Objekt-orientierte Programmierung - Vererbung und Polymorphie

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw.de, Raum 149B

Version: 1.0.4

Folien: https://delors.github.io/prog-java-oo-inheritance/folien.de.rst.html
https://delors.github.io/prog-java-oo-inheritance/folien.de.rst.html.pdf
Kontrollfragen: https://delors.github.io/prog-java-oo-inheritance/kontrollfragen.de.rst.html
Fehlermeldungen verstehen: https://delors.github.io/prog-java-oo-inheritance/fehlermeldungen_verstehen.de.rst.html
Fehler melden: https://github.com/Delors/delors.github.io/issues

1. Objektorientierte Programmierung mit Java - Vererbung und Polymorphie

Generalisierung und Spezialisierung

Implementierung einer Raumverwaltung

Ein Vorlesungsraum an der DHBW

```java
public class Vorlesungsraum {
    private int zustand; // 0: nutzbar,
                         // 1: zu renovieren
    private String gebaeude;
    private int ebene;
    private int raumNummer;
    private int kapazitaet;
    private boolean klimatisiert;
    private Ausstattung[] ausstattung;

    public void setzeZustand(int zustand)...
    public void setzeEbene(int ebene)...
    public void setzeRaum(int raumNr)...
    public void generiereBeschreibung()...

    public void reserviere()...
}
```

Eine Teeküche an der DHBW

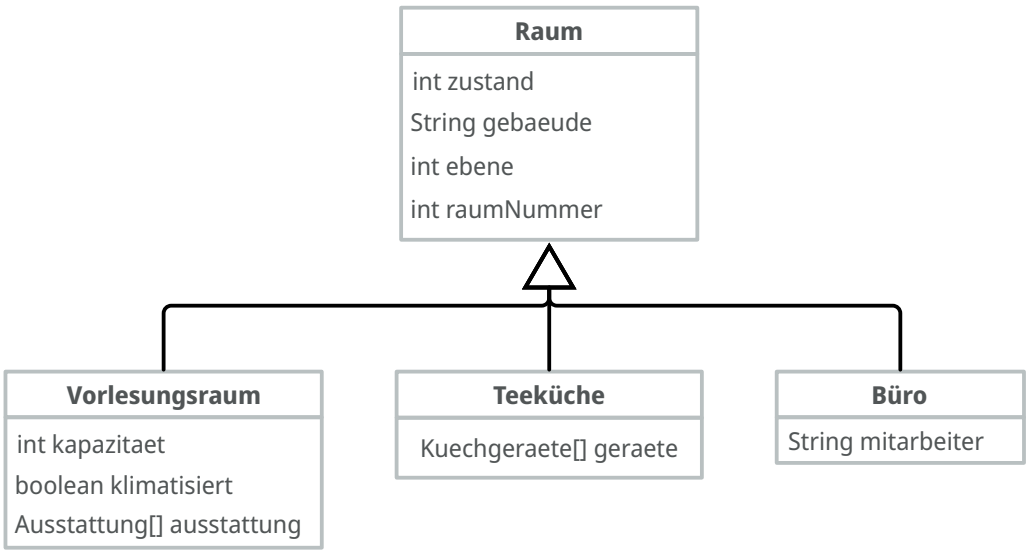
```java
public class Teekueche {
    private int zustand; // 0: nutzbar,
                         // 1: zu renovieren
    private String gebaeude;
    private int ebene;
    private int raumNummer;
    private Kuechgeraete[] geraete;



    public void setzeZustand(int zustand)...
    public void setzeEbene(int ebene)...
    public void setzeRaum(int raumNr)...
    public void generiereBeschreibung()...

    public void setzeSchliessberechtigung()...
}
```

Identifikation der Gemeinsamkeiten und Modellierung einer allgemeinen Klasse



Klassen können durch eine **Vererbungshierachie** in *Oberklassen* (*Superklassen*) (hier: `Raum`) und *Unterklassen* (Subklassen) (hier: `Vorlesungsraum`, `Buero`, `Teekueche`, ...) eingeteilt werden.

Unterklassen *spezialisieren* eine Oberklasse: Die Oberklasse definiert gemeinsame Attribute und Methoden. Eine Unterklasse kann neue Attribute und Methoden hinzufügen bzw. überschreiben. Dabei ist darauf zu achten, dass die Unterklasse sich verhaltenskonform zur Oberklasse verhält.

Vererbung (🇺🇸 *Inheritance*)

Definition

Erlaubt es, eine Klasse von einer anderen abzuleiten und deren Eigenschaften und Methoden zu erben.

Vorteile:

- Wiederverwendbarkeit des Codes / keine Code-Duplikation
- Erweiterbarkeit
- Hierarchische Strukturierung

Klassen werden in Vererbungshierachien eingeteilt.

Syntax:

```
class <Subklassenname>
        extends <Superklassenname> { ...
}
```

Beispiel:

```
class Auto {                                // Basisklasse
    String marke;
    void fahren() { System.out.println("Das Auto fährt."); }
}

class Elektroauto extends Auto {            // Abgeleitete Klasse
    int batteriestand;
    void aufladen() {
        System.out.println("Das Elektroauto wird aufgeladen.");
}   }
```

- Eine Unter- bzw. Subklasse erbt alle Attribute und Methoden der Super- bzw. Oberklasse.
- Auf `public` und `protected` Attribute und Methoden der Superklassen kann direkt zugegriffen werden.
- Auf `private` Attribute und Methoden kann nicht zugegriffen werden

  (Bei Attributen häufig indirekt nur über entsprechende öffentliche (`public`) `get`- und `set`-Methoden, welche auch als *Getter* und *Setter* bezeichnet werden.)
- Zyklen in der Vererbungshierarchie sind nicht erlaubt

Zugriff auf Methoden und Attribute von Superklassen

Mittels `super` ist der direkte Zugriff auf die Attribute und Methoden der Superklasse (wenn diese `protected` oder `public` sind) möglich.

- Dies ist notwendig, wenn die Elternklasse Attribute bzw. Methoden mit gleichem Namen enthält (ansonsten kann man `super` auch weglassen).

Verwendung von `super` für Aufruf der Methode der Superklasse

```
class Person {
```

```
    String name;
    int age;
    void display() {
        System.out.print("Name: " + name + " Age: " + age);
    }
}

class Kind extends Person {
    String hobby;
    void display() {
        super.display();
        System.out.print(" Hobby: " + hobby);
    }
}
```

Initialisierung von Superklassen

- Wird ein Objekt erzeugt (mittels `new`), so wird automatisch auch Speicher für die Attribute der Superklasse reserviert und initialisiert.
- Mittels eines `super(...)` Aufrufs ist es möglich einen bestimmten Konstruktor der Superklasse (innerhalb des Konstruktors der Subklasse) aufzurufen.
- Ruft der Konstruktor nicht explizit einen Konstruktor mit `super(...)` auf, dann wird der parameterlose Konstruktor `super()` implizit aufgerufen, wenn keiner explizit definiert wurde.
- Die Initialisierung startet immer bei der Superklasse und arbeitet sich dann rekursiv durch die Vererbungshierarchie nach unten.

Verwendung von `super` während der Initialisierung

```
class Angestellter {
    private String name;
    Angestellter(String name) { this.name = name; }
    String getName() { return name; }
}
class Professor extends Angestellter {
    private String fachgebiet;
    Professor(String name, String fachgebiet) {
        super(name);                // Aufruf des Konstruktors der Superklasse
        this.fachgebiet = fachgebiet;
    }
    public String toString() {
        return  "Professor { name = " + super.getName() // super hier optional
                + ", fachgebiet = " + fachgebiet + " }";
}   }
```

`java.lang.Object`

- Jede Klasse in Java erbt von der Klasse `java.lang.Object`.
- Die Klasse `java.lang.Object` definiert allgemein relevante Methoden wie `toString()`, `equals()` und `hashCode()`.

- Die Methode `toString()` gibt eine String-Repräsentation des Objekts zurück und wird aufgerufen, wenn ein Objekt in einem String-Kontext verwendet wird.

```
1 void main() {
2     IO.println("Mein Prof.: " + new Professor("Max Mustermann", "Informatik"));
3 }
```

- Die Methode `getClass()` erlaubt den Zugriff auf die Klasse eines Objekts und ermöglicht `Reflection`. Thema für spätere Vorlesung(en).

Methoden überschreiben

- Eine Methode in einer Subklasse kann eine Methode in der Superklasse überschreiben.
- Eine Methode, die eine Methode in der Superklasse überschreibt, hat den Kontrakt der Superklasse immer einzuhalten!

  D. h. Vorbedingungen können in der Subklassen „entspannt“ und Nachbedingungen „verschärft“ werden, aber nie umgekehrt.

Einfach- vs. Mehrfachvererbung

Einfachvererbung:

Jede Klasse kann nur eine Superklasse in der Vererbungshierachie besitzen

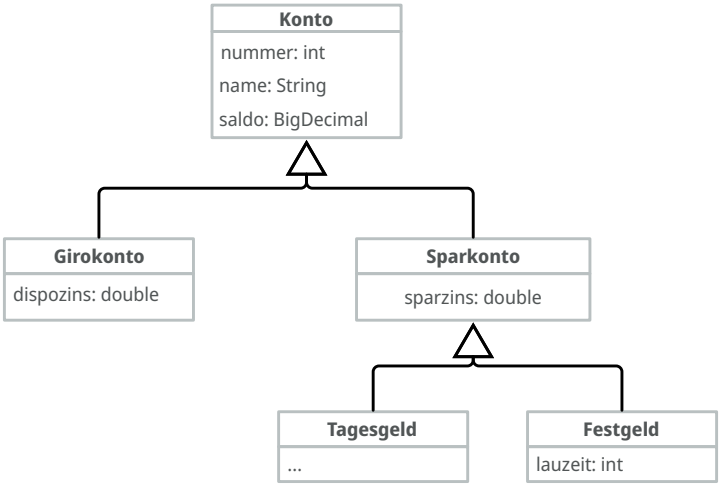
Mehrfachvererbung:

Jede Klasse kann mehrere Superklassen in der Vererbungshierachie besitzen

> ⚠ **Achtung!**
>
> Java unterstützt nur Einfachvererbung bei Klassen.
>
> Mehrfachvererbung wird nur bei Schnittstellen (`interface`) unterstützt. Hier spricht man jedoch in der Regel davon, dass eine Klasse mehrere Schnittstellen *implementiert* und nicht von Mehrfachvererbung.

Vererbung und Typkonvertierungen/-kompatibilität

Im Folgenden gehen wir von der folgenden Vererbungshierarchie aus:



Alle Attribute und Klassen sein `public`.

Statischer und Dynamischer Typ

- Eine Referenzvariable (für ein Objekt) hat einen statischen und einen dynamischen Typ.
- Der statische Typ ist durch die Deklaration der Referenzvariablen gegeben.

  Beispiel: `Konto k; //Statischer Typ "Konto"`
- Der dynamische Typ hängt vom konkreten Objekt ab; es ist der Typ der Klasse, von der das Objekt instanziiert wurde mittels `new`.

  Beispiel: `k = new Sparkonto(...); // Dynamischer Typ "Sparkonto“`
- Der dynamische Typ muss von einer (nicht echten) Unterklasse des statischen Typs sein (z. B. „Sparkonto“ als dynamischer und „Konto“ als statischer Typ.)
- Über die Referenzvariable sind nur die sichtbaren Attribute und Methoden des statischen Typs ansprechbar.

  Im Fall von `Konto k = new Sparkonto(...);` kann nicht auf `sparzins` zugegriffen werden.

Achtung!

Der dynamische Typ bestimmt die Methode, die ausgeführt wird.

D. h. eine Methode, die in der Subklasse überschrieben wurde, wird auch dann ausgeführt, wenn die Referenzvariable den statischen Typ der Oberklasse hat.

Implizite Typkonvertierung

- Eine implizite Typkonvertierung (ohne cast-Operator) ist in der Vererbungshierarchie aufwärts möglich (Upcast).

  Beispiel: Ein Tagesgeldkonto kann immer in ein Sparkonto konvertiert werden. Nach der Konvertierung sind über die Referenzvariable nur noch Attribute und Methoden des statischen Typs Sparkonto „sichtbar“.

- Das Objekt selbst wird bei einer impliziten Konvertierung nicht geändert, nur die sichtbaren Attribute und Methoden unterscheiden sich.
- Die implizite Typkonvertierung ist sicher; es kann kein Fehler bei der Typkonvertierung entstehen.

Explizite Typkonvertierung

- Typkonvertierung in der Vererbungshierarchie abwärts (Downcast) ist nur durch explizite Typkonvertierung (mit cast-Operator) möglich

  Beispiel

  Ein Konto kann „möglicherweise“ in ein Sparkonto konvertiert werden:

  ```
  Sparkonto sk = (Sparkonto) konto;
  ```

- Nach der Konvertierung sind über die Referenzvariable die Attribute und Methoden des statischen Typs Sparkonto „sichtbar“.
- Das Objekt selbst wird bei einer expliziten Konvertierung nicht verändert!
- Die Typkonvertierung ist nicht sicher; es kann ein Fehler bei der Typkonvertierung entstehen. Eine sogenannte *Typecast Exception* ist dann die Folge.

Typkonvertierung - Details

- Eine explizite Konvertierung eines Objektes ist nur dann möglich wenn der dynamische Typ des Objektes gleich der Ziel-Klasse ist bzw. der dynamische Typ des Objektes eine Subklasse der Ziel-Klasse ist.

  Beispiele:

  - Ein Objekt wird als Festgeldkonto angelegt und implizit in ein Konto konvertiert (d. h. der dynamische Typ ist Festgeldkonto). Eine explizite Konvertierung in ein Sparkonto ist möglich.
  - Wird allerdings ein Objekt als Sparkonto angelegt, dann kann es nicht explizit in ein Tagesgeldkonto konvertiert werden.

Typtest mit `instanceof`

- Der `instanceof`-Operator testet ob ein Objekt kompatibel zu einer Klasse ist (d. h. ob das Objekt in die Klasse konvertierbar ist). Der Operator gibt `true` oder `false` zurück:

  Syntax:

  ```
  <Objekt> instanceof <Klasse>
  ```

  Beispiel:

  `k instanceof Sparkonto` testet ob das Objekt `k` in ein Sparkonto explizit konvertiert werden kann. Hier nur möglich, wenn `k` den dynamischen Typ Sparkonto, Festgeldkonto oder Tagesgeldkonto hat.

Sollte `k` `null` sein, dann ist das Ergebnis immer `false`.

Beispiele

```
1    Konto k1 = new Festgeld (1, "Matt", 100, 2.5, 36);
2
3    // Test der Typkompatibilität mit instanceof Festgeld
4    if(k1 instanceof Festgeld){
5        // Explizite Konvertierung ist jetzt sicher:
6        Festgeld k2 = (Festgeld)k1;
7        System.out.println(k2);
8    }
```

Bzgl. des Zugriffs auf Methoden mit *Default* Sichtbarkeit gelten die Standardregeln.

Neben der klassischen Einfach- und Mehrfachvererbung gibt es noch viele weitere Konstrukte (z. B. traits, mixins, ...), die in anderen Programmiersprachen verwendet werden und ähnliche Konzepte ermöglichen.

Warnung

Die Klasse `java.lang.Object` definiert eine Reihe von Methoden, die als veraltet markiert sind. Diese sollten *nicht verwendet werden* und wir gehen hier auch nicht weiter auf diese ein!

Polymorphie (Polymorphism)

Definition

Eine Referenzvariable mit einem statischen Typ kann auf Objekte mit unterschiedlichem dynamischen Typ verweisen.

Verwendung:

- Überschreiben von Methoden (*Runtime Polymorphism*)
- Parameter und Rückgabewerte: Methoden können als Parameter Objekte einer beliebigen Subklasse übergeben bekommen bzw. zurückgeben.
- ein Array kann Objekte jeder Subklasse enthalten (z. B. ein Array mit dem Datentyp `Konto[]` kann alle Subklassen enthalten.)

Beispiel

Arrays und Polymorphie:

```
Festgeld k1 = new Festgeld(1, "Matt", 100, 2.5, 36);
Sparkonto k2 = new Sparkonto(1, "Michael", 100, 3);

// Objekte mit unterschiedlichem dynamischen Typ in einem Array
Konto[] konten = {k1, k2};
for(int i = 0; i < konten.length; ++i){
    println(konten[i]);
}
```

Beispiel

Methode `fahren` wird in verschiedenen Klassen unterschiedlich implementiert:

```
class Auto {
    void fahren() {
        System.out.println("Das Auto fährt.");
    }
}

class Elektroauto extends Auto {
    void fahren() { // Überschreiben der Methode
        System.out.println("Das Elektroauto fährt leise.");
    }
}
```

Wir sprechen hier vom überschreiben (*overriding*) von Methoden.

Methoden überschreiben:

- Deklaration einer Methode mit der gleichen Schnittstelle (Name, Rückgabetyp, Parameter) aber ggf. mit neuem Methodenrumpf.
- Eine Methode kann in einer Subklasse eine erhöhte Sichtbarkeit haben, aber keine einschränkendere!

- Methoden die `final` sind können in Subklassen nicht überschrieben werden.
- Methoden die `private` sind, sind in Subklassen nicht sichtbar und können daher nicht überschrieben werden.

  Wenn die Subklasse eine Methode mit dem gleichen Namen und den gleichen Parametern definiert, dann handelt es sich um eine neue Methode und nicht um eine Überschreibung. Ob diese neue Methode auch (wieder) `private` ist, ist nicht weiter von belang!

---

overriding und Overloading sind zwei verschiedene Konzepte. Bei Overloading wird eine Methode mit dem gleichen Namen aber unterschiedlichen Parametertypen definiert. Bei Overriding wird eine Methode mit dem gleichen Namen und den gleichen Parametertypen in einer Subklasse neu definiert.

Zusammenfassung und Vorteile von Objekt-orientierter Programmierung[1]

Kapselung:
: Schützt die Daten und kontrolliert den Zugriff.

Abstraktion:
: Vereinfacht die Komplexität des Codes.

Vererbung:
: Ermöglicht Code-Wiederverwendung und Hierarchien.

Polymorphie:
: Erlaubt flexiblen Code durch unterschiedliche Implementierungen.

---

[1] Diese Vorteile gelten im Wesentlichen für alle objektorientierten Programmiersprachen.

Übung

1.1. Meine Erste Klassenhierarchie

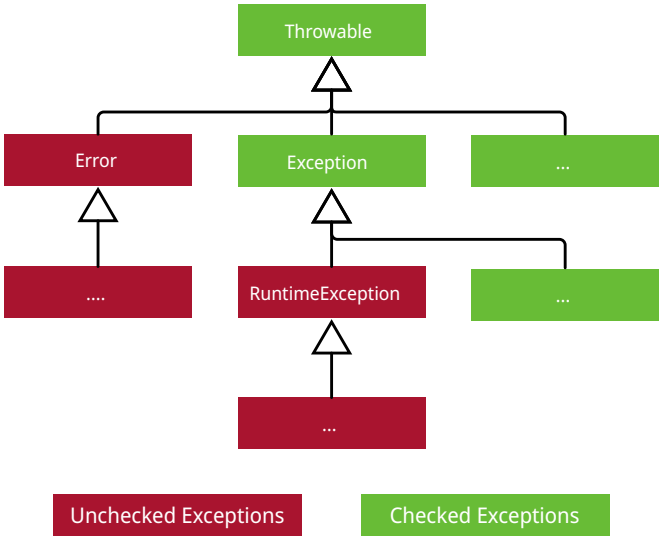
Erstelle eine einfache `Tier`-Klasse mit einem Attribut `decibel` vom Typ `float` und einer Methode `lautGeben()`, die den Laut des Tieres auf der Konsole ausgibt und einer Methode `decibel`, die die Lautstärke als `String` zurückgibt. Erstelle dann die Klassen `Hund` und `Katze`, die `Tier` erweitern bzw. von `Tier` erben. Überschreibe die Methode *lautGeben()* mit unterschiedlichen Ausgaben.

2. Fehlerbehandlung in Java

Fehlerbehandlung (🇺🇸 *Exceptions*, 🇩🇪 *Ausnahmen*)

- Die Fehlerbehandlung in Java erfolgt mittels Exceptions.
- Exceptions sind Objekte, die eine Fehlermeldung und den *Stacktrace* enthalten.
- Exceptions erben direkt oder indirekt von `Throwable`.
- Exceptions können geworfen (mit `throw`) und gefangen (mit `try` und `catch`) werden.
- Exceptions können *checked* oder *unchecked* sein:
  - *Checked Exceptions* (Klassen, die von `Throwable` erben aber nicht von `RuntimeException` oder `Error`) müssen gefangen oder deklariert werden.
  - *Unchecked Exceptions* (Exceptions, die von `java.lang.RuntimeException` erben) können im Code ignoriert werden; d. h. müssen nicht explizit beachtet werden. Sollten/müssen aber nicht.

Exceptions Typhierearchie



Einige ausgewählte typische Exceptions

Unchecked Exceptions:

- `ArithmeticException`: Division durch `0`.
- `NullPointerException`: Ein Objekt wird verwendet, obwohl es `null` ist.
- `ArrayIndexOutOfBoundsException`: Ein ungültiger Index wird verwendet.
- `IllegalArgumentException`: Ein ungültiges Argument wird übergeben.

Checked Exceptions:

- `IOException`: Fehler beim Lesen oder Schreiben von Dateien.
- `FileNotFoundException`: Datei nicht gefunden.
- `ParseException`: Fehler beim Parsen von Strings.

Handling von *Unchecked Exceptions* (`try ... catch (E e)`)

```
1 import static java.lang.System.err;
2 void main(String[] args) {
3     try {
```

```
        int i = Integer.parseInt(args[0]);
        int j = Integer.parseInt(args[1]);
        try {
            IO.println(i / j);
        } catch (ArithmeticException e) {
            err.println("Division durch 0 nicht erlaubt.");
        }
    } catch (NumberFormatException e) {
        err.println("Fehler beim Parsen der Argumente.");
    } catch (ArrayIndexOutOfBoundsException e) {
        err.println("Zu wenige Argumente.");
}   }
```

Handling von *Checked Exceptions* (`try ... catch (E e)`)

```
import java.text.DateFormat;
import static java.lang.System.err;

void main(String[] args) {
    DateFormat df = DateFormat.getDateInstance(DateFormat.SHORT);
    try {
        IO.println("Thats the day: " + df.parse(args[0]));
    } catch (Exception e) {
        err.println("Error: " + e.getMessage()+
                " Expected format: "+df.format(new Date()));
    }
}
```

Identische Behandlung von mehreren Exceptions (`... catch (A | B e)`)

```
import static java.lang.System.err;
void main(String[] args) {
    try {
        int i = Integer.parseInt(args[0]);
        int j = Integer.parseInt(args[1]);
        try {
            IO.println(i / j);
        } catch (ArithmeticException e) {
            err.println("Division durch 0 nicht erlaubt.");
        }
    } catch (NumberFormatException | ArrayIndexOutOfBoundsException e) {
        err.println("Argumente falsch.");
}   }
```

Es wäre auch möglich gewesen die gemeinsame Superklasse zu nehmen (`RuntimeException`). Dies würde jedoch dazu führen, dass man Ausnahmen fängt, die man gar nicht fangen will!

Deklaration, dass eine *Checked Exceptions* geworfen werden könnte (`throws`)

```
import java.text.DateFormat;
import java.text.ParseException;

void main(String[] args) throws ParseException {
    DateFormat df = DateFormat.getDateInstance(DateFormat.SHORT);
```

```
        IO.println("Thats the day: " + df.parse(args[0]));
}
```

Try-with-Resources (`try(var i = <Ressource>) { ... }`)

Stellt sicher, dass eine Ressource (z. B. eine Datei) immer geschlossen wird, auch wenn eine Exception auftritt.

```
import static java.lang.System.err;

void main(String []args){
    try (var in = new BufferedReader(new FileReader(args[0]))) {
        String line;
        while ((line = in.readLine()) != null) { println(line); }
    } catch (IOException e) {
        err.println("Error: " + e.getMessage());
    }
}
```

Der explizite Exceptionhandler wird nach dem Schließen der Ressource aufgerufen.

Exceptions können selbstverständlich auch selbst definiert werden. Im Allgemeinen empfiehlt es sich aber, die Standard-Exceptions zu verwenden, da diese von anderen Entwicklern erkannt und verstanden werden.

`Errors` sind Exceptions, die nicht gefangen werden sollten. Sie sind für den Programmierer nicht vorhersehbar und können im ganz Allgemeinen nicht sinnvoll behandelt werden. Sie signalisieren zum Beispiel Fehlerzustände der virtuellen Maschine. Ein Beispiel ist der `OutOfMemoryError`.

Übung

2.1. Einfache Fehlerbehandlung

Erweiteren Sie Ihre Methoden zum Berechnen der Kubikwurzel und zur Berechnung der Fibonacci-Zahlen um Fehlerbehandlung. D. h. testen Sie die Parameter auf Gültigkeit und werfen Sie ggf. eine `IllegalArgumentException`.

Deklarieren Sie in der Methodensignatur, dass eine `IllegalArgumentException` geworfen werden könnte.

Bedenken Sie bei der Berechnung der Methode für die Kubikwurzel, dass Double Werte auch Spezialwerte wie `Double.POSITIVE_INFINITY` und `Double.NaN` haben können!

Ändern Sie Ihre `main` Methode so, dass sie die Exceptions fängt und eine entsprechende Fehlermeldung ausgibt und dann sauber das Program beendet.

Übung

2.2. Nicht-leere Zeilen zählen

Schreiben Sie eine Methode (`countNonEmptyLines`), die die Anzahl der nicht-leeren Zeilen in einem Datenstrom zählt und zurückgibt. Eine Zeile wird als leer angesehen, wenn diese keine Zeichen oder nur Leerzeichen enthält. Verwenden Sie dazu die Klasse `BufferedReader` und die Methode `readLine()` (siehe Beispiel in den Folien).
Die Methode soll sich nicht um Fehlerbehandlung kümmern.

Schreiben Sie eine `main` Methode, die die Methode verwendet und sich um jegliche Fehlerbehandlung kümmert. D. h. die `main` Methode soll bei allen Fehlern eine *passende Fehlermeldung* ausgeben und das Programm sauber beenden. Verwenden Sie ggf. ein `try-with-ressource` Statement.

※ Hinweis

Studieren Sie die Dokumentation der Klasse `String` in Hinblick auf Methoden, die es Ihnen einfacher machen zu erkennen ob eine Zeile gemäß obiger Definition leer ist.

3. Abstrake Klassen und finale Klassen/Methoden

Abstrakte Klassen

- Abstrakte Klassen deklarieren ein Grundgerüst einer Klasse von der keine Objekte erzeugt werden können.
- Abstrakte Klassen können abstrakte Methoden enthalten, die nur die Schnittstelle einer Methode definieren, aber auch implementierte Methoden.
- Abstrakte Klassen und abstrakte Methoden werden durch den Modifizierer `abstract` gekennzeichnet.
- Nicht abstrakte Subklassen einer abstrakten Klasse müssen *alle* abstrakten Methoden der Elternklasse implementieren.

Beispiel

Eine *Form*-Klasse, die über verschiedene Unterklassen wie *Kreis*, *Quadrat* und *Dreieck* abstrahiert. Alle Formen bieten eine Möglichkeit zur Berechnung der Fläche.

```
public abstract class EinfacheForm {
    protected double hoehe;
    abstract double berechneFlaeche();
    double berechneVolumen() {
        return berechneFlaeche() * hoehe;
}   }
```

```
class Kreis extends EinfacheForm {
    double r = 0.0;
    double berechneFlaeche() {
        return Math.PI * r * r;
}   }
```

```
class Quadrat extends EinfacheForm {
    double seite = 0.0;
    double berechneFlaeche() {
        return seite * seite;
}   }
```

Abstrakte Methoden (`abstract`)

- Abstrakte Methoden, dürfen nicht `private`, `final` oder `static` sein.
- Abstrakte Methoden können von nicht-abstrakten Methoden aufgerufen werden.
- Abstrakte Klassen können von anderen (auch nicht-abstrakten) Klassen erben.
- Konkrete Subklassen müssen alle abstrakten Methoden implementieren.

Statischer Typ

- Abstrakte Klassen können als statischer Typ von Referenzvariablen verwendet werden.
- Klassen, die von einer abstrakten Klasse erben, sind typkonform zu der abstrakten Klasse und können implizit in diese konvertiert werden.
- Referenzvariablen (Abstrakte Klassen) können an den gewohnten Stellen verwendet werden.

Finale Klassen und Methoden (der `final` Modifikator)

- Durch den Modifikator `final` kann das Überschreiben von Methoden bzw. ganzen Klassen verhindert werden.
- Methoden, die durch den Modifikator `final` gekennzeichnet sind, können in Subklassen nicht überschrieben werden.
- Von Klassen, die durch den Modifikator `final` gekennzeichnet sind, können keine Subklassen abgeleitet werden

Konto.java

```java
public class Konto {
    private String name;
    protected double saldo;

    public final double getSaldo(){
        return saldo;
    }

    public final void setSaldo(double saldo){
        this.saldo = saldo;
    }
}
```

Festgeldkonto.java

```java
public final class Festgeldkonto extends Konto {
        private int laufzeit;
     //...
}
```

Attributen, die als `final` markiert sind, kann nur einmal einen Wert zuweisen. Dies hat mit Vererbung nichts zu tun.

Übung

3.1. Vererbung, Exceptions und Abstrakte Klassen

Wir möchten mathematische Ausdrücke repräsentieren, um darauf verschiedene Operationen auszuführen.

Erstellen Sie eine abstrakte Klasse `Term`, die eine Methode `int evaluate()` deklariert. Die Methode `evaluate` soll eine *Checked Exception* vom neu anzulegenden Typ `MathException` werfen, wenn die Auswertung nicht möglich ist. Die abstrakte Klasse `Term` hat ein privates Attribut mit der Priorität des Terms (als `int` Wert), welcher bei der Initialisierung gesetzt wird. Implementieren Sie eine passende finale Methode `int getPriority()` in der abstrakten Klasse. Die Priorität eines Terms ist relevant, wenn man einen Ausdruck ausgeben möchte und die Klammern minimieren möchte.

Erstellen Sie dann die Klassen `Number`, `Plus` und `Division`, die von `Term` erben und ggf. Referenzen auf weitere Terme halten. `Number` repräsentiert eine Zahl, `Plus` eine Addition und `Division` eine Division. Implementieren Sie die Methode `int evaluate()` in den Subklassen. Legen Sie für jede Klasse einen passenden Konstruktor an. Werfen Sie ggf. eine `MathException`, wenn die Auswertung nicht möglich ist.

Implementieren Sie für jede konkrete Klasse eine Methode `public String toString()`, die den Term als `String` zurückgibt und Klammerung durchführt *wenn notwendig*.
Die Methode `toString()` soll die Klammern so setzen, dass der Ausdruck korrekt ist.
D. h. `(1 + 2) * 3` soll als `(1 + 2) * 3` und nicht als `1 + 2 * 3` ausgegeben werden.
Weiterhin soll eine Ausdruck wie `1 + 2 + 3` als `1 + 2 + 3` und nicht als `1 + (2 + 3)` oder `(1 + 2) + 3` ausgegeben werden.

Schreiben Sie eine `main` Methode und testen Sie mit verschiedenen Termen die Auswertung und die Ausgabe.

Achten Sie darauf, dass im Falle einer Exception eine passende Fehlermeldung ausgegeben wird.

Beispiele für die Verwendung

Beispiel

```
System.out.println(
    new Division(
        new Number(1),
        new Plus(
            new Plus(new Number(1),new Number(2)),
            new Number(1))));
```

Ausgabe:

```
1 / (1 + 2 + 1)
```

Beispiel

```
System.out.println(
    new Plus(
        new Number(1),
        new Division(new Number(2), new Number(1))));
```

Ausgabe:

```
1 + 2 / 1
```

4. Definition und Verwendung von Schnittstellen in Java

Java interfaces

Schnittstellen (Interfaces) werden ähnlich wie Klassen deklariert, spezifizieren aber nur Methoden-Schnittstellen (abstrakte Methoden und `default` Methoden) und öffentliche statische finale Attribute.

Syntax:

```
<public>? interface <Schnittstellenname>{
    // statische, finale Attribute
    // Methodendeklarationen und "default" Methoden
}
```

Beispiel

Saeugetier.java

```java
interface Saeugetier {
    int[] getAnzahlZitzen();
    default int getAnzahlNachkommen() {
        int durchschnittlicheZitzenAnzahl =
            java.util.Arrays.stream(getAnzahlZitzen()).sum() / 2;
        return durchschnittlicheZitzenAnzahl / 2;
    }
}
```

Haustier.java

```java
interface Haustier {
    int MAXIMALES_ALTER = 100;
    public abstract String getRufname();
}
```

Details:

- Von Schnittstellen können keine Objekte erzeugt werden.
- Schnittstellen können aber als statischer Typ eines Objektes verwendet werden.
- Die Angabe von `public abstract` bei Methoden ist optional.
- Die Angabe von `public final static` bei Attributen ist optional.

Implementierung von Schnittstellen

Eine Klasse kann mehrere Schnittstellen implementieren. Die Methoden der Schnittstellen müssen in der Klasse implementiert werden.

Katze.java

```java
public class Katze implements Haustier, Saeugetier {
    private String rufname;
    public Katze(String rufname) { this.rufname = rufname; }

    public String getRufname() { return rufname; }              // von Haustier
    public int[] getAnzahlZitzen() { return new int[] {6, 8 }; }// von S-tier
```

```
7
8       public String toString() {
9           return "Katze { rufname = " + rufname + " }";
10  }   }
```

Vererbung von Schnittstellen

Eine Schnittstelle kann von einer oder mehrerer Schnittstellen erben.

Syntax:

```
interface <Schnittstelle>
    extends <Schnittstelle> (, <Schnittstelle>)* {
    //...
}
```

Beispiel

Schnittstellenvererbung

```
1  interface Carnivora /*Raubtiere*/ extends Saeugetier {
2      Tier bevorzugteJagdBeute();
3  }
```

Statischer Typ

- Schnittstellen können (wie Klassen) als statischer Typ von Objekten(Referenzvariablen) verwendet werden.
- Klassen, die eine Schnittstelle implementieren, sind typkonform zu der Schnittstelle und können implizit in diese konvertiert werden.
- Referenzvariablen (mit den statischen Datentyp einer Schnittstellen) können an den gewohnten Stellen verwendet werden.

Es ist nicht möglich Interfaces mit Methoden mit inkompatiblen Signaturen zu implementieren. Es ist aber möglich, dass eine Klasse mehrere Interfaces implementiert, die Methoden mit gleichen Signaturen haben. In diesem Fall muss die Klasse die Methode nur einmal implementieren.

Beispiel:

```
interface Foo {
    int m();
}

interface Bar {
    default double m(){return 1;}
}

class FooBar implements Bar, Foo {
    public int m() { /* Main.java:10: error: m() in Main.FooBar
                                             cannot implement m() in
                                             public int m() {
                                                        ^
                                             return type int is not c
                                             with double */
        return 2;
    }
}

void main() {
    FooBar fb = new FooBar();
    System.out.println(fb.m() + " " + ((Bar)fb).m());
}
```

Übung

4.1. Ausdrücke vergleichen (Schnittstellen, instanceof, Type Casts)

Erweitern Sie die Lösung der vorhergehenden Übung wie folgt.

Definieren Sie eine Schnittstelle `Comparable`, die eine Methode `boolean equal(Term t)` deklariert. Implementierungen der Methode sollen den aktuellen Term vergleichen mit dem Übergebenen und `true` zurückgeben, wenn der aktuelle Term (`this`) identisch zum übergebenen Term (`t`) ist. Beachten Sie das Kommutativgesetz beim Vergleich; d. h. `a + b` ist gleich `b + a`.

Die abstrakten Klasse `Term` soll die Schnittstelle implementieren. Die Implementierungen der Methoden müssen natürlich in den Subklassen erfolgen.

Beispiel

```
System.out.println(
        new Plus(new Number(1), new Number(2))
    .equal(
        null));
System.out.println(
        new Plus(new Number(1), new Number(2))
    .equal(
        new Plus(new Number(1), new Number(2)))
);
System.out.println(
        new Plus(new Number(2), new Number(1))
    .equal(
        new Plus(new Number(1), new Number(2)))
);
```

Ausgabe:

```
false
true
true
```

Übung

4.2. Modellierung einer Autoteile-Hierarchie

Ein Kfz-Ersatzteilhändler möchte sein Warenwirtschaftssystem um eine objektorientierte Verwaltung von Autoteilen erweitern.

1. Erstellen Sie eine Klasse `Autoteil`, die die gemeinsamen Attribute und Methoden aller Autoteile definiert. Jedes Autoteil hat eine eindeutige `Teilenummer` (als `String`) und eine `Bezeichnung` (als `String`). Implementieren Sie einen passenden Konstruktor und entsprechende Getter. Achten Sie auf sauberer Kapselung.
2. Es wurde entschieden als primäre Dimension bei der objektorientierten Modellierung die Unterscheidung der Autoteile nach Fahrzeugbaugruppen zu wählen. Diesbezüglich soll zwischen den Teilen, die zum Antrieb, und denen, die zum Fahrwerk gehören, unterschieden werden. Zur Bauteilgruppe des Fahrwerks gehören insbesondere auch die `Reifen`. Reifen sollen immer einen Reifenluftdruck und eine maximale Laufleistung besitzen. Bilden Sie diese Beziehungen durch eine Vererbungshierarchie ab. Achten Sie auch auf eine angemessene Bezeichnung der Klassen.
3. Orthogonal zur Vererbungshierarchie sollen die Autoteile nach der Sicherheitsrelevanz und der Einstufung als Verschleißteil unterschieden werden. Erweitern Sie Ihre Klassenhierarchie durch passende Schnittstellen. Verschleißteile sind insbesondere `Reifen`. Für Verschleißteile soll die Anzahl der Kilometer, die das Teil mindestens halten sollte, abfragbar sein. Bei sicherheitsrelevanten Teilen soll die Information über die zugrundeliegende Norm abrufbar sein.