
# Skript Informatik - Objektorientierte Programmierung mit Java

Dozentin:

Prof. Dr. Kirstin Baumann

Fakultät:

Mechanical and Medical Engineering

# Inhaltsverzeichnis

# 1 Einführung und Objektorientierung

***Lernziele:***

- **Wissen**
  - Wissen, was objektorientierte Softwareentwicklung ist
  - Die Begriffe objektorientierte Analyse und objektorientierter Entwurf kennen
  - Wissen, welche objektorientierten Konzepte es gibt
  - Wissen, was ein Pflichtenheft enthalten sollte
- **Verstehen**
  - Das Prinzip (die Methodik) der Objektorientierung verstehen
  - Erklären können, wie sich die Phasen Analyse und Entwurf voneinander unterscheiden

## 1.1 Einleitung

Wenn Sie ein Computerprogramm schreiben, dann erstellen Sie ein Modell eines Ausschnitts der realen Welt. Dieser Ausschnitt setzt sich aus Objekten zusammen, die im Anwendungsbereich vorkommen. In der Objektorientierung werden diese Objekte der realen Welt als Objekte im Modell abgebildet.
Für die Entwicklung umfangreicher Software reicht es nicht aus sich nur mit der Programmierung zu beschäftigen. die Softwareentwicklung besteht aus den Phasen Analyse (Problembeschreibung), Entwurf (erstellen eines Modells für das spätere Programm), Implementierung (schreiben des Quelltextes) und Test. Auch wenn wir uns schwerpunktmäßig mit der objektorientierten Programmierung beschäftigen wollen, werden Sie auch die Grundlagen von Analyse und Entwurf und einige Ansätze zum Testen kennen lernen.

## 1.2 Prozedurale Programmierung

Der Schwerpunkt bei der prozeduralen Programmierung liegt auf dem Algorithmus, d.h. der Ausführung der benötigten Berechnungen. Prozeduren oder Funktionen werden dazu verwendet, Ordnung in die Algorithmen zu bringen, indem mehrere Befehlsfolgen in einer Funktion zusammengefaßt werden. Sprachmittel, die Argumente an Funktionen übergeben und Werte aus Funktionen zurückgeben, unterstützen diese Technik. Ziel ist es, die zu lösende Aufgabe sinnvoll in einzelne Prozeduren zu unterteilen.
Beispiel 1.1: Es soll eine komplexe Fläche berechnet werden, die sich aus Kreisen, Dreiecken und Rechtecken zusammensetzt.

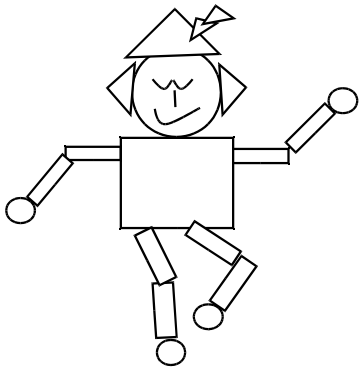


Abbildung 1.1

Zur Berechnung der Gesamtfläche werden zunächst die Einzelflächen ermittelt und dann wird die

Summe aller Flächen gebildet. Alle Kreisflächen werden mit der gleichen Formel berechnet, alle Rechtecksflächen mit einer zweiten Formel und alle Dreiecksflächen mit einer anderen jedoch für alle Dreiecke gleichen Formel. Lediglich die Werte der benötigten Seiten oder Radien sind bei jeder Fläche individuell. Daher ist es sinnvoll drei Funktionen oder Prozeduren zur programmieren. Jede Funktion steht für die Umsetzung einer der drei Formeln. Diese Funktionen erhalten Daten (Werte), die abhängig sind vom jeweiligen Anwendungsfall (hier: der zu berechnenden Fläche), und sie erzeugen ein Ergebnis. Möchten ie eine bestimmte Fläche berechnen, so benötigen Sie einen Funktionsaufruf. Dabei geben Sie Werte vor und erhalten ein Ergebnis. In obigem Beispiel können Sie anschließend alle Flächen aufsummieren und erhalten so den Wert der Gesamtfläche.
Mit Hilfe von Funktionen können wir ein Gesamtproblem sinnvoll in kleinere Teilprobleme zerlegen. Ein weiterer Vorteil ist, dass dadurch z.B. die Formel zur Berechnung der Kreisfläche nur einmal implementiert werden muss. Je komplexer die Aufgabenstellung ist, umso notwendiger wird die Zerlegung in kleinere Probleme

## 1.3 Die Methodik der Objektorientierung

Mit dem Anwachsen der Programmgröße und vor allem der zu verarbeitenden Datenmenge reicht die Konzentration auf Prozeduren nicht mehr aus. Der Schwerpunkt beim Entwurf von Programmen verlagert sich hin zur Organisation der Daten. Bei der prozeduralen Programmierung arbeiten die Funktionen zwar mit den Daten, sind aber prinzipiell von den Daten unabhängig. Bei der objekt-orientierten Programmierung wird die Abhängigkeit zwischen Funktionen und Daten berücksich-tigt.
Als **Modul** bezeichnet man eine Einheit von logisch zusammengehörenden Daten und Funktionen. Ziel ist es, das Programm so zu unterteilen, dass die Daten in Modulen **gekapselt** sind. (D.h. die Daten können nicht beliebig verändert werden, sondern nur nach vorgegebenen Funktionen.) Diese Modularisierung schützt z.B. davor bei der Berechnung der Kreisfläche mit negativen Radien zu arbeiten.
Module werden häufig so bestimmt, dass sie ein „Objekt“ der realen Welt beschreiben. Die Daten beschreiben dabei die Eigenschaften des Objekts und die Funktionen legen sein Verhalten fest. Solch ein Modul, das sozusagen ein Bauplan für gleichartige Objekte darstellt, bezeichnet man in der Objektorientierung als **Klasse**. Mit Hilfe der Klasse werden dann die konkreten **Objekte** erzeugt. Eine Klasse beschreibt – auf abstrakte Weise – alle gleichartigen Objekte.
Betrachten wir nochmal die Berechnung der Fläche aus Beispiel 1.1. Anstatt ausschließlich die Berechnung zu betrachten und in Teilberechnungen zu zerlegen, kann man auch die Gesamtfläche betrachten und in Teilflächen zerlegen. Jede Fläche besteht dann aus Daten, z.B. dem Radius für den Kreis oder den Seitenlängen für das Rechteck und zugehörigen Funktionen. Jede Einzelfläche ist ein Objekt. In Abbildung 1.1 existieren z.B. die Objekte „linke Hand“, „rechte Hand“, „linker Unterschenkel“, „rechtes Ohr“ und „Kopf“. Die Klasse Kreis beispielsweise dient dazu, alle Objekte (Flächen) zu beschreiben, die die Form eines Kreises haben, sie besitzt als Daten den Radius und eine Funktion „berechne Kreisfläche“. Eine **Funktion** ist damit in der Objektorientierung einer Klasse zugeordnet, sie wird auch als **Operation** oder **Methode** bezeichnet. Mit Hilfe der Klasse Kreis können dann mehrere Kreisobjekte erzeugt werden. Jedes Kreisobjekt hat einen Radius, der Wert des Radius ist aber für jedes Objekt ein anderer. Für alle Kreisobjekte kann die Funktion „berechne Kreisfläche“ verwendet werden.
Das wichtigste Konzept der Objektorientierung ist das **Klassenkonzept**. Um eine Durchgängigkeit bei der Programmentwicklung zu erreichen wird dieses Klassenkonzept in allen Phasen – Analyse, Entwurf und Implementierung – verwendet. Auch andere Konzepte der Objektorientierung werden durchgängig in allen Phasen eingesetzt. Dadurch kann z.B. eine Änderung im Entwurf leicht in der Analyse nachgetragen werden und umgekehrt.
Die Techniken zur Entwicklung guter Prozeduren, d.h. guter Algorithmen, werden jetzt auf jede Operation einer Klasse angewendet. Wo keine Gruppierung der Funktionen mit ihren zugehörigen Daten notwendig ist, reicht der prozedurale Programmierstil aus.

Bevor wir in die Programmierung einsteigen, wollen wir uns mit dem Entwurf objektorientierter Programme beschäftigen. Ende der 80er Jahre des letzten Jahrhunderts wurden die ersten Bücher über die Methodik der objektorientierten Analyse (*OOA, Object Oriented Analysis)* und des objektorientierten Entwurfs (*OOD, Object Oriented Design)* veröffentlicht.
Die **objektorientierten Grundkonzepte**, auf die später noch näher eingegangen wird, sind diejenigen Konzepte, die in allen Phasen der Softwareentwicklung – Analyse, Entwurf und Implementierung – vorhanden sind.
Dies sind:

- Objekt
- Klasse
- Attribut
- Operation
- Botschaft
- Vererbung

Eine Programmiersprache, die diese Grundkonzepte unterstützt wird als **objektorientierte Programmiersprache** bezeichnet.
Wir verwenden als **Notation** die **UML** (*Unified Modeling Language*). Sie besteht aus Grafiken (z.B. Klassendiagramm) und Texten (z.B. Spezifikationen). Die Notation ist von der Analyse zum Entwurf durchgängig. [1]
Ein einfaches Modell der Klassen Kreis und Rechteck wird in der UML folgendermaßen dargestellt:

| **Kreis** |
|---|
| Radius |
| berechneFläche() |

| **Rechteck** |
|---|
| Länge<br>Breite |
| berechneFläche() |

## 1.4 Werkzeuge

Für die Darstellung der **UML** (*Unified Modeling Language*) -Diagramme genügen Papier und Stift.
Für die Programmierung in der Programmiersprache **Java** brauchen wir ein Software Development Kit (SDK) der Java 2 Standard Edition (J2SE). Wir werden mit der Version Java SE Dev-Kit: JDK 14 oder neuer arbeiten. Sie finden das J2SE SDK auch auf

*http://www.oracle.com/technetwork/java/javase/downloads/index.html*

### *Programmübersetzung*

Java wurde, wie andere höhere Programmiersprachen, entworfen, um für den Menschen einfach lesbar zu sein. Der Programmtext, den wir formulieren, wird Quelltext genannt und muss in einer Datei mit der Endung **.java** gespeichert werden, z.B. Test.java. Ein Computer kann den Quelltext von Java nicht direkt ausführen, er benötigt einen sogenannten **Bytecode**, der für uns nur schwer zu lesen ist, von einer Maschine jedoch effizient ausgeführt werden kann. Daher benötigen wir ein spezielles Programm, das Java-Text in Bytecode übersetzt, den **Compiler**. Sobald wir den Quelltext ändern, stimmt der Bytecode nicht mehr mit unserem Quelltext überein und wir müssen den Compiler laufen lassen. Der Compiler überprüft auch, ob der Quelltext syntaktisch korrekt ist. Ist dies nicht der Fall erhalten wir eine entsprechende Fehlermeldung. Ist der Compiler ohne Fehlermeldung durchgelaufen, wird der Bytecode erzeugt und in einer Datei mit gleichem Namen wie die Quelltextdatei, aber der Endung **.class** abgespeichert. Der Compiler wird mit dem Befehl **javac** aufgerufen, z.B.

```
$ javac Test.java
```

Im Gegensatz zu vielen anderen Programmiersprachen, kann bei Java der Bytecode nicht direkt vom Prozessor des Rechners ausgeführt werden, sondern ist für einen hypothetischen Prozessor, die sogenannte **virtuelle Maschine** bestimmt. Da ein Rechner jedoch keinen hypothetischen Prozessor, sondern einen echten Prozessor besitzt, wird ein weiteres Programm verwendet, die **Java Virtual Machine** (JVM). Die JVM simuliert den hypothetischen Prozessor auf dem realen Prozessor. Die JVM wird mit dem Befehl **java** aufgerufen, danach folgt die Bytecode-Datei ohne die Endung .class, z.B.

**$ java Test**

Dieser Aufruf führt dann zur Programmausführung. Die Zeitdauer, die ein Programm zur Ausführung benötigt, wird **Laufzeit** genannt. Als Programm wird sowohl der Quelltext als auch der ausführbare Bytecode bezeichnet.
Der Einsatz der JVM hat den Vorteil, dass der übersetzte Bytecode auf jedem System verwendet werden kann und es keine Rolle spielt auf welchem System das Programm entwickelt wurde. Der Nachteil ist, dass Java-Programme langsamer sind und mehr Speicherplatz verbrauchen als Programme, die direkt auf dem Prozessor ausgeführt werden.[2]

***Entwicklungsumgebung***

Als Entwicklungsumgebung verwenden wir BlueJ. BlueJ ist eine Java-Entwicklungsumgebung, die an mehreren Universitäten in Australien und England mit dem Ziel entwickelt wurde, bei der Einführung in die objektorientierte Programmierung zu unterstützen. Sie bietet die folgenden Vorteile:

- ➔ Die Benutzungsschnittstelle ist sehr einfach.
- ➔ BlueJ zeigt ein UML-ähnliches Diagramm der Klassen und Beziehungen in einem Projekt.
- ➔ Objekte können direkt erzeugt werden, dadurch können Programme sehr modular erzeugt und getestet werden.

Wir verwenden die BlueJ-Version 4.2.2 (oder eine neuere), die Sie unter

*http://www.bluej.org/*

finden. Ein Tutorial finden Sie unter *http://www.bluej.org/doc/documentation.html.* In BlueJ finden Sie sowohl auf der Startseite als auch im Quelltext-Editor eine Schaltfläche mit der Aufschrift „Compiler“. Wenn Sie diese im Quelltext-Editor betätigen, wird die gerade geöffnete Klasse übersetzt, wenn Sie die Schaltfläche im Projektfenster betätigen, werden alle Klassen des Projekts übersetzt.

## 1.5 Literaturhinweise

***Heide Balzert: Lehrbuch der Objektmodellierung***

Analyse und Entwurf mit der UML 2
Spektrum Akademischer Verlag, ISBN 3-8274-1162-9
*guter Einstieg in den objektorientierten Entwurf, sehr umfangreiches Buch zur UML, Analyse und Entwurf in der Vorlesung lehnt sich an die ersten Kapitel dieses Buches an.*

***Chris Rupp, Stefan Queins: UML 2 glasklar** (in HSB als E-book und Print vorhanden)*

Praxiswissen für die UML-Modellierung
*Carl Hanser Verlag, Print ISBN: 978-3-446-43057-0; eISBN: 978-3-446-43197-3*

***Raul Sidnei Wazlawick: Object-Oriented Analysis and Design for Information Systems, Kap.6.4**(in HSB als e-book vorhanden)*

*Verlag Morgan Kaufmann, ISBN: 978-012-417-293-7*

***David J. Barnes, Michael Kölling: Java lernen mit BlueJ*** *(in HSB vorhanden)*
Objects first - eine Einführung in Java, 6., aktualisierte Auflage, Pearson Studium,
ISBN: 978-3-86894-911-7
*Sehr gute Einführung in die Objektorientierung und das Konzept der Objekte und Klassen, im Praktikum werden wir einige Projekte und Aufgaben aus dem Buch behandeln.*
bzw. in Englisch:

***David J. Barnes, Michael Kölling: Objects First with Java***
A practical Introduction using BlueJ
6. Auflage, Pearson
ISBN: 978-0134477367

***Michael Kölling: BlueJ Tutorial***

online: www.bluej.org/tutorial/blueJ-tutorial-deutsch.pdf

***Judy Bishop: Java lernen*** *(in HSB vorhanden)*

2. Aufl., unveränd. Nachdr., Pearson Studium Verlag
ISBN: 978-3827370853
*Rascher Einstieg in echte Programmieraufgaben, viele realistische Beispiele, Klassen und Objekte werden früh eingeführt*

***Bruce Eckel: Thinking in Java*** *(in HSB vorhanden)*

The definitive introduction to object-oriented programming in the language of the world wide web
*ISBN: 9780131872486*
*erwartet etwas Programmiererfahrung, beginnt direkt mit Objektorientierung! Ansatz: Objects first!*

***Reinhard Schiedermeier: Programmieren mit Java***

2.Auflage, Pearson Studium Verlag
ISBN: 978-3868940312
*beginnt mit Variablen und Kontrollstrukturen, Klassen und Objekte erst später*

***Cay S. Horstmann, Gary Cornell: Core Java*** *(in HSB vorhanden)*

Band 1 - Grundlagen
Prentice Hall,
*ISBN:* 3827295653
*sehr ausführlich, auch für grafische Benutzungsoberfläche*

bzw. in Englisch:

***Cay S. Horstmann, Gary Cornell: Core Java*** *(in HSB als E-Book und Print vorhanden)*
Volume 1 – Fundamentals
Prentice Hall
Print ISBN: 9780137673810 (12. Auflage);

***Dietmar Abts: Grundkurs Java (in HSB als E-Book vorhanden)***

Von den Grundlagen bis zu Datenbank- und Netzanwendungen
11. Auflage, Springer Vieweg
*ISBN: 9783658304942*

***Hanspeter Mössenböck: Sprechen Sie Java?*** *(in HSB vorhanden)*

Eine Einführung in das systematische Programmieren
Dpunkt.Verlag, 2014
ISBN: 9783864900990
*Algorithmisches Denken, systematischer Programmentwurf, Programmierstil; Objektorientierung sehr spät*

***Christian Ullenboom: Java ist auch eine Insel*** *(in HSB vorhanden)*
Einführung, Ausbildung, Praxis
16., aktualisierte Auflage, Bonn Rheinwerk Verlag
ISBN: 9783836287456 eISBN:9783836287470
*Gute Einführung.*

## 1.6 Objektorientierte Analyse

Das Ziel der Analyse ist es, die Wünsche und Anforderungen an ein neues Softwaresystem zu ermitteln und zu beschreiben. Das Ergebnis soll die Anforderungen vollständig, widerspruchsfrei, eindeutig, präzise und verständlich beschreiben. Es ist wichtig, dass bei der Modellbildung in der Analyse alle Aspekte der Implementierung ausgeklammert werden, es wird von einer perfekten Technik ausgegangen. Die Beschreibung soll aus der Sicht des Anwenders erfolgen. Die gesamten Anforderungen sind nicht plötzlich da, sondern die Analyse bildet einen kontinuierlichen Prozess, um Informationen zu sammeln, zu filtern und zu dokumentieren.
Bei der objektorientierten Analyse wird von Objekten ausgegangen, die sich in der realen Welt befinden. Dies können Gegenstände, Personen, Begriffe oder auch Ereignisse aus dem jeweiligen Anwendungsbereich sein. Aus einem realen Objekt wird durch Modellbildung und geeignete Abstraktion ein Objekt des objektorientierten Modells (und später bei der Implementierung ein Java-Objekt).
**Abstraktion** bedeutet, die Eigenschaften und Verhaltensweisen eines Objekts zu filtern, die für die jeweilige Aufgabenstellung relevant sind. Abstraktion ist aber auch die Fähigkeit, Details von Bestandteilen zu ignorieren, um den Fokus der Betrachtug auf eine höhere Ebene zu lenken. Was heißt das? Wir wollen dies an einem Beispiel verdeutlichen:
Stellen Sie sich Ingenieure vor, die ein neues Fahrzeug entwerfen sollen. Einige Ingenieure betrachten die äußere Form, die Größe und die Position des Motors, die Anzahl und Größe der Sitze, den exakten Radstand, etc. Andere Ingenieure, die den Motor entwickeln, denken an die Teile, aus denen ein Motor besteht: Zylinder, Einspritztechnik, Vergaser, Elektronik, etc. Diese Ingenieure betrachten den Motor nicht als Einheit, sondern als komplexes Gebilde aus vielen Einzelteilen. Ein weiterer Ingenieur entwirft die Zündkerzen. Er sieht die Zündkerze als ein aufwendiges Gebilde aus vielen Einzelteilen an. Der Fahrzeugdesigner **abstrahiert** von den technischen Details des Motors und ist nur an dessen Größe interessiert, um das Gesamtfahrzeug entwerfen zu können.

| |
|---|
| **Objekte**, die sich durch die gleichen Eigenschaften beschreiben lassen, gehören der gleichen **Klasse** an. |

Bei der objektorientierten Analyse wird **nicht** beschrieben, *wie* Objekte auf der Benutzungsoberfläche dargestellt werden oder *wie* sie gespeichert oder selektiert werden. Das Analyse-Modell soll die Struktur des Problems, aber noch keine technischen Lösungen beschreiben.

### *Produkte der Analysephase*

Der erste Schritt der Systemanalyse sollte darin bestehen, ein Pflichtenheft als Ausgangsbasis für eine systematische Modellbildung zu erstellen. Das **Pflichtenheft** ist eine textuelle Beschreibung dessen, was das zu realisierende System leisten soll. Das Pflichtenheft besitzt ein niedrigeres Detaillierungsniveau als das OOA-Modell. Es ist nicht das Ziel, anhand des Pflichtenheftes das System zu implementieren. Es sollte aber die wichtigsten Informationen enthalten, die zur Lösung der Aufgabe erforderlich sind.

**Pflichtenheft - Gliederungsschema**

1. **Zielbestimmung**
   Formulieren Sie Ziele und nicht die für deren Erreichung notwendigen Funktionen.
   **1.1. muss-Kriterien**
   Nennen Sie Ziele, die das Softwaresystem unbedingt erfüllen muss.
   **1.2. Kann-Kriterien**
   Nennen Sie hier diejenigen Ziele, die das Produkt zwar erfüllen sollte, auf die aber zunächst verzichtet werden kann. (dient auch der Projektplanung)
   **1.3.Abgrenzungskriterien**
   Machen Sie deutlich, welche Ziele mit dem Produkt bewußt *nicht* erreicht werden sollen.
2. **Funktionalität**
   Die Funktionalität des Systems ist auf oberster Abstraktionsebene zu beschreiben. Das bedeutet, dass die typischen Arbeitsabläufe, die mit dem System durchgeführt werden sollen, zu nennen sind. Ein Arbeitsablauf soll immer zu einem Ergebnis für den Bediener führen. Die Formulierung soll die Basis für das OOA-Modell legen, es wird keine vollständige textuelle Beschreibung der funktionalen Anforderungen verlangt.
3. **Daten**
   Die langfristig zu speichernden Daten und deren voraussichtlicher Umfang sind aus Benutzersicht aufzuführen.
4. **Benutzungsoberfläche**
   Formulieren Sie die grundlegenden Anforderungen an die Benutzungsoberfläche.

Das **OOA-Modell** (Analysemodell) bildet die fachliche Lösung des zu realisierenden Systems. Es wird daher auch Fachkonzept genannt. Es besteht aus einem statischen und einen dynamischen Modell.
Das **statische Modell** beschreibt insbesondere die Klassen des Systems, die Beziehungen zwischen den Klassen und die Vererbungsstrukturen. Außerdem enthält es die Daten des Systems (Attribute).
Das **dynamische Modell** zeigt Funktionsabläufe, beschreibt die durchzuführenden Aufgaben und zeigt, wie Objekte miteinander kommunizieren, um eine bestimmte Aufgabe zu erledigen.

Für die Erstellung des OOA-Modells ist fachliches Expertenwissen notwendig, denn nur der Fachexperte weiß, was das System leisten soll. Daher ist es auch wichtig, die zukünftigen Benutzer einzubeziehen. Es sollen hier nur die wichtigsten Konzepte und Notationselemente für die Analyse erarbeitet werden.[1]

## 1.7 Objektorientierter Entwurf

In der Analyse sind wir bei der Modellierung des Systems von einer idealen Umgebung ausgegangen. Aufgabe des **Entwurfs** ist es nun, die in der Analyse spezifizierte Anwendung unter den geforderten technischen Randbedingungen zu realisieren. Der **objektorientierte Entwurf (OOD)** wird dadurch vereinfacht, dass dieselben Konzepte und Notationen verwendet werden, wie in der Analyse. Der Entwurf muss so detailliert sein, dass jede entworfene Klasse direkt implementiert werden kann.
Viele heute „veraltete“ Systeme sind bezüglich ihrer Funktionalität noch ganz aktuell, während ihre Benutzungsoberfläche und ihre Datenhaltung veraltet sind. Daher verfolgen wir das Ziel, Fachkonzept, Benutzungsoberfläche und Datenhaltung weitgehend zu entkoppeln. Man spricht dabei auch von der **Drei-Schichten-Architektur**.

***Produkte der Entwurfsphase:***

Das **OOD-Modell** soll ein Abbild des späteren Programms sein. Jede Klasse, jedes Attribut und jede Operation des OOD-Modells kommt auch in den Programmen vor. Es werden sogar schon exakt die gleichen Namen wie im Programm verwendet. Das OOD-Modell macht vor allem das Zusammenwirken einzelner Elemente deutlich. Auch beim OOD-Modell werden ein statisches und ein dynamisches Modell erstellt.
Das **statische Modell** ist wesentlich umfangreicher als das beim OOA-Modell, es soll alle Klassen des Programms enthalten.
Das **dynamische Modell** ermöglicht eine übersichtliche Beschreibung der Kommunikation zwischen den Objekten, die anhand des Programmcodes nur schwer nachzuvollziehen ist.
Den objektorientierten Entwurf werden wir nur kurz anschneiden[1].

***Fragen:***

- ✗ Wodurch wird die gute Durchgängigkeit von der Analyse bis zur Implementierung erreicht?
- ✗ Wozu dient das Pflichtenheft?
- ✗ Wie lassen sich Analyse und Entwurf voneinander abgrenzen?
- ✗ Warum sollte in der Analyse nicht auf Implementierungsdetails eingegangen werden?
- ✗ Warum ist es sinnvoll, die fachliche Funktionalität, die Benutzungsschnittstelle und die Datenhaltung strikt zu trennen?

# 2 Objektorientierte Analyse – Basiskonzepte

***Lernziele:***

- **Verstehen**
  - Erklären können, was ein Objekt ist
  - Erklären können, was eine Klasse ist
  - Erklären können, was ein Attribut ist
  - Klassenattribut und Objektattribut unterscheiden können
  - Erklären können, was eine Operation ist
- **Anwenden**
  - Die UML für Objekt, Klasse, Attribut und Operation anwenden können
  - Objekte und Verbindungen im Text erkennen und im Objektdiagramm modellieren können
  - Klassen, Attribute und Operationen identifizieren und im Klassendiagramm modellieren können

Dieses Kapitel lehnt sich an das Buch „Lehrbuch der Objektmodellierung“ von Heide Balzert an. [3]

## 2.1 Objekt

Objekte können Dinge (z.B. Fahrrad, Messuhr), Personen (z.B. Kunde, Mitarbeiter) oder Begriffe (z.B. Programmiersprache, Schulbildung) sein. In der Objektorientierung besitzt ein **Objekt** einen bestimmten Zustand und reagiert mit einem definierten Verhalten auf seine Umgebung. Jedes Objekt besitzt eine Identität, die es von allen anderen Objekten unterscheidet. Ein Objekt kann andere Objekte kennen, man spricht von Verbindungen zwischen Objekten.
Der **Zustand** eines Objektes umfasst die Attribute bzw. deren aktuelle Werte und die Verbindungen zu anderen Objekten. **Attribute** sind unveränderliche Eigenschaften des Objekts, während die Attributwerte verändert werden können.
Das **Verhalten** eines Objekts wird durch Operationen beschrieben. Eine Änderung oder eine Abfrage des Zustands ist nur über die Operationen möglich (der direkte Zugriff auf die Attribute ist von der Außenwelt verborgen). Über den Aufruf von Operationen können Objekte miteinander kommunizieren.

In der **UML** (Unified Modeling Language) wird das Objekt als Rechteck dargestellt, das in zwei Felder aufgeteilt werden kann. Im oberen Feld wird das Objekt bezeichnet, wobei die Bezeichnung immer unterstrichen wird. Im unteren Feld werden optional die im jeweiligen Kontext erforderlichen Attribute des Objekts eingetragen.
Für die Bezeichnung gibt es die folgenden Möglichkeiten:

a) 

, wenn das Objekt über einen Namen angesprochen werden soll

b) 

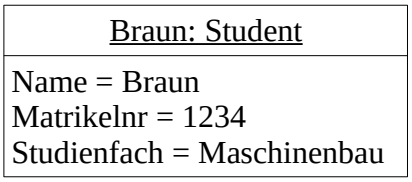
, wenn der Name der Klasse aus dem Kontext ersichtlich ist

c) 

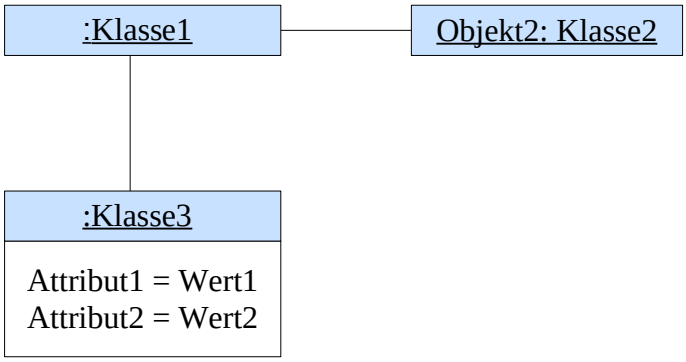
, wenn es sich um irgendein Objekt der Klasse handelt (anonymes Objekt)

Meist werden für die Attribute auch ihre Werte angegeben, in der Form
**Attribut = Wert** .
Ist der Wert des Attributes nicht von Interesse, kann der Wert auch weggelassen werden.
Die Operationen, die ein Objekt ausführen kann, werden im Objektdiagramm nicht angegeben, da sie bei der Klasse definiert werden.

Beispiel 2.1: Der Student mit dem Namen Braun hat die Matrikelnummer 1234 und studiert Maschinenbau. In der UML wird der Student Braun dann folgendermaßen dargestellt:

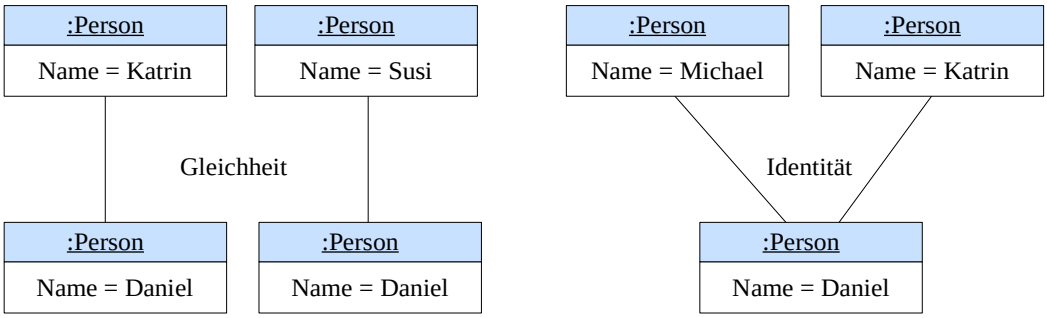



Im **Objektdiagramm** werden Objekte und ihre Verbindungen untereinander zu einem bestimmten Zeitpunkt dargestellt. Objektdiagramme sind also Momentaufnahmen des Systems, wobei häufig anonyme Objekte verwendet werden.



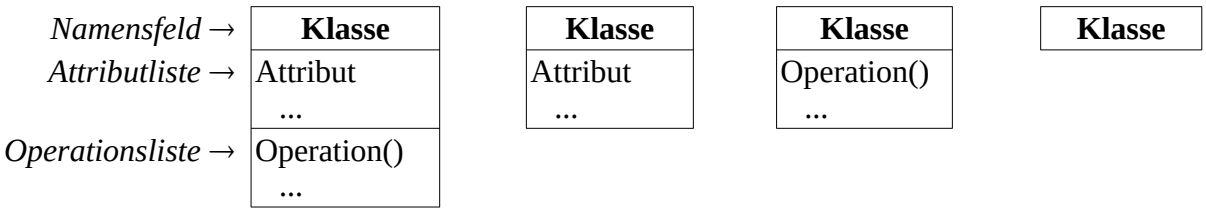

Zustand und Verhalten eines Objekts bilden eine Einheit. Die Daten eines Objekts können nur mit Hilfe der Operationen gelesen und geändert werden. Diese Eigenschaft bezeichnet man als **Geheimnisprinzip.**
Die **Objektidentität** ist die Eigenschaft, die ein Objekt von allen anderen Objekten unterscheidet, auch wenn sie zufällig identische Attributwerte besitzen. Keine zwei Objekte können dieselbe Identität besitzen. Besitzen zwei Objekte dieselben Attributwerte, so spricht man von Gleichheit der Objekte. Was entspricht dieser Identität später bei der Programmierung/im Computer?
In dem folgenden Objektdiagramm wird der Unterschied zwischen Gleichheit und Identität dargestellt. Die Personen Katrin und Susi haben beide ein Kind mit dem Namen Daniel. Da alle Personen nur das Attribut Name besitzen, sind diese beiden Objekte gleich. Michael und Katrin dagegen sind die Eltern desselben Kindes, mit dem Namen Daniel, es liegt Objektidentität vor.

Der Objektname identifiziert ein Objekt im Objektdiagramm. Im Gegensatz zur Objektidentität muss er nur innerhalb eines Diagramms eindeutig sein.
Für ein Objekt werden auch die Begriffe „Instanz“ (engl.: „class instance“) und „Exemplar“ verwendet.



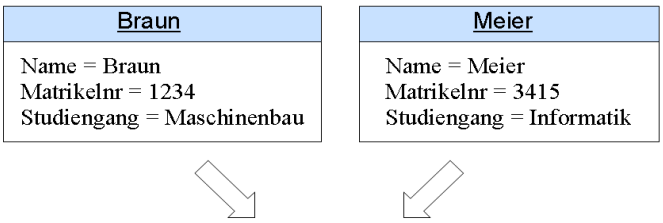

## 2.2 Klasse

Alle gleichartigen Objekte, d.h. Objekte mit den gleichen Attributen (aber im allgemeinen unterschiedlichen Attributwerten) und Operationen gehören zu der gleichen Klasse. Eine **Klasse** definiert für diese Sammlung von Objekten deren Struktur (über Attribute), das Verhalten (über Operationen) und Beziehungen (über Verbindungen zu anderen Objekten). Sie besitzt einen Mechanismus, um neue Objekte zu erzeugen. Jedes Objekt gehört zu genau einer Klasse. Eine Klasse beschreibt jedoch **nicht(!)**, welche Objekte erzeugt werden und welche Operationen aufgerufen werden.
Für die Darstellung von Klassen gibt es verschiedene Möglichkeiten:




Die Kurzformen werden verwendet, wenn die fehlenden Details unwichtig sind oder in anderen Klassendiagrammen definiert sind. Der Klassenname wird immer fett gedruckt und beginnt mit einem Großbuchstaben (Konvention). Das **Klassendiagramm** beschreibt das statische Modell des Systems. Der Klassenname muss innerhalb des gesamten Softwaresystems eindeutig sein.

Beispiel 2.2: Die beiden Studierenden in der nachfolgenden Abbildung besitzen die gleichen Attribute. Außerdem sollen sie beide an der Vorlesung teilnehmen können und das Studienfach wechseln können. Sie gehören daher beide zur gleichen Klasse. Versuchen Sie das Klassendiagramm für diese Klasse „Student“ zu zeichnen!




Jede Klasse besitzt einen Mechanismus, um Objekte zu erzeugen.
Jede Klasse soll einen ganz bestimmten Zweck innerhalb des Softwaresystems erfüllen. Um diesen

Zweck herauszustellen, erweitern wir die UML um eine **Kurzbeschreibung** der Klasse. Für die Klasse Student könnte dies folgendermaßen aussehen:

**Klasse** Student
Studierende/r, die oder der an einer Hochschule immatrikuliert ist.

Jedes Objekt „weiß“, zu welcher Klasse es gehört. Da alle Objekte einer Klasse gleiche Operationen besitzen, werden diese der Klasse zugeordnet. Da jedes Objekt seine Klasse kennt, kann es dort alle benötigten Operationen finden.
Umgekehrt „weiß“ eine Klasse nicht, welche Objekte von ihr erzeugt wurden. Da dieses Wissen jedoch häufig recht nützlich wäre und in der Analyse nicht auf Implementierungsdetails eingegangen werden soll, wird in der Analyse davon ausgegangen, dass eine Klasse ihre Objekte kennt. Diese Eigenschaft heißt **Objektverwaltung.** Die Klasse erhält dadurch die Möglichkeit auf die Menge der Objekte einer Klasse zuzugreifen. Diese Vereinfachung gilt jedoch nur in der Analyse. Im Entwurf und in der Implementierung muss diese Objektverwaltung noch realisiert werden. Unter Implementierung versteht man die Umsetzung eines Softwareentwurfs in ein Computerprogramm, d.h. die „Übersetzung“ des Entwurfs in eine konkrete Programmiersprache.

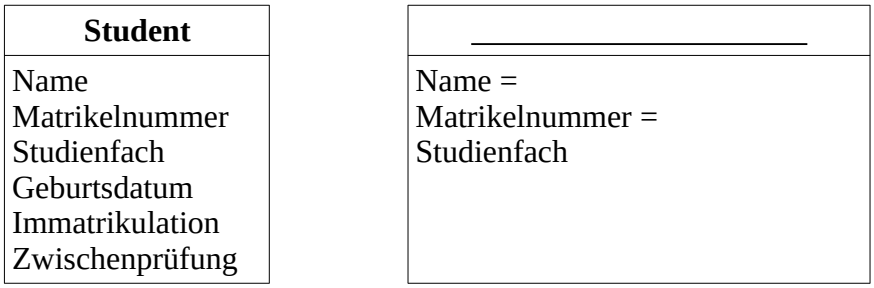
**Die Klasse ist eine Abstraktion, die Gemeinsamkeiten von Objekten beschreibt. Sie darf nicht mit der Menge aller Objekte dieser Klasse verwechselt werden!**

## 2.3 Attribut

Die **Attribute** beschreiben die Daten, die von den Objekten einer Klasse angenommen werden können. Jedes Attribut ist von einem bestimmten Typ. Alle Objekte einer Klasse besitzen dieselben Attribute, jedoch unterschiedliche Attributwerte. Dabei darf ein Feld für den Attributwert auch leer sein. D.h., dass dieses Attribut nicht bei der Erzeugung des Objekts, sondern zu einem späteren Zeitpunkt (evtl. auch nie) einen definierten Wert erhält.

Beispiel 2.3: Betrachten wir nochmal die Klasse Student, die wir noch durch einige Attribute erweitern. Ergänzen Sie das Objektdiagramm mit Ihren eigenen Daten!

Klasse: Objekt:




Was fällt hier auf?
Das Attribut Zwischenprüfung _____

Attribute werden durch ihren Namen und ihren Typ beschrieben. Zusätzlich kann ein Anfangswert angegeben werden, der festlegt, welchen Wert ein neu erzeugtes Objekt für dieses Attribut annimmt. Außerdem können die Merkmale bzw. Eigenschaften des Attributs angegeben werden. Im Analysemodell wird nur der Name des Attributs im Klassendiagramm eingetragen und seine weiteren Informationen (Merkmale, Eigenschaften) separat beschrieben, was zu der folgenden Notation führt:

| **Klasse** | Attribut:Typ = Anfangswert<br>{Merkmal1, Merkmal2,...} |
|---|---|
| Attributname | |

Der **Attributname** muss im Kontext der Klasse eindeutig sein.

Beispiel 2.4: Die Klasse Student mit Typangabe bei den Attributen

| **Student** | |
|---|---|
| Name | Name: Zeichenkette |
| Matrikelnummer | Matrikelnummer: Ganzzahl |
| Studienfach | Studienfach: Zeichenkette |
| Geburtsdatum | Geburtsdatum: Datum |
| Immatrikulation | Immatrikulation: Datum |
| Zwischenprüfung | Zwischenprüfung: Datum |

Die Attribute dürfen nur über die Operationen der zugehörigen Klassen geändert und gelesen werden (**Geheimnisprinzip**). Die Attribute sind somit für andere Klassen bzw. deren Objekte nicht sichtbar. Daher benötigt man zwei **Zugriffsoperationen**, eine zum Lesen und eine zum Schreiben. Zugriffsoperationen werden nicht ins Klassendiagramm eingetragen (siehe Kap. 2.4)

Betrachten wir nochmal die Klasse *Student*. Dort muss für die Attribute gelten:

Zwischenprüfung > _____

Solche Beziehungen zwischen Attributwerten eines Objekts, die während der Programmausführung erhalten bleiben müssen, werden **Restriktionen** genannt. Eine Restriktion ist ein Merkmal und wird daher im Klassendiagramm in {} gesetzt.
Beispiel 2.5: Für eine Klasse *Artikel* mit den Attributen *Einkaufspreis* und *Verkaufspreis* kann z.B. festgelegt werden, dass der Verkaufspreis mindestens 150% des Einkaufspreises betragen muss. Die Notation dafür ist:

| **Artikel** | {Verkaufspreis >= 1.5*Einkaufspreis} |
|---|---|
| Einkaufspreis<br>Verkaufspreis | |

Bei der Implementierung muss dann sichergestellt werden, dass nach der Änderung eines der beiden Attributwerte, die Restriktion noch erfüllt wird.

Es gibt zwei spezielle Attribute:

***Klassenattribut***

Für ein **Klassenattribut** existiert für alle Objekte einer Klasse nur ein Attributwert. Klassenattribute existieren auch dann, wenn es zu einer Klasse noch keine Objekte gibt. In der UML werden Klassenattribute unterstrichen.

Beispiel 2.6: Wir betrachten die Klasse Kreis mit den Attributen Radius und PI. Für alle Kreis-Objekte hat PI den gleichen Wert.

| **Kreis** |
| --- |
| Radius<br>PI |

***Abgeleitetes Attribut***

Für ein **abgeleitetes Attribut** kann der Wert aus anderen Attributwerten berechnet werden. Ein abgeleitetes Attribut darf von außen nicht geändert werden. Es wird mit „/“ gekennzeichnet.
Beispiel 2.7: Das Alter einer Person kann aus ihrem Geburtsdatum (und dem aktuellen Datum) berechnet werden. (Hier wird vorausgesetzt, dass auf jedem System vom Programm aus auf das aktuelle Datum zugegriffen werden kann.)

| **Person** |
| --- |
| Geburtsdatum<br>/Alter |

In der UML ist nicht festgelegt, wie der **Typ** eines Attributs definiert wird. In der Analyse dient die **Typdefinition** dazu, das Attribut aus fachlicher Sicht möglichst genau zu beschreiben. Die Bezeichnung muss für den Anwender verständlich sein und entspricht noch nicht dem Datentyp der später eingesetzten Programmiersprache. In Entwurf und Implementierung wird dann abhängig von der gewählten Programmiersprache der Typ neu definiert. **Der Typ eines Attributs kann selbst wieder eine Klasse sein**.

Attribute werden in objektorientierten Programmiersprachen auch als **Membervariablen**, in Java auch als **Datenfelder** oder **Elemente** (engl.:**field**s) bezeichnet. (Manchmal taucht auch die Bezeichnung Felder auf, was leicht zur Verwechslung mit Arrays führt.)

***Beispielaufgaben:***

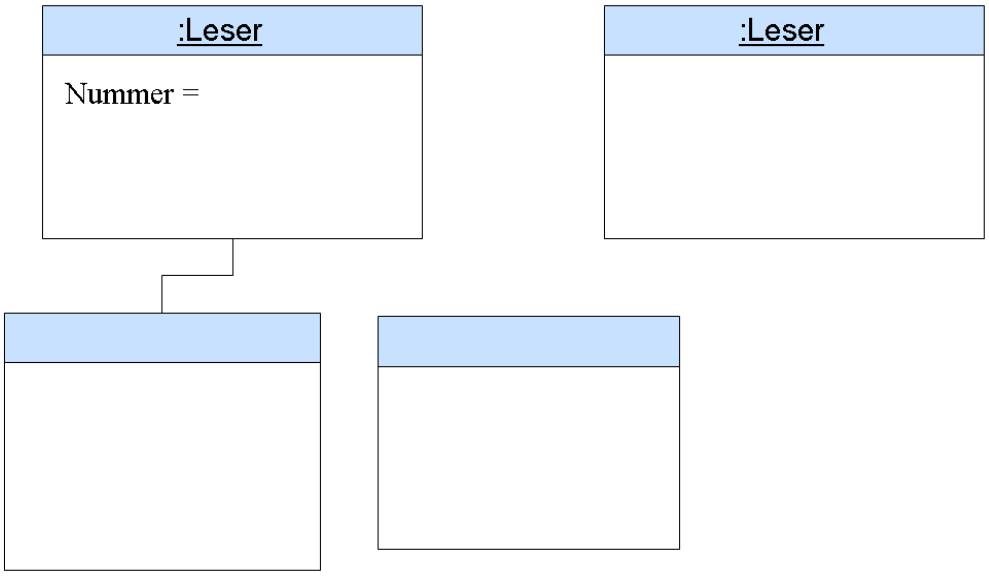
1) Ziel: Anhand einer Beschreibung Objekte und deren Verbindungen identifizieren und als Objektdiagramm darstellen.

In der Uni-Bibliothek stehen viele Bücher, beispielsweise

- Heide Balzert, Lehrbuch der Objektmodellierung, 1999
- Judy Bishop, Java lernen, 2001
- Joseph Schmuller, Jetzt lerne ich UML, 2000

Für jeden Leser werden Name, Adresse und Geburtsdatum gespeichert. Außerdem erhält jeder Leser eine Nummer. Hans Müller, geb. am 1.3. 1985 und wohnhaft in Rottweil leiht sich das „Lehrbuch der Objektmodellierung aus“. Spätestens am 15.4.2006 muss er es zurückgeben. Dieses Rückgabedatum wird ins Buch eingetragen. Inge Frels aus Villingen, geb. am 15.7.1982 leiht sich „Java lernen“ und „Jetzt lerne ich UML“ aus. Beide Bücher muss sie am 21.4.2006 zurückgeben.

Lösung:




2) Ziel: Klassenattribute und Objektattribute unterscheiden können

Die Klasse Videofilm soll mit ihren Attributen grafisch dargestellt werden. Achten Sie auf evtl. vorhandene Klassenattribute.
Für Videofilme sollen die folgenden Informationen festgehalten werden: Titel des Films, Laufzeit, Erscheinungsjahr. Jeder Film besitzt eine individuelle Ausleihgebühr. Wird ein Film beschädigt zurückgegeben, so ist eine (für alle Filme gleiche) Entschädigungsgebühr zu entrichten.
Lösung:


## 2.4 Operation

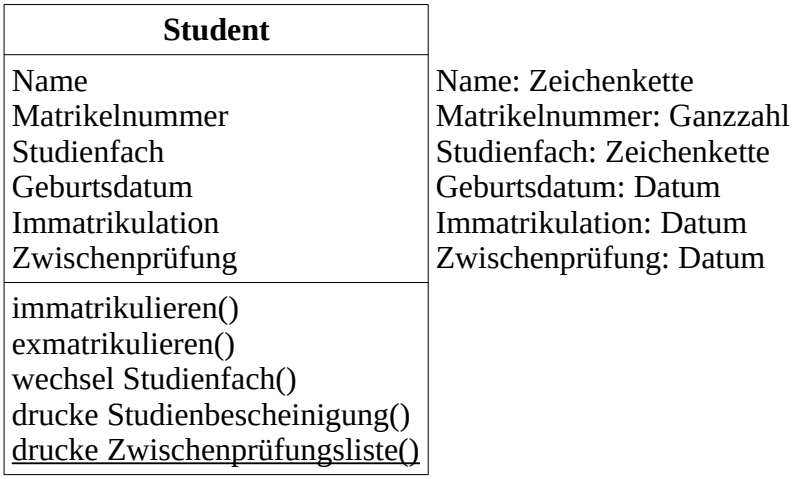
Eine **Operation** einer Klasse ist eine ausführbare Funktion, die auf alle Attribute (Daten) eines Objekts dieser Klasse direkt zugreifen kann. Alle Objekte einer Klasse verwenden dieselben Operationen. Die Menge aller Operationen wird als das Verhalten der Klasse oder als die Schnittstelle der Klasse bezeichnet.
Die Operationen werden unterhalb der Attribute in das Klassendiagramm eingetragen:

| **Klasse** |
|---|
| Attribut1<br>Attribut2 |
| Operation1()<br>Operation2()<br>Klassenoperation() |

Wir unterscheiden drei Arten von Operationen:

- Objektoperationen
- Konstruktoroperationen
- Klassenoperationen

**Objektoperationen,** kurz Operationen genannt, werden immer auf ein einzelnes, bereits existierendes Objekt angewendet. Operationen, wie *wechsel Studienfach* oder *drucke Studienbescheinigung* der Klasse *Student* sind solche Objektoperationen (siehe Beispiel 2.8)
Eine **Konstruktoroperation** erzeugt ein neues Objekt und führt entsprechende Initialisierungen (Anfangswerte von Attributen setzen) durch. Konstruktoroperationen werden im Klassendiagramm nicht dargestellt.
Beispiel 2.8: Die Klasse *Student* mit einigen Operationen wird in der UML folgendermaßen dargestellt:




Eine **Klassenoperation** ist der jeweiligen Klasse zugeordnet und kann nicht auf ein einzelnes Objekt angewendet werden. Sie wird im Klassendiagramm unterstrichen dargestellt. In der objektorientierten Analyse kann eine Klassenoperation in zwei Fällen verwendet werden:

1. Die Operation bezieht sich auf mehrere Objekte der Klasse. Dabei wird die Eigenschaft der Objektverwaltung ausgenutzt. Dies ist nur in der Analyse, nicht im Entwurf möglich.
   Die Operation *drucke Zwischenprüfungsliste* der Klasse *Student* aus Beispiel 2.8 ist eine solche Objektverwaltungsoperation.
2. Wenn ein Klassenattribut ohne Beteiligung eines Objekts geändert werden soll.

Beispiel 2.9:


Jede Aushilfe bekommt den gleichen Stundenlohn. Mit der Operation *erhöhe Stundenlohn* kann der Stundenlohn für alle erhöht werden, d.h. diese Operation bezieht sich nicht auf ein spezielles Objekt, sondern wirkt auf alle Objekte gleich.

Der Name einer Operation muss innerhalb der Klasse eindeutig sein. Außerhalb der Klasse wird die Operation mit **Klasse.Operation()** bezeichnet. Da der Operationsname ausdrücken soll, was die Operation „tut“, enthält er im allgemeinen ein Verb.
Für jede Operation sollte eine Beschreibung erstellt werden, die ihre Funktionsweise umgangssprachlich erklärt. Falls die Funktionsweise aus dem Namen der Funktion schon hervorgeht, kann auf die Beschreibung auch verzichtet werden.

Die Beschreibung sollte leicht erstellbar und leicht lesbar sein. Es sollten die Eingabedaten, die Rückgabedaten und die Wirkung aus Benutzersicht beschrieben werden (nicht die Umsetzung). Für den Anwender ist nur wichtig zu wissen: ***Was tut die Operation? - Welche Daten braucht die Operation? - Welches Ergebnis liefert die Operation?***

Unwichtig ist, wie sie es tut. Überlegen Sie als Beispiel, was Sie interessiert, wenn Sie die Lösung einer quadratischen Gleichung wissen möchten.

Für Operationsbeschreibungen verwenden wir die folgende Form:

**Name:**
**Eingabedaten:**
**Rückgabedaten:**
**Funktionsweise:**

Beispiel 2.10: Für die Operation *wechsel Studienfach()* der Klasse *Student* aus Beispiel 2.8 sieht die Operationsbeschreibung folgendermaßen aus:

**Name:** *wechsel Studienfach*
**Eingabedaten:** neues Studienfach
**Rückgabedaten:** keine
**Funktionsweise:** setzt das Attribut *Studienfach* auf den Wert des neuen Studienfachs.

**Verwaltungsoperationen** sind elementare Basisoperationen, die fast jede Klasse benötigt. Dazu gehören alle Operationen zum Lesen (*getAttribut()*) und Schreiben (*setAttribut()*) eines Attributwertes. Diese Operationen werden aus Gründen der Lesbarkeit in der Regel nicht in das Klassendiagramm eingetragen.[3]

***Fragen:***

- ✗ Welche Grundkonzepte der Objektorientierung wurden in diesem Kapitel eingeführt?
- ✗ Was ist der Unterschied zwischen einer Objektidentität und einem Objektnamen?
- ✗ Welche Eigenschaft hat ein Klassenattribut?
- ✗ Was ist ein abgeleitetes Attribut?
- ✗ Wofür verwendet man eine Klassenoperation?
- ✗ Welche Informationen werden in der Operationsbeschreibung angegeben?

Beispielaufgabe:

3) Ziel: Anhand einer Beschreibung Attribute und Operationen einer Klasse erkennen und im Klassendiagramm korrekt darstellen können

Wir wollen einen einfachen Ticketautomaten modellieren, wie er auf Bahnhöfen oder in S-Bahnen zu finden ist. Da wir noch am Einstieg in die Welt der Software-Entwicklung sind, soll unser Modell bewusst einfach gehalten werden. Wir treffen folgende Annahmen:

- Der Kunde kann Geld einwerfen.
- Es gibt nur ein Ticket, nicht wie in Realität eine Auswahl aus mehreren Tickets. Der Preis dieses Tickets kann jedoch geändert werden.
- Der Automat merkt sich, wie viel der Kunde schon bezahlt hat und wie viel Geld er insgesamt eingenommen hat.
- Der Kunde kann sich anzeigen lassen, wie viel Geld er bisher eingeworfen hat und er kann sich ein Ticket drucken lassen.
- Der Automat kann geleert werden.

Versuchen Sie anhand der Beschreibung ein Klassendiagramm mit Attributen und Operationen zu erstellen. Überlegen Sie anschließend, ob Ihnen noch weitere Operationen einfallen, die für einen solchen Ticketautomaten nützlich wären.

# 3 Klassen und Objekte in Java, Grundlagen

***Lernziele:***

- **Wissen**
  - Wissen, wie eine Klasse in Java deklariert wird
  - Wissen, welche Sichtbarkeiten es im Entwurf und in Java für Attribute gibt
  - Wissen, wie Variablen deklariert werden
  - Wissen, welche Datentypen es in Java gibt
  - Wissen, wie eine Konstante in Java deklariert wird
  - Konstruktoren kennen und wissen, wozu sie verwendet werden
  - Wissen, was eine gute Klassendokumentation enthalten sollte
- **Verstehen**
  - Verstehen, wie Operationen aufgebaut sind und wie ihr Aufruf funktioniert
  - Den Unterschied in der Speicherverwaltung von primitiven Variablen und solchen vom Klassentyp erklären können
  - Verstehen, wie mehrere Konstruktoren für die gleiche Klasse programmiert werden können
  - Verstehen, wie die Bildschirmausgabe in Java funktioniert
- **Anwenden**
  - Klassen mit Attributen und Operationen implementieren können
  - Operationen mit Parametern korrekt implementieren und anwenden können
  - Eine eigene Klassendokumentation erstellen können

## 3.1 Klassen in Java deklarieren

Die **Klasse** wird in Java durch das Schlüsselwort **class** angezeigt. Sie erlaubt die gleichzeitige Deklaration (Vereinbarung) von Attributen und Operationen.
Beispiel 3.1:
Es soll die Klasse *Ticketautomat* ([4]), die wir in Beispielaufgabe 3) bei der objektorientierten Analyse entworfen haben, implementiert werden. Wir betrachten dazu zunächst das vereinfachte Klassendiagramm:

| **Ticketautomat** | |
|---|---|
| preis<br>bishergezahlt<br>gesamtsumme | preis: Ganzzahl<br>bishergezahlt: Ganzzahl<br>gesamtsumme: Ganzzahl |
| setzePreis() | |

Ohne Attribute und Operationen sieht die Klasse *Ticketautomat* in Java folgendermaßen aus:

```
class Ticketautomat
{ // Anfang Klasse
} // Ende Klasse
```

Das Klammerpaar markiert Anfang und Ende der Klasse. Alle Anweisungen, die zur Klasse gehören, müssen zwischen diesen beiden Klammern stehen.
Als nächstes sollen die Attribute des UML-Klassendiagramms hinzugefügt werden. Attribute sind spezielle Variablen. Um zu wissen, wie Attribute deklariert werden, betrachten wir zunächst ganz allgemein die Deklaration von Variablen in Java.

## 3.2 Variablen in Java

Variablen haben die Aufgabe Daten bzw. Informationen zu speichern. Da es unterschiedliche Arten von Informationen gibt, gibt es auch unterschiedliche Variablen. Zur Unterscheidung wird jeder Variablen ein Datentyp zugeordnet. Um eine Variable im Programm ansprechen zu können, benötigt sie noch einen eindeutigen Namen. Die **Deklaration** einer Variablen muss mit einem Semikolon abgeschlossen werden und hat die Form:

```
datentyp variablenname;
```

Neben der Information welche Art von Daten in der Variablen gespeichert werden dürfen, legt der Datentyp auch fest, wie viel Speicherplatz für diese Variable reserviert wird. Jede Variablendeklaration erzeugt Speicherplatz der unten angegebenen Größe. In Java unterscheidet man zwischen **einfachen Datentypen** (auch **primitive Datentypen** genannt) und Klassen. D.h. die Deklaration einer Klasse, die sie im vorangegangenen Kapitel kennen gelernt haben, definiert einen neuen Datentyp. In der folgenden Übersicht sind alle primitive Datentypen von Java angegeben:

| **Datentyp** | **Beschreibung** | **Größe** |
|---|---|---|
| byte | 8-Bit-Zahlen (-128..127) | 8 Bit |
| short | ganze Zahl (-32 768.. 32 767) | 16 Bit |
| int | ganze Zahl (-2 147 483 648 .. 2 147 483 647) | 32 Bit |
| long | ganze Zahl (für sehr große Zahlen) | 64 Bit |
| float | Gleitpunktzahl ( $1{,}2\cdot10^{-38}..3{,}4\cdot10^{38}$) | 32 Bit |
| double | Gleitpunktzahl ($2{,}2\cdot10^{-308}..1{,}8\cdot10^{308}$) | 64 Bit |
| boolean | Wahrheitswert (wahr, falsch) | 8 Bit |
| char | Zeichen, Buchstabe (16-Bit-Unicode-Zeichen) | 16 Bit |
| void | Typangabe für keinen Typ (bei Methoden) | |

Die am häufigsten verwendeten primitiven Datentypen sind **int, double, boolean** und **char**.
Eine Variable, deren Datentyp eine Klasse ist wird als **Objektvariable** bezeichnet und ist die Umsetzung eines Objekts des objektorientierten Modells. Der Datentyp für Zeichenketten **String** spielt in Java eine Sonderrolle. Er ist eine Klasse, kann aber häufig wie ein einfacher Datentyp verwendet werden. Wir werden später noch genauer auf diesen Datentyp eingehen.
Wenn mehrere Variablen des gleichen Datentyps deklariert werden sollen, muss der Datentyp nur einmal angegeben werden, die Variablennamen müssen dann durch Kommata voneinander getrennt werden.
Beispiele für Variablendeklarationen sind die folgenden:

```
int anzahl;
double x,y,z3;
char buchstabe;
boolean kapiert;
String text_2;
Ticketautomat automat1;
```

Variablennamen müssen mit einem Buchstaben beginnen und dürfen ansonsten Buchstaben, Ziffern und den Unterstrich "_" enthalten. Dabei wird Groß- und Kleinschreibung beachtet, d.h. „timo",

„tiMO“ und „Timo“ bezeichnen drei verschiedene Variablen.
Nun haben wir unterschiedliche Variablen kennen gelernt. Wie können nun Werte in den Variablen gespeichert werden?
Dies geschieht über sogenannte **Wertzuweisungen**. Eine Zuweisung besteht aus dem Namen der Variablen, in die der Wert gespeichert werden soll, einem „=“-Zeichen als Zuweisungsoperator und dem Wert der gespeichert werden soll, z.B.

```
anzahl = 8;
x = 3.5;
buchstabe = 'd';
text = "Datentypen";
```

Die Zuweisung erfolgt immer von rechts nach links. Wert und Variable müssen dabei vom gleichen Datentyp sein („anzahl=12.7“ ist somit nicht erlaubt). Einzelne Zeichen werden in einfache Hochkommata `''` eingefasst, Zeichenketten in `" "`.
Auf der rechten Seite darf anstelle eines Wertes auch eine Variable vom gleichen Datentyp stehen. Dann wird der Wert der Variablen auf der rechten Seite in der Variablen auf der linken Seite gespeichert, z.B.

```
z3 = x;
```

In den Variablen z3 und x ist nun der gleiche Wert gespeichert, in unserem Beispiel die Zahl 3.5.
Einer Variablen darf erst ein Wert zugewiesen werden, wenn sie zuvor deklariert wurde.

#### *Unveränderliche Variablen - Konstanten*

Möchte man einer Variablen einen Wert zuweisen, der nicht mehr verändert werden darf, so stellt man ihrer Deklaration das Schlüsselwort **final** voraus, z.B.

```
final int LAENGE = 25;
```

Einer solchen Konstanten darf nur einmal ein Wert zugewiesen werden, jede weitere Wertzuweisung wird vom Compiler als Fehler abgelehnt. Eine weit verbreitete Konvention ist, Variablennamen von Konstanten in Großbuchstaben zu schreiben.

#### *Bedeutung der Variablen*

Die Stelle im Quellcode, an der eine Variable deklariert wird, legt sowohl die Bedeutung als auch die Nutzungsmöglichkeiten für diese Variable fest. Man unterscheidet dabei zwischen **lokalen Variablen**, die innerhalb einer Operation definiert werden und auch nur dort verwendet werden können, **Parametern**, die zur Übergabe von Daten an Operationen eingesetzt werden und **Attributen**. Auf diese unterschiedlichen Variablenarten werden wir in den folgenden Kapiteln noch genauer eingehen.

## 3.3 Attribute

Attribute sind Variablen, die über ihre Werte den Zustand der Objekte beschreiben. Sie gehören damit in die Klassendefinition zwischen die geschweiften Klammern. Zu den allgemein beschriebenen Datentypen im Klassendiagramm müssen nun passende Java-Datentypen ausgewählt werden. Um das in Kapitel 2.3 eingeführte Geheimnisprinzip zu erfüllen, muss der Deklaration der Attribute noch die **Sichtbarkeit** hinzugefügt werden.
Durch die Sichtbarkeit einer Variablen wird der Bereich innerhalb des Quelltextes definiert, indem eine Variable zugreifbar ist (= Gültigkeitsbereich der Variablen). Wir betrachten zunächst zwei unterschiedliche Sichtbarkeiten für Attribute:

| **Sichtbarkeit** | **UML** | **Java** |
|---|---|---|
| sichtbar nur innerhalb der Klasse | - | **private** |
| sichtbar für alle anderen Klassen | + | **public** |

Attribute sollten grundsätzlich als *private* vereinbart werden, bei einem *public*-Attribut wird das Geheimnisprinzip verletzt. Wird keine Sichtbarkeit angegeben, so entspricht dies der Sichtbarkeit *public* (solange wir nicht mit Paketen arbeiten).
Die Sichtbarkeit wird in der UML vor das jeweilige Attribut geschrieben, in Java vor den Datentyp.

Die Notation in der UML ist folgendermaßen

| **Klassenname** |
|---|
| - privateAttribut<br>+ publicAttribut |

Das Klassendiagramm aus Beispiel 3.1 sieht ergänzt durch die Sichtbarkeiten folgendermaßen aus:
Klassendiagramm 3.1:

| **Ticketautomat** | |
|---|---|
| - preis<br>- bishergezahlt<br>- gesamtsumme | preis: Ganzzahl<br>bishergezahlt: Ganzzahl<br>gesamtsumme: Ganzzahl |
| + setzePreis() | |

Die Klasse *Ticketautomat* durch die Attribute aus dem Klassendiagramm ergänzt, hat dann den folgenden Java-Quellcode[4]

```
public class Ticketautomat
{
    private int preis;
    private int bisherGezahlt;
    private int gesamtsumme;
} // Ende Klasse
```

**Attribute** werden auch **Datenfelder** oder **Membervariablen** genannt.

## 3.4 Operationen

**Operationen** werden in Java auch **Methoden** genannt. Es sind Funktionen, die mit den Daten der Klasse arbeiten. Die Operationen werden innerhalb der Klasse deklariert. Wir sehen uns nun einige Operationen an, bevor wir den allgemeinen Aufbau einer Operation beschreiben. Schon in der Analyse hatten wir gesehen, dass wir für die Attribute sogenannte Verwaltungsoperationen benötigen, wenn das Geheimnisprinzip erfüllt ist. Verwaltungsoperationen werden aus Gründen der Übersichtlichkeit nicht in das Klassendiagramm eingetragen. Sie müssen dennoch programmiert werden, da sonst auf die als „private“ deklarierten Attribute von außerhalb der Klasse nicht zugegriffen werden kann.Wir betrachten nochmal das Klassendiagramm 3.1 für einen Fahrkartenautomaten und wollen für das Attribut *preis* zwei Verwaltungsoperationen programmieren. Die Verwaltungsoperationen bekommen die Namen *setzePreis* und *gibPreis*.

```
public class Ticketautomat
{
    private int preis;
    private int bisherGezahlt;
    private int gesamtsumme;
    public void setzePreis(int neupreis)
    {
        preis = neupreis;
    }
    public int gibPreis()
    {
        return preis;
    }
} // Ende Klasse Ticketautomat
```

Vergleichen Sie den Aufbau der Java-Operationen mit der Operationsbeschreibung aus Kap. 2.4
Wo finden Sie Name, Eingabedaten, Rückgabedaten und Funktionsweise?

Die erste Operation dient dazu den Wert des Attributs *preis* zu verändern. Dazu benötigt die Operation die Information, auf welchen Wert der Preis geändert werden soll. Diese Information wird in der Variablen *neupreis* gespeichert und an die Methode (und damit auch an das Objekt, für das die Operation aufgerufen wird) übergeben.
Variablen, die Informationen an die Methode übergeben sollen, heißen **Parameter** und stehen immer in Klammern hinter dem Methodennamen. Jeder Parameter muss in der Klammer mit Datentyp angegeben werden, d.h. an dieser Stelle wird der Parameter deklariert.
Innerhalb der Methode steht lediglich eine Zuweisung, die dafür sorgt, dass der Wert der Variablen *neupreis* in dem Attribut *preis* gespeichert wird.
Operationen können nicht nur Informationen von außen erhalten, sie können auch Informationen nach außen geben. Das passiert in der zweiten Operation.
Als Ergebnis der Operation *gibPreis* erhalten wir den Wert des Attributs *preis*. Die Anweisung *return preis* bedeutet: Das Ergebnis der Methode ist der Wert der Variablen *preis.* Der Datentyp, der vor dem Methodennamen steht, gibt an welchen Datentyp das Ergebnis hat.
Methoden wie *gibPreis* werden auch **sondierende Methoden** oder **Get-Methoden** genannt, sie liefern Informationen über den Zustand des Objekts. Methoden wie *setzePreis* werden als **verändernde Methoden** oder **Set-Methoden** bezeichnet.

Wir können zwar mit Hilfe des Objektinspektors von BlueJ die Attributwerte anschauen, generell sind sie jedoch aufgrund des Geheimnisprinzips nach außen verborgen. Dies ist lediglich eine Hilfe der Entwicklungsumgebung BlueJ für Programmieranfänger.

Eine Methode oder Operation in Java besteht aus der
- **Signatur**, auch Funktionskopf genannt, und aus
- dem **Funktionsrumpf**, der Variablendeklarationen und Anweisungen enthält und immer in geschweifte Klammern gesetzt wird.

Die Signatur enthält alle Informationen, die für einen Aufruf der Methode benötigt werden und besteht aus
- der **Sichtbarkeit**, die auch weggelassen werden kann
- dem **Rückgabetyp,**
- dem **Methodennamen**, mit Hilfe dessen die Methode aufgerufen werden kann und aus
- einer **Parameterliste**, die auch leer sein kann.

Die Methode hat dann allgemein die Form

**Sichtbarkeit Rückgabetyp Methodenname ( Parameterliste )**
**{ Funktionsrumpf }**

Gibt eine Methode nichts zurück, so lautet der Rückgabetyp *void.*
Die Parameter in der Parameterliste werden durch Kommata voneinander getrennt, zu jedem Parameter muss der Datentyp angegeben werden.
**Achtung: Am Ende der Signatur steht kein Semikolon!**

Woher nimmt die Operation die Daten, mit denen sie arbeitet?
- Da sie Bestandteil der Klassendefinition ist, kann sie auf alle **Attribute** der Klasse zugreifen.
- Sie kann eigene **lokale Variablen** definieren, die dann nur in dieser Operation verwendet werden dürfen.
- Es können **Argumente** übergeben werden. Dies dient der Eingabe von Daten aus anderen Operationen und Klassen an die aktuelle Operation. Innerhalb der Klammern der Methodendeklaration werden Variablen definiert, die **Parameter**. Bei Aufruf der Methode werden diesen Parametern Werte übergeben (Argumente), die innerhalb der Operation wie lokale Variablen verwendet werden. Man spricht häufig auch von **aktuellen Parametern** statt Argumenten und von **formalen Parametern** statt Parametern.
- Mit der **Rückgabeanweisung** kann die Operation Daten nach außen geben. Daher benötigt jede Operation einen Rückgabetyp. Die Rückgabeanweisung lautet *return ...*

Bitte beachten Sie, dass Parameter eigenständige Variablen sind und daher andere Namen haben müssen als die Attribute! Ausnahme: siehe Kap. 5.2

## 3.5 Objekte

Die Klasse *Ticketautomat* stellt einen neuen Datentyp dar. **Jede Klasse definiert einen Datentyp.** Sie ist eine Beschreibung für einen Fahrscheinautomaten. Um mit ihr arbeiten zu können, müssen Objekte der Klasse *Ticketautomat* gebildet werden. Das Objekt *automat* der Klasse *Ticketautomat* ist somit in Java eine Variable des neuen Datentyps *Ticketautomat*. Wir nehmen Bezug auf den Quellext auf S. 26

**Objekte müssen mit dem Operator new gebildet werden**:
Die Deklaration besteht bei allen Objektvariablen aus zwei Teilen:

```
Ticketautomat automat;
automat = new Ticketautomat();
```

Die Deklaration kann auch zusammengefasst in einer Zeile geschrieben werden:

```
Ticketautomat automat = new Ticketautomat();
```

Damit mit der Variablen *automat* gearbeitet werden kann, müssen den Attributen noch Werte zugewiesen werden. Da wir das Geheimnisprinzip erfüllt haben, kann dies jedoch nicht direkt, sondern nur über entsprechende Operationen erfolgen. Die Operationen müssen Bestandteil der Klassendefinition sein und werden über den **Punkt-Operator** angesprochen, z.B.

```
automat.setzePreis(340);
```

Möchte man eine Operation aufrufen, die ein Ergebnis also einen Rückgabetyp besitzt, so sollte man das Ergebnis in einer Variablen mit entsprechendem Typ abspeichern, zum Beispiel

```
int ticketpreis;
ticketpreis = automat.gibPreis();
```

Beachten Sie den Unterschied zwischen der Klassendefinition und der Verwendung der Klasse zum Erzeugen von Objekten. Sie sollten für jede Klasse eine eigene Datei verwenden (BlueJ macht dies in der Regel automatisch). In BlueJ können Objekte auf der Objektleiste erzeugt werden (siehe: Kölling: Das BlueJ-Tutorial). Dies ist sehr hilfreich zum Testen einzelner Klassen und ist eine spezielle Funktionalität der Entwicklungsumgebung zur Erleichterung des Einstiegs in die Programmierung. Ansonsten können Objekte in jeder Operation einer beliebigen Klasse erzeugt werden. Als Beispiel sehen Sie nachfolgend die Klasse Ticketautomat und die Klasse Anwendung mit der Operation *starte()*.

```
public class Ticketautomat
{
  private int preis;
  private int bisherGezahlt;
  private int gesamtsumme;

  public void setzePreis(int p)
  {
      preis = p;
  }

  public int gibPreis()
  {
      return preis;
   }
} // Ende Klasse Ticketautomat
```

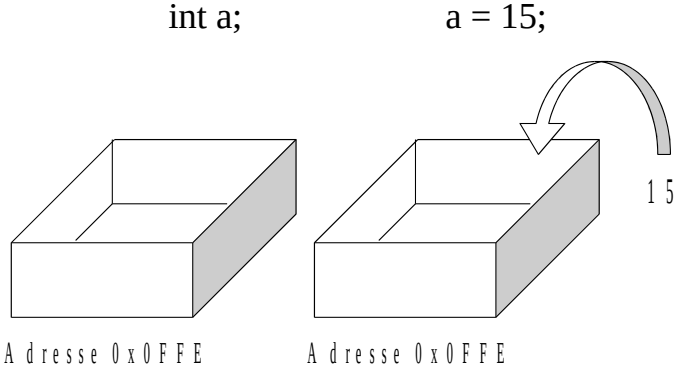
```
public class Anwendung
{
  public void starte()
  {
    // Objekt deklarieren
    Ticketautomat automat;
    // Objekt erzeugen mit new
    automat = new Ticketautomat();
    // Methode aufrufen,
    automat.setzePreis(340);
    int ticketp;
    // Preis auslesen
    ticketp = automat.gibPreis();
  } // Ende Operation starte

} // Ende Klasse Anwendung
```

Die Deklaration einer Objektvariablen ist also etwas umfangreicher als die Deklaration einer Variablen vom einfachen Datentyp. Für jeden einfachen Datentyp kann der Speicherbedarf genau angegeben werden, wie sieht das nun bei einer Klasse aus? Im folgenden schauen wir uns den Unterschied bezogen auf die Speicherverwaltung an.

Im Fall

```
int a;
a = 15;
```

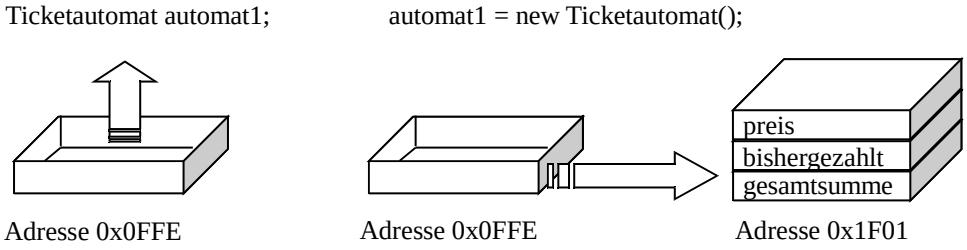
ordnet der Compiler dem Namen *a* einen bestimmten Speicherbereich zu (in diesem Beispiel der Größe 32 bit). Nach der **Initialisierung** (Wertzuweisung direkt bei bzw. nach der Definition der Variablen) mit dem Wert 15, wird der Wert 15 direkt in dem Speicherbereich der Variable abgelegt.




Bei der Deklaration

```
Ticketautomat automat1;
automat1 = new Ticketautomat();
```

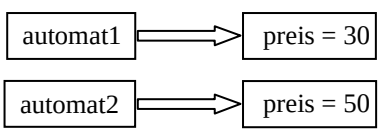
ordnet der Compiler dem Namen *automat1* auch einen bestimmten Speicherbereich zu. Dort wird aber nicht einfach ein Wert abgelegt. Bei dem Aufruf *new Ticketautomat()* wird ein Objekt der Klasse *Ticketautomat* gebildet und ihm ein separater Speicherbereich zugeordnet. Das Objekt existiert unabhängig von der Variablen *automat1* irgendwo im Speicher. Bei der Zuweisung *automat1 = ...* wird nicht der Inhalt des Objektes in den Speicherbereich der Variablen *automat1* kopiert, sondern lediglich die Adresse des Speicherbereichs des Objektes.




Man spricht zwar von dem Objekt *automat1*, in Wirklichkeit ist *automat1* eine **Referenz,** d.h. die Variable *automat1* enthält lediglich die Speicheradresse des Objektes *automat1*.
**Alle Variablen vom Typ einer Klasse sind in Java Referenzen!**

Dies wird besonders deutlich bei Zuweisungen. Es werden zwei Fahrscheinautomaten deklariert:

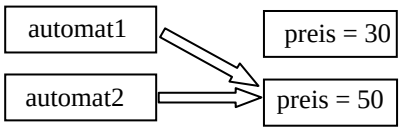
```
Ticketautomat automat1 = new Ticketautomat();
automat1.setzePreis(30);
Ticketautomat automat2 = new Ticketautomat();
automat2.setzePreis(50);
```




Durch die Zuweisung

```
automat1 = automat2;
```

wird nicht der Inhalt des Objekts *automat2* nach *automat1* kopiert, sondern die Variable *automat1* erhält die Adresse des Speicherbereichs, die auch in *automat2* abgelegt ist.




Das Objekt, welches ursprünglich von der Variablen *automat1* referenziert wurde, d.h. der zugehörige Speicherbereich ist zwar noch vorhanden jedoch nicht mehr verfügbar. Java besitzt einen

Mechanismus, mit dem der Speicherplatz automatisch freigegeben wird, wenn er nicht mehr benötigt wird.
Soll der Inhalt von *automat2* nach *automat1* kopiert werden, müssen die Werte aller Attribute kopiert werden. Die Daten liegen dann zweimal im Speicher, die Daten von *automat1* wurden überschrieben. Überlegen Sie, wie Sie das umsetzen können!

## 3.6 Konstruktoren

Konstruktoren sind spezielle Operationen. Sie haben den gleichen Namen wie die Klasse und dienen der Initialisierung eines Objekts. Die Anfangswerte für die Attribute können dem Konstruktor als Argumente übergeben werden oder direkt im Rumpf festgelegt werden. Der Konstruktor fasst sozusagen den Aufruf aller Set-Methoden in einer Operation zusammen. Bei der Programmierung eines Konstruktors ist es wichtig darauf zu achten, dass allen Attributen sinnvolle Werte zugewiesen werden.

Für die Klasse *Ticketautomat* aus Beispiel 3.1 kann ein Konstruktor folgendermaßen aussehen:

```
public Ticketautomat(int ticketpreis)
 {
     preis = ticketpreis;
     bisherGezahlt = 0;
     gesamtsumme = 0;
 }
```

Es wird davon ausgegangen, wenn ein Fahrkartenautomat neu aufgestellt wird, wurde noch nichts eingezahlt.
Die Deklaration und Initialisierung des Objektes *automat* aus Kapitel 3.5 kann jetzt in einer Zeile geschrieben werden:

```
Ticketautomat automat1 = new Ticketautomat(340);
```

Die Bezeichnung *new* wird immer zusammen mit einem Konstruktor verwendet. In Java kann kein Objekt einer Klasse erzeugt werden, ohne die Attribute zu **initialisieren**.
In Kapitel. 3.5 hatten wir jedoch einen Konstruktor aufgerufen, ohne ihn zuvor programmiert zu haben. In Java gilt die folgende Regel:

Jede Klasse braucht mindestens einen Konstruktor. Wird kein Konstruktor implementiert, weist der Compiler der Klasse einen **Standardkonstruktor** zu. Dieser Standardkonstruktor setzt alle Attribute auf ihre Standardwerte, d.h. numerische Daten auf den Wert 0, Objektvariablen auf den Wert *null* (sie besitzen keine Referenz) und alle booleschen Werte auf *false*. Da diese Werte häufig für die Anwendung nicht sinnvoll sind, sollten Sie für jede Klasse, die Sie programmieren, mindestens einen Konstruktor schreiben, der alle Attribute initialisiert. Sie können auch selber einen Standardkonstruktor programmieren und festlegen, welche Standardwerte die Attribute bekommen sollen.

Allgemein ist ein **Standardkonstruktor** ein Konstruktor ohne Argumente.

Er wird nur dann automatisch zugewiesen, wenn Sie keinen eigenen Konstruktor programmieren.

**Die wichtigsten Merkmale von Konstruktoren sind:**
- Ein Konstruktor hat den gleichen Namen wie die Klasse.
- Einem Konstruktor können Argumente übergeben werden.
- Ein Konstruktor wird immer mit *new* aufgerufen.
- Ein Konstruktor hat keinen Rückgabetyp.

***Überladen von Konstruktoren***

In vielen Anwendungen möchte man mehrere Möglichkeiten anbieten, ein Objekt zu erzeugen. Beim Ticketautomat wird in oben angegebenem Konstruktor das Attribut *gesamtsumme* auf 0 gesetzt. Es könnte aber auch Ticketautomaten geben, die schon mit Geld aufgeladen sind, um Wechselgeld zur Verfügung zu haben. Dazu benötigt man einen Konstruktor der folgende Form:

```
public Ticketautomat(int ticketpreis, int summe)
 {
     preis = ticketpreis;
     bisherGezahlt = 0;
     gesamtsumme = summe;
 }
```

In einer Klasse dürfen mehrere Konstruktoren programmiert werden. Da alle Konstruktoren den gleichen Namen haben, spricht man hier vom **Überladen von Konstruktoren**.
Der Compiler unterscheidet die Konstruktoren anhand der Parameterliste, d.h. wenn Sie mehrere Konstruktoren programmieren, müssen sich diese entweder in der Zahl der Parameter oder im Datentyp der Parameter unterscheiden. In Kapitel 5.4 finden Sie ein weiteres Beispiel zum Überladen von Konstruktoren und es wird auf das Überladen von Methoden eingegangen.

## 3.7 Die eigene Klassendokumentation

Auch wenn ein Klassendiagramm mit Operationsbeschreibungen existiert, sollte jedes Programm ausführlich dokumentiert werden. Die Dokumentation Ihrer eigenen Klassen sollte genau die Informationen bieten, die andere Programmierer benötigen, um die Klasse verwenden zu können, ohne die Implementierung (den Quellcode) zu kennen. Ein Kommentar kann mit „//“ eingeleitet werden. Mehrzeilige Kommentare werden folgendermaßen eingerahmt:

```
/*
        mehrere Zeilen Kommentar
*/
```

Die BlueJ-Umgebung benutzt ein Werkzeug namens **javadoc**, um für Ihre Klassen die Dokumentation zu erstellen (Menüpunkt *Tools – Project Documentation*). Kommentare, die in *javadoc* übernommen werden sollen, müssen mit

```
/**
```

eingeleitet werden. Vor einer Klassendefinition wird ein solcher Kommentar als Klassenkommentar aufgefasst, vor der Signatur einer Methoden als Methodenkommentar.
Im BlueJ-Editor können Sie direkt zwischen dem Quelltext und der Dokumentation wechseln.
Ihre Klassendokumentation sollte mindestens die folgenden Angaben enthalten:
- ✔ Klassenname
- ✔ Beschreibung von Zweck und Eigenschaften der Klasse
- ✔ Versionsnummer
- ✔ Autorennamen

- ✔ Dokumentation für jeden Konstruktor und jede Methode, diese sollte enthalten:
  - ✔ Namen der Methode
  - ✔ Ergebnistyp
  - ✔ Namen und Typen der Parameter
  - ✔ Beschreibung von Zweck und Arbeitsweise der Methode
  - ✔ Beschreibung jedes Parameters
  - ✔ Beschreibung des Ergebnisses

## 3.8 Der Debugger

Mit Hilfe des Debuggers kann die Ausführung eines Programms bzw. einzelner Methoden genauer untersucht werden. Er ist sehr nützlich beim Auffinden von Fehlerursachen und kann auch zum besseren Verständnis eines Programmablaufs bzw. geschachtelter Methodenaufrufe beitragen. Machen Sie sich mit Hilfe des BlueJ-Tutorials mit dem Debugger vertraut [5].

## 3.9 Bildschirmausgabe

Solange wir noch keine grafischen Benutzungsoberflächen programmieren können, müssen wir Informationen für den Programmbenutzer direkt auf den Bildschirm, genauer gesagt in ein Konsolenfenster schreiben. BlueJ besitzt ein eigenes Konsolenfenster, das sich öffnet, sobald Sie eine Bildschirmausgabe anstoßen. Dieses Fenster können Sie auch über den Menübefehl „*View – show Terminal*“ explizit öffnen. Mit dem Befehl

```
System.out.print("Test");
```

können Sie den Text „Test“ auf die Konsole schreiben.
*System* ist eine Klasse, die von der Java-Bibliothek (die wir später noch näher betrachten werden) zur Verfügung gestellt wird. *out* ist ein Attribut der Klasse *System*, dessen Datentyp auch eine Bibliotheksklasse, die Klasse *PrintStream*, ist. *PrintStream* steht für die Standardausgabe unseres Computers, den Bildschirm. *print* wiederum ist eine Methode der Klasse *PrintStream*, die als Argument eine Zeichenkette erwartet und diese Zeichenkette auf der Konsole ausgibt.
Verwenden Sie statt der Methode *print*, die Methode *println*, wird nach der Ausgabe noch ein Zeilenumbruch durchgeführt. Der Befehl lautet dann

```
System.out.println("Test");
```

Sie können mit diesem Befehl auch Zahlen ausgeben, müssen dabei aber genau auf die Schreibweise achten. Die folgenden Zeilen

```
int bsp = 465;
System.out.println(bsp);
System.out.println("bsp");
```

erzeugen die Bildschirmausgabe:

```
465
bsp
```

Möchten Sie Text und Variableninhalte in der gleichen Zeile ausgeben, so können Sie diese mit dem Operator '+' verketten:

```
System.out.println("Die Variable bsp hat den Wert " + bsp);
```

Diese Anweisung erzeugt die Ausgabe:

```
Die Variable bsp hat den Wert 465
```

***Fragen:***

- ✗ Mit welchem Schlüsselwort wird in Java eine Klasse definiert?
- ✗ Was muss bei der Deklaration einer Variablen angegeben werden?
- ✗ Wie lautet der Operator für Zuweisungen, in welche Richtung erfolgen sie?
- ✗ Wie werden Objekte in Java gebildet?
- ✗ Wie wird auf ein Attribut oder eine Operation eines Objekts zugegriffen?
- ✗ Was wird in einer Variable vom Typ einer Klasse gespeichert?
- ✗ Welche Sichtbarkeiten für Attribute kennen Sie ?
- ✗ Woraus besteht die Signatur einer Methode?
- ✗ Wozu wird der Compiler benötigt?
- ✗ Welche der folgenden Fehler kann der Compiler finden: Syntaxfehler, logische Fehler?
- ✗ Nennen Sie zwei wichtige Merkmale eines Konstruktors!
- ✗ Wodurch zeichnet sich der Standardkonstruktor aus?

# 4 Arithmetik, Variablen und Kontrollstrukturen

***Lernziele:***

- **Wissen**
  - Implizite und Explizite Typkonversion kennen
  - Operatoren in Java und ihre Prioritäten kennen
  - Die wichtigsten Methoden der Bibliotheksklasse Math kennen
  - Die Kontrollstrukturen Alternativen und Schleifen kennen
  - Wissen, wie eine bedingte Anweisung programmiert wird
- **Verstehen**
  - Die Besonderheiten der Ganzzahldivision kennen und verstehen
  - Die Auswertung zusammengesetzter Ausdrücke verstehen
  - Den Unterschied zwischen Ganzzahl- und Gleitpunkt-Arithmetik kennen
  - Den Unterschied zwischen dem Inkrementoperator als Präfixoperator und als Postfixoperator erklären können
- **Anwenden**
  - Ganzzahl- und Gleitpunktarithmetik in der Programmierung umsetzen können
  - Bedingungen in Flussdiagrammen und Struktogrammen formulieren können
  - Schleifen in Flussdiagrammen und Struktogrammen formulieren können
  - Vergleichsoperatoren und logische Operatoren in Anweisungen einsetzen können
  - Operationen mit Bedingungen implementieren können

## 4.1 Ganzzahlarithmetik

Ganze Zahlen werden in Java als Folge von Ziffern geschrieben. Ihnen kann ein positives oder negatives Vorzeichen vorangestellt werden. Beispiele sind

```
37
-13
+13
0
+327659430
```

Solche Zahlen, die direkt in das Programm geschrieben werden heißen **Numerale** (Zahlwörter). Zum Rechnen mit ganzen Zahlen formuliert man einen arithmetischen Ausdruck, dessen Schreibweise der mathematischen angelehnt ist. Ein arithmetischer Ausdruck besteht zunächst aus zwei Numeralen getrennt durch einen **Operator**. Die vier Grundrechenarten werden durch die folgenden Operatoren beschrieben:

- Addition: `+`
- Subtraktion: `-`
- Multiplikation: `*`
- Division: `/`

Da diese Operatoren stets zwischen zwei Operanden stehen, handelt es sich um **binäre Operatoren**. Anstelle der Numerale dürfen auch Variablen stehen. Diese müssen zuvor deklariert und initialisiert (siehe Seite 28) werden, z.B.

```
int a;
a = 3;
```

Die folgenden arithmetischen Ausdrücke sind gültig:

```
10/5          3-7          5+8          5*a
```

Ungültig dagegen sind die Ausdrücke `5a,` da der `*`-Operator nicht, wie in der Mathematik weggelassen werden darf, und `/3,` da das linke Numeral fehlt.
Das Ergebnis eines arithmetischen Ausdrucks zweier ganzer Zahlen ist eine ganze Zahl. Was ist dann das Ergebnis von

```
7/4   ?
```

Es ist nicht ____ , sondern ___, die Stellen hinter dem Dezimalpunkt werden abgeschnitten. Es gibt in Java einen weiteren binären Operator für die Ganzzahlarithmetik, den Modulo-Operator %, der den Rest der Ganzzahldivision ergibt, d.h.

```
7%4          ergibt 3
```

Allgemein gilt:

a%b = a- (a/b)*b.

Achtung: Eine ganzzahlige Division durch null ist nicht zugelassen und führt zum Programmabbruch! Dies bedeutet, dass Sie bei jeder ganzzahligen Division sicher stellen müssen, dass der zweite Operand nicht 0 ist.

### *Zusammengesetzte Ausdrücke*

Wird ein Numeral als elementarer Ausdruck angesehen, so kann jeder zusammengesetzte Ausdruck definiert werden als Verknüpfung zweier Ausdrücke mit einem Operator. Beispiele zusammengesetzter Ausdrücke sind

```
7-5*3
4/2+3*5
125-37-6
```

Um solche Ausdrücke auswerten zu können, müssen, wie in der Mathematik, Prioritäten festgelegt werden. In Java gilt die aus der Schulmathematik bekannte Regel „Punkt vor Strich“, d.h. die Operatoren `*, /` und `%` haben höhere Priorität als die Operatoren `+` und `-.` Bei mehreren aufeinanderfolgenden Operatoren gleicher Priorität, wird der Ausdruck von links nach rechts ausgewertet. Aus diesen Regeln ergeben sich die angegebenen Werte für die Ausdrücke:

```
5+3*2   → 5+(3*2) →  11
17-6-3  → (17-6)-3 →  8
4*2+12/2  → (4*2)+(12/2) → 14
```

Soll die Reihenfolge der Auswertung geändert werden, kann dies durch Hinzufügen runder Klammerpaare erreicht werden, wobei ein geklammerter Ausdruck vorrangig ausgewertet wird:

```
(5+3)*2  →  16
17-(6-3)  →  14
4*(2+12)/2  → 28
```

Da `*` und `/` die gleiche Priorität haben, führt weiteres Klammern zu dem selben Ergebnis:

```
(4*(2+12))/2  → 28
4*((2+12)/2)  → 28
```

### *Unäre Vorzeichenoperatoren*

Die Vorzeichen `+` und `-` gehören zu den unären Operatoren, da sie einem einzelnen Operanden zugeordnet bzw. vorangestellt werden. Sie haben eine höhere Priorität als alle binären Operatoren:

```
-(2+-5) → -(-3)  → 3
```

### *Lesen und Schreiben von Variablen*

Um das Ergebnis eines arithmetischen Ausdrucks verwenden zu können, müssen wir es in einer Variablen speichern, d.h. wir weisen den Wert des Ausdrucks einer Variablen zu (siehe auch Kapitel 3.2):

```
int b;
b = (7-4)*(4+3);
```

Dadurch wird der Wert der Variablen geändert, es wird in die Variable **geschrieben**.
Eine Variable kann auch auf der rechten Seite einer Wertzuweisung stehen, als Ausdruck oder als Teil eines Ausdrucks. In diesem Fall bleibt ihr Wert unverändert, die Variable wird **gelesen**. Eine Variable kann erst dann auf der rechten Seite verwendet werden, wenn sie zuvor initialisiert wurde, d.h. wenn ihr ein Wert zugewiesen wurde.
Beispiel:

```
int x,y;
x = 13-5*2;
y = 4*x*x+5*x-7;
```

Bei einer Wertzuweisung darf eine Variable auch auf beiden Seiten des Zuweisungsoperators stehen, d. h. die Anweisung

```
int i=5;

i=i+1;
```

die in der Mathematik ein Widerspruch ist, ist in Java erlaubt. Es wird zuerst der arithmetische Ausdruck auf der rechten Seite ausgewertet, in obigem Beispiel wird der Wert der Variable `i` gelesen und zu diesem Wert wird 1 addiert. Anschließend wird die Wertzuweisung ausgeführt und damit der berechnete Wert in die Variablen `i` geschrieben. Die Anweisung erhöht somit den Wert in der Variablen `i` um 1.

### *Zahlensysteme*

Wir hatten in der Einführung neben dem Dezimalsystem noch andere Zahlensysteme kennengelernt. Der Java-Compiler erlaubt das Speichern von Zahlen in drei weiteren Zahlensystemen, gekennzeichnet werden die Zahlen durch verschiedene Präfixe, die in folgender Tabelle dargestellt sind:

| **Zahlensystem** | **Präfix** | **Beispiel** | **Wert im Dezimalsystem** |
|---|---|---|---|
| Hexadezimalzahlen | 0x | 0xA5F | 2655 |
| Oktalzahlen | 0 | 015 | 13 |
| Binärzahlen | 0b | 0b1100111 | 103 |

Der Compiler erzeugt aus den Zahlen aller Zahlensysteme eine einheitliche interne Darstellung, die binäre. Die Wertzuweisungen in folgendem Beispiel werden alle in den gleichen Maschinencode übersetzt:

```
int z;
z = 27;
z = 0x1B;
z = 033;
z = 0b11011;
```

***Bereichsüberschreitung***

Wird der zulässige Zahlenbereich von int-Zahlen überschritten wird ohne Fehlermeldung falsch gerechnet, d.h.

2 147 483 647 + 1 → -2 147 483 648

## 4.2 Gleitpunktarithmetik

Für viele Berechnungen reichen ganze Zahlen nicht aus, wir brauchen gebrochene Werte, sehr kleine oder sehr große Zahlen. Dann werden **Gleitpunktzahlen**, auch Gleitkomma-, Fließkomma- oder Fließpunktzahlen genannt, verwendet. Bei Gleitpunkt-Numeralen werden Nachkommastellen durch einen Dezimalpunkt abgetrennt. Für sehr kleine oder große Zahlen gibt es die Exponentendarstellung, ein Zehnerexponent wird durch den Buchstaben E oder e markiert. Beispiele für Gleitpunkt-Numerale sind:

`7.35` `0.0003` `324.0` `1E23` ( $10^{23}$ )
`1E-31` ( $10^{-31}$ ) `-2.54E-4` ( $-2.54 \cdot 10^{-4} = -0.000254$ )

Der gleiche Zahlenwert kann also unterschiedlich geschrieben werden.
Aufgabe: Schreiben Sie die Zahl 14.5 auf drei verschiedene Arten!

Jedes Numeral, das einen Dezimalpunkt oder einen Exponenten enthält, hat den **Datentyp double.** Daraus ergibt sich für die nachfolgenden Numerale der angegebene Datentyp:

`25` ____
`25.0` _____
`2.5E1` _____

Die arithmetischen Operatoren `+, -, *` und `/` verarbeiten sowohl *int*- als auch *double*-Operanden. Rechenoperationen mit *int*-Werten sind schneller und liefern immer das exakte Ergebnis, *double*-Numerale brauchen mehr Speicherplatz und Ergebnisse können nicht exakt gespeichert werden, da eine Gleitpunktzahl nur mit einer begrenzten Zahl an Nachkommastellen gespeichert werden kann.
Sind alle Operanden vom gleichen Datentyp, so ist das Ergebnis auch von diesem Datentyp, d.h.

`16/5` → 3
`16.0/5.0` → 3.2

Was ist nun aber das Ergebnis von

`16/5.0` → ___

Wenn Sie als Ergebnis eine Gleitpunktzahl haben möchten, muss mindestens ein Operand ein Gleitpunkt-Numeral sein. Dabei wird intern der Ganzzahl-Operand in einen Gleitpunktwert umgewandelt. Man spricht hier von einer **impliziten Typkonversion**:

`4.0+5` → 4.0+5.0 → 9.0

Die implizite Typkonversion von *int* nach *double* ist möglich, da es zu jedem *int*-Wert einen äquivalenten *double*-Wert gibt. Da dies umgekehrt nicht der Fall ist, gibt es **keine** implizite Typumwandlung von *double* nach *int*.

***Explizite Typkonversion***

Dagegen gibt es die Möglichkeit eine **explizite Typkonversion** von *double* nach *int* durchzuführen. Der sogenannte **Typecast** wird als unärer Operator behandelt, der Datentyp steht in runden Klammern vor dem zu konvertierenden Ausdruck:

```
(int) 3.5  →                    3
(int) 3.5*3     →  3*3  →       9
(int) (3.5*3)   →  (int)10.5  →  10
```

In der Gleitpunkt-Arithmetik führt eine Division durch Null nicht zum Programmabbruch. Sie erhalten jedoch den Wert **Double.NaN**, der für „not a number“, also keinen sinnvollen Wert steht. Das Programm setzt die Ausführung fort, es gibt jedoch Schwierigkeiten, wenn Sie mit diesem Ergebnis weiter rechnen. **Division durch Null sollte also grundsätzlich vermieden werden!**

### *Mathematische Funktionen und die Bibliotheksklasse Math*

Für viele Berechnungen reichen die Grundrechenarten aus, in technischen Anwendungs-Programmen werden jedoch häufig weitere mathematische Funktionen, wie Wurzelberechnungen, trigonometrische Funktionen, Logarithmus und e-Funktion, benötigt. Damit nicht jeder Entwickler die entsprechenden Algorithmen selbst entwickeln und in Java-Quelltext umsetzen muss, werden diese für sehr viele Funktionen in der **Laufzeitbibliothek** zur Verfügung gestellt. Mit **Java-API** (Java Application Programming Interface) wird der Funktionsumfang der Laufzeitbibliothek bezeichnet. Wir werden uns später noch genauer mit der Laufzeitbibliothek beschäftigen, an dieser Stelle soll es ausreichen, die wichtigsten mathematischen Funktionen zu kennen und zu wissen, wie sie aufgerufen werden. Mathematische Funktionen werden als Klassenoperationen der Bibliotheksklasse **Math** bereitgestellt. Da es sich um Klassenoperationen handelt, benötigen wir kein Objekt, um sie aufzurufen, wir dürfen sie direkt über den Klassennamen aufrufen. Der Aufruf

```
double y = Math.sqrt(15);
```

berechnet die Wurzel der Zahl 15 und speichert das Ergebnis in der Variablen y.
In nachfolgender Tabelle finden Sie eine Auswahl mathematischer Funktionen mit ihren Operationsnamen und den erforderlichen Parametern.

| **Operationsname** | **Parameter** | **math. Funktion** | **Aufruf** |
|---|---|---|---|
| sqrt | x:double | $\sqrt{x}$ | Math.sqrt(x) |
| log | x:double | nat. Log.: ln(x) | Math.log(x) |
| exp | x:double | e-Funktion: $e^x$ | Math.exp(x) |
| sin | x:double, im Bogenmaß | Sinusfunktion: sin(x) | Math.sin(x) |
| cos | x:double, im Bogenmaß | Cosinusfunktion: cos(x) | Math.cos(x) |
| abs | x:double | Betrag: \|x\| | Math.abs(x) |
| pow | x,y: double, Basis und Exponent | $x^y$ | Math.pow(x,y) |

Die Bibliotheksklasse *Math* enthält außerdem nützliche mathematische Konstanten in Form von Klassenattributen, z.B. die Kreiszahl $\pi$ : **Math.PI** und die Eulerzahl e: **Math.E**. Weitere Operationen finden Sie in der API-Dokumentation in BlueJ unter dem Menü-Befehl *Help Java Class Libraries* oder im Internet unter https://docs.oracle.com/en/java/javase/11/docs/api/index.html. Suchen Sie mindestens drei weitere nützliche Methoden der Klasse *Math* in der API-Dokumentation.

## 4.3 Kontrollstrukturen, Flussdiagramme und Struktogramme

Wie wir am Beispiel des Fahrscheinautomaten gesehen haben, sollten manche Anweisungen nur ausgeführt werden, wenn gewisse Bedingungen erfüllt sind. In anderen Anwendungen kann es erforderlich sein, dass gewisse Anweisungen wiederholt ausgeführt werden müssen. Es reicht nicht aus, alle Anweisungen in der vorgegebenen Reihenfolge durchzuführen, wir benötigen spezielle Sprachmittel um die Ausführungsreihenfolge der Anweisungen zu steuern. Solche Sprachmittel werden **Kontrollstrukturen** genannt. Wir betrachten zwei wichtige Arten von Kontrollstrukturen: **Alternativen** und **Schleifen**.
Je mehr Kontrollstrukturen in einer Berechnungsmethode enthalten sind, um so wichtiger ist es schon im Entwurf in der Operationsbeschreibung den **Algorithmus** genau zu beschreiben. Übersichtlich darstellen lassen sich Algorithmen als Flussdiagramme oder Struktogramme.

### *Alternativen*

Bei einer Alternative, auch **bedingte Anweisung** genannt, wird eine Anweisung nur ausgeführt, wenn eine bestimmte Bedingung erfüllt ist. Ist die Bedingung nicht erfüllt, wird die Anweisung nicht ausgeführt. Nachfolgend sehen Sie die Darstellung im Flussdiagramm und im Struktogramm:

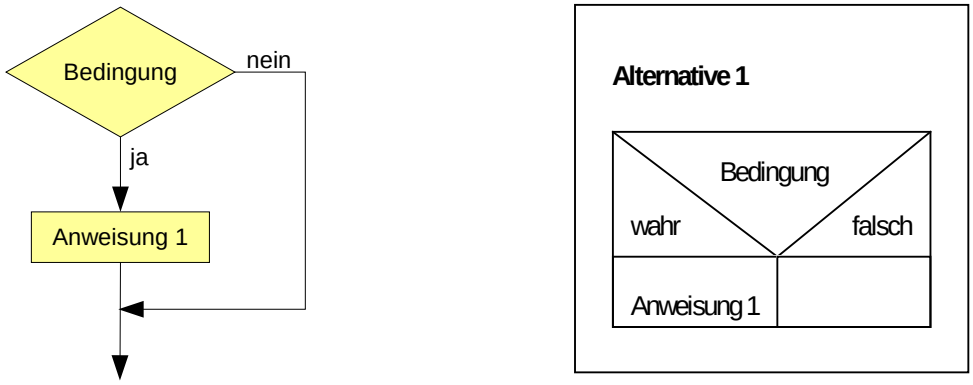



Abbildung 4.1 Flussdiagramm(links) und Struktogramm(rechts) für die einfache Alternative

Im Beispiel des Fahrscheinautomaten möchten wir, dass keine negativen Geldbeträge an die Operation *geldEinwerfen* übergeben werden dürfen. Dies erreichen wir, indem wir die Anweisung

```
bisherGezahlt = bisherGezahlt + betrag;
```

nur dann ausführen, wenn der Wert der Variablen *betrag* größer als Null ist. Im Struktogramm wird dies folgendermaßen dargestellt:

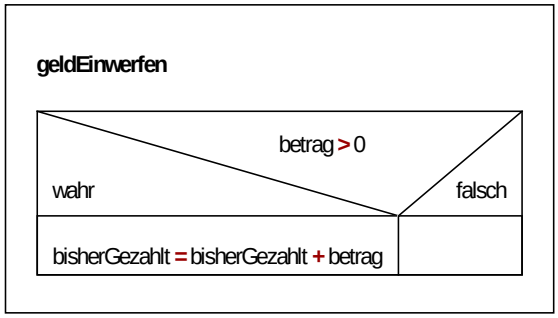



Abbildung 4.2 Struktogramm für die einfache Alternative am Beispiel der Methode *geldEinwerfen*

Eine Erweiterung dieser einseitigen Alternative ist die zweiseitige, bei der es zwei Anweisungen gibt: die erste Anweisung wird ausgeführt, wenn die Bedingung erfüllt ist, die zweite Anweisung, wenn die Bedingung nicht erfüllt ist.

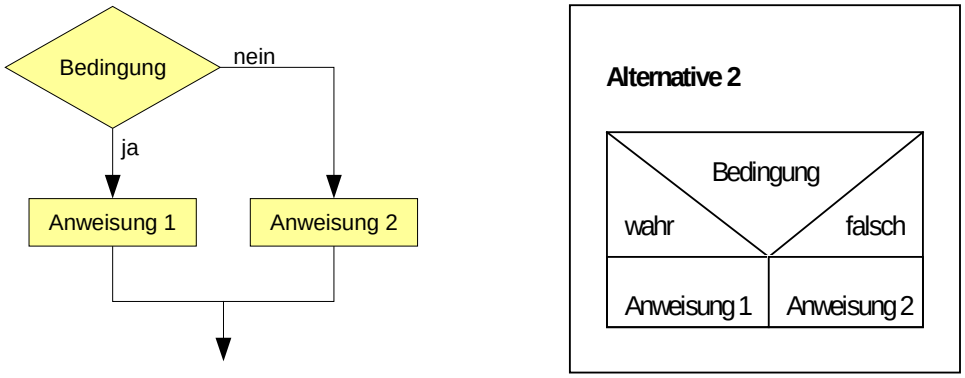



Abbildung 4.3 Flußdiagramm und Struktogramm für die zweiseitige Alternative

Das obige Beispiel (Abbildung 4.2) hat den Nachteil, dass bei einem negativen Wert der Variablen *betrag* nichts passiert, d.h. der Aufrufer der Methode merkt nicht, dass er einen unzulässigen Wert eingegeben hat. Besser wäre es in diesem Fall, dem Anwender einen Hinweis zu geben, dass nur positive Beträge eingegeben werden dürfen. Erstellen Sie das zugehörige Struktogramm:

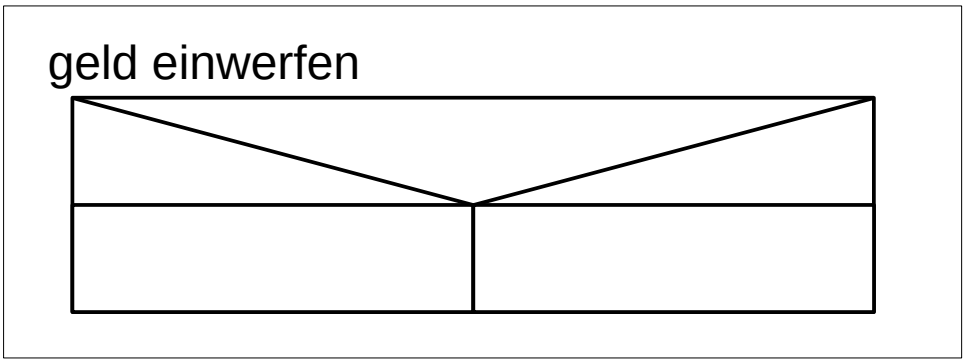



Abbildung 4.4 Struktogramm für die zweiseitige Alternative am Beispiel *geldEinwerfen*

Alternativen können auch beliebig tief geschachtelt werden:

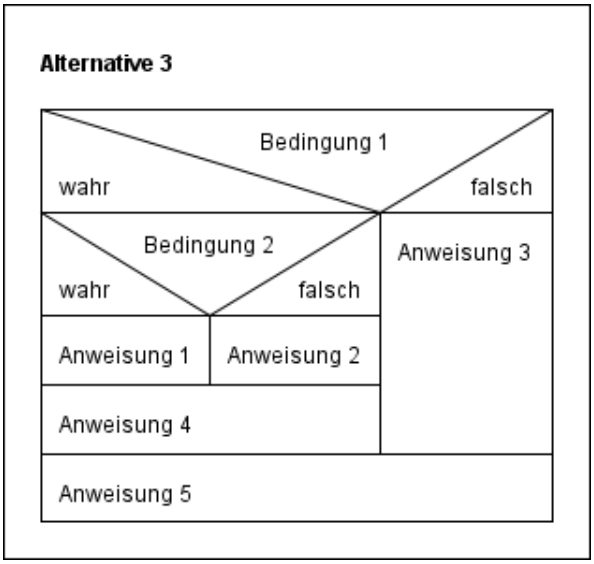



Abbildung 4.5: Geschachtelte Alternative im Struktogramm

Manchmal möchte man auch bei verschiedenen Werten einer Zahl unterschiedliche Anweisungen ausführen. Dies lässt sich im Struktogramm folgendermaßen darstellen:

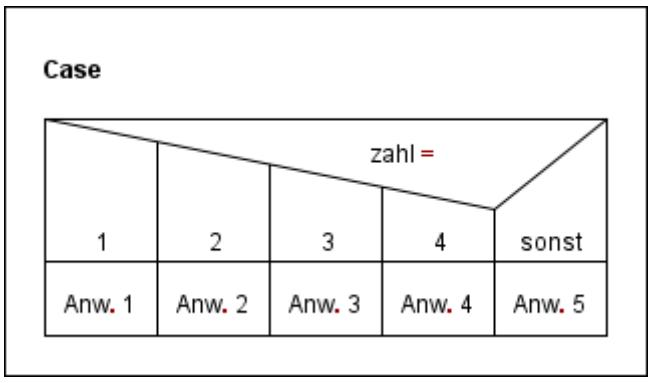



Abbildung 4.6: Struktogramm für Mehrfachauswahl

## 4.4 Bedingte Anweisungen in Java

In Java wird eine Alternative mit dem Schlüsselwort ***if*** gestartet und die Bedingung in runde Klammern gesetzt. Bedingungen können wahr oder falsch sein, d.h. sie müssen als Ergebnis einen Wahrheitswert liefern, also vom Datentyp **boolean** sein.Um solche Bedingungen zu formulieren benötigen wir **Vergleichsoperatoren**, sogenannte **relationale Operatoren**, die in der folgenden Übersicht zusammengestellt sind:

| **Operator:** | **Bedeutung:** |
|---|---|
| < | kleiner |
| > | größer |
| <= | kleiner oder gleich |
| >= | größer oder gleich |
| == | gleich |
| != | ungleich |

Besonders zu beachten ist, dass die Überprüfung auf Gleichheit nicht mit „=“ sondern mit dem Operator „==“ erfolgt. Der Operator „=“ ist schon für die Wertzuweisung reserviert (siehe Kapitel 3.2).

Diese Vergleichsoperatoren können auf ganze Zahlen und Gleitpunktzahlen angewendet werden. Die Operatoren „==“ und „!=“ liefern jedoch bei Gleitpunktzahlen meist nicht das gewünschte Ergebnis, da aufgrund von Rundungsfehlern die interne Darstellung zweier gleicher Gleitpunktzahlen nicht gleich sein muss. Daher prüft man, ob der Betrag der Differenz der beiden Zahlen kleiner als eine sehr kleine Zahl ist. Formulieren Sie den Vergleich $|x-y| < 10^{-6}$ in Java.

Relationale Operatoren haben eine niedrigere Priorität als die arithmetischen Operatoren, d.h.

2+3 < 5+7 ➔ (2+3) < (5+7) ➔ 5<12 ➔ true

Wir übertragen nun die Beispiele aus Kapitel 4.3 in Java-Quelltext:
Beispiel 1: Die einfache Alternative am Beispiel der Methode *geldEinwerfen*

```
if (betrag > 0)
     bishergezahlt = bishergezahlt + betrag;
```

Beispiel 2: Die zweiseitige Alternative am Beispiel der Methode *geldEinwerfen*

```
1    if (betrag > 0)
2         bishergezahlt = bishergezahlt + betrag;
3    else
4         System.out.println("Bitte positive Beträge eingeben!");
```

Ist die Bedingung `betrag >0` erfüllt, so wird die Zeile 2 ausgeführt, ist sie nicht erfüllt, d.h. ist der Betrag kleiner oder gleich 0, so wird die Zeile 4 ausgeführt.
Sollen mehrere Anweisungen ausgeführt werden, so müssen diese in geschweifte Klammern gesetzt werden, wie im nachfolgenden Beispiel dargestellt:

```
if (betrag > 0)
{
     bishergezahlt = bishergezahlt + betrag;
     System.out.println("Sie haben"+bishergezahlt+"bezahlt.");
}
```

Häufig hängt die Ausführung einer Anweisung von mehreren Bedingungen ab. Nehmen wir an der Fahrscheinautomat nimmt als größten Schein 10,- €-Scheine an. Dann soll in der Methode *geldEinwerfen* der Betrag größer Null und kleiner oder gleich Zehn sein. Dies könnte zum Beispiel mit den folgenden Anweisungen erreicht werden:

```
if (betrag > 0)
{
     if (betrag <=10)
          bishergezahlt = bishergezahlt+ betrag;
}
```

Hinzufügen einer Anweisung bei Nichterfüllung der Bedingungen führt zu einem schon recht umfangreichen Quelltext:

```
if (betrag > 0)
{
     if (betrag <=10)
          bishergezahlt = bishergezahlt+ betrag;
     else
          System.out.println("Ungültiger Betrag!");
}
```

```
else
     System.out.println("Ungültiger Betrag!");
```

Die Bildschirmausgabe muss hier in zwei Zweige der geschachtelten Bedingung geschrieben werden. Dies lässt sich vermeiden, indem die beiden Bedingungen miteinander verknüpft werden. Zur Verknüpfung von Wahrheitswerten stehen **logische Operatoren** zur Verfügung, die wichtigsten sind:

| **Operator:** | **Bedeutung:** |
|---|---|
| && | logisches Und: Liefert genau dann wahr, wenn beide Operanden wahr sind. |
| \|\| | inklusives logisches Oder: Liefert genau dann wahr, wenn mindestens einer der beiden Operanden wahr ist. |
| ^ | exklusives logisches Oder: Liefert wahr, wenn genau einer der Operanden wahr ist. |
| ! | logisches Nicht: dies ist ein unärer Operator, der genau dann wahr liefert, wenn der Operand falsch ist. |

Die binären logischen Operatoren haben eine geringere Priorität als die arithmetischen und relationalen Operatoren, das logische Nicht hat die gleiche Priorität, wie die Vorzeichenoperatoren. Obiges Beispiel lässt sich nun wesentlich kürzer schreiben:

```
if (betrag > 0 && betrag <=10)
          bishergezahlt = bishergezahlt+ betrag;
else
     System.out.println("Ungültiger Betrag!");
```

***Auswertung logischer Operatoren***

Da Wahrheitswerte nur die Werte true und false annehmen können, kann man die logischen Operatoren mit Wahrheitstabellen vollständig definieren. Die Wahrheitstabelle für && lautet:

| | | |
|---|---|---|
| true && true | → | true |
| true && false | → | false |
| false && true | → | false |
| false && false | → | false |

Aufgabe: Erstellen Sie selbst die Wahrheitstabellen für die anderen logischen Operatoren.

Bei zusammengesetzten arithmetischen Ausdrücken, werden zunächst die Operanden ausgewertet und diese dann verknüpft, z.B.

(4+7)*(6-3) → 11*3 → 33

Wie sieht das nun bei zusammengesetzten Bedingungen aus? Aus der Wahrheitstabelle für && sieht man, dass das Ergebnis nur wahr sein kann, wenn der erste Operand wahr ist. Ist der erste Operand *falsch*, so ist eine Auswertung des zweiten Operanden nicht erforderlich und wird auch nicht durchgeführt. Das folgende Beispiel zeigt, dass diese Vorgehensweise nicht nur Rechenzeit spart, sondern auch durchaus sinnvoll ist:

(b !=0) && (a/b >0)

Die Auswertung des zweiten Operanden in diesem Beispiel ist nur dann sinnvoll, wenn das Ergebnis des ersten Operanden *true* ist. Diese Art der Auswertung wird **teilweise Auswertung** genannt: der zweite Operand wird nur dann ausgewertet, wenn dies für das Ergebnis noch von Bedeutung ist.
Aufgabe: Wann muss bei || der zweite Operand ausgewertet werden?

Haben wir mehrere Bedingungen, so kann es schnell zu einer mehrfachen Verschachtelung kommen. Es gibt nun zwei Möglichkeiten diese Verschachtelung zu vermeiden, entweder mit Hilfe des Befehls ***else if*** oder mit einer ***switch***-Anweisung. Man könnte bei obigem Beispiel betrachten, dass es nur die Möglichkeit gibt Münzen mit dem Wert 1,2,5 oder 10 einzuwerfen. Dies kann folgendermaßen realisiert werden:

```
1    if (betrag == 1)
2              bishergezahlt = bishergezahlt+ 1;
3    else if (betrag == 2)
4              bishergezahlt = bishergezahlt+ 2;
5    else if (betrag == 5)
6              bishergezahlt = bishergezahlt+ 5;
7    else if (betrag == 10)
8              bishergezahlt = bishergezahlt+ 10;
9    else
10        System.out.println("Ungültiger Betrag!");
```

Besitzt der Betrag den Wert 1, so wird die Bedingung *1* überprüft und Zeile *2* ausgeführt, besitzt der Betrag den Wert 2, so werden die Bedingungen *1* (falsch) und *3* (wahr) überprüft und die Zeile *4* ausgeführt, usw.. Ist keine der 4 Bedingungen erfüllt, wird Zeile 10 ausgeführt. Der else-Abschnitt darf wieder weg gelassen werden, dann passiert nichts, wenn keine Bedingung erfüllt ist.

### *Die Switch-Anweisung*

Die zweite oben erwähnte Alternative, um verschachtelte if-Anweisungen zu vermeiden, ist in einigen Fällen eine switch-Anweisung. Bei einer switch-Anweisung wird ein Ausdruck nacheinander mit einer Liste fester Werte verglichen. Der Datentyp des Ausdrucks kann *int*, *char* oder *String* (ab der Java-Version 7) sein. Dies kann auf zwei verschiedene Arten erfolgen.

```
switch(Ausdruck)
{
     case wert1: Anweisungen1;
                 break;
     case wert2: Anweisungen2;
                 break;
     default: Anweisungen3;
                 break;
}
```

```
 switch(Ausdruck)
{
    case wert1:
    case wert2:
    case wert3:
         Anweisungen1;
         break;
    case wert4:
    case wert5:
         Anweisungen2;
         break;
    default: Anweisungen3;
         break;
 }
```

An der Stelle `Ausdruck` muss hier eine Variable vom Typ *int*, *char* oder *String* stehen, an den Stellen `wert1, wert2,` … ein beliebiger Wert vom gleichen Datentyp. Die Anweisungen geben an, welche Befehle bei Übereinstimmung des Ausdrucks mit dem entsprechenden Wert ausgeführt werden sollen.
Es können beliebig viele `case`-Abschnitte enthalten sein. Der `default`-Abschnitt deckt alle anderen Fälle ab und ist optional. Fehlt der `default`-Abschnitt und passt keiner der `case`-Ausdrücke, so bleibt die gesamte `switch`-Anweisung ohne Auswirkung.

Jeder Zweig, der Anweisungen enthält, sollte mit einem `break` enden. Das letzte `break` kann auch weggelassen werden, dies ist jedoch kein guter Programmierstil. Ist am Ende eines `case`-Abschnitts keine `break`-Anweisung vorhanden, so wird auch der nächste `case`-Abschnitt ausgeführt. Dies wurde beim zweiten Beispiel oben ausgenutzt: nimmt der Ausdruck den Wert `wert1` (Die Variable Ausdruck kann den Datentyp int, char, oder String besitzen) an, so werden die Anweisungen `Anweisungen1` durchgeführt. In manchen Fällen kann dies sinnvoll sein, sollte dann jedoch im Kommentar beschrieben sein.

Programmieren Sie eine `switch`-Anweisung, die abhängig von der `int`-Variablen `tag` mit den Werten 1 bis 7 der String-Variablen `tagText` die Werte „Montag“,. „Dienstag“, ... zuweist.

## 4.5 Weitere Operatoren

Für einige Operatoren gibt es Kurzschreibweisen. Die Anweisung

```
i = i + 1;
```

kann in der Kurzform

```
i++;
```

geschrieben werden. Das Erhöhen einer Variablen um den Wert eins wird insbesondere bei Schleifen (siehe Kapitel 6) häufig benötigt. Der Operator ++ heißt Inkrementoperator und ist ein unärer Operator. Analog gibt es den unären Dekrementoperator --. In folgender Tabelle sind weitere Operatoren mit Beispielen angegeben, die eine abkürzende Schreibweise darstellen.

| Operator | Bedeutung | Beispiel | Langform |
|---|---|---|---|
| ++ | erhöhe um 1 | k++; | k = k+1; |
| -- | erniedrige um 1 | j--; | j = j-1; |
| += | addiere Wert auf rechter Seite zur Variablen auf der linken Seite | x+=5; | x = x+5; |
| -= | subtrahiere Wert auf rechter Seite von der Variablen auf der linken Seite | y-=7; | y = y-7; |
| *= | multipliziere die Variable auf der linken Seite mit dem Wert auf der rechten Seite | z*=3; | z = z*3; |
| /= | dividiere die Variable auf der linken Seite durch den Wert auf der rechten Seite | w/=2; | w=w/2; |

Der Inkrement- und der Dekrementoperator weisen eine Besonderheit auf, wenn Sie innerhalb einer Anweisung verwendet werden. Welchen Wert liefert der Ausdruck `a++,` wenn er auf der rechten Seite einer Zuweisung verwendet wird? Welche Werte haben die Variablen `a` und `b` also nach den folgenden beiden Anweisungen?

```
int a = 1;
int b = a++;
```

Die Antwort lautet: _____ .

Dieser, auch **Postfixoperator** genannte, Ausdruck liefert zuerst den Wert der Variablen und inkrementiert dann. Der Operator `++` kann auch als **Präfixoperator** verwendet werden, der zuerst inkrementiert und dann den Wert liefert. Die Anweisungen

```
int a = 1;
int c = ++a;
```

ergeben somit _____ .

Analog kann der Dekrementoperator `--` als Präfix- und Postfixoperator verwendet werden.

### *Bedingte Operatoren*

Der bedingte Operator bietet eine Kurzform einer zweiseitigen **if**-Anweisung für solche if-Anweisungen, in denen der Wert einer Variablen abhängig von einer Bedingung unterschiedlich gesetzt werden soll. Zum Beispiel kann die Anweisung

```
int b;
if (a > 0)
     b = 1;
else
     b = 2;
```

in der Kurzform

```
int b = (a>0) ? 1 : 2;
```

geschrieben werden. Ist die Bedingung vor dem `?`, hier `a > 0`, erfüllt, so wird der Variablen, hier `b`, der Wert vor dem `:`, hier `1` zugewiesen, sonst der Wert nach dem `:`, hier `2`.

***Übersicht aller bisher behandelten Operatoren***

| Operator | Priorität | Operandentypen | Bemerkung |
|---|---|---|---|
| + | 1 | numerisch | positives Vorzeichen |
| - | 1 | numerisch | negatives Vorzeichen |
| ! | 1 | boolean | logisches Nicht |
| ++ | 1 | numerisch | Inkrementoperator als Präfixop. |
| -- | 1 | numerisch | Dekrementoperator als Präfixop. |
| * | 2 | numerisch | |
| / | 2 | numerisch | Division |
| % | 2 | ganzzahlig | Divisionsrest |
| + | 3 | numerisch | Addition |
| - | 3 | numerisch | Subtraktion |
| < | 4 | numerisch | Vergleich „kleiner“ |
| <= | 4 | numerisch | Vergleich „kleiner oder gleich“ |
| > | 4 | numerisch | Vergleich „größer“ |
| >= | 4 | numerisch | Vergleich „größer oder gleich“ |
| == | 5 | primitiv | Vergleich „gleicher Wert“ |
| != | 5 | primitiv | Vergleich „anderer Wert“ |
| ^ | 6 | boolean | logisches exklusives Oder |
| && | 7 | boolean | logisches Und |
| \|\| | 8 | boolean | logisches inklusives Oder |
| = | 9 | beliebig | Wertzuweisung |
| += | 9 | numerisch | direkte Addition |
| -= | 9 | numerisch | direkte Subtraktion |
| *= | 9 | numerisch | direkte Multiplikation |
| /= | 9 | numerisch | direkte Division |

Durch das Setzen von Klammern können die Prioritäten verändert werden, z.B.

```
x+=(3+5)*(8-4)
```

***Fragen:***

- ✗ Was ist der Unterschied zwischen 9/4 und 9.0/4.0?
- ✗ Was berechnet der %-Operator?
- ✗ Wann kommt es in arithmetischen Ausdrücken zu einer impliziten Typkonversion?
- ✗ Wie funktioniert die explizite Typumwandlung von *double* nach *int*?
- ✗ Wie lassen sich Wurzel bzw. Betrag einer Zahl berechnen?
- ✗ Mit welchem Operator können zwei Zahlen auf Gleichheit überprüft werden?
- ✗ Geben Sie die Operatoren für logisches Und und logisches Nicht an!
- ✗ Mit welchem Schlüsselwort wird eine Konstante in Java deklariert?
- ✗ Was steht links vom ? beim bedingten Operator?
- ✗ Wann kann eine switch-Anweisung anstelle einer if-Anweisung verwendet werden?
- ✗ Was passiert, wenn in einem case-Block einer switch-Anweisung kein break enthalten ist?
- ✗ Was ist der Unterschied zwischen `--` als Präfix- und Postfixoperator?

# 5 Klassen und Objekte in Java, Vertiefung 1

***Lernziele:***

- **Wissen**
  - Die Begriffe Sichtbarkeit und Lebensdauer einer Variablen kennen
- **Verstehen**
  - Die Begriffe Sichtbarkeit und Lebensdauer erklären und unterscheiden können
  - Verstehen, wie ein Objekt ein anderes Objekt erzeugen kann
  - Internen und externen Methodenaufruf unterscheiden können
- **Anwenden**
  - Klassen, die Objekte einer anderen Klasse verwenden implementieren können
  - Internen und externen Methodenaufruf korrekt implementieren können

## 5.1 Sichtbarkeit und Lebensdauer von Variablen

Wir wollen zunächst den Begriff Sichtbarkeit**,** den wir im Zusammenhang mit Attributen schon betrachtet hatten und den neuen Begriff Lebensdauer definieren und sie dann auf die bisher kennengelernten Arten von Variablen anwenden.
Unter der **Sichtbarkeit** einer Variablen versteht man den Bereich innerhalb des Quelltextes, von dem aus wir auf diese Variable zugreifen dürfen. Dies bedeutet, wenn Sie an einer bestimmten Stelle im Quelltext eine Variable verwenden möchten, müssen Sie überprüfen, ob Sie sich im Sichtbarkeitsbereich dieser Variablen befinden. Die Sichtbarkeit ist für uns also wichtig, wenn wir eine Klasse programmieren.
Die **Lebensdauer** dagegen bezieht sich auf die Programmausführung. Sie beschreibt, wie lange eine Variable (also der ihr zugeordnete Speicherplatz) während der Programmausführung existiert, bevor sie zerstört wird.
In Kapitel 3.4 hatten wir verschiedene Variablen betrachtet, mit der eine Operation arbeiten kann. Wir wollen diese Variablen hier nochmals betrachten und ihre Sichtbarkeit und Lebensdauer angeben.

### *Attribute*

Attribute werden innerhalb der Klasse, aber außerhalb aller Konstruktoren und Methoden deklariert. Daher haben Sie mindestens klassenweite Sichtbarkeit. Dies bedeutet, dass in allen Methoden und Konstruktoren der Klasse auf sie zugegriffen werden kann. Die in den Attributen gespeicherten Daten bleiben somit während der gesamten Lebensdauer eines Objektes erhalten, sie haben die gleiche Lebensdauer wie das Objekt. Wird also durch den Aufruf einer Methode der Wert eines Attributs verändert, so wird beim nächsten Methodenaufruf mit dem veränderten Wert gearbeitet. Nur wenn ein Attribut *public* deklariert ist, kann auch von außerhalb der Klasse darauf zugegriffen werden.

### *Lokale Variablen*

Eine **lokale Variable** ist eine Variable, die im Rumpf einer Methode deklariert wird. Daher dürfen wir auch nur innerhalb dieser Methode auf die Variable zugreifen. Die Sichtbarkeit einer lokalen Variable ist somit die Methode, in der sie deklariert ist. Diese Sichtbarkeit kann noch weiter eingeschränkt werden, wie folgendes Beispiel zeigt:

```
1    public double rechne (double preis, int rabattfall)
2    {
3         double erg;
```

```
4         if (rabattfall == 1)
5         {
6              double rabatt = 0.1*preis;
7              erg = preis- rabatt;
8         }
9         else
10        {
11             erg = preis;
12        }
13        return erg;
14   }
```

Die Variable `erg` ist nur innerhalb der Methode `rechne` sichtbar.
Die Sichtbarkeit der Variablen `rabatt` ist auf die Quelltextzeilen 6 bis 7 beschränkt, die Variable ist nur in dem Block sichtbar, in dem sie deklariert wurde, genauer gesagt zwischen ihrer Deklaration und dem Ende des Blocks, in dem sie deklariert wurde. Ein **Block** ist der Bereich innerhalb eines geschweiften Klammerpaars. Innerhalb der Methode `rechne` gibt es zwei Blöcke, einer von Zeile 5 bis 8 und einer von Zeile 10 bis 12. In diesem zweiten Block kann auf die Variable `rabatt` nicht zugegriffen werden. Lokale Variablen existieren nur für die Dauer der Ausführung einer Methode, d.h. ihre Lebensdauer ist auf die Zeit der Ausführung der Methode beschränkt. Bei einem späteren Aufruf der gleichen Methode wird eine neue Variable erzeugt, der Wert der letzten Ausführung ist nicht mehr gespeichert. Lokale Variablen dienen somit nur der kurzzeitigen und nicht der dauerhaften Datenspeicherung.

***Parameter***

Formale Parameter werden innerhalb der Signatur einer Methode deklariert. Sie bekommen ihre Werte beim Methodenaufruf von den aktuellen Parametern (auch Argumente genannt). Die Sichtbarkeit ist wie bei lokalen Variablen auf die jeweilige Methode oder den Konstruktor beschränkt und auch die Lebensdauer ist auf die Zeit der Ausführung der Methode beschränkt.

## 5.2 Die Objektreferenz „this“

Manchmal möchte man innerhalb einer Klasse auf das aktuelle Objekt in seiner Gesamtheit zurückgreifen und nicht nur auf ein bestimmtes Attribut. Dies geht mit dem Schlüsselwort **„this“**. Anhand eines Beispiels soll die Verwendung von „this“ veranschaulicht werden. Dieses Beispiel werden wir in den folgenden Kapiteln erneut verwenden. Ein weiteres Beispiel werden wir im Zusammenhang mit Vererbung kennen lernen.

Beispiel 5.1: Wir wollen einen Ortsvektor im zweidimensionalen Raum durch eine Klasse beschreiben. Die Attribute der Klasse sind somit seine x- und seine y-Koordinate:

| **COrt2** |
| --- |
| - x: double<br>- y: double |

Nachfolgend sehen Sie den Quelltext der Klasse COrt2. Ergänzen Sie zunächst die fehlenden Zeilen für die Get- und Set-Methoden:

```
/**
* Diese Klasse definiert einen Ortsvektor im zweidimensionalen
* Raum. Der Ortsvektor wird beschrieben durch seine x- und
* seine y-Koordinate.
* @author  Kirstin Baumann
* @version 2022.09.16
*/

public class COrt2
{ // Beginn Klasse COrt2
  // Attribute
  private double x; // x-Koordinate
  private double y; // y-Koordinate

  /**
  * Konstruktor zum Erzeugen von Objekten der Klasse COrt2
  * @param x   x-Koordinate
  * @param y   y-Koordinate
  */
  public COrt2(double x, double y)
  {



  }
  // Verwaltungsoperationen
  /**
  * Set-Methode für die x-Koordinate
  * @param xkoord     x-Koordinate
  */
  public void SetX(double xkoord)
  {

  }
  /**
  * Set-Methode für die y-Koordinate
  * @param ykoord     y-Koordinate
  */
  public
  {

  }
  /**
  * Get-Methode für die x-Koordinate
  * Gibt die x-Koordinate zurück.
  * @return x-Koordinate
  */
  public double X()
  {
    return       ;
  }
```

```
  /**
  * Get-Methode für die y-Koordinate
  * Gibt die y-Koordinate zurück.
  * @return y-Koordinate
  */
  public
  {
                ;
  }
}// Ende Klasse Ortsvektor
```

Nun betrachten wir den Konstruktor der Klasse *COrt2*. Wie viele Variablen existieren nun während der Konstruktor ausgeführt wird?
Die Antwort lautet_: _ Attribute und _ Parameter.
In dieser Situation sind 2 Namen überladen. Attribute und Parameter sind immer unterschiedliche Variablen, die unabhängig voneinander existieren, auch wenn sie den gleichen Namen haben. Auf welche Variable wird nun zugegriffen, wenn wir den Variablennamen *x* innerhalb des Konstruktors verwenden, auf das Attribut oder den Parameter?
Die Spezifikation der Sprache Java besagt, dass immer die Definition des nächsten umschließenden Blocks verwendet wird. Da der Parameter *x* im Konstruktor definiert ist, das Attribut *x* dagegen in der Klasse außerhalb vom Konstruktor, wird der Parameter verwendet. Die Zuweisung *x=x* würde dem Parameter *x* seinen eigenen Wert zuweisen, was keine Auswirkung hat.
Was jetzt noch fehlt ist ein Mechanismus um auf ein Attribut zugreifen zu können, wenn in einer Methode eine lokale Variable mit gleichem Namen definiert ist. Genau dafür wird das Schlüsselwort *this* verwendet. Der Ausdruck *this* bezieht sich auf das aktuelle Objekt, *this.x* bezeichnet somit das Attribut *x* des aktuellen *COrt2*-Objektes. Der Konstruktor lautet somit:

```
  public COrt2(double x, double y)
  {
     this.x = x;
     this.y = y;
  }
```

Man könnte statt dessen auch für die Parameter andere Namen verwenden als für die Attribute, es erhöht jedoch die Lesbarkeit des Quelltextes, wenn immer der Name verwendet wird, der die Bedeutung der Variablen am besten beschreibt. Sobald Sie eine Methode aufrufen, wird intern der Variablen *this* das aufrufende Objekt zugewiesen. Zum Beispiel führt die Anweisung

```
ort2.SetX(17);
```

intern zu der Zuweisung *this = ort2.* Anschließend wird dann in der Methode *SetX* die Anweisung *this.x=17* ausgeführt, d.h. dem Attribut *x* des Objekts *ort2* der Wert 17 zugewiesen.

## 5.3 Objekte als Attribute

Attribute können nicht nur vom einfachen Datentyp sein, sie können auch als Datentyp eine Klasse haben. Als Beispiel betrachten wir die Klasse Kreis. Ein Kreis wird beschrieben durch seinen Mittelpunkt und seinen Radius. Von welchem Datentyp ist nun der Mittelpunkt? Ein Punkt kann durch den in Kapitel 5.2 betrachteten Ortsvektor beschrieben werden.

Mit der Klasse *COrt2* haben wir einen neuen Datentyp, den Datentyp für das Attribut *mitte* der Klasse *CKreis* definiert.

Beispiel 5.2

```
/**
* Diese Klasse definiert einen Kreis. Er wird verwendet um ein
* Beispiel eines Attributs vom Typ einer Klasse zu geben.
* Der Kreis wird beschrieben durch seinen Mittelpunkt und den
* Radius.
*/
public class CKreis
{ // Beginn Klasse CKreis
  private COrt2 mitte;       // Mittelpunkt vom Typ COrt2
  private double radius;     // Radius
  /**
  * Erzeuge ein Objekt der Klasse CKreis
  * @param mx: x-Koordinate Mittelpunkt
  *        my: y-Koordinate Mittelpunkt
  *        r : Radius
  */
  public CKreis(double mx, double my, double r)
  {
    // es muss ein Objekt der Klasse COrt2 erzeugt werden,
    // also ein Konstruktor der Klasse COrt2 aufgerufen werden
    mitte =
    radius =    ;
  }
    // Verwaltungsoperationen
    /**
    * Setzen der x-Koordinate des Mittelpunkts
    * @param x    x-Koordinate Mittelpunkt
    */
    void SetMitteX(double x)
    {
        mitte.             ;
    }
     /**
     * Get-Methode für die x-Koordinate vom Mittelpunkt
     * @return x-Koordinate vom Mittelpunkt
     */
    double MitteX()
    {
        return mitte.    ;
    }
    /**
    * Setzen der y-Koordinate des Mittelpunkte
    * @param y    y-Koordinate Mittelpunkt
    */
    void SetMitteY(double y)
    {
                                          ;
    } // Ende Set-Methode für x-Koordinate Mittelpunkt
```

```
    /**
     * Get-Methode für die y-Koordinate vom Mittelpunkt
     * @return  y-Koordinate vom Mittelpunkt
     */
    double MitteY()
    { // Anfang Get-Methode für x-Koordinate Mittelpunkt
        return
;
    } // Ende Get-Methode für x-Koordinate Mittelpunkt
```

Ergänzen Sie selber die Verwaltungsoperationen für den Radius.
In der Vorlesung und im Praktikum werden wir ein weiteres Beispiel betrachten, mit dem die Digitalanzeige einer Uhr simuliert werden soll. Anhand dieses Beispiels werden die Kapitel 5.3 bis 5.5 wiederholt.

## 5.4 Überladen von Konstruktoren und Methoden

In Kapitel 3.6 wurde das **Überladen von Konstruktoren** eingeführt**.**
Im folgenden sind die Signaturen von mehreren möglichen Konstruktoren für die Klasse *COrt2* aus Beispiel 5.1 angegeben:

```
public COrt2()
public COrt2(double wert)
public COrt2(String x, String y)
public COrt2(COrt2 o)
```

Die letzte Signatur gehört zu dem sogenannten **Kopierkonstruktor**. Ein Kopierkonstruktor gibt uns die Möglichkeit, wie der Name schon sagt, ein vorhandenes Objekt zu kopieren. Dies bedeutet, dass beim Aufruf des Kopierkonstruktors ein neues Objekt erzeugt wird mit den Attributwerten eines schon vorhandenen Objekts. Im folgenden ist der vollständige Kopierkonstruktor und seine Verwendung angegeben:
Kopierkonstruktor:

```
public COrt2(COrt2 ort)
{
     x = ort.x;
     y = ort.y;
}
```

Aufruf des Kopierkonstruktors: Um den Kopierkonstruktor aufrufen zu können, muss zunächst ein Objekt über einen anderen Konstruktor erzeugt werden, welches dann kopiert werden soll.

```
COrt2 ort1 = new COrt2(3,6.5);
COrt2 ort2 = new COrt2(ort1);
```

Mit diesen Anweisungen werden die folgenden beiden Objekte erzeugt:

| ort1:COrt2 |
|---|
| x = 3<br>y = 6.5 |

| ort2:COrt2 |
|---|
| x = 3<br>y = 6.5 |

Nicht nur Konstruktoren, auch andere Methoden können überladen werden. Wir wollen nun der Klasse *COrt2* aus Beispiel 5.1 eine Methode hinzufügen. Diese Methode soll die Aufgabe haben, den Punkt, der durch den Ortsvektor beschrieben wird zu verschieben. Dies können wir auf zwei verschiedene Arten tun:

1. Verschiebe den Punkt um vorgegebene Werte in x- und in y-Richtung
2. Verschiebe den Punkt in x- und in y-Richtung um den gleichen Betrag

Die Operationsbeschreibungen für beide Varianten lauten

1. Name: verschieben
   Eingabe: Verschiebung in x-Richtung und Verschiebung in y-Richtung
   Rückgabe: keine
   Funktionsweise: Die x-Koordinate (also der Wert des Attributes *x*) wird um die Verschiebung in x-Richtung erhöht. Die y-Koordinate (also der Wert des Attributes *y*) wird um die Verschiebung in y-Richtung erhöht.
2. Name: verschieben
   Eingabe: Wert der Verschiebung
   Rückgabe: keine
   Funktionsweise: Beide Koordinaten werden um die Verschiebung erhöht.

Die Operationen in Java-Quelltext lauten dann:

```
public void verschieben(double v_x, double v_y)
{
     x = x + v_x;
     y = y + v_y;
}
public void verschieben(double v)
{
     x = x + v;
     y = y + v;
}
```

Rufen Sie die Methode *verschieben* auf, so erkennt der Compiler anhand der übergebenen Argumente, welche der beiden Methoden ausgeführt werden muss. Eine überladene Methode wird genauso behandelt, wie eine Methode mit einem anderen Namen. Der Vorteil ist, dass Sie Methoden mit der gleichen Funktionalität auch gleich benennen können.

## 5.5 Methodenaufrufe

Sie haben nun schon einige Methoden programmiert. Sie haben diese Methoden auch aufgerufen, indem Sie in BlueJ ein Objekt auf der Objektleiste erzeugt und aus dem Kontextmenü des Objekts die passende Methode ausgewählt haben. Dabei handelt es sich um einen **externen Methoden-aufruf.** In Kapitel 3.4 ist beschrieben, dass ein solcher externer Methodenaufruf über den Punkt-Operator erzeugt wird. In Beispiel 5.2 finden wir mehrere solcher externen Methodenaufrufe, z.B.

```
mitte.SetX(x);
```

Wir befinden uns hier in der Klasse *CKreis, mitte* ist ein Objekt der Klasse *COrt2*.

Um also innerhalb einer Methode einer Klasse eine Methode einer anderen Klasse aufzurufen, benötigt man ein Objekt dieser Klasse und kann dann über den Punkt-Operator auf die Methode zugreifen. Diesen Methodenaufruf nennt man **externen Methodenaufruf.**

Wie sieht es nun aus, wenn wir eine Methode aufrufen wollen, die in der selben Klasse steht? Betrachten wir dazu nochmal die zweite Variante der Methode *verschieben* aus Kapitel 5.4. Wir erhalten das gleiche Ergebnis, wenn wir die erste Variante mit zwei gleichen Argumenten aufrufen. Die zweite Variante der Methode *verschieben* kann damit kürzer geschrieben werden:

```
public void verschieben(double v)
{
     verschieben(v,v);
}
```

Wir rufen innerhalb der Methode eine andere Methode der gleichen Klasse auf. Dabei reicht es, den Methodennamen mit Argumenten direkt hinzuschreiben. Diesen Methodenaufruf nennt man **internen Methodenaufruf.**

Betrachten wir noch einmal genau den Aufruf der Methode *verschieben* und insbesondere, welche Variablen wann existieren und welche Werte sie annehmen.
Quellcode:

| ***Quellcodezeile*** | ***intern durchlaufend*** | ***erzeugte Var. und ihre Werte*** |
|---|---|---|
| `COrt2 o = new COrt2(5,3);` | Parameterübergabe<br>`x = 5`<br>`y = 3`<br>Quellcodezeilen Konstr.<br>`this.x = x`<br>`this.y = y` | Lokale Parameter<br>`x = 5`<br>`y = 3`<br>Attribute<br>`x = 5`<br>`y = 3` |
| `double v0 = 2;` | `v0 = 2` | lokale Variable<br>`v0 = 2` |
| `o.verschieben(v0);` | Parameterübergabe<br>`v =  v0`<br>Aufruf verschieben(v,v)<br>`v_x = v`<br>`v_y = v`<br><br>`x = x + v_x`<br>`y = y + v_y` | Parameter<br>`v = 2`<br><br>`v_x = 2`<br>`v_y = 2`<br>Attributsänderungen<br>`x = 7`<br>`y = 5` |

Beim Aufruf einer Methode werden den Parametern Werte übergeben (Argumente), die innerhalb der Methode wie lokale Variablen verwendet werden. Werden die Werte der Parameter innerhalb der Methode verändert, hat dies keine Auswirkung auf die Werte der Argumente. Das folgende Beispiel soll dies verdeutlichen.
Beispiel 5.3

```
/**
* Diese Klasse zeigt, dass Veränderungen an den Parametern
* nicht nach außen wirken.
*/
public class Test
{ // Beginn Klasse Test
     public void parameterTest(int z)
     {
          z = z+4;
     }
}
public class Test2
{ // Beginn Klasse Test2
     public void teste()
     {
          Test test = new Test();
          int zahl = 5;
          test.parameterTest(zahl);
          System.out.println("zahl: " + zahl);
     } }
```

Was passiert nun innerhalb der Methode *teste()?* Als erstes wird das Objekt *test* angelegt. Danach wird Speicherplatz für die Variable *zahl* reserviert und dort der Wert 5 abgelegt. Danach wird die Methode *parameterTest* der Klasse Test aufgerufen. Dabei wird intern Speicherplatz für den Parameter *z* angelegt und die Zuweisung „ *z=zahl*“ ausgeführt. Nun existieren 2 Variablen im Speicher, die Variablen *zahl* und *z*, beide Variablen haben den Wert 5. Als nächstes wird die Anweisung im Rumpf der Methode *parametertest* „*z=z+4*“ ausgeführt, d.h. der Wert der Variablen *z* wird um 4 erhöht. Anschließend steht in der Variablen *z* der Wert 9. An dem Wert in der Variablen *zahl* hat sich dadurch nichts verändert, er beträgt immer noch 5.

**Änderungen an Parametern innerhalb einer Methode wirken nicht nach außen!**

***Fragen:***

- ✗ Was versteht man unter der Sichtbarkeit einer Variablen?

- ✗ Was versteht man unter der Lebensdauer einer Variablen?
- ✗ Wozu wird „this“ verwendet?
- ✗ Woran erkennt der Compiler bei überladenen Methoden, welche Methode durchgeführt werden soll?
- ✗ Wie erfolgt ein interner Methodenaufruf?
- ✗ Wie erfolgt ein externer Methodenaufruf?

# 6 Schleifen und Arrays

***Lernziele:***

- **Wissen**
  - Wissen, welche Schleifen es in Java gibt
- **Verstehen**
  - Verstehen, wie eine for-Schleife programmiert wird
  - Verstehen, wie eine while-Schleife programmiert wird
  - Den Unterschied zwischen einer for-each- und einer for-Schleife erklären können
  - Verstehen, wie ein Array programmiert wird
  - Die API-Dokumentation der Klassenbibliothek lesen können
- **Anwenden**
  - Unterschiedliche Schleifentypen ineinander umwandeln können
  - Geschachtelte Schleifen programmieren können
  - Ein- und mehrdimensionale Arrays erzeugen und verwenden können
  - Mit den Klassen String und StringBuilder arbeiten können

## 6.1 Schleifen

Schleifen werden verwendet, wenn die gleichen Anweisungen mehrfach ausgeführt werden sollen. Im Flussdiagramm und Struktogramm kann dies folgendermaßen dargestellt werden:

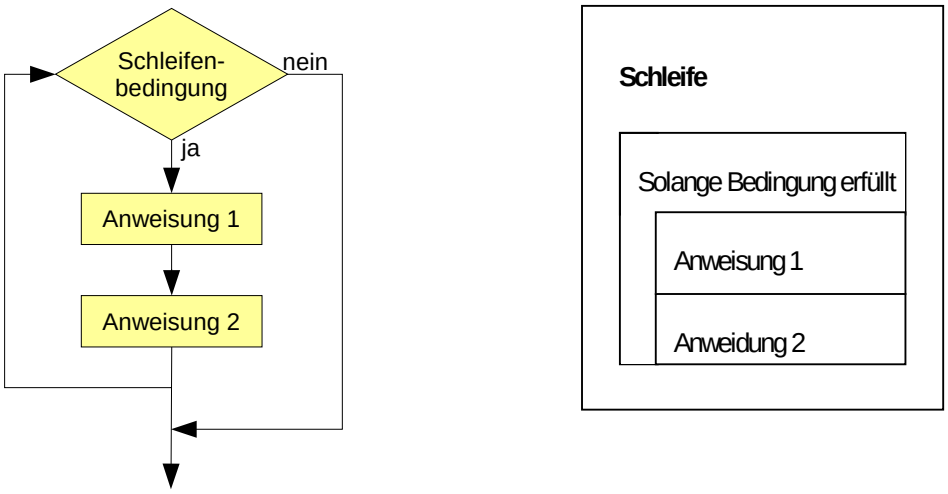



Abbildung 6.1: Flussdiagramm und Struktogramm für Schleifen

Solange die Schleifenbedingung erfüllt ist, werden die Anweisungen 1 und 2 wiederholt ausgeführt. Sobald die Bedingung nicht (mehr) erfüllt ist, wirde die Ausführung der Anweisungen 1 und 2 abgebrochen. Es gibt verschiedene Schleifenkonstruktionen in Java. Wir betrachten als erste die recht einfach aufgebaute **while-Schleife**, später noch die for-Schleife und die foreach-Schleife.

## 6.2 Die while-Schleife

Eine while-Schleife ist mit einer Bedingung, der sogenannten **Schleifenbedingung,** verknüpft. Solange die Schleifenbedingung erfüllt ist wird der Schleifenrumpf ausgeführt. Sobald die Schleifenbedingung nicht mehr erfüllt ist kommt es zu keiner Wiederholung des Schleifenrumpfs mehr. Die Java-Schreibweise lautet:

```
while (Schleifenbedingung)
{
     Schleifenrumpf
}
```

***Beispiele für eine while-Schleife***

Nehmen wir an, Sie wollen alle natürlichen Zahlen addieren, solange ihre Summe 50 nicht überschreitet. Der erste Lösungsansatz für dieses Problem ist im folgenden Beispiel umgesetzt.
Beispiel 6.1

```
int summe = 0;
int i = 1;
while (summe <= 50)
{
     summe += i;
     System.out.println(i + ": "+ summe);
     i++;
}
```

Die Ausführung dieser Schleife führt zu der Ausgabe
1: 1
2: 3
3: 6
4: 10
5: 15
6: 21
7: 28
8: 36
9: 45
10: 55

Haben wir unser Ziel erreicht? Wie müssen wir die Bedingung ändern, damit die letzte Zahl nicht mehr addiert wird und die Summe unter 50 bleibt?

```
while (_____________________ <= 50)
```

Beispiel 6.2:
Als zweites Beispiel wollen wir **Euklids Algorithmus** zur Bestimmung des größten gemeinsamen Teilers (ggT) zweier gegebener positiver ganzer Zahlen *m* und *n* programmieren. Wir wählen zunächst die größere der beiden Zahlen und teilen sie durch die kleinere. Erhalten wir bei der Division keinen Rest ist der Algorithmus beendet und die kleinere Zahl ist der ggT. Ansonsten wird das Verfahren wiederholt mit dem Divisionsrest als kleinere Zahl und dem Divisor(Teiler) als größere Zahl.

```
public class Euklid
{
     public int ggT(int m, int n)
     {
          int r;
          r = m%n;
          while (r>0)
          {
             m = n;
             n = r;
             r = m%n;
          }
          return n;
     }
     public Euklid(int zahl1, int zahl2)
     {  // Aufruf Methode ggt
          int ggt = ggT(zahl1,zahl2);
          System.out.print("ggT von " + zahl1 + " und " + zahl2);
          System.out.println(" ist " + ggt);
     }
}
```

Was passiert, wenn *m* kleiner als *n* ist?

### *Geschachtelte Schleifen*

Schleifen können geschachtelt werden, d.h. eine Schleife kann innerhalb einer anderen Schleife stehen.
Ein Beispiel dafür ist die Ausgabe des kleinen Einmaleins:
Beispiel 6.3:

```
int k = 1;
int l;
while (k<=10) //äußere Schleife
{
     l = 1;
     System.out.print(k + ":");
     while (l<=10)//innere Schleife
     {
          System.out.print(l*k + " , ");
          l++;
     }
     System.out.println();
     k++;
}
```

Wie oft wird der Rumpf der inneren Schleife durchlaufen? Nach wieviel Durchläufen wird k erhöht?

## 6.3 Die for-Schleife

Betrachten wir das letzte Beispiel, so fällt auf, dass die Anzahl der Schleifendurchläufe zu Beginn bekannt und fest ist. Für solche Fälle eignet sich die **for-Schleife**, die eine sehr kompakte Schreibweise besitzt.
Die allgemeine Form einer for-Schleife lautet

```
for (Initialisierung;Bedingung;Anweisung nach dem Rumpf)
{
     // Rumpf = Anweisungen, die wiederholt werden sollen
}
```

Die Ausführung folgt dann dem folgenden Ablauf:

- Initialisierung
- Überprüfen der Bedingung
- Ausführen des Rumpfes, wenn die Bedingung erfüllt ist
- Anweisung nach dem Rumpf
- Überprüfen der Bedingung
- Ausführen des Rumpfes, wenn die Bedingung erfüllt ist
- Anweisung nach dem Rumpf
- .....

Dies wird so lange wiederholt, bis die Schleifenbedingung nicht mehr erfüllt ist. Daraus folgt, dass entweder die Anweisung nach dem Rumpf oder die Anweisungen im Rumpf dazu führen sollten, dass nach einer gewissen Zahl von Schleifendurchläufen, die Bedingung nicht erfüllt ist.
Ist dies nicht der Fall, entsteht eine sogenannte Endlos-Schleife.
Mit einer for-Schleife können wir z.B. alle Zahlen von 1 bis 50 auf dem Bildschirm ausgeben:

```
for (int i=1;i<=50;i++)
{
     System.out.println(i);
}
```

Das Einmaleins lässt sich auch mit einer for-Schleife ausgeben:

Beispiel 6.4:

```
for
{
     System.out.print(k + ":");
     for

     {
          System.out.print(l*k + " , ");
     }
     System.out.println();
}
```

Schleifen eignen sich sehr gut zum Suchen einer Zahl in einer Liste von Zahlen oder eines Buchstabens in einem Text. Um diese Anwendungen von Schleifen zu zeigen, werden wir uns in den nächsten Kapiteln mit Arrays und mit der Java-Bibliotheksklasse String beschäftigen.

## 6.4 Arrays

Bisher definierten wir Variablen einzeln, um Informationen zu speichern. Sollen größere Mengen an Informationen gespeichert werden, müsste man sehr viele solcher Variablen definieren. Mit Hilfe von Arrays können wir eine große Anzahl gleichartiger Informationen in einer Variablen speichern. Arrays werden auch als Container für eine Anzahl Elemente eines anderen Typs bezeichnet. Die Länge eines Arrays wird zu Beginn festgelegt und kann dann nicht mehr verändert werden. Der **Arraytyp** bezieht sich auf den Datentyp der Elemente, die in dem Array gespeichert werden sollen. Zu jedem Datentyp in Java gibt es einen korrespondierenden Arraytyp.
Ein Array wird in Java folgendermaßen deklariert: Datentyp [] name, z.B. `int [] werte;`
Hierbei steht das eckige Klammerpaar [] für das Array, der Datentyp der Elemente, die in dem Array gespeichert werden sollen wird vor die eckigen Klammern geschrieben, der Variablenname dahinter. In obigem Beispiel wurde also eine Variable mit dem Namen *werte* deklariert, in der ein Array von ganzen Zahlen gespeichert werden kann. Das Klammerpaar [] darf auch hinter dem Variablnenamen stehen (`int werte[];`).
Analog bezeichnet *double[]* ein Array von Gleitpunktzahlen, *boolean[]* ein Array von Wahrheitswerten, usw..
Als zweiter Schritt muss bei einem Array festgelegt werden, wie viele Elemente in ihm gespeichert werden sollen. Da ein Array in Java eine Klasse ist, erfolgt dies mit *new*:

```
werte = new int[50];
```

In *werte* können nun 50 ganze Zahlen gespeichert werden. Zugreifen kann man auf die einzelnen Array-Elemente, indem man den Namen des Arrays angibt und dahinter den Index in den eckigen Klammern:

```
werte[5]
```

bezeichnet somit das 6. Element des Arrays **(Achtung: Das erste Element hat den Index 0)**.
Für die Initialisierung des Integer-Arrays (das heißt der Belegung mit Werten), können die Werte direkt in die Feldelemente geschrieben werden:

```
werte[0] = 1;
werte[1] = 2;
werte[2] = 3;
...........
```

Einfacher können die Werte mit Hilfe einer for-Schleife in dem Array gespeichert werden:

```
for (int i=0;i<50;i++)
{
     werte[i] = i+1;
}
```

In vielen Fällen lässt sich der Wert, der im Array gespeichert werden soll, nicht so einfach aus dem Arrayindex herleiten. Dann ist es sinnvoll mit einer zusätzlichen Variable für den Wert zu arbeiten. In folgendem Beispiel sollen alle ungeraden Zahlen von 25 bis 63 in einem Array gespeichert werden. Dies kann folgendermaßen programmiert werden.

```
int[] werte2 = new int[20];
int a = 25;
for (int i=0;i<20;i++)
{
     werte2[i] = a;
     a = a+2;
}
```

Wichtig hierbei ist, dass der Schleifenzähler *i* mit dem Arrayindex übereinstimmt und der zu speichernde Wert *(hier a)* im Schleifenrumpf erhöht wird (*a=a+2;*).

### *Explizite Initialisierung*

Arrays können auch direkt mit Werten belegt werden. Dabei wird die Länge des Arrays durch die Anzahl der vorgegebenen Werte festgelegt und muss nicht explizit angegeben werden. Das Array

```
int[] a = new int[] {5,8,24,0,-63,4296};
```

hat die Länge 6. In die geschweiften Klammern werden die Werte getrennt durch Kommata eingetragen und die eckige Klammer bleibt leer.
Die Länge eines Arrays kann über das öffentliche Attribut (public) **length** abgefragt werden:

```
int laenge = a.length;
```

### *Die foreach-Schleife*

Da es sehr häufig vorkommt, das ein Array vollständig durchlaufen wird, gibt es dafür eine spezielle Art von Schleife, die **foreach**-Schleife. Wir werden sie der for-Schleife gegenüberstellen. Es sollen alle Elemente des Arrays *a* von oben auf dem Bildschirm ausgegeben werden. Mit einer for-Schleife sieht dies folgendermaßen aus:

```
for (
     System.out.println(          ;
```

Eleganter lässt sich dies mit einer foreach-Schleife formulieren:

```
for (int e:a)
     System.out.println(e);
```

Allgemein formuliert hat die foreach-Schleife die folgende Form:

```
for (Arraytyp Variablenname : Arrayname)
```

Hinter dem : steht das Array, welches durchlaufen werden soll und vor dem : die Deklaration einer Variablen mit bel. Namen vom Datentyp der Arrayelemente.
Der Vorteil dieser Schleife liegt insbesondere darin, dass keine Indexvariable benötigt wird (sie wird vom Laufzeitsystem implizit verwaltet). Das Array *a* wird sequentiell vom ersten bis zum letzten Element durchlaufen. In jedem Schleifendurchlauf wird das entsprechende Element in der Variablen *e* gespeichert, welche dann im Schleifenrumpf verwendet wird.

### *Suchen in einem Array*

Wir wollen nun in einer Liste von Zahlen, die erste durch 3 teilbare Zahl suchen. Beim Suchen soll eine Schleife so lange wiederholt werden, bis die gesuchte Zahl gefunden wurde. Daher eignet sich die while-Schleife am besten. Um zu speichern, wann die gesuchte Zahl gefunden wurde, ist es hilfreich eine Hilfsvariable vom Typ *boolean* zu definieren. Sei die folgende Liste von Zahlen gegeben:

```
int [] liste = new int[]{5,1,22,8,25,18,4,9,28,30};
```

Dann lässt sich die erste durch 3 teilbare Zahl folgendermaßen finden:

```
boolean gefunden=false;
int index = 0;
while (                                  &&
                  )
{
     if (                                        )
          gefunden = true;
     index ++;
}
if (gefunden)
     System.out.println("Die erste durch 3 teilbare Zahl
                              lautet: " + liste[index-1]);
```

Möchte ich zählen, wie viele durch 3 teilbare Zahlen in der Liste sind, so eignet sich auch die foreach-Schleife:

```
int zaehler=0;
for (int e:liste)
{
     if (e%3 == 0)
          zaehler++;
}
System.out.println(" Es sind "+zahler+" durch 3 teilbare
                    Zahlen in der Liste");
```

### *Mehrdimensionale Arrays*

Da es zu jedem Java-Typ einen Arraytyp gibt, gilt dies auch für jedes Array, das heißt es gibt ein Array von Arrays. Ein solches Array wird als geschachteltes oder mehrdimensionales Array bezeichnet. Die Deklaration ergibt sich direkt aus dieser Definition. Ein zweidimensionales int-Array wird folgendermaßen deklariert:

```
int[][] m;
m = new int[2][3];
```

Das Array `m` hat als Array vom Typ `int[]` also die Länge 2. Jedes Element von `m` ist selber wieder ein `int`-Array der Länge 3. Insgesamt können in `m` also 2*3 = 6 `int`-Zahlen gespeichert werden.
Solche mehrdimensionalen Arrays werden meist mit Hilfe geschachtelter **for**-Schleifen mit Elementen gefüllt. Für obiges Array zum Beispiel mit

```
for (int i=0;i<2;i++)
     for (int j=0;j<3;j++)
          m[i][j]=5*i+j;
```

Können mehrdimensionale Arrays auch mit der for-each-Schleife durchlaufen werden? Ja, wie wird schnell klar, wenn man sich bewusst macht, dass `m` ein Array vom Typ `int[]` ist und `m[i]` ein Array vom Typ `int`. Die äussere for-each-Schleife hat also den Elementtyp `int[]`, die innere for-each-Schleife den Elementtyp `int`:

```
for (int[]a:m)
   for (int e:a)
     System.out.println(e);
```

Mit zwei geschachtelten for-Schleifen sieht die Ausgabe folgendermaßen aus.

```
for (int i=0;i<m.length;i++)
     for (int j=0;j<m[i].length;j++)
          System.out.println(m[i][j]);
```

Entscheiden Sie selbst, welche Variante Sie bevorzugen.
Auch mehrdimensionale Arrays dürfen direkt initialisiert werden. Das Array *m* von oben kann direkt mit

```
int[][] m = new int[][] {{0,1,2},{5,6,7}};
```

erzeugt werden.
Drei- und höherdimensionale Arrays werden analog definiert und verwendet. Mehrdimensionale Arrays müssen nicht rechteckig sein. Jedes Array im Hauptarray kann eine andere Länge haben.

### *Kopieren von Arrays*

Beim Kopieren von ein- oder mehrdimensionalen Arrays müssen Sie stets darauf achten, ob Sie lediglich die Referenz kopieren oder jedes Element. Um eine echte Kopie zu erzeugen, muss ein neues Array der gleichen Länge angelegt werden und jedes Element in das neue Array kopiert werden.

## 6.5 Arrays von Objekten

Betrachten wir die einfache Klasse *CZahl*.

```
public class CZahl
     {
       private int z;
       public CZahl(int zahl)
       {
         z = zahl;
       }
```

50 Objekte der Klasse *CZahl* können in einem Array gespeichert werden. Die Deklaration und die Festlegung der Länge des Arrays erfolgt analog zu Kapitel 6.4:

```
CZahl[] Zahlen = new CZahl[50];
```

In dem Array *Zahlen* werden nun keine ganzen Zahlen, sondern Objekte der Klasse *CZahl* gespeichert. Bei der Verwendung eines solchen Arrays muss wieder beachtet werden, dass die Objekte als Referenzen implementiert sind. Der Unterschied zwischen den beiden Deklarationen ist folgender:

Wenn Sie ein Array von n Elementen eines einfachen Datentyps deklarieren, z.B. *double*-Werte, dann sind im Array direkt n *double*-Werte enthalten. Wenn Sie ein Array von n Objekten einer Klasse definieren, dann enthält jedes Element eine Null-Referenz. Sie müssen den einzelnen Elementen erst Objekte zuweisen und damit Speicherplatz bereitstellen.

Für die Initialisierung des Objekt-Arrays müssen die einzelnen Elemente erst mit *new* erzeugt werden:

```
for (int i=0;i<50;i++)
{
    Zahlen[i] = new CZahl(i+1);
}
```

Betrachten wir als zweites Beispiel die Klasse `Ticketautomat` aus Kapitel 3.

Beispiel 6.5:

```
public class Ticketautomat
{
    private int preis;
    private int bisherGezahlt;
    private int gesamtsumme;
    public Ticketautomat(int ticketpreis)
    {
        preis = ticketpreis;
        bisherGezahlt = 0;
        gesamtsumme = 0;
    }
} // Ende Klasse Ticketautomat
```

Nun bilden wir ein Array und füllen es mit 3 Ticketautomaten mit den Ticketpreisen 180 Cent, 250 Cent und 360 Cent.

```
public class Anwendung
{
      public Anwendung()
      {
           Ticketautomat[] automaten;
           automaten = new Ticketautomat[3];
           automaten[0] = new Ticketautomat(180);
           automaten[1] = new Ticketautomat(250);
           automaten[2] = new Ticketautomat(360);
      }
}
```

## 6.6 Die Klassen String und StringBuilder

Im folgenden wollen wir zwei weitere Klassen aus der Standardklassenbibliothek von Java und ihre Anwendungsmöglichkeiten betrachten.

### *Die Klasse String*

Strings sind Folgen von Zeichen. Wie bei einem Array, gibt es auch bei der Klasse *String* syntaktische Besonderheiten. In Java ist jeder in Anführungszeichen eingeschlossene String ein Objekt der Klasse *String*. Daher kann ein Objekt der Klasse *String* auch ohne den *new*-Operator erzeugt werden:

```
String zug = "Eisenbahn";
```

Objekte der Klasse *String* können mit dem +-Operator verkettet werden:

```
zug + " fährt";
```

ergibt die Zeichenkette "Eisenbahn fährt".
Es gibt einen Unterschied zwischen einer leeren und einer null-Zeichenkette:

```
String s1;
String s2 = "";
```

Die Zeichenkette von *s2* ist leer, aber es wurde Speicherbereich für die Zeichenkette bereitgestellt. Für *s1* wurde kein Speicherbereich bereitgestellt, d.h. s1 verweist nicht auf ein konkretes Objekt, man sagt auch *s1* hat den Wert *null*. Versucht man auf *s1* zuzugreifen, so erhält man eine Fehlermeldung vom Compiler (*variable s1 might not have been initialized*).
Aus einem String lässt sich ein Teilstring extrahieren, mit den Methoden:

- `public String substring(int beginIndex, int endIndex)`
- `public String substring(int beginIndex)`

Das zweite Argument gibt die erste Position an, die nicht zu kopieren ist. Wie bei Arrays hat das erste Zeichen in einem *String* die Position 0, also gibt

```
String eis = zug.substring(0,3);
```

die Zeichenkette "Eis" zurück und

```
String bahn = zug.substring(5);
```

die Zeichenkette "bahn".
Mit der Methode

```
public boolean contains(CharSequence s)
```

kann überprüft werden, ob die Zeichenfolge *s* in der Zeichenkette enthalten ist. Der Aufruf

```
zug.contains("Eis")
```

liefert das Ergebnis *true.*
Die Länge einer Zeichenkette läßt sich mit der Methode

```
public int length()
```

ermitteln. Dabei ist zu beachten, dass es sich hierbei um die Methode *length* handelt und nicht wie bei einem Array um ein Attribut. Beachten Sie die Unterschiede bei der Verwendung:

```
String la = "Laenge";
double[] f =new double[4];
int l1,l2;
l1 = la.length();
l2 = f.length;
```

Die Objekte der Klasse *String* werden als unveränderlich bezeichnet, d.h. es gibt in Java keine Methoden, mit denen man Zeichen in einem vorhandenen String ändern kann. Allerdings kann der Inhalt der String-Variablen geändert werden. Möchte man z.B. den Text "Eisenbahn" in "Eisbahn" abändern, muss man folgendermaßen vorgehen:

```
zug = zug.substring(0,3) + zug.substring(5);
```

Dabei wird jedoch neuer Speicherplatz angelegt.
Mittels der beiden Methoden

```
public boolean equals(Object o)
public boolean equalsIgnoreCase (String s)
```

lassen sich zwei Strings auf Gleichheit testen, im zweiten Fall ohne Unterscheidung der Groß-/Kleinschreibung. Damit können wir z.B. in einem Array von Zeichenketten ein bestimmtes Wort suchen. Beachten Sie, dass Strings **nicht** mit == verglichen werden können! Dabei würde nur die Speicheradresse auf Gleichheit überprüft.
Es gibt noch weitere nützliche Methoden, um mit Zeichenketten zu arbeiten, die in der API-Dokumentation beschrieben sind.
Mit Hilfe einer Schleife und der Methode

```
public char charAt(int index)
```

kann ein Buchstabe in einem String gesucht werden:

```
int i=0;
while (i<zug.length() && 'b' != zug.charAt(i))
       i++;
```

Finden Sie eine Methode in der Klassenbibliothek, mit der ein Buchstabe direkt in einem String gesucht werden kann!

### *Die Klasse StringBuilder*

Wenn Zeichenketten häufig verändert werden, sollte man nicht die Klasse *String* verwenden, sondern zum Beispiel die Klasse **StringBuilder.**
Die Klasse StringBuilder legt die Zeichen in einem internen Buffer ab, der beliebig beschreibbar und erweiterbar ist. Dadurch wird ein schnelles Aneinanderketten von Zeichen ermöglicht. Dem Konstruktor der Klasse kann die benötigte Größe übergeben werden.
Einige wichtige Methoden der Klasse *StringBuilder*:

| | |
|---|---|
| int length() | Anzahl der Zeichen |
| int capacity() | maximale Anzahl von Zeichen |
| char charAt(int) | lese Zeichen an gegebener Position |
| void SetCharAt(int,char) | schreibe Zeichen |
| StringBuilder insert(int,String) | füge String ein |
| StringBuilder append(String) | hänge String an |
| String toString() | Umwandlung in ein String-Objekt |

Weitere Methoden finden Sie in der Java-Bibliothek!

***Fragen:***

- ✗ Wozu dient eine Schleife?
- ✗ Was steht bei einer while-Schleife in den Klammern hinter dem Schlüsselwort while?
- ✗ Was steht bei einer for-Schleife in den Klammern hinter dem Schlüsselwort for?
- ✗ Welches Schlüsselwort wird benötigt, um ein Array anzulegen?
- ✗ Wie wird auf das 3. Element des Arrays *list* zugegriffen?
- ✗ Was ist der Elementtyp eines zweidimensionalen Arrays?
- ✗ Wie viele Elemente eines Arrays werden mit der foreach-Schleife durchlaufen
- ✗ Mit welcher Methode können zwei *Strings* auf Gleichheit überprüft werden?
- ✗ Mit welcher Methode kann eine Zeichenkette in ein Objekt vom Typ *StringBuilder* eingefügt werden?
- ✗ Worin besteht der Unterschied zwischen der Bestimmung der Länge eines Arrays und der eines Strings?
- ✗ Was muss bei Arrays von Objekten im Unterschied zu Arrays von einfachen Datentypen beachtet werden?

# 7 Klassen und Objekte in Java, Vertiefung 2

***Lernziele:***

- **Wissen**
  - Wissen, mit welchem Schlüsselwort ein Klassenattribut definiert wird
- **Verstehen**
  - Erklären können, auf welche Attribute und Operationen eine statische Methode zugreifen kann
  - Erklären können, wie eine Anwendung ohne Verwendung von BlueJ gestartet werden kann
  - Erklären können, was Objekte als Parameter von primitiven Variablen unterscheidet
- **Anwenden**
  - Objekte als Parameter von Operationen einsetzen können.

## 7.1 Klassenattribute

In der Analyse haben wir
- **Objektattribute** und
- **Klassenattribute**

unterschieden (siehe Kapitel 2.3). In Java werden die Klassenattribute mit dem Schlüsselwort **static** definiert und auch **statische Attribute** genannt. Im folgenden betrachten wir zwei Beispiele zur Verwendung von Klassenattributen:

Beispiel 7.1: Konstante statische Attribute (auch Klassenkonstanten genannt)

```
public class Kreis0
{
     public static final double PI = 3.14159;
     private double r;
     // Konstruktor
     public Kreis0(double radius)
     {
         r = radius;
     }
} // Ende Klasse Kreis0
```

Der Zugriff auf die statischen Attribute kann (wenn sie public dekalriert sind)
- direkt über den Klassennamen erfolgen: `double pi1 = Kreis0.PI;`
- über ein Objekt erfolgen: `Kreis0 k = new Kreis0(5);`
  `double pi2 = k.PI;`

Beispiel 7.2: Veränderbare statische Attribute:

```
    class CVorlesung
    {
      private static int anzahl = 0; // Anzahl an Vorlesungen
      private String name;
      public CVorlesung(String name)
      {
        this.name = name;
        anzahl++;
       }} // Ende Konstruktor und Klasse CVorlesung
```

Erzeugen Sie in BlueJ mehrere Objekte der Klasse *CVorlesung* und schauen Sie sich den Wert der statischen Variablen *anzahl* mithilfe des Objektinspektors für die verschiedenen Objekte an.

Innerhalb einer Klasse gibt es drei verschiedene Arten von Variablen:

| | |
|---|---|
| **Lokale Variablen** | Variablen, die innerhalb einer Methode deklariert werden und nur dort gültig sind; auch Parameter haben nur methodenweite Sichtbarkeit |
| **Objektattribute** | Variablen, die außerhalb jeder Methode deklariert werden, jedes Objekt der Klasse erhält eine eigene Kopie |
| **Klassenattribute** | Variablen, die außerhalb jeder Methode deklariert werden, es gibt nur eine Kopie für alle Objekte der Klasse |

## 7.2 Klassenoperationen

Auch Methoden können als *static* deklariert werden. Sie werden dann **statische Methoden** oder **Klassenoperationen** (siehe Kapitel 2.4) genannt. Auf statische Methoden kann zugegriffen werden, ohne dass ein Objekt der Klasse existieren muss.
Eine statische Methode kann nur auf Klassenattribute zugreifen, nicht auf Objektattribute. Sie kann auch nur andere statische Methoden der Klasse aufrufen.
Beispiel 7.3

```
public class Kreis0
{
     public static final double PI = 3.14159;
     private double r;
     // Konstruktor
     public Kreis0(double radius)
     {
         r = radius;
     }
     public static double WinkelInGrad(double phi)
     {
        double phig;
        phig = phi*180.0/PI;  // Umrechnung in Grad
        return phig;
     }
} // Ende Klasse CKreis
```

Programmieren Sie diese Klasse in BlueJ. Wie können Sie die Methode *WinkelInGrad* aufrufen? Versuchen Sie innerhalb der Methode *WinkelInGrad* das Attribut *r* zu verwenden. Was passiert?

## 7.3 Java-Bibliotheksklassen

Die Standardklassenbibliothek von Java enthält viele Klassen, die sehr nützlich sind. Es ist wichtig zu wissen, wie man sich in der Bibliothek zurechtfindet, passende Klassen findet und ihre Methoden anwenden kann. In der API-Dokumentation (Application Programming Interface) finden Sie die Beschreibungen dieser Klassen mit ihren Attributen und Methoden, im Internet unter

*http://download.oracle.com/javase/7/docs/api/index.html*

Starten Sie die API-Dokumentation aus BlueJ über den Eintrag *Java Class Libraries* aus dem Menü *Help*. Der Webbrowser zeigt drei Bereiche. Links oben sehen Sie die Liste aller Pakete, darunter die Liste aller Klassen und rechts Detailinformationen über das ausgewählte Paket bzw. die ausgewählte Klasse. Wählen Sie beispielsweise die Klasse *String* aus.
Die Dokumentation liefert u.a. die folgenden Informationen, die als **Schnittstelle** der Klasse bezeichnet werden:

- ➔ den Namen der Klasse
- ➔ eine Beschreibung des Zwecks der Klasse
- ➔ eine Liste der Konstruktoren und Methoden der Klasse
- ➔ die Parameter und Ergebnistypen für jeden Konstruktor und jede Methode
- ➔ die Beschreibung des Zwecks jedes Konstruktors und jeder Methode

Der Quelltext, der eine Klasse definiert, wird als **Implementierung** bezeichnet. Zur Verwendung einer Bibliotheksklasse benötigen wir lediglich Informationen über die Schnittstelle.
Zum Verwenden der Klassen müssen Sie das jeweilige Paket mit der *import*-Anweisung zur Verfügung stellen. Das Paket *java.lang* enthält die grundlegenden Standardklassen, die Java zur Verfügung stellt. Dieses Paket muss nicht mit der *import*-Anweisung importiert werden, es wird automatisch in jedes Programm importiert.
In Java dürfen Werte von einfachen Datentypen nicht durch Objekte ersetzt werden oder umgekehrt. Manchmal benötigt eine Methode aus dem Standardpaket ein Objekt, man möchte aber z.B. einen Wert vom Typ *int* übergeben. Um dies zu realisieren gibt es in dem Paket *java.lang* für jeden einfachen Datentyp eine Klassenentsprechung. Diese Klassen heißen *Boolean, Character, Void, Byte, Double*, *Float, Integer, Short* und *Long*. Die Klassenbibliothek stellt Methoden zur Verfügung, um aus den Klassenobjekten die entsprechenden Werte als einfache Datentypen auszulesen, z.B.

*abstract double doubleValue()*

Im folgenden werden als Beispiel einige Methoden der Klasse *Integer* angegeben. Es gibt zwei Konstruktoren

*Integer(int i)*
*Integer(String s),*

d.h. man kann entweder aus einem *int*-Wert oder aus einem String (der eine ganze Zahl darstellen muss) ein Objekt der Klasse *Integer* erzeugen. Mit der Methode

*int intValue()*

erhält man den *int*-Wert und die folgenden Methoden konvertieren aus und in Zeichenketten

- *static Integer valueOf (String s)*
- *static int parseInt(String s)*
- *static String toString (int i)*
- *String toString()*

Die ersten drei dieser Methoden sind statisch, sie beziehen sich nicht auf den Wert eines *Integer*-Objektes und können direkt über ihren Klassennamen aufgerufen werden, z.B. in der Form

```
String s1="16";
int i1=Integer.parseInt(s1);
String s2 = Integer.toString(i1);
```

Die vierte Methode ist nicht statisch, daher wird ein Objekt der Klasse `Integer` benötigt, um die Methode aufzurufen:

```
int i1=16;
Integer I1 = new Integer(i1);
String s1 = I1.toString();
```

Nennen Sie die zwei Unterschiede zwischen den beiden Aufrufen der Methoden t`oString().`

## 7.4 Die Methode main

Bisher haben wir jede beliebige Methode starten können, indem wir ein Objekt (auf der Objektleiste von BlueJ) erzeugt und dann die Methode aufgerufen haben. Was muss man nun tun, wenn man ein Programm starten möchte ohne die Entwicklungsumgebung BlueJ zu verwenden? Welche Operationen kann man aufrufen ohne zuvor ein Objekt erzeugen zu müssen? _____
Dies bedeutet, die Startmethode für ein Programm muss eine _____ sein. Die Java-Definition zum Starten einer Anwendung ist die folgende: Der Benutzer gibt eine Klasse an, und das Java-System ruft die **main-Methode** dieser Klasse auf. Die main-Methode muss die folgende Signatur aufweisen:

```
public static void main(String[] arg),
```

wobei der Name `arg` des Parameters beliebig sein kann. In der Literatur wird häufig auch der Name `args` verwendet. Mit Hilfe des Parameters `arg` können mehrere Zeichenketten an das Programm übergeben werden.
Möchte man Zahlen als Eingabeparameter verwenden, so müssen diese als Zeichenketten eingegeben werden und anschließend in Zahlen umgewandelt werden. Dies kann z.B. mithilfe der in Kapitel 7.3 vorgestellten Methoden der Klassen *Integer, Double* oder *Float* erfolgen:

```
public class Anw_Test  { // Beginn Anwendungsklasse
  public static void main(String[] arg)
  {  // Beginn Methode main
    int z = Integer.parseInt(arg[0]);
    double d = Double.parseDouble(arg[1]);
    System.out.println(z + " " + d);
  } // Ende Methode main
} // Ende Anw_Test
```

Beim Übersetzen einer Klasse wird eine Datei mit dem Namen *Klassenname.class* erzeugt. Jedes Programm kann von der Kommandozeile mit dem Befehl *java Klassenname* gestartet werden. Nun wird die *main*-Methode der Klasse *Klassenname* ausgeführt. Damit Ihr Programm auch ohne die Entwicklungsumgebung BlueJ gestartet werden kann, sollten Sie eine solche Methode *main* ergänzen. In dieser Methode sollten die Befehle enthalten sein, die Sie sonst interaktiv in BlueJ ausführen, z.B. das Erzeugen eines Objekts und der Aufruf einer oder mehrerer Methoden. Alternativ können Sie auch eine zusätzliche Klasse, z.B. die Klasse *Anwendung*, erzeugen, die lediglich eine Methode *main* enthält.
Um eine Anwendung, die aus mehreren Klassen und somit mehreren Dateien besteht an Anwender weiterzugeben, können diese in einer Datei zusammengefasst werden. Dazu wird das *Java Archive Format (jar)* verwendet. Um eine solche *.jar*-Datei später ausführen zu können muss lediglich festgelegt werden, welche Klasse die Startklasse ist, d.h. die Klasse, deren *main*-Methode ausgeführt werden soll. In BlueJ kann man eine solche ausführbare jar-Datei erzeugen, indem man unter *Project* den Befehl *Create Jar File* ausführt, dabei kann die Startklasse ausgewählt werden. Als Beispiele betrachten wir zwei Beispiele aus Kapitel 6 und schreiben diese so um, dass sie über die main-Methode gestartet werden können.

Beispiel 7.4:

```
public class Euklid
{
  public static void main(String[] arg)
  {
     public static int ggT(int m, int n)
     {
          int r;
          r = m%n;
          while (r>0)
          {
               m = n;
               n = r;
               r = m%n;
          }
          return n;
     }
     public static void main(String args[])
     { // Beginn Methode main
          int z1 = Integer.parseInt(args[0]);
          int z2 = Integer.parseInt(args[1]);
          int ggt = ggT(z1,z2);
          System.out.print("ggT von " + zahl1 + " und " + zahl2);
          System.out.println(" ist " + ggt);
     } // Ende Methode main
} // Ende Klasse Euklid
```

Beispiel 7.5:

```
public class Ticketautomat
{
     private int preis;
     private int bisherGezahlt;
     private int gesamtsumme;
     public Ticketautomat(int ticketpreis)
     {
          preis = ticketpreis;
          bisherGezahlt = 0;
          gesamtsumme = 0;
     }
} // Ende Klasse Ticketautomat
```

Nun bilden wir ein Array und füllen es mit 3 Ticketautomaten mit den Ticketpreisen 180 Cent, 250 Cent und 360 Cent.

```
public class Anwendung
{
     public static void main(String arg[])
     {
          Ticketautomat[] automaten;
          automaten = new Ticketautomat[3];
          automaten[0] = new Ticketautomat(180);
          automaten[1] = new Ticketautomat(250);
          automaten[2] = new Ticketautomat(360);
     }
} // Ende Klasse Anwendung
```

## 7.5 Objekte als Parameter

In Kapitel 5.5 haben wir gezeigt, dass Änderungen an Parametern innerhalb einer Methode keine Auswirkungen auf die Werte der Argumente haben. Gilt dies nur für die einfachen Datentypen oder auch, wenn es sich bei dem Argument um ein Objekt handelt?
Bei Objekten als Parameter enthält der Parameter nicht das Objekt, sondern lediglich die Referenz (d.h. einen Verweis auf die Stelle im Speicher, an der das Objekt gespeichert ist). Es existiert damit in der Methode eine lokale Variable, die auf dasselbe Objekt verweist, wie das Argument. Die Auswirkungen dieser Situation soll an nachfolgendem Beispiel erläutert werden.

Beispiel 7.6

```
public class CZahl
 {
   private int z;
   public CZahl(int zahl)
   {
      z = zahl;
   }
   public int Z()
   {
      return z;
   }
   public void SetZ(int zneu)
   {
      z = zneu;
   }
   public void erhoeheZ()
   {
      z = z+1;
   }
 } // Ende Klasse CZahl
```

```
public class CParameter
{
  public void erhoehen(int para1,CZahl para2)
  {
      para1 = para1+1;
      para2.erhoeheZ();
  } // Ende Methode erhoehen

  public void tauschen(CZahl a, CZahl b)
  {
    CZahl temp = a;
    a = b;
    b = temp;
  } // Ende Methode tauschen
} // Ende Klasse CParameter
```

```
public class Parameter_Anwendung
{
  private CZahl zahl;
  private CParameter par;
  private int start;

    // Konstruktor
  public Parameter_Anwendung()
  { // Initialisierung der Attribute
      zahl = new CZahl(4);
      par = new CParameter();
      start = 4;
  }
  // Testen der Methode erhoehen der Klasse CParameter
  public void Erhoehen()
  {
    par.erhoehen(start,zahl);
    System.out.println("start: " + start);
    System.out.println(" zahl: " + zahl.Z());
  }
  // Testen der Methode tauschen der Klasse CParameter
  public void Tauschen()
  {
     CZahl zahl2 = new CZahl(7);
     System.out.print("Vor Tauschen zahl: " + zahl.Z());
     System.out.println(" zahl2: " + zahl2.Z());
     par.tauschen(zahl, zahl2);
     System.out.println("Nach Tauschen zahl: " + zahl.Z());
     System.out.println(" zahl2: " + zahl2.Z());
  }
} // Ende Klasse Parameter_Anwendung
```

Testen Sie die Methoden in BlueJ! Erhalten Sie die erwarteten Ergebnisse? Warum?
Ändern Sie die Methode *tauschen* so ab, dass die Attributwerte der Variablen *a* und *b* tatsächlich getauscht werden.

***Fragen:***

- ✗ Wie werden in Java Klassenattribute definiert?

- ✗ Welche Methode wird benötigt, um ein Programm ohne BlueJ starten zu können

# 8 Testen

***Lernziele:***

- **Wissen**
  - Einige Testmethoden kennen
- **Verstehen**
  - Den Unterschied zwischen einem Modultest und einem Akzeptanztest erklären können
  - Den Unterschied zwischen einem Black-Box- und einem White-Box-Test erklären können
- **Anwenden**
  - Sinnvolle Testfälle für Klassentests konstruieren können
  - Tests automatisieren können

## 8.1 Testen von Klassen und Anwendungen

Als Anwender wünschen wir uns ein fehlerfreies Programm. Dabei verstehen wir unter einem Fehler eine Abweichung von den zuvor festgelegten Anforderungen. Um diesem Ziel möglichst nahe zu kommen sind zwei Schritte notwendig, die **Validierung**, das heißt die Überprüfung, ob die Anforderungen an das Software-System erfüllt sind und die **Fehlerbeseitigung**. Zum Validieren von Software gibt es zwei Möglichkeiten:

**Verifikation:** Mit formalen Methoden wird anhand des Quelltextes gezeigt, dass ein Programm nur richtige Ergebnisse liefern kann.

**Testen:** Es werden verschiedene Fälle konstruiert, das Programm für diese Fälle durchgeführt und die Ergebnisse mit den erwarteten Ergebnissen verglichen.

Die Verifikation ist zwar dem Testen überlegen, in den meisten Fällen realer Problemstellungen jedoch zu aufwändig. Außerdem überlassen wir dieses Feld den Informatikern. Mit Testen können wir zwar nicht auf die Korrektheit von Programmen schließen, wir können jedoch Fehlermöglichkeiten ausschließen.

Unzureichendes Testen kann gravierenden Folgen haben, teuer sein und im schlimmsten Fall Menschenleben kosten. Beim Jungfernflug der Ariane 5 kam es im Juni 1996 39 Sekunden nach dem Start zur Selbstzerstörung. Grund dafür war nicht ausreichendes Testen der Software. Die Software für das Trägheitsnavigationssystem wird unverändert von der Ariane 4 übernommen. Ein Test dieser Software unterbleibt daher. Die übrigen Systeme der Rakete werden komponentenweise gründlich getestet. Ein gemeinsamer Test der gesamten Steuerungssoftware der Rakete unterbleibt aus Kosten-und Machbarkeitsgründen. In der Software für das Trägheitsnavigationssystem gibt es eine Vergleichsfunktion, deren Werte nur sinnvoll sind, solange die Rakete noch nicht fliegt.
Da die Ariane 5 eine andere Flugbahn hat als die Ariane 4, berechnet die Vergleichsfunktion einen Wert, der wesentlich größer ist als erwartet. Bei der Konvertierung dieses Werts von einer 64 Bit Gleitkommazahl in eine 16-Bit Festkommazahl tritt ein Überlauf ein, der Rechner erzeugt eine Ausnahmebedingung. Die Folge davon ist, dass der Selbstzerstörungsmechanismus anspricht.
Dieses Beispiel zeigt, wie wichtig systematisches Testen ist. Häufig testet der Entwickler selber und nur solange bis das Programm läuft und die Ergebnisse vernünftig aussehen. Die folgenden Anforderungen werden an systematisches Testen gestellt:

- Der Test ist geplant, eine Testvorschrift liegt vor
- Das Programm wird gemäß der Testvorschrift (= Testspezifikation) ausgeführt
- Die Ist-Resultate werden mit den Soll-Resultaten verglichen
- Alle Testergebnisse werden dokumentiert
- Fehlersuche und -behebung erfolgen separat
- Nicht bestandene Tests werden nach der Fehlerbehebung wiederholt

Glenford J. Myers [6] hat 10 Testprinzipien formuliert, hier die wichtigsten

- Ein wichtiger Bestandteil eines Testfalls ist die Definition des erwarteten Ergebnisses.
- Ein Programmierer sollte nicht sein eigenes Programm testen.
- Jeder Testprozess sollte eine gründliche Überprüfung der Testergebnisse enthalten.
- Testfälle müssen sowohl für gültige und erwartete Ausgangsbedingungen (**Positives Testen**) als auch für ungültige und unerwartete (**Negatives Testen)** definiert werden.
- Ein Programm darauf hin zu untersuchen, ob es nicht tut, was es tun sollte, ist nur die eine Hälfte der Schlacht. Die andere Hälfte besteht darin, zu untersuchen, ob das Programm etwas tut, was es nicht tun soll.
- Man sollte keinen Testaufwand betreiben unter der stillschweigenden Annahme, dass keine Fehler gefunden werden.
- Die Wahrscheinlichkeit, dass ein Programmabschnitt noch weitere Fehler enthält ist proportional zur Anzahl der schon gefundenen Fehler.
- Testen ist eine sehr kreative und intellektuell herausfordernde Aufgabe

Je mehr Testfälle wir untersuchen, umso mehr Fehlermöglichkeiten können wir ausschließen. Jedoch wird dadurch das Testen auch sehr zeitaufwändig.Wir sollten daher versuchen, die Testfälle möglichst sinnvoll auszuwählen. Im folgenden lernen wir einige Testmethoden und Kriterien kennen, die uns die Auswahl der Testfälle vereinfacht.

## 8.2 Testmethoden

**Modultest:** Unter einem Modultest versteht man das Testen einzelner Einheiten einer Anwendung. Dies können Gruppen von Klassen, einzelne Klassen oder auch einzelne Methoden sein. Ein Modultest kann lange vor der Fertigstellung einer Anwendung durchgeführt werden. BlueJ bietet gute Möglichkeiten zum Testen einzelner Klassen und Methoden mit Hilfe der Objektleiste und des Debuggers.

**Akzeptanztest**: Bei einem Akzeptanztest wird die gesamte Anwendung getestet, d.h. das Zusammenspiel mehrerer Klassen, gegebenenfalls auch mit der Oberfläche. Beim Akzeptanztest wird auch überprüft, ob das Programm die Anforderungen aus dem Pflichtenheft erfüllt.

**Black-Box-Test**: Black-Box-Tests werden aus den Anforderungen heraus entwickelt, ohne Berücksichtigung des Quelltexts. Sie lassen sich daher auch auf alternative Implementierungen anwenden, wenn die gleichen Anforderungen vorliegen. Sie haben den Vorteil, dass sie schon vor der Implementierung der Klassen konstruiert werden können.

**White-Box-Test**: White-Box-Tests dagegen orientieren sich direkt am Quelltext. Die Testfälle werden so konstruiert, dass jede Quelltextzeile von mindestens einem Testfall durchlaufen wird. Durch Änderungen am Quelltext können hierbei einzelne Testfälle nutzlos werden.

**Walkthrough:** Das Durchdenken eines Prozesses wird beim Testen von Software durch manuelles Ausführen von Funktionen bewerkstelligt. Dazu druckt man sich ein Stück Quellcode aus und versucht den Ablauf von Hand mit verschiedenen Beispielen nachzuvollziehen.

**Regressionstests**: Nach dem Entdecken von Fehlern durch die Testgruppe und dem Herausfinden und Beheben der Fehler durch die Programmierer müssen sämtliche Tests (auch solche, die beim ersten Testdurchlauf erfolgreich verlaufen waren) erneut durchgeführt werden, denn nicht selten entstehen beim Beheben von Fehlern an anderer Stelle neue Fehler. Daher ist eine gute **Testdokumentation** unerlässlich.

## 8.3 Auswahl geeigneter Testfälle

Einige wichtige Kriterien für die Auswahl geeigneter und ausreichender Testfälle seien im folgenden genannt.

**Bereichsgrenzen**: Beim Testen ist es wichtig nicht nur Werte im Innern eines zulässigen

Wertebereichs zu wählen, sondern besonders auch an den Grenzen.

**Testen von Schleifen**: Testfälle für Schleifen sollten so konstruiert werden, dass Schleifen gar nicht, einmal, mehrere Male oder sehr oft durchlaufen werden. Dies gilt nur für Schleifen, bei denen die Anzahl der Durchläufe nicht fest ist.

**Code-Abdeckung**: Testfälle sollen so konstruiert werden, dass möglichst jeder Quelltextabschnitt durchlaufen wird, d.h. es müssen alle Alternativ-Zweige (if) berücksichtigt werden.

**Pfad-Abdeckung**: Hierbei sollen die Testfälle so konstruiert werden, dass möglichst alle Pfade abgedeckt werden

Nachfolgend noch ein Beispiel, das den Unterschied zwischen Code- und Pfadabdeckung verdeutlicht. Die angegebene Methode soll die größte von 3 Zahlen bestimmen:

```
1    public int max(int a, int b, int c)
2    {
3         int max = a;
4         if (b > a)
5              max = b;
6         if (c > max)
7              max = c;
8         return max;
9    }
```

Für eine Code-Abdeckung reicht es aus ein Beispiel zu wählen, bei dem c die größte Zahl ist, denn dann wird jede Quelltextzeile abgedeckt. Für eine Pfadabdeckung brauchen wir vier Testfälle, da es 4 Pfade gibt: nur Zeile 3 wird ausgeführt, Zeile 3 und 5, Zeile 3 und 7 oder Zeile 3,5 und 7 werden ausgeführt. Für einen Black-Boxtest würde man wieder anders vorgehen, man würde die Testfälle so wählen, dass nach Größe sortiert alle Zahlenkombinationen vertreten sind (also a,b,c; a,c,b; b,a,c; b,c,a; c,a,b; c,b,a). Darüber hinaus würde man Testfälle betrachten, bei denen zwei oder alle drei Zahlen gleich groß sind.

## 8.4 Automatisieren von Tests

In BlueJ können wir Objekte auf der Objektleiste erzeugen und so direkt einzelne Methoden testen. Der Objektinspektor hilft uns, die Ergebnisse zu überprüfen. Wie in Kapitel 8.2 schon erwähnt, sollten Tests nach dem Beheben von Fehlern wiederholt werden. Bei manueller Ausführung von Tests ist dies sehr aufwändig. Das Schreiben von Programmen als Testgerüste und das Automatisieren von Tests bietet hierbei eine Hilfe. Dazu verwenden wir das Test-Rahmenwerk **JUnit**, das von Kent Beck und Erich Gamma unabhängig von speziellen Entwicklungsumgebungen entwickelt wurde (siehe auch www.junit.org) und in BlueJ bereits integriert ist. Aktivieren können Sie das Testwerkzeug unter dem Menü *Tools/Preferences* auf der Registerseite *Interface*, indem Sie den Punkt „*Show unit testing tools*“ aktivieren.

Um nun Tests wiederholen zu können müssen die Tests zuvor bei der Ausführung aufgezeichnet werden. Dazu sind die folgenden Schritte notwendig:

1. Erzeugen einer Testklasse für die zu testende Klasse: Wählen Sie im Kontextmenü der entsprechenden Klasse den Eintrag „*Create Test Class*“.
2. Erzeugen einer Testmethode: Wählen Sie im Kontextmenü der Testklasse den Eintrag „*Create Test Method*“ und geben Sie einen Namen für die Methode an.
3. Test aufzeichnen: Führen Sie, wie beim manuellen Testen alle Aktionen aus, die für den Test notwendig sind und betätigen Sie die Schaltfläche *End*, wenn Sie den Test beenden wollen. Rufen Sie dabei eine Methode mit Rückgabewert auf, so können Sie in einem Dialogfenster das zu erwartende Ergebnis angeben. Man spricht hierbei von einer **Zusicherung** (engl. assertion).

Eine Zusicherung überprüft ob ein vorgegebener Wert gleich dem Ergebnis des Methodenaufrufs ist. Sind sie ungleich, so deutet dies auf einen Fehler im Programm hin. Alle aufgezeichneten Tests können über die Schaltfläche „*Run Tests*“ gestartet werden.

***Fragen:***

- ✗ Was versteht man unter Test-Verifikation?
- ✗ Was ist der Unterschied zwischen einem Black-Box- und einem White-Box-Test?
- ✗ Was versteht man unter negativem Testen?

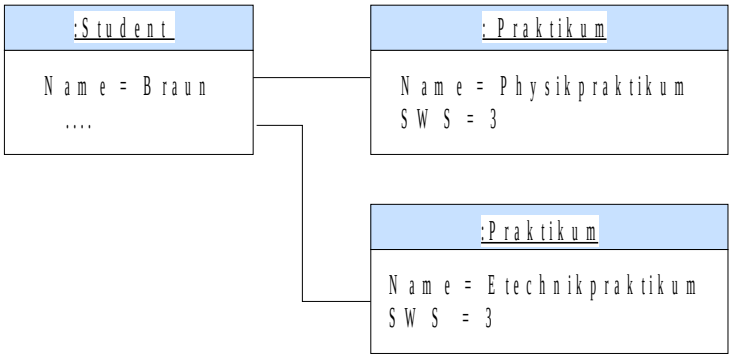
# 9 Objektorientierte Analyse – statische Konzepte

***Lernziele:***

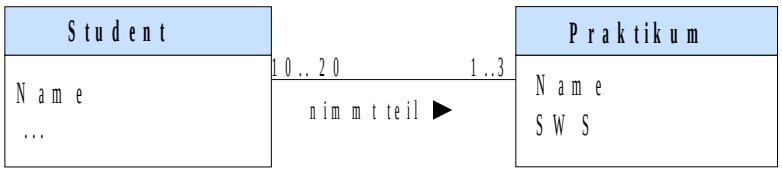
- **Verstehen**
  - Erklären können, was eine Assoziation ist
  - Erklären können, was eine assoziative Klasse ist
  - Erklären können, was Aggregation und Komposition bedeuten
  - Erklären können, was Vererbung ist
- **Anwenden**
  - Die UML-Notation für Assoziation und Vererbung anwenden können
  - Assoziationen in einem Text erkennen und darstellen können
  - Vererbungsstrukturen in einem Text identifizieren und darstellen können

## 9.1 Assoziation

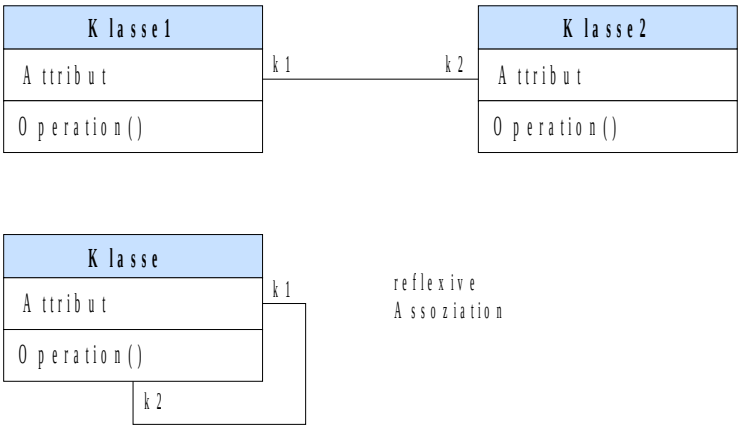
Eine **Assoziation** modelliert Verbindungen zwischen Objekten. Handelt es sich dabei um Objekte derselben Klasse, so spricht man von einer **reflexiven Assoziation**. Obwohl eine Assoziation eine Beziehungen zwischen Objekten beschreibt, ist es üblich von einer Assoziation zwischen Klassen zu sprechen.
Beispiel 9.1: Betrachten wir wieder die Klasse *Student*. Jede/r Studierende kann an einem oder mehreren Praktika teilnehmen. Nimmt der Student Braun am Physikpraktikum und am Elektrotechnikpraktikum teil, so besteht eine Verbindung zwischen Braun und dem Physikpraktikum, sowie zwischen Braun und dem Elektrotechnikpraktikum, was grafisch im Objektdiagramm folgendermaßen dargestellt wird:



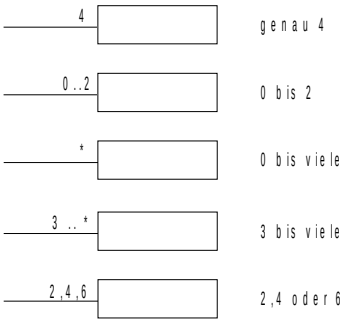

Wenn jede/r Studierende an mindestens einem und höchstens 3 Praktika teilnehmen muss und an jedem Praktikum zwischen 10 und 20 Studierende teilnehmen können, so wird dies im Klassendiagramm folgendermaßen dargestellt:




Die Verbindungen werden als Assoziationen (Beziehungen) zwischen den Objekten der Klassen *Student* und *Praktikum* bezeichnet. Man spricht dabei auch von einer „verwendet-ein“-Beziehung. Die allgemeine Notation im Klassendiagramm der UML sieht folgendermaßen aus:

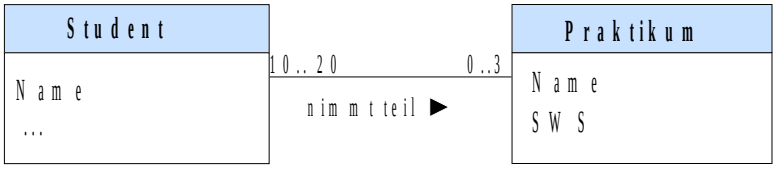



An jedem Ende der Linie zwischen einer oder zwei Klassen muss die **Multiplizität** bzw. **Kardinalität** angegeben werden.
Sie beschreibt wie viele Objekte einer Klasse an der Beziehung teilnehmen können. Beispiele für Kardinalitäten der UML sind:




Man unterscheidet Kann- und Muss-Assoziationen. Eine **Kann-Assoziation** hat als Untergrenze die Kardinalität 0, eine **Muss-Assoziation** eine Kardinalität größer oder gleich 1.

Beispiel 9.2: Eine Kann-Assoziation von *Student* zu *Praktikum* bedeutet, dass es Studierende geben kann, die an keinem Praktikum teilnehmen. Die Muss-Assoziation von *Student* zu *Praktikum* aus Beispiel 9.1 besagt, dass es keinen Studierenden geben darf, der kein Praktikum besucht.

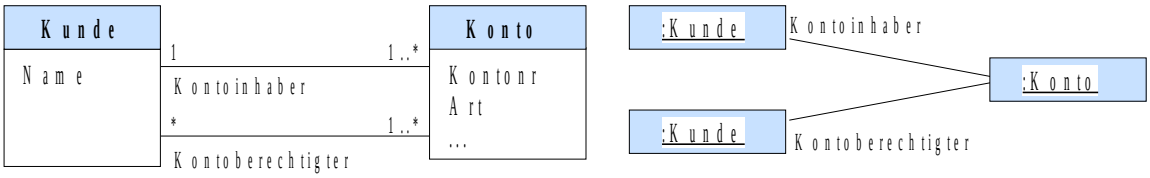


Der **Assoziationsname**, der optional angegeben werden kann, beschreibt im allgemeinen nur eine Richtung der Assoziation, die durch ein schwarzes Dreieck angegeben wird.

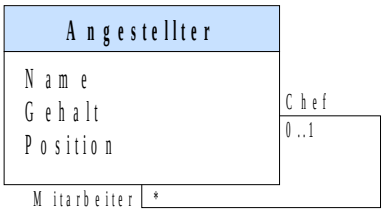
Beispiel 9.3: In Beispiel 9.2 ist „nimmt teil“ der Name der Assoziation von *Student* zu *Praktikum*. Überlegen Sie sich einen Namen für die Assoziation von *Praktikum* zu *Student*.

Häufiger noch als der Assoziationsname wird der Rollenname oder kurz die Rolle angegeben. Die **Rolle** enthält Informationen über die Bedeutung eines Objekts in der Assoziation. Der Rollenname wird an das jeweilige Ende der Assoziation geschrieben.

Besteht zwischen zwei Klassen mehr als eine Verbindung, so müssen Rollennamen oder Assoziationsnamen angegeben werden.
Beispiel 9.4: Betrachten wir ein Bankkonto. Ein Kunde der Bank, kann bezogen auf ein Konto entweder Kontoinhaber oder Kontoberechtigter sein. Im Klassen- bzw. Objektdiagramm wird dies folgendermaßen dargestellt:




Beispiel 9.5:
Bei der reflexiven Assoziation, die nachfolgend dargestellt ist, kann ein Angestellter Chef eines anderen Angestellten sein. Umgekehrt ist ein Angestellter Mitarbeiter eines anderen Angestellten.

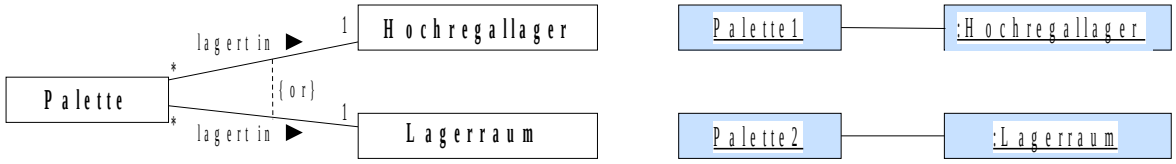



Assoziationen können durch **Restriktionen** (Einschränkungen) erweitert werden. Dabei können die Restriktionen entweder frei formuliert werden oder es können Standardrestriktionen geschaffen werden.
Beispiel 9.6: In Beispiel 9.4 mit Kunde und Konto kann ein Kontoinhaber nicht gleichzeitig Kontoberechtigter sein. Dies wird durch die folgende Restriktion ausgedrückt:

{ Kunde.Kontoinhaber ≠ Kunde.Kontoberechtigter }

Zwei häufig verwendete Standardrestriktionen sind die **or-Restriktion** und die **subset-Restriktion**. Bei der **or-Restriktion** kann zu jedem beliebigen Zeitpunkt nur eine Assoziation gelten.

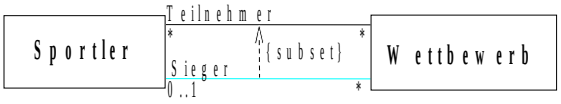
Beispiel 9.7:



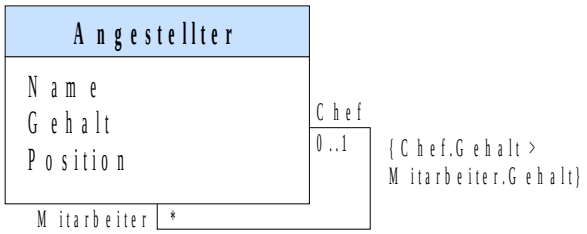

In diesem Beispiel muss für jede Palette eine Verbindung zu einem Lager aufgebaut werden. Ist dies nicht notwendig muss die Kardinalität 0..1 gewählt werden.

Die **subset-Restriktion** bedeutet, dass eine Assoziation eine Teilmenge einer anderen Assoziation ist. Nur wenn eine „Haupt-“Verbindung zwischen zwei Objekten existiert, kann auch eine subset-Verbindung aufgebaut werden.

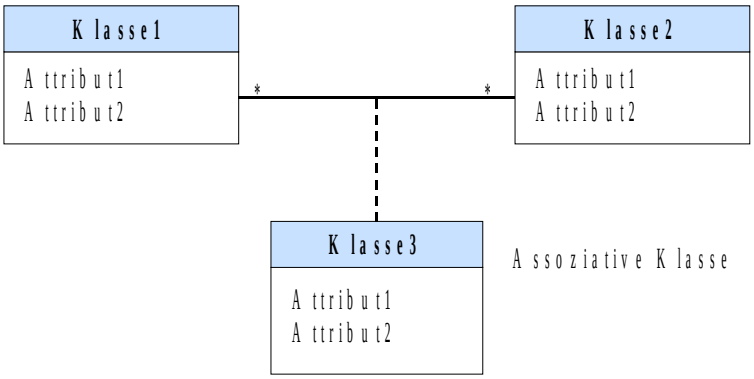
Beispiel 9.8:



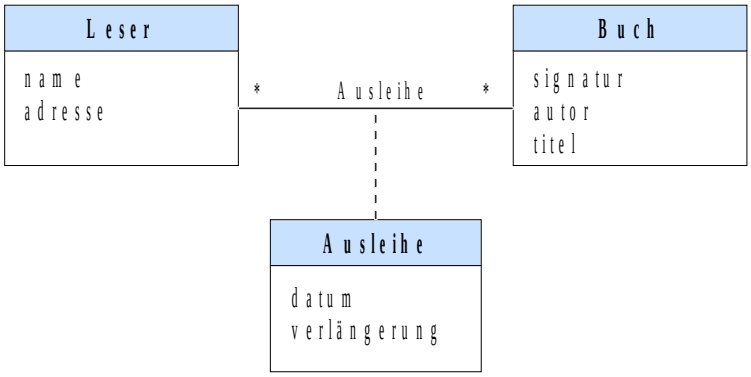

Die Sportler s1 und s2 haben beide am Wettbewerb w1 teilgenommen, die Sportler s2 und s3 sind Teilnehmer des Wettbewerbs w2. s1 ist Sieger von w1, s3 Sieger von w2. Versuchen Sie das dazugehörige Objektdiagramm zu zeichnen.
Nur wenn eine „Haupt-“Verbindung zwischen zwei Objekten existiert, kann auch eine subset-Verbindung aufgebaut werden.
Eine Restriktion kann sich auch nur auf eine einzige Assoziation beziehen. In Beispiel 4.5 wird die Einschränkung, dass das Gehalt des Chefs größer sein soll, als die Gehälter seiner Mitarbeiter folgendermaßen dargestellt:




Eine Assoziation kann zusätzlich die Eigenschaften einer Klasse (Attribute, Operationen, weitere Assoziationen) besitzen. Man spricht dann von einer **assoziativen Klasse.** Die assoziative Klasse oder **Assoziationsklasse** wird durch ein Klassensymbol dargestellt, das über eine gestrichelte Linie mit der Assoziation verbunden wird.




Die Assoziationsklasse besitzt die Eigenschaften einer Klasse und einer Assoziation. Beim Aufbau der Objektbeziehung zwischen zwei Objekten wird ein Objekt der Assoziationsklasse erzeugt und mit den entsprechenden Attributwerten gefüllt.
Beispiel 9.9: Wir betrachten einen Leser, der ein Buch ausleiht. Der Ausleihvorgang führt dazu, dass ein Objekt (der Klasse Ausleihe) erzeugt wird, in dem gespeichert wird, wann das Buch zurückgegeben werden muss, und ob es schon verlängert wurde.



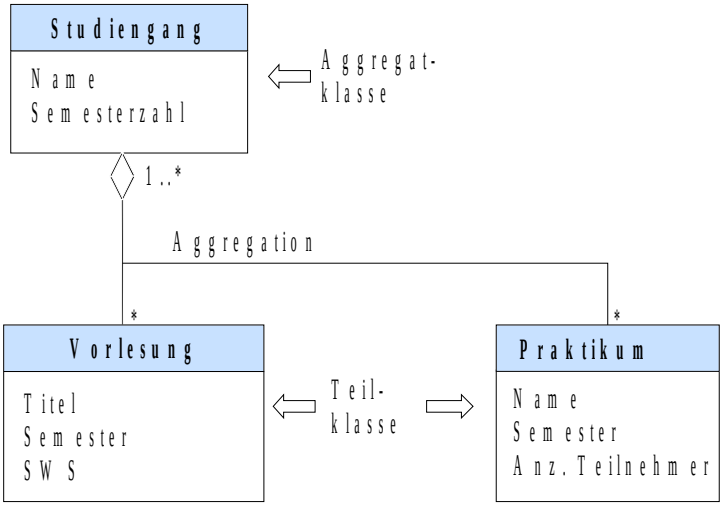

## 9.2 Aggregation und Komposition

Wenn in der Analyse schon genauere Informationen über eine Beziehung bekannt sind, kann die Assoziation eindeutig spezifiziert werden. Es gibt zwei unterschiedliche Assoziationen.
Eine **Aggregation** zwischen Objekten besteht, wenn sich die Verbindung als „ist Teil von“ bzw. „besteht aus“ beschreiben lässt. Eine Aggregation ist gerichtet, d.h. wenn B Teil von A ist, dann darf A nicht Teil von B sein. B wird als **Teilklasse**, A als **Aggregatklasse** bezeichnet.
Eine **Komposition** ist eine starke Form der Aggregation. Auch hier besteht eine gerichtete „Ganzes-Teile-Beziehung“. Zusätzlich gilt für eine Komposition:

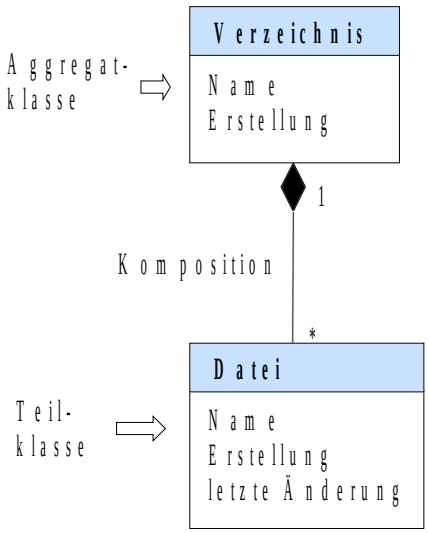
- Jedes Objekt der Teilklasse kann zu einem Zeitpunkt nur Komponente eines einzigen Objekts der Aggregatklasse sein (d.h. die Kardinalität bei der Aggragatklasse darf nicht größer als 1 sein).
- Die Dynamik des Ganzen gilt auch für seine Teile. Wird z.B. das Ganze kopiert, so werden auch seine Teile kopiert.
- Wird das Ganze gelöscht, so werden auch seine Teile gelöscht. Ein Teil darf jedoch auch zuvor explizit gelöscht werden.

In der UML werden Aggregation und Komposition durch eine Raute an der Aggregatklasse gekennzeichnet, bei der Aggregation ist diese transparent, bei der Komposition gefüllt.

Beispiel 9.10: Ein Studiengang setzt sich aus mehreren Vorlesungen und Praktika zusammen. Die Vorlesungen existieren unabhängig von dem Studiengang, sie können auch gleichzeitig zu einem anderen Studiengang gehören. Daher liegt eine Aggregation vor.




Beispiel 9.11: Ein Verzeichnis enthält mehrere Dateien, wobei jede Datei nur in einem Verzeichnis enthalten sein kann. Wird zum Beispiel das Verzeichnis kopiert, so werden auch alle darin enthaltenen Dateien kopiert. Es handelt sich hier also um eine Komposition.




Viele Notationselemente der Assoziation können auch bei Objektdiagrammen verwendet werden, z.B. Rollennamen oder Symbole für die Aggregation bzw. Komposition.

Beispiel 9.12:
Es soll ein Kreis mit dem Mittelpunkt (5,20) und dem Radius 50 erzeugt werden. Des weiteren sollen 100 gleichmäßig auf dem Kreisumfang verteilte Punkte erzeugt und in einem Array abgespeichert werden. Die Punkte sollen dann auf dem Bildschirm ausgegeben werden.

Modellbildung:
Aus der Beschreibung ergeben sich zunächst die Objekte:

| Kreis1 |
| --- |
| Mittelpunkt = (5,20)<br>radius = 50 |

| Mittelpunkt |
| --- |
| x = 5<br>y = 20 |

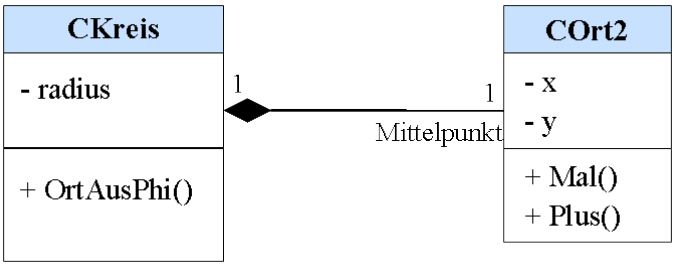
Um diese beiden Objekte umzusetzen, können wir die Klassen *COrt2* und *CKreis* verwenden, die wir in den Kapiteln 5.2 und 5.3 programmiert haben.
Es soll nun ein Punkt auf dem Kreis berechnet werden, dies erfolgt nach der Formel:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} M_x \\ M_y \end{pmatrix} + r \cdot \begin{pmatrix} \cos(\varphi) \\ \sin(\varphi) \end{pmatrix}$$

Aus dieser Formel ergeben sich noch zwei Verhaltensweisen, die für einen Ortsvektor implementiert müssen:

- das Multiplizieren eines Ortsvektors mit einer reellen Zahl
- das Addieren von zwei Ortsvektoren

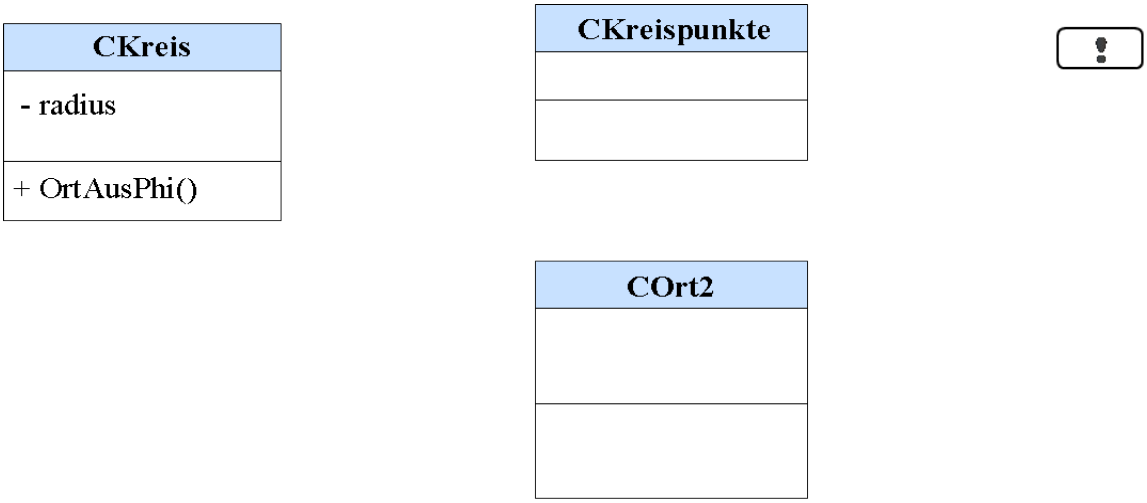
Des weiteren benötigt die Klasse *CKreis* eine Methode, mit der ein Punkt auf dem Kreis berechnet werden kann. Innerhalb dieser Methode muss dann auf die beiden anderen Methoden der Klasse *COrt2* zugegriffen werden und die Umsetzung obiger Formel programmiert werden.
Da jeder Kreis einen eigenen Mittelpunkt **hat**, besteht eine Komposition zwischen der Klasse *CKreis* und der Klasse *COrt2*. Daraus ergibt sich dann das folgende Klassendiagramm




Die Operationsbeschreibungen der drei Methoden lauten:

1. Mal(): Eingabe: Faktor vom Typ double
   Funktionsweise: Berechnen des Produktes aus Faktor und Ortsvektor
   Rückgabe: Berechnetes Produkt vom Typ *COrt2*
2. Plus(): Eingabe: Ortsvektor vom Typ *COrt2*
   Funktionsweise: Berechnen der Summe des eingegebenen Ortsvektors und des aufrufenden
   Rückgabe: Berechnete Summe vom Typ *COrt2*
3. OrtAusPhi():
   Eingabe: Winkel vom Typ double
   Funktionsweise: Berechnet Punkt auf dem Kreis, der dem Winkel entspricht
   Rückgabe: Berechneter Punkt vom Typ *COrt2*

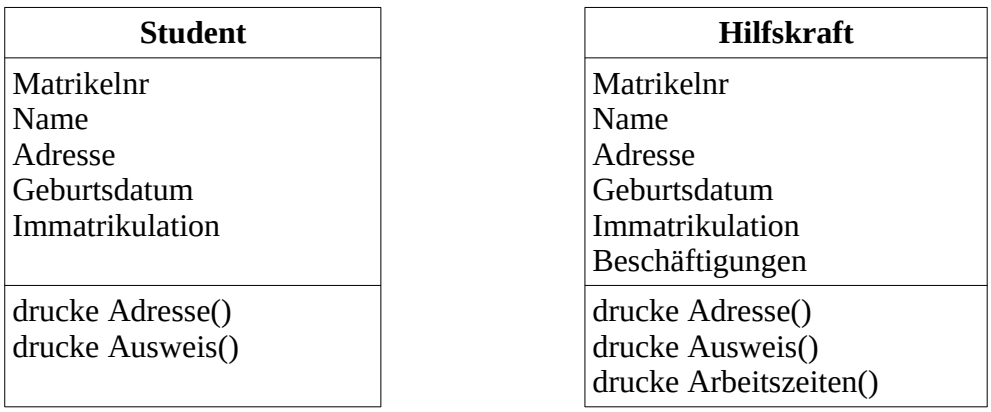
Um die 100 Punkte zu verwalten,wurde eine Sammlungs-Klasse verwendet, die Klasse *CKreispunkte*. Die Klasse *CKreispunkte* enthält einen Konstruktor, in dem die Punkte erzeugt werden und eine Methode zur Ausgabe der Punkte. Damit ergibt sich das folgende Klassendiagramm:




Betrachten Sie nochmal die Klasse *CKreis* aus Kapitel 5.3. Wie wird die Komposition in der Programmierung der Klasse *CKreis* umgesetzt?
**Dies bedeutet, eine Komposition oder Aggregation im Klassendiagramm führt immer zu einem Attribut vom Datentyp der Teilklasse in der Aggregatklasse!**

## 9.3 Vererbung

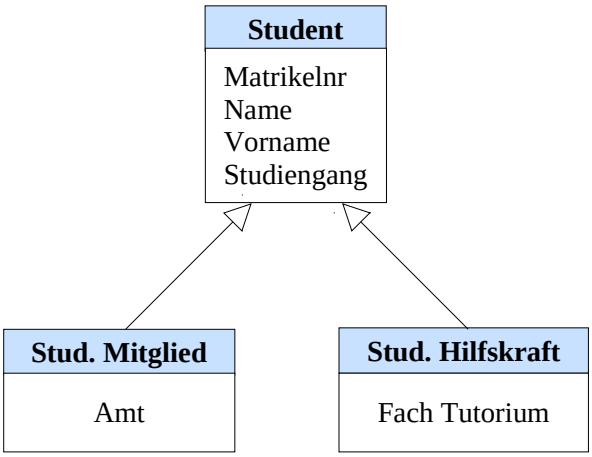
**Vererbung** erlaubt uns eine Klasse als Erweiterung einer anderen Klasse zu definieren oder Gemeinsamkeiten mehrerer Klassen in einer sogenannten **Basisklasse** zusammenzufassen. Die **Vererbung** beschreibt sozusagen die Beziehung zwischen einer allgemeinen Klasse (Basisklasse) und einer spezialisierten (abgeleiteten) Klasse. Die **spezialisierte Klasse** kann alle Informationen (Attribute, Operationen, Assoziationen) der **Basisklasse** verwenden und enthält zusätzliche Informationen. Man sagt, sie **erbt** die Attribute und Operationen der Basisklasse. Ein Objekt der spezialisierten Klasse kann überall verwendet werden, wo ein Objekt der Basisklasse erlaubt ist. Es wird auch von einer **»ist ein«-Beziehung** gesprochen, denn ein Objekt der spezialisierten Klasse ist ein Objekt der Basisklasse.
Klassen, die über Vererbungsbeziehungen miteinander verknüpft sind, bilden eine **Vererbungshierarchie.** Man spricht auch von einer **Klassenhierarchie** oder einer **Vererbungsstruktur**. Die allgemeine Klasse wird auch als **Superklasse** oder **Oberklasse**, die abgeleitete Klasse als **Subklasse** oder **Unterklasse** bezeichnet.
Beispiel 9.13:




Betrachtet man die beiden Klassen, so fällt auf, dass sie sehr viele gemeinsame Attribute und Operationen besitzen.
Die Klasse Hilfskraft besitzt alle Attribute und Operationen der Klasse Student und zusätzlich noch das Attribut *Beschäftigungen* und die Operation *drucke Arbeitszeiten.* Das heißt, eine Hilfskraft ist

ein Student mit zusätzlichen Eigenschaften. Die Klasse Hilfskraft ist somit eine Erweiterung der Klasse Student.

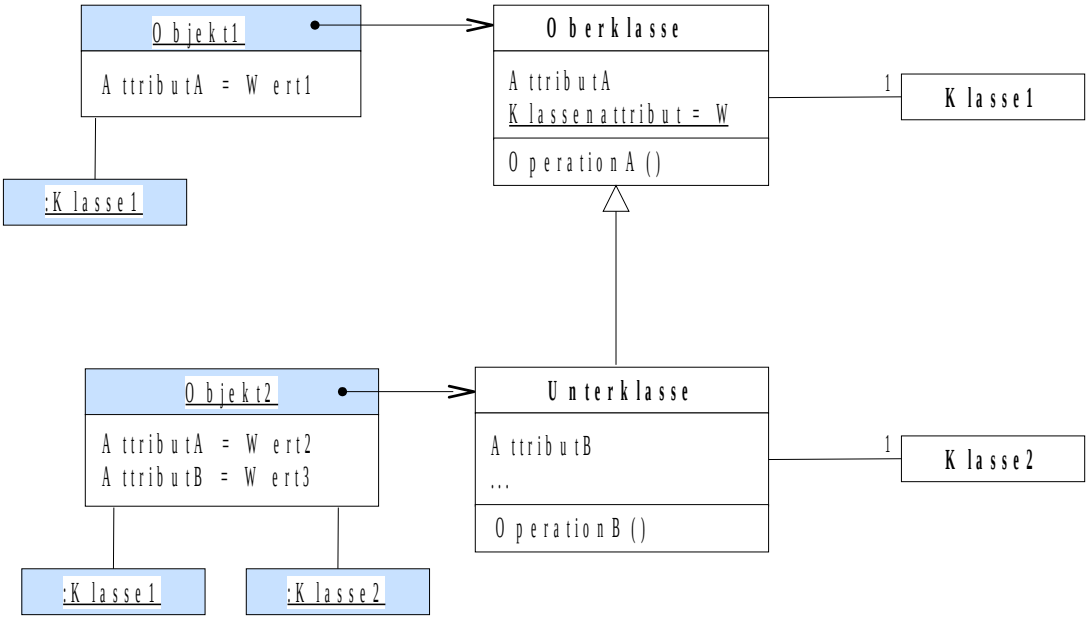
In der UML wird diese Vererbung folgendermaßen dargestellt:




Die folgenden Informationen einer Basisklasse werden vererbt:

- Attribute, aber nicht die Attributwerte
- Operationen, auch Klassenoperationen
- Klassenattribute mit Attributwert
- Assoziationen

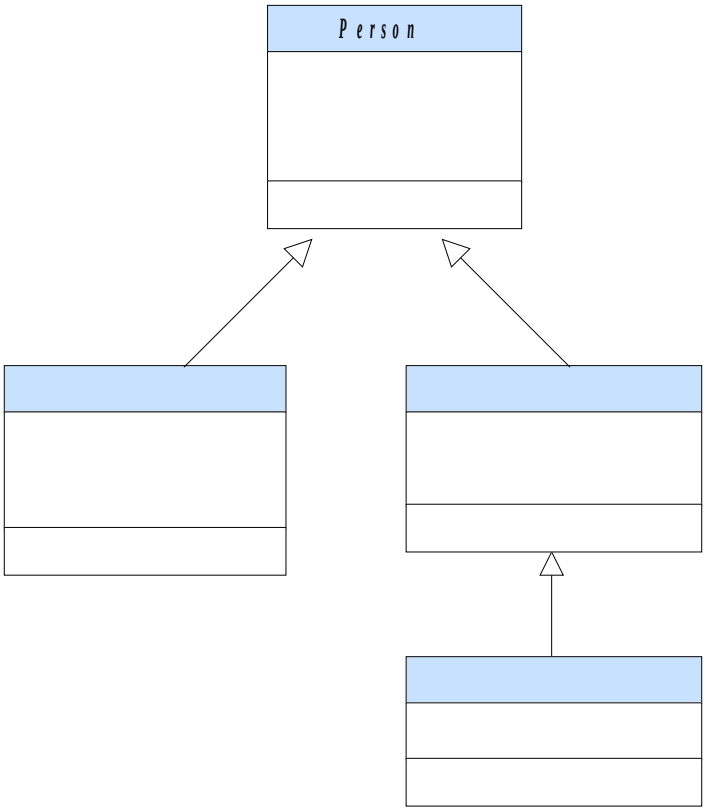
Ein Objekt der abgeleiteten Klasse besitzt also die Attribute der Basisklasse und zusätzliche Attribute der abgeleiteten Klasse. Auf das Objekt können Operationen der Oberklasse und zusätzlich Operationen der Unterklasse angewendet werden. Der Mechanismus ist in folgendem Diagramm dargestellt.



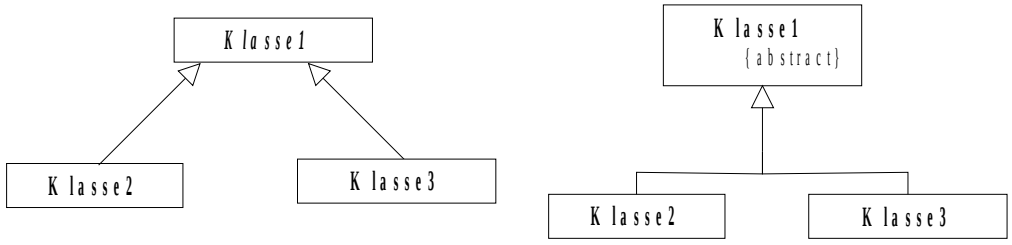

Unterklassen können das Verhalten ihrer Oberklassen verfeinern oder neu definieren.
Nun sollen zusätzlich noch Angestellte der Hochschule verwaltet werden.
Beispiel 9.14:

| Personalnr<br>Name<br>Adresse<br>Geburtsdatum<br>Gehalt<br>Bank | Matrikelnr<br>Name<br>Adresse<br>Geburtsdatum<br>Immatrikulation | Matrikelnr<br>Name<br>Adresse<br>Geburtsdatum<br>Immatrikulation<br>Beschäftigungen |
|---|---|---|
| drucke Adresse()<br>überweise Gehalt() | drucke Adresse()<br>drucke Ausweis() | drucke Adresse()<br>drucke Ausweis()<br>drucke Arbeitszeiten() |

Wir finden wieder gemeinsame Attribute und Operationen, aber es existiert keine Klasse, die alle Gemeinsamkeiten enthält. Um trotzdem die gemeinsamen Attribute und Operationen zusammenfassen zu können, erzeugen wir eine zusätzliche Klasse „Person“ und können die Klassen in einer Klassenhierarchie darstellen. Sie enthält die gleichen Informationen, gleiche Attribute und Operationen kommen jetzt jedoch nur einmal vor.
Die Klasse Person ist als sogenannte **abstrakte Klasse** modelliert. Von abstrakten Klassen dürfen keine Objekte gebildet werden, wir haben sie ja auch nachträglich hinzugefügt. Abstrakte Klassen werden nur modelliert, um ihre Informationen an spezialisierte Klassen zu vererben.




In der UML kann eine Klassenhierarchie mit abstrakter Basisklasse wie in den beiden folgenden Darstellungen modelliert werden:




Bei der Analyse wird nur die sogenannte **Einfachvererbung** verwendet, bei der jede Klasse höchstens eine direkte Oberklasse besitzt. Die Mehrfachvererbung, bei der eine Klasse mehrere direkte Oberklassen besitzen kann, wird auch von Java nicht unterstützt.
Statt von Vererbung wird auch von **Generalisierung** gesprochen.

## 9.4 Paket

Ein **Paket** faßt Modellelemente (z.B. Klassen) zusammen. Das Paket wird benötigt um Elemente des Modells sinnvoll zu gruppieren. Ein Paket kann selber Pakete enthalten. Das gesamte Softwaresystem ist ein Paket, das andere Pakete enthält. Auf die Notation gehen wir hier nicht ein.

***Fragen:***

- ✗ Was modelliert eine Assoziation?
- ✗ Welche speziellen Assoziationen gibt es?
- ✗ Was beschreibt die Kardinalität?
- ✗ Was versteht man unter Vererbung?
- ✗ Wozu dient die abstrakte Klasse in einer Vererbungsstruktur?

***Beispielaufgabe:***
1) Ziel: Klassendiagramm mit Assoziationen und Vererbung erstellen können.
Identifizieren Sie anhand der folgenden Beschreibung Klassen, Attribute, Operationen, Assoziationen und Vererbungsstrukturen und zeichnen Sie sie in ein Klassendiagramm ein. Prüfen Sie, welche Art der Assoziation vorliegt.

> Wir betrachten eine Bank und ihre Kunden. Eine Person wird Kunde, wenn sie ein Konto eröffnet. Ein Kunde kann beliebig viele weitere Konten eröffnen. Für jeden neuen Kunden werden dessen Name, Adresse und das Datum der ersten Kontoeröffnung erfaßt. Bei der Kontoeröffnung muss der Kunde gleich eine erste Einzahlung vornehmen. Wir unterscheiden Girokonten und Sparkonten. Girokonten dürfen bis zu einem bestimmten Betrag überzogen werden. Für jedes Konto wird ein individueller Habenzins, für Girokonten auch ein individueller Sollzins festgelegt; außerdem besitzt jedes Konto eine eindeutige Kontonummer. Für jedes Sparkonto wird die Art des Sparens – z.B. Festgeld – gespeichert. Ein Kunde kann Beträge einzahlen und abheben. Desweiteren werden Zinsen gutgeschrieben und bei Girokonten Überziehungszinsen abgebucht. Um die Zinsen zu berechnen, muss für jede Kontobewegung das Datum und der Betrag notiert werden. Die Gutschrift/Abbuchung der Zinsen erfolgt bei den Sparkonten jährlich und bei den Girokonten quartalsweise. Ein Kunde kann jedes seiner Konten wieder auflösen. Bei der Auflösung des letzten Kontos hört er auf, Kunde zu sein.

Lösung:

# 10 Aggregation und Komposition in der Programmierung

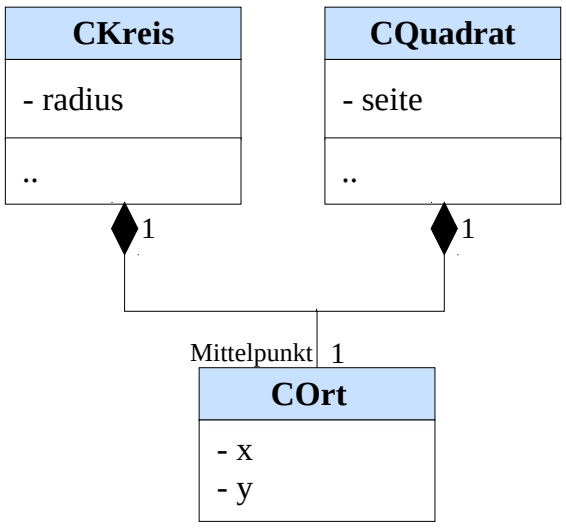
***Lernziele:***

- **Wissen**
  -
- **Verstehen**
  - Verstehen, was der Unterschied in der Implementierung einer Aggregation und einer Komposition ist
  -
- **Anwenden**
  - Eigene Programme mit Aggregation und Komposition schreiben können

Alle Assoziationen im Klassendiagramm beschreiben Beziehungen zwischen Objekten. Dies bedeutet: bei der Umsetzung einer Assoziation im Quellcode werden mindestens 2 Objekte erzeugt. Im folgenden wird auf die Umsetzung von Aggregation und Komposition in der Programmierung mit Java näher eingegangen.

## 10.1 Unterschied zwischen Aggregation und Komposition in der Programmierung

Die »hat ein«-Beziehung bedeutet, dass Objekte der Klasse A Objekte der Klasse B enthalten. Dies ist ein spezieller Fall der Assoziation und sie ist gerichtet.
Wir betrachten erneut die Klasse *CKreis* aus Kapitel 5.3. Dort wurde der Mittelpunkt des Kreises durch ein Attribut vom Typ *COrt2* umgesetzt. Es besteht damit eine Komposition zwischen den Klassen *CKreis* und *COrt2*: Der Kreis **hat** einen Mittelpunkt. Analog sei ein Quadrat durch seine Seitenlänge und sein Zentrum definiert. Das entsprechende Klassendiagramm sieht dann folgendermaßen aus:

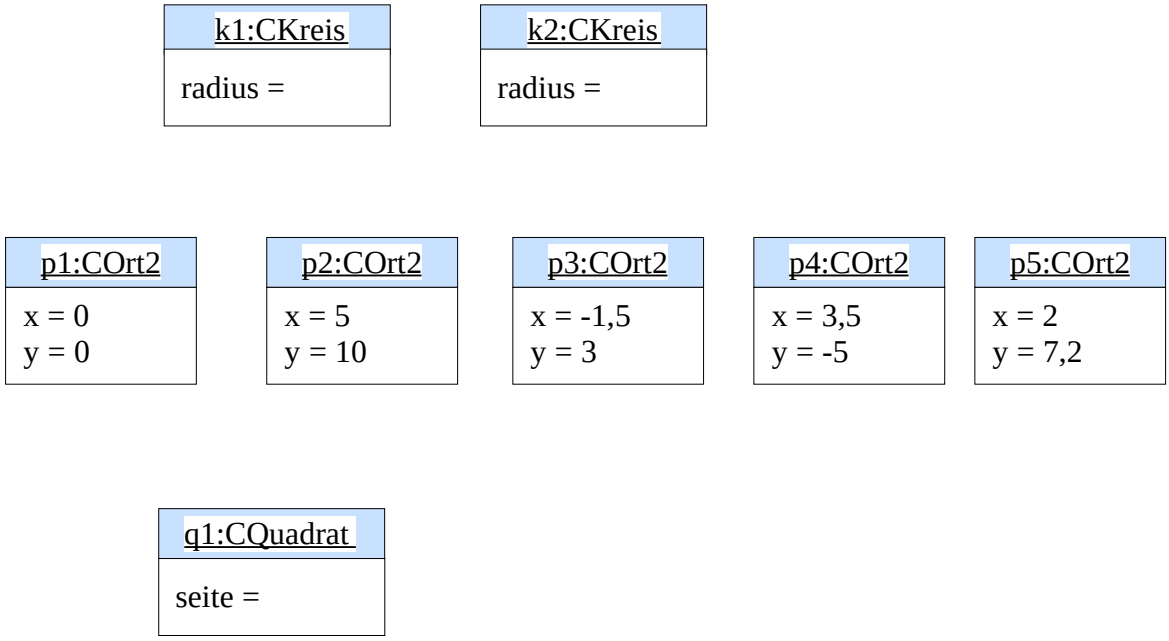



Zeichnen Sie das Objektdiagramm für einen Kreis mit dem Radius 5 und dem Mittelpunkt (0,10) daneben!
Der Java-Quellcode für die Komposition zwischen *CKreis* und *COrt2* sieht folgendermaßen aus

Beispiel 10.1: (Projekt *Kreis1)*

```
public class CKreis
{
    // Attribute
    private COrt2 mitte;
    private double radius;

    /**
    * Konstruktor mit Radius und Mittelpunkt
    */
    public CKreis(double mx, double my, double r)
    {  // Initialisierung
        mitte = new COrt2(mx,my);
        radius = r;
    }
```

Versuchen Sie die Klasse *CQuadrat* mit ihrem Konstruktor selber zu schreiben!
Nehmen Sie nun an, Sie haben 5 Punkte *p1*,*p2*,*p3*,*p4* und *p5* gegeben und die Aufgabe um die Punkte *p1*,*p3* und *p5* je einen Kreis mit dem Radius 3 zu zeichnen und um die Punkte *p1*,*p2* und *p5* je ein Quadrat mit der Seitenlänge 5. Ergänzen Sie das unten dargestellte Objektdiagramm durch die fehlenden Attributwerte, Objekte und Beziehungen. Worin unterscheidet sich das Diagramm, wenn Sie statt der Komposition eine Aggregation verwenden?



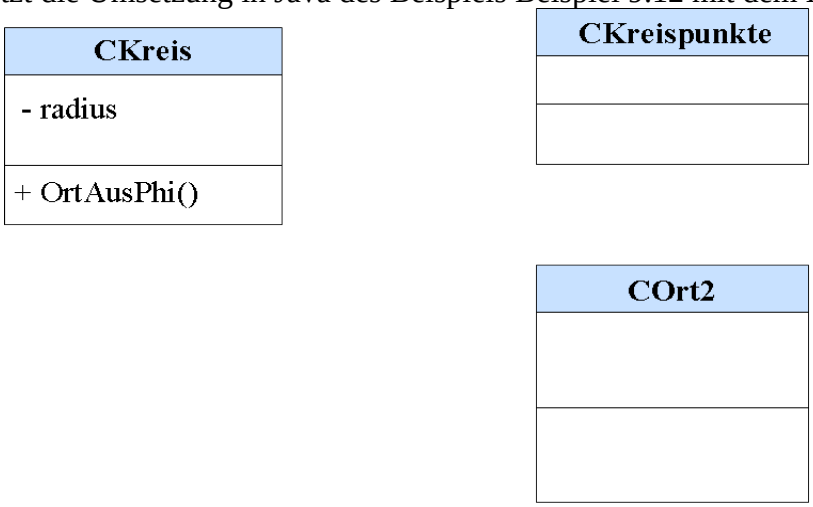

Damit die Punkte-Objekte unabhängig von den Kreis- und Quadrat-Objekten existieren (Aggregation), müssen die Konstruktoren abgeändert werden:

```
    public CKreis(COrt2 mitte, double r)
    {  // Initialisierung
        this.mitte = mitte;
        radius = r;
    }
```

Jetzt wird für den Mittelpunkt eines Kreises kein neuer Speicherplatz angelegt. Es wird lediglich das Attribut auf die Referenz des übergebenen Punktes gesetzt. Der Punkt, der Mittelpunkt des Kreises ist, existiert unabhängig von dem Kreis-Objekt. Es liegt also in diesem Fall keine Komposition vor, jedoch eine Aggregation.

## 10.2 Beispiel

Wir betrachten jetzt die Umsetzung in Java des Beispiels Beispiel 9.12 mit dem Klassendiagramm




und den Operationsbeschreibungen
*Mal()*: Eingabe: Faktor vom Typ double
Funktionsweise: Berechnen des Produktes aus Faktor und Ortsvektor
Rückgabe: Berechnetes Produkt vom Typ *COrt2*
*Plus():* Eingabe: Ortsvektor vom Typ *COrt2*
Funktionsweise: Berechnen der Summe des eingegebenen Ortsvektors und des aufrufenden
Rückgabe: Berechnete Summe vom Typ *COrt2*
*OrtAusPhi():*Eingabe: Winkel vom Typ double
Funktionsweise: Berechnet Punkt auf dem Kreis, der dem Winkel entspricht
Rückgabe: Berechneter Punkt vom Typ *COrt2*

Die Klasse *COrt2* ohne die Get()- und Set()-Methoden lautet dann:

```
public class COrt2
{ // Beginn Klasse Ortsvektor
     private double x; // x-Koordinate
     private double y; // y-Koordinate
     /**
     * Konstruktor
     * @param x,y     x- und y-Koordinate
     */
     public COrt2(double x, double y)
     {
          _________;
          _________;
     }

     /**
     * Kopierkonstruktor
     * @param ort  Ortsvektor
     */
     public COrt2(COrt2 ort)
     {
          x = ort.x;
          _________________;
     }

     /**
     * Addition zweier Vektoren
     * @param ort   Vektor, der addiert werden soll
     * @return      Summe der beiden Vektoren
     */
     public            Plus(           ort)
     {
                     erg;
          erg = new COrt2(x + ort.x,________);
          return erg;
     }
     /**
     * Multiplikation eines Vektors mit einer Zahl
     * @param d     Zahl, mit der der Vektor multipliziert
     * werden soll
     * @return      Produkt vom Typ COrt2
     */
     public            Mal(                    d)
     {
                     erg;
          erg = new COrt2(_________________);
          return erg;
     }
}
```

Wenn wir als Rückgabetyp eine Klasse haben (hier *COrt2*), müssen wir innerhalb der Methode eine lokale Variable vom gleichen Typ anlegen (hier *COrt2*), diesem Objekt Werte zuweisen und es mit *return* als Ergebnis der Methode zurückgeben. Versuchen Sie die Methodenaufrufe schrittweise mit Hilfe des Debuggers nachzuvollziehen.
Die Klasse *CKreis* aus Beispiel 5.2 müssen wir um die Operation *OrtAusPhi* ergänzen. Im folgenden ist der Quelltext der Klasse *CKreis* (ohne Verwaltungsoperationen) angegeben.
Klasse *CKreis*:

```
public class CKreis
{ // Beginn Klasse CKreis
     private COrt2 mitte;  // Mittelpunkt
     private double radius;     // Radius
     /**
     *  Konstruktor
     */
     public CKreis(double mx, double my, double r)
     {
          mitte = new COrt2(    ,    );
          radius =    ;
     }
     /**
     * Methode, um aus einem Winkel den zugehörigen Punkt
     * auf dem Kreis zu berechnen
     * @param phi   Winkel auf dem Kreis
     */
     public COrt2 OrtAusPhi(double phi)
     {
          COrt2 erg, h; // Hilfsvektoren
          // Richtungsvektor Winkel
          h = new COrt2(                              ,
                                );
          // Vektor vom Mittelpunkt zum gesuchten Punkt
          h = h.Mal(radius);
          erg =
                                 ; // gesuchter Ortsvektor
          return erg;
     }
} // Ende Klasse CKreis
```

Um die 100 Punkte (im Klassendiagramm durch die Kardinalität 100 dargestellt) zu verwalten können wir ein Objekt-Array verwenden. Dazu programmieren wir eine Klasse (*CKreispunkte*), die das Array als Attribut erhält. Die Klasse *CKreispunkte* benötigt nur einen Konstruktor, in dem die Punkte erzeugt werden und eine Methode zur Ausgabe der Punkte.
Im folgenden ist der Quelltext der Klasse *CKreispunkte* angegeben (siehe auch in Felix das Projekt *Kreispunkte).*

Klasse *CKreispunkte:*

```
/**
* Klasse zur Erzeugung und Speicherung der Punkte auf dem
* Kreis
*/
public class CKreispunkte
{ // Klasse Kreispunkte
     private                    Kreispkte; // Array für die
Punkte
     /**
     *  Konstruktor
     * @param k: Kreis, auf dem die Punkte erzeugt werden
     * sollen
     * @param anzahl    Anzahl der zu erzeugenden Punkte
     */
     public CKreispunkte(                    k, int anzahl)
     { // Parameter anzahl, damit auch andere Punktezahl als
       // 100 gewählt werden kann
          int i; // Schleifenvariable
          double dphi,phi;  // Rechenvariablen
          COrt2 ort;   // Hilfsvektor

          // Speicherplatz für Array anlegen
          Kreispkte = new                    [anzahl];
          // Erzeuge anzahl Kreispunkte
          dphi = 2.*Math.PI/                   ; //
Winkelschritt
          for (i=0;i<anzahl;i++)
          {
               phi = i*dphi; // aktueller Winkel
          // Berechnung des Punktes mit der Methode
          // der Klasse CKreis
               ort = k.OrtAusPhi(phi);
               // Speichern des Punktes im Array
               Kreispkte[i] = new COrt2(ort); // oder = ort
          }// Ende for-Schleife
     } // Ende Konstruktor

     /*
     *  Methode zur Ausgabe der Kreispunkte
     */
     public void Ausgabe()
     { // Beginn Methode Ausgabe
          int i = 1;
          for (                   p:Kreispkte)
          {
               System.out.print( i + ".Kreispunkt: ");
               System.out.println(p.X() + ", " +p.Y());
               i++;
     } // Ende Methode Ausgabe
} // Ende Klasse CKreispunkte
```

***Fragen:***

- ✗ Woran erkennt man Aggregation und Komposition im Quelltext?
- ✗ Woran kann man beide unterscheiden?

# 11 Objektsammlungen

***Lernziele:***

- **Wissen**
  - Wissen, was eine Objektsammlung ist
  - Wissen, was ein Sammlungsobjekt ist
  - Wissen, was unter einer generischen Klasse zu verstehen ist
- **Verstehen**
  - Erklären können, was der Parameter in der Klasse ArrayList bedeutet
  - Verstehen, wann bei einer Objektsammlung der Iterator eingesetzt wird
  - Die Dokumentation generischer Klassen lesen können
- **Anwenden**
  - Arrays von Objekten in der Programmierung einsetzen können.
  - Klassen für Objektsammlungen anwenden können
  - Methoden der Klasse ArrayList in der Programmierung verwenden können

Konzepte zum Ablegen und Organisieren von Daten werden in der Informatik als **Datenstrukturen** bezeichnet. In der Objektorientierung werden Datenstrukturen von Objekten betrachtet und als **Objektsammlungen** bezeichnet. **Sammlungsobjekte** sind Objekte, die eine beliebige Anzahl anderer Objekte enthalten können.
Effiziente Datenstrukturen und Algorithmen zu programmieren ist schwierig. Wir müssen uns damit jedoch nicht beschäftigen, da Java in seinen Bibliotheken Klassen enthält, die uns effiziente Implementierungen bieten. Diese Klassen können wir auf die gleiche Weise verwenden, wie wir unsere eigenen Klassen benutzen: Objekte werden durch new-Anweisungen erzeugt. Die Klassen enthalten Attribute, Konstruktoren und Methoden, auf die wir zugreifen können.
Es werden Sammlungen mit fester Größe und solche mit flexibler Größe unterschieden. Sammlungen mit fester Größe können durch ein **Array von Objekten** realisiert werden, das wir in den Kapiteln 6.5 und 10.2 behandelt haben.

## 11.1 Objektsammlungen mit flexibler Größe

In vielen Anwendungen ändert sich die Anzahl der zu verwaltenden Elemente mit der Zeit stark. Dies bedeutet, wir wissen am Anfang noch nicht, wie viele Objekte wir verwalten wollen und es werden während der Programmausführung noch Objekte hinzugefügt und gelöscht. Denken Sie an die Verwaltung der Studierenden an einer Hochschule, die Zahl der zu verwaltenden Studierenden ändert sich ständig. In der **Java-Klassenbibliothek** finden wir mehrere Klassen für Objektsammlungen mit flexibler Größe. Als Beispiel betrachten wir die Klasse **ArrayList**, deren Verwendung wir uns anhand des Projektes *Notizbuch2* (aus „Barnes, Kölling: Java lernen mit BlueJ“ *[7]*; in Felix unter *Vorlesung\Beispiele* ) anschauen wollen. Die Klasse *Notizbuch* soll eine beliebige Anzahl von Notizen (Zeichenketten) speichern können.
Zunächst müssen wir den Zugriff auf die Klasse *ArrayList* aus der Java-Bibliothek deklarieren, dies erfolgt mit

```
import java.util.ArrayList;
```

da sich die Klasse *ArrayList* in dem Paket *util* befindet.
Wenn wir eine Variable einer Objektsammlungsklasse deklarieren wollen, müssen wir zwei Datentypen angeben, den Datentyp der Sammlung und den Datentyp der Elemente, die in der Sammlung gespeichert werden sollen. In dem Notizbuch-Beispiel sollen Notizen gesammelt werden, d.h. die Elemente der Objektsammlung haben den Datentyp *String.* Als Beispiel für eine Objektsammlung wollen wir wie oben beschrieben die *ArrayList* wählen. Eine Variable, in der eine Liste von Notizen gespeichert werden kann wird folgendermaßen deklariert:

```
ArrayList<String> notizen;
```

In den spitzen Klammern steht der Datentyp der Elemente, die in *notizen* gespeichert werden sollen. Solche Klassen werden als **generische Klassen** bezeichnet. Generische Klassen definieren nicht einen einzelnen Typ in Java sondern viele Typen. Sie können auch eine ArrayList von Kreisen oder Ortsvektoren definieren:

```
ArrayList<CKreis> kreise;
ArrayList<COrt2> punkte;
```

In der Objektsammlung *kreise* dürfen nur Objekte vom Typ *CKreis* gespeichert werden, in *punkte* nur Objekte vom Typ *COrt2* und in *notizen* nur Objekte vom Typ *String*.
Zwei Konstruktoren der Klasse *ArrayList* sind für uns von Interesse, einer ohne Parameter und ein zweiter, bei dem wir eine Anfangskapazität für unsere Objektsammlung vorgeben können.
Speicherplatz für unsere Notizen kann also reserviert werden über

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

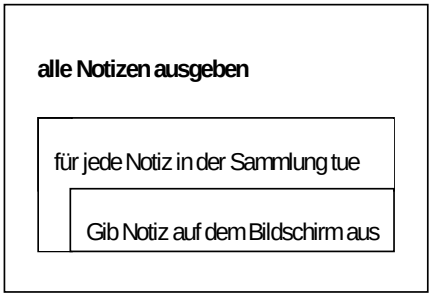
oder

```
notizen = new ArrayList<String>(50);
```

Für die wichtigsten Eigenschaften, die eine Objektsammlung benötigt, bietet die Klasse *ArrayList* Methoden an:

1. Sie kann ihre Kapazität bei Bedarf vergrößern. Mit der Methode *add* können Objekte hinzugefügt werden, z.B.
   ```
   String notiz = "Brot kaufen";
   notizen.add(notiz);
   ```
   Dabei ist zu beachten, dass die Reihenfolge, in der die Elemente eingefügt werden, beibehalten wird und dass **das erste eingefügte Element die Indexnummer 0 besitzt**.
2. Sie merkt sich die Anzahl der aktuell gespeicherten Elemente. Mit
   ```
   notizen.size();
   ```
   kann die Anzahl abgefragt werden.
3. Es können Elemente aus der Objektsammlung gelöscht werden. Mit
   ```
   notizen.remove(2);
   ```
   wird die dritte Notiz aus dem Notizbuch entfernt. Dabei muss beachtet werden, dass sich dadurch die Indexwerte aller nachfolgenden Elemente verändern.
4. Es kann auch direkt auf ein Element zugegriffen werden, wenn man seine Indexnummer kennt:
   ```
   notizen.get(i);
   ```

Wir müssen uns **nicht** darum kümmern, wie diese Klasse und ihre Methoden implementiert sind, wir müssen nur wissen, wie wir sie verwenden können. Allerdings müssen wir, wenn wir mit Indexwerten arbeiten, zuvor überprüfen, ob an der Stelle des angegebenen Indexwertes auch ein Objekt gespeichert ist.
Sollen alle Notizen, die im Notizbuch gespeichert sind ausgegeben werden, können wir das mit einer Schleife erledigen. Wir wollen jedes Element in der Sammlung auf dem Bildschirm ausgeben. Im Struktogramm sieht das folgendermaßen aus:




Da hier auf alle Elemente der Sammlung zugegriffen werden soll, eignet sich die foreach-Schleife, die wir für Arrays schon angewendet haben.

1. Variante um alle Notizen auf der Konsole auszugeben:

```
public void alleNotizenAusgeben()
{
     for (String notiz : notizen)
          System.out.println(notiz);
}
```

Wie bei Arrays wird zunächst die lokale Variable *notiz* angelegt. Dann wird das erste Element der Objektsammlung *notizen* in dieser Variablen gespeichert. Für dieses Element wird dann die Anweisung *System.out.println* ausgeführt. Anschließend wird das zweite Element der Sammlung in der Variablen *notiz* gespeichert und auf dem Bildschirm ausgegeben. Dies wird solange wiederholt, bis das letzte Element in der Sammlung (also die letzte Notiz) auf dem Bildschirm ausgegeben wird. Wichtig ist dabei wieder, dass der Datentyp der lokalen Variable in der foreach-Schleife dem Datentyp der Elemente in der Objektsammlung entspricht. Diese foreach-Schleife ist identisch, wenn *notizen* ein Array ist.
Im folgenden finden Sie das gesamte Projekt *Notizbuch2* aus „Barnes, Kölling: Java lernen mit BlueJ“[7]

```
import java.util.ArrayList;
/**
 * Eine Klasse zur Verwaltung von beliebig langen
 * Notizlisten.
 * Notizen sind nummeriert, um durch einen Benutzer
 * referenziert werden zu können.
 * In dieser Version starten die Notiznummern bei 0.
 * @author David J. Barnes und Michael Kölling.
 * @version 2006.03.30
 */
public class Notizbuch
{
    // Speicher für eine beliebige Anzahl an Notizen.
    private ArrayList<String> notizen;
    /**
     * Initialisierungen
     */
    public Notizbuch()
    {
        notizen = new ArrayList<String>();
    }
    /**
     * Speichere eine neue Notiz in diesem Notizbuch.
     * @param notiz die zu speichernde Notiz.
     */
    public void speichereNotiz(String notiz)
    {
        notizen.add(notiz);
    }
```

```
    /**
     * @return die Anzahl der Notizen in diesem Notizbuch.
     */
    public int anzahlNotizen()
    {
        return notizen.size();
    }
    /**
    * Entferne die Notiz mit der angegebenen Nummer aus
    * diesem Notizbuch, wenn sie existiert.
    * @param notiznummer: Nummer der zu entfernenden Notiz.
    */
    public void entferneNotiz(int notiznummer)
    {
        if(notiznummer < 0) {
            // Keine gültige Nummer, nichts zu tun.
        }
        else if(notiznummer < anzahlNotizen()) {
            // Die Notiznummer ist gültig.
            notizen.remove(notiznummer);
        }
        else {
            // Keine gültige Nummer, nichts zu tun.
        }
    }
    /**
    * Gib alle Notizen dieses Notizbuchs auf die Konsole
    * aus.
    */
    public void notizenAusgeben()
    {
        for (String notiz : notizen) {
            System.out.println(notiz);
        }
    }
} // Ende Klasse Notizbuch
```

***Die Dokumentation generischer Klassen***

Bei **generischen Klassen**, auch parametrisierte Klassen genannt, werden in spitzen Klammern Zusatzinformationen angegeben. Bei der Klasse *ArrayList* haben wir in spitzen Klammern hinter den Klassennamen *String* angegeben, um eine Sammlung von Zeichenketten zu definieren. Wir haben die Klasse *ArrayList* mit dem Typnamen *String* **parametrisiert**. In der Klassendokumentation wird die Parametrisierung allgemein

```
ArrayList<E>
```

geschrieben, E steht für einen beliebigen Klassentyp. Sie finden folgende Methodensignaturen:

```
boolean add(E o)
E get(int index)
```

D.h. der Typ der Objekte, die in eine *ArrayList* mit *add* eingefügt werden dürfen, hängt von dem Typ ab, mit dem die Klasse parametrisiert wurde. Wenn wir ein ArrayList-Objekt des Typs

*ArrayList<String>* erzeugen, sehen die obigen Methoden folgendermaßen aus:

```
boolean add(String o)
String get(int index)
```

Die Methode *get* gibt einen String als Ergebnis zurück. Erzeugen wir dagegen ein Objekt der Typs *ArrayList<COrt2>*, so entsteht die folgende Signatur

```
boolean add(COrt2 o)
COrt2 get(int index)
```

Die Methode *get* gibt nun einen Ortsvektor, also ein Objekt der Klasse *COrt2* zurück.

***Der Iterator***

Da es sehr häufig vorkommt, dass jedes Element einer Sammlung untersucht werden muss, bietet die Klasse *ArrayList* (und auch die anderen Sammlungsklassen) eine explizite Möglichkeit über ihren Inhalt zu iterieren. Die Methode *iterator* der Klasse *ArrayList* liefert als Rückgabe ein Indexobjekt. Ein Indexobjekt ist eine Verallgemeinerung von Indexwerten und wird in Java als Iterator bezeichnet. In Java implementieren alle Iteratoren das Interface *Iterator*, auf Interfaces gehen wir später in Kapitel Fehler: Verweis nicht gefunden genauer ein. Das Interface *Iterator* befindet sich auch in dem Paket *java.util*. Um es zu verwenden, benötigen wir zusätzlich die Import-Anweisung

```
import java.util.Iterator
```

Iteratoren sind ebenfalls generisch, wie die Klasse *ArrayList*, wir müssen also bei der Deklaration eines Iterator-Typs wieder den Datentyp der Elemente angeben, über die iteriert werden soll, z.B.

```
Iterator<String> it1;
Iterator<COrt2> it2;
```

Mit *it1* kann über eine Objektsammlung von Zeichenketten iteriert werden, mit *it2* über eine Objektsammlung von Ortsvektoren.

Das Interface *Iterator* bietet zwei Methoden, um über eine Sammlung zu iterieren:

*hasNext* prüft, ob es noch weitere Elemente in der Sammlung gibt

*next* holt das nächste Objekt aus der Sammlung

Damit können alle Notizen aus dem Notizbuch auch folgendermaßen ausgegeben werden:

2. Variante um alle Notizen auf der Konsole auszugeben:

```
public void alleNotizenAusgeben()
{
     Iterator<String> it = notizen.iterator();
     while (it.hasNext())
     {
          System.out.println(it.next());
     }
}
```

Der Iterator verwaltet selbst die Informationen darüber, wie weit er die Sammlung bereits durchlaufen hat, daher benötigen wir keine Indexvariable, können aber dennoch mit der while-Schleife arbeiten.

### *Vergleich der Schleifentypen*

Auch mit einer while-Schleife können alle Notizen auf der Konsole ausgegeben werden:
3. Variante um alle Notizen auf der Konsole auszugeben:

```
public void alleNotizenAusgeben()
{
     int index = 0;
     while (index < notizen.size())
     {
          System.out.println(notizen.get(index));
          index++;
     }
}
```

Für unser Notizbuch-Beispiel ist die *foreach*-Schleife einfacher und daher besser geeignet. Bei anderen Beispielen, insbesondere, wenn wir nicht wissen, wie oft ein Schleifenrumpf wiederholt werden muss, ist die while-Schleife die bessere Wahl.
Um im Notizbuch eine bestimmte Notiz zu suchen, verwenden wir die Methode *equals* der Klasse *String* aus der Klassenbibliothek, mit der zwei Zeichenketten auf Gleichheit überprüft werden können. Damit erhalten wir die folgende Schleife zum Suchen einer bestimmten Notiz:

```
String gesucht = "DV1 Klausur";
boolean gefunden = false;
int i = 0;
while (
   )
{
     gefunden = gesucht.equals(notizen.get(i))
     i++;
}
if (i < notizen.size())
     System.out.println("Klausur in Notizbuch eingetragen");
else
     System.out.println("Klausur fehlt in Notizbuch");
```

Die ArrayList *notizen* muss jetzt nur solange durchsucht werden, bis die gesuchte Notiz gefunden wird.
Schreiben Sie die while-Schleife so um, dass ein Iterator verwendet wird!

**Vergleichen Sie alle drei Varianten zur Ausgabe aller Notizen!**
Wann sollte nun welche Variante verwendet werden?
Alle drei Varianten scheinen gleich gut zu sein, wobei die erste Variante mit der *foreach-Schleife* am kürzesten zu schreiben ist und auch am einfachsten zu verstehen. Sie sollte auch immer dann verwendet werden, wenn alle Elemente einer Sammlung durchlaufen werden.
Möchte man die Verarbeitung der Objekte jedoch in der Mitte der Sammlung stoppen, ist die while-Schleife zu bevorzugen. Für die *ArrayList* sind diese beiden Varianten auch gleich gut, wobei die Version mit dem Iterator-Objekt den Vorteil hat, dass man sich nicht um den Index kümmern muss.
Die Java-Bibliothek bietet noch eine Vielzahl weiterer Objektsammlungen an, nicht bei allen dieser Objektsammlungen ist der Zugriff über einen Index möglich. Die letzte Variante mit dem Iterator-Objekt funktioniert dagegen bei allen Sammlungstypen. In Fällen, bei denen Elemente der Sammlung gelöscht werden ist die Iterator-Variante die beste Wahl, da sich beim Löschen die Indexwerte der nachfolgenden Elemente ändern.

### Array oder ArrayList

Sammlungen mit fester Größe können durch ein **Array** realisiert werden. Wenn wir zu Beginn wissen, wie viele Elemente wir in einer Objektsammlung speichern wollen, ist ein *Array* einer *ArrayList* vorzuziehen, da es die folgenden Vorteile bietet:

- Die Elemente eines Arrays können nicht nur Objekte sein, sondern auch vom einfachen Datentyp.
- Der Zugriff auf die Elemente in einem Array ist effizienter als bei den meisten Objekt-sammlungen mit flexibler Größe.
- Für Arrays gibt es eine spezielle und einfache Syntax.

Wenn jedoch die Zahl der Elemente nicht fest ist und häufig Elemente hinzugefügt oder gelöscht werden sollen, ist die *ArrayList* die erste Wahl.

## 11.2 Weitere Sammlungs-Klassen

Neben der Klasse ArrayList gibt es noch weitere nützliche Sammlungsklassen, von denen wir hier einige betrachten wollen.

### *Abbildung*

Eine Abbildung ist eine Sammlung von Schlüssel-Wert-Paaren. Ein Wert kann ausgelesen werden, indem ein Schlüssel angegeben wird. Eine typische Anwendung für eine solche Abbildung ist ein Telefonverzeichnis, die Telefonnummern sind die Werte, die Namen die Schlüssel. Eine solche Abbildung repräsentiert die Klasse *HashMap,* eine Spezialisierung von *Map.* Auf die Elemente einer *Map* wird nicht über den Index sondern über den Schlüssel zugegriffen. Untersuchen Sie die Klassendokumentation!
Eine Hashmap besitzt zwei Parameter, einen für den Schlüssel (engl.: key) und einen für den Wert (engl.: value):

*HashMap<K,V>*

Die wichtigsten Methoden der Klasse *HashMap* sind

### *Menge*

Eine **Menge** ist eine Sammlung, in der jedes Element nur einmal enthalten ist. Eine Liste (z.B. *ArrayList* ) dagegen kann dasselbe Element mehrfach enthalten. Die Elemente einer Menge haben keine vorgegebene Ordnung. Es gibt in Java mehrere Klassen, die eine Menge repräsentieren, ein Beispiel ist ***HashSet.*** Wie bei der *ArrayList* kann ein Objekt mit Hilfe der Methode *add* hinzugefügt werden.

### *Keller und Warteschlangen*

Ein Keller ist eine Sammlung, bei der Einfüge-, Lösch- und Leseoperationen auf das Ende der Sammlung beschränkt sind. Man spricht auch von dem **LIFO**-Prinzip (**L**ast **i**n **f**irst **o**ut). Ein Keller wird durch die Klasse *Stack<E>* repräsentiert. Einige Methoden der Klasse *Stack* sind:

| | |
|---|---|
| | fügt ein Element ein |
| | liefert das oberste Element und entfernt es vom Stack |
| | liefert das oberste Element, ohne es zu löschen |

Eine Warteschlange dagegen arbeitet nach dem **FIFO**-Prinzip (**F**irst **i**n **f**irst **o**ut), d.h. Einfügeoperationen werden am Anfang der Warteschlange, Lösch- und Leseoperationen am Ende der Warteschlange durchgeführt. Studieren Sie selber die Methoden der Klasse *PriorityQueue*.

## 11.3 Algorithmen für Objektsammlungen

Bei der Verwaltung von Daten sind häufig vorkommende Aufgaben das Suchen und Sortieren. Sowohl für Arrays als auch für Objektsammlungen mit flexibler Größe, bietet die Java-Bibliothek Algorithmen zum Suchen und Sortieren.

### *Die Klasse Arrays*

Für normale Arrays bietet die Klasse *Arrays* statische Methoden zum Sortieren und Suchen:

*sort(E[] feld)*
*sort(E[] feld, int start, int ende)*
*int binarySearch(E[] feld, E key)*

Der Typ E kann dabei ein elementarer Datentyp oder eine Klasse sein. Ein Aufruf kann beispielsweise so aussehen:

```
double[] feld = new double[100];
// ... Array mit Werten füllen
Arrays.sort(feld);
```

Ist E eine Klasse, so muss diese Klasse das *Comparable*-Interface aus dem Paket *java.lang* implementieren, d.h. eine Methode

*int compareTo(E o)*

zum Vergleich von Objekten dieser Klasse bereitstellen. Diese Methode muss 0 bei Gleichheit, einen Wert < 0, falls das übergebene Objekt größer ist als das die Methode aufrufende Objekt und einen Wert > 0, falls das übergebene Objekt kleiner ist als das die Methode aufrufende Objekt, als Ergebnis liefern. Viele Standardklassen von Java haben eine solche Methode schon definiert.
In der Methode *binarySearch* ist *key* der Wert, nachdem gesucht werden soll. Wenn das Element gefunden wurde, ist der Rückgabewert der Index des gesuchten Eintrags im Array, ansonsten ist das Ergebnis < 0. Um *binarySearch* aufrufen zu können, muss das Array sortiert sein!

### *Die Klasse Collections*

Für Listen bietet die Klasse *Collections* statische Methoden. Suchen Sie in der Klassendokumentation Methoden zum Sortieren von Listen und zum Suchen in Listen, sowie zur Bestimmung des kleinsten und größten Werts innerhalb einer Objektsammlung!

***Fragen:***

- ✗ Was ist der Unterschied zwischen Arrays von einfachen Datentypen und von Objekten?
- ✗ In welchem Java-Paket befinden sich die Sammlungs-Klassen?
- ✗ Welche Typen müssen für Sammlungen angegeben werden?
- ✗ Wie viele Elemente einer Sammlung werden mit der for-each-Schleife durchlaufen
- ✗ Welchen Vorteil bietet der Iterator gegenüber einer while- oder for-Schleife? In welcher Situation wird dieser Vorteil besonders deutlich?
- ✗ Wann sollte beim Durchlaufen einer Sammlung die for-each-Schleife verwendet werden, wann der Iterator?
- ✗ Was unterscheidet die Klasse *HashMap* von den meisten anderen Objektsammlungen
- ✗ Nennen Sie zwei wichtige Methoden der Klasse Stack

# 12 Vererbung

***Lernziele:***

- **Wissen**
  - Wissen, wie die Vererbung in Java umgesetzt wird
  - Wissen, was Polymorphismus bedeutet
  - Wissen, was eine abstrakte Klasse ist
- **Verstehen**
  - Verstehen, wann Komposition und wann Vererbung eingesetzt wird
  - Verstehen, wie Methoden und Konstruktoren überschrieben werden
  - Erklären können, wo und wie auf überschriebene Methoden einer Basisklasse zugegriffen werden kann
  - Verstehen, wozu abstrakte Klassen verwendet werden
- **Anwenden**
  - Vererbung sinnvoll in eigenen Programmen einsetzen können
  - Vererbung mit abstrakter Basisklasse sinnvoll in eigenen Programmen einsetzen können

In der Analyse haben wir verschiedene Beziehungen zwischen Klassen kennengelernt:
- **Assoziation**(allgemein): Ein Objekt einer Klasse verwendet ein oder mehrere Objekte einer anderen Klasse, auch »verwendet ein«-Beziehung genannt.
- **Aggregation/Komposition**(genauere Spezifizierung der Assoziation): Ein Objekt einer Klasse besteht aus einem oder mehreren Objekten einer anderen Klasse, auch »hat ein«-Beziehung genannt.
- **Vererbung**: Eine abgeleitete Klasse ist eine Basisklasse, auch »ist ein«-Beziehung genannt.

**Alle Assoziationen im Klassendiagramm beschreiben Beziehungen zwischen Objekten, die Vererbung ist eine Beziehung zwischen Klassen!** Bei der Vererbung wird daher nur 1 Objekt erzeugt, es bekommt jedoch Attribute aus mehreren Klassen.
Im folgenden wird auf die Umsetzung der Vererbung in der Programmierung mit Java näher eingegangen.

## 12.1 Das Grundprinzip der Vererbung

Das Grundprinzip der Vererbung ist, dass man sowohl Eigenschaften als auch Methoden vorhandener Klassen wiederverwenden kann (siehe Kapitel 9.3). Die Methoden können dabei auch verändert werden und es können neue Methoden und Eigenschaften hinzugefügt werden. Dazu wird von einer vorhandenen Klasse, die Basisklasse genannt wird, eine neue Klasse abgeleitet. Die neue Klasse verfügt automatisch über alle Eigenschaften der Basisklasse, d.h. sie kann alle Attribute und Methoden der Basisklasse verwenden.
Die Basisklasse wird auch Superklasse oder übergeordnete Klasse genannt.
Die neue Klasse heißt abgeleitete Klasse, Subklasse oder untergeordnete Klasse.
Am gebräuchlichsten in Java sind die Begriffe **Superklasse** und **Subklasse**.

Beispiel 12.1:
Wir betrachten die Klasse Student, die wir in Kapitel 9.3 eingeführt haben, in etwas vereinfachter Form.

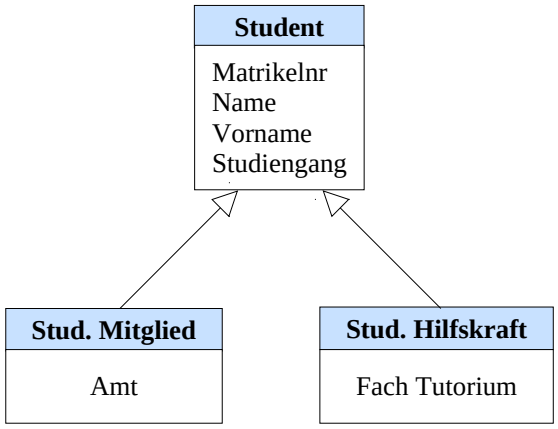
| **Student** |
|---|
| Matrikelnr<br>Name<br>Vorname<br>Studiengang |

Manche Studenten sind Semestersprecher, Mitglieder in der Studienkommission oder in einem anderen Gremium, andere betreuen ein Tutorium und sind dazu als studentische Hilfskräfte angestellt. Um dies abzubilden führen wir zusätzlich die Klassen studentisches Mitglied und studentische Hilfskraft ein:

| **Stud. Mitglied** |
|---|
| Matrikelnr<br>Name<br>Vorname<br>Studiengang<br>Amt |

| **Stud. Hilfskraft** |
|---|
| Matrikelnr<br>Name<br>Vorname<br>Studiengang<br>Fach Tutorium |

Alle drei Klassen haben die Attribute *Matrikelnr*, *Name*, *Vorname* und *Studiengang* gemeinsam, jedoch haben die unteren beiden Klassen jeweils ein zusätzliches Attribut. Um die gemeinsamen Attribute nicht dreimal implementieren zu müssen, kann Vererbung eingesetzt werden. Im Klassendiagramm sieht das dann folgendermaßen aus:




Um in Java darzustellen, dass eine Klasse von einer anderen abgeleitet wird, wird das Schlüsselwort **extends** verwendet.
Zunächst wird die Basisklasse *Student* erzeugt und von ihr werden die anderen Klassen abgeleitet:

```
public class Student
{
    private int Matrikelnr;
    private String Name;
    private String Vorname;
    private String Studiengang;
}

public class Mitglied extends Student
```

```
    {
        private String Amt;
    }

    public class Hilfskraft extends Student
    {
        private String Fach_Tut;
    }
```

Die Klassen *Mitglied* und *Hilfskraft* erben die Attribute *Matrikelnr, Name, Vorname* und *Studiengang* der Klasse *Student*. Wird ein Objekt der Klasse *Mitglied* angelegt, so wird Speicherplatz für alle diese Attribute reserviert. Damit die abgeleiteten Klassen auf die Attribute zugreifen können, müssen entweder get-Methoden programmiert werden oder es muss die Sichtbarkeit *protected* gesetzt werden, die besagt, dass das Attribut innerhalb der Klasse und in allen abgeleiteten Klassen verwendet werden darf.

### *Zugriffsspezifikationen und Vererbung*

Wir hatten bisher die beiden Sichtbarkeiten **private** und **public** kennen gelernt. In Zusammenhang mit Vererbung gibt es noch die Sichtbarkeit
**protected:** sichtbar innerhalb der Klasse und in deren Subklassen
Für Attribute sollte *protected* möglichst nicht verwendet werden, für Operationen kann es sinnvoll sein.
Durch richtigen Einsatz dieser Zugriffsspezifikationen können Implementierungen in einer Klasse geändert werden, ohne dass der Anwender der Klasse sein Programm ändern muss.
Wir erweitern die Klasse *Student* noch um zwei Methoden:

```
public class Student
{
   protected int Matrikelnr;
   protected String Name;
   protected String Vorname;
   protected String Studiengang;

   // Konstruktor
   public Student(int m, String n, String vn, String st)
   {
       Matrikelnr = m;
       Name = n;
       Vorname = vn;
       Studiengang = st;
   }
    // Beispielmethoden
   public void SetName(String name)
   {
       Name = name;
   }
```

```
    public void Ausgabe()
    {
      System.out.println("Matrikelnummer: " + Matrikelnr);
      System.out.println("Name: " + Name);
      System.out.println("Vorname: " + Vorname);
      System.out.println("Studiengang: " + Studiengang);
    }
} // Ende Klasse Student
```

Diese Methoden können jetzt von den abgeleiteten Klassen *Mitglied* und *Hilfskraft* direkt verwendet werden.
Es ist wichtig sich bei der Auswahl der Attribute und Operationen der Basisklasse auf das Wesentliche zu beschränken, damit die Basisklasse so allgemein wie möglich bleibt. Für alle abgeleiteten Klassen müssen die Eigenschaften ihrer Superklasse gültig sein.
Neben der Verwendung der Attribute und Methoden der Superklasse, kann eine Subklasse

- Methoden der Superklasse **überschreiben** und
- neue Attribute und Methoden definieren

Die Klasse *Mitglied* benötigt z.B. eine Methode zum Setzen des Amts und möchte bei der Ausgabe auch das Amt mit ausgeben. Die Klasse *Mitglied* kann folgendermaßen erweitert werden:

```
public class Mitglied extends Student
{
    private String Amt;
    // Methoden
    public void SetAmt(String a)
    {
        Amt = a;
    }
    public void Ausgabe()
    {
       System.out.println("Matrikelnummer: " + Matrikelnr);
       System.out.println("Name: " + Name);
       System.out.println("Vorname: " + Vorname);
       System.out.println("Studiengang: " + Studiengang);
       System.out.println("Amt in Gremium: " + Amt);
       }
} // Ende Klasse Mitglied
```

Was fällt Ihnen an der Methode Ausgabe auf?

### *Methoden und Konstruktoren überschreiben*

Gibt es in der Superklasse und in den Subklassen je eine Methode, deren Signaturen exakt übereinstimmen und die sich nur in ihrem Rumpf unterscheiden, so sagt man:
**Die Methode der Superklasse ist von der Methode der Subklasse überschrieben**.
Das Überschreiben einer Methode hat keine Auswirkungen auf die Basisklasse. Vererbung wirkt nur in eine Richtung. Der Compiler erzeugt Kopien der geerbten Attribute für die abgeleitete Klasse und die Aufrufe der geerbten Methoden werden in Aufrufe der Methoden der Basisklasse umgewandelt.
Wenn eine Methode überschrieben wird, lädt der Compiler bei einem Aufruf der Methode die neue Version der abgeleiteten Klasse.
Die geerbte Methode der Basisklasse ist allerdings immer noch vorhanden, nur nicht mehr mit dem

Methodennamen verbunden. Soll sie aus der abgeleiteten Klasse heraus aufgerufen werden, so muss dem Methodennamen das Schlüsselwort **super** vorangestellt werden.
Damit kann die in der Klasse *Mitglied* überschriebene Methode *Ausgabe* kürzer geschrieben werden:

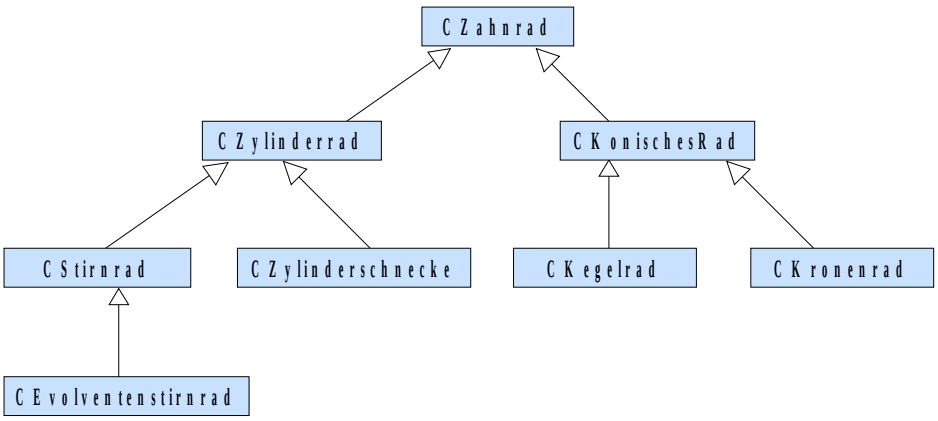
```
public void Ausgabe()
{
// Aufruf der Methode Ausgabe der Klasse Student
    super.Ausgabe();
    System.out.println("Amt in Gremium: " + Amt);
}
```

Zur Entscheidung, ob in der abgeleiteten Klasse eine neue Methode implementiert oder eine Methode der Basisklasse überschrieben werden soll, muss wieder objektorientiert gedacht werden: Sollen die Verhaltensweisen aus der Basisklasse abgewandelt werden oder handelt es sich um ganz neue Verhaltensweisen? Jede von *Student* abgeleitete Klasse soll ihre Attribute ausgeben, dies ist eine gemeinsame Verhaltensweise aller Klassen. Sie unterscheidet sich lediglich darin, welche Größen ausgegeben werden. Das Setzen des Attributs *Amt* in der Klasse *Mitglied* dagegen ist eine neue Verhaltensweise der Klasse *Mitglied*.
Auch Konstruktoren können überschrieben werden. Der Konstruktor der Basisklasse wird direkt mit **super** aufgerufen. **Der Aufruf eines Konstruktors der Superklasse muss die erste Anweisung im Konstruktor der Subklasse sein** (ansonsten entsteht ein Compilerfehler). Wird im Quelltext kein Konstruktor der Superklasse aufgerufen, versucht Java automatisch einen parameterlosen Aufruf (Standardkonstruktor) einzufügen. Daher sollte auch für jede Klasse der Standardkonstruktor definiert sein.
Damit kann ein Konstruktor für die Klasse *Mitglied* definiert werden:

```
public Mitglied(int m, String n, String v, String st, String a)
{
    super(m, n, v, st);
    Amt = a;
}
```

### *Vererbungshierarchien*

Von Subklassen können weitere Klassen abgeleitet werden. Die Sammlung aller Klassen, die über Vererbungsbeziehungen miteinander verknüpft sind, bilden eine **Vererbungshierarchie**. Ich möchte Ihnen hier eine Klassenhierarchie vorstellen, die in einem Berechnungsprogramm für Verzahnungen verwendet wird:



### *Subtyping*

Wir wollen nun in einem Array mehrere unterschiedliche Zahnräder speichern. Welchen Typ sollte dieses Array haben? Dazu einige Konzepte (aus [8]):
**Subtyp:** Analog zur Klassenhierarchie bilden die Objekttypen eine Typhierarchie. Der Typ, der durch eine Subklasse definiert ist, ist ein Subtyp des Typs, der durch die zugeordnete Superklasse definiert wird.
**Variablen und Subtypen:** Eine Variable kann ein Objekt halten, dessen Typ entweder gleich dem deklarierten Typ der Variablen oder ein beliebiger Subtyp des deklarierten Typs ist.
**Ersetzbarkeit:** Objekte von Subtypen können an allen Stellen verwendet werden, an denen ein Supertyp erwartet wird.
Somit können wir ein Array vom Typ *CZahnrad* anlegen und darin alle Objekte der von *CZahnrad* abgeleiteten Klassen speichern.

```
CZahnrad[] liste = new CZahnrad[4];
liste[0] = new CKegelrad();
liste[1] = new CEvolventenstirnrad();
liste[2] = new CZylinderschnecke();
liste[3] = new CKonischesRad();
```

Betrachten wir noch die folgenden beiden Konzepte:
Der **statische Typ** einer Variablen ist der Typ, mit dem die Variable im Quelltext der Klasse deklariert wurde.
Der **dynamische Typ** einer Variablen v ist der Typ des Objekts, das zur Zeit in der Variablen v gehalten wird.
Bei den folgenden Beispielen

```
CZahnrad z1 = new CZahnrad();
CZahnrad z2 = new CStirnrad();
```

hat z1 den statischen und den dynamischen Typ *CZahnrad*, z2 hat den statischen Typ *CZahnrad*, jedoch den dynamischen Typ *CStirnrad.*
Der dynamische Typ einer Variablen kann sich während des Programmablaufs ändern.

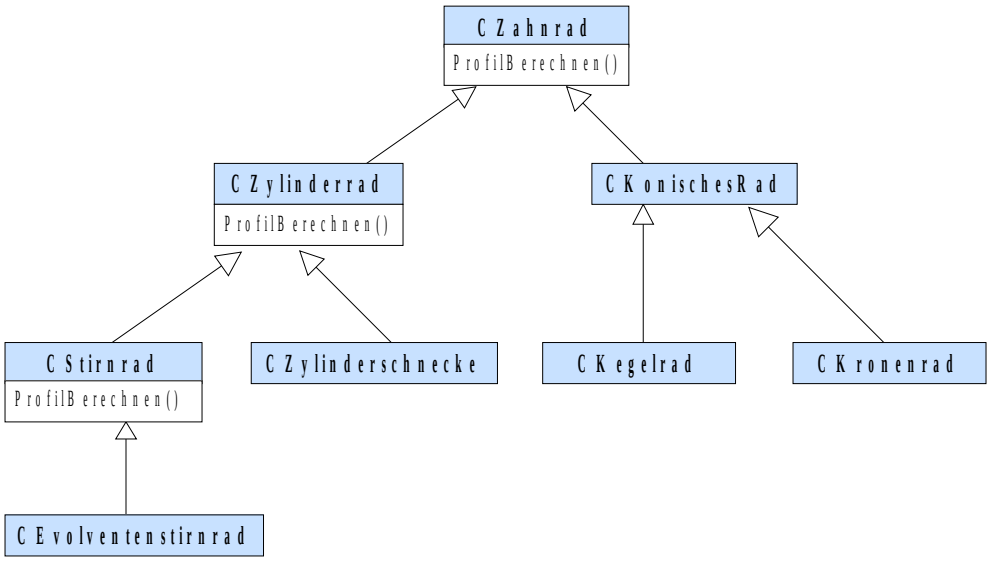
## 12.2 Polymorphismus

Polymorphismus heißt kurz: Objekte kennen ihre Bestimmung. Im folgenden soll genauer untersucht werden, was passiert, wenn ein Methodenaufruf auf die Objekte verschiedener Klassen in einer Vererbungshierarchie angewandt wird.
Wird eine Methode einer Subklasse mit bestimmten Parametern aufgerufen passiert folgendes:

- Die Subklasse prüft, ob sie über eine Methode mit diesem Namen und genau den gleichen Parametern (d.h. gleiche Anzahl und gleicher Typ) verfügt. Ist dies der Fall, ruft sie die Methode auf.
- Wenn nicht, dann geht die Verantwortlichkeit zum Aufruf der Methode an die übergeordnete Klasse weiter. Diese sucht wiederum nach einer Methode mit diesem Namen und den Parametern. Ist eine solche Methode vorhanden, wird sie von dieser Klasse aufgerufen.
- Auf diese Weise geht die Verantwortlichkeit gegebenenfalls in der Vererbungskette weiter nach oben, bis eine übereinstimmende Methode gefunden wird oder das Ende der Vererbungskette erreicht ist. (Wird keine passende Methode gefunden, entsteht ein Laufzeitfehler)

Da Methoden mit dem gleichen Namen auf verschiedenen Ebenen der Kette existieren können braucht man folgende **Regel der Vererbung**:
Eine Methode, die in einer Subklasse überschrieben wird (gleicher Name und gleiche Parameterliste) verdeckt die Methode der übergeordneten Klasse gegenüber der Subklasse.
Achtung: Es dürfen keine Methoden mit gleichem Namen, gleicher Parameterliste, aber unterschiedlichem Rückgabetyp implementiert werden.
Sei nun eine Methode *ProfilBerechnen* in der Klasse CZahnrad implementiert und in folgenden

Klassen der Vererbungshierarchie überschrieben:




Welche Methode wird nun bei den folgenden Aufrufen verwendet?

```
CKronenrad.profilBerechnen()          → _____
CStirnrad.profilBerechnen()           _____
CZylinderschnecke.profilBerechnen()   _____
CEvolventenstirnrad.profilBerechnen() → _____
```

Welchen Vorteil hat dieses Konzept?
Betrachten wir das auf Seite 113 definierte Array *liste*, das unterschiedliche Zahnräder enthält. Möchte man für alle Zahnräder das Profil berechnen, so kann dies mit folgender Schleife erfolgen:

```
for (int i=0;i<4;i++)
  liste[i].profilBerechnen();
```

In jedem Schleifendurchlauf muss nun die jeweils passende Methode ausgeführt werden. Man spricht hier von **Methoden-Polymorphie**: Derselbe Methodenaufruf kann zu unterschiedlichen Zeitpunkten verschiedene Methoden aufrufen, abhängig vom dynamischen Typ der Variablen.

In diesem Beispiel weiß der Compiler welches Objekt in welchem Listenelement steckt. Es könnte aber auch sein, dass die Zahnräder über eine Eingabeoberfläche eingelesen werden. Dann ist erst während der Laufzeit bekannt welches Objekt in welchem Listenelement steckt.
Die Umsetzung dieser Arbeitsweise des Polymorphismus heißt **späte Bindung:** Der Compiler erzeugt nicht zur Compilierzeit den Code zum Aufruf der Methode, sondern erst zur Laufzeit. Bei jeder Anwendung einer Methode auf ein Objekt wird spezieller Code erzeugt, der anhand der Typinformationen des Objekts die tatsächlich aufzurufende Methode bestimmt. Man spricht auch von **dynamischer Bindung.**

## 12.3 Konstante und abstrakte Klassen

Konstante Variablen wurden in Kapitel 3.2 schon verwendet:

```
final double PI = 3.14159;
```

Steht das Schlüsselwort *final* vor einer Klasse, so darf von dieser Klasse nicht abgeleitet werden. Das Schlüsselwort *final* vor einer Methodendefinition, bewirkt, dass man die Methode nicht ändern darf. Das bedeutet, dass *final*-Methoden nicht in abgeleiteten Klassen überschrieben werden können.

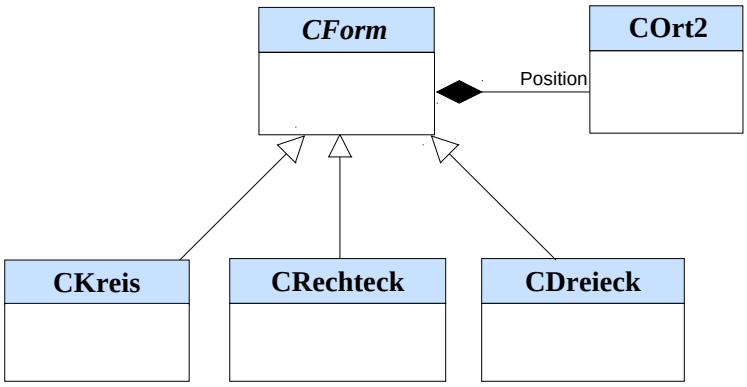
Umgekehrt könnte es auch interessant sein, das Überschreiben einer Methode zu erzwingen.
Dazu betrachten wir ein weiteres Beispiel.
Beispiel 12.2:
Wir betrachten die Formen Kreis, Rechteck und Dreieck. Es werden also die Klassen *CKreis*, *CRechteck* und *CDreieck* benötigt. Im Klassendiagramm sieht das zunächst so aus:

| **CKreis** |
|---|
| Position<br>Radius<br>Farbe |
| zeichnen() |

| **CRechteck** |
|---|
| Position<br>Länge<br>Breite<br>Farbe |
| zeichnen() |

| **CDreieck** |
|---|
| Position<br>Seiten<br>Winkel<br>Farbe |
| zeichnen() |

Die drei Klassen haben die gemeinsamen Eigenschaften Position und Farbe und eine gemeinsame Methode. Es ist also sinnvoll eine Basisklasse zu entwerfen mit den Attributen Position und Farbe. Für die Position verwenden wir die schon bekannte Klasse *COrt2*. Von dieser Basisklasse werden die Klassen *CKreis*, *CRechteck* und *CDreieck* abgeleitet, was im OOA-Modell nur mit Attributen folgendermaßen dargestellt wird:




Die Methode *zeichnen* soll für alle Formen das Zeichnen realisieren. Daher macht es Sinn, diese Methode in der Klasse *CForm* zu deklarieren und in den abgeleiteten Klassen zu überschreiben. Die Klasse *CForm* hat jedoch nicht genügend Eigenschaften definiert, um das Zeichnen auszuführen, d.h. jede von *CForm* abgeleitete Klasse sollte eine eigene Methode *zeichnen* implementieren.
Es macht eigentlich auch gar keinen Sinn ein Objekt von der Basisklasse *CForm* zu erzeugen. In einem solchen Fall werden **abstrakte Methoden** in einer **abstrakten Basisklasse** deklariert.
Unsere Klasse *CForm* sieht dann folgendermaßen aus (die Farbe wird im folgenden zunächst weg gelassen):

```
abstract public class CForm
{
  private COrt2 pos;
  public CForm(double x, double y)
  {
      pos = new COrt2(x,y);
  }
  abstract public void zeichnen();
}
```

**Von abstrakten Klassen dürfen keine Objekte gebildet werden!**
Die Definition einer abstrakten Methode besteht aus einer Methodensignatur ohne Rumpf. Sie wird mit dem Schlüsselwort *abstract* markiert.

Abstrakte Methoden einer Basisklasse müssen in abgeleiteten Klassen überschrieben werden. Ansonsten wird die abgeleitete Klasse selbst zu einer abstrakten Klasse.
Wozu braucht man überhaupt die abstrakte Methode *zeichnen* in der Klasse CForm?

- Da alle verschiedenen Formen (Objekte von abgeleiteten Klassen von *CForm*) gezeichnet werden sollen, benötigt jede abgeleitete Klasse eine Methode *zeichnen*. Die abstrakte Methode *zeichnen* in der abstrakten Basisklasse *CForm* erzwingt, sie in jeder abgeleiteten Klasse zu überschreiben.
- Sollen viele Formen gezeichnet werden, ist es sinnvoll diese in einem Array zu verwalten (siehe Beispiel mit den Zahnradklassen). Um Objekte von allen abgeleiteten Klassen in dem Array speichern zu können, muss der Datentyp der Arrayelemente *CForm* sein. Die Methode *zeichnen* kann dann für ein Array-Element nur aufgerufen werden, wenn sie in der Basisklasse definiert ist.

Hieraus ergibt sich eine wichtige Regel für den Entwurf von Klassen:
Alle Verhaltensweisen, die den abgeleiteten Klassen gemeinsam sind, sollten bereits als Methode in der gemeinsamen Basisklasse deklariert werden. Es ist jedoch nicht erforderlich für alle diese Methoden Implementierungen vorzugeben.
Die Klassen *CKreis* und *CRechteck* werden jetzt von der abstrakten Klasse *CForm* abgeleitet und über ein Array sollen die *zeichnen*-Methoden aufgerufen werden:

```
// Abgeleitete Klasse
public class CKreis extends CForm
{
  private double r;  // Radius
  public CKreis(double x, double y, double r)
  {
      super(x,y);
      this.r = r;
  }
  public void zeichnen()
  {
      System.out.println("Zeichne Kreis");
  }
}
// Abgeleitete Klasse
public class CRechteck extends CForm
{
  private double b, l; // Breite, Laenge
  public CRechteck(double x, double y, double l, double b)
  {
      super(x,y);
      this.l = l;
      this.b = b;
  }
  public void zeichnen()
  {
      System.out.println("Zeichne Rechteck");
  }
}
```

```
public class CZeichnen
{
   private CForm[] formen;
   public CZeichnen()
   {
      formen = new CForm[4];  // Speicherpl. für 4 Formen
      formen[0] = new CKreis(0,0,30);
      formen[1] = new CRechteck(5,10,50,50);
      formen[2] = new CKreis(20,30,10);
      formen[3] = new CRechteck(50,20,100,30);
   }
   public void zeichnen()
   {
        for(int i=0;i<formen.length;i++)
           formen[i].zeichnen();
   }
} // Ende Klasse Zeichnen
```

## 12.4 Die Oberklasse *Object*

Die Klasse *Object* befindet sich in dem Paket java.lang. Sie nimmt eine Sonderstellung ein, denn sie ist allen Klassen (auch den selbst definierten Klassen) übergeordnet. Es sind also alle Klassen in Java von der Klasse *Object* abgeleitet, man kann in selbst definierten Klassen die Methoden der Klasse *Object* verwenden und überschreiben. Außerdem kann ein Array vom Datentyp *Object* gebildet werden, und dort Objekte der verschiedensten Klassen abgespeichert werden, z.B.

```
Object[] meineObjekte = new Object[4];
meineObjekte[0] = new CKreis(0,0,50);
meineObjekte[1] = new CVorlesung("DV2",2,2);
meineObjekte[2] = new CRechteck(5,20,30,46);
meineObjekte[3] = new CStudent(34572,"Meier","Markus","ME");
```

Wichtige Methoden der Klasse *Object*:

- *public String toString()*
  Diese Methode liefert einen *String* zurück, der das Objekt der Klasse beschreiben soll. Es wird empfohlen, diese Methode in eigenen Klassen zu überschreiben, so dass sie eine textliche Beschreibung des Objektes zurück liefert. Wurde für alle oben verwendeten Klassen die Methode *toString* überschrieben, so kann der Inhalt aller Objekte mit Hilfe einer Schleife ausgegeben werden:

  ```
  for (int i=0;i<meineObjekte.length;i++)
      System.out.println(meineObjekte[i]);
  ```

  Übergibt man ein Objekt an die Operation System.out.println, ruft diese Operation die Operation toString auf und gibt den Ergebnisstring aus.
- *public boolean equals(Object obj)*
  Mit der Anweisung
  *meineObjekte[0] == meineObjekte[1];*
  werden die beiden Referenzen verglichen, d.h. es wird geprüft, ob sie sich auf dasselbe Objekt beziehen. Möchte man den Inhalt zweier Objekte vergleichen, so muss man die Methode *equals* der Klasse *Object* in der eigenen Klasse überschreiben. Dabei muss der Anwender

entscheiden, unter welcher Bedingung zwei Objekte gleich sind. Zwei Kreise seien gleich, wenn ihre Radien gleich sind. D.h.in der Klasse *CKreis* benötigt man die Methode

```
public boolean equals (CKreis k)
{
     return (r == k.r);
}
```

Dann können zwei Kreise verglichen werden:

```
CKreis k1 = new CKreis(0,0,35);
CKreis k2 = new CKreis(5,10,35);
CKreis k3 = new CKreis(0,0,5.5);
boolean erg1,erg2;
erg1 = k1.equals(k2);
erg2 = k1.equals(k3);
```

*erg1* liefert *true*, *erg2* liefert *false*.

## 12.5 Pakete

Bei einfachen Programmen mit wenigen Klassen ist es kein Problem, alle Klassen im selben Verzeichnis abzulegen. Der Compiler erzeugt für jede definierte Klasse eine eigene binäre .class-Datei. Solange die binäre .class-Datei der zu verwendenden Klasse im gleichen Verzeichnis wie das aktuelle Programm liegt, kann die Klasse verwendet werden.
Soll jetzt aber eine der implementierten Klassen in einem anderen Programm verwendet werden, oder liegt ein sehr umfangreiches Programm mit vielen Klassen vor, so wäre es unpraktisch und unübersichtlich, wenn alle benötigten Klassen in dem selben Verzeichnis stehen müssten.
Um eine größere Menge an selbst implementierten Klassen zu verwalten dient in Java das Konzept der Pakete:
Pakete verwalten Klassen und werden aus Quelltextdateien aufgebaut. Steht am Anfang einer Quelltextdatei die Anweisung

package paketname;

so werden alle Klassen, die in dieser Datei definiert werden, dem Paket *paketname* zugeordnet.
Um Klassen aus einem bestimmten Paket zu verwenden gibt es mehrere Möglichkeiten:

- dem Klassennamen den Paketnamen voranstellen:
  paketname.klassenname.methode1();
- importieren der zu verwendenden Klasse, dann kann die Klasse ohne Paketname verwendet werden:
  import paketname.klassenname;
  klassenname.methode1();
- importieren des gesamten Pakets, alle Klassen aus dem Paket können verwendet werden
  import paketname.*;
  klassenname.methode1();

Für jedes Paket muss ein eigenes gleichnamiges Verzeichnis angelegt werden, in dem alle .class-Dateien der Klassen des Pakets liegen. Der Compiler erfährt den Ort der Pakete durch den sogenannten Klassenpfad, der alle Ordner enthält, indem sich eigene Pakete befinden. Entweder wird der Klassenpfad als Compileroption gesetzt:

javac –classpath *Klassenpfad* bsp.java

oder der Klassenpfad wird als Umgebungsvariable gesetzt (Systemsteuerung, Systemeigenschaften, Umgebungsvariablen: Umgebungsvar. CLASSPATH setzen).
Wird eine Quelltextdatei nicht explizit einem bestimmten Paket zugeordnet, so gehören die Klassen dieser Datei einem sogenannten Standardpaket an.

***Fragen:***

- ✗ Mit welchem Schlüsselwort wird die Vererbung implementiert?
- ✗ Wie kann auf überschriebene Methoden einer Basisklasse zugegriffen werden?
- ✗ Was versteht man unter Ersetzbarkeit im Zusammenhang mit Vererbung?
- ✗ Was ist der Unterschied zwischen dem statischen und dem dynamischen Typ?
- ✗ Wie werden abstrakte Methoden definiert, woraus bestehen sie?
- ✗ Wozu werden abstrakte Methoden verwendet?

# *13 Objektorientierte Analyse – dynamische Konzepte*

***Lernziele:***

- **Verstehen**
  - Erklären können, was ein Geschäftsprozess ist
  - Erklären können, was eine Botschaft ist
  - Erklären können, was ein Aktivitätsdiagramm ist
  - Erklären können, wie das Klassendiagramm und Diagramme des dynamischen Modells zusammenwirken
- **Anwenden**
  - Geschäftsprozesse modellieren können
  - Geschäftsprozesse spezifizieren können

Dieses Kapitel dient zum besseren Verständnis der Modellierung von Programmabläufen und der Rolle, die Operationen in diesem Zusammenhang spielen.

## 13.1 Geschäftsprozess

Ein **Geschäftsprozess (use case)** besteht aus mehreren zusammenhängenden Aufgaben, die von einem Akteur durchgeführt werden, um ein Ziel zu erreichen. Der Begriff „use case“ wird in der Literatur teilweise auch als „Anwendungsfall“ übersetzt.
Ein **Akteur** ist dabei eine Rolle, die ein Benutzer des Systems spielt. Ein Akteur kann eine Person, eine Organisation oder ein externes System, das mit dem zu modellierenden System kommuniziert, sein. Akteure befinden sich stets außerhalb des Systems.
Beispiel 13.1: Wir betrachten ein Softwaresystem, welches das Auftrags- und Bestellwesen eines Versandhauses unterstützt. Akteure sind hier die Kunden-, Lieferanten- und Lagersachbearbeiter und die Buchhaltung. Auch wenn in einer kleineren Firma die Aufgaben von Kunden- und Lieferantensachbearbeiter von der selben Person ausgeführt werden, werden zwei Akteure – die Rollen, die diese Person spielt – identifiziert.
Häufig ist die Identifikation der Geschäftsprozesse der erste Schritt in der Analyse. Hier soll ermittelt werden, welche Aufgaben mit dem neuen Softwaresystem zu bewältigen sind, um die gewünschten Ergebnisse zu erzielen. Der Geschäftsprozess spezifiziert sozusagen die Interaktion zwischen einem Akteur und dem System, d.h. er beschreibt eine spezielle Verwendung des Systems. Alle Geschäftsprozesse zusammen dokumentieren alle Möglichkeiten der Verwendung des Systems. Ein Geschäftsprozess wird semiformal oder umgangssprachlich beschrieben. Man spricht auch von der **Spezifikation** des Geschäftsprozesses.
Die folgende Checkliste kann beim Erstellen einer Spezifikation hilfreich sein:

| | |
|---|---|
| *Ziel:* | globale Zielsetzung bei erfolgreicher Ausführung des Geschäftsprozesses |
| *Vorbedingung:* | Erwarteter Zustand, bevor der Geschäftsprozess beginnt |
| *Nachbedingung Erfolg:* | Erwarteter Zustand nach erfolgreicher Ausführung des Geschäftsprozesses. |
| *Nachbedingung Fehlschlag:* | Erwarteter Zustand, wenn das Ziel nicht erreicht werden kann. |
| *Akteure:* | Rollen der Akteure, die den Geschäftsprozess auslösen |
| *Beschreibung in Teilschritten mit Erweiterungen und Alternativen:* | |

Beispiel 13.2: Wir betrachten die Bearbeitung eines Auftrags in einem Versandhaus, die wir unter dem Geschäftsprozeß *Auftrag ausführen* zusammenfassen. Diesen Geschäftsprozeß wollen wir

zunächst umgangssprachlich und danach mittels obiger Checkliste beschreiben:

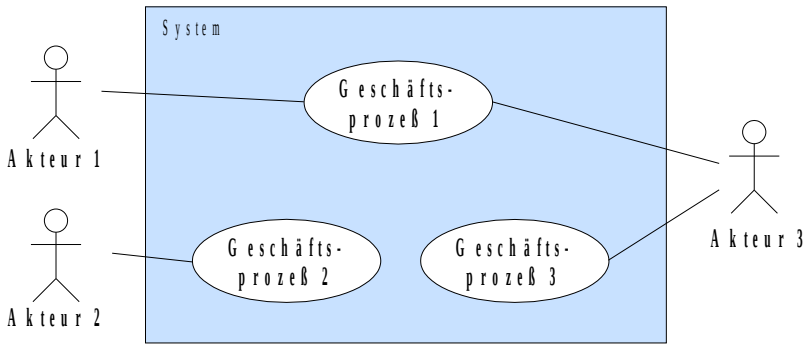
Umgangssprachlich:

Neukunden werden im System registriert und der Versand an diese Kunden erfolgt ausschließlich per Nachnahme oder Bankeinzug. Für alle lieferbaren Artikel wird die Rechnung erstellt und als Auftrag an das Lager weitergegeben. Sind einige der gewünschten Artikel nicht lieferbar, so wird der Kunde informiert. Alle erstellten Rechnungen werden an die Buchhaltung weitergegeben. An diesem Geschäftsprozeß sind die Akteure Kundensachbearbeiter, Lagersachbearbeiter und Buchhaltung beteiligt.

Mittels Checkliste:

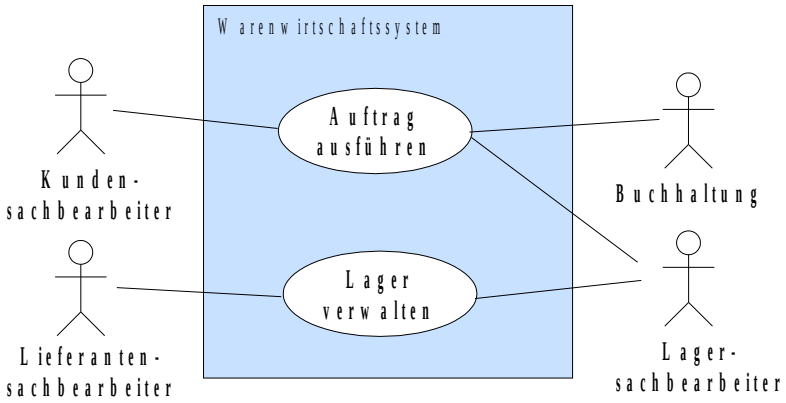
| | |
|---|---|
| *Geschäftsprozeß*: | Auftrag ausführen |
| *Ziel:* | Ware an Kunde geliefert |
| *Vorbedingung:* | - |
| *Nachbedingung Erfolg:* | Ware ausgeliefert (auch Teillieferungen), Rechnungskopie bei Buchhaltung |
| *Nachbedingung Fehlschlag:* | Mitteilung an Kunden, dass nichts lieferbar ist |
| *Akteure:* | Kundensachbearbeiter, Lagersachbearbeiter und Buchhaltung |
| *Beschreibung in Teilschritten mit Erweiterungen und Alternativen:* | **1** Kundendaten abrufen<br>**2** Lieferbarkeit prüfen<br>**3** Rechnung erstellen<br>**4** Auftrag vom Lager ausführen lassen<br>**5** Rechnungskopie an Buchhaltung geben<br>*Erweiterung:*<br>**1a** Kundendaten aktualisieren<br>*Alternativen*:<br>**1a** Neukunden erfassen<br>**3a** Rechnung mit Nachnahme erstellen<br>**3b** Rechnung mit Bankeinzug erstellen |

Im **Geschäftsprozessdiagramm** wird das Zusammenspiel mehrerer Geschäftsprozesse untereinander und mit den Akteuren beschrieben. Es gibt einen guten Überblick über das System und seine Schnittstellen zur Umgebung. Die Geschäftsprozesse werden als Ovale, die Akteure als Strichmännchen eingetragen. Eine Kommunikation zwischen Akteur und Geschäftsprozess wird als Linie dargestellt.



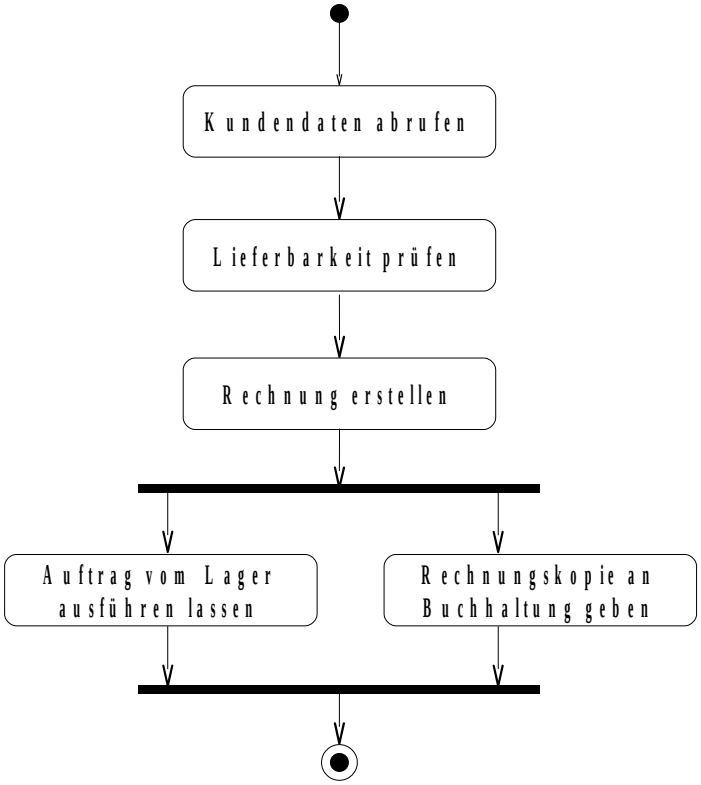

Beispiel 13.3: Für das oben beschriebene Versandhaus sieht das Geschäftsprozessdiagramm, wenn

man es um den Geschäftsprozess *Lager verwalten* ergänzt, folgendermaßen aus:




Die oben verwendete Checkliste ist ein gutes Hilfsmittel zur Beschreibung von Geschäftsprozessen. Es kann jedoch nicht ausgedrückt werden, dass für bestimmte Schritte die Reihenfolge aus fachlicher Sicht keine Rolle spielt.
Diese Möglichkeit bietet in der UML das **Aktivitätsdiagramm**. Es ist ein Sonderfall des Zustandsdiagramms, auf dessen Notation wir aber nicht näher eingehen. Die Zustände werden in abgerundeten Rechtecken dargestellt, die Zustandsübergänge durch Pfeile.

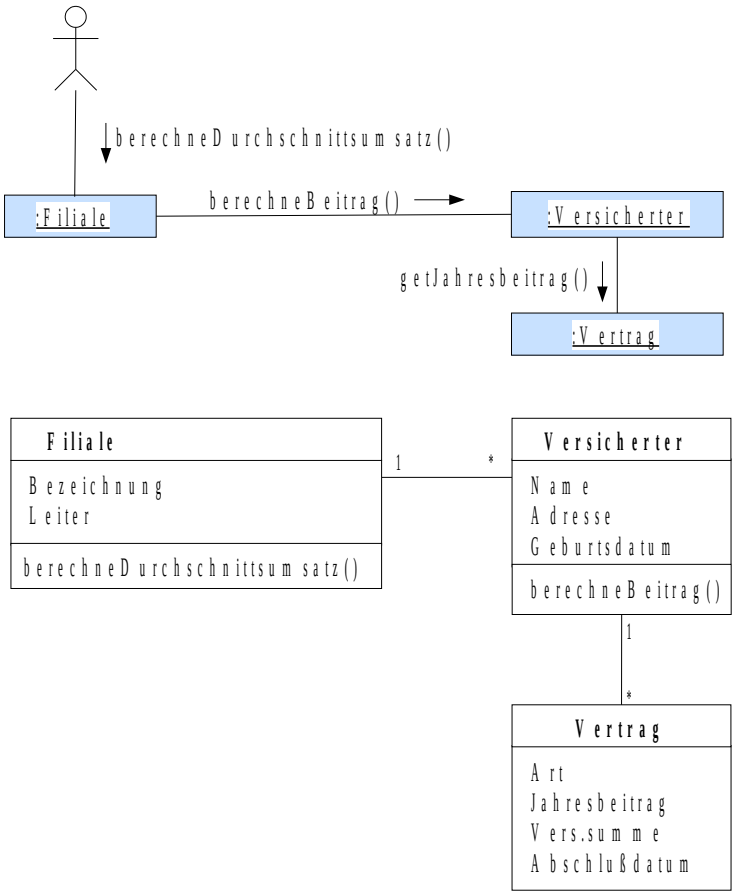
Beispiel 13.4: Das folgende Aktivitätsdiagramm modelliert die Standardfallbeschreibung des oben beschriebenen Geschäftsprozesses *Auftrag ausführen.*




In einem Aktivitätsdiagramm sind alle Zustände mit einer Verarbeitung verknüpft. Jeder Zustand modelliert einen Schritt innerhalb der Gesamtverarbeitung. Ein Zustand wird verlassen, wenn die mit ihm verbundene Verarbeitung beendet ist. Durch den Balken wird angezeigt, dass die Reihenfolge aus fachlicher Sicht unbedeutend ist, die Verarbeitungsschritte können somit auch parallel ausgeführt werden. Der zweite Balken steht für die „Synchronisation“.
Das Aktivitätsdiagramm kann, wie hier, zur Beschreibung von Geschäftsprozessen eingesetzt werden. Es kann aber auch zur Spezifikation komplexer Operationen verwendet werden.

## 13.2 Botschaft

Eine **Botschaft** ist die Aufforderung eines Senders an einen Empfänger, eine Dienstleistung zu erbringen. Der Empfänger interpretiert diese Botschaft und führt eine Operation aus.
Eine Botschaft löst eine Operation gleichen Namens aus. Das Verhalten eines objektorientierten Systems wird somit durch die Botschaften beschrieben, mit denen Objekte untereinander kommunizieren. Botschaften können in der UML in verschiedenen Diagrammen dargestellt werden. Wir betrachten hier als Beispiel ein **Kommunikationsdiagramm** und schauen, wie die Botschaften ins Klassendiagramm übertragen werden.
Beispiel 13.5: Für jede Filiale einer Versicherung soll der Durchschnittsumsatz berechnet werden. Um für jeden Versicherten die Beitragssumme zu ermitteln, muss der Jahresbeitrag eines jeden Vertrags dieses Versicherten bekannt sein. Wenn die Filiale die Botschaft *berechneDurchschnittsumsatz()* erhält, dann sendet sie jedem ihrer Versicherten die Botschaft *berechneBeitrag(),* die wiederum die Botschaft *getJahresbeitrag()* an alle ihre Vertragsobjekte schickt. Im Klassendiagramm führt dies dann zu Beziehungen, bei denen jede Filiale ihre Versicherten und jedes Versicherten-Objekt seine Vertrags-Objekte kennt.




Für Botschaft ist auch der englische Begriff „message“ üblich. Teilweise wird auch von „Operationsaufruf“ oder „Methodenaufruf“ gesprochen.

## 13.3 Weitere Diagramme

Die UML bietet noch weitere Diagramme, auf die wir hier nicht näher eingehen wollen. Um eine Sequenz von Verarbeitungsschritten darzustellen gibt es das Sequenzdiagramm und das Kollaborationsdiagramm. Ein Zustandsdiagramm stellt Zustände und Zustandsübergänge dar.

***Fragen:***

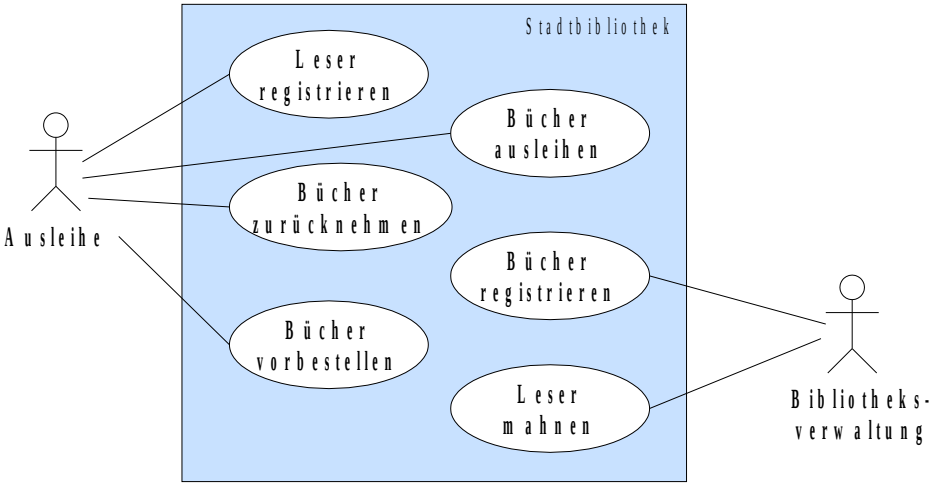
- ✗ Wie wird ein Geschäftsprozeß dokumentiert?
- ✗ Was versteht man unter einem Akteur?
- ✗ Was kann mit einem Aktivitätsdiagramm modelliert werden?
- ✗ Was versteht man unter einer Botschaft?

***Beispielaufgabe:***

Ziel: Geschäftsprozesse identifizieren und ein Geschäftsprozeßdiagramm erstellen.

Für eine Stadtbibliothek soll ein Softwaresystem entwickelt werden. Analysieren Sie die typischen Geschäftsprozesse zur Ausleihe und Verwaltung von Büchern und erstellen Sie ein Geschäftsprozeßdiagramm.

Lösungsbeispiel:



# *14 Fehlerbehandlung*

***Lernziele:***

- **Wissen**
  - Wissen, was eine Exception ist
  - Wissen, woraus ein Exception-Handler besteht
- **Verstehen**
  - Erklären können, wie die Ausnahmebehandlung funktioniert
- **Anwenden**
  - Exceptions in eigenen Programmen werfen und fangen können

## 14.1 Ausnahmen

Auch nach gründlichem Testen der eigenen Programme, können bei der Programmausführung Fehler auftreten. Im folgenden sind einige Beispiele von Fehlern genannt, nicht alle können durch sorgfältige Programmierung vermieden werden:

- ➔ Eingabefehler: Tippfehler, Datei mit falschen Informationen
- ➔ Gerätefehler: z.B. Drucker ausgeschaltet, Webseite nicht verfügbar
- ➔ physikalische Grenzen: auf dem Datenträger ist nicht mehr genügend freier Platz vorhanden, der Arbeitsspeicher ist ausgereizt
- ➔ Codefehler: Methode liefert falsche Ergebnisse, ungültiger Array-Index, Verwendung einer Objektreferenz ohne zugehörigen Speicherbereich

Unser Ziel ist es, dass beim Auftreten eines solchen Fehlers das Programm nicht abstürzt, sondern auf eine sanfte Art und Weise beendet werden kann, dass Daten gespeichert werden und evtl. der Benutzer benachrichtigt wird.
Dafür bietet Java die Möglichkeit der **Ausnahmebehandlung** (engl.: Exception Handling). Eine **Exception** ist ein Objekt, das Informationen über einen Programmfehler enthält. Möchte man signalisieren, dass ein Fehler aufgetreten ist, so löst man eine Exception aus.
Das Auslösen einer Exception besteht aus

- dem Erzeugen eines Exception-Objekts und
- dem Werfen dieses Objekts mit dem Schlüsselwort *throw.*

Beispiel 14.1:

```
    CStudent student1;
    if (student1 == null)
         throw new NullPointerException("student1 ist
null");
```

## 14.2 Die Ausnahmehierarchie

Eine Exception ist immer ein Objekt einer Klasse aus einer speziellen Vererbungshierarchie. Wir können entweder ein Objekt einer vorhandenen Exception-Klasse bilden oder einen eigenen Exception-Typ definieren, indem wir eine Subklasse einer dieser Klassen bilden. Jede Exception-Klasse ist von der Klasse *Throwable* abgeleitet.

Die folgenden Exception-Klassen sind von der Klasse *RunTimeException* abgeleitet, Objekte dieser Klassen werden auch als Standardausnahmen bezeichnet.

- ➔ **ArrayIndexOutOfBoundsException:** Zugriff auf nicht vorhandenes Array-Element
- ➔ **ArithmeticException:** ungültige Rechenoperation, z.B. Ganzzahl-Division durch 0, Wurzel aus negativer Zahl
- ➔ **ClassCastException:** Versuch ein Objekt in ein unzulässigen Klassentyp umzuwandeln.
- ➔ **NullPointerException:** Versuch auf eine Objektreferenz zuzugreifen, der kein Speicherplatz zugewiesen wurde.

## 14.3 Die Behandlung von Exceptions

Es werden ungeprüfte und geprüfte Exceptions unterschieden.
**Ungeprüfte Exceptions** sind Objekte von Subklassen der Klasse *RunTimeException.* Für das Auslösen einer ungeprüften Exception wird lediglich eine *throw*-Anweisung verwendet.
**Geprüfte Exceptions** müssen behandelt werden:

1. Jede Methode, die eine Exception werfen kann, muss eine *throws-Klausel* im Kopf der Methode deklarieren, z.B.
   *public void speichern(String dateiname) throws IOException*
   Es wird empfohlen auch einen entsprechenden *javadoc*-Kommentar direkt vor der Methode einzufügen.
2. Exceptions müssen gefangen werden: Der Aufrufer einer Methode, die eine geprüfte Exception werfen kann, muss den Methodenaufruf durch einen **Exception-Handler** schützen.

Ein Exception-Handler besteht aus einem **try-Block** zum schützen von Anweisungen und einem **catch-Block** zum Fangen der Exceptions. Der catch-Block wird nur dann ausgeführt, wenn eine Exception geworfen wird. Die allgemeine Form ist

```
try {
        eine oder mehrere geschützte Anweisungen
}
catch (ExceptionTyp e) {
        die Exception melden oder weiterleiten
}
```

Das Werfen einer Exception unterbricht immer den normalen Kontrollfluß. Die Programmausführung wird direkt in dem passenden catch-Block fortgeführt. Wird kein passender catch-Block gefunden kommt es zum Programmabsturz.

Es können mehrere Anweisungen, in denen Exceptions auftreten können in einem try-Block zusammengefaßt werden.

*Tips zur Verwendung von Ausnahmen:*

- ➔ Eine Methode, in der eine Exception aufgetreten ist, muss diese Exception entweder abfangen und behandeln (eigener try- und catch-Block) oder die Exception weiterleiten. (dann: throws-Abschnitt in Methodendeklaration)
- ➔ Die Ausnahmebehandlung sollte nicht als Ersatz für die Fehlersuche und das Testen des Programms verwendet werden.
- ➔ Fassen Sie mehrere Anweisungen in einem try-Block zusammen, nicht für jede Anweisung einen eigenen try-Block verwenden.
- ➔ Leiten Sie Ausnahmen weiter, wenn sie außerhalb Ihrer Methode besser behandelt werden können.

***Fragen:***

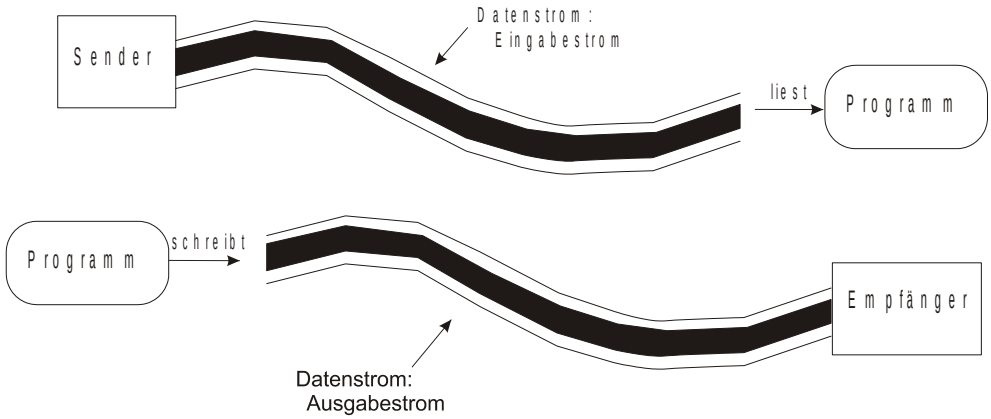
- ✗ Was ist eine Exception?
- ✗ Woraus besteht das Auslösen einer Exception?
- ✗ Welche Exceptions müssen behandelt werden?
- ✗ Woraus besteht ein Exception-Handler?

# *15 Ein- und Ausgabe*

***Lernziele:***

- **Wissen**
  - Die Standardeingabe und die Standardausgabe kennen
  - Eine Klasse kennen, die für die Ausgabe auf dem Bildschirm und in eine Datei verwendet werden kann
  - Wissen, mit welchen Klassen die Eingabe von Tastatur realisiert werden kann
  - Eine Klasse kennen, mit der aus einer Datei gelesen werden kann.
- **Verstehen**
  - Erklären können, wie die Dateneingabe und -ausgabe in Java funktioniert
  - Verstehen, wie der Ablauf des Einlesens von Tastatur mit Hilfe der Klasse Scanner ist
  - Methoden einer Klasse in der API-Dokumentation finden können
- **Anwenden**
  - Eigene Programme mit Ein- und Ausgabe schreiben können

Das Grundkonzept in Java für die Ein- und Ausgabe ist der Datenstrom.
Ein **Datenstrom** beschreibt den Datenfluß zwischen einem Sender und einem Empfänger. Je nach Art des Senders bzw. Empfängers gibt es verschiedene Arten von Datenströmen, wobei man auch die Richtung der Datenströme beachten muss. Ein Datenstrom ist als Klasse implementiert. Die Klassen zur Unterstützung von plattformunabhängigen Eingabe- und Ausgabe-Operationen findet man im Paket *java.io.*




Bei der Kommunikation über die Tastatur werden Zeichen eingegeben. Die Tastatur ist somit der Sender, es handelt sich um einen Eingabestrom. Auf dem Bildschirm werden Zeichen ausgegeben, der Bildschirm ist ein Empfänger, es handelt sich um einen Ausgabestrom. Die Tastatur wird auch als **Standardeingabe** bezeichnet, der Bildschirm als **Standardausgabe**.
Man unterscheidet zwei Haupkategorien von Klassen:

- Klassen für den Umgang mit Text, sie werden *Reader* und *Writer* genannt.
- Klassen für den Umgang mit Binärdateien, sie werden *Streams* genannt.

## 15.1 Die Ausgabe auf dem Bildschirm

Mit *System.out* haben wir schon häufig gearbeitet. *System* ist eine konstante Klasse, die nur Klassenattribute und Klassenoperationen besitzt und von der kein Objekt gebildet werden kann. Das Klassenattribut *out* der Klasse System ist vom Typ *PrintStream* (abgeleitet von der Klasse *OutPutStream*) und bildet die Standardausgabe.
Mit *System.out.printf* ist eine formatierte Ausgabe auf dem Bildschirm möglich. Die Methode erwartet als erstes Argument einen **Formatstring**, der Platzhalter für die nachfolgend übergebenen Argumente enthalten kann. Ein Platzhalter besteht aus einem %-Zeichen und einem Kennbuchstaben, der die Art der Formatierung angibt. Einige Beispiele für Formatierungen sind:

| | |
|---|---|
| %d | Ausgabe als Ganzzahl zur Basis 10 |
| %x | Ausgabe als Ganzzahl zur Basis 16 (Hexadezimal) |
| %f | Ausgabe als Fließkommazahl |
| %s | Ausgabe als String |

Dabei bedeutet %8.2f die Ausgabe einer Fließkommazahl mit 2 Nachkommastellen und insgesamt mindestens 8 Zeichen.
Der Quelltext in Beispiel 15.1:

```
String ware1  = "Heft";
double preis1 = 1.75;
String ware2  = "Fachbuch";
double preis2 = 12.80;
System.out.printf("1 %8s kostet %5.2f Euro \n", ware1, preis1);
System.out.printf("1 %8s kostet %5.2f Euro \n", ware2, preis2);
```

erzeugt die Ausgabe

```
1     Heft kostet  1.75 Euro
1 Fachbuch kostet 12.80 Euro
```

Eine zweite Möglichkeit für die Ausgabe auf dem Bildschirm ist die Verwendung der Klasse **PrintWriter.** Dazu erzeugt man ein Objekt der Klasse *PrintWriter* und übergibt dem Konstruktor den Ausgabestrom, d.h. *System.out.* Die wichtigsten Methoden der Klasse *PrintWriter* sind *println()* und *print().* Wenn die Ausgabe abgeschlossen ist, muss das Ausgabeobjekt mit der Methode *close()* geschlossen werden.
Beispiel 15.2

```
PrintWriter ausgabe = new PrintWriter(System.out);
String ware1  = "Heft";
double preis1 = 1.75;
String ware2  = "Fachbuch";
double preis2 = 12.80;
ausgabe.println("1 " + ware1 + " kostet " + preis1 + " Euro");
ausgabe.println("1 " + ware2 + " kostet " + preis2 + " Euro");
ausgabe.close();
```

Was ist nun der Vorteil gegenüber der Verwendung von *System.out.println()* ?

## 15.2 Ausgabe in Dateien

Mit der Klasse *PrintWriter* kann auch in eine Datei geschrieben werden, indem man dem Konstruktor bei der Objekterzeugung den Dateinamen übergibt:

```
PrintWriter ausgabe = new PrintWriter("Test.txt ");
```

Die Datei „Test.txt“ wird im aktuellen Verzeichnis neu angelegt oder, falls sie schon vorhanden ist, überschrieben. Kann sie weder gefunden noch angelegt werden, wird eine *FileNotFoundException* ausgelöst. Daher muss die Ausgabe in einen try-Block gesetzt werden.
Das Beispiel 15.2 lässt sich leicht so umschreiben, dass in die Datei „Test.txt“ anstatt auf den Bildschirm geschrieben wird!

Beispiel 15.3

```
try
  {
     PrintWriter ausgabe = new PrintWriter("Test.txt ");
     String ware1  = "Heft";
     double preis1 = 1.75;
     String ware2  = "Fachbuch";
     double preis2 = 12.80;
     ausgabe.println("1 " + ware1 + " kostet " + preis1 + " Euro");
     ausgabe.println("1 " + ware2 + " kostet " + preis2 + " Euro");
     ausgabe.close();
  }
catch (IOException e)
  {
     System.out.println("Fehler beim Speichern in die Datei Test.txt: " + e.getMessage());
  }
```

Dadurch, dass der eigentliche Ausgabecode vom Ziel der Ausgabe unabhängig ist, kann das gewünschte Ausgabeziel auch per *if*-Abfrage festgelegt werden. Der restliche Quellcode ist dann identisch.

## 15.3 Eingabe von Tastatur

Mit Hilfe der Klassen **BufferedReader** und **InputStreamReader** kann zeilenweise von der Tastatur eingelesen werden:

Beispiel 15.4

```
BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
try {
        System.out.println ("Name: ");
        String name = stdin.readLine()
}
catch (IOException e) {
        System.out.println("IOException");
}
```

Mit der Klasse *BufferedReader* können die Zeichen nur als String eingelesen werden. Möchte man Zahlen einlesen müssen diese anschließend mithilfe der statischen methoden der Klassen *Integer* oder Double in Zahlen umgewandelt werden.
Im Paket *java.util* gibt es eine weitere Klasse, die sich zum Einlesen von der Tastatur eignet, die Klasse **Scanner.** Der Vorteil dieser Klasse ist, dass Zahlen direkt eingelesen werden können. Man erzeugt ein Scanner-Objekt, indem man dem Konstruktor die Standardeingabe *System.in* übergibt und kann dann mit den *next*-Methoden der Klasse Scanner die Eingabe lesen. Der Ablauf des Einlesens ist dabei der folgende:

- ➔ Der Benutzer gibt Zeichen ein.
- ➔ Die Eingabe wird mit der Return-Taste abgeschlossen.
- ➔ Die Java Virtual Machine erzeugt einen String mit allen Zeichen und übergibt diesen String dem Scanner-Objekt.
- ➔ Das Scanner-Objekt zerlegt den String in **Tokens.** Ein Token ist eine Zeichenfolge, als Trennzeichen dient das Leerzeichen.
- ➔ Die next-Methode von Scanner liest einen Token.

Nachfolgend einige Methoden der Klasse Scanner:

| | |
|---|---|
| *String next()* | Liefert das nächste Token als String |
| *int nextInt(), float nextFloat() double nextDouble()* | Liefert das nächste Token als elementaren Datentyp |
| *String nextLine()* | Liefert die nächste Eingabe als String |
| *boolean hasNext()* | Liefert true, wenn es weitere, noch nicht extrahierte Tokens in der Eingabezeile gibt |

Die meisten *next*-Methoden lesen nur einen Token, nur die *nextLine()*-Methode liest alle Tokens bis zum Ende der Eingabe.
Ein Beispiel finden Sie in Felix unter *Vorlesung/Beispiele/Tastatureingabe:*
*Beispiel 15.5*

```
Scanner tastatur = new Scanner(System.in);
System.out.println("\nGeben Sie Ihren Nachnamen ein!");
String name = tastatur.nextLine();
System.out.println("\nGeben Sie Ihr Alter ein!");
int alter = tastatur.nextInt();
System.out.println("\n" + alter + " Jahre?");
System.out.println("Haben Sie da auch nicht geflunkert,"
            + " Herr oder Frau " + name + "?");
```

## 15.4 Einlesen aus Dateien

Mit der Klasse Scanner kann auch aus Dateien gelesen werden, indem der Eingabestrom mit der zu lesenden Datei verknüpft wird. Dazu verwenden wir die Klasse **File**:

```
File datei = new File("Test.txt");
Scanner eingabe = new Scanner(datei);
```

Außerdem benötigen wir wieder einen try- und catch-Block.
Der gesamte Quellcode lautet dann:
Beispiel 15.6

```
public class DateiEinlesen
{
   public static void main(String[] args)
   {
      File datei = new File("Test.txt");
      try
      {
         Scanner eingabe = new Scanner(datei);
         String name = eingabe.nextLine();
         int alter = eingabe.nextInt();
         System.out.println( name + " ist " + alter + " Jahre alt.");
      }
```

```
      catch (IOException e)
      {
         System.out.println("Fehler beim Einlesen aus der Datei Test.txt.");
      }
   }
}
```

Aufgabe: Suchen Sie in der API-Dokumentation einige Methoden der Klasse *File.*

Eine weitere Möglichkeit ist die Verwendung der Klasse **FileReader**. Dem Konstruktor wird der Dateiname übergeben und mit der Methode *read()* werden einzelne Zeichen als Ganzzahl gelesen. Wenn das Ende der Datei erreicht ist, liefert die Methode *read()* das Ergebnis -1.
Ein Beispiel mit den Klassen *FileReader* und *StringBuilder* finden Sie im folgenden.
Beispiel 15.7

```
public class CDateiLesen
{
   public static void main(String[] args) throws IOException {
      FileReader eingabestrom = new FileReader("TestReader.txt");
      StringBuilder text = new StringBuilder(10);
      int gelesen;
      boolean ende = false;

      // lese Zeichen, bis Dateiende erreicht ist
      while(!ende) {
         gelesen = eingabestrom.read();

         if(gelesen == -1)
            ende = true;
         else
            text.append( (char) gelesen);
      }
      System.out.println(text);
   }
}
```

***Fragen:***

- ✗ Was ist die Standardeingabe, was die Standardausgabe?
- ✗ Welche Klasse kann sowohl für die Ausgabe auf dem Bildschirm, als auch für die Ausgabe in eine Datei verwendet werden?
- ✗ Welche Klasse kann sowohl für das Einlesen von Tastatur als auch für das Einlesen in eine Datei verwendet werden?

# Literaturverzeichnis

[1]: Balzert, Heide, Lehrbuch der Objektmodellierung Analyse und Entwurf mit der UML 2, 2005
[2]: Schiedermeier, Reinhard, Programmieren mit Java, 2010
[3]: Balzert, Heide, Lehrbuch der Objektmodellierung Analyse und Entwurf mit der UML 2, 2005
[4]: Barnes, David J. und Kölling, Michael, Java lernen mit BlueJ, 2009
[5]: , BlueJ Tutorial,
[6]: Glenford J. Myers, The art of software testing, 2012
[7]: Barnes, David J. und Kölling, Michael, Java lernen mit BlueJ, 2009
[8]: Barnes, David J. und Kölling, Michael, Java lernen mit BlueJ, 2009

# Stichwortverzeichnis