# Übung 11, Lösungsvorschlag

TECHNISCHE UNIVERSITÄT DARMSTADT

## Aufgabe 1 (Wissensfrage)

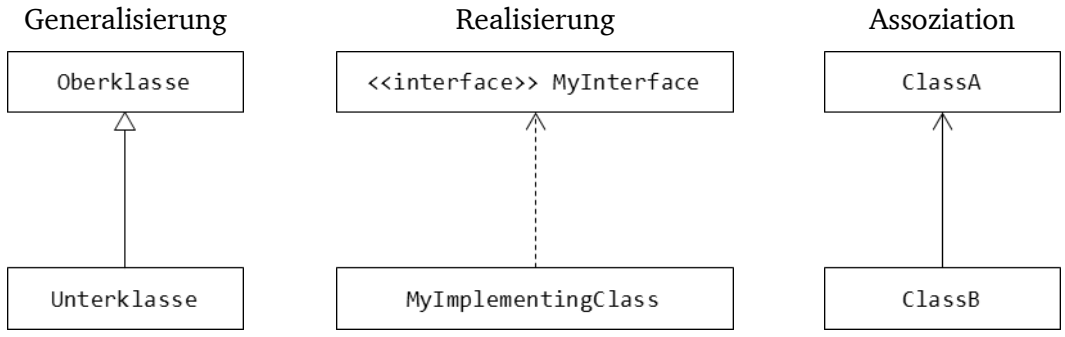
a) Welche drei Arten von Beziehungen können zwischen Klassen bestehen und wie werden diese in einem UML-Klassendiagramm dargestellt?
b) Was ist der Unterschied zwischen „checked“ und „unchecked“ Exceptions?

## Lösung Aufgabe 1

a) Die **Generalisierung** ist eine Beziehung zwischen einem allgemeinen und einem speziellen Element.
In Java entspricht dies der Vererbung (zwischen einer Ober- und einer Unterklasse) oder der Erweiterung (zwischen zwei Interfaces).
*Darstellung: Eine durchgezogene Verbindungslinie mit einem unausgefüllten Dreieck am Ende, das auf das allgemeine Element zeigt.*

Die **Realisierung** ist eine Beziehung zwischen einem Element, das Anforderungen stellt und einem anderen Element, das die Anforderungen erfüllt.
In Java entspricht dies der Implementierung eines Interfaces durch eine Klasse.
*Darstellung durch eine gestrichelte Linie zwischen der Klasse und dem Interface, mit einem unausgefüllten Dreieck am Ende, das auf das Interface zeigt.*

Die **Assoziation** ist eine Zugriffsbeziehung zwischen zwei Elementen.
In Java entspricht dies dem direkten Zugriff auf eine andere Klasse (z. B. Verwendung einer anderen Klasse als Datentyp).
*Darstellung als eine durchgezogene Linie mit einer offenen Pfeilspitze am Ende, die vom zugreifenden auf das zugegriffene Element zeigt.*



b) Sogenannte „**unchecked exceptions“** werden vom Kompiler nicht geprüft und entstehen somit erst während der Ausführung eines Programms und sind von der Klasse RuntimeException abgeleitet (bspw. ArrayIndexOutOfBoundsException). Die Ursachen für solche Fehler können jedoch meist vorhergesehen werden. Deshalb sollten diese durch sorgfältiges Programmieren vorgebeugt werden (Teilen durch Null, Ende der Zählervariable beim Durchlaufen eines Arrays). Sie können also, müssen aber nicht, explizit behandelt werden.
Vor sogenannte „**checked exceptions**“ werden wir dagegen bereits durch den Kompiler gewarnt. Wird eine Methode aufgerufen, deren Klasse eine „Checked Exception“ *werfen* kann (throws ….) so muss dieser Aufruf innerhalb eines try-catch-Blocks geschehen, der die entsprechende Exception-Subklasse *fängt*. Sie müssen also vor dem kompilieren explizit behandelt werden.

## Aufgabe 2 (Personalverwaltung)

Gegeben sei folgende Klassenhierarchie für eine Personalverwaltung an einer Universität.

a) Erstellen Sie für die angegebene Implementierung dieser Klassenhierarchie ein UML-Klassendiagramm. Achten Sie dabei darauf, dass Sie die Klassen, Attribute, Methoden und insbesondere die Beziehungen zwischen den Klassen korrekt darstellen.

```
public class Person {
  private String name;
  private String strasse;
  private String plz;
  private String ort;

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

  public void druckeAdressetikett() {
    System.out.println(name);
    System.out.println(strasse);
    System.out.println(plz + " " + ort);
  }

  public boolean equals(Object obj) {
    if (obj instanceof Person) {
      Person p = (Person) obj;
      return name.equals(p.name);
    } else
        return false;
  }

  public String getName() { return name; }
  public void setName(String name) { this.name = name; }
  public String getStrasse() { return strasse; }
  public void setStrasse(String strasse) { this.strasse = strasse; }
  public String getPlz() { return plz; }
  public void setPlz(String plz) { this.plz = plz; }
  public String getOrt() { return ort; }
  public void setOrt(String ort) { this.ort = ort; }
}
```

```java
public class Student extends Person {
  private String studiengang;
  private int semesterzahl;
  private int matrikelNr;
  private static int lastMatrikelNr = 1000000;

  public Student(String name) {
    super(name);
    matrikelNr = getNextMatrikelNr();
  }

  public static int getNextMatrikelNr(){
    lastMatrikelNr++;
    return lastMatrikelNr;
  }
  public void druckeAdressetikett() {
    System.out.println(matrikelNr);
    super.druckeAdressetikett();
  }
  public String getStudiengang() {
    return studiengang;
  }
  public void setStudiengang(String studiengang) {
    this.studiengang = studiengang;
  }
  public int getSemesterzahl() {
    return semesterzahl;
  }
  public void setSemesterzahl(int semesterzahl) {
    this.semesterzahl = semesterzahl;
  }
  public int getMatrikelNr() {
    return matrikelNr;
  }
}
```

```java
public class Mitarbeiter extends Person {

  private int fachbereich;
  private String besoldungsgruppe;

  public Mitarbeiter(String name) {
    super(name);
  }

  public int getFachbereich() {
    return fachbereich;
  }
  public void setFachbereich(int fachbereich) {
    this.fachbereich = fachbereich;
  }
  public String getBesoldungsgruppe() {
    return besoldungsgruppe;
  }
  public void setBesoldungsgruppe(String besoldungsgruppe) {
    this.besoldungsgruppe = besoldungsgruppe;
  }
}
```

b) Wir stellen fest, dass sich die Adresse einer Person sehr gut in eine eigene Klasse auslagern und als Objekt verwalten lässt. Daher erstellen wir eine neue Klasse `Adresse` und fügen ein Objektattribut vom Typ `Adresse` zur Klasse `Person` hinzu. Nachfolgend ist der relevante Ausschnitt des angepassten Programms dargestellt.

```
public class Person {
  private String name;
  private Adresse adresse;

  //...

}
```

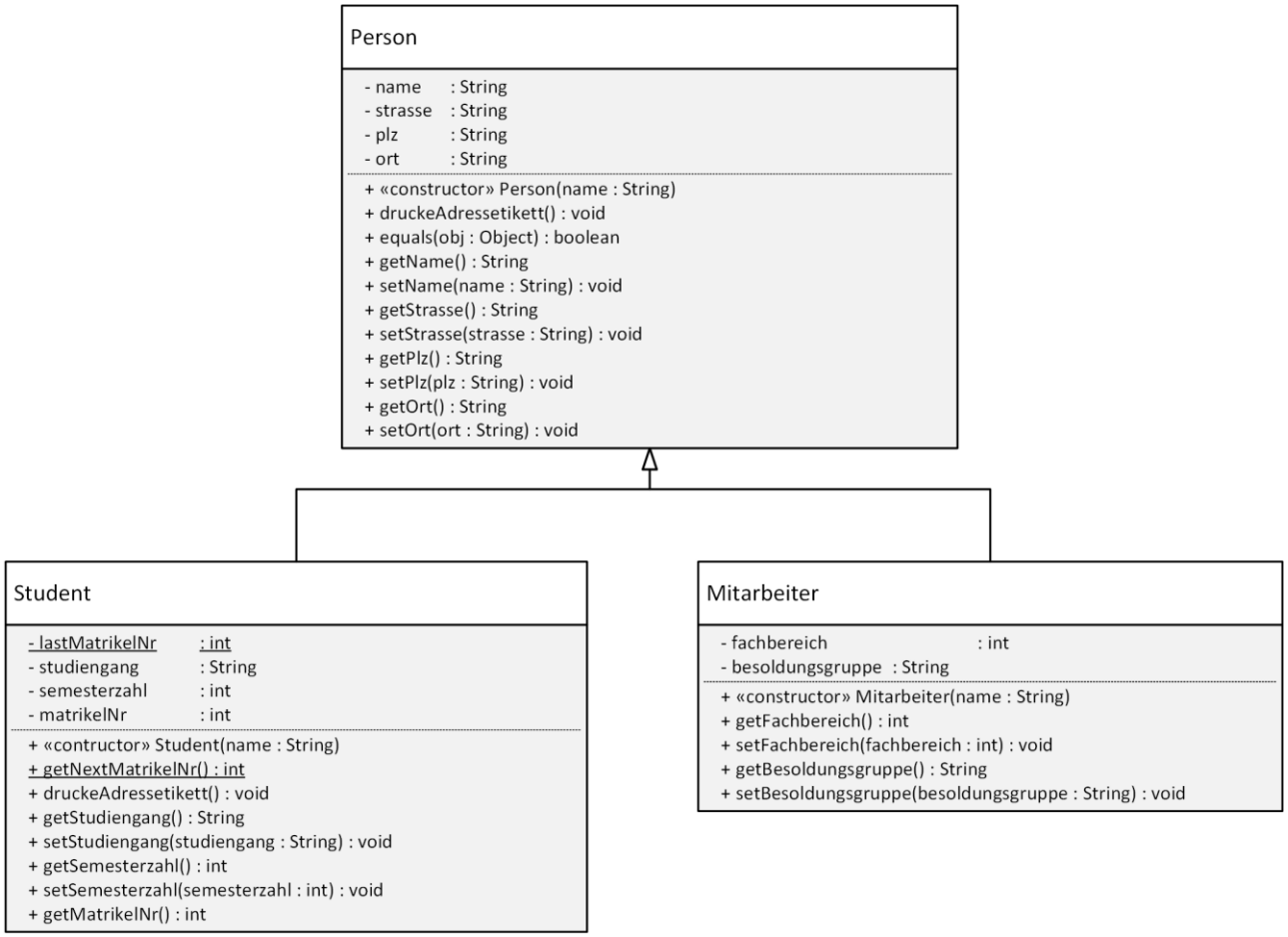
```
public class Adresse {
  private String strasse;
  private String plz;
  private String ort;

  //...
}
```

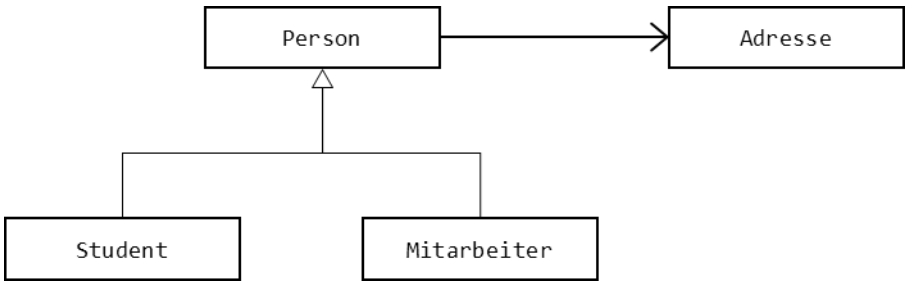
Ihre Aufgabe besteht nun darin, die Beziehungen zwischen den Klassen `Person`, `Adresse`, `Mitarbeiter` und `Student` in einem UML-Klassendiagramm zu modellieren. Dabei ist eine grobgranulare Modellierung vollkommen ausreichend, d. h. Sie können in diesem Fall auf die Darstellung von Attributen und Methoden verzichten.

## Lösung Aufgabe 2

a)




b)



## Aufgabe 3 (Fehlerbehandlung bei der Kontoverwaltung)

In den letzten Übungen haben wir eine kleine Kontenverwaltung mit verschiedenen Kontotypen geschrieben. In dieser Aufgabe soll nun ausschließlich die Klasse `Konto` (sowie die Klasse `Bank` als Hauptprogramm) betrachtet werden.

In den Methoden `einzahlen(double betrag)` und `auszahlen(double betrag)` der Klasse `Konto` werden Fehlersituationen durch Rückgabewerte vom Datentyp `boolean` signalisiert. Wie Sie in der Vorlesung gelernt haben, gibt es in Java eine bessere Möglichkeit, Fehlersituationen zu behandeln: mit Exceptions. Zur Erinnerung, hier der Quellcode der Klasse `Konto` aus dem letzten Lösungsvorschlag:

```
1   package gdp.uebung11.bank;
2
3   public class Konto {
4         public int kontonummer;
5         public double kontostand;
6
7         public Konto(int nr) {
8               kontonummer = nr;
9               kontostand = 0.0;
10        }
11
12        public boolean einzahlen(double betrag) {
13              if (betrag > 0) {
14                    kontostand = kontostand + betrag;
15                    return true;
16              } else {
17                    return false;
18              }
19        }
20
21        public boolean auszahlen(double betrag) {
22              if (betrag > 0 && kontostand >= betrag) {
23                    kontostand = kontostand - betrag;
24                    return true;
25              } else {
26                    return false;
27              }
28        }
29
30        public void druckeKontoauszug() {
31              System.out.println("Konto " + kontonummer + ": " + kontostand);
32        }
33  }
```

Überarbeiten Sie Ihr Programm zur Kontenverwaltung dahingehend, dass in der Klasse `Konto` Exceptions statt `boolean`-Werten zur Behandlung von Fehlersituationen verwendet werden. Gehen Sie dazu folgendermaßen vor:

a) Erstellen Sie eine Klasse `BankException`. Beachten Sie hierbei, von welcher Klasse diese abgeleitet werden muss! Die Klasse `BankException` soll im Folgenden die Basisklasse für die beiden programmspezifischen Ausnahmefälle `NegativerBetragException` und `KontoNichtGedecktException` sein. Erstellen Sie auch diese Klassen. Die Exceptions sollen sich alle im Paket `gdp.uebung11.bank.exceptions` befinden. Damit Sie beim Erstellen der Exceptions spezifische Fehlermeldungen definieren können, implementieren Sie außerdem in allen Exceptions einen Konstruktor mit einem String-Parameter, der den Konstruktor seiner Oberklasse mit dem übergebenen Wert aufruft.

b) Ändern Sie die Methoden `einzahlen` und `auszahlen` der Klasse `Konto`, sodass sie Exceptions anstelle von Rückgabewerten verwenden, um Fehlersituationen zu signalisieren. Diese Methoden benötigen dann keine Rückgabewerte mehr. Bei fehlgeschlagenen Transaktionen sollen entsprechende Fehlermeldungen in den geworfenen Exceptions gespeichert werden („Einzahlung/Auszahlung fehlgeschlagen, der Betrag ist negativ.“ und „Auszahlung fehlgeschlagen, das Konto ist nicht ausreichend gedeckt.“), wozu Sie die von Ihnen implementierten Konstruktoren verwenden können.

c) Erstellen Sie in der main-Methode ein `Konto` mit der Kontonummer 1042. Es sollen die folgenden Kontobewegungen vorgenommen werden und der Kontoauszug für das Konto nach jeder Transaktion ausgegeben werden. Wenn eine Transaktion fehlschlägt soll außerdem die Fehlermeldung aus der gefangenen Exception auf der Konsole ausgegeben werden. Nutzen Sie hierzu im catch-Block die von der Exception-Klasse geerbte Objektmethode `getMessage()`. Diese gibt die Fehlermeldung als String zurück, welche Sie dem Exception-Objekt bei Initialisierung übergeben haben.

| | |
|---|---|
| Einzahlung | -10,00 |
| Einzahlung | 500,00 |
| Auszahlung | 1,99 |
| Auszahlung | 500,00 |

## Lösung Aufgabe 3

a)

```
1  package gdp.uebung11.bank.exceptions;
2
3  public class BankException extends Exception {
4
5        public BankException(String msg) {
6              super(msg);
7        }
8  }
```

```
1  package gdp.uebung11.bank.exceptions;
2
3  public class NegativerBetragException extends BankException {
4
5        public NegativerBetragException(String msg) {
6              super(msg);
7        }
8  }
```

```
1  package gdp.uebung11.bank.exceptions;
2
3  public class KontoNichtGedecktException extends BankException {
4
5        public KontoNichtGedecktException(String msg) {
6              super(msg);
7        }
8  }
```

b) Unten sind nur die modifizierten Methoden gezeigt. Zusätzlich muss in der Klasse Bank ein Import der Exception-Klassen vorgenommen werden:

```
import gdp.uebung11.bank.exceptions.*;
```

```
1   package gdp.uebung11.bank;
2
3   import gdp.uebung11.bank.exceptions.*;
4
5   public class Konto {
6          public int kontonummer;
7          public double kontostand;
8
9          public Konto(int nr) {
10                kontonummer = nr;
11                kontostand = 0.0;
12         }
13
14         public void einzahlen(double betrag) throws NegativerBetragException {
15                if (betrag >= 0) {
16                       kontostand = kontostand + betrag;
17                } else {
18                       throw new NegativerBetragException("Einzahlung
                         fehlgeschlagen, der Betrag ist negativ.");
19                }
20         }
21
22         public void auszahlen(double betrag) throws NegativerBetragException,
           KontoNichtGedecktException {
23                if (betrag < 0) {
24                       throw new NegativerBetragException("Auszahlung
                         fehlgeschlagen, der Betrag ist negativ.");
25                } else if (kontostand < betrag) {
26                       throw new KontoNichtGedecktException("Auszahlung
                         fehlgeschlagen, das Konto ist nicht ausreichend gedeckt.");
27                } else {
28                       kontostand = kontostand - betrag;
29                }
30         }
31
32         public void druckeKontoauszug() {
33                System.out.println("Konto " + kontonummer + ": " + kontostand);
34         }
35  }
```

c)

```
1   package gdp.uebung11.bank;
2
3   import gdp.uebung11.bank.exceptions.*;
4
5   public class Bank {
6
7         public static void main(String[] args) {
8               Konto k = new Konto(1042);
9               try {
10                    k.einzahlen(-10);
11              } catch (BankException e) {
12                    System.out.println(e.getMessage());
13              }
14              k.druckeKontoauszug();
15
16              try {
17                    k.einzahlen(500);
18              } catch (BankException e) {
19                    System.out.println(e.getMessage());
20              }
21              k.druckeKontoauszug();
22
23              try {
24                    k.auszahlen(1.99);
25              } catch (BankException e) {
26                    System.out.println(e.getMessage());
27              }
28              k.druckeKontoauszug();
29
30              try {
31                    k.auszahlen(500);
32              } catch (BankException e) {
33                    System.out.println(e.getMessage());
34              }
35              k.druckeKontoauszug();
36        }
37  }
```

Ausgabe auf der Konsole:

```
Einzahlung fehlgeschlagen, der Betrag ist negativ.
Konto 1042: 0.0
Konto 1042: 500.0
Konto 1042: 498.01
Auszahlung fehlgeschlagen, das Konto ist nicht ausreichend gedeckt.
Konto 1042: 498.01
```

Anmerkung:
Wie Sie sehen können, kann in einem `catch` Block mit einer Oberklasse für Exceptions mehrere Typen einer Exception gefangen werden, hier also `NegativerBetragException` und `KontoNichtGedecktException` mittels der `BankException`. Wenn sie eine Exception in einer Methode werfen wollen (mit `throws`), müssen sie jedoch alle Exceptions, die geworfen werden sollen, einzeln nennen (siehe `auszahlen` in 3)b)).