

Sprachliche Beschreibung eines Systems

Angenommen, wir möchten ein System modellieren, das eine **Bibliothek** und ihre **Bücher** beschreibt:

- Eine Bibliothek hat einen **Namen** und eine **Sammlung von Büchern**.
- Jedes **Buch** hat einen **Titel**, einen **Autor** und eine **ISBN**.
- Ein Buch kann an einen **Benutzer** ausgeliehen werden.
- Ein **Benutzer** hat einen **Namen** und eine **Benutzernummer**.
- Ein Benutzer kann **mehrere Bücher** ausleihen, aber jedes Buch kann nur von einem Benutzer gleichzeitig ausgeliehen werden.

Vom Text zur Beziehung

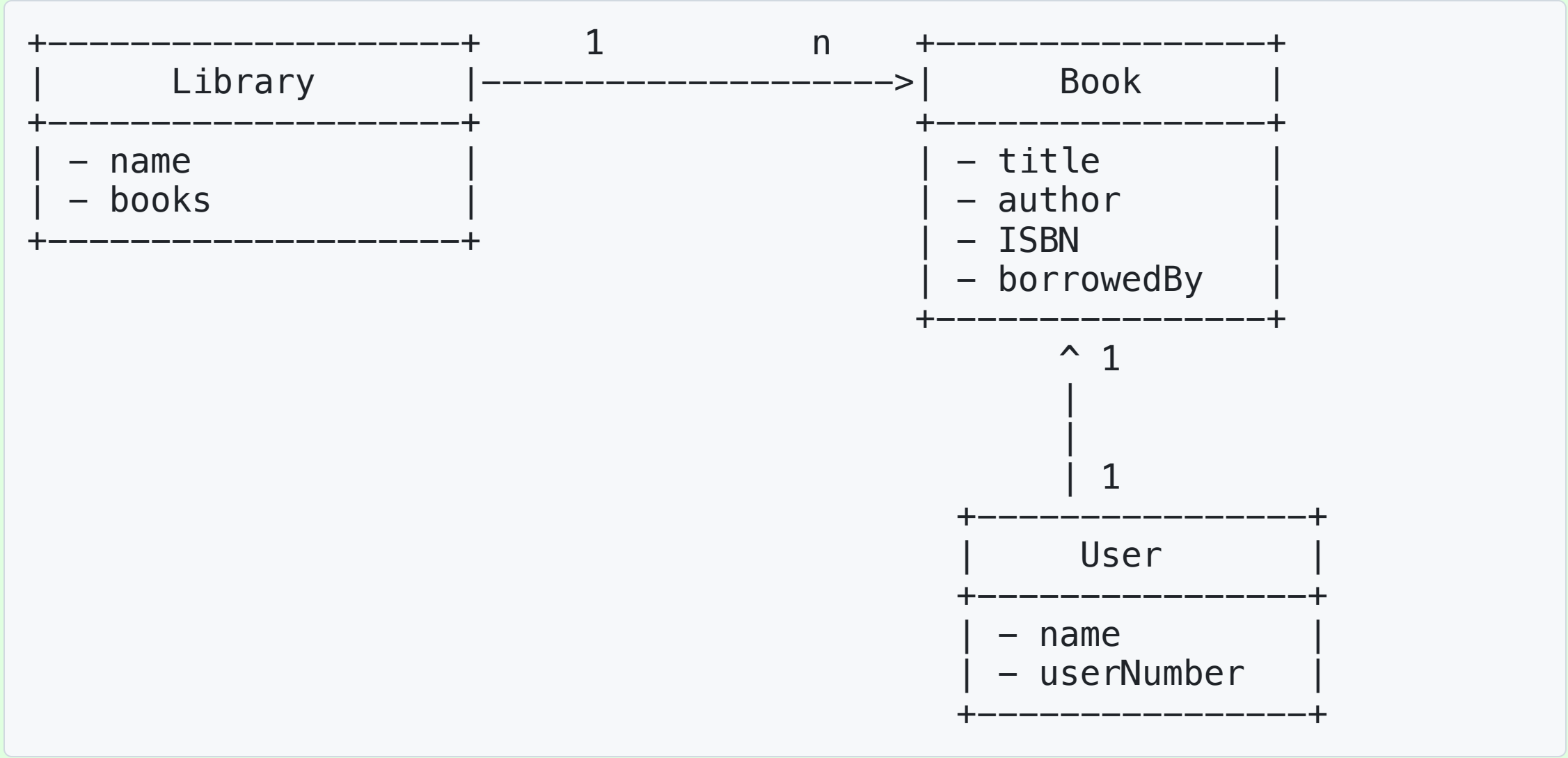
Wichtige Objekte

- **Bibliothek**: Hauptobjekt, enthält Bücher.
- **Buch**: Teil der Bibliothek, kann ausgeliehen werden.
- **Benutzer**: Kann Bücher ausleihen.

Beziehungen

- Eine **Bibliothek** hat eine **1:n-Beziehung** zu Büchern (eine Bibliothek hat viele Bücher).
- Ein **Buch** hat eine **1:1-Beziehung** zu einem Benutzer (ein Buch kann zu einem bestimmten Zeitpunkt nur von einem Benutzer ausgeliehen werden).
- Ein **Benutzer** hat eine **1:n-Beziehung** zu Büchern (ein Benutzer kann mehrere Bücher ausleihen).

Klassendiagramm (vereinfacht)

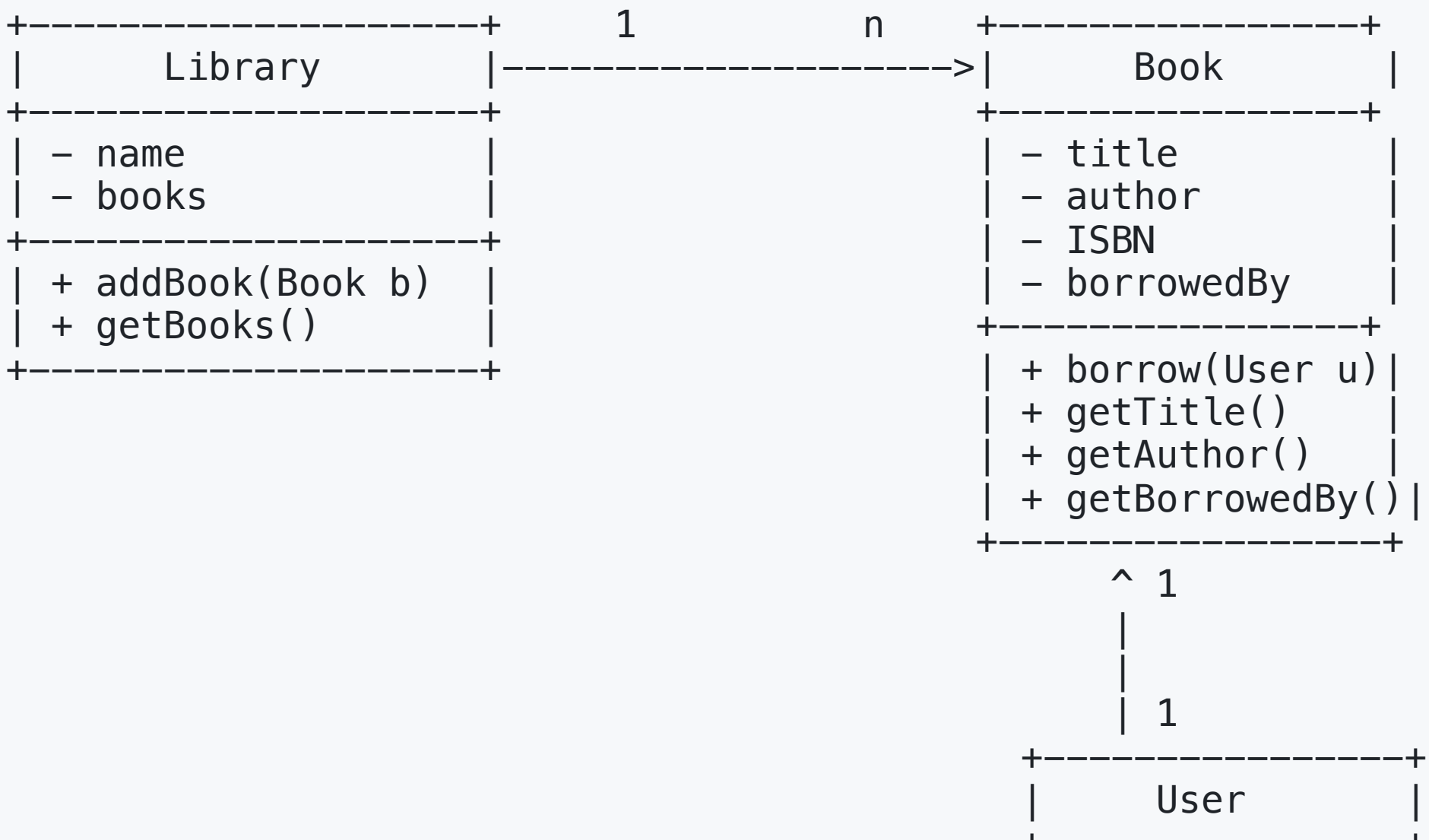


Klassendiagramm mit öffentlichen Methoden

- **Bibliothek** (Library)
- **Buch** (Book)
- **Benutzer** (User)

Wir erweitern das vorherige Klassendiagramm um **öffentliche Methoden**, die durch das "`+`"-Zeichen dargestellt werden.

Vom Text zum Klassendiagramm (vereinfacht)



Öffentliche Methoden

- `+` zeigt, dass die Methode öffentlich ist.
- Methoden, die von anderen Klassen verwendet werden können, um auf Attribute und Funktionen der Objekte zuzugreifen.
- **Getter**- und **Setter**-Methoden sind üblich, um den Zugriff auf private Attribute zu kontrollieren.

Beispiel: Bibliothek (Library)

```java
public class Library {
    private String name;
    private ArrayList<Book> books;

    // Öffentliche Methode zum Hinzufügen eines Buches
    public void addBook(Book book) {
        this.books.add(book);
    }

    // Öffentliche Methode zum Abrufen der Liste von Büchern
    public ArrayList<Book> getBooks() {
        return this.books;
    }
}
```

- **`addBook(Book book)`**: Fügt ein Buch zur Bibliothek hinzu.
- **`getBooks()`**: Gibt die Liste der Bücher in der Bibliothek zurück.

Beispiel: Buch (Book)

```java
public class Book {
    private String title;
    private String author;
    private User borrowedBy;

    // Methode zum Ausleihen des Buches
    public void borrow(User user) {
        this.borrowedBy = user;
    }

    // Getter für Titel und Autor
    public String getTitle() {
        return this.title;
    }

    public String getAuthor() {
        return this.author;
    }
}
```

Beispiel: Benutzer (User)

```java
public class User {
    private String name;
    private String userNumber;

    // Getter für Name und Benutzernummer
    public String getName() {
        return this.name;
    }

    public String getUserNumber() {
        return this.userNumber;
    }
}
```

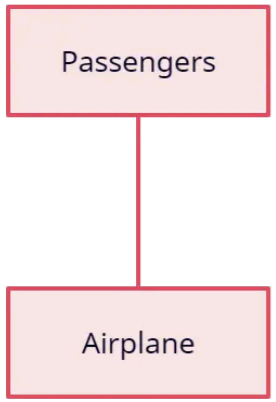
- **`getName()`** und **`getUserNumber()`**: Öffentliche Methoden, um die privaten Attribute `name` und `userNumber` des Benutzers abzufragen.

Entscheidungen im Design

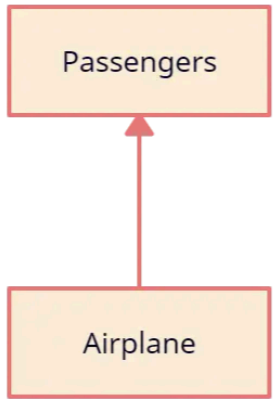
- **Kapselung**: Die Attribute sind **privat** und nur über öffentliche **Getter- und Setter-Methoden** zugänglich.
- **Kardinalitäten**: Beziehungen zwischen Klassen werden durch **1:n** und **1:1** Kardinalitäten dargestellt.
- **Methoden**: Öffentliche Methoden werden zur Steuerung des Zugriffs auf die Daten verwendet (z.B. `addBook()`, `borrow()`).

Klassen-Diagramm

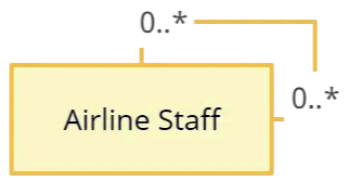
- **Aggregation**: Die Bibliothek hat viele Bücher, daher repräsentiert die **1:n-Beziehung** dies als Aggregation.
- **Assoziation**: Bücher haben eine Assoziation zu Benutzern durch die **borrowedBy**-Referenz, was bedeutet, dass jedes Buch von einem Benutzer ausgeliehen werden kann.
- **Kardinalitäten**: Die Kardinalitäten an den Verbindungen zeigen an, wie viele Objekte in Beziehung stehen. Beispielsweise steht „n“ für viele und „1“ für eine Instanz.



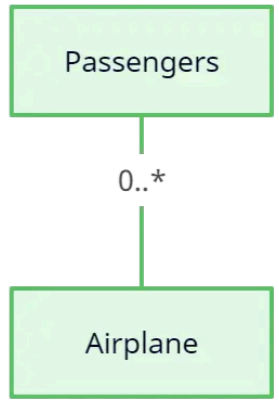
Association



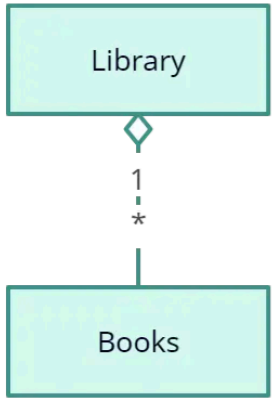
Directed Association



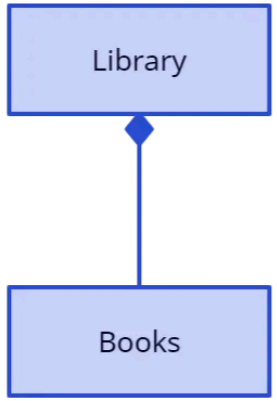
Reflexive Association



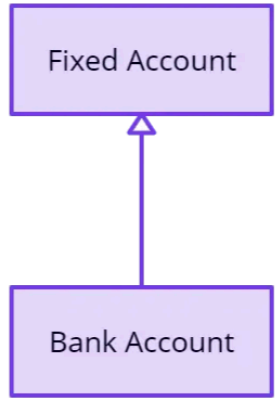
Multiplicity



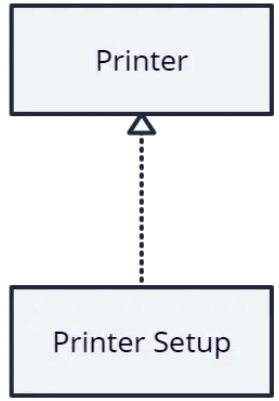
Aggregation



Composition



Inheritance



Realization

UML-Relationen: Unvollständiger Überblick

Unvollständige Liste von UML-Relationen:

- Assoziation
- Aggregation
- Komposition
- Multiplizität
- Vererbung

Assoziation

- **Definition:** Allgemeine Beziehung zwischen zwei Klassen.
- **Beispiel:** Ein **Kunde** kann eine **Bestellung** aufgeben, aber beide Klassen existieren unabhängig voneinander.

```
Kunde  ------  Bestellung
```

Aggregation

- **Definition:** "Ganzes-Teil"-Beziehung, bei der das Teil unabhängig vom Ganzen existiert.
- **Beispiel:** Ein **Team** besteht aus **Spielern**, aber die Spieler können unabhängig vom Team existieren.

```
Team ◇------ Spieler
```

Komposition

- **Definition:** Stärkere Form der Aggregation, bei der das Teil ohne das Ganze nicht existieren kann.
- **Beispiel:** Ein **Haus** besteht aus **Räumen**, aber ohne das Haus existieren die Räume nicht.

```
Haus ◆------- Raum
```

Multiplizität

- **Definition:** Gibt an, wie viele Instanzen einer Klasse mit einer Instanz einer anderen Klasse in Beziehung stehen können.
- **Beispiel:** Ein **Kunde** kann 0 bis viele **Bestellungen** haben, aber jede Bestellung gehört zu genau einem Kunden.

```
Kunde  1------0..* Bestellung
```

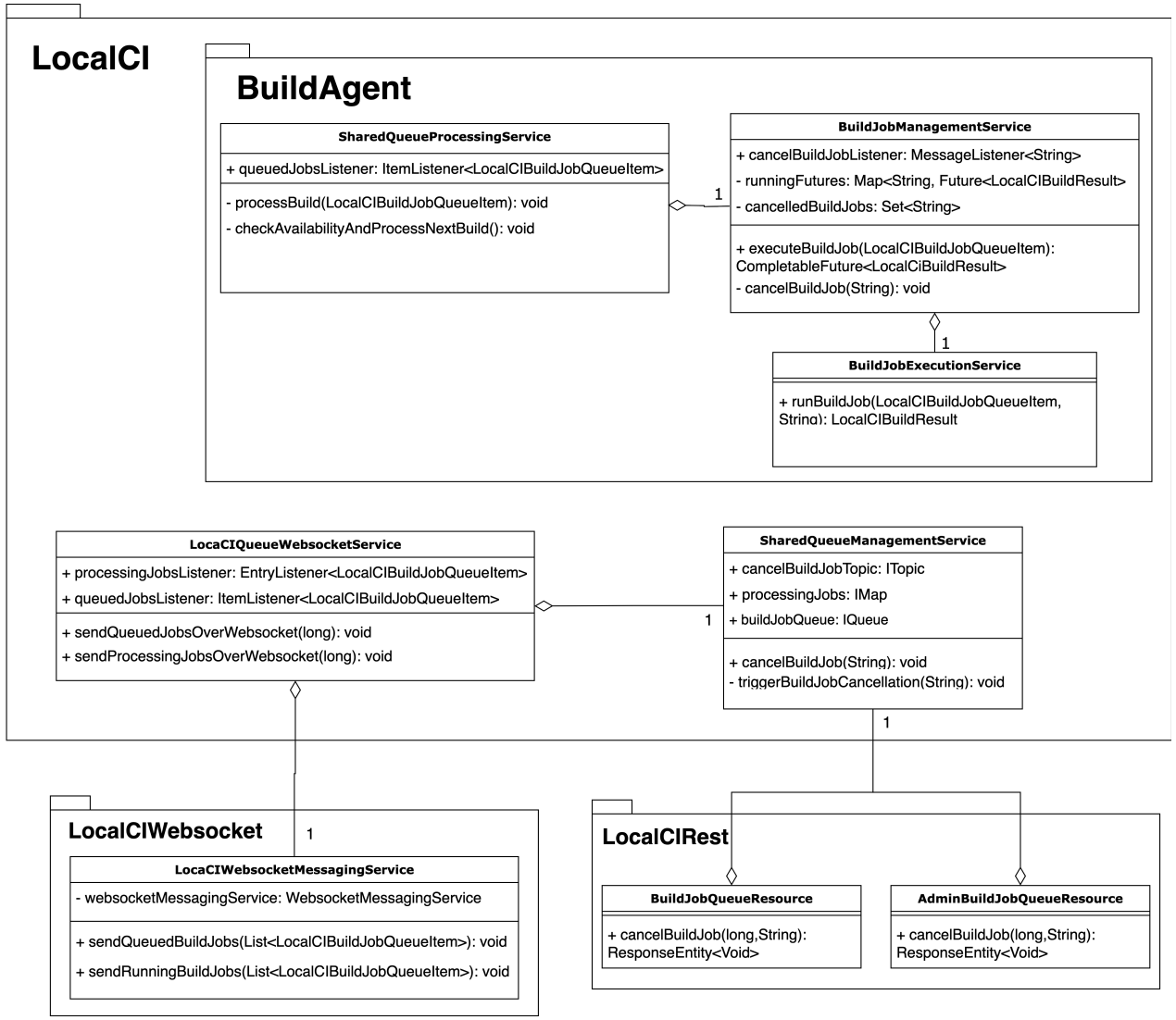
Vererbung

- **Definition:** Eine Klasse erbt Eigenschaften und Methoden von einer Oberklasse.
- **Beispiel:** Ein **Auto** ist eine spezifische Form eines **Fahrzeugs**.

```
Fahrzeug △------ Auto
```

Zusammenfassung der Designentscheidungen

- **Klassenidentifikation**: Jede Klasse repräsentiert ein reales oder konzeptionelles Objekt aus der Problembeschreibung.
- **Beziehungen und Kardinalitäten**: Die Diagrammbeziehungen spiegeln wider, wie viele Objekte zueinander in Beziehung stehen (z.B. eine Bibliothek hat viele Bücher).
- **Zustandsinformationen**: Jedes Objekt enthält Informationen über seinen Zustand, z.B. `Book` enthält `borrowedBy`, um anzugeben, welcher Benutzer das Buch ausgeliehen hat.



von Artemis System Design

hier: **Build Job Cancellation** The build job cancellation mechanism is a crucial feature of the CI system. It allows users to cancel build jobs that are taking too long or are no longer needed. The following diagram shows an overview of the components in the build job cancellation mechanism

Der Zweck von UML-Diagrammen

- UML (Unified Modeling Language) bietet eine **grafische Darstellung** von Systemen in der Softwareentwicklung.
- Ziel ist es, die **Struktur** und das **Verhalten** des Systems klar und verständlich darzustellen.

Kommunikation von Systemstrukturen

- UML-Diagramme helfen Entwicklern, die **gewünschte Struktur** eines Systems zu kommunizieren.
- Sie machen komplexe Ideen einfacher und schaffen eine **gemeinsame Basis** für alle Beteiligten (Entwickler, Designer, Kunden).
- **Beispiel**: Ein Klassendiagramm zeigt, wie Objekte miteinander interagieren und welche Beziehungen sie haben.

Vereinfachung komplexer Konzepte

- Systeme in der Softwareentwicklung können sehr komplex sein. UML-Diagramme vereinfachen diese Konzepte durch:
    - **Visualisierung**: Komplexe Abhängigkeiten und Beziehungen werden klar sichtbar.
    - **Abstraktion**: Nur die wesentlichen Elemente des Systems werden gezeigt.
- UML-Diagramme dienen als **Werkzeug zur Kommunikation** und helfen, **komplexe Ideen zu vereinfachen**, sodass alle Beteiligten das gleiche Verständnis vom System haben.

Teaser

Zugriff auf Attribute in Vererbung

- **private** Attribute und Methoden sind in erbenden Klassen **nicht** zugreifbar.
- Eine Instanz der erbenden Klasse kennt das private Attribut, kann aber nicht darauf zugreifen.

Beispiel: Private Attribute in Vererbung

```
class Konto {
    private String iban;
    // Getter und Setter für iban
}
class PrivatKonto extends Konto {
    // Kann nicht direkt auf iban zugreifen
}
```

- **PrivatKonto** kann nur über Getter/Setter auf `iban` zugreifen, da es private ist.

Das Schlüsselwort protected

- **protected** (`#`) Attribute und Methoden sind für die eigene Klasse und alle abgeleiteten Klassen zugreifbar.
- Geschützt vor externem Zugriff.

Beispiel: Protected Attribute

```
class Konto {
    protected double kontostand;
}
class GeschaeftsKonto extends Konto {
    // Kann direkt auf kontostand zugreifen
}
```

- **GeschaeftsKonto** kann ohne Getter und Setter auf `kontostand` zugreifen, da es protected ist.

Ein bisschen Wiederholung. Keine Zeit? 15 Folien vorwärts.

Übung: Tür

- Erstelle die Klasse `Door`.
- Definiere die Methode `knock()`:

```
Door door = new Door();
door.knock();  // Ausgabe: "Who's there?"
```

Rückgabe eines Werts von einer Methode

- Methoden können Werte zurückgeben.
- Das Schlüsselwort **void** bedeutet, dass eine Methode keinen Wert zurückgibt.
- Wenn ein Wert zurückgegeben wird, wird der Typ des Rückgabewerts spezifiziert.

Beispiel: Rückgabewert in einer Methode

```
public class Teacher {
    public int grade() {
        return 10;
    }
}
```

- Die Methode `grade()` gibt den Wert `10` zurück.
- Typ der zurückgegebenen Variable: `int` .

Verwenden des Rückgabewerts

```
Teacher teacher = new Teacher();
int grading = teacher.grade();
System.out.println("The grade received is " + grading);
```

- Der Rückgabewert wird einer Variablen zugewiesen.
- Ausgabe: "The grade received is 10".

Beispiel: Berechnung mit Rückgabewerten

```
Teacher first = new Teacher();
Teacher second = new Teacher();
double average = (first.grade() + second.grade()) / 2.0;
System.out.println("Grading average " + average);
```

- Rückgabewerte können Teil von Berechnungen sein.
- Ausgabe: "Grading average 10.0".

Methoden mit verschiedenen Rückgabetypen

- Methoden können verschiedene Datentypen zurückgeben:

```
public String getName() {
    return "John";
}
public double getBalance() {
    return 732.50;
}
```

- Rückgabetypen: `String`, `double`, `int`, etc.

Methoden mit booleschen Rückgabewerten

- Methoden können auch `boolean` zurückgeben:

```
public boolean isOfLegalAge() {
    return this.age >= 18;
}
```

- Gibt `true` oder `false` zurück.

Beispiel: Rückgabewerte in Bedingungen

```
if (pekka.isOfLegalAge()) {
    System.out.println("Pekka is of legal age");
}
```

- Rückgabewerte können in Bedingungen verwendet werden.

Quiz, Rückgabewerte von Mewthoden, `Counter`, 1-intro

Methodenparameter

- Methoden können Parameter akzeptieren, um ihnen Werte zu übergeben.
- Parameter können verwendet werden, um Instanzvariablen zu ändern.

Beispiel: BMI-Berechnung

```
public void setHeight(int newHeight) {
    this.height = newHeight;
}

public void setWeight(int newWeight) {
    this.weight = newWeight;
}

public double bodyMassIndex() {
    double heightPerHundred = this.height / 100.0;
    return this.weight / (heightPerHundred * heightPerHundred);
}
```

- Parameter `newHeight` und `newWeight` setzen die Instanzvariablen `height` und `weight`.
- Die Methode `bodyMassIndex()` berechnet den Body-Mass-Index (BMI).

Verwenden der Parameter

```
Person matti = new Person("Matti");
matti.setHeight(180);
matti.setWeight(86);
System.out.println(matti.getName() + ", BMI: " + matti.bodyMassIndex());
```

- Übergabe der Parameter `180` und `86` an `setHeight` und `setWeight`.
- Ausgabe: "Matti, BMI: 26.54".

Aufrufen einer internen Methode

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

- Eine Methode kann eine andere Methode im selben Objekt aufrufen.
- `this.bodyMassIndex()` ruft die Methode `bodyMassIndex` innerhalb der Klasse auf.

Übung: Statistik

- Schreibe eine Klasse `Statistics`, die Zahlen verarbeitet:

```
statistics.addNumber(3);
statistics.addNumber(5);
System.out.println("Count: " + statistics.getCount());  // Ausgabe: 2
```

- Verwende Methodenparameter, um Werte an die Statistikklasse zu übergeben.

Objektorientierte Programmierung (OOP) - Grundkonzepte

- OOP basiert auf der Idee, dass die Welt aus **Objekten** besteht.
- Ein **Objekt** ist eine Einheit mit Zustand (Daten) und Verhalten (Funktionalitäten).
- OOP stellt einen paradigmatischen Ansatz zur Problemlösung dar.

Klassen und Objekte

- Eine **Klasse** ist eine abstrakte Definition eines Objekts.
- Ein **Objekt** ist eine konkrete Instanz einer Klasse.
- Analogie: Ein **Prototyp** ist wie eine Blaupause, ein Objekt ist die physische Manifestation dieser Blaupause.

Beispiel: Ein **Haus** kann als Klasse definiert werden. Jedes physische Haus (mit variierenden Farben, Türen etc.) ist ein Objekt dieser Klasse.

Abstraktion

- **Abstraktion** bedeutet, komplexe Realitäten zu vereinfachen.
- In OOP wird die Abstraktion genutzt, um die wesentlichen Eigenschaften eines Objekts zu definieren und Unwichtiges auszublenden.

Beispiel: Ein **Auto** kann durch die Klassen "Motor", "Räder" und "Karosserie" repräsentiert werden. Wir ignorieren Details wie Schrauben oder Lackfarbe.

Kapselung

- **Kapselung** ist das Verbergen der inneren Funktionsweise eines Objekts.
- Nur wichtige Schnittstellen werden für die Außenwelt zugänglich gemacht.

Beispiel: Ein **Buch** hat Seiten und einen Inhalt. Man sieht nur den Buchumschlag, der Inhalt wird nicht direkt manipuliert.

Vererbung

- **Vererbung** erlaubt es, von einer bestehenden Klasse eine neue Klasse zu erzeugen, die deren Eigenschaften erbt.
- Vererbung ermöglicht eine **Hierarchie** von Klassen, wobei gemeinsame Merkmale in der Oberklasse definiert werden.

Beispiel: Eine **"Person"**-Klasse könnte allgemeine Eigenschaften wie Name und Alter haben. Eine **"Student"**-Klasse erbt diese und fügt spezialisierte Eigenschaften wie Studienfach hinzu.

Polymorphismus

- **Polymorphismus** erlaubt es, dass verschiedene Objekte auf die gleiche Nachricht unterschiedlich reagieren.
- Durch Polymorphismus können mehrere Formen eines Objekts unter derselben Schnittstelle existieren.

Beispiel: Ein **Künstler** kann sowohl ein Maler als auch ein Bildhauer sein. Der Befehl "Erstellen" könnte in beiden Fällen zu einem anderen Ergebnis führen.

Objekte und Kommunikation

- Objekte **kommunizieren** durch den Austausch von Nachrichten.
- Eine Nachricht enthält Informationen, auf die das Objekt reagiert.

Beispiel: In einem **Büro**-Szenario fordert ein Manager Informationen von einem Mitarbeiter an (Nachricht senden), und der Mitarbeiter antwortet mit den angeforderten Daten.

Dynamische Systeme und OOP

- OOP wird oft für **dynamische Systeme** verwendet, bei denen Objekte erstellt, geändert und zerstört werden.
- Objekte haben **Lebenszyklen**, die durch die Interaktion mit anderen Objekten bestimmt werden.

Beispiel: Ein **Ökosystem** besteht aus Pflanzen, Tieren und Umwelteinflüssen. Jedes Objekt interagiert dynamisch mit anderen und beeinflusst deren Verhalten.

Zusammenfassung

- OOP bietet einen **modularen, abstrahierten** Ansatz zur Problemlösung.
- Durch Abstraktion, Kapselung, Vererbung und Polymorphismus können komplexe Systeme vereinfacht und strukturiert werden.

Lernziele (Objekte in einer Liste)

- Sie können Objekte zu einer Liste hinzufügen.
- Sie können über Objekte einer Liste iterieren.

Liste von Zeichenketten

```
ArrayList<String> names = new ArrayList<>();
names.add("Betty Jennings");
names.add("Betty Snyder");
names.add("Frances Spence");
// und mehr
```

- Eine `ArrayList<String>` enthält Zeichenketten.

Iteration über eine Liste

```
for (String name: names) {
    System.out.println(name);
}
```

- Mit einer **for-each-Schleife** wird über die Elemente der Liste iteriert.

Objekte zu einer Liste hinzufügen

```
ArrayList<Person> persons = new ArrayList<>();
persons.add(new Person("John"));
persons.add(new Person("Matthew"));
```

- Eine Liste kann Objekte enthalten, wie `Person` -Objekte.

Benutzereingaben zu einer Liste hinzufügen

```
while (true) {
    System.out.print("Enter a name: ");
    String name = scanner.nextLine();
    if (name.isEmpty()) {
        break;
    }
    persons.add(new Person(name));
}
```

- Benutzereingaben können verwendet werden, um Objekte zur Liste hinzuzufügen.

Mehrere Konstruktorparameter

```
persons.add(new Person(name, age));
```

- Beim Hinzufügen von Objekten können mehrere Parameter übergeben werden, z.B. Name und Alter.

Iteration und gefilterte Ausgabe

```
for (Person person: persons) {
    if (person.getAge() >= ageLimit) {
        System.out.println(person);
    }
}
```

- Es ist möglich, während der Iteration bestimmte Bedingungen auf Objekte anzuwenden.

Zusammenfassung

- Listen speichern Objekte.
- Mit Schleifen (for-each, while) iteriert man über Listen.
- Objekte können durch Benutzereingaben dynamisch zur Liste hinzugefügt werden.

Lernziele (Dateien und das Lesen von Daten)

- Sie wiederholen das Lesen von Tastatureingaben.
- Sie wissen, was eine Datei und ein Dateisystem sind.
- Sie können ein Programm erstellen, das Daten aus einer Datei liest.

Tastatureingaben wiederholen

```java
Scanner scanner = new Scanner(System.in);
while (true) {
    String line = scanner.nextLine();
    if (line.equals("end")) {
        break;
    }
}
```

- Der **Scanner** liest Eingaben von der Tastatur.
- Die Eingabe endet, wenn der Benutzer "end" eingibt.

Verarbeitung von Benutzereingaben als Ganzzahl

```java
int number = Integer.valueOf(scanner.nextLine());
```

- Zeichenfolgen können in andere Datentypen wie **Integer** konvertiert werden.

Dateien und das Dateisystem

- **Dateien** sind Datensammlungen, die auf Computern gespeichert werden.
- Das **Dateisystem** verwaltet den Speicherort von Dateien auf der Festplatte.

Beispiel: Textdateien, Bilddateien, Musikdateien.

Erstellen einer neuen Datei

- Dateien können in der Entwicklungsumgebung erstellt werden.
- Beispiel: Erstellen Sie `file.txt` in IntelliJ.

```
Hello, world!
```

Lesen einer Datei mit Scanner

```java
import java.util.Scanner;
import java.nio.file.Paths;

try (Scanner scanner = new Scanner(Paths.get("file.txt"))) {
    while (scanner.hasNextLine()) {
        System.out.println(scanner.nextLine());
    }
} catch (Exception e) {
    System.out.println("Error: " + e.getMessage());
}
```

- Der **Scanner** kann verwendet werden, um Zeilen aus einer Datei zu lesen.
- Fehler werden mit **try-catch** behandelt.

Daten aus einer Datei in eine Liste lesen

```
ArrayList<String> lines = new ArrayList<>();

try (Scanner scanner = new Scanner(Paths.get("file.txt"))) {
    while (scanner.hasNextLine()) {
        lines.add(scanner.nextLine());
    }
}
System.out.println("Total lines: " + lines.size());
```

- Dateien können Zeile für Zeile in eine **ArrayList** eingelesen werden.

Arbeiten mit CSV-Dateien

```
String[] parts = line.split(",");
String name = parts[0];
int age = Integer.valueOf(parts[1]);
```

- Daten aus **CSV-Dateien** (comma-separated values) können einfach mit `split()` aufgeteilt und verarbeitet werden.

Einlesen einer Datei mit 3 Spalten

- Datei enthält Daten in 3 Spalten, getrennt durch **Semikolon** (`;`).
- Jede Zeile wird mit der Methode `split` in ein `String[]` (String-Array) aufgeteilt.

Beispielinhalt der Datei

```
Name;Alter;Stadt
Anna;25;Berlin
Bob;30;München
Charlie;22;Hamburg
```

- **Spalte 1**: Name
- **Spalte 2**: Alter
- **Spalte 3**: Stadt

Jede Zeile wird in drei Teile aufgeteilt.

Einlesen der Datei mit `split`

```java
import java.nio.file.Paths;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        try (Scanner scanner = new Scanner(Paths.get("data.txt"))) {
            while (scanner.hasNextLine()) {
                String line = scanner.nextLine();
                String[] parts = line.split(";");  // Zeile wird in Teile aufgesplittet
                System.out.println("Name: " + parts[0]);
                System.out.println("Alter: " + parts[1]);
                System.out.println("Stadt: " + parts[2]);
            }
        } catch (Exception e) {
            System.out.println("Fehler beim Lesen der Datei: " + e.getMessage());
        }
    }
}
```

- `line.split(";")` teilt die Zeile anhand des Semikolons in ein `String[]` (String-Array) auf.
- `parts[0]` enthält den Namen, `parts[1]` das Alter, und `parts[2]` die Stadt.

Was ist `String[]`?

- `String[]` ist ein **Array** von Strings.
  - Ein Array ist eine **feste Sammlung** von Elementen.
  - Die Größe des Arrays wird beim Erstellen festgelegt und kann nicht geändert werden.
- Beispiel für ein **String-Array**:

```
String[] parts = {"Anna", "25", "Berlin"};
```

- Zugriff auf die Elemente erfolgt über **Indizes**:
  - `parts[0]` gibt den ersten Wert zurück (z.B. "Anna").
  - `parts[1]` gibt den zweiten Wert zurück (z.B. "25").

Unterschied zu `ArrayList<String>`

- `String[]`:
  - Ein **Array** hat eine feste Größe. Sobald es erstellt wurde, kann die Anzahl der Elemente nicht geändert werden.
  - Der Zugriff auf Elemente erfolgt über **Indizes**.
- `ArrayList<String>`:
  - Eine **ArrayList** ist eine **dynamische Liste**, die ihre Größe ändern kann (Elemente können hinzugefügt oder entfernt werden).
  - ArrayLists bieten viele **nützliche Methoden**, z.B. `add()`, `remove()` oder `size()`.
- **Beispiel: ArrayList**:

```
ArrayList<String> list = new ArrayList<>();
list.add("Anna");
list.add("25");
list.add("Berlin");
```

- `String[]` wird verwendet, wenn die Anzahl der Elemente fest ist (z.B. 3 Spalten in einer CSV-Datei).
- `ArrayList<String>` wird verwendet, wenn die Anzahl der Elemente variabel ist.
- In unserem Beispiel hilft `split`, um eine Zeile in ein Array von Strings zu zerlegen, die dann separat verarbeitet werden können.

Daten mit `split` in eine ArrayList speichern

- Jede Zeile der Datei wird mit `split` in einzelne Werte zerlegt.
- Die Werte jeder Zeile werden in eine **innere ArrayList** gespeichert.
- Diese **innere ArrayList** wird einer äußeren **ArrayList** hinzugefügt, um alle Zeilen zu speichern.

Beispiel: Daten in `ArrayList<ArrayList<String>>` speichern

```java
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        ArrayList<ArrayList<String>> data = new ArrayList<>();  // Äußere ArrayList für alle Zei

        try (Scanner scanner = new Scanner(Paths.get("data.txt"))) {
            while (scanner.hasNextLine()) {
                String line = scanner.nextLine();
                String[] parts = line.split(";");  // Zeile aufsplitten

                ArrayList<String> row = new ArrayList<>();  // Innere ArrayList für eine Zeile
                for (String part : parts) {
                    row.add(part);  // Füge jeden Teil der Zeile der inneren Liste hinzu
                }
                data.add(row);  // Füge die innere Liste zur äußeren Liste hinzu
            }

            // Ausgabe der Daten
            for (ArrayList<String> row : data) {
                System.out.println(row);
            }
```

Erklärung: Speicherung in `ArrayList`

- **`ArrayList<ArrayList<String>> data`**: Eine äußere `ArrayList`, die mehrere `ArrayList<String>` speichert – eine für jede Zeile.
- **`line.split(";")`**: Zerlegt jede Zeile in ein **`String[]`**, das die Spalten der Zeile enthält.
- Für jede Zeile wird eine **innere** **`ArrayList<String>`** erstellt, die die Teile der Zeile speichert.
- Diese innere Liste wird zur **äußeren** **`ArrayList`** hinzugefügt, um alle Zeilen zu speichern.

Beispiel: Inhalt der Datei

```
Anna;25;Berlin
Bob;30;München
Charlie;22;Hamburg
```

- **Zeile 1** wird in die `ArrayList` `["Anna", "25", "Berlin"]` aufgeteilt.
- **Zeile 2** wird in die `ArrayList` `["Bob", "30", "München"]` aufgeteilt.
- **Zeile 3** wird in die `ArrayList` `["Charlie", "22", "Hamburg"]` aufgeteilt.
- Alle diese Listen werden in der äußeren **`ArrayList<ArrayList<String>>`** gespeichert.

Ausgabe der Daten

- Nach dem Einlesen und Speichern können die Daten einfach ausgegeben werden:

```
for (ArrayList<String> row : data) {
    System.out.println(row);
}
```

- Beispielausgabe:

```
[Anna, 25, Berlin]
[Bob, 30, München]
[Charlie, 22, Hamburg]
```

Zusammenfassung `ArrayList<ArrayList<String>>`

- `split` wird verwendet, um jede Zeile in einzelne Werte zu zerlegen.
- Diese Werte werden in einer **inneren ArrayList** gespeichert.
- Die **äußere ArrayList** speichert alle Zeilen (jede als `ArrayList<String>`).

Zusammenfassung

- Sie können Daten aus Tastatureingaben oder Dateien einlesen.
- Der **Scanner** kann sowohl für Benutzereingaben als auch für das Lesen von Dateien verwendet werden.
- **CSV-Dateien** sind eine einfache Möglichkeit, Daten zu strukturieren und zu speichern.

Ende Teil 04