# Modellierung und Programmierung 1

Übung 1

Stefan Preußner

2. / 3. November 2020

# Download der Übungsunterlagen

- Im Rahmen dieser Übung gibt es Folien und Codebeispiele
- Homepage des Lehrstuhls für Schwarmintelligenz und Komplexe Systeme:
  `http://pacosy.informatik.uni-leipzig.de`
- Homepage → Mitarbeiter → Stefan Preußner → MuP1
- Benutzer: `mup`
- Passwort: `MuP1WS20/21`

# Organisatorisches

# Inhalt und Ziel dieser Übung

- (Nachbesprechung der Vorlesung)
- (Vertiefung von Vorlesungsinhalten)
- Praktische Modellier- und Programmierübungen
- **Vor**- und Nachbesprechung der Übungsserien

- Die Teilnahme an den Übungen ist freiwillig!

# Übungsserien / Prüfungszulassung

- Es gibt insgesamt 6 Übungsserien:
  - Die erste (Probe-)Übungsserie hat einen Umfang von 3 (Zusatz-)Punkten
  - Die übrigen 5 Übungsserien haben einen Umfang von je ca. 40 Punkten
- Das Bearbeiten der Übungsaufgaben und Erreichen von 50 % der Maximalpunktzahl ist Voraussetzung für die Zulassung zur Klausur
  - Ausgenommen hiervon sind Studierende, die eine gültige Prüfungszulassung aus einem früheren Semester besitzen

# Einschreibung in die Übungsgruppen

- Die Bereitstellung der Vorlesungsunterlagen und der Übungszettel sowie deren Abgabe erfolgt über Moodle: `moodle2.uni-leipzig.de/course/view.php?id=29679`
- Die Einschreibung sollte in die gleiche Übungsgruppe wie im AlmaWeb erfolgen
  - Montag, 9:15 - 10:45 Uhr: Gruppe b
  - Dienstag, 13:15 - 14:45 Uhr: Gruppe f
- Einschreibeschlüssel: MUP1

# Übungsserien / Abgabe der Lösungen

- Die Frist zur Abgabe einer Übungsserie ist auf dem jeweiligen Übungszettel angegeben (i.d.R. Mittwoch, 22:00 Uhr)
- Zu spät abgebene Lösungen können nicht bewertet werden
- Die Übungsaufgaben können zusammen bearbeitet werden, die abgegebenen Lösungen müssen jedoch erkennbar eigenständige Arbeiten sein
  - Offensichtliche Duplikate werden mit 0 Punkten bewertet

# Übungsserien / Abgabe der Lösungen

- Texte und Abbildungen in **einem PDF-Dokument**
- Die Abgabe von Code erfolgt in Form von **einem ZIP-Archiv**
  - kein `RAR`, `tar`, `gzip`, `7z` etc.
- Quellcode immer in Form von **kompilierbaren Dateien**, also reiner Text
  - keinen Quelltext als `PDF`, `DOC`, `DOCX` etc.
- Keine kompilierten Dateien abgeben, insbesondere keine `class`-Dateien
- Dateien so packen, dass eine eventuelle Ordnerstruktur erhalten bleibt

# Übungsserien / Abgabe der Lösungen

- Die erste *Übungsserie* ist eine Testserie, um die Abgabe über Moodle vor Beginn der eigentlichen Übungsserien üben zu können
- Die für die Testserie vergebenen Punkte sind **Bonus**punkte

# Übungsserien / Korrektur und Bewertung

- Die Korrektur der abgebenen Lösungen sollte innerhalb einer Woche erfolgen
- Falls
  - eine Lösung in diesem Zeitraum nicht korrigiert wurde oder
  - sich Unstimmigkeiten bezüglich der Bewertung von Lösungen ergeben,

  bitte eine E-Mail an den **Übungsleiter**:
  `preussner@informatik.uni-leipzig.de`

# Ausfall von Übungsgruppen / Änderungen

- Sollte eine Übungsgruppe ausfallen (Feiertage, dies academicus, Krankheit), dann bitte nach Möglichkeit eine der anderen Übungsgruppen aufsuchen
- Änderungen bei den Übungsgruppen, insbesondere Zusammenlegungen, werden rechtzeitig in Moodle bekanntgegeben

# Programme

# Darstellung von UML-Diagrammen

- Dia Diagram Editor
  - `https://wiki.gnome.org/Apps/Dia/Download`
  - `http://dia-installer.de/`
- yEd Graph Editor
  - `https://www.yworks.com/products/yed`
- diverse Online-Tools

# Entwicklungsumgebungen (IDEs) für Java

- Eclipse
  - `https://www.eclipse.org/`
  - Eclipse IDE for Java Developers
- IntelliJ
  - `https://www.jetbrains.com/idea/`
  - Studierende können die *Ultimate Edition* kostenlos nutzen
- NetBeans
  - `https://netbeans.org/`

# Modellierung und Programmierung 1

Übung 2

Stefan Preußner

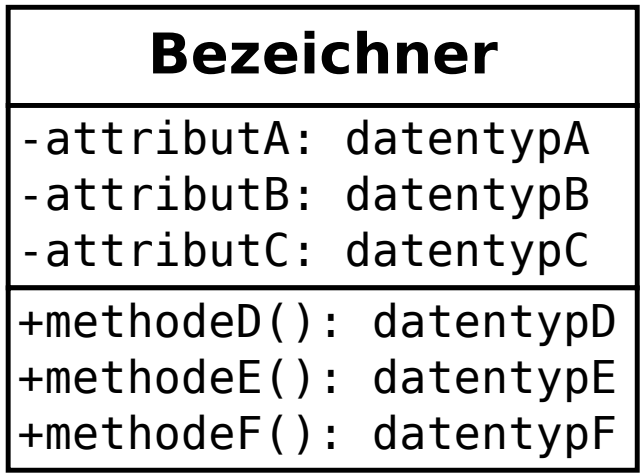
9. / 10. November 2020

# Download der Übungsunterlagen

- Im Rahmen dieser Übung gibt es Folien und Codebeispiele
- Homepage des Lehrstuhls für Schwarmintelligenz und Komplexe Systeme:
  `http://pacosy.informatik.uni-leipzig.de`
- Homepage → Mitarbeiter → Stefan Preußner → MuP1
- Benutzer: `mup`
- Passwort: `MuP1WS20/21`

# Objekte und Klassen

- Annahme: alle materiellen und immateriellen Dinge sind Objekte
- **Ein Objekt ist eine Instanz (Ausprägung) einer Klasse**
- Klassen haben einen Bezeichner, ggf. mehrere Eigenschaften/Attribute und ggf. mehrere Funktionalitäten/Methoden



# Datentypen

Jedes Attribut hat einen bestimmten Datentyp, beispielsweise:

- `boolean`: Boolesche Variablen können zwei mögliche Werte annehmen - wahr oder falsch
- `int` und `long`: Ganzzahlen (*integer*)
- `float` und `double`: Gleitkommazahlen (*floating-point number*)
- `char`: Buchstaben. Ein `char` wird von **einfachen** Anführungszeichen eingeschlossen ('A')
- `String`: Zeichenketten. Ein `String` wird von **doppelten** Anführungszeichen eingeschlossen ("A")

# Datentypen

Weitere Datentypen:

- `Klassenname`: Klassen (und Schnittstellen) sind ebenfalls Datentypen. Ist der Wert eines Attributs die Instanz einer Klasse, dann ist die Klasse selbst der Datentyp dieses Attributs.
  - Beispiel: die Klasse Vorlesung besitzt das Attribut `vortragender` mit dem Datentyp `Professor`
    $\rightarrow$ `vortragender: Professor`
- `void`: kennzeichnet eine Methode, die nichts zurückgibt, und steht **anstelle** eines Datentyps

# Datentypen

Weitere Datentypen:

- String[]: Ein Array vom Typ String. Arrays sind Listen mit einer festen Größe.
- Collection<String>: Eine `Collection` vom Typ String. In Java gibt es zahlreiche Datentypen, die Listen mit veränderlicher Größe, Mengen und/oder Queues darstellen. Da bei der Modellierung die konkrete Implementierung einer Sammlung von Elementen oftmals unwichtig ist, kann im UML-Klassendiagramm der viele Container umfassende Datentyp `Collection` verwendet werden.

# Funktionen

Beispiele:

```
public String getName()
private void erstelleAdresse( String strasse, int hausnummer, int
postleitzahl, String ort)
boolean halteVorlesung( String titel, String inhalt, int dauer )
```

---

Sichtbarkeit: private, protected, *leer*, public
Rückgabe: ein Datentyp oder void ($\rightarrow$ kein Rückgabewert)
Funktionsname
Datentyp: boolean, int, float, String, Collection<String>...
Parameter

# Funktionen in UML

```
public String getName()
private void erstelleAdresse( String strasse, int hausnummer, int
postleitzahl, String ort)
boolean halteVorlesung( String titel, String inhalt, int dauer )
```

↓

| **Professor** |
| --- |
| +getName(): String<br>-erstelleAdresse(strasse:String,hausnummer:int,postleitzahl:int,ort:String): void<br>~halteVorlesung(titel:String,inhalt:String,dauer:int): boolean |

# Funktionen in UML

| **Professor** |
| --- |
| +getName(): String<br>-erstelleAdresse(strasse:String,hausnummer:int,postleitzahl:int,ort:String): void<br>~halteVorlesung(titel:String,inhalt:String,dauer:int): boolean |

Sichtbarkeit: private, protected, *leer*, public
Rückgabe: ein Datentyp oder void ($\rightarrow$ kein Rückgabewert)
Funktionsname
Datentyp: boolean, int, float, String, Collection<Integer>...
Parameter

# Eine Beispielklasse

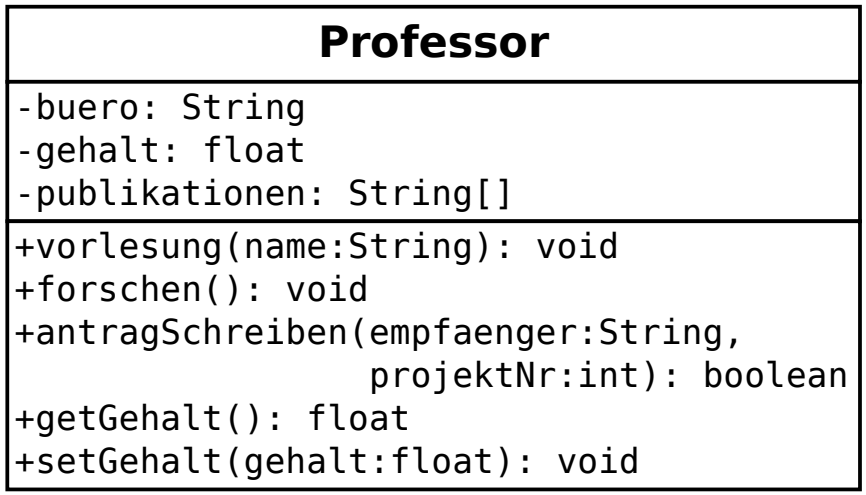
An der Uni Leipzig arbeiten zahlreiche Professoren, welche hier modelliert werden sollen. Ein Professor besitzt ein Büro (bspw. im Raum P3-456) und bekommt ein jährliches Gehalt (bspw. 87654,32 €) gezahlt. Ein Professor kann eine Vorlesung (bspw. "MuP1") halten, forschen oder Anträge schreiben. Bei einem Antrag müssen der Empfänger (bspw. "DFG") und die Projektnummer (bspw. 715299) angegeben werden, das Ergebnis der Antragsstellung kann positiv oder negativ sein. Viele Professoren haben eine lange Liste von Publikationen (bspw. "Java ist auch eine Insel" und "Java in 21 Wochen") vorzuweisen.
Für das Gehalt sollen ein Getter und ein Setter erstellt werden. Auf weitere Getter und Setter sowie Konstruktoren kann verzichtet werden.

# Die Beispielklasse in UML

Ein Professor besitzt ein Büro (bspw. im Raum P3-456) und bekommt ein jährliches Gehalt (bspw. 87654,32 €) gezahlt. Ein Professor kann eine Vorlesung (bspw. "MuP1") halten, forschen oder Anträge schreiben. Bei einem Antrag müssen der Empfänger (bspw. "DFG") und die Projektnummer (bspw. 715299) angegeben werden, das Ergebnis der Antragsstellung kann positiv oder negativ sein. Viele Professoren haben eine lange Liste von Publikationen (bspw. "Java ist auch eine Insel" und "Java in 21 Wochen") vorzuweisen.
Für das Gehalt sollen ein Getter und ein Setter erstellt werden. Auf weitere Getter und Setter sowie Konstruktoren kann verzichtet werden.

# Die Beispielklasse in UML

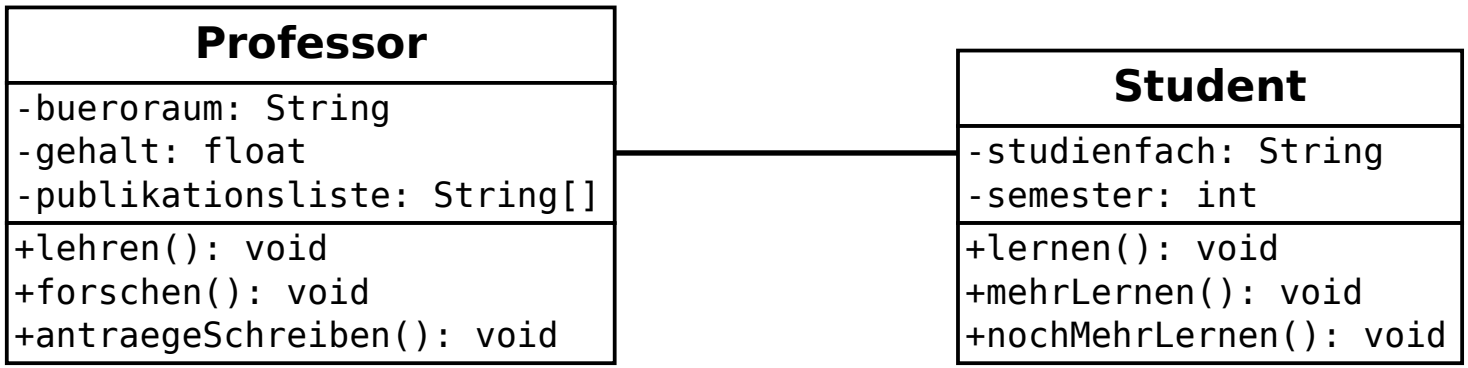
Ein Professor besitzt ein Büro (bspw. im Raum P3-456) und bekommt ein jährliches Gehalt (bspw. 87654,32 €) gezahlt. Ein Professor kann eine Vorlesung (bspw. "MuP1") halten, forschen oder Anträge schreiben. Bei einem Antrag müssen der Empfänger (bspw. "DFG") und die Projektnummer (bspw. 715299) angegeben werden, das Ergebnis der Antragsstellung kann positiv oder negativ sein. Viele Professoren haben eine lange Liste von Publikationen (bspw. "Java ist auch eine Insel" und "Java in 21 Wochen") vorzuweisen.
Für das Gehalt sollen ein Getter und ein Setter erstellt werden. Auf weitere Getter und Setter sowie Konstruktoren kann verzichtet werden.



# Klassenbeziehungen

- Klassen können fünf unterschiedliche Arten von Beziehungen zueinander haben:
  - Assoziation
  - Aggregation
  - Komposition
  - Vererbung
  - Realisierung / Interface

# Assoziation

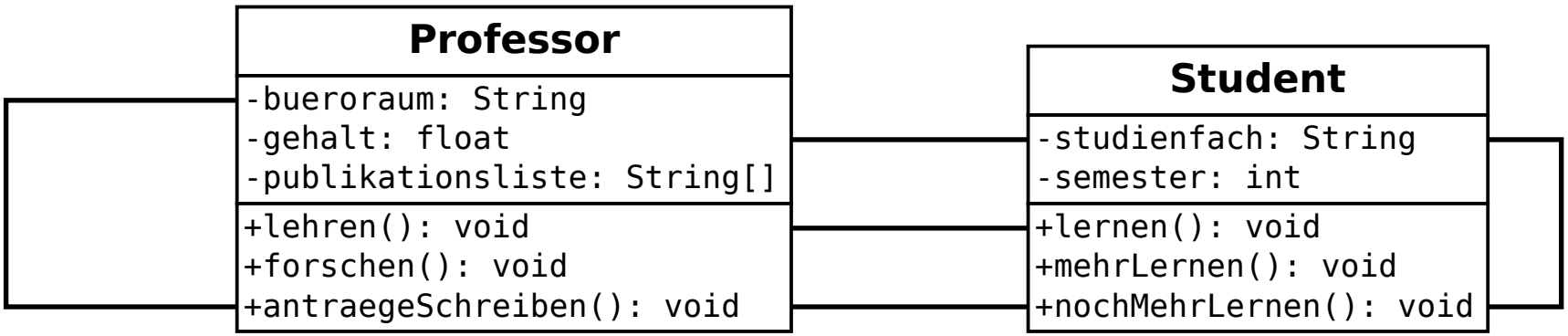



- „X steht in Beziehung zu Y“
- „X kommuniziert mit Y“
- „X nutzt Y“

# Assoziation

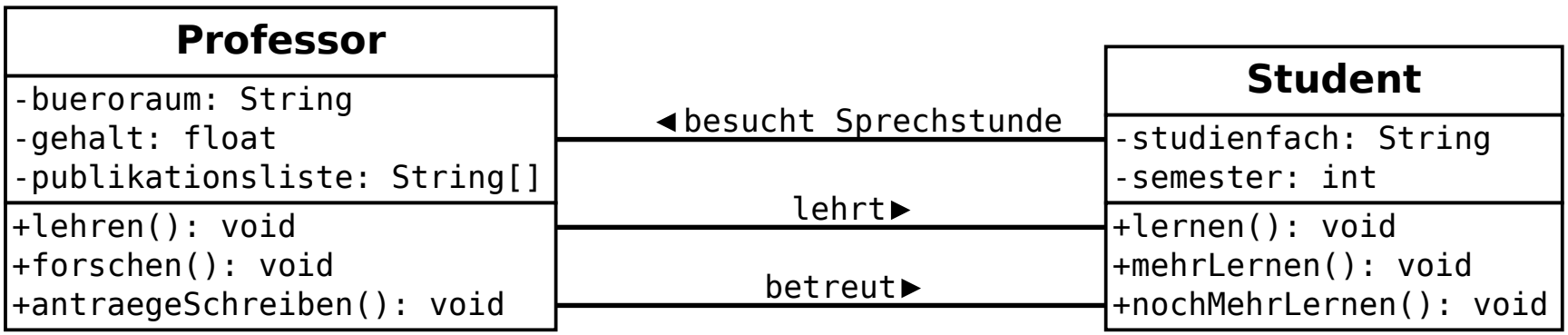

- Selbstassoziativität ist möglich
  - Beziehungen von Professoren untereinander
  - Beziehungen von Studenten untereinander

# Assoziation



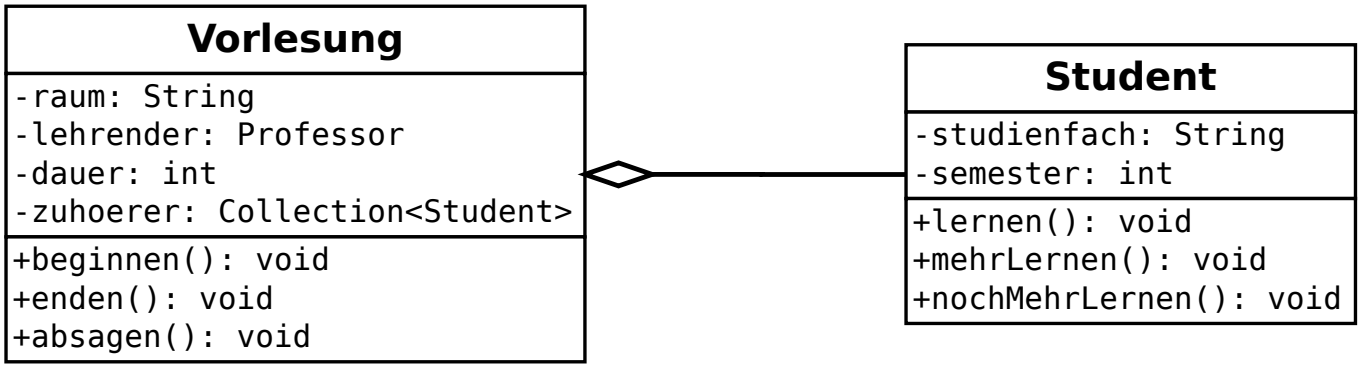

- Mehrere Beziehungen sind möglich
  - Professoren halten vor Studenten Vorlesungen, betreuen Arbeiten, nehmen Prüfungen ab und beraten in Sprechstunden

# Bezeichnete Beziehungen



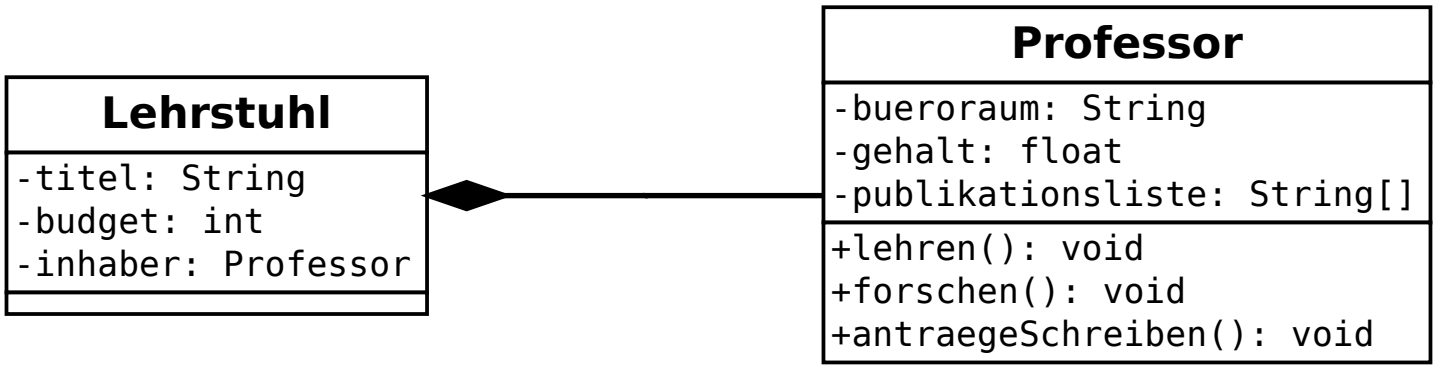

- Beziehungen können bezeichnet werden
- Durch einen Pfeil kann die Richtung der Beziehung angegeben werden

# Aggregation




- „besitzt ein(e)“
- „ist ein Teil von“
- Beide Objekte können unabhängig voneinander existieren
- Eine Aggregation impliziert fast immer ein entsprechendes Attribut bei der besitzenden Klasse!

# Komposition

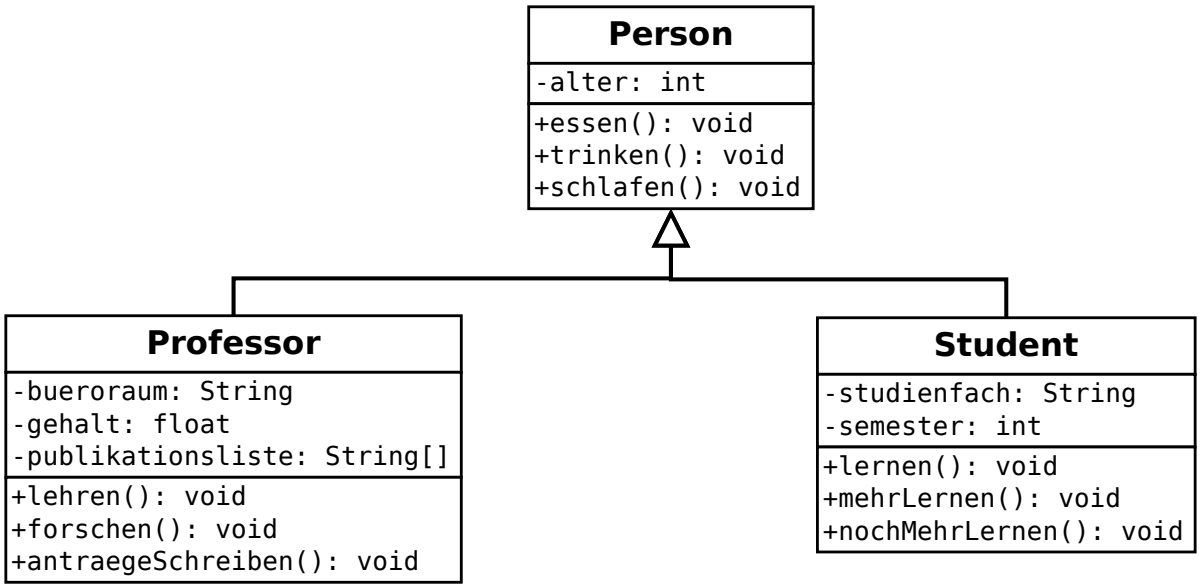



- „besteht aus“
- Die Lebzeit des einen Objekts ist an die Lebzeit des anderen Objekts gekoppelt

# Assoziation vs. Aggregation vs. Komposition

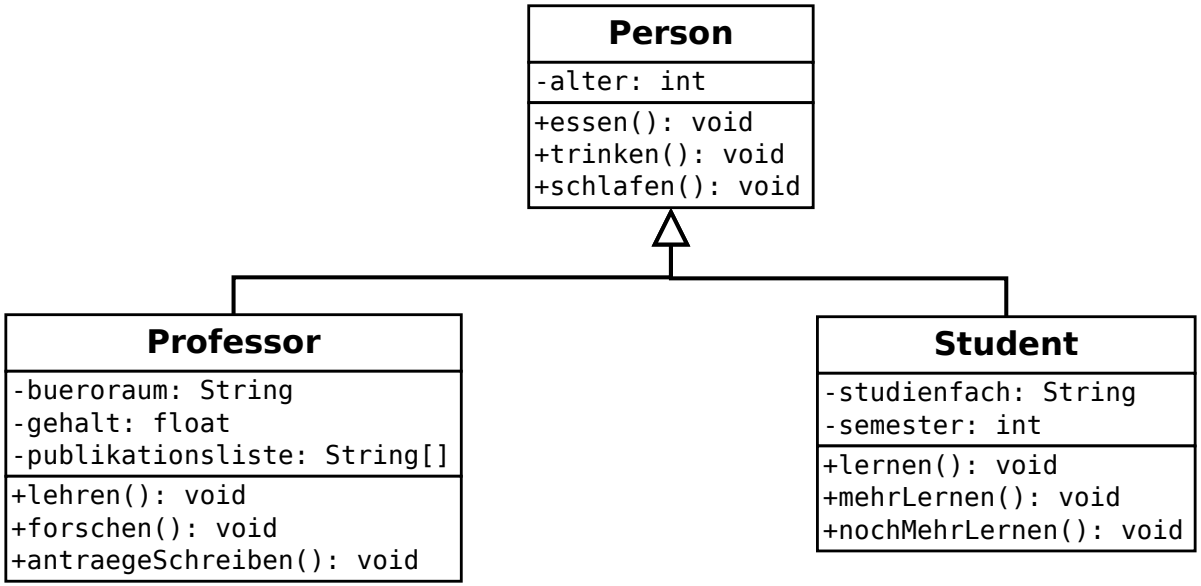
- Die Aggregation ist ein Spezialfall der Assoziation
  - $\rightarrow$ Eine Aggregation kann immer durch eine Assoziation ersetzt werden
- Die Komposition ist ein Spezialfall der Aggregation
  - $\rightarrow$ Eine Komposition kann immer durch eine Aggregation und damit auch durch eine Assoziation ersetzt werden
- Die Verwendung von Kompoisitonen und Aggregationen liegt im Ermessen des Modellierers, im UML-Standard gibt es hierzu kaum Vorgaben
  - Aber: eine Klasse kann nicht über jeweils eine Komposition Teil zwei anderer Klassen sein

# Vererbung



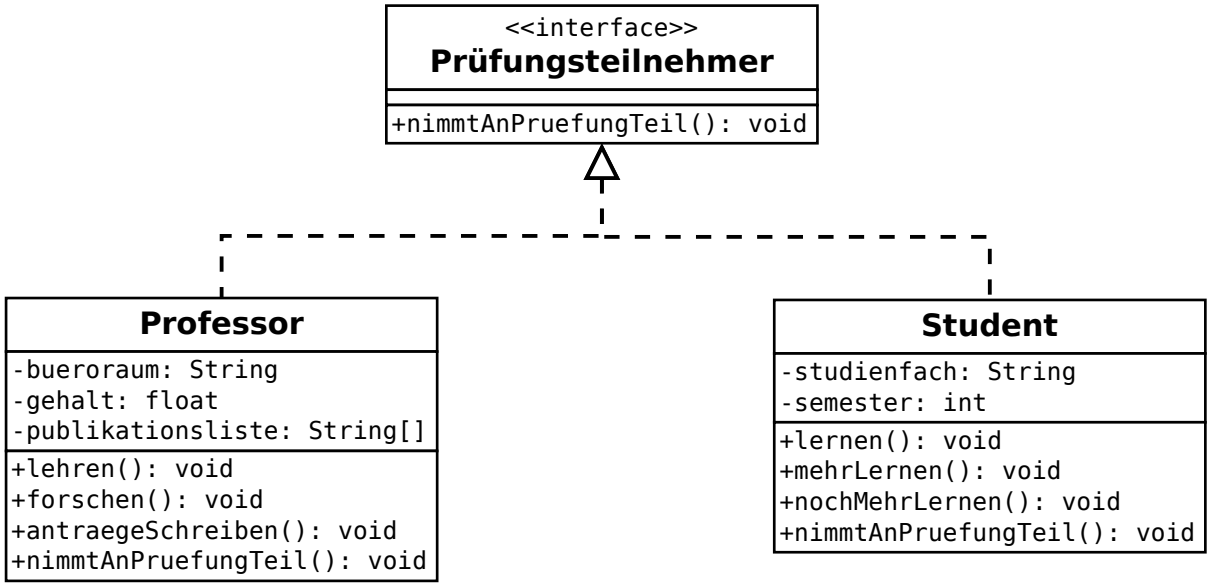

- „ist ein(e)“
- **Person** ist hier die Generalisierung der Klassen **Professor** und **Student**
- **Professor** und **Student** sind Spezialisierungen von **Person**

# Vererbung



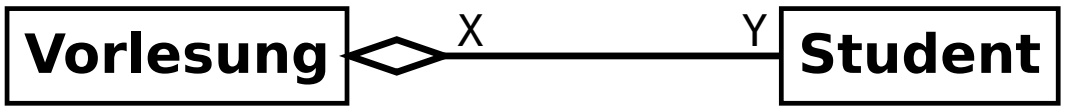

- **Professor** und **Student** erben alle Attribute und Methoden von **Person**, d.h. ein `Professor` besitzt ebenfalls ein Attribut `alter`, auch wenn dieses nicht explizit angegeben wird

# Realisierung / Interfaces




- Vom **Prüfungsteilnehmer** selbst kann keine Instanz erzeugt werden
- Die Funktion nimmtAnpruefungTeil() ist bei **Prüfungsteilnehmer** nicht implementiert

# Multiplizitäten



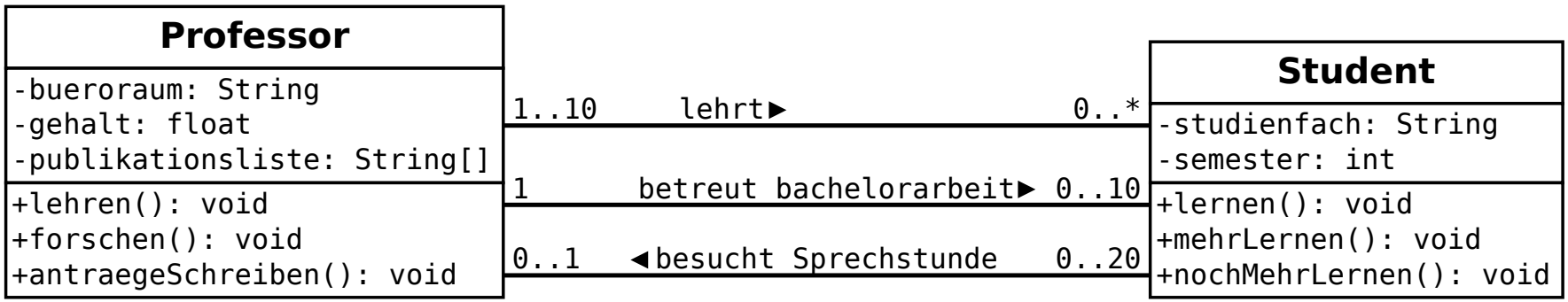

- „Eine Vorlesung besteht aus Y Studenten“
- „Ein Student ist Teil von X Vorlesungen“

# Multiplizitäten

| Wert | Bedeutung |
|---|---|
| | genau 1 oder beliebig viele [1] |
| m | genau m |
| m..n | mindestens m, höchstens n |
| * | beliebig viele |
| m..* | mindestens m |
| 0..* | beliebig viele |
| m,n | m oder n |

[1] Im Rahmen von MuP wird angenommen, dass eine nicht angegebene Multiplizität gleichbedeutend mit "beliebig viele" ist.

# Multiplizitäten



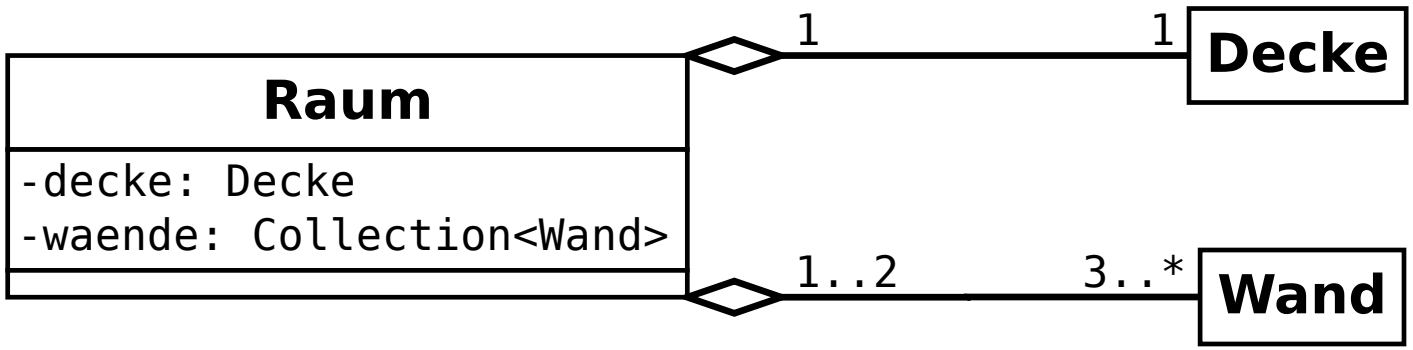

(Die Zahlen sind rein willkürlich gewählt und sollen nur verdeutlichen, dass die Benennung von Assoziationen besonders bei unterschiedlichen Multiplizitäten hilfreich oder notwendig sein kann.)

# Beispiele

Ein **Raum** besteht aus mindestens drei **Wänden** und einer **Decke**. Eine Wand gehört zu einem bis zwei Räumen, eine Decke gehört zu genau einem Raum.

# Beispiele

Ein **Raum** besteht aus mindestens drei **Wänden** und einer **Decke**. Eine Wand gehört zu einem bis zwei Räumen, eine Decke gehört zu genau einem Raum.

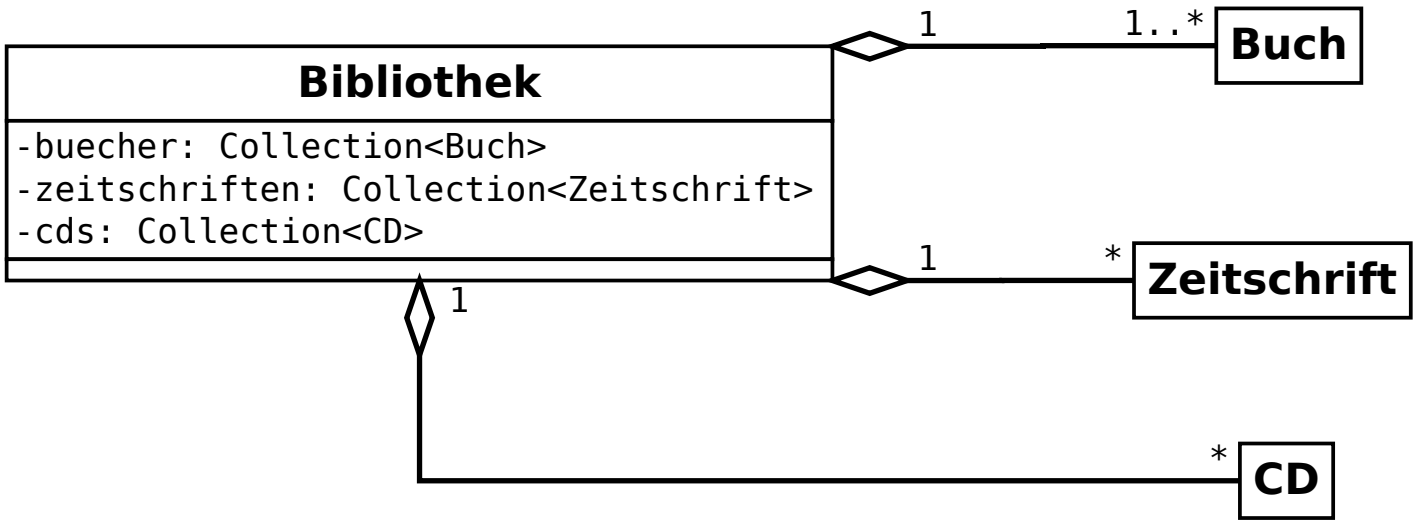


# Beispiele

Eine **Bibliothek** besteht aus mindestens einem **Buch**. Eine Bibliothek kann auch **Zeitschriften** und **CDs** umfassen. Jedes Buch, jede Zeitschrift und jede CD gehört in genau eine Bibliothek.

# Beispiele

Eine **Bibliothek** besteht aus mindestens einem **Buch**. Eine Bibliothek kann auch **Zeitschriften** und **CDs** umfassen. Jedes Buch, jede Zeitschrift und jede CD gehört in genau eine Bibliothek.

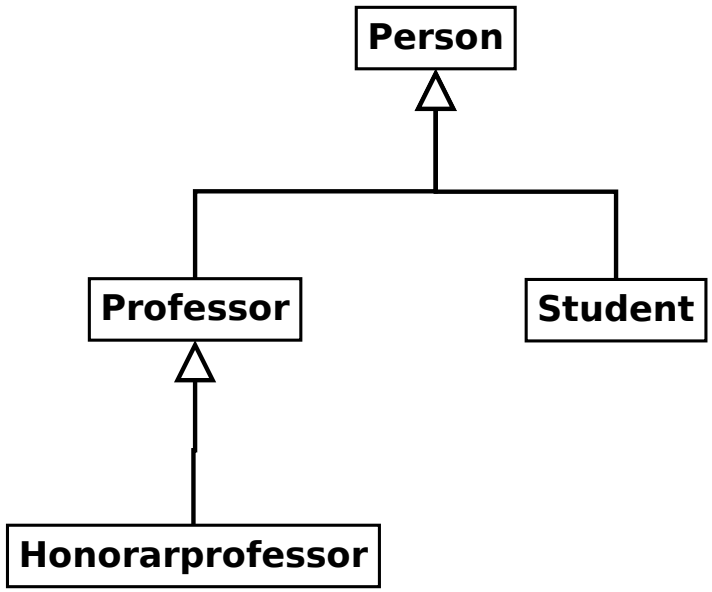


# Beispiele

**Honorarprofessoren** sind sehr spezielle Professoren. Wie alle anderen **Professoren** und **Studenten** sind sie **Personen**.

# Beispiele

**Honorarprofessoren** sind sehr spezielle Professoren. Wie alle anderen **Professoren** und **Studenten** sind sie **Personen**.



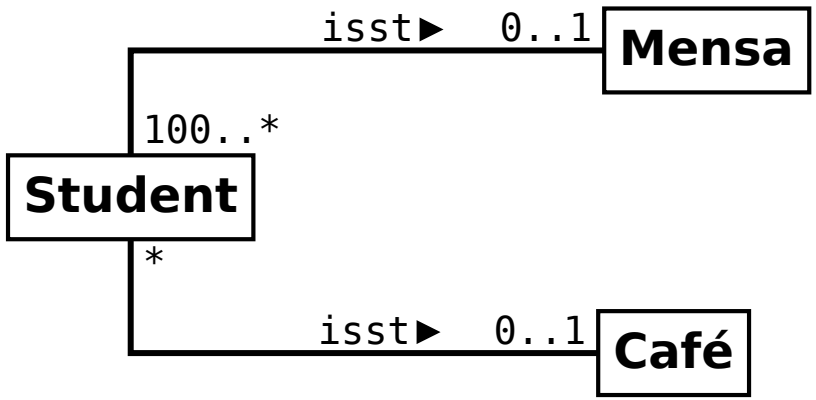
# Modellierung und Programmierung 1

Übung 3

Stefan Preußner

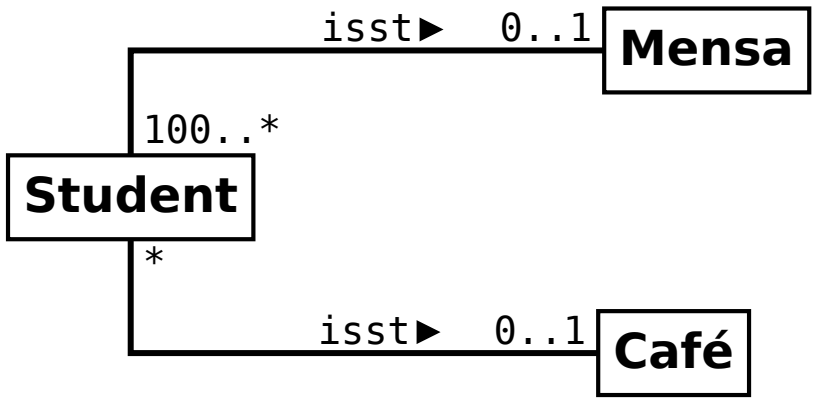
16. / 17. November 2020

# Welche Aussagen sind aus dem Diagramm ableitbar?




- Jeder Student geht entweder in die Mensa oder in ein Café.
- Manche Studenten gehen gar nicht essen.
- Die Mensa ist immer besser besucht als das Café.
- Die Mensa kann leer bleiben.
- Das Café kann leer bleiben.

# Welche Aussagen sind aus dem Diagramm ableitbar?

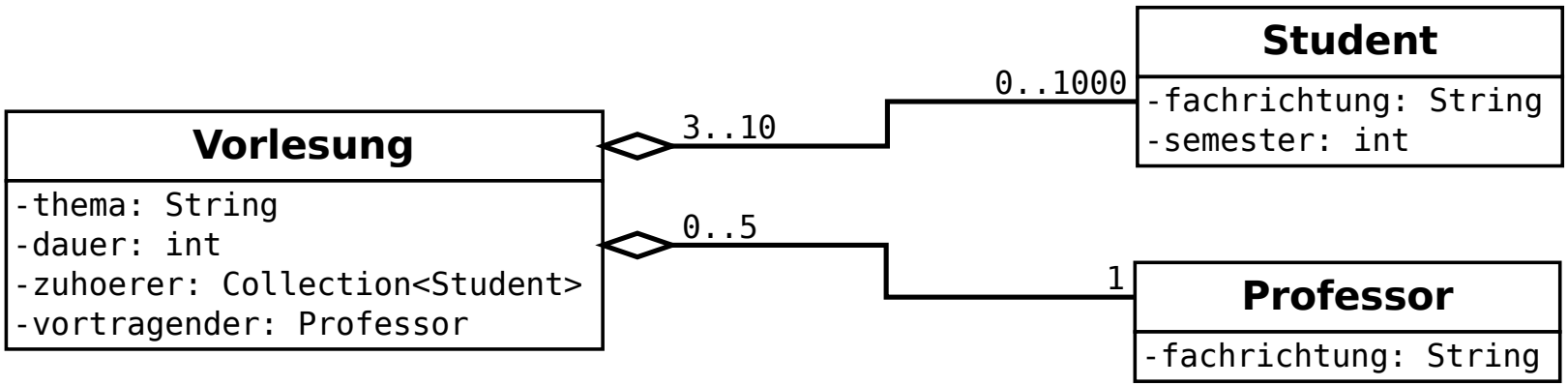



- Jeder Student geht entweder in die Mensa oder in ein Café. ×
- Manche Studenten gehen gar nicht essen. ✓
- Die Mensa ist immer besser besucht als das Café. ×
- Die Mensa kann leer bleiben. ×
- Das Café kann leer bleiben. ✓

# Welche Aussagen sind aus dem Diagramm ableitbar?

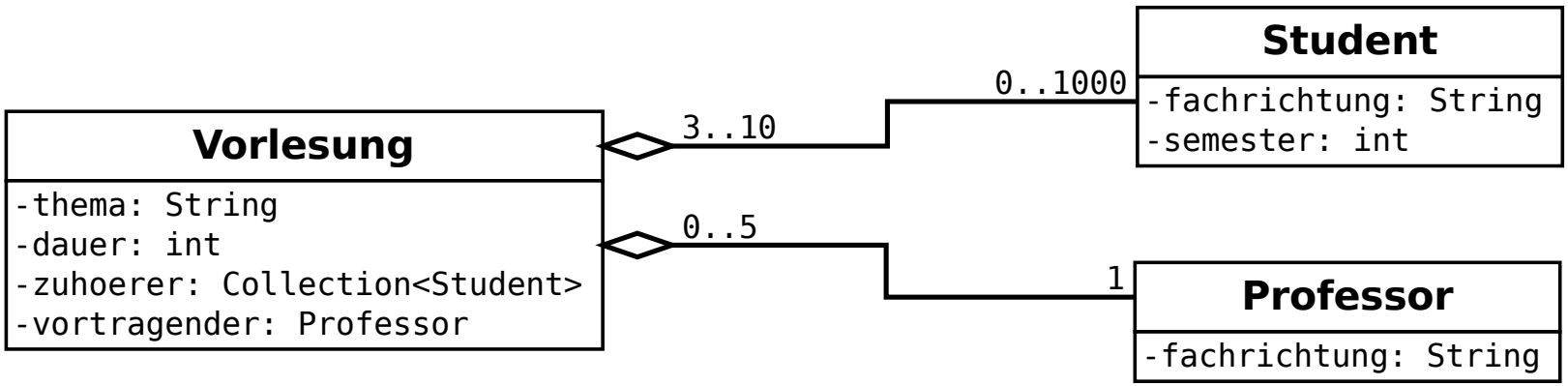
- Zu 1. und 2.: Die untere Grenze der Assoziation Student → Mensa ist 0, die untere Grenze der Assoziation Student → Café ist 0. Damit kann ein konkreter Student weder mit der Mensa noch mit dem Café assoziiert sein.
- Zu 3.: Die untere Grenze der Assoziation Mensa → Student ist 100, die obere Grenze der Assoziation Café → ist * und damit größer als 100.
- Zu 4.: Die untere Grenze der Assoziation Mensa → Student ist 100.
- Zu 5.: Die untere Grenze der Assoziation Café → Student ist 0.

# Welche Aussagen sind aus dem Diagramm ableitbar?




- Zuhörer und Vortragender einer Vorlesung haben immer die gleiche Fachrichtung.
- Das Thema einer Vorlesung hängt immer von der Fachrichtung des Professors ab.
- Manchmal hat ein Professor keine Zuhörer.
- Manche Studenten gehen zu keiner Vorlesung.
- Eine Vorlesung wird von bis zu 5 Professoren gehalten.

# Welche Aussagen sind aus dem Diagramm ableitbar?

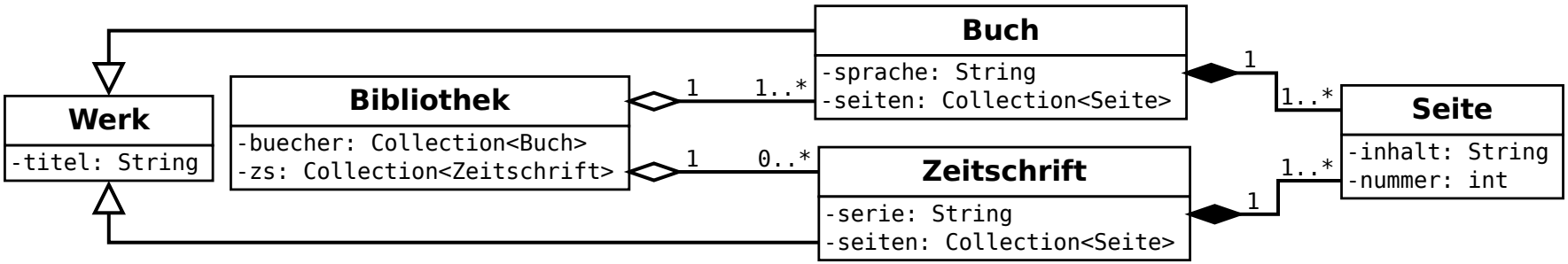



- Zuhörer und Vortragender einer Vorlesung haben immer die gleiche Fachrichtung.×
- Das Thema einer Vorlesung hängt immer von der Fachrichtung des Professors ab.×
- Manchmal hat ein Professor keine Zuhörer.✓
- Manche Studenten gehen zu keiner Vorlesung.×
- Eine Vorlesung wird von bis zu 5 Professoren gehalten.×

# Welche Aussagen sind aus dem Diagramm ableitbar?

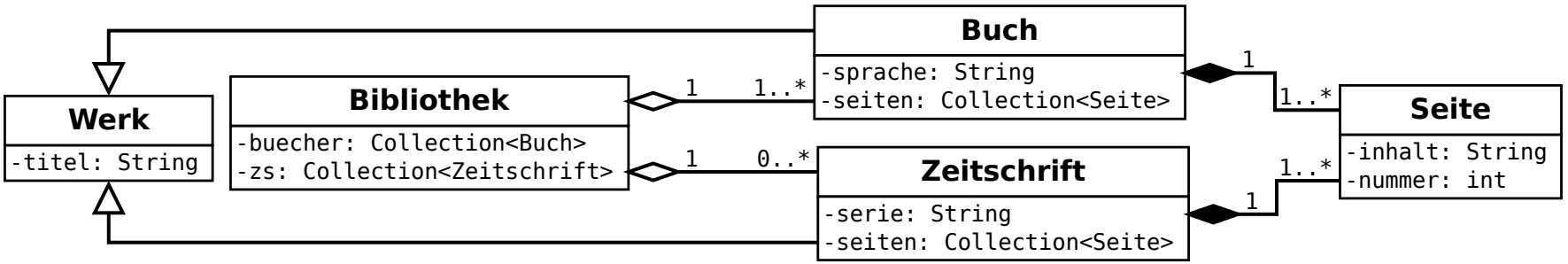
- Zu 1.: Aussagen zum Wert von Attributen lassen sich nicht anhand von UML-Diagrammen treffen.
- Zu 2.: Wie 1.
- Zu 3.: Die untere Grenze der Aggregation Vorlesung $\rightarrow$ Student ist 1.
- Zu 4.: Die untere Grenze der Aggregation Student $\rightarrow$ Vorlesung ist 3.
- Zu 5.: Die obere Grenze der Aggregation Vorlesung $\rightarrow$ Professor ist 1.

# Welche Aussagen sind aus dem Diagramm ableitbar?




- Eine Bibliothek enthält mindestens eine Zeitschriftenseite.
- Eine Bibliothek enthält mindestens eine Buchseite.
- Eine Bibliothek kann Bücher in verschiedenen Sprachen enthalten.
- Jedes Buch hat einen Titel.
- Jede Seite einer Zeitschrift befindet sich in genau einer Bibliothek.

# Welche Aussagen sind aus dem Diagramm ableitbar?




- Eine Bibliothek enthält mindestens eine Zeitschriftenseite. ×
- Eine Bibliothek enthält mindestens eine Buchseite. ✓
- Eine Bibliothek kann Bücher in verschiedenen Sprachen enthalten. ×
- Jedes Buch hat einen Titel. ✓
- Jede Seite einer Zeitschrift befindet sich in genau einer Bibliothek. ✓

# Welche Aussagen sind aus dem Diagramm ableitbar?

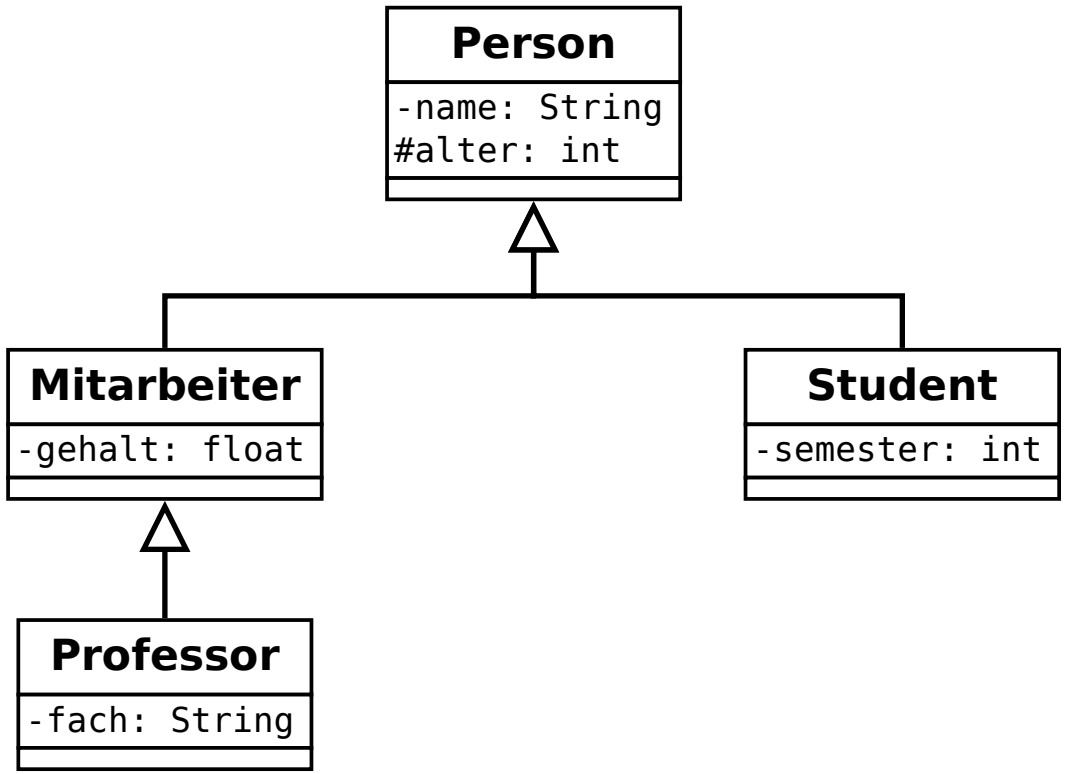
- Zu 1.: Die untere Grenze der (implizierten) Aggregation Bibliothek $\rightarrow$ Zeitschrift $\rightarrow$ Seite ergibt sich durch Multiplikation der unteren Grenze der Aggregation Bibliothek $\rightarrow$ Zeitschrift mit der unteren Grenze der Aggregation Zeitschrift $\rightarrow$ Seite. Es gilt $0 \cdot 1 = 0$.
- Zu 2.: Äquivalent zu 1. (Buch statt Zeitschrift). Es gilt $1 \cdot 1 = 1$.
- Zu 3.: Aussagen zum Wert von Attributen lassen sich nicht anhand von UML-Diagrammen treffen.
- Zu 4.: Buch ist eine Unterklasse von Werk. Buch erbt somit das Attribut titel.
- Zu 5.: Die obere Grenze der (implizierten) Aggregation Seite $\rightarrow$ Zeitschrift $\rightarrow$ Bibliothek ergibt sich durch Multiplikation der oberen Grenze der Aggregation Seite $\rightarrow$ Zeitschrift mit der oberen Grenze der Aggregation Zeitschrift $\rightarrow$ Bibliothek. Es gibt $1 \cdot 1 = 1$.

# Schreibweise von Bezeichnern in Java

Die folgenden Hinweise geben nur (sehr weit verbreitete) Konventionen wieder und sind keine verbindlichen Regeln:

- Paketnamen werden vollständig klein geschrieben
- Klassen- und Schnittstellennamen beginnen mit einem Großbuchstaben
- Methoden und Variablennamen beginnen mit einem Kleinbuchstaben, innerhalb eines Namens beginnen Wörter mit einem Binnenmajuskel (*camel case*, bspw. `addiereZahl`)
  - Ausnahme: Konstanten werden vollständig in Großbuchstaben geschrieben
- Bezeichner sollten "sprechende" Namen haben

# Sichtbarkeit und Vererbung



| | | Person | Mitarbeiter | Professor | Student |
|---|---|---|---|---|---|
| besitzt Attribut | name | ✓ | ✓ | ✓ | ✓ |
| | alter | ✓ | ✓ | ✓ | ✓ |
| | gehalt | | ✓ | ✓ | |
| | semester | | | | ✓ |
| | fach | | | ✓ | |
| Zugriff auf Attribut | name | ✓ | | | |
| | alter | ✓ | ✓ | ✓ | ✓ |
| | gehalt | | ✓ | | |
| | semester | | | | ✓ |
| | fach | | | ✓ | |

# Übung - Konstruktoren

Erstellen Sie für alle im obigen UML-Diagramm gezeigten Klassen jeweils einen Konstruktor (in UML) so, dass im Konstruktor alle Attribute gesetzt werden können.
Hinweis: Konstruktoren haben **keinen** Rückgabedatentyp.

# Übung - Konstruktoren

Erstellen Sie für alle im obigen UML-Diagramm gezeigten Klassen jeweils einen Konstruktor (in UML) so, dass im Konstruktor alle Attribute gesetzt werden können.
Hinweis: Konstruktoren haben **keinen** Rückgabedatentyp.

```
+Person(name:String,alter:int)

+Mitarbeiter(name:String,alter:int,gehalt:float)

+Professor(name:String,alter:int,gehalt:float,
fach:String)

+Student(name:String,alter:int,semester:int)
```

Für ein Datum wird der Tag, der Monat und das Jahr jeweils als Zahl gespeichert. Ein Datum kann mit einem anderen Datum verglichen werden, das Ergebnis des Vergleiches ist wiederum ein Datum. Ein Datum kann in lesbarer Form ausgegeben werden.

Ein Streamingdienst hat mindestens 500 Filme und 20 Serien im Angebot. Das Angebot besteht dabei bereits bei der Gründung des Diensts. Nicht alle Filme und Serien sind exklusiv bei einem Streamingdienst zu sehen. Jeder Streamingdienst besitzt eine Nutzerdatenbank (die hier aus einer einfachen Liste bestehen soll), in der alle Nutzer gespeichert sind. Nutzer können nachträglich hinzugefügt werden. Manche Streamingdienste bieten verschiedene Abonnements (Abos) an, andere finanzieren sich über Werbung. Auch das Abomodell ändert sich nach Gründung des Streamingdienstes nicht.

Für Filme wird der Titel, eine Beschreibung sowie das Datum der Veröffentlichung gespeichert. Außerdem wird bei Filmen vermerkt, welche alternativen Fassungen es gibt; diese können nachträglich hinzugefügt werden.

Eine Serie hat einen Titel und besteht aus mindestens zwei Episoden. Es können neue Episoden hinzugefügt werden, dabei muss (damit die Reihenfolge eindeutig ist) neben der neuen auch die letzte alte Episode angegeben werden.

Episoden sind Filme, bei denen zusätzlich die dazugehörige Serie sowie Staffel- und Episodennummer gespeichert wird. Für jede Episode wird die nachfolgende Episode (soweit vorhanden) gespeichert; diese kann auch abgefragt werden.

Die Abos unterscheiden sich im Preis, in der maximalen Auflösung ('S' für SD, 'H' für HD usw.) sowie darin, ob mobiles Streamen unterstützt wird. Zwei Streamingdienste bieten niemals das gleiche Abo an.

Für jeden Nutzer wird eine Kundennummer sowie der Typ und das Ablaufdatum des aktuellen Abonnements (sofern vorhanden) gespeichert. Weiterhin muss jeder Nutzer sein Geburtsdatum angeben. Nutzer können Filme kaufen. Für den Nutzer wird gespeichert, welche Filme bereits gekauft wurden. Ein Nutzer kann sein Abo unter Angabe des Kündigungsdatums kündigen, in diesem Fall wird das tatsächliche Kündigungsdatum ermittelt und dem Kunden mitgeteilt. Ein Nutzer bleibt auch dann in der Nutzerdatenbank gespeichert, wenn er gerade kein Abo besitzt. Außerdem kann ein Nutzer den Abo-Typ ändern; dem Nutzer wird dann mitgeteilt, ob die gewünschte Änderung möglich ist.

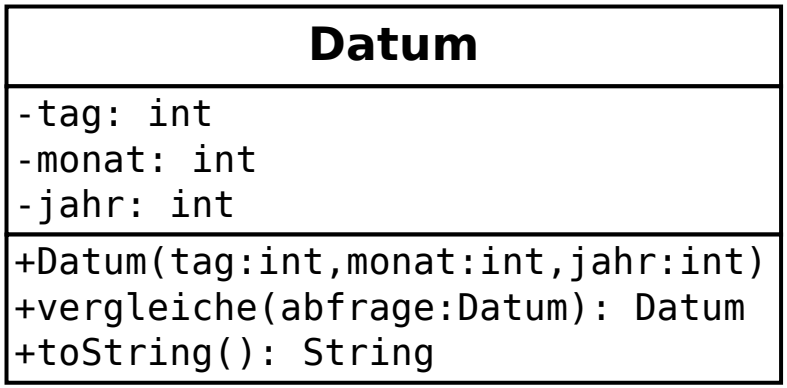
Bei allen Klassen soll genau ein Konstruktor angegeben werden. Getter und Setter sollen nur dann hinzugefügt werden, wenn sie explizit gefordert sind.

# Klasse Datum

Für ein Datum wird der Tag, der Monat und das Jahr jeweils als Zahl gespeichert. Ein Datum kann mit einem anderen Datum verglichen werden, das Ergebnis des Vergleiches ist wiederum ein Datum. Ein Datum kann in lesbarer Form ausgegeben werden.

# Klasse Datum

Für ein Datum wird der Tag, der Monat und das Jahr jeweils als Zahl gespeichert. Ein Datum kann mit einem anderen Datum verglichen werden, das Ergebnis des Vergleiches ist wiederum ein Datum. Ein Datum kann in lesbarer Form ausgegeben werden.

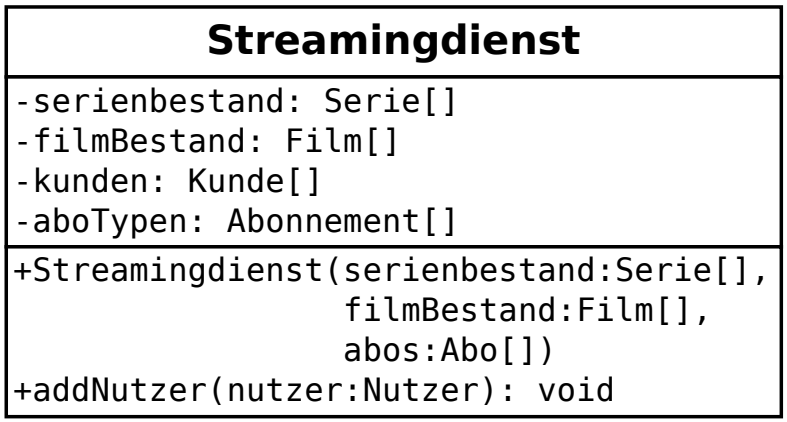


# Klasse Streamingdienst

Ein Streamingdienst hat mindestens 500 Filme und 20 Serien im Angebot. Das Angebot besteht dabei bereits bei der Gründung des Diensts. Nicht alle Filme und Serien sind exklusiv bei einem Streamingdienst zu sehen. Jeder Streamingdienst besitzt eine Nutzerdatenbank (die hier aus einer einfachen Liste bestehen soll), in der alle Nutzer gespeichert sind. Nutzer können nachträglich hinzugefügt werden. Manche Streamingdienste bieten verschiedene Abonnements (Abos) an, andere finanzieren sich über Werbung. Auch das Abomodell ändert sich nach Gründung des Streamingdienstes nicht.

# Klasse Streamingdienst

Ein Streamingdienst hat mindestens 500 Filme und 20 Serien im Angebot. Das Angebot besteht dabei bereits bei der Gründung des Diensts. Nicht alle Filme und Serien sind exklusiv bei einem Streamingdienst zu sehen. Jeder Streamingdienst besitzt eine Nutzerdatenbank (die hier aus einer einfachen Liste bestehen soll), in der alle Nutzer gespeichert sind. Nutzer können nachträglich hinzugefügt werden. Manche Streamingdienste bieten verschiedene Abonnements (Abos) an, andere finanzieren sich über Werbung. Auch das Abomodell ändert sich nach Gründung des Streamingdienstes nicht.

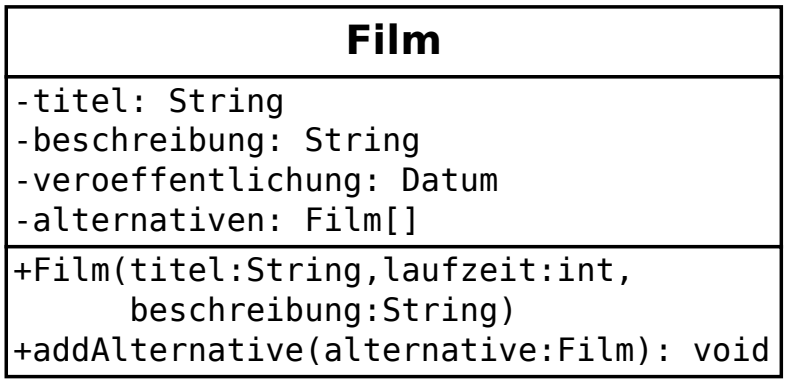


# Klasse Film

Für Filme wird der Titel, eine Beschreibung sowie das Datum der Veröffentlichung gespeichert. Außerdem wird bei Filmen vermerkt, welche alternativen Fassungen gibt; diese können nachträglich hinzugefügt werden.

# Klasse Film

Für Filme wird der Titel, eine Beschreibung sowie das Datum der Veröffentlichung gespeichert. Außerdem wird bei Filmen vermerkt, welche alternativen Fassungen gibt; diese können nachträglich hinzugefügt werden.

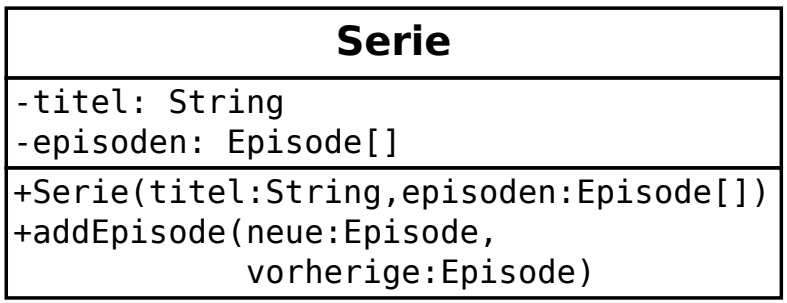


# Klasse Serie

Eine Serie hat einen Titel und besteht aus mindestens zwei Episoden. Es können neue Episoden hinzugefügt werden, dabei muss (damit die Reihenfolge eindeutig ist) neben der neuen auch die letzte alte Episode angegeben werden.

# Klasse Serie

Eine Serie hat einen Titel und besteht aus mindestens zwei Episoden. Es können neue Episoden hinzugefügt werden, dabei muss (damit die Reihenfolge eindeutig ist) neben der neuen auch die letzte alte Episode angegeben werden.

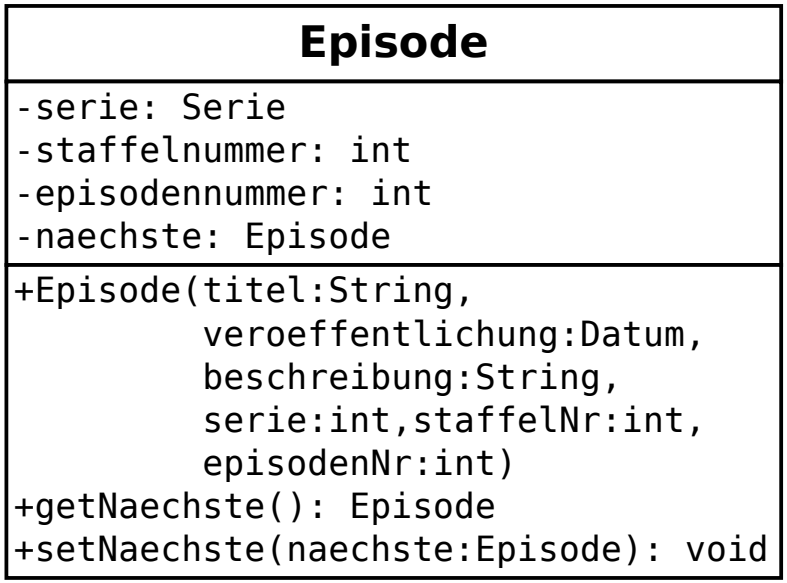


# Klasse Episode

Episoden sind Filme, bei denen zusätzlich die dazugehörige Serie sowie Staffel- und Episodennummer gespeichert wird. Für jede Episode wird die nachfolgende Episode (soweit vorhanden) gespeichert; diese kann auch abgefragt werden.

# Klasse Episode

Episoden sind Filme, bei denen zusätzlich die dazugehörige Serie sowie Staffel- und Episodennummer gespeichert wird. Für jede Episode wird die nachfolgende Episode (soweit vorhanden) gespeichert; diese kann auch abgefragt werden.

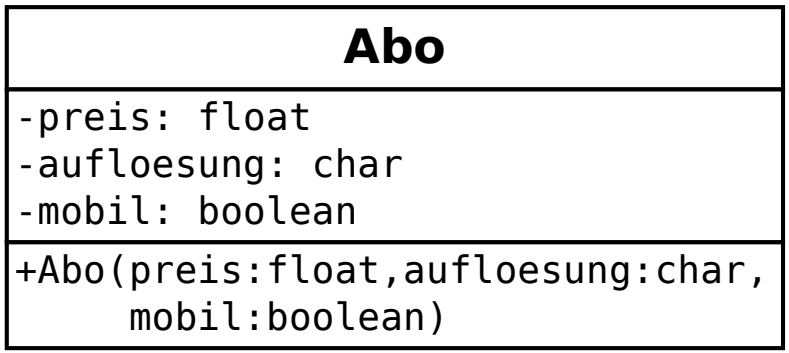


# Klasse Abo

Die Abos unterscheiden sich im Preis, in der maximalen Auflösung ('S' für SD, 'H' für HD usw.) sowie darin, ob mobiles Streamen unterstützt wird. Zwei Streamingdienste bieten niemals das gleiche Abo an.

# Klasse Abo

Die Abos unterscheiden sich im Preis, in der maximalen Auflösung ('S' für SD, 'H' für HD usw.) sowie darin, ob mobiles Streamen unterstützt wird. Zwei Streamingdienste bieten niemals das gleiche Abo an.

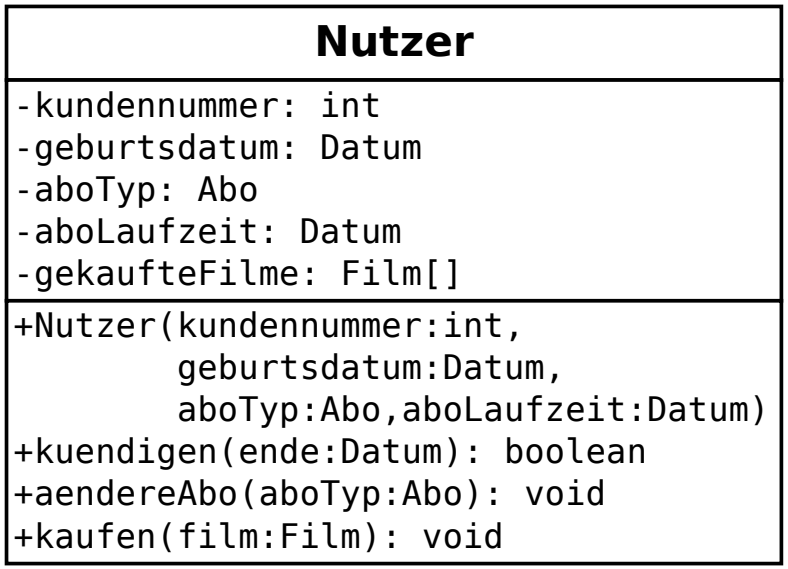


# Klasse Nutzer

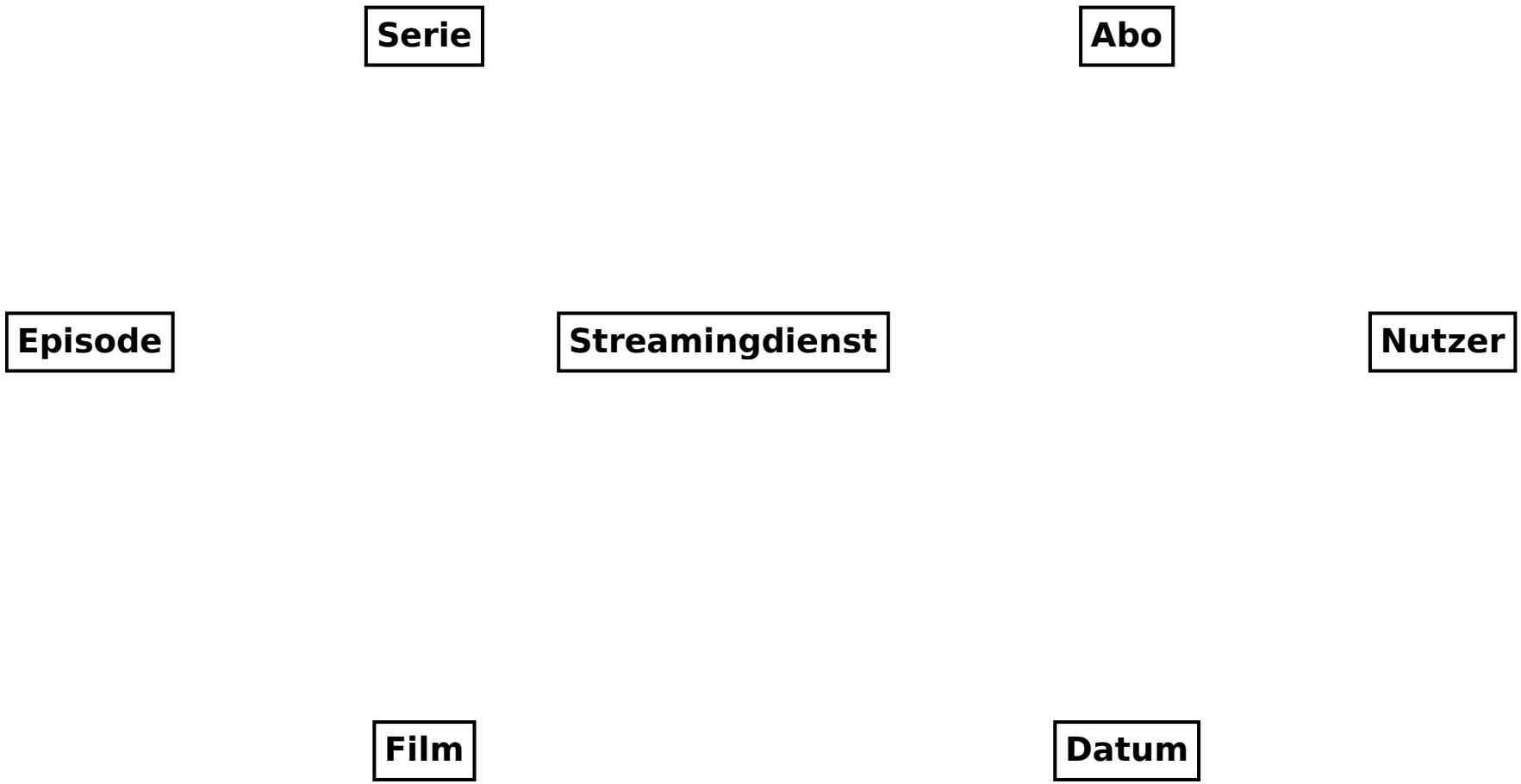
Für jeden Nutzer wird eine Kundennummer sowie der Typ und das Ablaufdatum des aktuellen Abonnements (sofern vorhanden) gespeichert. Weiterhin muss jeder Nutzer sein Geburtsdatum angeben. Nutzer können Filme kaufen. Für den Nutzer wird gespeichert, welche Filme bereits gekauft wurden. Ein Nutzer kann sein Abo unter Angabe des Kündigungsdatums kündigen, in diesem Fall wird das tatsächliche Kündigungsdatum ermittelt und dem Kunden mitgeteilt. Ein Nutzer bleibt auch dann in der Nutzerdatenbank gespeichert, wenn er gerade kein Abo besitzt. Außerdem kann ein Nutzer den Abo-Typ ändern; dem Nutzer wird dann mitgeteilt, ob die gewünschte Änderung möglich ist.

# Klasse Nutzer

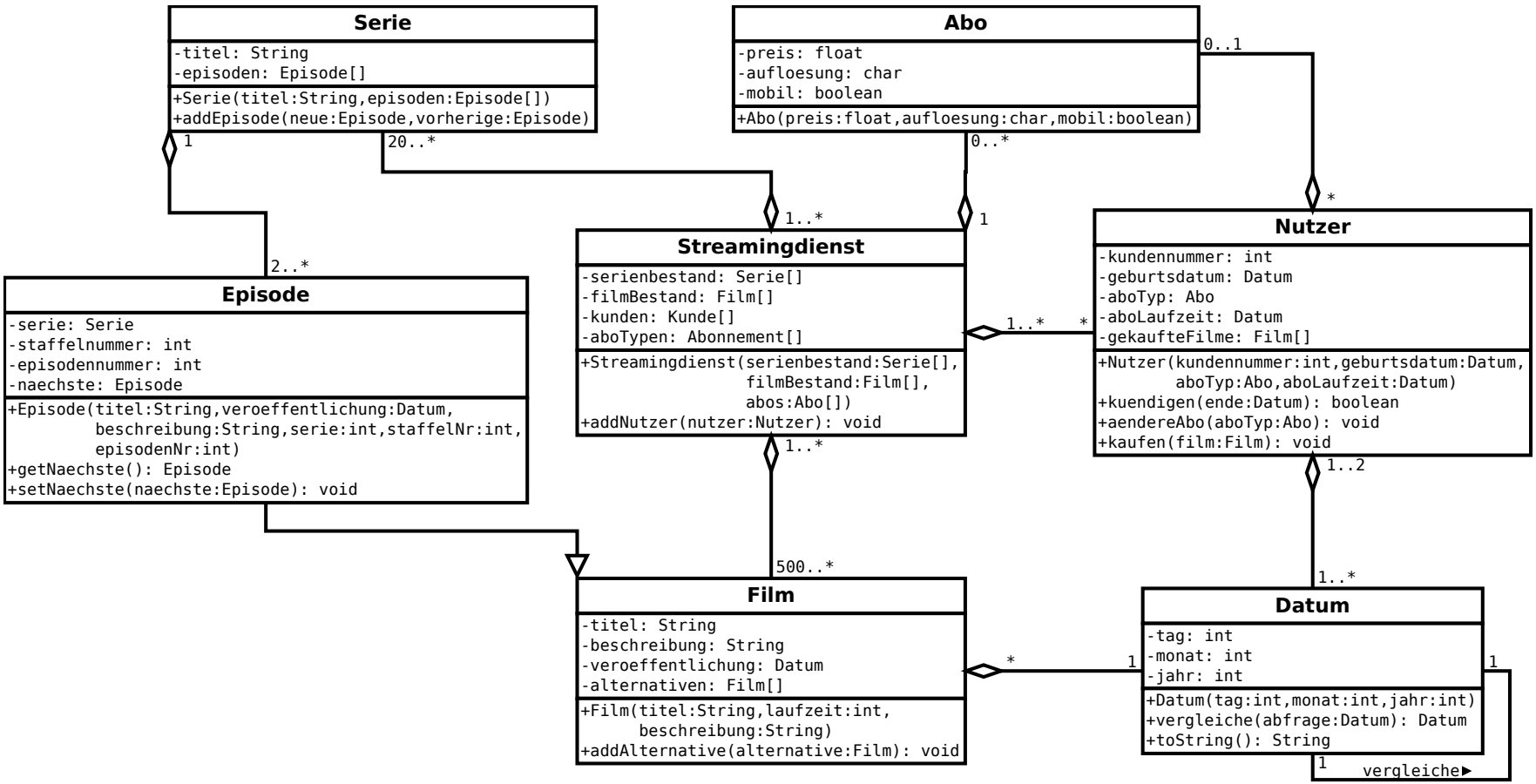
Für jeden Nutzer wird eine Kundennummer sowie der Typ und das Ablaufdatum des aktuellen Abonnements (sofern vorhanden) gespeichert. Weiterhin muss jeder Nutzer sein Geburtsdatum angeben. Nutzer können Filme kaufen. Für den Nutzer wird gespeichert, welche Filme bereits gekauft wurden. Ein Nutzer kann sein Abo unter Angabe des Kündigungsdatums kündigen, in diesem Fall wird das tatsächliche Kündigungsdatum ermittelt und dem Kunden mitgeteilt. Ein Nutzer bleibt auch dann in der Nutzerdatenbank gespeichert, wenn er gerade kein Abo besitzt. Außerdem kann ein Nutzer den Abo-Typ ändern; dem Nutzer wird dann mitgeteilt, ob die gewünschte Änderung möglich ist.



# Beziehungen und Multiplizitäten



# Beziehungen und Multiplizitäten



# Download der Übungsunterlagen

- Im Rahmen dieser Übung gibt es Folien und Codebeispiele
- Homepage des Lehrstuhls für Schwarmintelligenz und Komplexe Systeme:
  `http://pacosy.informatik.uni-leipzig.de`
- Homepage → Mitarbeiter → Stefan Preußner → MuP1
- Benutzer: `mup`
- Passwort: `MuP1WS20/21`

# Modellierung und Programmierung 1

Übung 4

Stefan Preußner

23. / 24. November 2020

# Die `main`-Methode

- In Java muss es mindestens eine Methode mit der folgenden Signatur geben:
  `public static void main(String[] args)`

  (die Variable `args` kann auch anders bezeichnet werden)
- Die `main`-Methode dient als Einstiegspunkt bei der Ausführung des Programms (die eigentliche Programmausführung besteht im Kern darin, die `main`-Methode abzuarbeiten)
- Mehrere Klassen können eine solche `main`-Methode besitzen. Bei der Ausführung des Programms muss dann die `main`-Methode angegeben werden, welche abgearbeitet werden soll
  - Die Klasse, welche diese `main`-Methode enthält, ist die Hauptklasse des Programms

# Statische und nicht-statische Methoden

In Java gibt es statische und nicht-statische Methoden:

- **Nicht-Statische** Methoden
  - können immer nur für konkrete Objekte aufgerufen werden, der Aufruf erfolgt mit der Syntax `Objektname.Methodenname`
  - stellen über das Schlüsselwort `this` eine Referenz auf das Objekt, für das eine Methode aufgerufen wurde, zur Verfügung
  - haben Zugriff auf statische und nicht-statische Attribute und Methoden der Klasse, zu der sie gehören

# Statische und nicht-statische Methoden

- **Statische** Methoden
  - werden durch das Schlüsselwort `static` gekennzeichnet: `public static void main(String[] args)`
  - können über den Befehl `Klassenname.Methodenname` an einer beliebigen Stelle im Code aufgerufen werden
    - Bspw. gibt es in der Klasse `Math` die Methode `public static int round(float x)`, damit kann an einer beliebigen Stelle im Code mittels `Math.round(zahl)` eine `float`-Variable namens `zahl` gerundet werden
  - können von der Klasse, zu der sie gehören, nur statische Attribute und andere statische Methoden verwenden

# Scope / Geltungsbereich von Variablen

- Jede Variable hat einen Geltungsbereich (*scope*), bspw.
  - sind lokale Variablen nur innerhalb eines Anweisungsblocks,
  - Parameter nur innerhalb einer Methode und
  - Instanzvariablen nur innerhalb einer Klasse gültig
- Geltungsbereiche können sich überlagern (variable shadowing), bspw. wenn der Parameter einer Funktion und das Attribut einer Klasse den gleichen Namen haben
  - in diesem Fall kann mit `this` explizit auf das Klassenattribut zugegriffen werden

# super

- Das Schlüsselwort `super` referenziert in Java die direkte Oberklasse und hat zwei Funktionen:
- Mit `super()` wird der Konstruktor der Oberklasse aufgerufen
  - `super()` können auch Argumente übergeben, dann wird der entsprechend parametrisierte Konstruktor der Oberklasse aufgerufen
  - `super()` **muss immer die erste Anweisung in einem Konstruktor sein!**
  - Wenn kein Aufruf von super() angegeben ist, dann wird dieser automatisch eingefügt - d.h. es erfolgt immer ein Aufruf des Konstruktors der Oberklasse

# super

- Mit `super` kann auf Methoden sowie Instanz- und Klassenvariablen der Oberklasse zugegriffen werden
  - Bsp.: super.toString() ruft die toString()-Methode der Oberklasse auf
  - Diese Funktionalität ist insbesondere bei überschriebenen Methoden sinnvoll
  - Der Zugriff auf Variablen ist nur insoweit möglich, wie deren Sichtbarkeit dies zulässt

# instanceof

- Mit dem Schlüsselwort `instanceof` kann die Zugehörigkeit eines Objekts zu einer Klasse geprüft werden
- Syntax: `Objekt instanceof Klassenname`
- Es wird dann `true` zurückgegeben, wenn das Objekt zu einer Klasse gehört, die entweder der angegebenen Klasse entspricht oder eine ihrer Unterklassen ist
  - Bsp.: `Episode` erbt von `Film`. `ep` sei eine Instanz der Klasse `Episode`, `fi` sei eine Instanz der Klasse `Film`.
    - `ep instanceof Episode` → `true`
    - `ep instanceof Film` → `true`
    - `fi instanceof Episode` → `false`
    - `fi instanceof Film` → `true`

# Typenkonvertierung / type casting

- Der Wert einer Variablen kann, unter bestimmten Bedingungen, einer Variablen mit einem anderen Datentyp zugewiesen werden
- Hierzu wird der Zieldatentyp in Klammern vor den umzuwandelnden Wert gesetzt
- Beispiel:

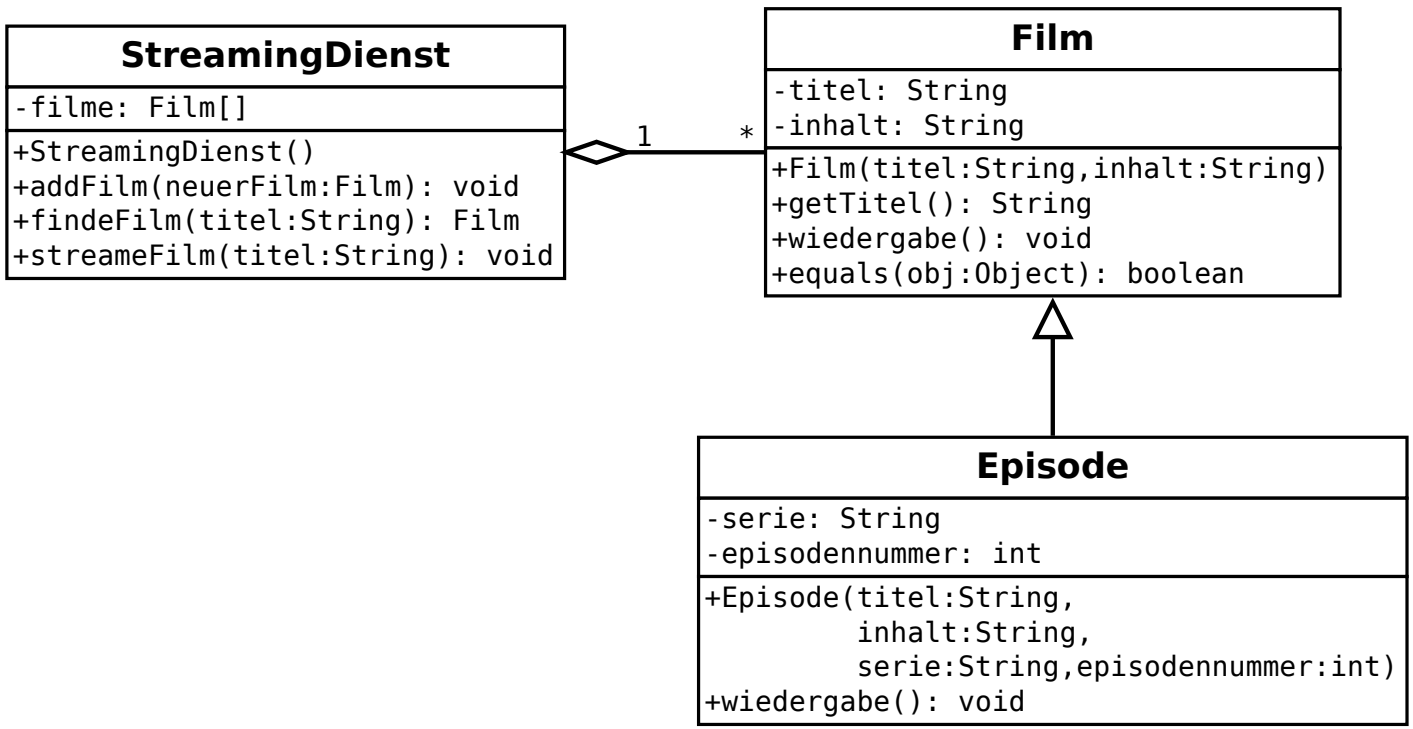
```
double x = 3.5;
int y = (int)x; // y hat den Wert 3
```

# Typenkonvertierung / type casting

- Umgewandelt werden können u.a.:
  - Primitive Zahlendatentypen untereinander (`char` ist auch eine Zahl!)
  - Objekte einer Klasse X in Objekte einer Oberklasse von X
    - Wichtig: Die Information, zu welcher Klasse ein Objekt ursprünglich gehörte, wird zusammen mit den Instanzvariablen bei einer Typenkonvertierung weiterhin gespeichert
- Nicht umgewandelt werden können u.a.:
  - `boolean` in primitive Zahlendatentypen und umgekehrt
  - Objekte einer Klasse X in Objekte einer Unterklasse von X
    - Ausnahme: das Objekt gehört ursprünglich der Unterklasse an

# Übung - Programmierung

Implementiert werden soll folgendes UML-Klassendiagramm:



# Klasse StreamingDienst

- `filme` speichert alle Filme des Streamingdienstes
- Der Konstruktor soll ein neues `Film`-Array der Länge 100 erzeugen
- `addFilm` soll den übergebenen Film an der ersten freien Stelle in `filme` speichern
- `findeFilm` soll den Film mit dem übergebenen Titel zurückgeben. Existiert kein solcher Film, dann soll `null` zurückgegeben werden
- `streameFilm` soll den Film mit dem übergebenen Titel wiedergeben. Existiert kein solcher Film, dann soll eine Fehlermeldung ausgegeben werden

# Klasse Film

- `titel` speichert den Titel, `inhalt` den Inhalt eines Films
- `wiedergabe` soll erst den Titel und dann den Inhalt ausgeben
- `equals` soll dann `true` zurückgeben, wenn das Vergleichsobjekt ein `Film` mit dem gleichen Titel ist

# Klasse Episode

- `serie` speichert den Namen der Serie, zu der die Episode gehört
- `episodennummer` speichert die Nummer der Episode innerhalb der Serie
- `wiedergabe` soll vor der Ausgabe des Titels und des Inhalts (wie in der Klasse `Film`) die Serie und die Episodennummer ausgeben

# Klasse Main

- In der `main`-Methode soll ein neuer Streamingdienst erstellt werden
- Neue Filme und Episoden sollen erstellt und dem Streamingdienst hinzugefügt werden
- Anschließend soll ein beim Streamingdienst gespeicherter Film wiedergegeben werden

# Modellierung und Programmierung 1

Übung 5

Stefan Preußner

30. November / 1. Dezember 2020

# Zufallszahlen

- Die Klasse `Random` aus dem Paket `java.util` ermöglicht die Generierung von (Pseudo-)Zufallszahlen
  - Alle Klassen, die sich nicht im Paket `java.lang` befinden, müssen importiert werden. Hier ist also der Befehl `import java.util.Random` erforderlich.
- Der Zufallszahlengenerator wird mit einem *seed* initialisiert
  - Wenn dieser nicht explizit angegeben wird, dann wird die Systemzeit zur Berechnung des seeds genutzt
  - Mit der Methode `setSeed(long seed)` kann der seed explizit gesetzt werden
    - Dies vereinfacht die Fehlersuche, da die Zufallszahlen dann reproduzierbar generiert werden

# Zufallszahlen

- `nextInt()` erzeugt einen neuen int zwischen `Integer.MIN_VALUE` und `Integer.MAX_VALUE` (zwei Konstanten, die in Java den kleinsten bzw. größten durch einen int darstellbaren Wert speichern)
- `nextInt(int wert)` erzeugt einen neuen int zwischen 0 (eingeschlossen) und `wert` (nicht eingeschlossen)
- `nextFloat()` erzeugt einen neuen float zwischen 0.0 (eingeschlossen) und 1.0 (nicht eingeschlossen)
- `nextDouble()` erzeugt einen neuen double zwischen 0.0 (eingeschlossen) und 1.0 (nicht eingeschlossen)
  - Die Auflösung ist hier höher als bei `nextFloat()`

# Zufallszahlen

- Erzeugung einer zufälligen Ganzzahl zwischen `untereGrenze` und `obereGrenze`:
  - `nextInt(obereGrenze - untereGrenze) + untereGrenze`
  - `(int)(nextDouble() * (obereGrenze - untereGrenze)) + untereGrenze`

# Formatierte Strings

- Formatierte Strings können mit der statischen Methode `String.format` erzeugt werden
- Syntax: `String.format(Formatstring, Wert1, Wert2, ...)`
- `Formatstring` ist hierbei ein String, welcher neben beliebigen Text auch Formatparameter enthalten kann - diese werden in der Reihenfolge, in der sie im Formatstring auftreten, durch die angegebenen Werte ersetzt
  - Formatparameter beginnen mit einem Prozentzeichen %
  - Eine ausführliche Beschreibung findet sich in der Java-Dokumentation der Klasse `java.util.Formatter`

# Formatierte Strings

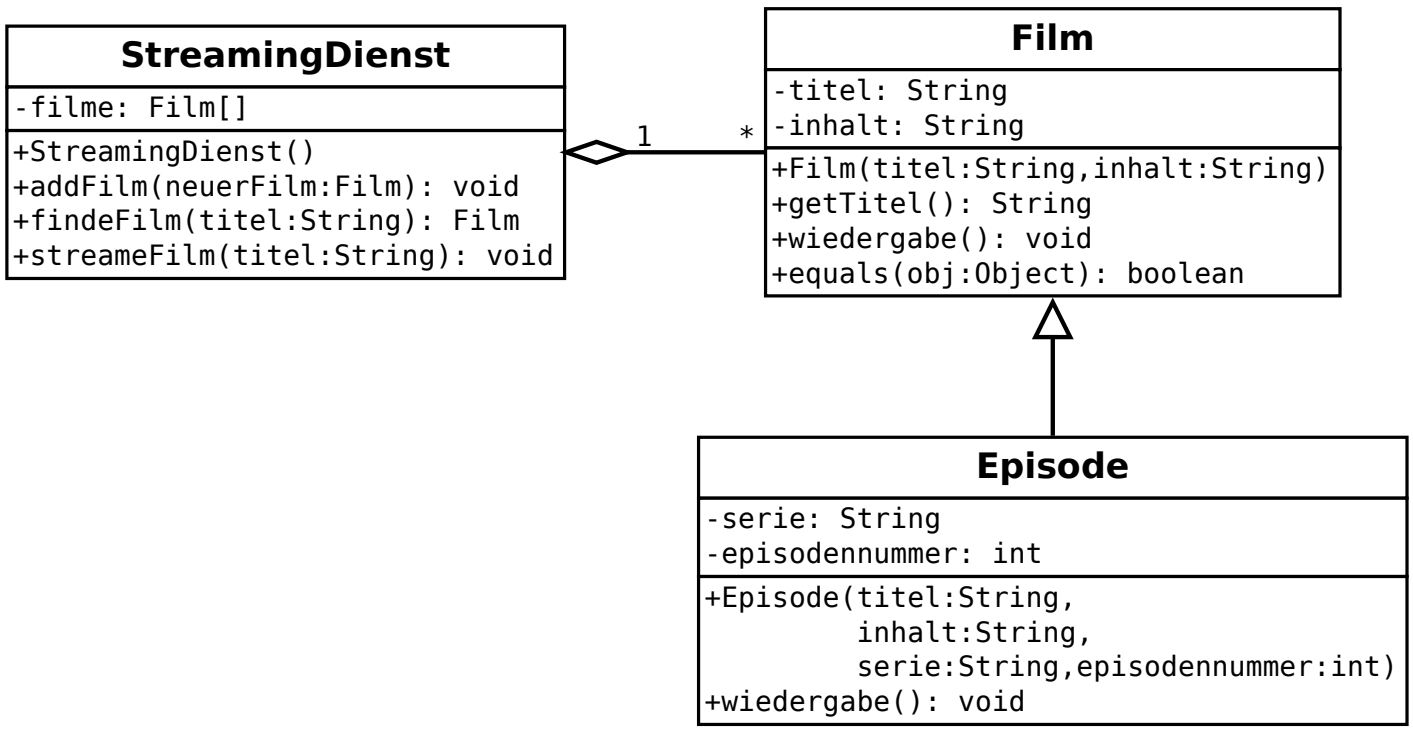
- Beispiel 1:
  - `String.format("%s %s %s", "MuP", "ist", "toll")`
  - Rückgabe: `"MuP ist toll"`
- Beispiel 2:
  - `String.format("Ein Würfel hat %2d Seiten", 6)`
  - Rückgabe: `"Ein Würfel hat 6 Seiten"`
  - `d` steht für eine Ganzzahl, die Zahl davor ist optional und gibt die Mindestzahl an auszugebenden Zeichen an

# Formatierte Strings

- Beispiel 3:
  - `String.format("2 durch 3 ist %10.8f", 2.0/3)`
  - Rückgabe: `2 durch 3 ist 0.66666667`
  - `f` steht für eine Fließkommazahl
  - Die Zahl vor dem Punkt gibt die Gesamtzahl an auszugebenden Zeichen, einschließlich des Dezimalpunkts, an
  - Die Zahl nach dem Punkt gibt die Anzahl an auszugebenden Nachkommastellen an
  - Es wird automatisch gerundet

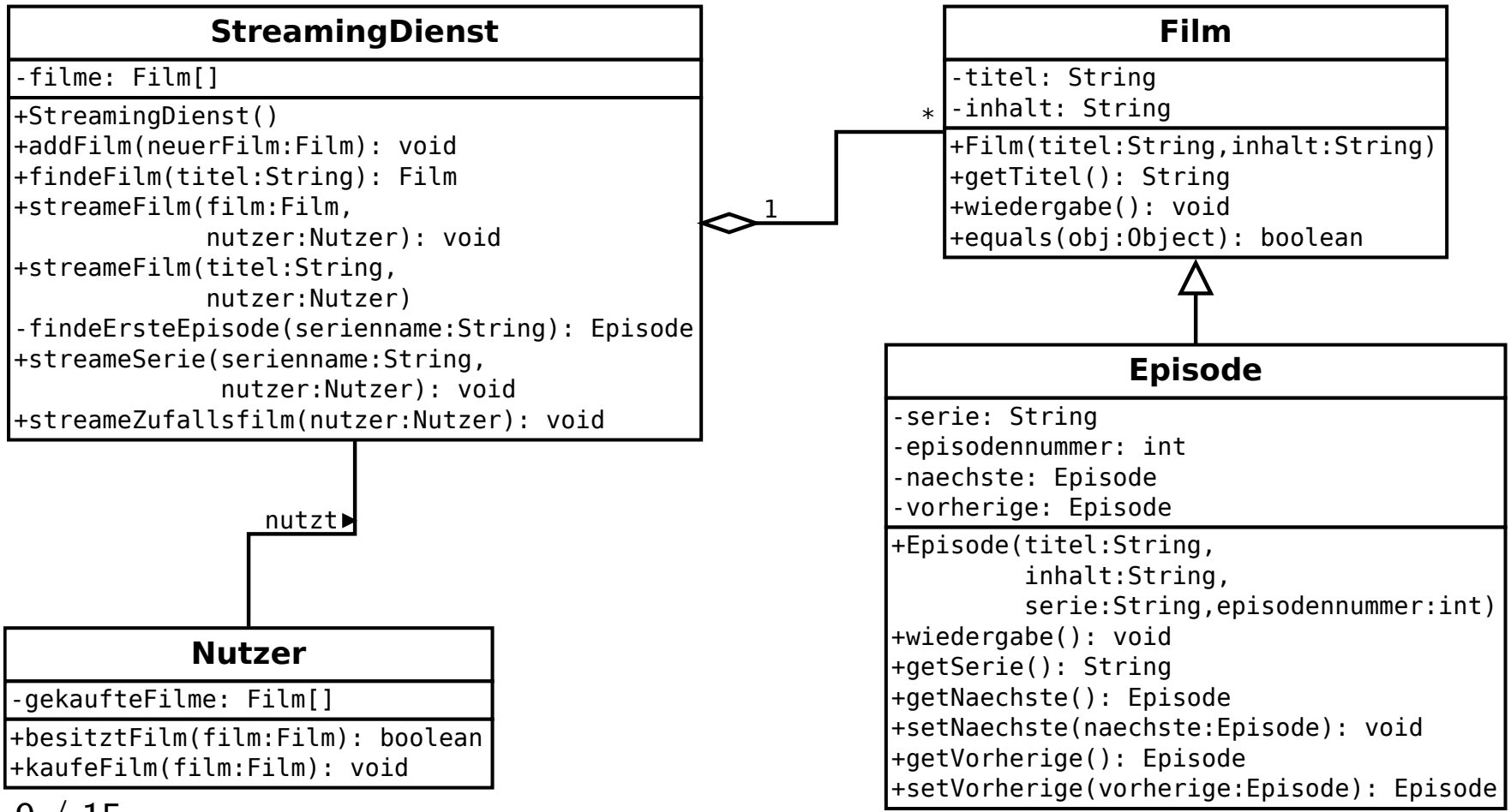
# Übung - Programmierung

UML-Diagramm aus Übung 4:



# Übung - Programmierung

## Erweitertes UML-Diagramm:



# Klasse StreamingDienst

- `filme` speichert alle Filme des Streamingdienstes
- Der Konstruktor soll ein neues `Film`-Array der Länge 100 erzeugen
- `addFilm` soll den übergebenen Film an der ersten freien Stelle in `filme` speichern
- `findeFilm` soll den Film mit dem übergebenen Titel zurückgeben. Existiert kein solcher Film, dann soll `null` zurückgegeben werden
- `streameFilm(Film film, Nutzer nutzer)` soll einen Film kaufen lassen und dann wiedergeben

# Klasse StreamingDienst

- `streameFilm(String titel, Nutzer nutzer)` sucht einen Film mit dem angegebenen Titel. Ist der Film vorhanden, dann wird er gekauft und wiedergegeben.
- `findeErsteEpisode` findet die erste Episode einer Serie mit dem angegebenen Seriennamen. Existiert keine solche Serie, dann wird `null` zurückgegeben.
- `streameSerie` gibt alle Episoden einer Serie wieder
- `streameZufallsfilm` gibt einen zufälligen Film wieder

# Klasse Film

- `titel` speichert den Titel, `inhalt` den Inhalt eines Films
- `wiedergabe` soll erst den Titel und dann den Inhalt ausgeben
- `equals` soll dann `true` zurückgeben, wenn das Vergleichsobjekt ein `Film` mit dem gleichen Titel ist

# Klasse Episode

- `serie` speichert den Namen der Serie, zu der die Episode gehört
- `episodennummer` speichert die Nummer der Episode innerhalb der Serie
- `vorherige` bzw. `naechste` geben die vorherige bzw. nächste Episode innerhalb einer Serie an
- `wiedergabe` soll vor der Ausgabe des Titels und des Inhalts (wie in der Klasse `Film`) die Serie und die Episodennummer ausgeben
- `setNaechste` soll nicht nur `naechste` setzen, sondern auch das Attribut `vorherige` bei der nächsten Episode

# Klasse Nutzer

- `gekaufteFilme` speichert bis zu 100 vom Nutzer gekaufte Filme
- `besitztFilm` gibt zurück, ob ein Nutzer einen Film bereits gekauft hat
- `kaufeFilm` fügt den angegebenen Film zur Liste der gekauften Filme hinzu, falls der Nutzer den Film nicht bereits besitzt

# Klasse Main

- In der `main`-Methode soll ein neuer Streamingdienst erstellt werden
- Neue Filme und Episoden sollen erstellt und dem Streamingdienst hinzugefügt werden
- Anschließend sollen ein Film mit bekanntem Titel, eine Serie sowie ein zufälliger Film wiedergegeben werden

# Modellierung und Programmierung 1

Übung 6

Stefan Preußner

7./ 8. Dezember 2020

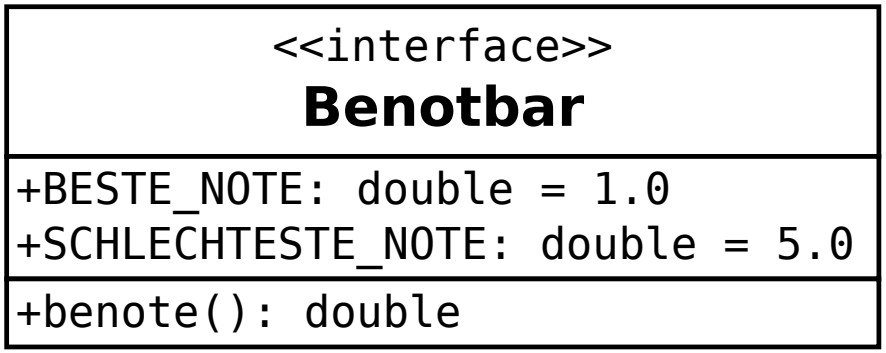
# Generics

siehe `Paar.java` und `PaarMain.java`

# Schnittstellen (Interfaces)

- In Java gibt es keine Mehrfachvererbung, eine Klasse kann also nicht von mehreren Klassen erben
- Soll eine Klasse mehrere Typen haben, so kann sie **Schnittstellen** - **Interfaces** - implementieren
- Eine Schnittstelle legt fest, welche Methoden eine Klasse besitzen muss, stellt aber selbst keine Implementierung zur Verfügung

# Schnittstellen (Interfaces)




```java
public interface Benotbar
{
    double BESTE_NOTE = 1.0;
    double SCHLECHTESTE_NOTE = 5.0;

    double benote();
}
```

# Schnittstellen (Interfaces)

- Von einer Schnittstelle können keine Objekte erzeugt werden, nur von den sie implementierenden Klassen
- Eine Schnittstelle darf deshalb keinen Konstruktor haben
- Die Methoden eines Interfaces sind automatisch `public` und `abstract`
  - `abstract` Methoden werden nur deklariert, aber nicht implementiert
  - Die Deklaration einer Schnittstellenmethode enthält nur Modifizierer, Rückgabetyp und Signatur

# Variablen in Schnittstellen

- Instanzvariablen sind immer Teil einer Implementierung; da Schnittstellen keine Implementierung enthalten, besitzen sie auch keine Instanzvariablen
- In Schnittstellen können Konstanten festgelegt werden
  - Diese müssen `public`, `static` und `final` sein
  - Erweitert eine Schnittstelle eine andere Schnittstelle, so kann sie deren Konstanten mit eigenen Werten überschreiben

# Implementierung

- Schlüsselwort `implements`

```
public class Klausur implements Benotbar
```

- Mehrere Interfaces können durch Kommata getrennt werden:

```
public class Klausur implements Benotbar, Comparable<Klausur>
```

# Implementierung

- Eine Klasse **muss** alle Methoden eines Interfaces implementieren
  - Dies gilt nicht für Klassen, welche `abstract` sind (von abstrakten Klassen können keine Instanzen erzeugt werden, daher müssen sie keine Methoden implementieren)
  - Wird ein Interface erweitert, müssen alle Klassen, welche das Interface implementieren, entsprechend angepasst werden
- Implementierte Methoden **müssen** `public` sein

# Schnittstellen und instanceof

`instanceof` funktioniert bei Schnittstellen wie bei Klassen. Im obigen Beispiel geben die Tests

```
Klausur mup;
mup instanceof Klausur;
mup instanceof Benotbar;
mup instanceof Comparable;
```

alle `true` zurück.

# Array vs. ArrayList

- Arrays und ArrayLists sind beides Strukturen, welche Daten eines vorgegebenen Typs sequenziell speichern
- Arrays haben eine feste Größe; Operationen sind im wesentlichen auf auf das Auslesen und Setzen von Elementen und die Ermittlung der Größe beschränkt
- Die Größe von ArrayLists ist dynamisch, es gibt zahlreiche Funktionen zur Manipulation der Liste

- Die Klasse `ArrayList` muss aus `java.util` importiert werden:
  - `import java.util.ArrayList`

  oder
  - `import java.util.*`
- Eine neue, leere `ArrayList`, welche Objekte vom Typ `Datentyp` speichert, wird wie folgt erzeugt:

```
ArrayList<Datentyp> listenname = new ArrayList<Datentyp>();
```

- `List` ist eine Schnittstelle, welche Basismethoden zur Listenmanipulation zur Verfügung stellt und von zahlreichen Listenklassen implementiert wird (`ArrayList`, `LinkedList`, `Stack`, `Vector`, ...).
- Oft ist ein Deklaration als `List` anstelle einer `ArrayList` besser:

```
List<Datentyp> listenname = new ArrayList<Datentyp>();
```

(Grund: soll später der Datentyp von `listenname` geändert werden, weil eine andere Listenimplementierung von Java besser geeignet ist, muss nur diese eine Zeile im Code verändert werden)

Einige nützliche Funktionen einer ArrayList<Datentyp>:

- `add(Datentyp obj)` - fügt das Objekt `obj` am Ende der Liste ein
- `add(int index, Datentyp obj)` - fügt das Objekt `obj` an der angegebenen Stelle `index` ein
- `addAll(Collection<Datentyp> objs)` - fügt alle Objekte aus einer `Collection` (hierzu zählen auch ArrayLists) am Ende der Liste ein
- `removeAll(Collection<Datentyp> objs)` - entfernt alle Elemente, die in der Collection `objs` enthalten sind, aus der Liste

Einige nützliche Funktionen einer ArrayList<Datentyp>:

- `contains(Object obj)` - testet, ob die Liste das Objekt `obj` enthält
- `get(int index)` - liefert das Element an der Stelle `index` zurück
- `remove(int index)` - entfernt das Element an der Stelle `index` und gibt es zurück
- `remove(Object obj)` - entfernt das erste Element in der Liste, welches `obj` entspricht
- `set(int index, Datentyp element)` - ersetzt das Element an der Stelle `index` mit `element`

Einige nützliche Funktionen einer ArrayList<Datentyp>:

- `size()` - liefert die Länge der Liste
- `toArray()` - wandelt die ArrayList in ein Array `Datentyp[]` um
- `isEmpty()` - testet, ob die Liste leer ist
- `iterator()` - erzeugt einen Listeniterator

# Interface `java.lang.Comparable`

- `java.lang.Comparable` ist nützlich, um Objekte leicht sortieren zu können
- Es muss lediglich die Funktion `compareTo` implementiert werden. `x.compareTo(y)` soll folgende Werte zurückgeben:
  - eine Zahl größer 0, wenn `x` in einer sortierten Liste nach `y` stehen soll
  - 0, wenn beide Objekte gleich(wertig) sind
  - eine Zahl kleiner 0, wenn `x` in der sortierten Liste vor `y` stehen soll

```
public class Klausur implements Comparable<Klausur>
{
    private double note;

    public int compareTo(Klausur query)
    {
        if (this.note < query.note)
        {
            return -1;
        }
        if (this.note == query.note)
        {
            return 0;
        }
        return 1;
    }
}
```

Die Sortierung kann dann mit `Collections.sort()` (oder einer der Sortierfunktionen aus den von `Collections` abgeleiteten Klassen) erfolgen:

```
List<Klausur> klausuren = new ArrayList<Klausur>();
klausuren.add(new Klausur( 1.3 ));
klausuren.add(new Klausur( 3.7 ));
klausuren.add(new Klausur( 2.0 ));

Collections.sort(klausuren);
```

Die Liste `klausuren` ist nun so sortiert, dass die Klausur mit der besten Note ganz vorne und die Klausur mit der schlechtesten Note ganz hinten in der Liste steht.

# Assoziative Datenfelder / Dictionary / Map

- Interface: `java.util.Map`
- Beispiele für implementierende Klassen: HashMap, TreeMap
- Die Klassen repräsentieren *assoziative Datenfelder*: gespeichert werden Paare bestehend aus einem Schlüssel und einem Wert - mit jedem Schlüssel ist also ein Wert assoziiert
- Jeder Schlüssel darf in einer `Map` nur einmal vorkommen

# HashMap

- **HashMap** speichert die Schlüssel-Wert-Paare in einer Hashtabelle, vom Schlüssel wird also der Hash berechnet
  - Für eigene Klassen müssen i.d.R. `equals()` und `hashCode()` überschrieben werden
  - Das Einfügen und Suchen von Schlüsseln erfolgt im Optimalfall in konstanter Zeit

# TreeMap

- **TreeMap** speichert die Daten anhand des Schlüssels sortiert in einem balanzierten Binärbaum
  - Die Schlüssel müssen vergleichbar sein - bei eigenen Klassen muss wenigstens eins der Interfaces `Comparable` oder Comparator implementiert sein
  - In der Regel langsamer als HashMap

- Eine neue HashMap wird wie folgt erzeugt:

```
Map<K, V> variablenname = new HashMap<K, V>();

Map<String, Integer> meinemap = new HashMap<String, Integer>();
```

  - `K` ist der Datentyp des Schlüssels
  - `V` ist der Datentyp des gespeicherten Werts
- Ein neuer Wert wird mit `put` gespeichert:

```
Map.put(K, V);

meinemap.put("Hallo", 12345)
```

- Der zu einem Schlüssel gehörende Wert wird mit `get` zurückgegeben:

```
V variablenname = Map.get(K);

Integer meinint = meinemap.get("Hallo");
```

  - Ist ein Schlüssel nicht vorhanden, so wird `null` zurückgegeben
- Mit `containsKey` kann überprüft werden, ob ein Schlüssel in der Map vorkommt:

```
boolean Map.containsKey(K);

boolean meinboolA = meinemap.containsKey("Test");  // false
boolean meinboolB = meinemap.containsKey("Hallo"); // true
```

- Mit `containsValue` kann überprüft werden, ob ein Wert in der Map vorkommt:

```
boolean Map.containsValue(V);

boolean meinboolA = meinemap.containsValue(7890);  // false
boolean meinboolB = meinemap.containsValue(12345);  // true
```

  - Achtung: der Sinn von Maps ist es, anhand eines bestimmten Schlüssels schnell auf Daten zugreifen zu können. `containsKey` ist deshalb sehr schnell, `containsValue` dagegen sehr langsam!

- Die Menge aller Schlüssel lässt sich mit `keySet` zurückgeben:

```
Set<K> variablenname = Map.keySet();

Set<String> meinset = meinemap.keySet();
```

- Die Menge aller Schlüssel-Wert-Paare erhält man mit `entrySet()`:

```
Set<Map.Entry<K,V>> variablenname = Map.entrySet();

Set<Map.Entry<String,Integer>> meinset = meinemap.entrySet();
```

  - Über dieses Set kann iteriert werden, um nacheinander auf alle Schlüssel-Wert-Paare zuzugreifen
  - Map.Entry besitzt die Methoden `getKey()` und `getValue()` für den Zugriff auf den Schlüssel bzw. den Wert

# Die Klassen Integer und Double

- `int` und `double` sind primitive Datentypen in Java
  - Primitive Datentypen sind keine Klassen, sie haben daher keine Attribute oder Methoden
- Durch die Klassen Integer und Long bzw. Float und Double können Ganzzahlen bzw. Gleitkommazahlen als Objekte dargestellt werden
- Die Erzeugung eines neuen Integer- bzw. Double-Objekts erfolgt wie bei anderen Klassen:

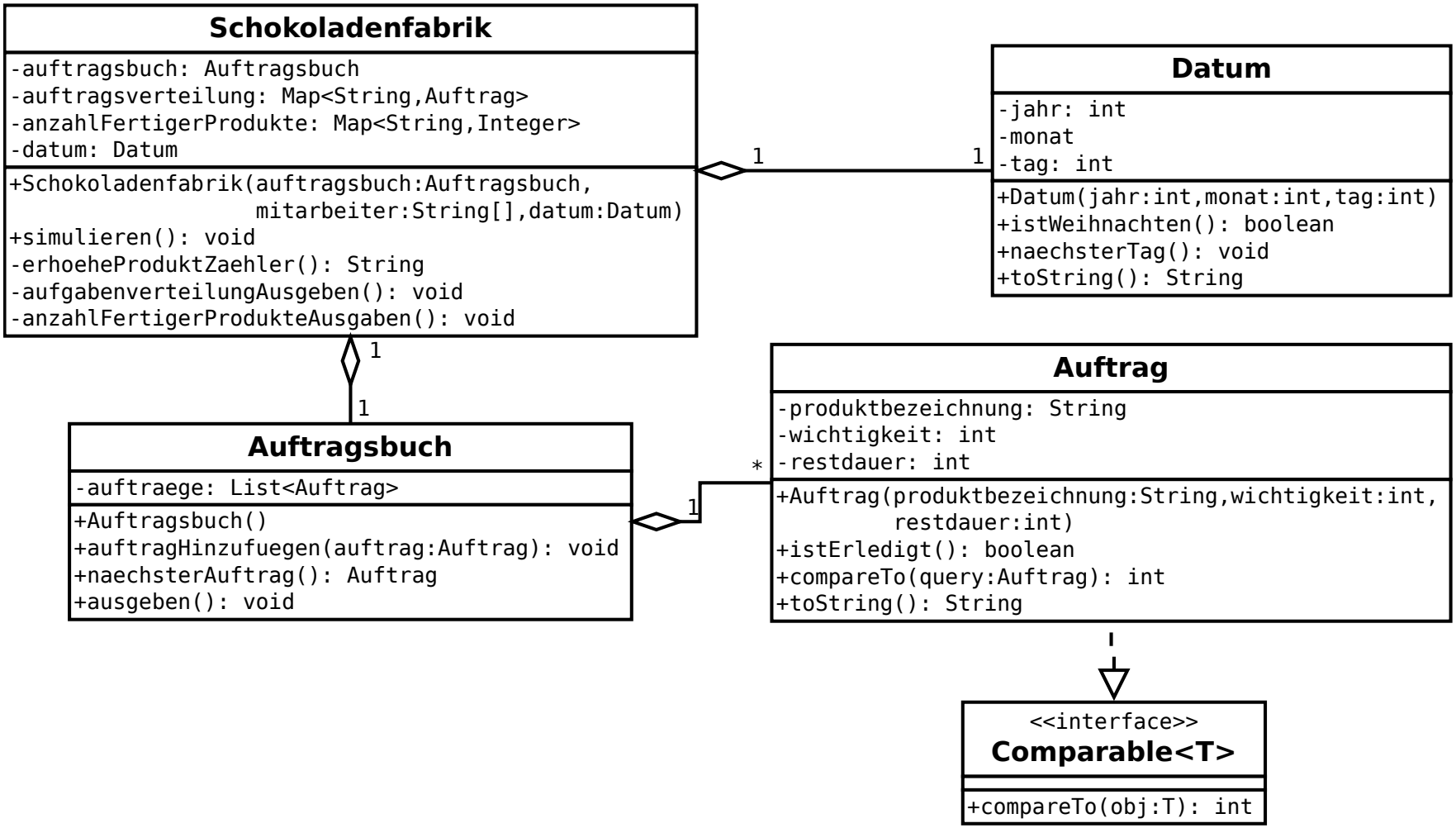
```
Double d = new Double(3.1);
Integer i = new Integer(13);
```

# Java-Programmierung - List, Map, Comparable

Programmierübung:

Simulation der Produktion von Süßigkeiten in einer Schokoladenfabrik in den Tagen vor Weihnachten

# Java-Programmierung - List, Map, Comparable



# Java-Programmierung - List, Map, Comparable

Klasse `Datum`

- `istWeihnachten` gibt `true` zurück, wenn es der 24.12. eines beliebigen Jahres ist
- `naechsterTag` ändert das Datum auf den nächsten Tag und berücksichtigt dabei Monats- bzw. Jahreswechsel (aber keine Schaltjahre)
- `toString` gibt das Datum im Format Tag.Monat.Jahr zurück

# Java-Programmierung - List, Map, Comparable

Klasse `Auftrag`

- Die Restdauer muss bei der Erzeugung eines neuen Auftrags mindestens 1 betragen
- Die Wichtigkeit muss einen Wert zwischen 0 und 3 haben
- `istErledigt` gibt `true` zurück, wenn die Restdauer 0 ist
- `compareTo` soll so implementiert werden, dass Aufträge nach absteigender Wichtigkeit sortiert werden. Bei gleicher Wichtigkeit soll nach absteigender Restdauer sortiert werden.
- `toString` gibt die Produktbezeichnung, Wichtigkeit und Restdauer als String zurück

# Java-Programmierung - List, Map, Comparable

Klasse `Auftragsbuch`

- `auftragHinzufuegen` fügt den übergebenen Auftrag zur Liste aller Aufträge hinzu und sortiert die Liste anschließend
- `naechsterAuftrag` entfernt den ersten Auftrag aus der Auftragsliste und gibt ihn zurück. Ist die Auftragsliste leer, dann soll `null` zurückgegeben werden.
- `ausgeben` gibt alle Aufträge aus

# Java-Programmierung - List, Map, Comparable

Klasse `Schokoladenfabrik`

- `simulieren` simuliert tagesweise den Zeitraum zwischen `datum` und Weihnachten
  - Jedem Mitarbeiter wird, solange das Auftragsbuch noch Aufträge enthält, ein Auftrag zugewiesen
  - Bei jedem Auftrag, der einem Mitarbeiter zugewiesen wurde, verringert sich die Restdauer jeden Tag um einen Tag
  - Ist ein Auftrag erledigt, wird dem Mitarbeiter ein neuer Auftrag zugewiesen und der Zähler für das entsprechende Produkt um 1 erhöht
  - Es werden täglich die Auftragsverteilung und Zahl der insgesamt fertiggestellten Produkte ausgegeben

# Java-Programmierung - List, Map, Comparable

Klasse `Schokoladenfabrik`

- `auftragsverteilung` bildet jeden Mitarbeiter (repräsentiert durch seinen Namen) auf einen Auftrag ab
- `anzahlFertigerProdukte` gibt an, wie oft ein Produkt (repräsentiert durch seine Bezeichnung) bereits fertiggestellt wurde
- `erhoeheProduktZaehler` erhöht den Zähler für ein fertiggestelltes Produkt um 1
- `aufgabenverteilungAusgeben` gibt die aktuelle Aufgabenverteilung aus
- `anzahlFertigerProdukteAusgaben` gibt den Produktzähler aus

# Modellierung und Programmierung 1

Übung 7

Stefan Preußner

14./ 15. Dezember 2020

# Organisatorisches

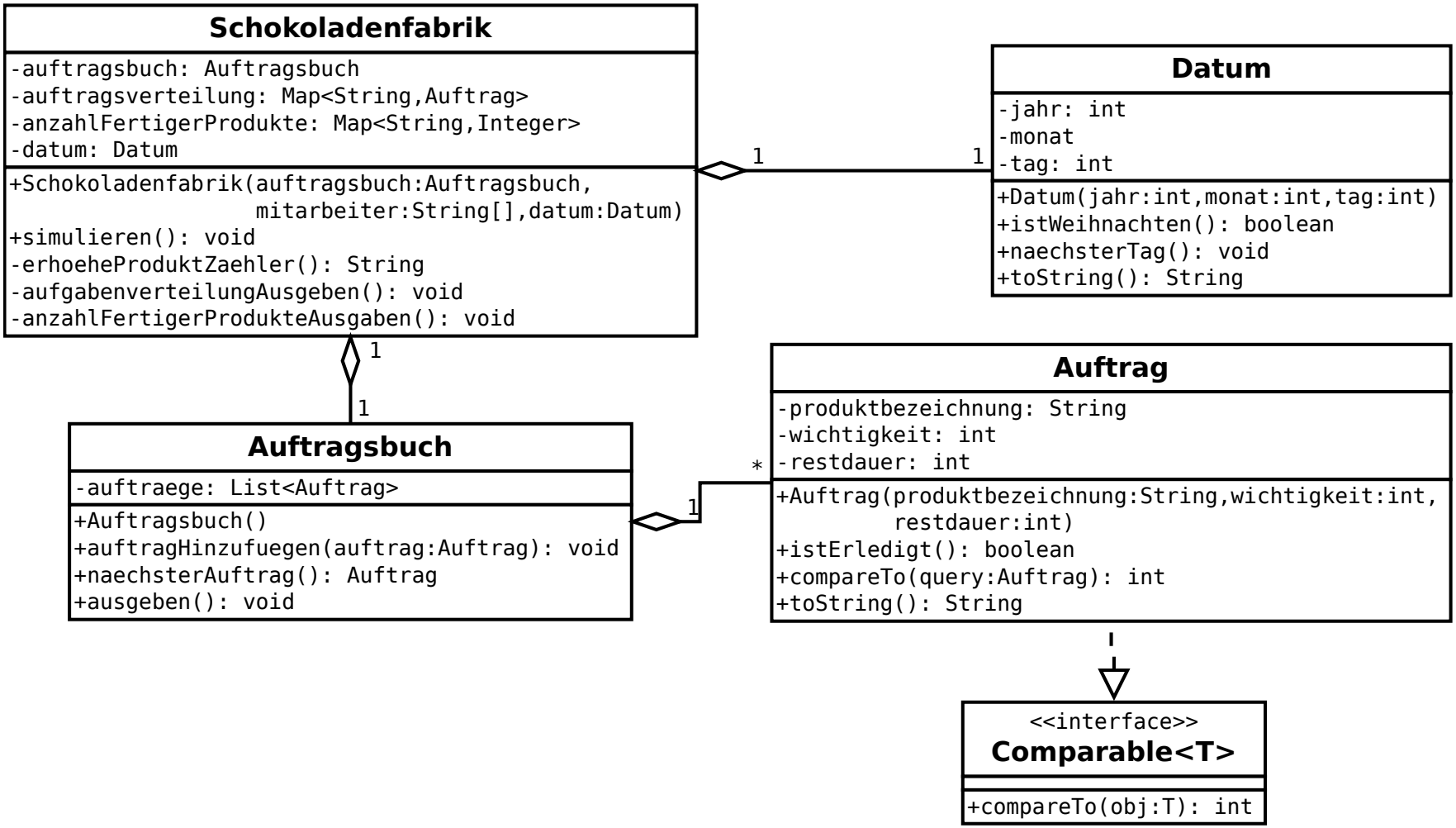
- Einstellung des Lehrbetriebs vom 17.12. bis zum 10.01.
- Der Übungsbetrieb läuft bis zum 16.12. normal weiter
- Der Abgabetermin der Serie 3 (16.12., 22:00 Uhr) bleibt bestehen!
- Die Abgabetermine für die Serien 4 und 5 werden angepasst

# Java-Programmierung - List, Map, Comparable

Programmierübung:

Simulation der Produktion von Süßigkeiten in einer Schokoladenfabrik in den Tagen vor Weihnachten

# Java-Programmierung - List, Map, Comparable



# Die Klasse `Datum`

- `istWeihnachten` gibt `true` zurück, wenn es der 24.12. eines beliebigen Jahres ist
- `naechsterTag` ändert das Datum auf den nächsten Tag und berücksichtigt dabei Monats- bzw. Jahreswechsel (aber keine Schaltjahre)
- `toString` gibt das Datum im Format Tag.Monat.Jahr zurück

# Die Klasse `Auftrag`

- Die Restdauer muss bei der Erzeugung eines neuen Auftrags mindestens 1 betragen
- Die Wichtigkeit muss einen Wert zwischen 0 und 3 haben
- `istErledigt` gibt `true` zurück, wenn die Restdauer 0 ist
- `compareTo` soll so implementiert werden, dass Aufträge nach absteigender Wichtigkeit sortiert werden. Bei gleicher Wichtigkeit soll nach absteigender Restdauer sortiert werden.
- `toString` gibt die Produktbezeichnung, Wichtigkeit und Restdauer als String zurück

# Die Klasse Auftragsbuch

- `auftragHinzufuegen` fügt den übergebenen Auftrag zur Liste aller Aufträge hinzu und sortiert die Liste anschließend
- `naechsterAuftrag` entfernt den ersten Auftrag aus der Auftragsliste und gibt ihn zurück. Ist die Auftragsliste leer, dann soll `null` zurückgegeben werden.
- `ausgeben` gibt alle Aufträge aus

# Die Klasse Schokoladenfabrik

- `simulieren` simuliert tagesweise den Zeitraum zwischen `datum` und Weihnachten
  - Jedem Mitarbeiter wird, solange das Auftragsbuch noch Aufträge enthält, ein Auftrag zugewiesen
  - Bei jedem Auftrag, der einem Mitarbeiter zugewiesen wurde, verringert sich die Restdauer jeden Tag um einen Tag
  - Ist ein Auftrag erledigt, wird dem Mitarbeiter ein neuer Auftrag zugewiesen und der Zähler für das entsprechende Produkt um 1 erhöht
  - Es werden täglich die Auftragsverteilung und Zahl der insgesamt fertiggestellten Produkte ausgegeben

# Die Klasse Schokoladenfabrik

- `auftragsverteilung` bildet jeden Mitarbeiter (repräsentiert durch seinen Namen) auf einen Auftrag ab
- `anzahlFertigerProdukte` gibt an, wie oft ein Produkt (repräsentiert durch seine Bezeichnung) bereits fertiggestellt wurde
- `erhoeheProduktZaehler` erhöht den Zähler für ein fertiggestelltes Produkt um 1
- `aufgabenverteilungAusgeben` gibt die aktuelle Aufgabenverteilung aus
- `anzahlFertigerProdukteAusgaben` gibt den Produktzähler aus

# char

- `char` ist ein primitiver Datentyp mit einer Größe von 2 Bytes
- Ein `char` kann einen Wert zwischen 0 und 65535 ($2^{16} - 1$, da 2 Bytes = 16 Bits) annehmen und verhält sich wie eine vorzeichenlose Zahl
  - $\rightarrow$ `chars` können wie Zahlen addiert, subtrahiert etc. werden
  - Das Ergebnis der Addition zweier `chars` ist ein `int` (**!**)
- Die Zuordnung eines `chars` zu einem bestimmten Buchstaben/Zeichen/Symbol hängt vom verwendeten Zeichensatz (z.B. UTF-8) ab

# char[]

- Ein Array von `chars`, `char[]`, verhält sich wie ein Array von Zahlen
- Zwei Arrays können mit der Funktion `Arrays.equals(char[], char[])` aus dem Paket `java.util` verglichen werden
  - $\rightarrow$ Als primitiver Datentyp hat ein `char[]` selbst keine `equals`-Funktion
- Arrays sind **veränderlich** (*mutable*) - einzelne `chars` im Array können beliebig durch andere ersetzt werden

# CharSequence

- CharSequence ist ein **Interface**
- Die Schnittstelle wird u.a. von `String` und `StringBuilder` implementiert
  - Einige Methoden in diesen Klassen akzeptieren als Parameter Objekte von allen Klassen, die dieses Interface implementiert haben

# CharSequence

- Die in `CharSequence` deklarierten Methoden sind
  - `charAt(int)` - gibt den `char` an der angegebenen Stelle zurück
  - `length()` - gibt die Länge der Sequenz zurück
  - `subSequence(int start, int ende)` - gibt eine neue `CharSequence` zurück, die alle Zeichen zwischen der (mit eingeschlossenen) Position `start` und der (nicht mit eingeschlossenen) Position `ende` der ursprünglichen `CharSequence` enthält
  - `toString()` - gibt die in der `CharSequence` gespeicherte Zeichenfolge als `String` zurück

# String

- `String` hat in Java eine Sonderstellung: es ist sowohl eine Klasse als auch ein eingebauter Datentyp
  - `String`-Objekte müssen (im Gegensatz zu allen anderen Objekten) nicht mit `new` erzeugt werden
  - `Strings` können mit dem +-Operator aneinandergefügt (konkateniert) werden, dieser Operator ist sonst primitiven Datentypen vorbehalten
- Die in `String`-Objekten gespeicherte Zeichenkette ist **unveränderlich** (immutable)
  - Funktionen, die Zeichen in `Strings` verändern, erzeugen immer neue `String`-Objekte

Einige wichtige Funktionen der Klasse String:

- `char charAt(int index)` und `int length()` aus dem `CharSequence`-Interface

```
String s = "MuP ist toll!";
char c = s.charAt(2);   // c == 'P'
int l = s.length();     // l == 13
```

- `String substring(int start, int ende)` - wie `subSequence`, nur gibt `substring` einen String zurück. Wichtig: Groß-/Kleinschreibung bei `subSequence`/`substring` beachten!

```
String s = "MuP ist toll!";
String subs = s.substring(5,10);      // subs ist "st to"
```

Einige wichtige Funktionen der Klasse String:

- `boolean equals(Object obj)` - testet zwei Strings auf Gleichheit unter Berücksichtigung von Groß-/Kleinschreibung

```
String s = "MuP ist toll!";
String t = "mup ist toll!";
String u = "MuP ist toll!";
boolean s_gleich_t = s.equals(t);       // false
boolean s_gleich_u = s.equals(u);       // true
```

  - `equals` übernimmt zwar beliebige Objekte, kann aber nur `true` zurückgeben, wenn `obj` auch ein `String` ist
  - Wichtig: der ==-Operator testet zwei String-Objekte auf Identität, nicht auf die Gleichheit der gespeicherten Zeichenketten. `s == t` gilt für zwei Strings also nur dann, wenn `s` und `t` ein und das selbe Objekt sind.

Einige wichtige Funktionen der Klasse String:

- `boolean equalsIgnoreCase(String str)` - testet zwei Strings auf Gleichheit und ignoriert dabei Groß-/Kleinschreibung

```
String s = "MuP ist toll!";
String t = "mup ist toll!";
String u = "MuP ist toll!";
boolean s_gleich_t = s.equalsIgnoreCase(t);     // true
boolean s_gleich_u = s.equalsIgnoreCase(u);     // true
```

Einige wichtige Funktionen der Klasse String:

- `int compareTo(String str)` - vergleicht zwei Strings lexikographisch unter Berücksichtigung der Groß-/Kleinschreibung
  - `s.compareTo(t)` gibt eine Zahl <=-1 zurück, wenn s lexikographisch kleiner als t ist
  - `s.compareTo(t)` gibt 0 zurück, wenn s gleich t ist
  - `s.compareTo(t)` gibt eine Zahl >= 1 zurück, wenn s lexikographisch größer als t ist

```
String s = "MuP ist toll!";
String t = "mup ist toll!";
String u = "MuP ist toll!";
String v = "A&D ist toll!";
int s_compared_t = s.compareTo(t);      // -32
int s_compared_u = s.compareTo(u);      // 0, da Strings gleich
int s_compared_v = s.compareTo(v);      // 12
```

Einige wichtige Funktionen der Klasse String:

- `int compareTo(String str)` - durch die bereits vorhandene Implementierung von `compareTo` können alle `Collections` (List, ArrayList, HashSet, ...) von `String` mit `Collection.sort()` sortiert werden. Die von `s.compareTo(t)` zurückgegebene Zahl ergibt sich wie folgt:
  - Unterscheiden sich `s` und `t` an der Stelle k, wird `s.charAt(k) - t.charAt(k)` zurückgegebenen
  - Unterscheiden sich `s` und `t` an keiner Stelle, wird `s.length() - t.length()` zurückgegeben
    - Hierdurch hat der kürzere String bei der Sortierung immer Vorrang
    - Sind beide Strings gleich lang, wird 0 zurückgegeben

Einige wichtige Funktionen der Klasse String:

- `int indexOf(int ch)` - gibt die Position des ersten Auftretens des Zeichens `ch` zurück, oder -1, falls das Zeichen nicht auftritt
- `int indexOf(int ch, int start)` - wie `indexOf(int ch)`, beginnt mit der Suche an der Position `start`
- `int lastIndexOf(int ch)`, `int lastIndexOf(int ch, int start)` - wie indexOf, sucht von rechts nach links

```
String s = "MuP ist toll!";
int t = s.indexOf('i');          // 4
int u = s.indexOf('t', 7);       // 8
int v = s.indexOf('x');          // -1
int w = s.lastIndexOf('l');      // 11
System.out.println(t);
System.out.println(u);
System.out.println(v);
System.out.println(w);
```

Einige wichtige Funktionen der Klasse String:

- int indexOf(String str) - wie oben, übernimmt `String` statt `char`
- boolean contains(CharSequence s) - gibt `true` zurück, wenn die gesuchte CharSequence `s` enthalten ist

```
String s = "MuP ist toll!";
String t = "P ist t";
String u = "mup";
String v = "xyz";
boolean s_contains_t = s.contains(t);   // true
boolean s_contains_u = s.contains(u);   // false
boolean s_contains_v = s.contains(v);   // false
System.out.println(s_contains_t);
System.out.println(s_contains_u);
System.out.println(s_contains_v);
```

Einige wichtige Funktionen der Klasse String:

- String toLowerCase() - gibt den String in Kleinbuchstaben zurück
- String toUpperCase() - gibt den String in Großbuchstaben zurück

```
String s = "MuP ist toll!";
String t = s.toLowerCase();     // mup ist toll!
String u = s.toUpperCase();     // MUP IST TOLL!
System.out.println(t);
System.out.println(u);
```

Einige wichtige Funktionen der Klasse String:

- String replace(char orig, char repl) - gibt einen neuen String zurück, in dem jedes Auftreten von `orig` durch `repl` ersetzt wurde. Der ursprüngliche String wird von links nach rechts prozessiert (wichtig, wenn mehrere ersetzbare Zeichen direkt aufeinander folgen).
- String replace(CharSequence orig, Charsequence repl) - methodisch identisch zur obigen Funktion, nur anderer Datentyp des Parameters

```
String s = "MuP ist toooll!";
String t = s.replace('o', 'z');   // MuP ist tzzzll!
String u = s.replace("oo", "z");  // MuP ist tzoll!
System.out.println(t);
System.out.println(u);
```

Einige wichtige Funktionen der Klasse String:

- static String valueOf(*) - erzeugt die Stringrepräsentation für beliebige Objekte und alle primitiven Datentypen
  - Bei Objekten wird die Methode `toString()` aufgerufen
  - Da die Methode statisch ist, kann sie direkt als Funktion der Klasse String aufgerufen werden: `String.valueOf()`

```
String s = String.valueOf(3.1);
char tmp[] = {'H', 'a', 'l', 'l', 'o'};
String t = String.valueOf(tmp);
String u = String.valueOf(new Testobjekt());
String v = String.valueOf(true);
```

Einige wichtige Funktionen der Klasse String:

- static String format(String format, Object obj1, Object obj2, ...)
  - Erzeugt einen formatierten String, bei dem bestimmte Platzhalter durch konkrete Werte ersetzt werden
    - `String.format("Der Wert von x beträgt %f", x)` $\rightarrow$ der Platzhalter `%f` wird durch den Wert von `x` ersetzt und der sich dadurch ergebende String zurückgegeben
  - Die Methode ist statisch, d.h. es muss kein `String`-Objekt erzeugt werden; stattdessen kann direkt `String.format()` aufgerufen werden

Einige wichtige Funktionen der Klasse String:

- static String format(String format, Object obj1, Object obj2, ...)
  - Einige der möglichen Platzhalter sind:
    - `%d` - Ersetzung durch Ganzzahl wie `byte`, `int`, `long`
    - `%7d` - Ersetzung durch Ganzzahl. Der Platzhalter wird durch mindestens sieben Zeichen ersetzt, bei Zahlen mit weniger als 7 Ziffern wird links mit Leerzeichen aufgefüllt.
    - `%f` - Ersetzung durch Fließkommazahl wie `float`, `double`
    - `%7.2f` - Ersetzung durch Fließkommazahl. Mindestens 7 Zeichen, davon genau zwei Zeichen für Nachkommastellen, ein Zeichen für das Komma und mindestens 4 (7-2-1) Zeichen für Vorkommastellen.
    - `%.2f` - Fließkommazahl mit genau zwei Nachkommastellen und beliebig vielen Vorkommastellen

Einige wichtige Funktionen der Klasse String:

- static String format(String format, Object obj1, Object obj2, ...)
  - Einige der möglichen Platzhalter sind:
    - `%s` - Ersetzung durch String (ein Nullzeiger wird durch "null" ersetzt, bei Objekten wird die `toString()`-Funktion aufgerufen)
    - `%7s` - Ein String mit einer Länge von mindestens 7 Zeichen. Fehlende Zeichen werden von links mit Leerzeichen aufgefüllt.
    - %c - Ein einzelner Buchstabe. Eine positive Ganzzahl wird dabei (entsprechend der *Locale*) in den Buchstaben umgewandelt, den sie repräsentiert.

Einige wichtige Funktionen der Klasse String:

- static String format(String format, Object obj1, Object obj2, ...)
  - Nebenbei:
    - Weitere Formatierungsoptionen ermöglichen bspw. Linksbündigkeit, führende Nullen oder erzwungene Vorzeichen
    - Zur Datums- und Uhrzeitformatierung gibt es eigene Platzhalter
    - Weitere Informationen liefert die Dokumentation der Klasse `Formatter`

Einige wichtige Funktionen der Klasse String:

- boolean matches(String regex) - gibt an, ob der String dem regulären Ausdruck `regex`
- String replaceAll(String regex, String replacement) - ersetzt alle Treffer des regulären Ausdrucks `regex` durch den String replacement
- String[] split(String regex) - teilt den String bei jedem Auftreten von `regex` und gibt ein Array aller so entstandenen Teilstrings zurück

- Mehr Informationen zu regulären Ausdrücken liefert die Dokumentation zur Klasse Pattern

# StringBuilder

- Ein `StringBuilder`-Objekt repräsentiert wie ein String eine Folge von Zeichen
- Während Strings unveränderlich sind, sind StringBuilder **veränderliche** Zeichenketten
- Die Klasse StringBuilder stellt einige Funktionen bereit, um die Zeichenkette zu manipulieren
  - Bspw. können Zeichen hinzugefügt, gelöscht oder geändert werden
  - Da die im Objekt gespeicherte Zeichenkette manipuliert wird, müssen nicht ständig neue StringBuilder-Objekte oder andere Zwischenvariablen erzeugt werden

Einige wichtige Funktionen der Klasse StringBuilder:

- `char charAt(int index)`, `int length()` und `String substring(int start, int ende)`, da `StringBuilder` das Interface `CharSequence` implementiert
- `int indexOf(String str)`, `int indexOf(String str, int start)`, `int lastIndexOf(String str)` - wie `indexOf` bzw. `lastIndexOf` der Klasse `String`

Einige wichtige Funktionen der Klasse StringBuilder:

- StringBuilder append(*)
  - Die `append()`-Funktion ist für alle primitiven Datentypen sowie für Objekte der Klasse `Object` überladen, sie akzeptiert somit beliebige Argumente
  - Wird `append(Object obj)` aufgerufen, so wird automatisch `obj.toString()` aufgerufen und der zurückgegebene String angehangen

```
Testobjekt obj = new Testobjekt();
StringBuilder s = new StringBuilder();
s.append("Hallo!");
s.append(123);
s.append(obj);
System.out.println(s);  //  Hallo!123Testobjekt@4aa298b7
```

Einige wichtige Funktionen der Klasse StringBuilder:

- StringBuilder insert(int offset, *)
  - Die `insert()`-Funktion akzeptiert wie `append` alle Datentypen als Argumente
  - Während `append` einen neuen String immer am Ende des bisherigen Strings **anfügt**, fügt `insert` den neuen String an der gegebenen Position `offset` **ein**

```
Testobjekt obj = new Testobjekt();
StringBuilder s = new StringBuilder();
s.insert(0, "Hallo!");
s.insert(0, 123);
s.insert(3, obj);
System.out.println(s);  //   123Testobjekt@4aa298b7Hallo!
```

Einige wichtige Funktionen der Klasse StringBuilder:

- StringBuilder delete(int start, int ende) - löscht den String, der an der Position `start` beginnt und an der Position `ende - 1` endet, aus dem Gesamtstring heraus
  - Gilt `start == ende` wird nichts gelöscht
- StringBuilder deleteCharAt(int index) - löscht einen einzelnen Buchstaben an der angegebenen Position
- Einige Eigenschaften des Strings (Länge u.ä.) werden automatisch angepasst

```
StringBuilder s = new StringBuilder("MuP ist toll!");
s.delete(4, 8);
System.out.println(s);      // MuP toll!
s.deleteCharAt(8);
System.out.println(s);      // MuP toll
```

Einige wichtige Funktionen der Klasse StringBuilder:

- `StringBuilder replace(int start, int ende, String str)`
  - Ersetzt den String, der an der Position `start` beginnt und an der Position `ende - 1` endet, durch einen neuen String `str`
  - Entspricht der Kombination von `delete(start, ende)` und `insert(start, str)`
  - Ist `str` der leere String "", entspricht `replace` der Funktion `delete`

```
StringBuilder s = new StringBuilder("MuP ist toll!");
s.replace(8, 12, "super");
System.out.println(s);      // MuP ist super!
s.replace(4, 8, "");
System.out.println(s);      // MuP super!!
```

Einige wichtige Funktionen der Klasse StringBuilder:

- reverse()
  - Kehrt den kompletten String um
- setCharAt(int index, char ch)
  - Ersetzt einen einzelnen Buchstaben an der gegebenen Stelle `index` durch den Buchstaben `ch`

```
StringBuilder s = new StringBuilder("MuP ist toll!");
s.reverse();
System.out.println(s);     //  !llot tsi PuM
s.setCharAt(5, '.');
s.setCharAt(9, '_');
System.out.println(s);     //  !llot.tsi_PuM
```

# Modellierung und Programmierung 1

Übung 8

Stefan Preußner

11./ 12. Januar 2021

# Ausnahmen

# Exceptions und Errors

- In vielen alten Programmiersprachen ist es üblich, Funktionen -1 oder 0 zurückgeben zu lassen, wenn ein Fehler aufgetreten ist
- Probleme
  - Keine Rückgabe eines Fehlercodes möglich, wenn z.B. jede mögliche Ganzzahl ein regulärer Rückgabewert einer Funktion ist
  - Mehrere mögliche Fehler erfordern ggf. verschiedene Fehlercodes
  - Analyse der Fehlerursache schwierig

# Exceptions und Errors

- In Java können beim Auftreten von Fehlern **Exceptions** und **Errors** ausgelöst werden
- `Exception` und `Error` sind Unterklassen der Klasse **Throwable**
- `Exceptions` sind vor allem für Fehler gedacht, die innerhalb des Programms behandelt werden können
- `Errors` sind vor allem für schwerwiegende Fehler gedacht, die nicht behandelt werden sollten und in der Regel einen Programmabbruch zur Folge haben

# try und catch

- Mit den Schlüsselwörtern **try** und **catch** können Fehler abgefangen und behandelt werden
- Innerhalb des `try`-Blocks steht der Code, welcher eine Exception auslösen kann:

```
try
{
    // Code, welche eine oder mehrere
    // Exceptions auslösen kann
}
```

# try und catch

- Nach `catch` folgt in runden Klammern der Name des Fehlers (eine von `Throwable` abgeleitete Klasse), der behandelt werden soll und in anschließend in geschweiften Klammern der Code zur Fehlerbehandlung:

```
catch (ExceptionTyp fehlervariable)
{
    // Code, welcher die Fehlerbehandlung durchführt
    // Der Fehler ist in fehlervariable gespeichert
    // und kann analysiert werden
}
```

# try und catch

- Die `catch`-Anweisung gilt automatisch für alle Unterklassen der angegebenen Exception
  - Tritt eine `Exception` auf, die nicht explizit behandelt wird, so wird automatisch überprüft, ob ihre Oberklasse behandelt wird
  - Bspw. können mit **catch (Exception e)** alle in Java vordefinierten Exceptions abgefangen werden
- Auf eine `try`-Anweisung können beliebig viele `catch`-Anweisungen folgen
  - Der erste `catch`-Block mit einer passenden Fehlerklasse wird ausgeführt, `catch (Exception e)` sollte (wenn überhaupt) also immer als letztes im Code stehen

# try und catch

```
public int dividiereGanzzahlig(int x, int y)  {
    return x/y;    // wirft bei y == 0 eine ArithmeticException
}

for (int k = 2; k >= -2; k--)
{
    try {
        dividiereGanzzahlig(9, k);
    }
    catch (ArithmeticException e) {
        System.out.println("Fehler bei k = " + k + ": " + e.getMessage());
    }
}
```

# Abfangen mehrerer Exceptions

```
public class ZahlenArray {
    public int[] daten;
}

public static void leseDaten(ZahlenArray arr, int index) {
    try {
        System.out.println(arr.daten[index]);
    }
    catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Index ist ungueltig!");
    }
    catch (NullPointerException e) {
        System.out.println("Das ZahlenArray darf nicht null sein!");
    }
}
```

# Multi-catch

- Mehrere unterschiedliche Fehler können zusammen abgefangen und gleich behandelt werden:

```
public class ZahlenArray {
    public int[] daten;
}

public static void leseDaten(ZahlenArray arr, int index) {
    try {
        System.out.println(arr.daten[index]);
    }
    catch (ArrayIndexOutOfBoundsException | NullPointerException e) {
        System.out.println("Fehler!");
    }
}
```

# finally

- Nach dem letzten `catch`-Block kann optional ein **finally**-Block folgen
- Code im `finally`-Block wird immer ausgeführt, unabhängig davon, ob ein Fehler aufgetreten ist
- Beispiel für einen sinnvollen Einsatz:
  - Eine Datei wird geöffnet, ausgelesen und der Inhalt verarbeitet
  - Die Datei kann beschädigt sein, in diesem Fall kann der Inhalt nicht verarbeitet werden
  - In jedem Fall sollte die Datei nach dem Auslesen wieder geschlossen werden
  - Das Schließen der Datei erfolgt im `finally`-Block

# finally

```
public class ZahlenArray {  public int[] daten;  }

public static void leseDaten(ZahlenArray arr, int index) {
    try {
        System.out.println(arr.daten[index]);
    }
    catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Index ist ungueltig!");
    }
    catch (NullPointerException e) {
        System.out.println("Das ZahlenArray darf nicht null sein!");
    }
    finally {
        System.out.println("Die Argumente des Funktionsaufrufs
        waren arr=" + arr + " und index=" + index);
    }
}
```

# throws

- Eine Exception kann, statt sie zu behandeln, an die aufrufende Funktion weitergeleitet werden
- Dies geschieht durch das Schlüsselwort **throws** im Methodenkopf:

```
public void foo()
    throws ArrayIndexOutOfBoundsException, NullPointerException {
    // Code, der unter Umstaenden eine der oben angegebenen
    // Exceptions ausloest
}
```

- Die Fehlerbehandlung (ignorieren, `catch`, Weiterleitung) wird dann der aufrufenden Funktion überlassen

# throws - Beispiel

```
public class ZahlenArray {
    public int[] daten;
}

public static void leseDaten(ZahlenArray arr, int index)
    throws ArrayIndexOutOfBoundsException, NullPointerException {
    System.out.println(arr.daten[index]);
}
```

# Geprüfte und ungeprüfte Exceptions

- Java unterscheidet zwischen geprüften (*checked*) und ungeprüften (*unchecked*) Exceptions
- Ungeprüfte Exceptions müssen nicht mit `try` und `catch` abgefangen werden (daher der Name: der Compiler prüft bei der Compilierung nicht, ob die Exception behandelt wird)
  - Tritt eine solche Exception außerhalb eines `try`-Blocks auf, wird das Programm beendet
  - Vom Konzept her deuten auftretende ungeprüfte Exceptions auf Fehler im Programmcode hin und sollten anderweitig vermieden werden

# Geprüfte und ungeprüfte Exceptions

- Geprüfte Exceptions müssen immer behandelt werden
  - Löst eine Methode eine geprüfte Exception aus, so muss dies im Methodenkopf mit dem Schlüsselwort `throws` angegeben werden
  - Geprüfte Exceptions sind vor allem *extern* verursachte Fehler, die sich schwer oder nicht durch programminterne Überprüfungen vermeiden lassen (Ein-/Ausgabefehler, Fehler beim Datenbankzugriff, *Interrupts*, ...))

# Die Klasse RuntimeException

- **RuntimeException** ist eine Oberklasse, von der alle **ungeprüften** Exceptions abgeleitet werden
- Alle von RuntimeException abgeleiteten Klassen sind automatisch ungeprüfte Exceptions, alle anderen sind automatisch geprüfte Exceptions
- Beim Schreiben von eigenen Fehlerklassen muss abgewägt werden, ob von RuntimeException oder von Exception abgeleitet wird

# Auslösen von Exceptions

- Eine Exception wird mit dem Schlüsselwort **throw** geworfen
- Syntax:

```
throw new NullPointerException();
throw new ArrayIndexOutOfBoundsException("Der Index ist ungueltig!");
```

- Alle von Throwable abgeleiteten Klassen haben vier wichtige Konstruktoren:
  - Throwable() - Keine Fehlermeldung
  - Throwable(String s) - Mit Fehlermeldung
  - Throwable(Throwable ursache) - Vor allem gedacht zum Weiterleiten von Fehlern an aufrufende Funktionen
  - Throwable(String s, Throwable ursache)

# Eigene Fehlerklassen

- Eigene Fehlerklassen können von beliebigen Throwable-Klassen abgeleitet werden (wobei am häufigsten von RuntimeException oder Exception geerbt wird):

```
public class ZahlZuGrossException extends Exception {

    public ZahlZuGrossException(String fehlermeldung) {
        super(fehlermeldung);
    }

}
```

- Wichtig: Aufruf von `super` mit den geeigneten Parametern

# Stacktrace

- Auf dem Stack werden (u.a.) die Rücksprungadressen der Funktionen abgelegt, die eine neue Funktion aufrufen
- Der **Stacktrace** ist eine lesbare Ausgabe des Stacks
- Durch den Stacktrace kann bei einem Fehler festgestellt werden, welche Kette von Funktionsaufrufen zum Fehler geführt hat

# Stacktrace - Beispiel

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at ZahlenArray.qux(ZahlenArray.java:16)
at ZahlenArray.bar(ZahlenArray.java:21)
at ZahlenArray.foo(ZahlenArray.java:26)
at Main.main(Main.java:25)
```

- Funktion `Main.main()` ruft in Zeile 25 der `Main.java` die ...
- Funktion `Zahlenarray.foo()` auf, welche in Zeile 26 der `Zahlenarray.java` die ...
- Funktion `Zahlenarray.bar()` aufruft, welche in Zeile 21 der `Zahlenarray.java` die ...
- Funktion `Zahlenarray.qux()` aufruft, welche in Zeile 16 der `Zahlenarray.java` eine `ArithmeticException` auslöst

# Stacktrace

- Zur Ermittlung des Stacktraces gibt es zwei Methoden:
- Throwable.printStackTrace() gibt den Stacktrace auf der Konsole aus
  - Die Methode ist überladen, um die Ausgabe z.B. in eine Datei umleiten zu können
- Throwable.getStackTrace() gibt den Stacktrace als `StackTraceElement[]` zurück

# Häufige Exceptions

- ArithmeticException
  - Ursache: mathematischer Fehler, z.B. Ganzzahldivision durch 0 oder Logarithmieren einer negativen Zahl
  - Vermeidung: Test auf $!= 0$, $>= 0$ usw.
- IndexOutOfBoundsException, ArrayIndexOutOfBoundsException, StringIndexOutOfBoundsException
  - Ursache: ungültiger Index beim Zugriff auf ein Array oder einen String
  - Vermeidung: Test, dass der Index nicht negativ und kleiner als die Länge der Datenstruktur ist

# Häufige Exceptions

- ClassCastException
  - Ursache: Objekt wird auf eine Subklasse gecastet, von der es keine Instanz ist
  - Vermeidung: Test mit instanceof und andere Tests auf Klassenzugehoerigkeit; ggf. Überladen von Funktionen, um Oberklassen als Parameter zu vermeiden
- NullPointerException
  - Ursache: Programm erwartet ein Objekt, erhält aber einen Null-Zeiger
  - Vermeidung: Test auf null

# Häufige Exceptions

- IllegalArgumentException, NumberFormatException, IllegalFormatException
  - Ursache: einer Funktion wurde ein ungültiges Argument übergeben (z.B. `Integer.parseInt("ABC")`)
  - Vermeidung: kaum möglich, wenn z.B. im Programm in irgendeiner Form Nutzereingaben möglich sind
  - Solche Exceptions sollten möglichst immer mit `try/catch` abgefangen werden

# Häufige Exceptions und Errors

- IOException
  - Ursache: Oberklasse aller Ein- und Ausgabefehler
  - Vermeidung: IOException sind geprüfte Exceptions, eine Vermeidung ist nicht vorgesehen, stattdessen sollen diese Exceptions behandelt werden
- StackOverflowError
  - Ursache: maximale Größe des Stacks wird überschritten, i.d.R. durch eine zu hohe Rekursionstiefe
  - Vermeidung: Änderungen am Programmablauf/Algorithmus
  - Ein StackOverflowError ist ein Error, keine Exception, und sollte daher nicht behandelt werden

# Ein- und Ausgabe

# Ein- und Ausgabe

- Arten von Ein- und Ausgaben:
  - Abfragen von Nutzereingaben in der Konsole
  - Einlesen von Dateien bzw. Schreiben in Dateien
  - Netzwerkdatenströme
  - Daten von Peripheriegeräten
  - usw.
- Wir beschäftigen uns ausschließlich mit Dateien

# `java.io` und `java.nio`

- Java stellt zwei Pakete für die Ausgabe in bzw. das Lesen von Dateien zur Verfügung: `java.io` und `java.nio`

- `java.io`:
  - Datenstromorientiert (Datenstrom = Stream)
  - Daten werden ungepuffert und sequentiell eingelesen, d.h. ein Zurückspringen im Datenstrom ist nicht möglich (außer durch eine eigene Implementierung eines Puffers)
  - Dateien werden beim Lesen bzw. Schreiben blockiert, d.h. es kann immer nur ein Thread auf eine Datei zugreifen

# `java.io` und `java.nio`

- `java.nio`:
  - Pufferorientiert
  - Daten werden in einen Puffer gelesen, in dem sich frei bewegt werden kann (ein Zurückspringen ist also möglich)
  - Dateien werden beim Lesen bzw. Schreiben nicht blockiert (`nio` = non-blocking I/O), d.h. es können gleichzeitig und parallel Daten gelesen und geschrieben werden

# Zeichen- und Byte-basierte Ein- und Ausgabe

- Daten können im Allgemeinen in zwei Kategorien aufgeteilt werden: Zeichenfolgen und Bytefolgen
- Zeichenfolgen werden immer dann gespeichert, wenn Daten menschlich lesbar sein sollen
  - reiner Text, HTML, XML, Base64, ...
- Bytefolgen werden dann verwendet, wenn die exakte Reihenfolge von Bits wichtig ist
  - praktisch alle Bild-, Musik- und Videoformate, Speicherung von Rohdaten, ...

# Zeichen- und Byte-basierte Ein- und Ausgabe

- In `java.io` existieren vier Basisklassen für die Ein- und Ausgabe:

| | **Für Bytefolgen** | **Für Zeichenfolgen** |
|---|---|---|
| **Eingabe** | `InputStream` | `Reader` |
| **Ausgabe** | `OutputStream` | `Writer` |

- Alle hiervon abgeleiteten Klassen tragen den Namen dieser Oberklassen jeweils am Ende ihres eigenen Namens
  - Ein `StringBufferInputStream` ist also - trotz des `String` im Namen - für die Eingabe von Bytefolgen gedacht

- Für die Ein- und Ausgabe aus/in Dateien gibt es die folgenden vier Basisklassen:

| | **Für Bytefolgen** | **Für Zeichenfolgen** |
|---|---|---|
| **Eingabe** | `FileInputStream` | `FileReader` |
| **Ausgabe** | `FileOutputStream` | `FileWriter` |

# FileReader

- Basisklasse zum Lesen von Zeichenfolgen aus Dateien
- Dem Konstruktor kann entweder ein `File`-Objekt

```
File f = new File("Hallo.txt");
FileReader r = new FileReader(f);
```

oder ein String

```
FileReader r = new FileReader("Hallo.txt");
```

oder ein `FileDescriptor`-Objekt übergeben werden

# FileReader

- Das Lesen von Daten erfolgt mit der Methode `read`
  - `read()` liest einen einzelnen Buchstaben und gibt ihn als `int` zurück. Gibt -1 zurück, wenn das Ende der Datei erreicht wurde.
  - `read(char[] puffer)` liest Buchstaben in einen `char`-Array ein. Gibt die Anzahl der eingelesenen Zeichen zurück oder -1, wenn das Ende der Datei erreicht wurde.
- $\rightarrow$ FileReader sind für das zeichenweise Einlesen ausgelegt und erfordern eine relativ aufwendige Verarbeitung der eingelesenen Zeichen

# BufferedReader

- Übernimmt im Konstruktor einen anderen `Reader` und benutzt dann diesen, um Daten gepuffert einzulesen
- Wichtigste Funktion: `readLine()` - liest Dateien zeilenweise ein

```
FileReader fr = new FileReader("Hallo.txt");
BufferedReader br = new BufferedReader(fr);
String zeile = br.readLine();
```

- `readLine()` gibt `null` zurück, wenn das Dateiende erreicht wurde

# FileInputStream

- Klasse zum Lesen von Bytefolgen aus Dateien
- Dem Konstruktor kann (äquivalent zu `FileReader`) ein `File`-Objekt oder ein Dateiname übergeben werden
- Die wichtigsten Funktionen:
  - `int read()` - gibt das nächste Byte in der Datei zurück. Gibt beim Erreichen des Dateiendes -1 zurück.
  - `int read(byte[] b)` - liest `b.length` viele Bytes aus der Datei und speichert sie in `b`. Gibt die Anzahl der gelesenen Bytes zurück.
  - `long skip(long n)` - überspringt `n` viele Bytes in der Datei

# FileWriter

- `FileWriter` ermöglicht das Schreiben von Zeichenfolgen in Dateien
- Dem Konstruktor kann (äquivalent zu `FileReader`) entweder ein `File`-Objekt, ein String oder ein `FileDescriptor`-Objekt übergeben werden
- Daneben existieren die Konstruktoren `FileWriter(File f, boolean append)` und `FileWriter(String s, boolean append)`, mit denen durch Setzen von `append` auf `true` eine Datei im Anhängemodus geöffnet werden kann

# FileWriter

- `FileWriter` stellt folgende (von `Writer` geerbte) Methoden zur Verfügung:
  - `write(char c)` schreibt ein einzelnes Zeichen
  - `write(char[] carr)` schreibt ein `char`-Array
  - `write(String s)` schreibt einen String
  - `append(char c)`
  - `append(CharSequence s)`

# FileWriter und BufferedWriter

- Da das Schreiben sehr vieler einzelner Zeichen sehr ineffizient ist, sollte ein `FileWriter` innerhalb einer anderen `Writer`-Klasse wie `BufferedWriter` verwendet werden:

```
try {
    FileWriter fw = new FileWriter("Hallo.txt", true);
    BufferedWriter bw = new BufferedWriter(fw);
    bw.write("MuP ist toll!");
    bw.close();
}
catch (IOException e) {
    System.out.println("Fehler!");
}
```

# close()

- Erinnerung: alle Klassen in `java.io` blockieren Dateien beim Öffnen
- Dateien sollten daher immer mit der Funktion `close()` geschlossen werden, sobald mit ihnen nicht mehr gearbeitet wird
- Dies gilt insbesondere beim Schreiben in Dateien[1]

[1] `close()` ruft automatisch die Methode `flush()` auf. `flush()` leert evtl. vorhandene Ausgabepuffer und überträgt alle noch zu schreibenden Daten (z.B. an das Betriebssystem).

# try mit Ressourcen

- Nach einem `try` können, in **runden Klammern**, **Ressourcen** wie Reader und Writer (also effektiv Datenströme) angegeben werden:

```
try (BufferedReader br = new BufferedReader(new FileReader("a.txt")))
{
  // Code, welcher den Reader br nutzen kann
}
```

- Am Ende des `try`-Anweisungsblocks oder bei Fehlern beim Lesen/Schreiben werden alle Ressourcen, die `java.lang.AutoCloseable` implementieren, automatisch geschlossen und freigegeben

# try mit Ressourcen

```
public static ArrayList<String> leseZeilenweise(String datei)
    throws IOException
{
    ArrayList<String> alleZeilen = new ArrayList<String>();
    try (BufferedReader br = new BufferedReader(new FileReader(datei)))
    {
        while (true)
        {
            // lese zeilenweise bis zum Ende der Datei
            // Abbruch der Endlosschleife mittels break, falls die
            // eingelesene Zeile gleich null ist
        }
    }
    return alleZeilen;
}
```

# java.nio.file

- `java.nio.file` ist ein Paket, welches Klassen und Schnittstellen für den Zugriff auf Dateien, Dateiattribute und Dateisysteme zur Verfügung stellt
- `java.nio.file.Files` ist eine Klasse, welche zahlreiche statische Funktionen für die Ein- und Ausgabe sowie für die Verwaltung von Dateien und Ordnern zur Verfügung stellt
- `java.nio.file.Paths` stellt statische Methoden zur Erzeugung von `Path`-Objekten (Datei- bzw. Ordnerpfaden) zur Verfügung

# java.nio.file

- `public static Path get(String pfad)` in der Klasse `Paths` erzeugt ein `Path`-Objekt aus dem angegebenen Dateipfad
- `public static List<String> readAllLines(Path p)` in der Klasse `Files` liest alle Zeilen einer zeichenbasierten Datei ein und gibt sie als Liste von Strings zurück
- `public static byte[] readAllBytes(Path pfad)` in der Klasse `Files` liest eine bytebasierte Datei vollständig aus und gibt den Dateiinhalt als Byte-Array zurück
- Achtung: die beiden Lesemethoden laden den kompletten Dateiinhalt in den Speicher, was bei großen Dateien problematisch sein kann

# java.nio

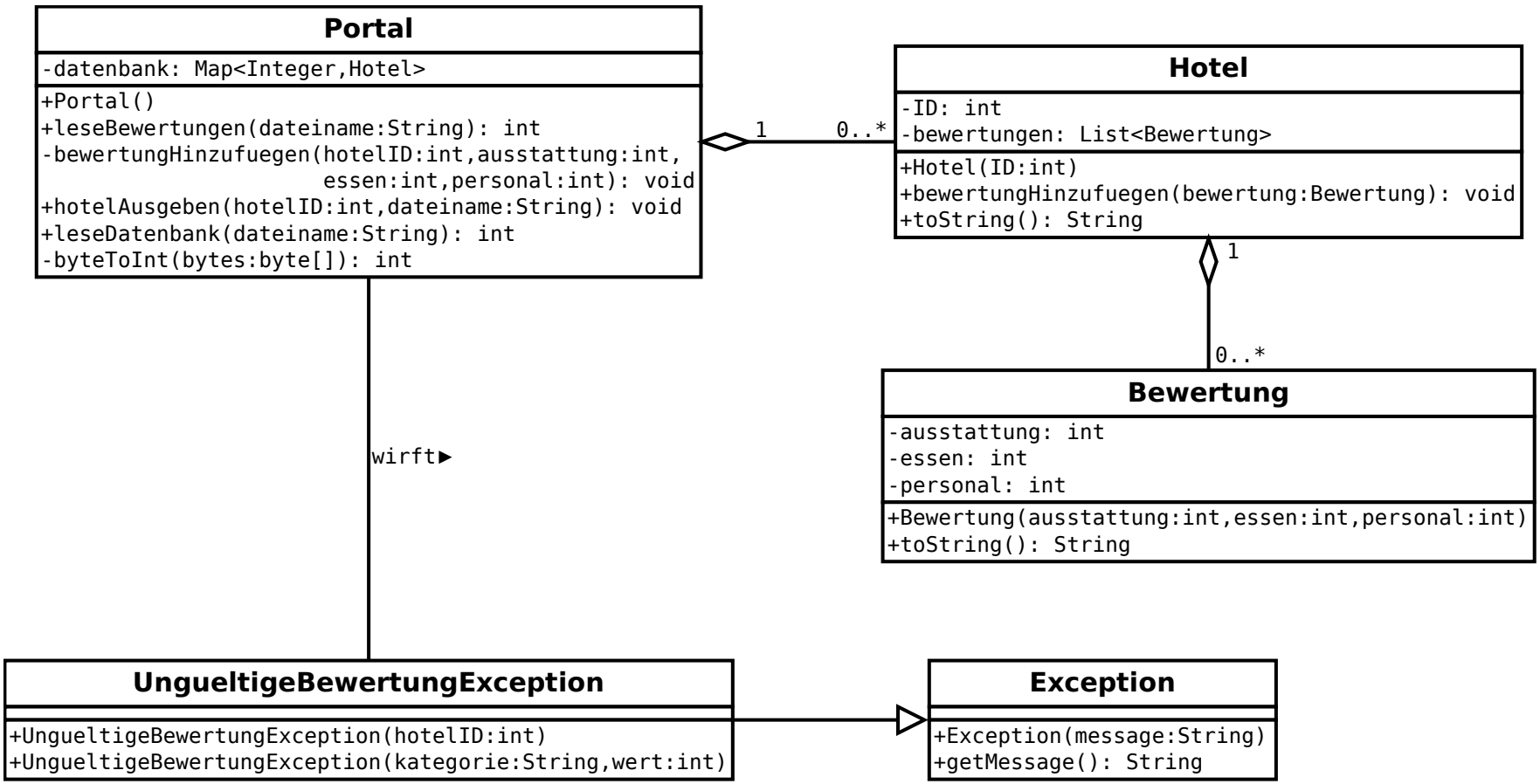
```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public static void main(String[] args)) {
    try {
        Path pfad = Paths.get("Hallo.txt");
        List<String> zeilen = Files.readAllLines(pfad);
        System.out.println(zeilen);
    }
    catch (IOException e) {
        System.out.println("Fehler!");
    }
}
```

# Programmierübung

Ein Bewertungsportal für Hotels möchte einen Teil seiner Datenbankverwaltung neu organisieren. Sie haben die Aufgabe bekommen, sich insbesondere um das Einlesen der Datenbank zu kümmern. Nach der Modellierungsphase planen Sie, das Programm auf der nachfolgenden Folie zu realisieren.

# Programmierübung - Hotelbewertungen



Implementieren Sie die Methode `bewertungHinzufuegen(int hotelID, int ausstattung,int essen, int personal)` der Klasse Portal, welche eine neue Bewertung erzeugt und zu dem entsprechenden Hotel hinzufügt.
Die Methode soll eine Ausnahme werfen, wenn entweder die ID des Hotels nicht in der Datenbank enthalten ist oder wenn die Punktzahl in einer der drei Kategorien außerhalb des gültigen Wertebereichs von 0 bis 10 liegt. Erstellen Sie hierzu eine Klasse `UngueltigeBewertungException`, welche von der Java-Klasse Exception erben soll. Bei einer ungültigen ID soll die Fehlermeldung (der von `getMessage()` zurückgegebene String) die Form

```
ID 128513 nicht gefunden
```

haben. Die Fehlernachricht bei einer ungültigen Teilbewertung in einer der drei Kategorien soll die Form

```
Ungültige Bewertung (-10) in der Kategorie Essen
```

haben.

Implementieren Sie die Methode `leseBewertungen(String dateiname)` der Klasse Portal, welche die Hotelbewertungen aus der Bewertungsdatei `dateiname` einliest. Tritt beim Öffnen oder Einlesen der Datei ein Fehler auf, dann soll eine aussagekräftige Meldung auf der Konsole ausgegeben werden.
Die Datei enthält je Zeile genau eine Bewertung im folgendem Format:

`HotelID;WertungAusstattung;WertungEssen;WertungPersonal`

Hierbei ist `HotelID` die ID des bewerteten Hotels; `WertungAusstattung`, `WertungEssen` und `WertungPersonal` sind die vergebenen Punkte in den Kategorien Ausstattung, Essen und Personal. Fügen Sie die Bewertungen unter Verwendung der Methode `bewertungHinzufuegen` zum jeweiligen Hotel hinzu.

Sollte eine Bewertung ungültig sein, dann soll eine Fehlermeldung in dem folgenden Format auf der Konsole ausgegeben werden:

`Fehler in Zeile <Zeilennummer> von <Dateiname>: <Fehlermeldung>`

Beispiel:

`Fehler in Zeile 131 von Bewertungen.txt: Ungültige Bewertung (11) in der Kategorie Ausstattung`

Das Einlesen der Bewertungsdatei soll bei einer ungültigen Bewertung nach Ausgabe der Fehlermeldung fortgesetzt werden.
Geben Sie die Gesamtzahl der eingelesenen Bewertungen (einschließlich der ungültigen) zurück.

Hinweis: Sie können annehmen, dass die Formatierung sämtlicher Zeilen der Bewertungsdatei korrekt ist.

Implementieren Sie die Methode `hotelAusgeben(int hotelID, String dateiname)` der Klasse Portal, welche für das Hotel mit der ID `hotelID` den von der `toString()`-Methode zurückgegebenen String in die Datei `dateiname` schreibt. Tritt beim Erstellen/Öffnen oder Schreiben ein Fehler auf, dann soll eine aussagekräftige Fehlermeldung auf der Konsole ausgegeben werden.

Hinweis: Sie können annehmen, dass die Methode nur für gültige IDs aufgerufen wird.

Führen Sie in der `main`-Methode folgende Aktionen durch:

- Erzeugen Sie ein neues Portal, lesen Sie die Datenbankdatei `Hotels.mupdb` ein und geben Sie die Anzahl der eingelesenen IDs aus.
- Lesen Sie die Bewertungsdatei `Bewertungen.txt` ein und geben Sie die Anzahl der eingelesenen Bewertungen aus.
- Schreiben Sie das Hotel mit der ID 135693 in die Datei `Hotel135693.txt`.

# Modellierung und Programmierung 1

Übung 9

Stefan Preußner

18./ 19. Januar 2021

# Interfaces

(Wiederholung)

# Schnittstellen (Interfaces)

- In Java gibt es keine Mehrfachvererbung, eine Klasse kann also nicht von mehreren Klassen erben
- Soll eine Klasse mehrere Typen haben, so kann sie **Schnittstellen** - **Interfaces** - implementieren
- Eine Schnittstelle legt fest, welche Methoden eine Klasse besitzen muss, stellt aber selbst i.d.R. keine Implementierung zur Verfügung
  - In Interfaces können aber, u.a. aus Gründen der Rückwärtskompatibilität, `default`-Methoden implementiert werden

# Interfaces erstellen

- Ein Interface kann mit der Syntax

```
<Sichtbarkeit> interface <Name>
```

  deklariert werden
- Beispiel:

```
public interface Connection
```

- Die Sichtbarkeit ist auf `public` und *package* (also das Fehlen eines Modifizierers) beschränkt
- Ein Interface kann mit dem Schlüsselwort `extends` andere Interfaces erweitern:

```
public interface Connection extends AutoCloseable, Wrapper
```

# Interfaces implementieren

- Ein Interface kann mit dem Schlüsselwort `implements` durch eine Klasse implementiert werden:

```
public class NetworkConnection implements Connection
```

- Eine Klasse kann beliebig viele Interfaces implementieren:

```
public class NetworkConnection implements
    Connection, Resettable
```

# Interfaces implementieren

- Eine Klasse **muss** alle Methoden eines Interfaces implementieren
  - Dies gilt nicht für Klassen, welche `abstract` sind (von abstrakten Klassen können keine Instanzen erzeugt werden, daher müssen sie keine Methoden implementieren)
  - Dies gilt nicht für Methoden, für welche es eine `default`-Implementierung im Interface gibt
- Implementierte Methoden **müssen** `public` sein

# Interfaces - Konstruktoren und Methoden

- Von einer Schnittstelle können keine Objekte erzeugt werden, nur von den sie implementierenden Klassen
- Eine Schnittstelle darf deshalb keinen Konstruktor haben
- Die Methoden eines Interfaces sind automatisch `public` und `abstract`
  - `abstract` Methoden werden nur deklariert, aber nicht implementiert
  - Die Deklaration einer Schnittstellenmethode enthält nur Modifizierer, Rückgabetyp und Signatur

# Variablen in Schnittstellen

- Instanzvariablen sind immer Teil einer Implementierung; da Schnittstellen keine Implementierung enthalten, besitzen sie auch keine Instanzvariablen
- In Schnittstellen können Konstanten festgelegt werden
  - Diese sind automatisch `public`, `static` und `final`
  - Erweitert eine Schnittstelle eine andere Schnittstelle, so kann sie deren Konstanten mit eigenen Werten überschreiben

# Schnittstellen und instanceof

`instanceof` funktioniert bei Schnittstellen wie bei Klassen. In dem Beispiel

```
public class NetworkConnection implements
        Connection, Resettable
```

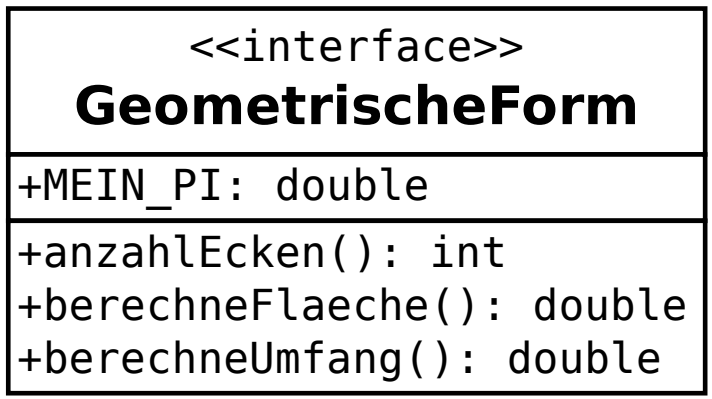
geben die Tests

```
NetworkConnection verbindung;
verbindung instanceof NetworkConnection;
verbindung instanceof Connection;
verbindung instanceof Resettable;
```

alle `true` zurück.

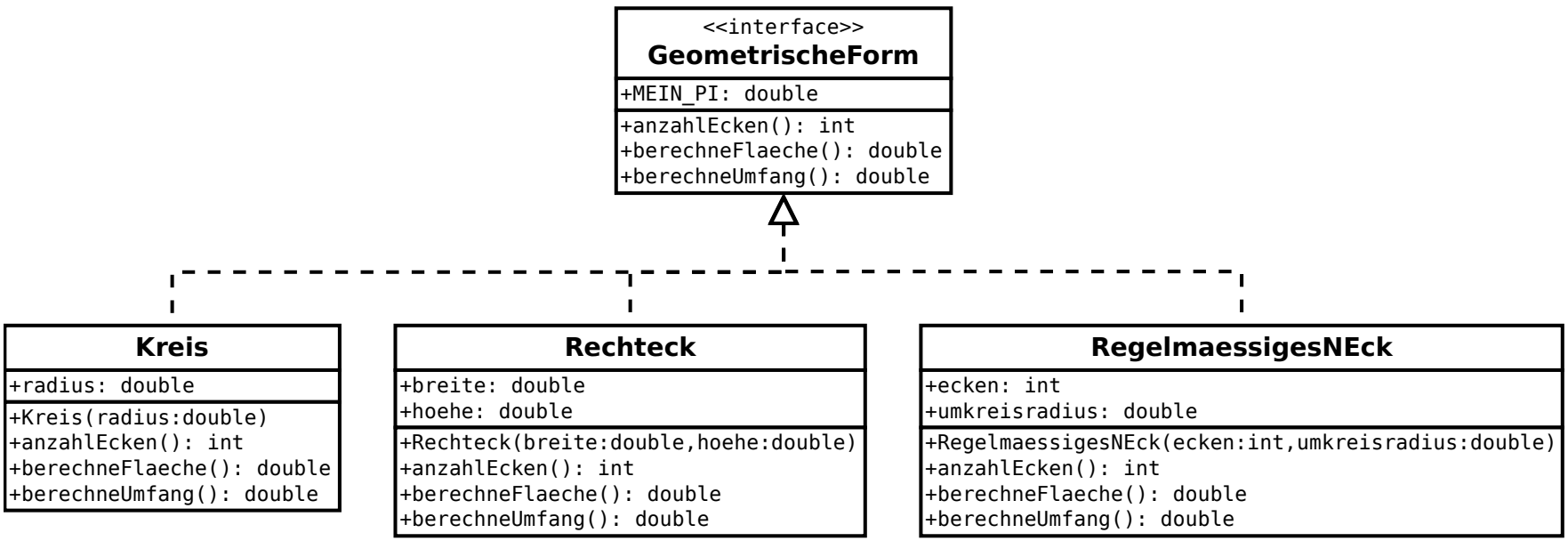
# Programmierübung

Erstellen Sie die Schnittstelle `GeometrischeForm` entsprechend des folgenden UML-Diagramms:



# Programmierübung

Erstellen Sie weiterhin die Klassen `Kreis`, `Rechteck` und `RegelmaessigesNEck` (für ein regelmäßiges N-Eck), welche alle `GeometrischeForm` implementieren sollen:



# Pseudocode

# Pseudocode

- Pseudocode dient dazu, einen Algorithmus oder ein Programm darzustellen und (für Menschen) verständlich zu machen, ohne dafür auf Bestandteile einer bestimmten Programmiersprache zurückgreifen zu müssen
- Pseudocode ist
  - weniger formal und i.d.R. verständlicher als Programmcode
  - formaler und kompakter als eine Beschreibung *im Fließtext*
- Für Pseudocode gibt es **keinen Standard**, d.h. grundsätzlich kann Pseudocode frei gestaltet werden

# Umwandlung von Java-Code in Pseudocode

| Java | mögliche Entsprechung im Pseudocode |
|---|---|
| `if` | FALLS |
| `else` | SONST |
| `while` | SOLANGE |
| `for` | FÜR |
| for-each-loop | FÜR ALLE |
| `return` | ENDE / GEBE ... ZURÜCK |
| `System.out.println` | GEBE ... AUS |
| Wertzuweisung | INITIALISIERE / SETZE |
| Anweisungsblock | Einrückung |

# Modellierung und Programmierung 1

Übung 10

Stefan Preußner

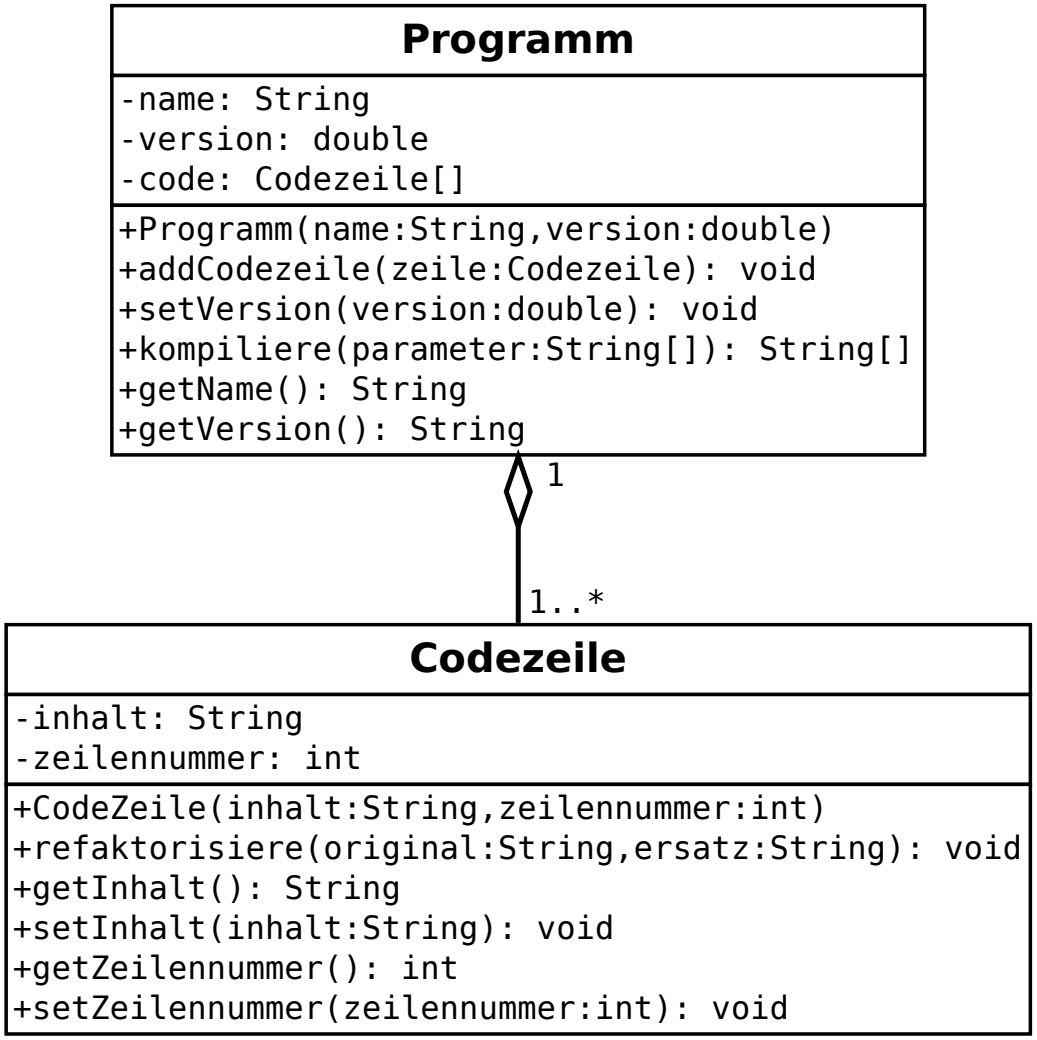
25./ 26. Januar 2021

| Thema | Vorlesung | Übung |
|---|---|---|
| UML | Vorlesung 2 | Übungen 2, 3 |
| Datentypen, Ausdrücke & Kontrollstrukturen | Vorlesungen 3, 4, 5 | — |
| Vererbung & Interfaces | (Vorlesung 2) | Übungen 4, 9 |
| Ausnahmen | Vorlesung 6 | Übung 8 |
| Collections | Vorlesung 7 | Übungen 6, 7 |
| Strings | Vorlesung 8 | Übung 7 |
| I/O | Vorlesung 9 | Übung 8 |
| Pseudocode | — | Übung 9 |
| Rekursion | Vorlesung 10 | (Übung 9) |
| Threads | Vorlesung 11 | — |
| Grafik | Vorlesung 13 | — |

# UML

Es soll ein `Programm` modelliert werden. Ein `Programm` besitzt einen Namen (z.B. "MuPTest") und eine Versionsnummer (z.B. 1.23). Die Versionsnummer kann, im Gegensatz zum Namen, nachträglich geändert werden. Ein Programm kann unter Angabe einer Liste von Parametern (Strings) kompiliert werden, dabei wird eine Liste von Fehlermeldungen zurückgegeben. Ein Programm besteht aus mindestens einer `CodeZeile`, wobei jede `CodeZeile` nur zu genau einem Programm gehört. Codezeilen können nachträglich zu einem Programm hinzugefügt werden. Eine `CodeZeile` besteht aus einer Zeilennummer und einem Inhalt (z.B. "x++;"). Zeilennummer und Inhalt können über entsprechende Getter und Setter abgefragt und geändert werden. Eine `CodeZeile` kann außerdem refaktorisiert werden, dabei wird der zu ersetzende Text und der Ersatztext angegeben.

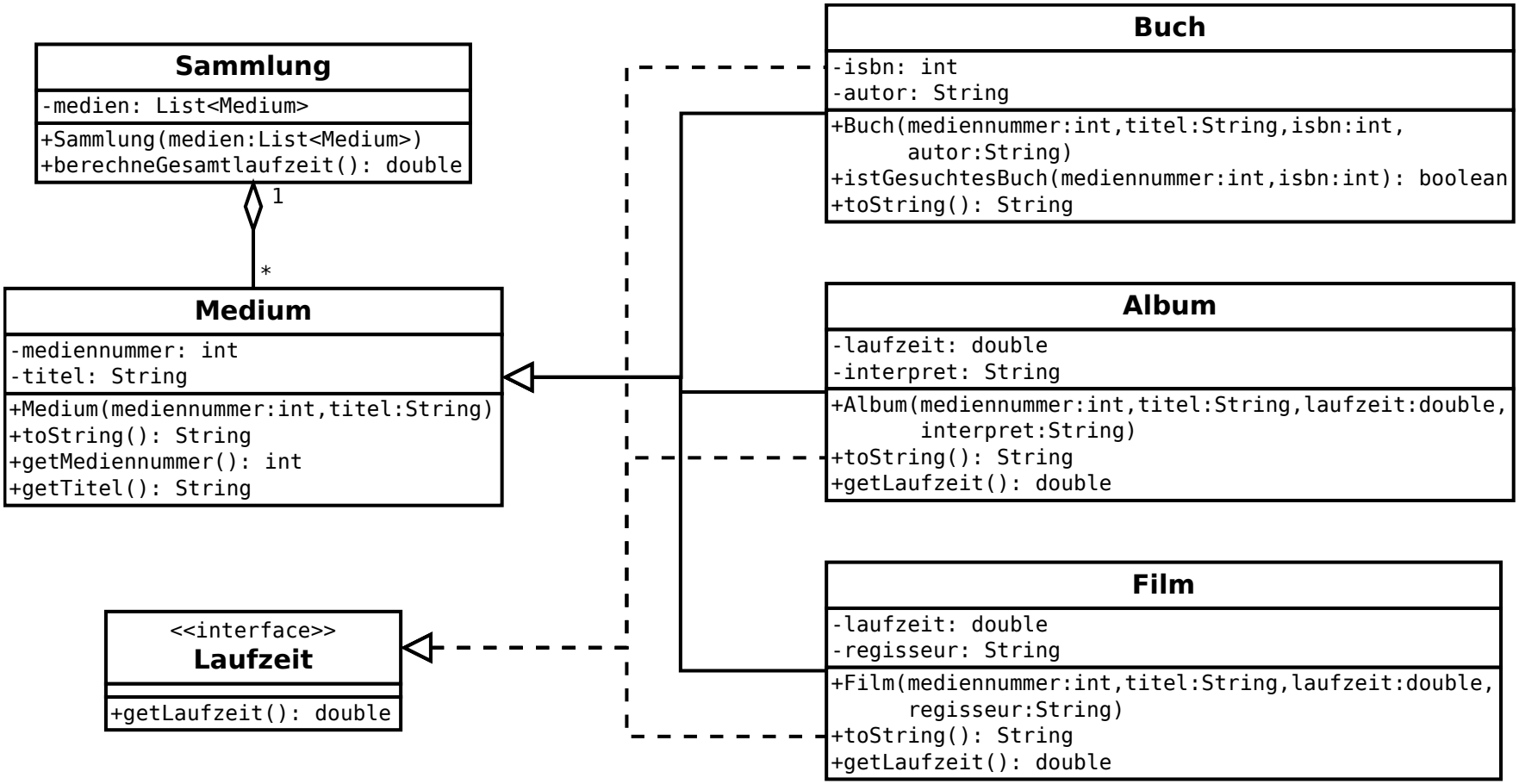
Es soll ein `Programm` modelliert werden. Ein `Programm` besitzt einen Namen (z.B. "MuPTest")und eine Versionsnummer (z.B. 1.23). Die Versionsnummer kann, im Gegensatz zum Namen, nachträglich geändert werden. Ein Programm kann unter Angabe einer Liste von Parametern (Strings) kompiliert werden, dabei wird eine Liste von Fehlermeldungen zurückgegeben. Ein Programm besteht aus mindestens einer `CodeZeile`, wobei jede `CodeZeile` nur zu genau einem Programm gehört. Codezeilen können nachträglich zu einem Programm hinzugefügt werden. Eine `CodeZeile` besteht aus einer Zeilennummer und einem Inhalt (z.B. "x++;"). Zeilennummer und Inhalt können über entsprechende Getter und Setter abgefragt und geändert werden. Eine `CodeZeile` kann außerdem refaktorisiert werden, dabei wird der zu ersetzende Text und der Ersatztext angegeben.




Es soll ein `Programm` modelliert werden. Ein `Programm` besitzt einen Namen (z.B. "MuPTest")und eine Versionsnummer (z.B. 1.23). Die Versionsnummer kann, im Gegensatz zum Namen, nachträglich geändert werden. Ein Programm kann unter Angabe einer Liste von Parametern (Strings) kompiliert werden, dabei wird eine Liste von Fehlermeldungen zurückgegeben. Ein Programm besteht aus mindestens einer `CodeZeile`, wobei jede `CodeZeile` nur zu genau einem Programm gehört. Codezeilen können nachträglich zu einem Programm hinzugefügt werden. Eine `CodeZeile` besteht aus einer Zeilennummer und einem Inhalt (z.B. "x++;"). Zeilennummer und Inhalt können über entsprechende Getter und Setter abgefragt und geändert werden. Eine `CodeZeile` kann außerdem refaktorisiert werden, dabei wird der zu ersetzende Text und der Ersatztext angegeben.

# Vererbung & Interfaces

# Implementieren Sie das nachfolgende UML-Diagramm:



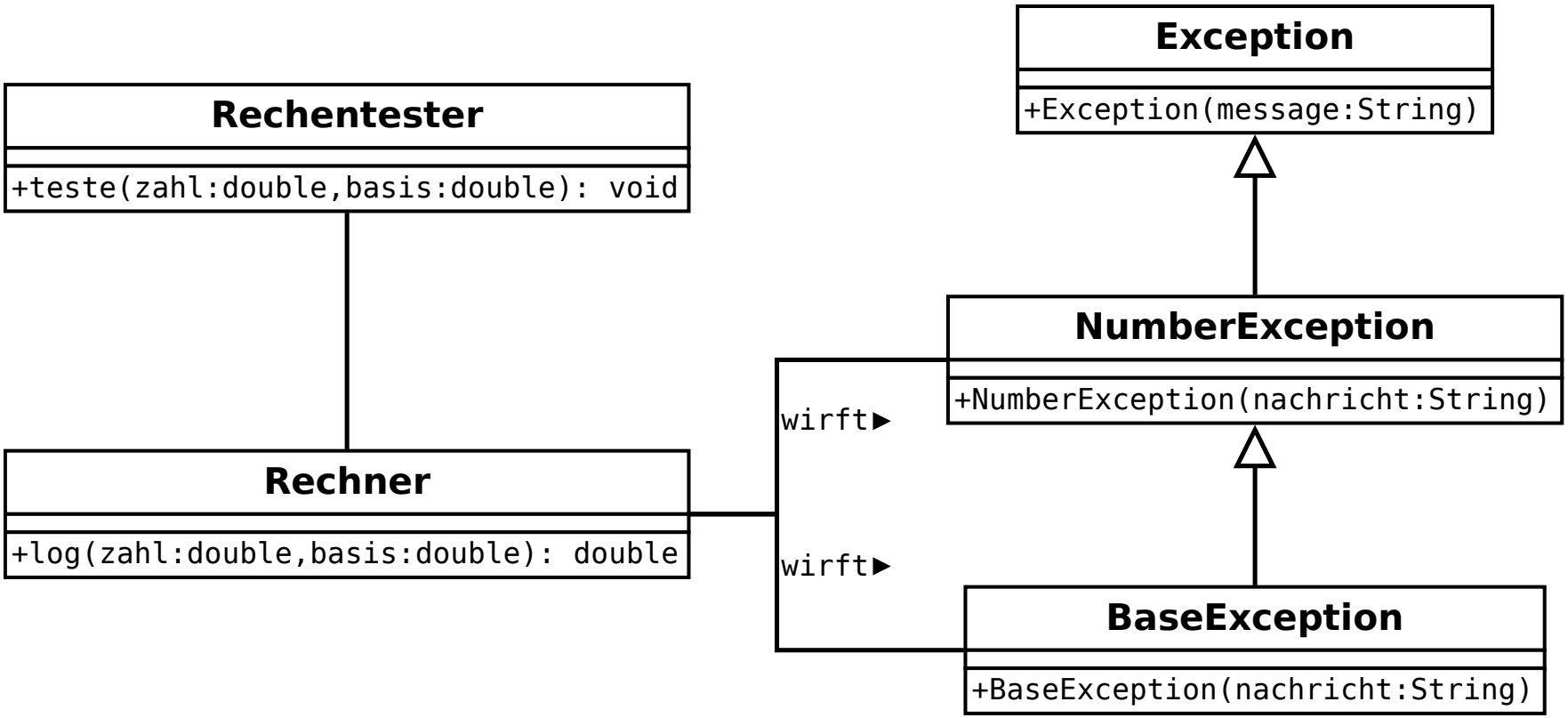
Beachten Sie bei der Umsetzung die folgenden Hinweise:

- Die Klasse `Medium` sowie das Interface `Laufzeit` sind bereits vorgegeben.
- Klasse `Album`:
  - Die Methode `toString()` soll folgenden String zurückgeben:
    `Mediennummer: <mediennummer>, Titel: <titel>, Laufzeit: <laufzeit>, Interpret: <interpret>`
- Klasse `Film`:
  - Die Methode `toString()` soll folgenden String zurückgeben:
    `Mediennummer: <mediennummer>, Titel: <titel>, Laufzeit: <laufzeit>, Regisseur: <regisseur>`

- Klasse `Buch`:
  - Die Methode `toString()` soll folgenden String zurückgeben:
    `Mediennummer: <mediennummer>, Titel: <titel>, ISBN: <isbn>, Autor: <autor>`
  - Die Methode `istGesuchtesBuch` soll eine Mediennummer und eine ISBN übernehmen und genau dann `true` zurückgeben, wenn wenigstens eine der beiden Nummern übereinstimmt.

# Ausnahmen

Implementieren Sie das nachfolgende UML-Diagramm:



Beachten Sie bei der Umsetzung die folgenden Hinweise:

- Die Methode `log(zahl, basis)` der Klasse `Rechner` soll:
  - eine `NumberException` mit der entsprechenden Fehlermeldung werfen, wenn `zahl` negativ oder 0 ist
  - eine `BaseException` mit der entsprechenden Fehlermeldung werfen, wenn `basis` negativ, 0 oder 1 ist.

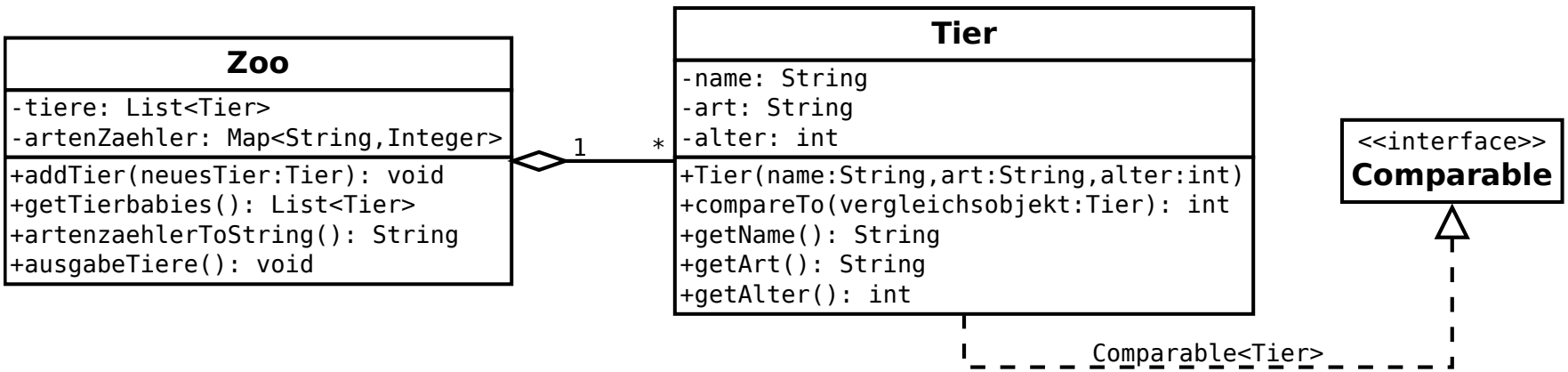
  Die Ausnahmen sollen nicht weiter behandelt werden.

Beachten Sie bei der Umsetzung die folgenden Hinweise:

- Die Methode `teste(zahl, basis)` der Klasse `Rechentester` soll das Ergebnis der Berechnung `Rechner.log(zahl, basis)` ausgeben. Alle auftretenden Ausnahmen sollen so behandelt werden, dass eine entsprechende Fehlermeldung auf der Fehlerkonsole `System.err` ausgegeben wird. Unabhöngig davon, ob eine Ausnahme behandelt wurde oder nicht, sollen zum Schluss die Parameter der Methode ausgegeben werden.

# Collections

Implementieren Sie das nachfolgende UML-Diagramm:



Beachten Sie bei der Umsetzung die folgenden Hinweise:

- Klasse `Tier`:
  - `name` ist der Name des Tieres. `art` gibt die Tierart und `alter` das Alter in Tagen an.
  - `compareTo` soll zuerst die Art lexikographisch vergleichen. Ist `art` bei beiden Tieren gleich, dann soll der Name lexikographisch verglichen werden.
  - `toString` soll die Informationen über ein Tier in folgendem Format zurückgeben:
    `<Art> <Name>, <Alter> Tage alt`

Beachten Sie bei der Umsetzung die folgenden Hinweise:

- Klasse `Zoo`:
  - `tiere` ist die Liste aller Tiere im Zoo. `artenZaehler` soll für jede Tierart zählen, wie häufig sie im Zoo vorkommt.
  - `addTier` soll das übergebene Tier zur Liste aller Tiere hinzufügen und den Artenzählen aktualisieren.
  - `getTierbabies` soll eine Liste aller Tiere, die jünger sind als 100 Tage, zurückgeben.
  - `artenzaehlerToString` soll einen String zurückgeben, der tabellarisch die Häufigkeit jeder Tierart angibt.
  - `ausgabeTiere` soll alle Tiere im Zoo ausgeben. Die Ausgabe soll dabei in der Reihenfolge erfolgen, wie sie durch die Sortierfunktion der Klasse Tier vorgegeben ist. tiere soll nicht verändert werden.

# Strings

(siehe Paket `strings`)

# Rekursion und Pseudocode

(siehe Paket `rekursion` sowie
die Übungen 5 und 6 in der Klasse `StringUebung`)