Einführung in die Programmierung

Dr. Peer Kröger, Dr. Arthur Zimek,
Andreas Züfle

Ludwig-Maximilians-Universität München,
Institut für Informatik,
LFE Datenbanksysteme

Wintersemester 2008/2009

Allgemeine Informationen zum Studium

Studiengangskoordinator:

Dr. Reinhold Letz

Büro: F12 (Oe)
Kontaktinformationen: `http://www.tcs.ifi.lmu.de/~letz/`

Informationen zum Studium:
`http://www.tcs.ifi.lmu.de/~letz/informationen.shtml`

Team für Vorlesung und Übungen

Dr. Peer Kröger

Sprechzeiten:
Mittwoch, 10-11 Uhr
Büro: E1.08 (Oe)

Dr. Arthur Zimek

Sprechzeiten:
Donnerstag, 14-15 Uhr
Büro: E1.06 (Oe)

Andreas Züfle

Sprechzeiten:
nach Vereinbarung
Büro: E1.04 (Oe)

Tutoren

- Thomas Mair
- Florian Nücke
- Lisa Reichert
- Bernhard Slawik
- Alice Thudt
- Eduard Vodicka
- Fabian Winter

Vorlesung

- Zeit und Ort:
  - Dienstags, 14.00 – 16.00 Uhr, Raum C 122 (Theresienstr. 41)
  - Freitags, 8.00 – 10.00 Uhr, Raum C 122 (Theresienstr. 41)
- Webseite zur Vorlesung:
  `http://www.dbs.ifi.lmu.de/Lehre/EIP/WS_2008/`

Inhalt der Vorlesung

- Thema: Imperative und objektorientierte Programmierung, grundlegende Datenstrukturen und Algorithmen
- Die theoretischen Konzepte werden mittels der Programmiersprache Java veranschaulicht und eingeübt.
- Es sind keine Vorkenntnisse dieser Programmiersprache nötig.

Anmeldung

- Sie benötigen eine Kennung am CIP-Pool der Informatik.
- Über die Vergabe dieser Kennungen informieren Sie sich bitte auf der Website der Rechnerbetriebsgruppe (RBG):
  `http://www.rz.ifi.lmu.de`

Kennungsvergabe am CIP-Pool

- Anmeldung und Ausgabe der Rechnerkennung erfolgt im Raum 001/Antarktis (weisser Flachbau im Norden des Geländes), Oettingenstr. 67
- Termine nach Nachname:

Mittwoch:	A - H	(15. Oktober)
Donnerstag:	I - S	(16. Oktober)
Montag:	S - Z	(20. Oktober)
Dienstag:	A - Z	(21. Oktober)

Anmeldung

- Mit der erhaltenen CIP-Kennung können Sie sich im UniWorX-System anmelden:
  `http://www.pst.ifi.lmu.de/uniworx/`
- In UniWorX können Sie sich zu dieser Vorlesung (EIP) anmelden.
- Ihre Anmeldung zu einer Übung soll ebenfalls dort erfolgen.
- Über UniWorX können Sie Ihre Lösungen zu den Übungsaufgaben abgeben und erhalten die Korrekturen zurück.
- Später wird über UniWorX auch die Anmeldung zu den Klausuren erfolgen.

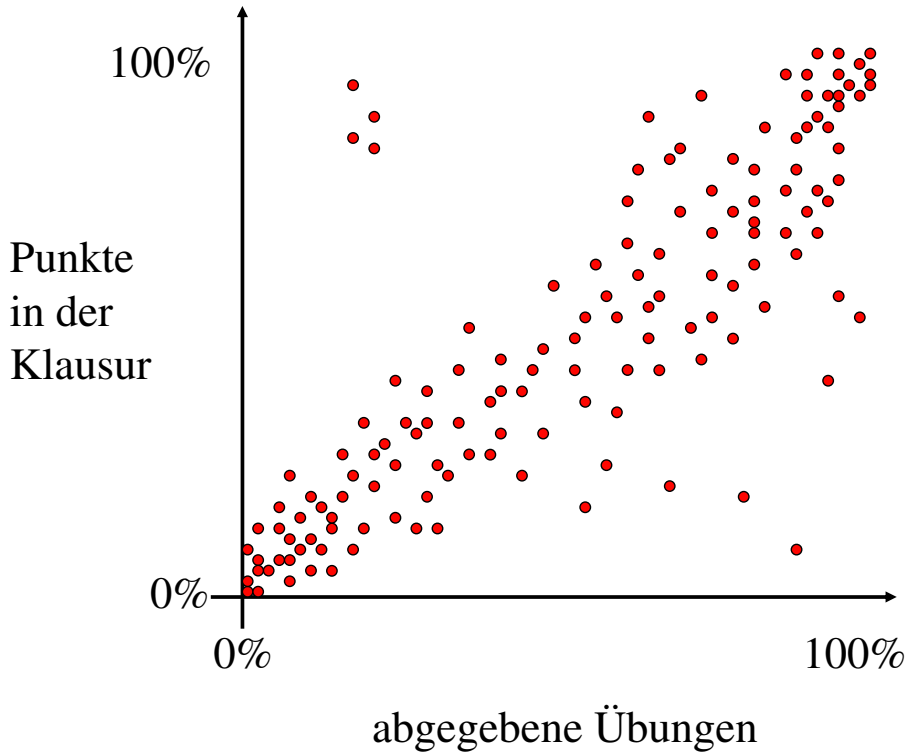
Hilfsmittel

- Die Programmiersprache Java ist eine wichtige Grundlage, um der Vorlesung zu folgen. Wir verwenden die Java Standard Edition 6.0, die auch am CIP-Pool installiert ist. Falls Sie zuhause arbeiten möchten, benötigen Sie das Java Development Kit (JDK 6), das Sie unter `http://java.sun.com/javase/downloads/index.jsp` frei herunterladen können.
- Wichtigstes Hilfsmittel für Java ist die allgemeine Dokumentation (API) unter `http://java.sun.com/javase/6/docs/api/`.
- Eine Sammlung von Tutorials von sun finden Sie unter `http://java.sun.com/docs/books/tutorial/`.
- Studenten helfen Studenten: `http://www.die-informatiker.net/forum/Einfprog_WS0809`
- Weitere Hilfsmittel werden auf der Vorlesungswebseite angegeben.

Übungen/Übungsaufgaben

- Es erscheint jede Woche freitags (ab 17.10.08) ein Übungsblatt, auf dem die in der Vorlesung besprochenen Konzepte vertieft werden.
- Sie haben die Möglichkeit, das jeweilige Übungsblatt bis zum darauffolgenden Freitag zu bearbeiten und bei uns abzugeben. Ihre Lösung wird dann von uns korrigiert. Dies ist eine reine Serviceleistung von uns für Sie. Insbesondere beeinflusst Ihre Leistung beim Bearbeiten der Blätter **nicht** Ihre Note in der Klausur!

Übungen/Übungsaufgaben



Folgerung:

Wir empfehlen **dringend** die regelmäßige Bearbeitung der Übungsblätter und Teilnahme an einer Übung!

Übungen/Übungsaufgaben

- Jedes Übungsblatt wird in der Woche nach der Abgabe in den Übungen besprochen (Beginn: 27.10.08).
- Die Übungstermine entnehmen Sie bitte der Webseite zur Vorlesung.
- Sie können sich zu Semesterbeginn aussuchen, welchen Übungstermin Sie wahrnehmen wollen – zu diesem Termin können Sie sich dann in UniWorX anmelden.

Erstes Übungsblatt

- Zusätzlich ist diese Woche bereits ein erstes Übungsblatt erschienen, das Ihnen einen Einstieg in die technischen Voraussetzungen für die Erstellung von Java Programmen bietet.
- Dieses Übungsblatt wird nicht korrigiert.
- Dieses Übungsblatt wird in der nächsten Woche (20.10.08 – 24.10.08) in den Übungen besprochen.
  Achtung: Die Übungen finden in dieser Woche ausnahmsweise im Raum Z08 (Keller), Oettingenstr. 67 (CIP-Raum) am Rechner statt.
- Dieses Übungsblatt wendet sich insbesondere an Anfänger ohne Computer-/Programmier-Kenntnisse. Wir bitten Sie, dieses Übungsblatt zunächst selbstständig zu bearbeiten. Sollten Sie Probleme dabei haben, empfehlen wir Ihnen den Besuch der Übung in der kommenden Woche.

Klausur

- Es werden zwei Klausuren angeboten:
  1. Zum Ende der Vorlesungszeit (vorläufig geplant für 31.01.09 vormittags): Diese Klausur ist die sog. “Testklausur”.
  2. Zum Ende des Wintersemesters (März/April): Diese Klausur ist die Hauptklausur.
- Genaueres wird noch bekanntgegeben.
- Die Klausuren beziehen sich auf den gesamten Stoff der Vorlesung und der Übungen. Sie sollen in der Lage sein, theoretische Fragen zu beantworten und praktische Aufgaben (z.B. durch ein Java-Programm) zu lösen.

Abschnitt 1: Einführung

Überblick

Was ist Informatik?

Franz. *informatique* (= information + mathématiques)
Engl. *computer science*, neuerdings auch *informatics*

- DUDEN Informatik:
  Wissenschaft von der systematischen Verarbeitung von Informationen, besonders der automatischen Verarbeitung mit Computern.
- Gesellschaft für Informatik (GI):
  Wissenschaft, Technik und Anwendung der maschinellen Verarbeitung und Übermittlung von Informationen.
- Association for Computer Machinery (ACM):
  Systematic study of algorithms and data structures.

Primäres Lernziel: nicht die Komponenten eines Computers oder die Sprachen zu seiner Programmierung sondern die *Prinzipien und Techniken zur Verarbeitung von Information.*

Teilgebiete der Informatik

- Theoretische Informatik:
  Theoretische Durchdringung und Grundlegung von Fragen und Konzepten der Informatik (z.B. “Berechenbarkeit” von Aufgabestellungen, “Komplexität” von Aufgabenstellungen und Lösungen).
- Technische Informatik:
  Beschäftigung mit der *Hardware* (z.B. maschinengerechte Darstellung von Daten und Algorithmen).
- Praktische Informatik:
  Konstruktion, Darstellung und Ausführung von Algorithmen (*Software*).

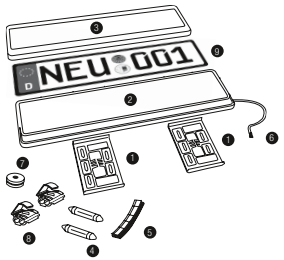
Algorithmusbegriff

- Zentraler Begriff in der Informatik.
- Abgeleitet von *Muhammad ibn Musa al-Chwarizmi*, einem bedeutenden persischen Mathematiker des 9. Jahrhunderts.
- Algorithmus = Handlungsvorschrift, um ausgehend von bestimmten Vorbedingungen ein bestimmtes Ziel zu erreichen.

Algorithmusbegriff: Beispiel

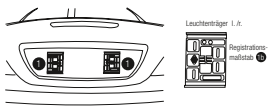
Montageanleitung

Einzelteile



❶ Leuchtenträger
❷ Flächenleuchte
❸ Abdeckrahmen
❹ Lastwiderstände
❺ Klebemasse zur Vibrationsdämpfung
❻ Kabel
❼ Kabeldurchführung
❽ Abzweigklemmen
❾ Nummernschild (nicht in Packung enthalten)

1

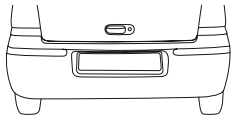


Altes Nummernschild abschrauben und die Leuchtenträger ❶ am Schraublochmuster der Karosserie befestigen. Dazu können Sie die bereits vorhandenen Schrauben benutzen. Bei Installation auf Kunststoffuntergrund empfiehlt sich die Nutzung von 4 Schrauben. Die Langlöcher ermöglichen eine flexible Anpassung, der Registrationsmaßstab ⓫ dient der vertikalen Ausrichtung. **Hinweis:** Halten Sie die genaue Position der Leuchtenträger zur Schraube fest, indem Sie die Markierungen abzählen, und stellen Sie auf beiden Seiten dasselbe Maß ein. Die Leuchtenträger sollten später möglichst nahe der rechten und linken Kante der Flächenleuchte greifen.

2a

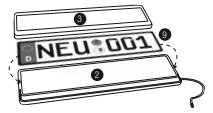
Montage bei Kennzeichen in der Heckklappe: Die Innenverkleidung der Heckklappe so weit entfernen, dass Sie von innen an den Kennzeichenbereich gelangen. Die Demontage ist von Fahrzeug zu Fahrzeug unterschiedlich. Fragen Sie im Zweifel einen Fachbetrieb. Bohren Sie jetzt mit einem Metallbohrer (Ø 7 mm) ein Loch für die Kabel unter der vorhandenen Kennzeichenbeleuchtung. Bohrloch entgraten und mit Lackstift gegen Rostbefall schützen. **Hinweis:** Wählen Sie für das Bohrloch eine Stelle hinter dem Kennzeichen, die von innen zugänglich ist und später von der Flächenleuchte verdeckt wird.

2b



Montage bei Kennzeichen im Stoßfänger: Bohren Sie mit einem Metallbohrer (Ø 7 mm) ein Loch für die Kabel unter der vorhandenen Kennzeichenbeleuchtung. Bohrloch entgraten und mit Lackstift gegen Rostbefall schützen. **Hinweis:** Wählen Sie für das Bohrloch eine Stelle hinter dem Kennzeichen, die von innen bzw. unten zugänglich ist und später von der Flächenleuchte verdeckt wird.

3

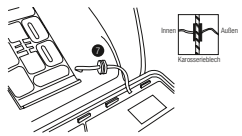


Platzieren Sie die umlaufende Randprägung des Nummernschildes ❸ und den Dichtungsring exakt übereinander. Abdeckrahmen ❸ durch **kräftiges Andrücken** auf der Flächenleuchte ❷ befestigen! Der Rahmen muss an 4 Stellen hörbar einrasten. Drehen Sie das Schild um und prüfen Sie, ob alle Rastnasen eingerastet sind. Der Abdeckrahmen ❸ kann zum Kennzeichenwechsel oder zur Reinigung gelöst werden. Hierzu biegen Sie mit einem Schraubendreher an den 4 dafür vorgesehenen Öffnungen den Kennzeichenrahmen nach außen.

4

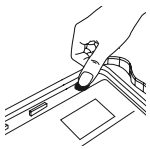
Nehmen Sie die Original-Kennzeichenbeleuchtung außer Betrieb (vorgeschrieben gemäß StVZO), indem Sie beide Leuchtmittel der Kennzeichenbeleuchtung durch die mitgelieferten Lastwiderstände ❹ ersetzen. Sollten bei Ihrem Fahrzeug andere Leuchtmittel eingesetzt werden, nehmen Sie die werkseitig angebrachte Nummernschildbeleuchtung durch Demontage der Leuchtmittel außer Betrieb.

5



Jetzt die mitgelieferte Kabeldurchführung ❼ mit Kabel montieren und in den Innenbereich führen. Prüfen Sie, ob die Aussparung der Kabeldurchführung fest im Karosserieblech sitzt.

6

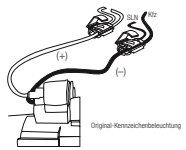


Nehmen Sie nun die Klebemasse zur Vibrationsdämpfung ❺ und bringen Sie diese auf der Rückseite des Kennzeichens an. Die genaue Menge bemessen Sie je nach Abstand zwischen Heckblech und Kennzeichen.

7

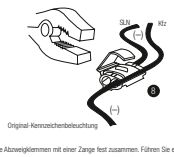
Richten Sie die Flächenleuchte ❷ auf den Leuchtenträgern horizontal aus. Jetzt kräftig aufdrücken (obere Kante zuerst) und **hörbar einrasten** lassen.

8



Verbinden Sie mit den mitgelieferten Abzweigklemmen ❽ die Kabel mit der Spannungsversorgung der ursprünglichen Kennzeichenbeleuchtung: 1 x mit „plus", 1 x mit „minus". Achtung: Nennspannung 12 V Gleichstrom, Nennleistung 2,9 Watt.

9



Pressen Sie beide Abzweigklemmen mit einer Zange fest zusammen. Führen Sie einen Funktionstest in abgedunkelter Umgebung durch. **Hinweis:** Aufgrund gesetzlicher Bestimmungen darf die Flächenleuchte bei Dämmerung und Dunkelheit nicht überstrahlen. Aus diesem Grund werden Sie unter Umständen bei Tageslichtbedingungen den Betrieb der Flächenleuchte nicht ohne weiteres erkennen. Anschließend die Innenverkleidung wieder montieren.

Algorithmusbegriff: Beispiel

Kochrezept für einen Afrikanischen Hühnertopf (von `www.chefkoch.de`)

- Zutaten:
  1 Huhn (küchenfertige Poularde), 125 ml Hühnerbrühe, 8 Tomaten, 150 g Erdnussbutter, 2 Zucchini, Salz und Pfeffer, Rosmarin.
- Zubereitung:
  - Von der Poularde die Haut abziehen, Huhn zerteilen.
  - Hühnerteile abspülen und in einen breiten Topf legen.
  - Die Brühe angießen und langsam zum Kochen bringen.
  - Die Tomaten mit kochendem Wasser übergießen, enthäuten und unzerteilt zum Fleisch geben.
  - Zugedeckt bei geringer Hitze ca. 40 Minuten köcheln.
  - Nach 30 Minuten Garzeit die Erdnussbutter (Erdnüsse) einrühren.
  - Mit Salz, Pfeffer und Rosmarin abschmecken.
  - Die Zucchini putzen, Blüten und Stengelansatz entfernen.
  - Zucchini waschen und in kleine Würfel schneiden.
  - 10 Minuten vor Garzeitende in den Topf geben und alles noch einmal abschmecken.

Definition: Algorithmus

Ein *Algorithmus* ist ein Verfahren zur Verarbeitung von Daten mit einer präzisen, endlichen Beschreibung unter Verwendung effektiver, elementarer Verarbeitungsschritte.

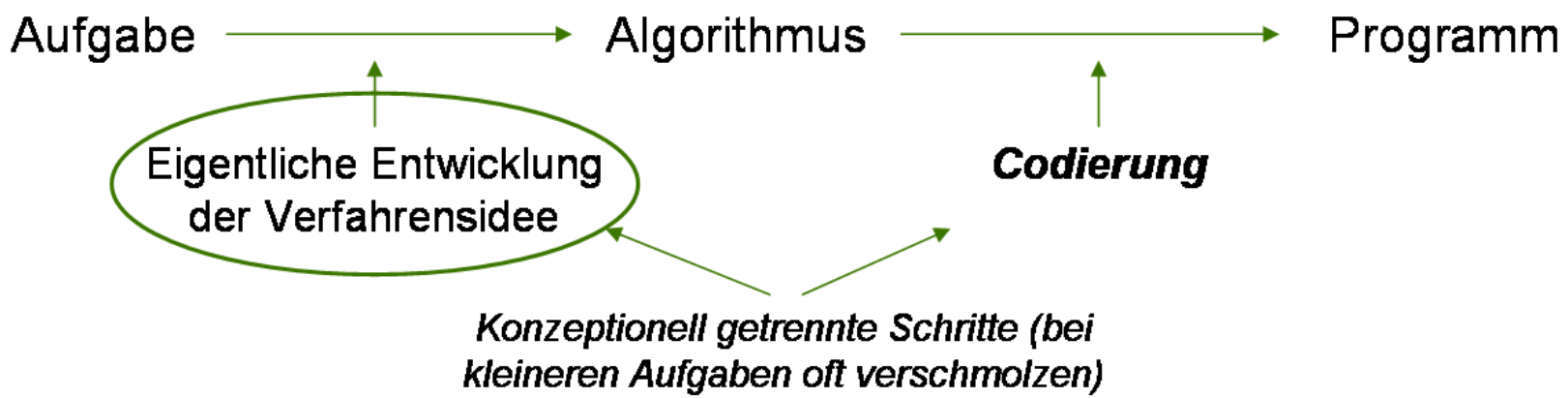
- Daten:
  Die Repräsentation und der Wertebereich sowohl der Eingabe als auch des Ergebnisses müssen eindeutig definiert sein.
- Präzise, endliche Beschreibung:
  In einem endlichen Text in einer eindeutigen Sprache genau festgelegte Abfolge von Schritten.
- Effektiver Verarbeitungsschritt:
  Jeder einzelne Schritt ist tatsächlich ausführbar, benötigt z.B. zu jedem Zeitpunkt der Ausführung nur endlich viel Speicherplatz.
- Elementarer Verarbeitungsschritt:
  Jeder Schritt ist entweder eine Basisoperation, d.h. muss von der zugrundeliegenden "Verarbeitungseinheit" (z.B. einem Computer) ausführbar sein, oder selbst durch einen Algorithmus spezifiziert.

Was macht eigentlich ein Informatiker?

Zentrale Aufgabe des Informatikers: Entwicklung von Algorithmen (und oft auch deren Realisierung auf dem Rechner als Programm)

Programm: Formale Darstellung eines Algorithmus (oder mehrerer) in einer Programmiersprache

Programmiersprache: Formale (eindeutige) Sprache, stellt insbesondere elementare Verarbeitungsschritte und eindeutig definierte Datentypen für die Ein-/Ausgabe zur Verfügung



Überblick

Aspekte von Java

- objektorientiert: Klassenkonzept, strenge Typisierung
- plattformunabhängig: Übersetzung in Virtuelle Maschine (JVM)
- netzwerkfähig, nebenläufig
- Sicherheitsaspekt in der Entwicklung der Sprache wichtig

Nachteile:

Laufzeithandicap durch Interpretation (JVM), wird aber stetig verbessert

Vorteile:

- Plattformunabhängigkeit
- verteilte Anwendungen, Web-Anwendungen
- Rechnerunabhängigkeit von Graphikanwendungen

Grober Aufbau eines Java-Programms

- Ein Java-Programm besteht aus Klassen und Schnittstellen
- Eine Klasse besteht aus
  - Klassenvariablen: beschreiben Eigenschaften aller Objekte dieser Klasse.
  - Attributen (fields, Instanzvariablen): beschreiben den Zustand eines Objekts.
  - statischen Methoden: Prozeduren einer Klasse, unabhängig vom Zustand eines Objekts
  - Objekt-Methoden: Operationen, die ein Objekt ausführen kann, abhängig vom Zustand des Objektes
  - Konstruktoren: Operationen zur Erzeugung von Objekten einer bestimmten Klasse

Wir betrachten nun zunächst nur die statischen Elemente. Zu den Aspekten der Objektorientierung kommen wir zu einem späteren Zeitpunkt der Vorlesung.

Ein einfaches imperatives Java-Programm

Keine Angst!!!
All das lernen wir später noch genauer kennen. Für den Anfang merken wir uns: ein einfaches imperatives Java-Programm besteht aus nur einer Klassendeklaration und einer Methode “main”:

```
public class KlassenName
{
  public static void main(String[] args)
  {
    // Hier geht’s los mit
    // Anweisungen (elementare Verarbeitungsschritte)
    // ...
  }
}
```

Die Textdatei, die den Java-Code enthält, heißt `KlassenName.java`, also genauso wie die enthaltene Klasse, mit der Endung `java`.

Beispiel: `HelloWorld.java`

```
public class HelloWorld
{
  public static void main(String[] args)
  {
    System.out.println("Hello, world!");
  }
}
```

Konventionen

In der Java-Programmierung gibt es einige Konventionen. Deren Einhaltung erleichtert das Lesen von Programmen. Beispiele solcher Konventionen sind:

- Klassennamen beginnen mit großen Buchstaben.
  - `HelloWorld`
- Methodennamen, Attributnamen und Variablennamen beginnen mit kleinen Buchstaben.
  - Methoden: `main`, `println`,
  - Klassenvariable: `out`,
  - Variable: `args` (weder Instanz- noch Klassenvariable, sondern Parametervariable)
- Zusammengesetzte Namen werden zusammengeschrieben, jeder innere Teilname beginnt mit einem großen Buchstaben.
  - Klasse `HelloWorld`

Konventionen

Warum Konventionen?

- In der “Lebenszeit” eines Programmes werden typischerweise 80% der Kosten von Wartung verursacht.
- Selten wird ein Software-Produkt dauerhaft vom ursprünglichen Autor gewartet.
- Code-Konventionen erleichtern die Lesbarkeit von Programmen und ermöglichen Programmierern, fremden (und eigenen!) Code schneller und besser zu verstehen.

Die von Sun empfohlenen Konventionen für Java finden Sie unter:
`http://java.sun.com/docs/codeconv/`
Momentan ist dort vieles vielleicht noch unverständlich, aber schlagen Sie im Lauf des Semesters immer mal wieder dort nach!

Compiler und Interpreter

- Ein Programm einer höheren Programmiersprache ist für Menschen lesbar, folgt aber auch einer festen Syntax (Grammatik), so dass *elementare Verarbeitungsschritte* klar definiert sind.
- Ein *Übersetzer* (*Compiler*) erzeugt aus der Programmdatei *Maschinencode*, der von Rechnern ausgeführt werden kann.
  - Der Compiler kennt die Syntax der Programmiersprache und bemerkt *syntaktische* Fehler (Verstösse gegen die Grammatik).
  - Nur, wenn das Programm syntaktisch korrekt ist, wird auch Maschinencode erzeugt. Andernfalls reagiert der Compiler mit Fehlermeldungen.
  - Achtung: Der Compiler erkennt nur *syntaktische*, nicht aber *semantische* Fehler, d.h. Fehler durch die das Programm eine andere *Bedeutung* annimmt als vom Programmierer beabsichtigt.
- Bei vielen Sprachen (z.B. C/C++) erzeugt der *Compiler* plattformabhängigen Maschinencode (kann nur auf bestimmten Rechnerarchitekturen/Betriebssystemen ausgeführt werden).

Compiler und Interpreter

- Sogenannte Skript-Sprachen (z.B. Perl, PHP, SML) werden nicht kompiliert, sondern von einem plattformspezifischen *Interpreter* interpretiert, wobei auch eine Syntaxprüfung durchgeführt werden muss.
- Vorteil: Die Programme sind plattformunabhängig.
- Nachteil: Der Sourcecode bleibt unübersetzt und wird bei jeder Ausführung des Programmes von neuem interpretiert.

Best of both worlds: Kompilierung und Plattformunabhängigkeit

- Plattformunabhängigkeit eines Java-Programmes wird durch einen Kompromiß erreicht:
  - Der Sourcecode wird durch einen Compiler übersetzt in Bytecode, der plattformunabhängig verwendet werden kann.
  - Der Bytecode ist also bereits syntaktisch überprüft, kann aber nicht direkt von einem Computer verarbeitet werden.
  - Bytecode wird von einer *virtuellen Maschine* ausgeführt (interpretiert).
  - Diese Interpretation muss aber keine syntaktische Prüfung mehr vornehmen.
  - Die virtuelle Maschine gibt es in verschiedenen Versionen für verschiedene Plattformen (JVM = Java Virtual Machine, Teil des JRE = Java Runtime Environment).

Übersetzung in Bytecode

- Aus einer Textdatei `KlassenName.java` erzeugt der Java Compiler `javac` eine Binärdatei `KlassenName.class`.
  Beispiel:

  ```
  javac HelloWorld.java
  ```

  erzeugt die Binärdatei `HelloWorld.class`.
- Die Binärdatei `KlassenName.class` enthält den Bytecode für die JVM.
- Der Compiler `javac` ist Teil des JDK (= Java Development Kit). Das JDK enthält JRE, Sie benötigen also das JDK für die Übungen zu dieser Vorlesung.

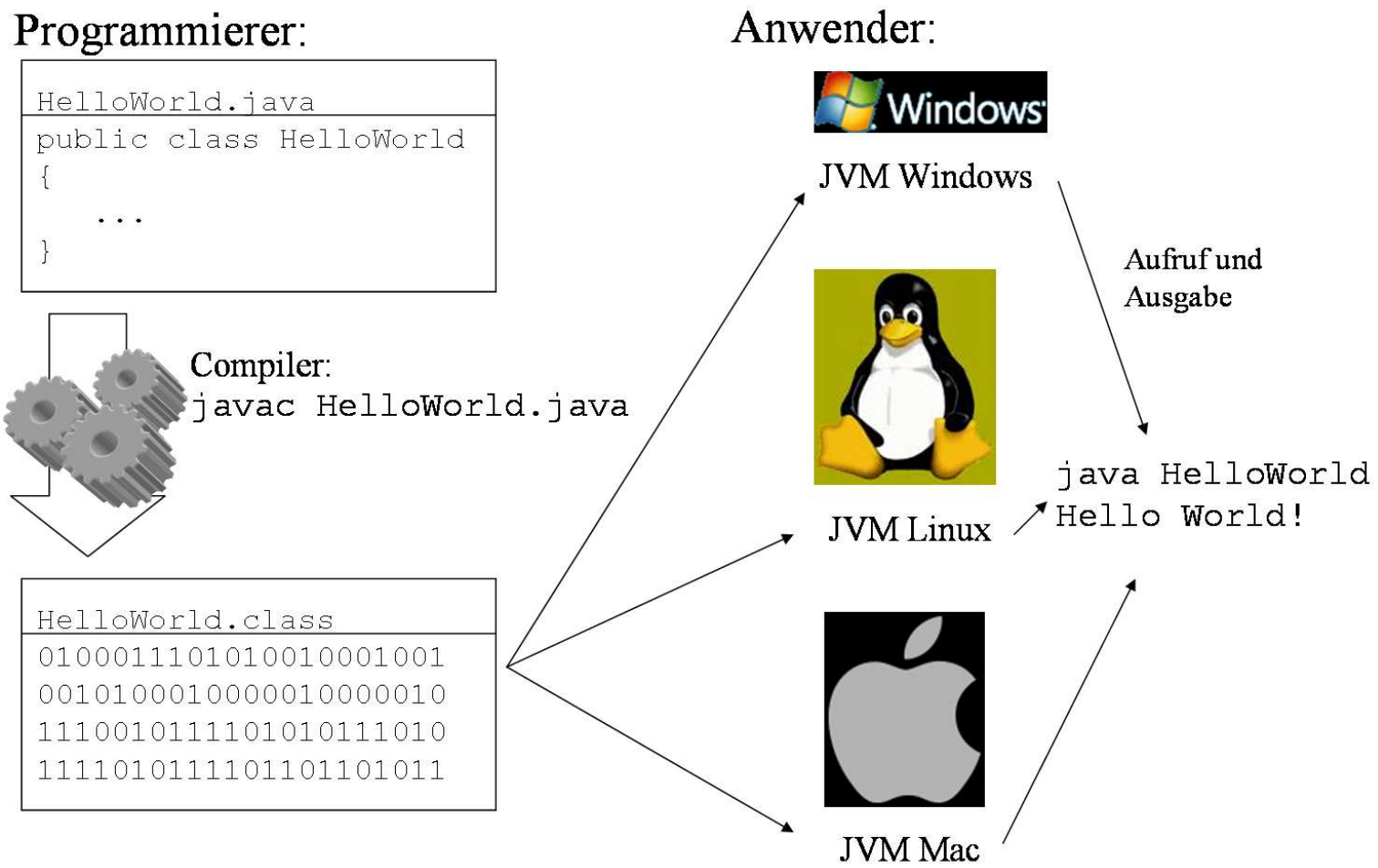
Ausführen des Programms

Die Binärdatei `KlassenName.class` wird der JVM übergeben und von dieser ausgeführt (interpretiert). Durch den Aufruf `java KlassenName` wird die `main`-Methode der Klasse `KlassenName` aufgerufen.
Beispiel:

```
java HelloWorld
```

gibt “Hello, World!” auf dem Bildschirm aus.

Übersetzung und Ausführung



Überblick

Prinzip: Programmcode und Kommentar gehören zusammen

"The view that documentation is something that is added to a program after it has been commissioned seems to be wrong in principle, and counterproductive in practice.

Instead, documentation must be regarded as an integral part of the process of design and coding."

C. A. R. Hoare (Turing-Preisträger):
Hints on Programming Language Design, 1973

C. A. R. Hoare, *1934
Erfinder von Quicksort, Hoare Logik, Strukt. Programmierung, CSP, Occam
Turing-Preis 1980

Verschiedene Arten von Kommentaren

Es gibt in Java 3 Arten von Kommentaren:

- Einzeilige Kommentare beginnen mit `//` und enden am Ende der aktuellen Zeile
- Mehrzeilige Kommentare beginnen mit `/*` und enden mit `*/`
- Dokumentationskommentare beginnen mit `/**` und enden mit `*/` und können sich ebenfalls über mehrere Zeilen erstrecken.

Kommentare derselben Art sind nicht schachtelbar. Ein Java-Compiler akzeptiert aber einen einzeiligen innerhalb eines mehrzeiligen Kommentars.

Dokumentationskommentare dienen dazu, Programme im Quelltext zu dokumentieren. Sie werden in den mit dem Befehl `javadoc` erzeugten Report mit aufgenommen.

Beispiel: Die Klasse `HelloWorld` dokumentiert

```
/**
 * HelloWorld Klasse um eine einfache Benutzung einer java-Klasse zu illustrieren.
 *
 * Diese Klasse dient nur dem Anzeigen des Strings
 * "Hello, world!" auf dem Bildschirm
 *
 * @author Arthur Zimek
 */
public class HelloWorld
{

    /**
     * Die main-Methode wird automatisch aufgerufen, wenn die Klasse mit
     * <code>java HelloWorld</code>
     * aufgerufen wird.
     *
     * Die Methode main druckt "Hello, world!" auf die Standard-Ausgabe.
     *
     * @param args Array mit Parametern - wird von dieser Methode nicht verwendet.
     */
    public static void main(String[] args)
    {
        // Ausgabe von "Hello World!" auf die Standard-Ausgabe
        System.out.println("Hello World!");
    }

}
```

Erzeugen der Dokumentation

Mit dem Befehl

```
javadoc HelloWorld.java
```

wird automatisch eine Beschreibung der Klasse `HelloWorld` erzeugt und in die Datei

```
HelloWorld.html
```

geschrieben.
Die Klassenbeschreibung wird eingebettet in eine organisierte Menge von html-Dokumenten:

`http://www.dbs.ifi.lmu.de/Lehre/EIP/WS_2008/skript/programmbeispiele/einfuehrung/helloWorld/index.html`

Diese Dokumentation kann auch für viele Klassen gleichzeitig erfolgen (`javadoc *.java`).

Spezielle Variablen (tags) für javadoc

Die durch @ eingeleiteten Elemente in einem Dokumentationskommentar haben eine besondere Bedeutung, z.B.:

- `@see` für Verweise
- `@author` für Name des Autors / Namen der Autoren
- `@version` für die Version
- `@param` für die Methodenparameter
- `@return` für die Beschreibung des Ergebnisses einer Methode

Java Standard Edition API Dokumentation

Auch die Bibliothek der Standard Edition ist mit javadoc erzeugt:
`http://java.sun.com/javase/6/docs/api/`
Für das fortgeschrittene Programmieren mit Java ist diese API Doc ein sehr wichtiges Hilfsmittel.

Abschnitt 2: Daten und Algorithmen

Überblick

Daten

Wir betrachten zunächst die Daten (Objekte), die durch Algorithmen verarbeitet werden sollen.
Typische Daten sind Zahlen, z.B. die Zahl “drei”, die wie folgt dargestellt werden kann:

- 3
- DREI
- III
- drei ausgestreckte Finger einer Hand
- ...

Datendarstellung

- Wir unterscheiden bei einem Objekt
  - die Darstellung, (*Syntax*, “Bezeichnung”),
  - seine Bedeutung, (*Semantik*, “Information”).
- Einige Datendarstellungen sind für maschinelle Verarbeitung nicht geeignet.
- Alle geeigneten Datendarstellungen beruhen auf dem Grundprinzip der *Zeichenreihe*.

Alphabet

- Ein *Alphabet* ist eine endliche Menge, deren Elemente *Zeichen* genannt werden.
- Beispiele:
  - Menge der Großbuchstaben: {A,B,C, . . .,Z}
  - Menge der Dezimalziffern: {1,2,3, . . .,9}
  - Menge der Vorzeichen: $\{+, -\}$
  - Menge der Richtungszeiger eines Lifts: $\{\uparrow, \downarrow\}$
- Alphabete, die genau zwei Zeichen enthalten heißen *binär*.
- Ein wichtiges binäres Alphabet besteht aus den *Binärziffern* (*Bits*) $\{0, 1\}$.

Zeichenreihe

- Eine Zeichenreihe über einem Alphabet $\mathcal{A}$ ist eine (endliche) Folge von Zeichen aus $\mathcal{A}$.
- Formal ist auch eine leere Folge eine Zeichenreihe.
- Wir schreiben Zeichenreihen/Folgen $(x_1, x_2, \ldots, x_n)$ auch als $x_1 x_2 \ldots x_n$.
- Beispiele:
  - Sei $\mathcal{A}_1 = \{\text{A,B,C,...,Z}\}$
    - Die Folge INFORMATIK ist eine Zeichenreihe über $\mathcal{A}_1$.
    - Die Folge (Z,I,M,E,K) ist eine Zeichenreihe über $\mathcal{A}_1$.
    - Die Folgen Kröger und (H1234U) sind *keine* Zeichenreihen über $\mathcal{A}_1$. Warum?
  - Sei $\mathcal{A}_2 = \{0, 1\}$
    - Die Folge 0 ist eine Zeichenreihe über $\mathcal{A}_2$.
    - Die Folge 1 ist eine Zeichenreihe über $\mathcal{A}_2$.
    - Die Folge 01 ist eine Zeichenreihe über $\mathcal{A}_2$.
    - Die Folge 10 ist eine Zeichenreihe über $\mathcal{A}_2$.
    - Die Folge 11 ist eine Zeichenreihe über $\mathcal{A}_2$.
    - Die Folge 00 ist eine Zeichenreihe über $\mathcal{A}_2$.
    - ...

Überblick

Bezeichnung von Daten

- Wir verwenden ausschließlich Zeichenreihen zur Bezeichnung von Daten.
- Im folgenden betrachten wir als Beispiel die Darstellung von natürlichen Zahlen, also Elementen der Menge $\mathbb{N}_0$.
- Die Zahl “dreizehn” lässt sich u.a. durch folgende Zeichenreihen bezeichnen:
  - 13 ($\mathcal{A} = \{0, 1, 2, \ldots, 9\}$),
  - DREIZEHN ($\mathcal{A} = \{$A,B,...,Z$\}$),
  - |||||||||||||| ($\mathcal{A} = \{|\}$).
- Nicht alle diese Darstellungen sind für den praktischen Gebrauch (z.B. Rechnen) geeignet.
- Am besten geeignet ist die *Zifferndarstellung*, z.B. die allgemein gebräuchliche *Dezimaldarstellung* über dem Alphabet $\{0, 1, 2, \ldots, 9\}$.

Zifferndarstellung / p-adische Zahlendarstellung

- Das allgemeine Prinzip der Zifferndarstellung ist wie folgt definiert:

  Sei $p \in \mathbb{N}$, $p \geq 2$ und $\mathcal{A}_p = \{z_0, z_1 \ldots, z_{p-1}\}$ ein Alphabet mit p Zeichen (*Ziffern*) $z_0, z_1 \ldots, z_{p-1}$.

  Die Funktion $Z : \mathcal{A}_p \to \mathbb{N}_0$ bildet jedes Zeichen aus $\mathcal{A}_p$ auf eine natürliche Zahl wie folgt ab:

  $Z(z_i) = i \qquad$ für $i = 0, \ldots, p-1$.

  Eine Zeichenreihe $x = x_n x_{n-1} \ldots x_1 x_0$ über $\mathcal{A}_p$ (d.h. $x_i \in \mathcal{A}_p$ für $0 \leq i \leq n$) bezeichnet die Zahl

  $$\mathcal{Z}(x) = p^n \cdot Z(x_n) + p^{n-1} \cdot Z(x_{n-1}) + \ldots + p \cdot Z(x_1) + Z(x_0).$$

- Zur Verdeutlichung schreiben wir auch x_p. x_p heißt *p-adische Zahlendarstellung* der Zahl $\mathcal{Z}(x_p) \in \mathbb{N}_0$.

Zifferndarstellung / p-adische Zahlendarstellung

- Nochmal die Formel

$$\mathcal{Z}(x) = p^n \cdot Z(x_n) + p^{n-1} \cdot Z(x_{n-1}) + \ldots + p \cdot Z(x_1) + Z(x_0).$$

- Beispiele:
  - $p = 10$ und $\mathcal{A}_{10} = \{z_0, z_1 \ldots, z_9\}$ erhält man die Dezimaldarstellung wenn man statt z_i gleich $Z(z_i)$ schreibt (also z.B. statt z_3 schreibe $Z(z_3) = 3$):

    $\mathcal{Z}(983_{10}) = 10^2 \cdot 9 + 10^1 \cdot 8 + 10^0 \cdot 3 =$ "neunhundertdreiundachtzig".

    (Wir schreiben direkt $\mathcal{A}_{10} = \{0, 1 \ldots, 9\}$)
  - $p = 2$ und $\mathcal{A}_2 = \{0, 1\}$ (*Binärdarstellung*):
    $\mathcal{Z}(1111010111_2) =$
    $2^9 \cdot 1 + 2^8 \cdot 1 + 2^7 \cdot 1 + 2^6 \cdot 1 + 2^5 \cdot 0 + 2^4 \cdot 1 + 2^3 \cdot 0 + 2^2 \cdot 1 + 2 \cdot 1 + 1 =$
    "neunhundertdreiundachtzig".
  - $p = 8$ und $\mathcal{A}_8 = \{0, 1, 2, 3, 4, 5, 6, 7\}$ (*Oktaldarstellung*):
    $\mathcal{Z}(1727_8) = 8^3 \cdot 1 + 8^2 \cdot 7 + 8 \cdot 2 + 7 =$ "neunhundertdreiundachtzig".
  - $p = 16$ und $\mathcal{A}_{16} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$ (*Hexadezimaldarstellung*):
    $\mathcal{Z}(3D7_{16}) = 16^2 \cdot 3 + 16 \cdot 13 + 7 =$ "neunhundertdreiundachtzig".

Zifferndarstellung / *p*-adische Zahlendarstellung

- Führende Nullen sind in der Definition der *p*-adischen Zahldarstellung zugelassen, z.B. ist die Zeichenreihe
  000983
  eine zulässige Dezimaldarstellung und bezeichnet die gleiche Zahl wie die Zeichenreihe 983.
- Offensichtlich können führende Nullen (d.h. führende Ziffern 0, also z_0) immer weggelassen werden, außer bei der Bezeichnung “0” für die Zahl “null”.
- Für jede Zahl aus $\mathbb{N}_0$ gibt es für beliebiges $p \geq 2$ eine *p*-adische Darstellung.
- Betrachtet man nur Darstellungen ohne führende Nullen, so ist (zu festem $p \geq 2$) die *p*-adische Zahldarstellung eindeutig.

Menschengerechte Darstellung von Daten

- Zur Entwicklung von Algorithmen werden wir typischerweise die Dezimaldarstellung der natürlichen Zahlen verwenden.
- Allgemein gibt es für die meisten Daten eine *Standarddarstellung* bei denen die Lesbarkeit der Darstellung für den menschlichen Benutzer im Vordergrund steht.
- Wir verwenden hier folgende Standardbezeichnungen wie sie in den üblichen höheren Programmiersprachen gebräuchlich sind:
  - Natürliche Zahlen $\mathbb{N}_0$: Dezimaldarstellung (ohne führende Nullen).
  - Ganze Zahlen $\mathbb{Z}$: wie natürliche Zahlen, ggf. mit Vorzeichen "-", z.B. 3,-3.
  - Reelle Zahlen $\mathbb{R}$: *Gleitpunktdarstellung*, siehe später.
  - Wahrheitswerte $\mathbb{B}$: *TRUE* und *FALSE* für "wahr" bzw. "falsch".

Darstellung von Zeichen und Texten

- Eine Dezimaldarstellung (z.B. 983) stellt eine natürliche Zahl dar, die verarbeitet werden kann.
- Die Zeichenreihe selbst (und nicht die dargestellte Zahl) könnte aber auch Gegenstand der Verarbeitung sein.
- Zeichenreihen können also nicht nur Darstellungen von Objekten sein, sondern auch selbst Objekte, die dargestellt werden müssen.
- Zeichenreihen heißen in diesem Zusammenhang *Texte*.
- In der Praxis wird zur Bildung von Texten häufig das sog. *ASCII-Alphabet* benützt.
- Das ASCII-Alphabet repräsentiert eine Menge von Zeichen, die wir im folgenden als *CHAR* bezeichnen.
- Die Elemente von *CHAR* finden Sie in allen gängigen Lehrbüchern.

Maschinengerechte Darstellung von Daten

- Auf Rechenanlagen wird meist eine andere Darstellung der Daten gewählt (typischerweise Zeichenreihen über den oben benannten Alphabeten $\mathcal{A}_2$, $\mathcal{A}_8$ und $\mathcal{A}_{16}$).
- Aus technischen Gründen kann die kleinste Speichereinheit eines Computers (ein sog. *Bit*) nur zwei Zustände speichern:
  - Zustand 1: es liegt (elektr.) Spannung an.
  - Zustand 0: es liegt keine Spannung an.
- Daher werden Werte (Daten/Objekte) als Bitmuster (Zeichenreihe über dem Alphabet $\mathcal{A}_2 = \{0, 1\}$) codiert gespeichert.
- Intern kann der Computer also z.B. die natürlichen Zahlen in Binärcodierung repräsentieren.
- Ganze Zahlen können intern ebenfalls leicht als Zeichenkette über dem Alphabet $\mathcal{A}_2 = \{0, 1\}$ codiert werden (Genaueres darüber werden Sie in der Vorlesung “Rechnerarchitektur” lernen).

Maschinengerechte Darstellung von Daten

- Die Repräsentation der reellen Zahlen in Binärdarstellung ist etwas komplizierter.
- Üblicherweise wird sowohl im Rechner als auch in den höheren Programmiersprachen die sogenannte *Gleitpunktdarstellung* verwendet. Eine Zahl z wird z.B. im Rechner dargestellt durch

$$z = m \cdot 2^e,$$

  wobei sowohl m (*Mantisse*) als auch e (*Exponent*) wiederum binär repräsentiert werden (können).
- In den meisten höheren Programmiersprachen (z.B. Java) wird eine Zahl dargestellt durch $z = m \cdot 10^e$, z.B. 3.14, -7.45, $1.33E-2$ (für $1.33 \cdot 10^{-2}$).
- Eine genaue Spezifikation lernen wir im nächsten Abschnitt kennen.
- Wichtig: Für viele reelle Zahlen gibt es gar keine derartige Darstellung (z.B. für $\sqrt{2}$). Die darstellbaren Zahlen heißen auch *Gleitpunktzahlen*.
- Dieser Aspekt der maschinengerechten Darstellung von Daten ist bei der Entwicklung von Algorithmen möglicherweise wichtig! Warum?

Maschinengerechte Darstellung von Daten

- Ein weiterer Aspekt der maschinengerechten Darstellung ist, dass die Ressourcen (Speicherzellen) einer Rechenanlagen begrenzt sind.
- Es stehen daher für die Darstellung von Daten immer nur endlich viele Bits zur Verfügung.
- Typischerweise definiert jede Programmiersprache elementare (*primitive* oder *atomare*) *Datentypen*, die Teilmengen der obengenannten Mengen $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{B}$, *CHAR* sind.
- Dabei stehen zur Darstellung der Werte für jeden Datentyp eine fixe Anzahl von Bits zur Verfügung, d.h. die Zeichenketten der Werte eines Typs haben eine fixe *Länge*.
- Werte, deren Darstellung mehr als die für den Typ zur Verfügung stehenden Bits benötigt, können nicht dargestellt werden.
- Die Länge eines Datentyps hat damit offenbar Einfluss auf den Wertebereich des Typs.
- Auch dieser Aspekt der maschinengerechten Darstellung von Daten ist bei der Entwicklung von Algorithmen möglicherweise wichtig!
  Warum?

Grunddatentypen in Java

- Die Programmiersprache Java bietet folgende primitive Datentypen (1Byte = 8 Bit):

Typname	Länge	Wertebereich
`boolean`	1 Byte	Wahrheitswerte {`true`,`false`}
`char`	2 Byte	Alle Unicode-Zeichen
`byte`	1 Byte	Ganze Zahlen von -2^7 bis $2^7 - 1$
`short`	2 Byte	Ganze Zahlen von -2^{15} bis $2^{15} - 1$
`int`	4 Byte	Ganze Zahlen von -2^{31} bis $2^{31} - 1$
`long`	8 Byte	Ganze Zahlen von -2^{63} bis $2^{63} - 1$
`float`	4 Byte	Gleitkommazahlen (einfache Genauigkeit)
`double`	8 Byte	Gleitkommazahlen (doppelte Genauigkeit)

- Es gibt in Java also Grunddatentypen für $\mathbb{B}$, *CHAR*, verschiedene Teilmengen von $\mathbb{Z}$ und verschiedene Teilmengen von $\mathbb{R}$.

Operationen auf Grunddatentypen

- Um Objekte eines bestimmten Typs zu verarbeiten stellen höhere Programmiersprachen auch Operationen auf Objekten des entsprechenden Typs zur Verfügung.
- Beispiel:
  Für das "Rechnen" mit natürlichen Zahlen stehen die Grundrechenarten wie $+$, $\cdot$, usw. als Basisoperationen zur Verfügung.
- Die Operationen, die Java für seine Grunddatentypen bereitstellt, lernen wir später kennen.
- Mathematisch formal handelt es sich bei diesen Grundoperationen typischerweise um *Funktionen*.
- Beispiel:
  Zur Addition zweier natürlicher Zahlen verwenden wir die Funktion

$$+ : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$$

  die zwei natürliche Zahlen auf eine natürliche Zahl abbildet.
- Die Funktion $+$ ist also eine zweistellige Funktion. Grundsätzlich sind natürlich n-stellige Funktionen ($n \geq 0$) erlaubt. Ist die Bedingung $n \geq 0$ im Kontext von Programmiersprachen sinnvoll?

Operationen auf Grunddatentypen

- Üblicherweise schreiben wir statt "$+(x, y)$" "$x + y$" um die beiden Zahlen $x \in \mathbb{N}_0$ und $y \in \mathbb{N}_0$ zu addieren.
- Diese Schreibweise wird *Infixschreibweise* genannt.
- Grundsätzlich gibt es
  - *Präfixschreibweise*: im Beispiel $+(x, y)$;
  - *Infixschreibweise*: im Beispiel $(x + y)$;
  - *Postfixschreibweise*: im Beispiel $(x, y)+$;
- Operationen, die als Ergebnis Objekte vom Typ $\mathbb{B}$ ergeben, heißen auch *Prädikate*, z.B. die Operation $<$ zum Vergleich zweier natürlicher Zahlen:

$$<: \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{B}$$

($1 < 2$ ergibt den Wert $TRUE \in \mathbb{B}$, $3 < 1$ ergibt den Wert $FALSE \in \mathbb{B}$).

Typische Grundoperationen

- Boolesche (Wahrheits-)Werte $\mathbb{B} = \{TRUE, FALSE\}$:

$$\begin{aligned} \wedge &: \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\ \vee &: \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\ \neg &: \mathbb{B} \to \mathbb{B} \end{aligned}$$

- Natürliche Zahlen $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$:

$$\begin{aligned} + &: \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ - &: \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ \cdot &: \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ : &: \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ = &: \mathbb{N} \times \mathbb{N} \to \mathbb{B} \\ \neq &: \mathbb{N} \times \mathbb{N} \to \mathbb{B} \\ > &: \mathbb{N} \times \mathbb{N} \to \mathbb{B} \\ < &: \mathbb{N} \times \mathbb{N} \to \mathbb{B} \end{aligned}$$

Typische Grundoperationen

- Ganze Zahlen $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$:
  $+, -, \cdot, :, =, \neq, <, >, \ldots$
- Reelle Zahlen $\mathbb{R}$ (die reellen Zahlen):
  $+, -, \cdot, :, =, \neq, <, >, \ldots$
- Zeichen (Charakter) *CHAR* = {“A”, “B”, . . ., “a”, “b”, . . ., “1”, “2”, . . ., “!”, . . .}
  (z.B. alle druckbaren ASCII-Zeichen)
  $=, \neq, <, >, \ldots$
- Achtung: obwohl z.B. $+ : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ und $+ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ gleich “benannt” sind, sind es zwei unterschiedliche Operationen!
- Wenn zwei unterschiedliche Operationen gleich benannt sind, spricht man von *Überladen*.
- Achtung: Obwohl Operationen auf den Grunddatentypen meist als gegeben vorausgesetzt werden, verbirgt sich hinter jeder Operation typischerweise wieder ein Algorithmus zu deren Berechnung.

Überblick

Syntax

- Die Gestalt einer Datendarstellung nennt man *Syntax*.
- Die Bedeutung der dargestellten Objekte heißt *Semantik*.
- Die Syntax von Darstellungen von Daten/Objekten kann ohne Bezugnahme auf die Semantik definiert werden.

Syntaxdefinition von natürlichen Zahlen

- Die Menge der natürlichen Zahlen in Dezimaldarstellung kann durch folgende Regeln definiert werden:
  1. 0 ist eine $\langle \textit{Dezimalzahl} \rangle$.
  2. Jede Ziffer $x \in \mathcal{A}_{10} \setminus \{0\}$ ist eine $\langle \textit{Nichtnulldarstellung} \rangle$.
  3. Ist a eine $\langle \textit{Nichtnulldarstellung} \rangle$ und $y \in \mathcal{A}_{10}$ so ist $a \circ y$ eine $\langle \textit{Nichtnulldarstellung} \rangle$.
  4. Jede $\langle \textit{Nichtnulldarstellung} \rangle$ ist eine $\langle \textit{Dezimalzahl} \rangle$.
- Dabei bezeichnet $\circ$ die *Konkatenation* zweier Zeichenketten, z.B. ergibt die Konkatenation $123 \circ 456$ die Zeichenkette 123456.
- Beispiel: Die Zeichenreihe 308 ist eine Dezimalzahl gemäß folgender Anwendungen der Regeln:

  3 ist $\langle \textit{Nichtnulldarstellung} \rangle$ nach Regel 2
  30 ist $\langle \textit{Nichtnulldarstellung} \rangle$ nach Regel 3
  308 ist $\langle \textit{Nichtnulldarstellung} \rangle$ nach Regel 3
  308 ist $\langle \textit{Dezimalzahl} \rangle$ nach Regel 4

Syntaxdefinition von natürlichen Zahlen

- Die Begriffe ⟨*Dezimalzahl*⟩ und ⟨*Nichtnulldarstellung*⟩ in den Regeln werden *syntaktische Variablen* genannt. Sie kennzeichnen den zu definierenden Begriff sowie einen Hilfsbegriff.
- Allgemein können in Syntaxdefinitionen noch mehr syntaktische Variablen vorkommen. Wir heben sie durch kursive Schrift und die eckigen Klammern hervor.
- Die Zeichen des vorgegeben Alphabets heißen *Terminalzeichen*.

Backus-Naur-Form

- Eine formale, häufig gebrauchte Form von Syntaxdefinitionen ist die *Backus-Naur-Form (BNF)*
- Eine Syntaxdefinition in BNF is gegeben durch eine endliche Anzahl von *Syntaxregeln* der Form

$$\alpha ::= \beta$$

- Dabei ist α eine syntaktische Variable und β eine *BNF-Satzform* (siehe nächste Folie).
- Eine ausgezeichnete syntaktische Variable ist das *Startsymbol* α_S.

BNF-Satzform

- Syntaktische Variablen und Terminalzeichen sind BNF-Satzformen.
- Auswahl:
  Sind $\gamma_1, \ldots, \gamma_n$ BNF-Satzformen, dann auch $\gamma_1 | \ldots | \gamma_n$.
- Verkettung (Konkatenation):
  Sind $\gamma_1, \ldots, \gamma_n$ BNF-Satzformen, dann auch $\gamma_1 \ldots \gamma_n$.
- Klammerung:
  Ist γ eine BNF-Satzform, dann auch (γ).
- Iteration:
  Ist γ eine BNF-Satzform, dann auch $(\gamma)^*$.
- nichtleere Iteration:
  Ist γ eine BNF-Satzform, dann auch $(\gamma)^+$.
- Option:
  Ist γ eine BNF-Satzform, dann auch $[\gamma]$.

Bedeutung (Semantik) der BNF

- Eine Zeichenreihe w ist *herleitbar* mit einer BNF Syntaxdefinition, wenn man w aus α_S durch eine oder mehrere der folgenden Ersetzungsoperationen erhalten kann:
  - Eine syntaktische Variable α, für die die Regel $\alpha ::= \gamma_1 | \ldots | \gamma_n$ vorhanden ist, darf durch eines der γ_i $(1 \leq i \leq n)$ ersetzt werden.
  - Ein Vorkommen von $(\gamma_1 | \ldots | \gamma_n)$ darf man durch eines der γ_i $(1 \leq i \leq n)$ ersetzen.
  - Ein Vorkommen von $(\gamma)^*$ darf man durch $\underbrace{(\gamma)(\gamma)\ldots(\gamma)}_{n-\text{mal}}$ mit einem beliebigen $(n \geq 0)$ ersetzen.
  - Ein Vorkommen von $(\gamma)^+$ darf man durch $\underbrace{(\gamma)(\gamma)\ldots(\gamma)}_{n-\text{mal}}$ mit einem beliebigen $(n > 0)$ ersetzen.
  - Ein Vorkommen von (γ) darf man durch γ ersetzen falls γ nicht von der Form $\gamma_1 | \ldots | \gamma_n$ ist.
  - Ein Vorkommen von $[\gamma]$ darf man durch (γ) ersetzen, oder ersatzlos streichen.
- Wir schreiben auch $\alpha_S \rightarrow w$.

Beispiele

- BNF Syntaxdefinition für natürliche Zahlen:

  $\langle Dezimalzahl \rangle$ ::= 0 | $\langle Nichtnulldarstellung \rangle$

  $\langle Nichtnulldarstellung \rangle$ ::= $\langle Nichtnullziffer \rangle$ $(0 \mid \langle Nichtnullziffer \rangle)^*$

  $\langle Nichtnullziffer \rangle$ ::= 1|2|3|4|5|6|7|8|9

  Die syntaktische Variable $\langle Dezimalzahl \rangle$ dient als Startsymbol α_S.

- Ableitung der Zeichenreihe 308 aus $\alpha_S = \langle Dezimalzahl \rangle$ ist:

$$
\begin{aligned}
\langle Dezimalzahl \rangle &\rightarrow \langle Nichtnulldarstellung \rangle \\
&\rightarrow \langle Nichtnullziffer \rangle \ (0 \mid \langle Nichtnullziffer \rangle)^* \\
&\rightarrow \langle Nichtnullziffer \rangle \ (0 \mid \langle Nichtnullziffer \rangle)(0 \mid \langle Nichtnullziffer \rangle) \\
&\rightarrow \langle Nichtnullziffer \rangle \ 0 \ \langle Nichtnullziffer \rangle \\
&\rightarrow 308
\end{aligned}
$$

Beispiele

- BNF Syntaxdefinition für ganze Zahlen ("$-$" und die Dezimalziffern des Alphabets $\mathcal{A}_{10}$ sind Terminalzeichen):

  $\langle GanzeZahl \rangle ::= [-] \langle Dezimalzahl \rangle$

  Die syntaktische Variable $\langle GanzeZahl \rangle$ dient hier als Startsymbol α_S.

- BNF Syntaxdefinition für Gleitpunktzahlen ("$-$", "E" und die Dezimalziffern des Alphabets $\mathcal{A}_{10}$ sind Terminalzeichen):

  $\langle Gleitpunktzahl \rangle ::= [-] \langle Mantisse \rangle (E \langle GanzeZahl \rangle)$

  $\langle Mantisse \rangle ::= \langle Dezimalzahl \rangle . \langle Dezimalstellen \rangle$

  $\langle Dezimalstellen \rangle ::= 0 | \langle Nichtnullstellen \rangle$

  $\langle Nichtnullstellen \rangle ::= (0 | \langle Nichtnullziffer \rangle)^* \langle Nichtnullziffer \rangle$

  Die syntaktische Variable $\langle Gleitpunktzahl \rangle$ dient hier als Startsymbol α_S.

Diskussion

- Mit Hilfe der BNF kann man nicht nur die Syntax (Darstellung) von Daten formal definieren.
- Viele Programmiersprachen benützen BNF Syntaxdefinitionen auch für die eindeutige Definition ihrer Syntax, d.h. der Spracheelemente, die erlaubt sind und der Sprache an sich.
- Manche Eigenschaften von Darstellungen oder Sprachen kann man in BNF allerdings nicht darstellen. Diese werden dann typischerweise zusätzlich “verbal” angegeben, d.h. sie werden als Zusatzanforderungen notiert. Diese zusätzlichen Angaben heißen *Kontextbedingungen*.

Überblick

Eigenschaften von Algorithmen

- Nach dem wir die erste der auf Folie 23 genannten Eigenschaften, eindeutige Datendarstellung, diskutiert haben, widmen wir uns im folgenden den restlichen drei Eigenschaften von Algorithmen.
- Neben diesen Forderungen gibt es noch weitere interessante Eigenschaften, die Algorithmen haben können.
- Wir werden all diese Eigenschaften an einem Beispiel erläutern.

Eigenschaften von Algorithmen

Die restlichen drei Eigenschaften von Algorithmen auf Folie 23 sind

1. Präzise, endliche Beschreibung.
2. Effektiver Verarbeitungsschritt.
3. Elementarer Verarbeitungsschritt.

Dazu kommen folgende Eigenschaften:

4. Ein Algorithmus heißt *terminierend*, wenn er bei jeder Anwendung nach endlich vielen Verarbeitungsschritten zum Ende kommt.
5. Ein Algorithmus heißt *deterministisch*, wenn die Wirkung und die Reihenfolge der Einzelschritte eindeutig festgelegt ist, andernfalls *nicht-deterministisch*.
6. Ein Algorithmus heißt *determiniert*, wenn das Ergebnis der Verarbeitung für jede einzelne Anwendung eindeutig bestimmt ist, andernfalls *nicht-determiniert*.

Eigenschaften von Algorithmen

Desweiteren ist natürlich die möglicherweise wichtigste Eigenschaft eines Algorithmus die *Korrektheit*, d.h. informell, “der Algorithmus tut, was er tun soll”.

1. Ein Algorithmus heißt *partiell korrekt*, wenn für alle gültigen Eingaben das Resultat der Spezifikation des Algorithmus entspricht.
2. Ein Algorithmus heißt *(total) korrekt*, wenn der Algorithmus partiell korrekt ist, und für alle gültigen Eingaben terminiert.

Wir werden uns später noch genauer mit der Frage beschäftigen, wie man sicher sein kann, dass ein Algorithmus partiell/total korrekt ist.

Running Example

Wir betrachten folgende Aufgabe:

Ein Kunde kauft Waren für $1 \leq r \leq 100$ EUR und bezahlt mit einem 100 EUR Schein (r sei ein voller EUR Betrag ohne Cent-Anteil). Gesucht ist ein Algorithmus, der zum Rechnungsbetrag r das Wechselgeld w bestimmt. Zur Vereinfachung nehmen wir an, dass w nur aus 1 EUR oder 2 EUR Münzen oder 5 EUR Scheinen bestehen soll. Es sollen möglichst wenige Münzen/Scheine ausgegeben werden (also ein 5 EUR Schein statt fünf 1 EUR Münzen).

Datendarstellung

- Zunächst müssen wir zur Lösung dieser Aufgabe die Darstellung (Modellierung) der relevanten Daten festlegen.
- Für den Rechnungsbetrag r ist dies trivial, denn offensichtlich ist $r \in \mathbb{N}$. Wir nehmen an, dass r in Dezimaldarstellung gegeben ist.
- Das Wechselgeld w kann auf verschiedene Weise modelliert werden, z.B. als Folge oder Multimenge von Wechselgeldmünzen. Ein aus zwei 1-EUR-Münzen, einer 2-EUR-Münze und zwei 5-EUR-Scheinen bestehendes Wechselgeld könnte als Folge $(1, 1, 2, 5, 5)$ dargestellt sein.
- Wir legen folgende Datendarstellung fest:
  - r: als natürliche Zahl in Dezimaldarstellung.
  - w: als Folge von Werten 1, 2 oder 5.
- Wir benutzen dabei die Bezeichnung $()$ für die *leere Folge* und die Funktion $\circ$ zur Konkatenation zweier Folgen wie oben. Desweiteren sagen wir auch “nimm x zu w hinzu” für die Operation $w \circ x$.

Erster Lösungsansatz

- Idee:
  Ausgehend von r sukzessive um 1,2 und 5 hochzählen (unter Hinzunahme der entsprechenden Münzen/Scheine) bis man bei 100 angelangt ist. Dabei möglichst schnell eine durch 5 teilbare Zahl erreichen, um möglichst wenige Münzen/Scheine auszugeben.
- Algorithmus *Wechselgeld 1*

  Führe folgende Schritte der Reihe nach aus:

  1. Setze $w = ()$.
  2. Falls die letzte Ziffer von r eine 2,4,7 oder 9 ist, dann erhöhe r um 1 und nimm 1 zu w hinzu.
  3. Falls die letzte Ziffer von r eine 1 oder 6 ist, dann erhöhe r um 2 und nimm 2 zu w hinzu.
  4. Falls die letzte Ziffer von r eine 3 oder 8 ist, dann erhöhe r um 2 und nimm 2 zu w hinzu.
  5. Solange $r < 100$: Erhöhe r um 5 und nimm 5 zu w hinzu.

Erster Lösungsansatz

- Ablaufbeispiel: Sei $r = 81$.

	$r = 81$		(Ausgangssituation)
Schritt 1	$r = 81$	$w = ()$	
Schritt 2	(keine Änderung, da die letzte Ziffer keine 2,4,7,9 ist)		
Schritt 3	$r = 83$	$w = (2)$	
Schritt 4	$r = 85$	$w = (2, 2)$	
Schritt 5	$r = 90$	$w = (2, 2, 5)$	
	$r = 95$	$w = (2, 2, 5, 5)$	
	$r = 100$	$w = (2, 2, 5, 5, 5)$	

- Eigenschaften:
  - Endliche Aufschreibung: OK.
  - Effektive und elementare Einzelschritte: ?
  - Der Algorithmus ist terminierend, deterministisch, determiniert und partiell korrekt (und damit auch total korrekt).

Variante des ersten Lösungsansatzes

- Idee:
  Fasse ähnliche Schritte zusammen
- Algorithmus *Wechselgeld 2*

  Führe folgende Schritte der Reihe nach aus:

  1. Setze $w = ()$.
  2. Solange $r < 100$: Führe jeweils (wahlweise) einen der folgenden Schritte aus:
     1. Falls die letzte Ziffer von r eine 2,4,7 oder 9 ist, dann erhöhe r um 1 und nimm 1 zu w hinzu.
     2. Falls die letzte Ziffer von r eine 1,2,3,6,7 oder 8 ist, dann erhöhe r um 2 und nimm 2 zu w hinzu.
     3. Falls die letzte Ziffer von r eine 0,1,2,3,4 oder 5 ist, dann erhöhe r um 5 und nimm 5 zu w hinzu.

Variante des ersten Lösungsansatzes

- Ablaufbeispiel:

	$r = 81$		(Ausgangssituation)
Schritt 1	$r = 81$	$w = ()$	
Schritt 2b	$r = 83$	$w = (2)$	
Schritt 2b	$r = 85$	$w = (2, 2)$	
Schritt 2c	$r = 90$	$w = (2, 2, 5)$	
Schritt 2c	$r = 95$	$w = (2, 2, 5, 5)$	
Schritt 2c	$r = 100$	$w = (2, 2, 5, 5, 5)$	

- Alternativer Ablauf:

	$r = 81$		(Ausgangssituation)
Schritt 1	$r = 81$	$w = ()$	
Schritt 2b	$r = 83$	$w = (2)$	
Schritt 2c	$r = 88$	$w = (2, 5)$	
Schritt 2b	$r = 90$	$w = (2, 5, 2)$	
Schritt 2c	$r = 95$	$w = (2, 5, 2, 5)$	
Schritt 2c	$r = 100$	$w = (2, 5, 2, 5, 5)$	

Variante des ersten Lösungsansatzes

- Eigenschaften:
  - Algorithmus 2 arbeitet nach dem selben Grundprinzip wie Algorithmus 1, läßt aber gewisse “Freiheiten” bei der Auswahl des nächsten Schrittes beim wiederholt auszuführenden Schritt 2.
    Daher: Algorithmus 2 ist nicht-deterministisch.
  - Algorithmus 2 ist nicht determiniert (siehe alternativer Ablauf): Bei ein und derselben Anwendung erzeugt der Algorithmus zwei unterschiedliche Folgen $w = (2, 2, 5, 5, 5)$ und $w = (2, 5, 2, 5, 5)$.
  - Bemerkung: wenn wir statt Folgen Multimengen (bei denen die Reihenfolge der Elemente keine Rolle spielt) bei der Darstellung von w verwendet hätten, wären die Ergebnisse $w = \{2, 2, 5, 5, 5\}$ und $w = \{2, 5, 2, 5, 5\}$ gleich, d.h. in diesem Fall wäre Algorithmus 2 determiniert.
- Die Wahl der Datendarstellung kann also Einfluss auf Eigenschaften von Algorithmen haben.
- Bemerkung: Würden wir r nicht in Dezimal- sondern z.B. in Binärdarstellung modellieren, wären beide Algorithmen in der angegebenen Form sinnlos.

Zweite Variante des ersten Lösungsansatzes

- Idee:
  Wir lösen uns von der Abhängigkeit von der Datendarstellung, indem wir r nicht mehr hochzählen, sondern die Differenz $100 - r$ berechnen und diese in möglichst wenige Teile der Größen 1, 2 und 5 aufteilen. Wir gehen dabei davon aus, dass die Differenz "$-$" für beliebige Zifferndarstellungen (dezimal, binär, oktal, etc.) definiert ist. Die Zahl 100 müsste in die selbe Darstellung gebracht werden, die auch für r benutzt wird.
- Algorithmus *Wechselgeld 3*

  Führe folgende Schritte der Reihe nach aus:

  1. Berechne $d = 100 - r$ und setze $w = ()$.
  2. Solange $d \geq 5$: Vermindere d um 5 und nimm 5 zu w hinzu.
  3. Falls $d \geq 2$, dann vermindere d um 2 und nimm 2 zu w hinzu.
  4. Falls $d \geq 2$, dann vermindere d um 2 und nimm 2 zu w hinzu.
  5. Falls $d \geq 1$, dann vermindere d um 1 und nimm 1 zu w hinzu.

Zweite Variante des ersten Lösungsansatzes

- Ablaufbeispiel:

	$r = 81$		(Ausgangssituation)
Schritt 1	$d = 19$	$w = ()$	
Schritt 2	$d = 14$	$w = (5)$	
	$d = 9$	$w = (5, 5)$	
	$d = 4$	$w = (5, 5, 5)$	
Schritt 3	$d = 2$	$w = (5, 5, 5, 2)$	
Schritt 4	$d = 0$	$w = (5, 5, 5, 2, 2)$	
Schritt 5	(keine Änderung, da $d \geq 1$ nicht gilt)		

Diskussion

- Alle drei Varianten weisen eine gemeinsame “operative” Auffassung auf: Die Berechnung des Rückgelds wird durch eine Folge von “Handlungen” vollzogen.
- Diese Handlungen sind charakterisiert durch Veränderungen der Größen *r* bzw. *d* und *w*.
- Die Abfolge dieser “Aktionen” wird durch typische Konstruktionen gesteuert:
  - *Fallunterscheidung*: Falls . . ., dann . . .
    ermöglicht, die Ausführungen von Aktionen vom Erfülltsein gewisser Bedingungen abhängig sein zu lassen.
  - *Iteration*: Solange . . .: . . .
    steuert die Wiederholung von Aktionen in Abhängigkeit gewisser Bedingungen.

Diskussion

- Neben der strikten operativen Auffassung gibt es eine Sichtweise von Algorithmen, die durch eine rigorose mathematische Abstraktion gewonnen wird.
- In unserem Beispiel ist die Zuordnung eines Wechselgelds w zu einem Rechnungsbetrag r mathematisch nichts anderes als eine Abbildung

$$h : r \mapsto \text{“herauszugebendes Wechselgeld”}$$

- Die Aufgabe ist dann, eine Darstellung der Abbildung h zu finden, die “maschinell auswertbar” ist.
- Eine triviale Lösung ist, in einer kompletten Aufstellung alle möglichen Werte für r mit zugehörigem Wechselgeld $w = h(r)$ aufzulisten.
- Dies ist allgemein aber nur für endliche Definitionsbereiche möglich und sogar nur für “kleine” Definitionsbereiche sinnvoll.
- Allgemein wird man eine kompakte Darstellung suchen, in der die zu bestimmende Abbildung in geeigneter Weise aus einfachen (elementar auswertbaren) Abbildungen zusammengesetzt ist.

Vorbereitungen

- Für eine solche Darstellung der Abbildung h benötigen wir zwei "Hilfsabbildungen", die in der Informatik oft gebräuchlich sind:

$$DIV : \mathbb{N}_0 \times \mathbb{N} \to \mathbb{N}_0$$
$$MOD : \mathbb{N}_0 \times \mathbb{N} \to \mathbb{N}_0$$

- *DIV* berechnet das Ergebnis der ganzzahlige Division zweier natürlicher Zahlen, z.B. $DIV(7,3) = 2$.
- *MOD* berechnet den Rest der ganzzahligen Division zweier natürlicher Zahlen, z.B. $MOD(7,3) = 1$.
- Formal: Sind $k \in \mathbb{N}_0$ und $l \in \mathbb{N}$, so kann man k durch l mit ganzzahligem Ergebnis q teilen, wobei eventuell ein Rest r übrigbleibt, d.h. es gibt $q, r \in \mathbb{N}_0$ mit $r < l$ und $k = q \cdot l + r$. Dann ist

$$DIV(k,l) = q \quad \text{und} \quad MOD(k,l) = r.$$

Zweiter Lösungsansatz

- Idee:
  Ähnlich wie Algorithmus 3 teilen wir $100 - r$ durch 5. Der ganzzahlige Quotient $q_1 = DIV(100 - r, 5)$ ist die Anzahl der 5-EUR-Scheine im Wechselgeld $h(r)$. Der Rest $r_1 = MOD(100 - r, 5)$ ist der noch zu verarbeitende Wechselbetrag. Offensichtlich gilt $r_1 < 5$. r_1 muss nun auf 1 und 2 aufgeteilt werden, d.h. analog bilden wir $q_2 = DIV(r_1, 2)$ und $r_2 = MOD(r_1, 2)$. q_2 bestimmt die Anzahl der Elemente 2 und r_2 die Anzahl der Elemente 1 in $h(r)$.
- Algorithmus *Wechselgeld 4*

$$\begin{aligned} h(r) \quad = \quad & (5_1, \ldots, 5_{DIV(100-r,5)} \\ & 2_1, \ldots, 2_{DIV(MOD(100-r,5),2)} \\ & 1_1, \ldots, 2_{MOD(MOD(100-r,5),2)}) \end{aligned}$$

  Dabei sind alle Elemente in der Ergebnisfolge durchnummeriert, um die Anzahl der entsprechenden Elemente anzudeuten, d.h. das Element 5_i bezeichnet den i-ten 5-EUR-Schein im Ergebnis.

Zweiter Lösungsansatz

- Bemerkung:
  In dieser Schreibweise kann z.B. $(5_1, \ldots 5_0)$ auftreten, was bedeuten soll, dass die Folge kein Element 5 enthält.
- Beispielanwendung $r = 81$:

$$DIV(100 - r, 5) = DIV(19, 5) = 3$$
$$MOD(100 - r, 5) = MOD(19, 5) = 4$$
$$DIV(MOD(100 - r, 5), 2) = DIV(4, 2) = 2$$
$$MOD(MOD(100 - r, 5), 2) = MOD(4, 2) = 0$$

d.h. man erhält $h(81) = (5_1, 5_2, 5_3, 2_1, 2_2, 1_1, 1_0)$ oder, einfacher geschrieben,

$$h(81) = (5, 5, 5, 2, 2).$$

Zweiter Lösungsansatz

- Eigenschaften:
  - Wenn man von der Bildung der Folgen aus den berechneten Anzahlen der Elemente 1, 2 und 5 absieht, sind nur die Auswertungen der Operationen "$-$", *DIV* und *MOD* als Einzelschritte anzusehen. Setzt man diese als elementar voraus, ist die angegebene Definition von *h* eine Beschreibung des gesuchten Algorithmus.
  - Diese Darstellung nimmt keinen Bezug mehr auf eine "Abfolge von Aktionen" sondern beschreibt den "funktionalen Zusammenhang" zwischen *r* und *w* durch ineinander eingesetzte Anwendungen gewisser Basisoperationen.

Variante des zweiten Lösungsansatzes

- Leider läßt sich in vielen Fällen die gesuchte Abbildung nicht in derartig einfacher Weise explizit angeben.
- Ein wichtiges Prinzip für den Allgemeinfall von Abbildungen auf natürlichen Zahlen (und anderen Mengen) ist das Prinzip der *Rekursion*, das wir im nächsten Abschnitt noch etwas genauer untersuchen, hier aber schon einmal informell einführen.
- Kernidee einer rekursiven Definition einer Abbildung f mit Definitionsbereich $\mathbb{N}_0$ ist es, für einen oder mehrere *Basisfälle* die Abbildung explizit anzugeben (typischerweise $f(0)$) und den allgemeinen *Rekursionsfall* $f(n)$ auf den Fall $f(n-1)$ zurückzuführen.

Variante des zweiten Lösungsansatzes

- Idee:
  Auch im Fall der Abbildung $h : \mathbb{N}_0 \rightarrow$ “Folgen über 1,2,5” kann man sehr leicht eine rekursive Definition finden, die die Operationen *DIV* und *MOD* nicht mehr verwendet.
- Algorithmus *Wechselgeld 5*

$$h(r) = \begin{cases} \text{falls } r = 100, \text{ dann} \quad (), & (1) \\ \text{falls } 100 - r \geq 5, \text{ dann} \quad (5) \circ h(r+5), & (2) \\ \text{falls } 5 > 100 - r \geq 2, \text{ dann} \quad (2) \circ h(r+2), & (3) \\ \text{falls } 2 > 100 - r \geq 1, \text{ dann} \quad (1) \circ h(r+1), & (4) \end{cases}$$

- Hier ist der Basisfall (1) für $h(100)$ definiert und die Rekursionsfälle (2),(3),(4) werden auf die Fälle $h(r+5)$, $h(r+2)$ und $h(r+1)$ zurückgeführt.

Variante des zweiten Lösungsansatzes

- Beispielanwendung $r = 81$:

$$
\begin{aligned}
h(81) &= (5) \circ h(86) && \text{(Fall 2)}\\
&= (5) \circ (5) \circ h(91) && \text{(Fall 2)}\\
&= (5) \circ (5) \circ (5) \circ h(96) && \text{(Fall 2)}\\
&= (5) \circ (5) \circ (5) \circ (2) \circ h(98) && \text{(Fall 3)}\\
&= (5) \circ (5) \circ (5) \circ (2) \circ (2) \circ h(100) && \text{(Fall 3)}\\
&= (5) \circ (5) \circ (5) \circ (2) \circ (2) \circ () && \text{(Fall 1)}\\
&= (5, 5, 5, 2, 2)
\end{aligned}
$$

Variante des zweiten Lösungsansatzes

- Trotz der rein funktionalen Darstellung der Zuordnung $r \mapsto w$ erinnert die Auswertung wieder an die Abfolge der Aktionen im entsprechenden Ablaufbeispiel (ähnlich wie in Algorithmus 3).
- Tatsächlich kann das Prinzip der Rekursion auch in der operativen Auffassung der Algorithmen 1-3 eingerichtet werden und dabei einen ähnlichen Effekt wie die Iteration erzielen.
- In einer rekursiven Abbildungsdefinition greift man auf Werte dieser Abbildung für kleinere (oder hier: größere) Argumente zurück.
- Analog kann in einem Algorithmus, der die Abfolge gewisser Aktionen in Abhängigkeit von Eingabewerten steuert, die Ausführung des Algorithmus selbst auch als eine derartige Aktion auftreten.

Zweite Variante des zweiten Lösungsansatzes

- Algorithmus *Wechselgeld 6*

  Setze $w = ()$ und führe anschließend folgenden in Abhängigkeit der Eingabegröße r rekursiv definierten Algorithmus $A(r)$ aus:
  $A(r)$: Führe denjenigen der folgenden Schritte (1) - (3) aus, dessen Bedingung erfüllt ist (ist keine Bedingung erfüllt, ist die Ausführung zu Ende):

  1. Falls $100 - r \geq 5$, dann nimm 5 zu w hinzu und führe anschließend $A(r+5)$ aus.
  2. Falls $5 > 100 - r \geq 2$, dann nimm 2 zu w hinzu und führe anschließend $A(r+2)$ aus.
  3. Falls $5 > 100 - r \geq 1$, dann nimm 1 zu w hinzu und führe anschließend $A(r+1)$ aus.

Diskussion

- Grundlegende algorithmische Konzepte sind (z.T. abhängig vom verwendeten Programmierparadigma):
  - Eingabe- und Ergebnisdaten (insbesondere solche von komplexer Art) müssen geeignet dargestellt werden.
  - Die Ausführung gewisser Aktionen bzw. die Auswertung gewisser Operationen wird als in elementaren Schritten durchführbar vorausgesetzt.
  - Einzelne Schritte können zusammengesetzt werden (z.B. Nacheinanderausführung, Auswahl von Aktionen, Kompositionen von Abbildungen, etc.)
  - Fallunterscheidung, Iteration und Rekursion ermöglichen die Steuerung des durch den Algorithmus beschriebenen Verfahrens.

Unterschiedliche Programmierparadigmen

- *Imperative Algorithmen*:
  - Imperative Algorithmen spezifizieren die Abfolge von Aktionen, die typischerweise bestimmte Größen verändern.
  - Diese Auffassung spiegelt auch die technischen Möglichkeiten von Rechneranlagen wider, die Schritt für Schritt bestimmte grundlegende Aktionen ausführen können.
- *Funktionale Algorithmen*:
  - Funktionale Algorithmen spezifizieren eine auswertbare Darstellung des funktionalen Zusammenhangs zwischen Ein- und Ergebniswerten.
  - Diese Auffassung spiegelt ein höheres Abstraktionsniveau wider: Obwohl intern (d.h. zur Auswertung der entsprechenden Abbildung) wiederum bestimmte Schritte ausgeführt werden, ist die eigentliche Spezifikation des Algorithmus als Abbildung praktisch losgelöst von den technischen Möglichkeiten der Rechenanlage.
- *Objektorientierte Algorithmen*:
  - Objektorientierte Algorithmen spezifizieren eine höhere Abstraktionsebene zur Darstellung von komplexen Eingabe- und Ergebnisdaten sowie Zwischenzuständen während der Berechnung.
  - Zur eigentlichen Lösung der gegebenen Aufgabe können sie sowohl die funktionale als auch die imperative Auffassung verwenden.

Abschnitt 3: Mathematische Grundlagen

Überblick

Mengen

- Die Charakterisierung von Daten in der Vorlesung setzt den Mengen-Begriff voraus.
- Als informelle Definition genügt uns:

  Eine *Menge M* ist eine Zusammenfassung von verschiedenen Objekten, den *Elementen* der Menge. Die Notation $a \in M$ bedeutet: a ist ein Element der Menge M.
- Eine Menge kann beliebig viele Elemente enthalten, also z.B. auch gar keine. In diesem Fall spricht man von der leeren Menge, geschrieben $\emptyset$ oder $\{\}$.

Mengen

- Eine Menge kann z.B. *extensional* durch Aufzählung der Elemente definiert werden.
- Die Reihenfolge der Elemente spielt dabei keine Rolle.
- Man kann eine Menge aber auch *intensional* definieren, d.h. durch Angabe einer Bedingung, die alle Elemente und nur die Elemente der Menge erfüllen.
- Beispiel: Menge M der Quadratzahlen, die kleiner als 30 sind
  - Extensional: $M = \{1, 4, 9, 16, 25\} = \{4, 1, 9, 25, 16\}$
  - Intensional: $M = \{a | a \in \mathbb{N}, a \text{ ist Quadratzahl und } a < 30\}$
- Beispiel leere Menge:
  - Extensional: $M = \{\}$
  - Intensional: $M = \{a | a \in \mathbb{N}, a < 2 \text{ und } a > 1\}$

Eigenschaften von Mengen

- Alle Elemente einer Menge sind verschieden.
- Man könnte zwar eine Menge $\{1, 2, 2, 3\}$ angeben, dies wäre aber redundant. Die gleiche Menge wird durch $\{1, 2, 3\}$ definiert.
- Mit dem Konzept einer Menge kann man also nicht mehrfaches Vorkommen eines gleichen Elementes (Wertes) modellieren.
- Hierzu dient z.B. das Konzept der Multimengen, die wie Mengen geschrieben werden, aber andere Eigenschaften und Rechenregeln haben.
- Soll auch die Reihenfolge der Elemente eine Rolle spielen, sind Folgen eine geeignete Modellierung (wie wir sie in den Algorithmen in Kapitel 2 verwendet haben).

Eigenschaften von Mengen

- Es gelten folgende wichtige Eigenschaften von Mengen und Beziehungen zwischen Mengen:

Bezeichnung	Notation	Bedeutung
M ist Teilmenge von N	$M \subseteq N$	aus $a \in M$ folgt $a \in N$
M ist echte Teilmenge von N	$M \subset N$	es gilt $M \subseteq N$ und $M \neq N$
Vereinigung von M und N	$M \cup N$	$\{x \mid x \in M$ oder $x \in N\}$
Schnittmenge von N und M	$M \cap N$	$\{x \mid x \in M$ und $x \in N\}$
Differenz M ohne N	$M \setminus N$	$\{x \mid x \in M$ und $x \notin N\}$
M und N sind disjunkt	$M \cap N = \emptyset$	M und N haben keine gemeinsamen Elemente
Kardinalität einer Menge M	$\lvert M \rvert$	Anzahl der Elemente von M

Tupel, kartesisches Produkt von zwei Mengen

- Die Elemente einer Menge können selbst zusammengesetzt sein aus verschiedenen Mengen.
- Ein *geordnetes Paar (Tupel)* (x, y) besteht aus zwei Werten x und y, wobei x die erste und y die zweite Komponente ist.
- Beispiel: Eine Spielkarte hat eine Farbe und ein Symbol: *(Karo,Bube)*, *(Herz,Dame)*.
- Das *kartesische Produkt* $M \times N$ zweier Mengen M und N ist die Menge aller geordneten Paare mit erster Komponente aus M und zweiter Komponente aus N:

$$M \times N := \{(x, y) | x \in M \text{ und } y \in N\}.$$

- Beispiel: Für $S = \{7, 8, 9, 10, \mathit{Bube}, \mathit{Dame}, \mathit{König}, \mathit{Ass}\}$ und $F = \{\mathit{Kreuz}, \mathit{Pik}, \mathit{Herz}, \mathit{Karo}\}$, können wir ein Kartenspiel als die Menge $F \times S$ definieren.

n-Tupel, kartesisches Produkt von n Mengen

- Die Konzepte “Tupel” und “kartesisches Produkt zweier Mengen” lassen sich leicht auf eine beliebige Anzahl n von Mengen verallgemeinern.
- Das allgemeine kartesische Produkt über n Mengen ist die Menge aller geordneten n-Tupel mit den Komponenten aus diesen Mengen:

$$M_1 \times M_2 \times \ldots \times M_n := \{(a_1, a_2, \ldots, a_n) | a_1 \in M_1 \text{ und } a_2 \in M_2 \ldots \text{ und } a_n \in M_n\}.$$

- Sind alle Mengen identisch ($M_i = M_j$ für alle $1 \leq i, j \leq n$), schreibt man für $M \times M \times \ldots \times M$ häufig auch M^n.

Potenzmenge

- Viele Objekte werden durch Mengen beschrieben.
- Der Wertebereich einer Datenmenge solcher Objekte ist dann eine Menge, die Mengen enthält.
- Eine spezielle Form solcher Mengen von Mengen ist die *Potenzmenge*:
- Die Potenzmenge einer Grundmenge U ist die Menge aller Teilmengen von U, geschrieben $\wp(U)$.
- Beispiel: $U = \{d, f, s\}$
  $\wp(U) = \{\emptyset, \{d\}, \{f\}, \{s\}, \{d, f\}, \{d, s\}, \{f, s\}, \{d, f, s\}\}$
- Für eine Menge der Kardinalität n, welche Kardinalität hat ihre Potenzmenge?

Potenzmenge

- Anwendungsbeispiel: Modellieren der Lösungsmenge einer Aufgabe.
- Aufgabe: Anzahl der Münzen, die nötig sind, einen bestimmten Geldbetrag auszuzahlen.
- Je nach Geldbetrag und den Werten der verfügbaren Münzarten gibt es keine, eine oder mehrere Zahlen als Antwort.
- Der Wertebereich der prinzipiell möglichen Lösungen ist $\wp(\mathbb{N})$.

Relationen

- Eine *n-stellige Relation* ist eine Menge von *n*-Tupeln.
- Prädikate wie z.B. die Beziehung “kleiner” ($a < b$) sind Beispiele für Relationen. Wir können also z.B. schreiben:
  - $<\subseteq \mathbb{N} \times \mathbb{N}$
  - $(1, 2) \in <$
  - $(2, 1) \notin <$
- Eine Relation R ist *erfüllt* (oder *wahr*) für alle Tupel a mit $a \in R$ und nur für diese Tupel. Man schreibt auch: Ra.
- Für zweistellige Relationen schreibt man auch xRy (z.B.: $x < y$).

Eigenschaften zweistelliger Relationen

Sei $R \subseteq M \times M$ (d.h. $R \in \wp(M \times M)$).

- R ist reflexiv, wenn für alle $x \in M$ gilt: xRx.
- R ist symmetrisch, wenn für alle $x, y \in M$ gilt: aus xRy folgt yRx.
- R ist antisymmetrisch, wenn für alle $x, y \in M$ gilt: aus xRy und yRx folgt $x = y$.
- R ist transitiv, wenn für alle $x, y, z \in M$ gilt: aus xRy und yRz folgt xRz.
- R ist alternativ, wenn für alle $x, y \in M$ gilt: xRy oder yRx.

Äquivalenzrelation

- Sei $R \in \wp(M \times M)$.
- R ist eine *Äquivalenzrelation*, wenn R reflexiv, symmetrisch und transitiv ist.
- Beispiel: Wenn für ein Kartenspiel $F \times S$ mit
  $S = \{7, 8, 9, 10, \mathit{Bube}, \mathit{Dame}, \mathit{König}, \mathit{Ass}\}$ und
  $F = \{\mathit{Kreuz}, \mathit{Pik}, \mathit{Herz}, \mathit{Karo}\}$ beim Vergleich zweier Karten die Farbe keine Rolle spielt, sondern zwei Karten mit dem gleichen Symbol als gleich gelten, ist die entsprechende Äquivalenzrelation:

$$\{((f_1, s_1), (f_2, s_2)) \mid f_1 \in F, f_2 \in F, s_1 \in S, s_2 \in S, s_1 = s_2\}$$

Ordungsrelationen

Bestimmte Kombinationen von Eigenschaften qualifizieren eine Relation zur *Ordnungsrelation*, durch deren Anwendung man beispielsweise eine Menge sortieren könnte:
Sei $R \in \wp(M \times M)$.

- R ist eine partielle Ordnung, wenn R reflexiv, antisymmetrisch und transitiv ist.
- R ist eine totale Ordnung, wenn R eine alternative partielle Ordnung ist.

Auf den ganzen Zahlen gibt es die Ordnungsrelation $\leq \in \wp(\mathbb{Z} \times \mathbb{Z})$. Ist diese Ordnungsrelation total?

Funktionen (Abbildungen)

- Eine *Funktion* ist eine Abbildung von einer bestimmten Menge auf eine bestimmte Menge.
- Funktionen können wir als Relationen mit speziellen Eigenschaften auffassen, sie stellen nämlich eine *rechtseindeutige* Beziehung zwischen Definitionsbereich und Bildbereich dar.
- Formal gesehen ist eine Funktion f eine 2-stellige Relation $f \subseteq D \times B$, für die gilt: Aus $(x, y) \in f$ und $(x, z) \in f$ folgt $y = z$, d.h. einem Element aus D ist höchstens ein Element aus B eindeutig zugeordnet.
- Die Menge D heißt *Definitionsbereich* von f.
- Die Menge B heißt *Bildbereich* von f.
- Als Schreibweisen sind gebräuchlich:

$$(x, y) \in f \iff y = f(x) \iff f(x) = y \iff f : x \mapsto y$$

Signatur von Funktionen

Da Funktionen spezielle Relationen sind, stammen sie aus Wertebereichen, die Teilmengen der Wertebereiche entsprechend strukturierter Relationen sind:

- Die Menge $D \to B$ ist die Menge aller Funktionen, die D als Definitionsbereich und B als Bildbereich haben. $D \to B$ enthält als Elemente alle Mengen von Tupeln $(d, b) \in (D \times B)$, die Funktionen sind.
- Es gilt: $D \to B \subseteq \wp(D \times B)$.
- Für eine Funktion $f \in D \to B$ gilt: $f \subseteq D \times B$.
- Man schreibt: $f : D \to B$, d.h. f hat die *Signatur* $D \to B$.
- "Signatur" ist ein zentrales Konzept in der Spezifikation von Programmen.

Eigenschaften von Funktionen

- Eine Funktion $f : D \to B$ ist
  - *total*, wenn es für jedes $x \in D$ ein Paar $(x, y) \in f$ gibt;
  - *surjektiv*, wenn es zu jedem $y \in B$ ein Paar $(x, y) \in f$ gibt;
  - *injektiv*, wenn es zu jedem $y \in B$ höchstens ein Paar $(x, y) \in f$ gibt;
  - *bijektiv*, wenn f zugleich surjektiv und injektiv ist.
- Man sagt auch, eine Funktion ist *partiell*, wenn es gleichgültig ist, ob sie total ist oder nicht.
- Vorsicht: Wenn ein Mathematiker nicht angibt, ob eine Funktion total oder partiell ist, meint er in der Regel eine totale Funktion. Für einen Informatiker ist die partielle Funktion der Normalfall.

Stelligkeit von Funktionen

- Funktionen aus $D \to B$ sind n-stellig, wenn der Definitionsbereich D eine Menge von n-Tupeln ist.
- Ist D nicht als kartesisches Produkt strukturiert, so sind die Funktionen aus $D \to B$ 1-stellig, wenn D nicht leer ist, und 0-stellig, wenn D leer ist.
- 0-stellige Funktionen sind *konstante Funktionen*, kurz *Konstanten*, für jeweils einen Wert aus B, d.h. jeder Wert b aus B ($b \in B$) kann als 0-stellige Funktion $b : \emptyset \to B$ aufgefasst werden.
- In diesem Sinne kann man den Ausdruck $1 + 2$ als Verschachtelung von mehreren Funktionen auffassen:
  - $1 : \emptyset \to \mathbb{N}$
  - $2 : \emptyset \to \mathbb{N}$
  - $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$

Nocheinmal: Schreibweisen

- Die Notation $f(x_1, \ldots, x_n)$, die die "Anwendung" von f auf die Argumente $x_1, \ldots, x_n$ beschreibt, nennen wir *Funktionsschreibweise*.
- Bei 2-stelligen Funktionen verwendet man häufig auch die *Infixschreibweise* $x_1 f x_2$.
  Beispiel: $17 + 8$ statt $+(17, 8)$ für die Addition.
- Bei 1-stelligen Funktionen verwendet man gerne die *Präfixschreibweise* $f x_1$.
  Beispiel $\log 13$ statt $\log(13)$ für die Logarithmusfunktion.
- Manchmal ist jedoch auch die *Postfixschreibweise* $x_1 \ldots x_n f$ gebräuchlich.
  Beispiel: $21!$ statt $!(21)$ für die Fakultätsfunktion.
- Bemerkung: In der Mathematik gibt es Schreibweisen, die sich nicht direkt einer dieser Schreibweisen unterordnen lassen, z.B. $\sqrt{x}$ (Quadratwurzel), x^y (Potenzierung), $\frac{x}{y}$ (Division).

Prädikate

- Ein *Prädikat* ist eine Funktion aus $D \to \mathbb{B}$ wobei $\mathbb{B} = \{TRUE, FALSE\}$.
- Prädikate sind also eine Abbildung auf die Menge der booleschen Werte, d.h. der Bildbereich dieser Funktionen ist $\mathbb{B}$.
- Beipiel:
  $=: \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}$ ist ein Prädikat (die Gleichheitsrelation auf $\mathbb{Z}$).
  Es ist z.B.
  $$= (-321, -321) = TRUE$$
  oder anders ausgedrückt
  $$= (-321, -321) \mapsto TRUE$$
- Wir lassen als Schreibweise "$= TRUE$" meist weg und schreiben bei 2-stelligen Prädikaten in Infixschreibweise
  $$-321 = -321.$$

Folgen

- Eine *Folge* $(x_1, \ldots, x_n)$ der Länge n über die Elemente einer Menge M (d.h. $x_i \in M$) ist ein n-Tupel von Werten aus M, d.h.

$$(x_1, \ldots, x_n) \in M^n.$$

- Eine Folge $x \in M^0$ der Länge $n = 0$ wird leere Folge genannt.
- Die Menge aller nicht-leeren Folgen über M wird meist mit

$$M^+ = M^1 \cup M^2 \cup M^3 \cup \ldots$$

  bezeichnet.
- Die Menge aller Folgen (auch leerer) über M ist dann

$$M^* = M^0 \cup M^+.$$

- Die Länge einer Folge x wird auch mit $|x|$ bezeichnet.

Folgen

- Die Konkatenation $\circ$ aus dem vorherigen Kapitel ist damit formal eine Abbildung
$$\circ : M^* \times M^* \rightarrow M^*.$$
- Für eine formale Definition von $\circ$ benötigen wir zunächst noch die *Projektion*, eine Abbildung
$$\pi : M^n \times I_n \rightarrow M.$$
- Die Menge $I_n = \{i | i \in \mathbb{N} \text{ und } 1 \leq i \leq n\}$ heißt *Indexmenge*.
- Die Abbildung π bildet eine Folge $x = (x_1, \ldots, x_n)$ der Länge n und ein i $(1 \leq i \leq n)$ auf die i-te Komponente x_i der Folge ab.
- Beispiel: $x = (4, 5, 6)$

  $\pi(x, 1) = 4$, $\pi(x, 2) = 5$, $\pi(x, 3) = 6$.

Folgen

- Offensichtlich ist auch $x = (\pi(x,1), \ldots, \pi(x,|x|))$ eine alternative Schreibweise für eine Folge x.
- Damit können wir die Konkatenation $\circ$ wie folgt formal definieren:

$$x \circ y = (\pi(x,1), \ldots, \pi(x,|x|), \pi(y,1), \ldots, \pi(y,|y|)),$$

oder anders ausgedrückt:

$$\pi((x \circ y), i) = \begin{cases} \pi(x,i) & \text{für } 1 \leq i \leq |x| \\ \pi(y, i-|x|) & \text{für } |x|+1 \leq i \leq |x|+|y| \end{cases}$$

- Beispiel: Sei $M = \mathbb{N}_0$ und $x = (7,0,3,18), y = (21,3,7)$, dann ist

$$x \circ y = (\pi(x,1), \pi(x,2), \pi(x,3), \pi(x,4), \pi(y,5-4), \pi(y,6-4), \pi(y,7-4))$$

$$= (7,0,3,18,21,3,7).$$

Überblick

Beweisprinzip der “vollständigen Induktion”

Ein fundamentales mathematisches Beweisprinzip ist die *vollständige Induktion*:
Sei $p : \mathbb{N}_0 \to \mathbb{B}$ ein totales Prädikat.
Falls

1. $p(0)$ (*Induktionsanfang*) und
2. für beliebiges $n \in \mathbb{N}_0$ gilt der *Induktionsschluss*:
   “Falls $p(n)$ (*Induktionsvoraussetzung*), dann $p(n+1)$.”

dann: $p(n)$ für alle $n \in \mathbb{N}_0$.
Die vollständige Induktion (“nach n”) ermöglicht es zu beweisen, dass eine Aussage (“p”) für alle $n \in \mathbb{N}_0$ gilt.

Beweisprinzip der “vollständigen Induktion”

Beispiel:

- Sei $p(n) : \Longleftrightarrow \sum_{i=0}^{n} i = \frac{n \cdot (n+1)}{2}$
- Zu beweisen: Gültigkeit von $p(n)$ für alle $n \in \mathbb{N}_0$.
- Induktionsanfang:
  - Zu zeigen, dass $p(0)$, d.h. $\sum_{i=0}^{0} i = \frac{n \cdot (n+1)}{2}$.
  - $\sum_{i=0}^{0} i = 0$
  - $\frac{0 \cdot (0+1)}{2} = 0 \surd$
- Für den Induktionsschluss können wir für $n \in \mathbb{N}_0$ als Induktionsvoraussetzung $p(n)$, d.h. $\sum_{i=0}^{n} i = \frac{n \cdot (n+1)}{2}$ annehmen.
- Zu zeigen ist die Gültigkeit von $p(n+1)$, d.h. $\sum_{i=0}^{n+1} i = \frac{(n+1) \cdot (n+1+1)}{2}$

Beweisprinzip der “vollständigen Induktion”

Beweis (unter Verwendung der Induktionsvoraussetzung):

$$
\begin{aligned}
\sum_{i=0}^{n+1} i &= 0 + 1 + \ldots + n + (n+1) \\
&= \sum_{i=0}^{n} i + (n+1) \\
&= \frac{n \cdot (n+1)}{2} + n + 1 \\
&= \frac{n \cdot (n+1) + 2 \cdot (n+1)}{2} \\
&= \frac{(n+1) \cdot (n+2)}{2} \\
&= \frac{(n+1) \cdot (n+1+1)}{2} \qquad \surd
\end{aligned}
$$

Warum ist dieses Beweisprinzip gültig?

Wie kann man sicher sein, dass p für alle Zahlen aus $\mathbb{N}_0$ gilt, wenn $p(0)$ gilt und man für ein beliebiges festes $n \in \mathbb{N}_0$ von $p(n)$ auf $p(n+1)$ schließen kann?

Induktive Definition der Menge $\mathbb{N}_0$

Die Menge $\mathbb{N}_0$ lässt sich durch folgende Regeln *induktiv definieren*:

1. $0 \in \mathbb{N}_0$
2. Ist $n \in \mathbb{N}_0$, dann ist auch $n + 1 \in \mathbb{N}_0$.
3. Außer den Elementen gemäß Regeln 1 und 2 enthält $\mathbb{N}_0$ keine weiteren Objekte.

Aufbau der Menge $\mathbb{N}_0$

Die Elemente der Menge $\mathbb{N}_0$ werden gemäß dieser induktiven Definition der Reihe nach "konstruiert":

- Zunächst wird 0 gemäß Regel 1 als Element von $\mathbb{N}_0$ festgelegt.
- Wegen Regel 2 ist dann $0 + 1 = 1$ Element von $\mathbb{N}_0$.
- Erneute Anwendung von Regel 2 ergibt $1 + 1 = 2$ als Element von $\mathbb{N}_0$ usw.

"$n + 1$" können wir auch die Nachfolger-Funktion nennen:

$$\begin{aligned} \mathit{successor} \quad &: \quad \mathbb{N}_0 \to \mathbb{N}_0 \\ \text{mit} \quad &\quad n \mapsto n + 1 \end{aligned}$$

Jede Zahl aus $\mathbb{N}_0$ wird erzeugt durch endlich-oft-malige Anwendung von *successor* auf 0, z.B.:

$$3 = \mathit{successor}(\mathit{successor}(\mathit{successor}(0))), \text{ also gilt: } 3 \in \mathbb{N}_0$$

Zusammenhang zwischen “Vollständiger Induktion” und “Induktiver Definition”

- Dieser “induktive Aufbau” der Menge $\mathbb{N}_0$ ist der Grund für die Gültigkeit des Beweisprinzips der vollständigen Induktion.
- Das Prinzip der vollständigen Induktion vollzieht genau diesen Erzeugungsmechanismus der Menge $\mathbb{N}_0$ nach:
  - Der Induktionsanfang verifiziert $p(0)$.
  - Mit dem Induktionsschluss, angewendet auf $n = 0$, erhält man $p(0 + 1)$, d.h. $p(1)$.
  - Mit einem weiteren Induktionsschluss, angewendet auf $n = 1$, erhält man $p(1 + 1)$, d.h. $p(2)$ usw.
- Da $\mathbb{N}_0$ nur Elemente enthält, die gemäß der induktiven Definition von $\mathbb{N}_0$ konstruiert sind, gilt dann also p tatsächlich für alle Zahlen aus $\mathbb{N}_0$.

Rekursive Definition von Abbildungen

- Eine weitere Konsequenz der induktiven Definition von $\mathbb{N}_0$ ist die Ermöglichung der *rekursiven Definition* von Abbildungen von $\mathbb{N}_0$.
- Die rekursive Definition einer Funktion f mit Definitionsbereich $\mathbb{N}_0$ bedeutet intuitiv:
  - $f(0)$ wird explizit festgelegt.
  - $f(n+1)$ für ein beliebiges $n \in \mathbb{N}_0$ wird auf $f(n)$ "zurückgeführt", d.h. in Abhängigkeit von $f(n)$ definiert.
  - Die Werte der Funktion $f(0)$, $f(1)$, $f(2)$ usw. sind dann wie oben erzeugbar, was $f(m)$ für alle $m \in \mathbb{N}_0$ festlegt.

Beispiel: Fakultäts-Funktion

- Die Fakultäts-Funktion $! : \mathbb{N}_0 \to \mathbb{N}_0$ ist rekursiv definiert wie folgt:
  - $0! = 1$
  - $(n+1)! = (n+1) \cdot (n!)$
- Oft wird äquivalent statt der Rückführung von $n+1$ auf n der Fall $n \neq 0$ auf $n-1$ zurückgeführt:

$$n! = \begin{cases} 1, & \text{falls } n = 0, \\ n \cdot (n-1)! & \text{sonst.} \end{cases}$$

Beispiel: Summen-Formel

- Die Summenformel aus dem obigen Beispiel zum Beweis durch vollständige Induktion lässt sich ebenfalls rekursiv definieren:

$$\sum_{i=0}^{n} i = \begin{cases} 0, & \text{falls } n = 0, \\ \sum_{i=0}^{n-1} i + n & \text{sonst.} \end{cases}$$

- Das Beweisprinzip der vollständigen Induktion eignet sich besonders gut, wenn in der zu beweisenden Aussage rekursiv definierte Abbildungen auftreten.

Beispiel: Summen-Formel

- Unabhängig von der Gestalt des Summanden lässt sich eine Summenformel grundsätzlich immer rekursiv definieren.
- Sei $a : \mathbb{N} \to \mathbb{N}$.

$$\sum_{i=0}^{n} a(i) = \begin{cases} 0, & \text{falls } n = 0, \\ \sum_{i=0}^{n-1} a(i) + a(n) & \text{sonst.} \end{cases}$$

Induktive Definition von Folgen

- Folgen haben wir oben als n-Tupel definiert, also als Elemente aus M^*.
- Hierbei gilt offensichtlich, dass eine Folge der Länge 1 $(a) \in M$ mit ihrem einzigen Element $a \in M$ identisch ist.
- Unter Ausnutzung dieser Eigenschaft können wir Folgen auch induktiv definieren.
- Hilfsfunktionen hierzu ermöglichen das Anfügen eines Elementes $a \in M$ an eine Folge $x \in M^*$:

$$postfix : M^* \times M \rightarrow M^*$$

$$\text{mit } postfix(x, a) = x \circ (a)$$

- oder analog das Anfügen einer Folge $x \in M^*$ an ein Element $a \in M$:

$$prefix : M \times M^* \rightarrow M^*$$

$$\text{mit } prefix(a, x) = (a) \circ x$$

Induktive Definition von Folgen

- Damit kann eine Folge $x \in M^*, x = (x_1, \ldots, x_n)$ schrittweise aufgebaut werden.
- Ausgehend von der leeren Folge $()$ werden die Elemente x_1, x_2, ..., x_n angefügt:

$$\begin{aligned} x &= \mathit{postfix}(\ldots \mathit{postfix}(\mathit{postfix}((), x_1), x_2), \ldots, x_n) \\ &= () \circ (x_1) \circ (x_2) \circ \ldots \circ (x_n) \end{aligned}$$

Induktive Definition von Folgen

Induktive Definition von M^*:

1. $() \in M^*$
2. Ist $x \in M^*$ und $a \in M$, dann ist $\mathit{postfix}(x, a) \in M^*$.

Analoge Definition unter Verwendung von *prefix*:

1. $() \in M^*$
2. Ist $a \in M$ und $x \in M^*$, dann ist $\mathit{prefix}(a, x) \in M^*$.

Rekursive Funktionen über Folgen

- Da nun Folgen induktiv definiert sind, liegt es nahe, dass wir sehr einfach rekursive Abbildungen über Folgen definieren können.
- $first : M^+ \rightarrow M$ mit $first(prefix(a, x)) = a$
- $rest : M^+ \rightarrow M^*$ mit $rest(prefix(a, x)) = x$
- Die Bedeutung dieser Funktionen ist offensichtlich. Für eine nicht leere Folge $(x_1, \ldots, x_n) \neq ()$ gilt:

$$first(x_1, x_2, \ldots, x_n) = x_1$$

$$rest(x_1, x_2, \ldots, x_n) = (x_2, \ldots, x_n)$$

Rekursive Funktionen über Folgen

Die *Projektion* auf Folgen

$$\pi : M^n \times I_n \to M.$$

mit $\pi(x, i) = x_i$ können wir nun auch rekursiv definieren, z.B.:

$$\pi(x,i) = \begin{cases} first(x), & \text{falls } i = 1, \\ \pi(rest(x), i-1) & \text{sonst.} \end{cases}$$

Verallgemeinerung

- Die induktive Struktur von Folgen lässt sich leicht verallgemeinern.
- Jede nicht-leere Folge ist zusammengesetzt aus einem Element $a \in M$ und einer anderen Folge x:

$$y = prefix(a, x)$$

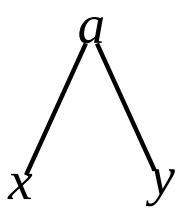
- Abstrahiert von der konkreten Funktion *prefix*, ist y durch das Paar (a, x) bestimmt, wobei x selbst wieder derartig bestimmt ist.
- Eine Verallgemeinerung kann darin bestehen, dass wir Objekte einführen, die aus einem Element a und mehreren "Resten" bestehen:

$$(a, x, y)$$

- Hierbei gilt induktiv, dass die "Reste" x und y selbst von der gleichen Art sind.

Binärbäume

- Solche Objekte heißen *Binärbäume* (über M).
- Analog zu den Folgen lassen wir den *leeren Baum* zu, den wir mit ε bezeichnen.
- Induktive Definition der Menge $binarytree_M$ der Binärbäume über M:
  1. $\varepsilon \in binarytree_M$
  2. Wenn $a \in M$ und $x, y \in binarytree_M$, dann ist $(a, x, y) \in binarytree_M$.
- Hierbei heißt a *Wurzel*, x *linker Teilbaum*, y *rechter Teilbaum* eines Binärbaumes (a, x, y).
- Ein Binärbaum der Gestalt $(a, \varepsilon, \varepsilon)$ heißt *Blatt*.
- Ein von ε verschiedener Binärbaum heißt *nicht-leer*.
- Es ist auch üblich, Bäume graphisch darzustellen. Die kanonische Darstellung für (a, x, y) ist:

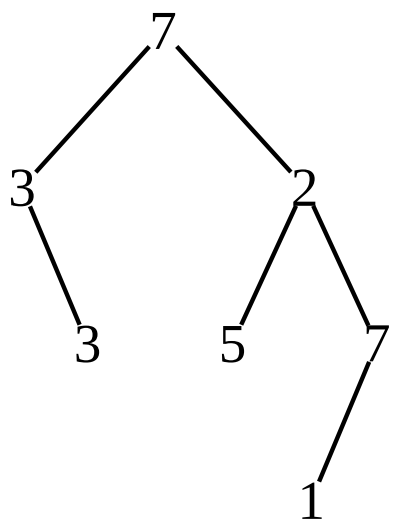


Binärbäume

- Beispiel: Das Objekt

$$(7, (3, \varepsilon, (3, \varepsilon, \varepsilon)), (2, (5, \varepsilon, \varepsilon), (7, (1, \varepsilon, \varepsilon), \varepsilon)))$$

  ist ein Binärbaum über $\mathbb{N}_0$.
- Graphisch:



- Die Wurzel des Baumes und die Wurzeln von Teilbäumen (linker bzw. rechter Unterbaum) sind *Knoten*.

Rekursive Funktionen über Binärbäumen

- Entsprechend der induktiven Definition lassen sich auch leicht wieder rekursive Funktionen über Binärbäumen definieren, die auf die einzelnen Elemente zugreifen:
- $root : binarytree_M \setminus \{\varepsilon\} \to M$
  mit $root(a, x, y) = a$
- $left : binarytree_M \setminus \{\varepsilon\} \to binarytree_M$
  mit $left(a, x, y) = x$
- $right : binarytree_M \setminus \{\varepsilon\} \to binarytree_M$
  mit $right(a, x, y) = y$

Rekursive Funktionen über Binärbäumen

Damit können wir z.B. die Anzahl der Knoten bestimmen:

$$nodes : binarytree_M \to \mathbb{N}_0$$

$$nodes(z) = \begin{cases} 0, & \text{falls } z = \varepsilon, \\ 1 + nodes(left(z)) + nodes(right(z)) & \text{sonst.} \end{cases}$$

Schlussbemerkung

- Ein Binärbaum besteht letztlich aus der Multimenge seiner Knoten.
- Auch Folgen bestehen aus der Multimenge ihrer Elemente.
- In beiden Fällen sind die Multimengen in bestimmter Weise angeordnet. Dadurch enthalten sowohl Folgen als auch Bäume mehr Information als die Multimengen ihrer Elemente bzw. Knoten.
  - Bei Folgen sind die Elemente *linear* angeordnet.
  - Bäume beschreiben eine *verzweigte* Struktur der Elemente der Multimenge der Knoten.

Überblick

Ausdrücke

- Betrachten wir folgende Zeichenketten:
  - $a + 5$
  - blau & gelb
- Diese Zeichenketten können wir als Folgen von Symbolen begreifen.
- Eine solche Folge von Symbolen heißt auch *Ausdruck* oder *Term*.
- Das Konzept der Ausdrücke bildet einen Grundbaustein der meisten höheren Programmiersprachen, daher schauen wir uns dieses Konzept im Folgenden etwas genauer an.

Struktur und Bedeutung

- Für einen Ausdruck können wir Regeln aufstellen. Intuitiv ist klar, dass etwa die Symbolfolge “5+” nicht korrekt ist, “$5 + 2$” oder “$a + 5$” aber schon. Es gibt also offenbar eine *Struktur*, die den Aufbau von korrekten Symbolfolgen (Ausdrücken) beschreibt.
- Außerdem hat eine Folge von Symbolen (ein Ausdruck) eine *Bedeutung* (vorausgesetzt, die verwendeten Symbole haben ihrerseits auch eine Bedeutung). Andernfalls könnte man sagen, “den Ausdruck verstehe ich nicht”.
- Die Struktur von korrekten Ausdrücken können wir auch “Syntax” oder “Grammatik” nennen, die Bedeutung “Semantik”.
- Wir können eine korrekte Struktur beschreiben oder überprüfen, ohne etwas über die Bedeutung zu wissen.

Sorten von Ausdrücken

- Offenbar beschreibt der Ausdruck "$a + 5$" die Addition einer Variablen mit einer Konstanten, der Ausdruck "blau & gelb" ein Farbmischung.
- Die beiden Ausdrücke gehören damit zu unterschiedlichen *Sorten*.
- Wir könnten z.B. festlegen:
  - "$a + 5$" ist ein Ausdruck der Sorte $\mathbb{N}_0$.
  - "blau & gelb" ist ein Ausdruck der Sorte "Farbe".
- Die Bezeichnung "Sorte" anstelle von "Wertebereich" macht deutlich, dass wir zunächst nicht an den konkreten Werten interessiert sind.
- Die Werte sind erst interessant, wenn es um die Bedeutung (Semantik) der Ausdrücke geht.

Struktur von Ausdrücken

- Ausdrücke werden zusammengesetzt aus *Operatoren* und *Variablen*.
- Operatoren bezeichnen Funktionen und haben daher wie diese eine Stelligkeit.
- Wie bei Funktionen stehen 0-stellige Operatoren für Konstanten.
- Variablen sind Namen für Ausdrücke. Jede Variable hat eine Sorte.

Operatoren

- Ein 1-stelliger Operator wird auf einen Ausdruck angewendet.
- Ein *n*-stelliger Operator wird auf ein *n*-Tupel von Ausdrücken angewendet.
- Die *n* Komponenten des *n*-Tupels heißen *Operanden* des Operators, der auf das *n*-Tupel angewendet wird.
- Ein Operator bildet einen Ausdruck einer Sorte, die dem Bildbereich der vom ihm bezeichneten Funktion entspricht.

Einige Operator-Beschreibungen

$$
\begin{array}{rcl}
+ & : & \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0 \\
- & : & \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0 \\
= & : & \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{B} \\
> & : & \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{B} \\
< & : & \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{B} \\
\wedge & : & \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\
\vee & : & \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\
\neg & : & \mathbb{B} \to \mathbb{B} \\
\mathit{TRUE} & : & \emptyset \to \mathbb{B} \\
\mathit{FALSE} & : & \emptyset \to \mathbb{B} \\
0 & : & \emptyset \to \mathbb{N}_0 \\
1 & : & \emptyset \to \mathbb{N}_0
\end{array}
$$

Operatoren und Sorten

- Eine Operatorbeschreibung enthält ein Operatorsymbol und gibt dazu die Signatur der vom Operatorsymbol bezeichneten Funktion an, d.h. die Sorten der Operanden und die Sorte des vom Operator gebildeten Ausdrucks.
- In den Operatorbeschreibungen auf der vorherigen Folie kommen die Sorten $\mathbb{N}_0$ und $\mathbb{B}$ vor.
- Der Operator $<$ verknüpft zwei Ausdrücke der Sorte $\mathbb{N}_0$ (seine Operanden) und bildet einen Ausdruck der Sorte $\mathbb{B}$.
- Der 0-stellige Operator 0 ist ein Ausdruck der Sorte $\mathbb{N}_0$ und hat keine Operanden.

Induktive Definition von Ausdrücken

Sei gegeben

- eine Menge von Sorten S,
- eine Menge von Operator-Beschreibungen F und
- eine Menge von Variablen V, die verschieden sind von allen Operator-Symbolen in F.
  Es gilt: $V = \bigcup_{S_i \in S} V_{S_i}$, wobei V_{S_i} die Menge aller Variablen der Sorte $S_i \in S$ bezeichnet.

Die Menge der Ausdrücke ist dann wie folgt induktiv definiert:

- Eine Variable $v \in V$ ist ein Ausdruck.
- Das Symbol op eines 0-stelligen Operators aus F ist ein Ausdruck.
- Sind $a_1, \ldots, a_n$ Ausdrücke der Sorten $S_1, \ldots, S_n$ und $op \in F$ ein Operator der Signatur $S_1 \times \ldots \times S_n \rightarrow S_0$ (wobei $S_0, \ldots, S_n \in S$), dann ist $op(a_1, \ldots, a_n)$ ein Ausdruck der Sorte S_0.

Schreibweisen

Als Schreibweisen für die Anwendung eines mehrstelligen Operators *op* gibt es wie für Funktionen die

- Funktionsform als $op(a_1, \ldots, a_n)$, wobei auch die a_i in Funktionsform geschrieben sind.
- Präfixform als $op\, a_1 \ldots a_n$, wobei auch die a_i in Präfixform geschrieben sind.
- Postfixform als $a_1 \ldots a_n op$, wobei auch die a_i in Postfixform geschrieben sind.
- Infixform als $(op\, a_1)$ für einstellige Operatoren *op*, $(a_1 op\, a_2)$ für zweistellige Operatoren *op*, wobei auch die a_i in Infixform geschrieben sind. Bei höherer Stelligkeit als 2 kann der Operator *op* auch aus mehreren Teilen $op_1 \ldots op_{n-1}$ bestehen: $(a_1 op_1 a_2 \ldots op_{n-1} a_n)$. Wenn die Zuordnung der Operanden zu den Operatoren eindeutig ist, kann die Klammerung entfallen. Die hier definierte Form heißt *vollständig geklammert*.

Schreibweisen

Die Funktion f_2 aus Übungsaufgabe 2-1 ist bestimmt durch den Ausdruck in Infixform "$(x \wedge z) \vee (y \wedge z)$". Hierbei sind x, y, z Variablen, $\vee$ und $\wedge$ Operatoren (aus der obigen Menge von Operator-Beschreibungen). Die äußeren Klammern sind weggelassen, da die Zuordnung der Teilausdrücke zu den Operatoren eindeutig ist. Übersetzt in die anderen Schreibweisen lautet der Ausdruck:

- Präfixform: $\vee \wedge x\, z \wedge y\, z$
- Postfixform: $x\, z \wedge y\, z \wedge \vee$
- Funktionsform: $\vee(\wedge(x, z), \wedge(y, z))$

Bindung

- Während man in der Präfixform und in der Postfixform auf Klammern verzichten kann (Warum eigentlich?), ist Klammerung in der Infixform grundsätzlich nötig, um Operanden ihren Operatoren eindeutig zuzuordnen.
- Lassen wir die Klammern im Ausdruck "$(x \wedge z) \vee (y \wedge z)$" weg, so erhalten wir

$$x \wedge z \vee y \wedge z$$

- Nun wäre auch eine völlig andere Bindung möglich, z.B.

$$x \wedge ((z \vee y) \wedge z)$$

Bindungsstärke (Präzedenz)

- Eindeutigkeit ohne Klammerung kann man in der Infixform erreichen, indem man den Operatoren unterschiedliche *Bindungsstärken* (Präzedenzen) zuweist.
- Beispielsweise hat unter den logischen Operatoren $\neg$ höhere Präzedenz als $\wedge$, $\wedge$ hat höhere Präzedenz als $\vee$.
- Damit erhalten wir auch für den Ausdruck

$$x \wedge z \vee y \wedge z$$

die ursprünglich gewünschte, nun *implizite* Klammerung:

$$(x \wedge z) \vee (y \wedge z)$$

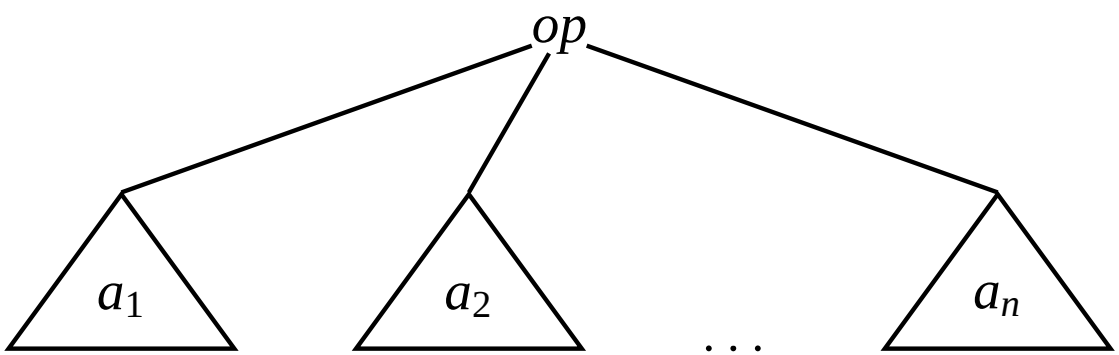
Links-/Rechts-Assoziativität

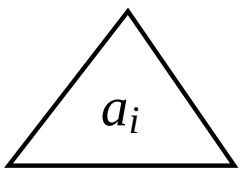
- Wenn Operatoren gleicher Bindungsstärke konkurrieren, muss außerdem entschieden werden, ob der linke oder der rechte “gewinnt”.
- Man legt hierzu fest, ob die Operanden eines Operators *linksassoziativ* oder *rechtsassoziativ* binden.
- Beispiel: Der Ausdruck “$x \vee y \vee z$” bedeutet
  - linksassoziativ:
    $$(x \vee y) \vee z$$
  - rechtsassoziativ:
    $$x \vee (y \vee z)$$
- Die linksassoziative Bindung ist gebräuchlicher.
- Natürlich ist das Setzen von Klammern dennoch erlaubt und sogar sinnvoll, um anzuzeigen, dass die implizite Klammerung der erwünschten Klammerung entspricht.

Baumdarstellung

Eine weitere wichtige Art der Darstellung von Ausdrücken mit eindeutiger Zuordnung von Operanden zu Operatoren (durch implizite Klammerung) ist die *Baumdarstellung von Ausdrücken*:

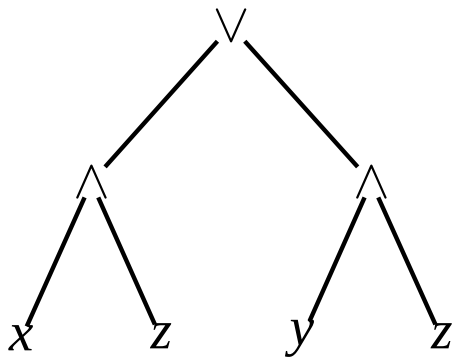
- Eine Variable *v* wird durch den Knoten *v* dargestellt.
- Ein *n*-stelliger Operator *op* wird durch den Knoten *op* mit den *n* Operanden als Teilbäumen dargestellt.



- Hierbei ist  die Baumdarstellung des Ausdrucks a_i.

Beispiel

Die Baumdarstellung des Ausdrucks "$(x \wedge z) \vee (y \wedge z)$" ist demnach:



Baumdurchlauf

- Da die Baumdarstellung (wie ja auch die Ausdrücke selbst) induktiv defniert ist, lassen sich auch naheliegende Funktionen leicht rekursiv definieren, die die Baumdarstellung auf die Präfixform, Postfixform bzw. Infixform abbilden.
- Man durchläuft dazu den Baum links-abwärts:
  1. besuche den Wurzelknoten
  2. durchlaufe jeden Unterbaum links-abwärts
  3. besuche dann wieder den Wurzelknoten

Baumdurchlauf

- Man erhält die Präfixform, wenn man den Knotennamen beim ersten Besuch eines Knotens ausgibt.
- Man erhält die Postfixform, wenn man den Knotennamen beim letzten Besuch eines Knotens ausgibt.
- Man erhält die Infixform, wenn man beim ersten Besuch die öffnende, beim letzten die schließende Klammer ausgibt und bei den dazwischenliegenden Besuchen die Operatorteile.
- Bemerkung: Dieses Verfahren verdeutlicht, dass die verschiedenen Schreibweisen äquivalent sind.

Überladung von Operatoren

Betrachten Sie nun folgende Menge F von Operatorbeschreibungen:

$$
\begin{array}{rcl}
+ & : & \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0 \\
0 & : & \emptyset \to \mathbb{N}_0 \\
1 & : & \emptyset \to \mathbb{N}_0 \\
+ & : & \mathbb{R} \times \mathbb{R} \to \mathbb{R} \\
0.0 & : & \emptyset \to \mathbb{R} \\
1.0 & : & \emptyset \to \mathbb{R}
\end{array}
$$

- "$0 + 1$" ist ein gültiger Ausdruck.
- "$0.0 + 1.0$" ist ein gültiger Ausdruck.
- "$0.0 + 1$" ist kein gültiger Ausdruck.

Überladen

- Das Operatorsymbol “+” kommt in F zweimal vor, erhält aber jeweils eine unterschiedliche Beschreibung (Signatur).
- Das Operatorsymbol “+” beschreibt also in F zwei unterschiedliche Funktionen.
- Man sagt: Das Operatorsymbol “+” ist *überladen*.
- Durch die Erfüllbarkeit einer Signatur kann man entscheiden, welcher Operator tatsächlich zu verwenden ist.
- Der Ausdruck “$0.0 + 1$” erfüllt in F keine Signatur.

Bedeutung

- Um die *Bedeutung* (Semantik) eines Ausdrucks festzulegen, müssen einerseits die Funktionen, die von den Operatoren bezeichnet werden, definiert sein.
- Andererseits müssen alle Variablen ersetzt werden können durch den von ihnen bezeichneten Ausdruck.
- Dann können alle im Ausdruck vorkommenden Operationen ausgeführt werden.
- Schließlich kann man für einen Ausdruck einen Wert einsetzen.

Bedeutung

Beispiel: Um den Ausdruck “$2 + 5$” zu interpretieren, vereinbaren wir:

- Der Operator 2 steht für die Zahl $2 \in \mathbb{N}_0$.
- Der Operator 5 steht für die Zahl $5 \in \mathbb{N}_0$.
- Der Operator $+$ steht für die arithmetische Addition zweier natürlicher Zahlen.

Durch Anwendung der vereinbarten Funktionen erhalten wir als Wert des Ausdrucks die Zahl $7 \in \mathbb{N}_0$.

Substitution

- Um Variablen durch die von ihnen bezeichneten Ausdrücke zu ersetzen, muß wiederum die Bedeutung einer Variablen vereinbart werden.
- Aus einer Liste von vereinbarten Bedeutungen von Variablen kann man dann die Ersetzung (*Substitution*) der Variablen in einem Ausdruck vornehmen.
- Als einfache Substitution kann man ein Paar σ aus einer Variablen und einem Ausdruck $\sigma = [v/a]$ ansehen, wobei v und a zur selben Sorte gehören müssen.
- Festzustellen, ob zwei Ausdrücke zur selben Sorte gehören, ist nicht trivial und wird unter dem Thema “Unifikation” studiert.

Anwendung einer Substitution

Die Anwendung einer Substitution $\sigma = [v/t]$ auf einen Ausdruck u wird geschrieben

$$u\sigma \text{ bzw. } u[v/t].$$

Das Ergebnis dieser Anwendung ist ein Ausdruck, der durch einen der folgenden drei Fälle bestimmt ist:

1. $u[v/t] = t$, falls u die zu ersetzende Variable v ist,
2. $u[v/t] = u$, falls u ein 0-stelliger Operator oder eine andere Variable als v ist,
3. $u[v/t] = f(u_1[v/t], u_2[v/t], \ldots, u_n[v/t]))$, falls $u = f(u_1, u_2, \ldots, u_n)$.

Anwendung einer Substitution

Beispiel:
Sei $\sigma = [x/2 * b]$, angewendet auf den Ausdruck "$x + x - 1$". In Funktionsform notiert:

$$\begin{aligned} -(+(x,x),1)[x/*(2,b)] &= -(+(x,x)[x/*(2,b)],1[x/*(2,b)]) \\ &= -(+(x,x)[x/*(2,b)],1) \\ &= -(+(x[x/*(2,b)],x[x/*(2,b)]),1) \\ &= -(+(*(2,b),*(2,b)),1) \end{aligned}$$

Bemerkungen

- Ausdrücke stellen ein klassisches funktionales Konzept dar:
  Der Ausdruck $3 + 5$ hat offenbar den Wert $8 \in \mathbb{N}_0$.
- Die Funktionen, die durch die Operatoren 3, $+$ und 5 bezeichnet werden, sind in einer vollständigen Semantik definiert.
- Wie der Wert dieses Ausdrucks auf einem Rechner konkret berechnet wird, ist typischerweise aber nicht näher spezifiziert.
- Die formalere Beschäftigung mit Ausdrücken sowie die hier nur skizzierten Problembereiche
  - Sorten und Überladung,
  - Definition und Eigenschaften rekursiver Funktionen z.B. für Baumdurchläufe,
  - Substitution und Unifikation

  bilden einen wichtigen Inhalt der Vorlesung “Programmierung und Modellierung”.

Bemerkungen

- Ausdrücke bilden einen wichtigen Grundbaustein in den meisten höheren Programmiersprachen, funktionalen wie imperativen, auch in Java.
- Die meisten höheren Programmiersprachen verwenden funktionale *und* imperative Konzepte. Java ist – neben der Möglichkeit, Daten objektorientiert zu modellieren – insbesondere als imperative Sprache angelegt, d.h. Algorithmen spezifizieren die Abfolge von Aktionen statt den funktionalen Zusammenhang zwischen Eingabe und Ausgabe. Wir werden aber sehen, dass Ausdrücke ein wichtiger Bestandteil von Anweisungen sind.
- Die Schreibweise, die wir hier für Ausdrücke kennengelernt haben, ist sowohl menschenlesbar als auch maschinenlesbar (obwohl beide Interpretationsprozesse z.T. sehr unterschiedlich ablaufen).
- Dasselbe ist grundsätzlich für Programme zu fordern: Ein Programm muß maschinenlesbar sein (also der vom Compiler zu überprüfenden Syntax folgen) als auch menschenlesbar, da es für Menschen auf die korrekte Semantik überprüfbar sein muss.

Teil I

Konzepte imperativer Programmierung

Abschnitt 4: Imperative Programmierung

4. Imperative Programmierung

Vorbemerkung

- Induktion, Rekursion und das Konzept der (induktiv definierten) Ausdrücke sind funktionale Konzepte und werden als solche in der Einführungsvorlesung “Programmierung und Modellierung” vertieft.
- Allerdings spielen diese (und andere funktionale) Konzepte auch im imperativen Programmierparadigma eine wichtige Rolle.
- Auch als Entwurfsmuster spielt Rekursion in imperativen Sprachen eine wichtige Rolle.
- Wir wenden uns nun aber dem imperativen Programmierparadigma und der Realisierung am Beispiel von Java zu.

Überblick

Grunddatentypen

- Typischerweise stellt jede höhere Programmiersprache gewisse *Grunddatentypen* zur Verfügung.
- Rein konzeptionell handelt es sich dabei um eine Menge von Sorten.
- Zusätzlich zu diesen Grunddatentypen werden auch gewisse *Grundoperationen* bereitgestellt, also eine Menge von (teilweise überladenen) Operationssymbolen, die auf die bereitgestellten Datentypen (Sorten) angewandt werden können.
- Die Semantik dieser Operationen ist durch den zugrundeliegenden Algorithmus zur Berechnung der entsprechenden Funktion definiert.
- Aus den Literalen der Grunddatentypen und den zugehörigen Grundoperationen können nun, wie im vorherigen Abschnitt definiert, Ausdrücke gebildet werden.
- Achtung: Diese Ausdrücke enthalten zunächst noch keine Variablen. Zu Variablen kommen wir später.

Grunddatentypen (primitive Typen) in Java

- Wir wissen bereits, dass es in Java Grunddatentypen (auch *atomare* oder *primitive Typen*) für $\mathbb{B}$, *CHAR*, eine Teilmenge von $\mathbb{Z}$ und für eine Teilmenge von $\mathbb{R}$ gibt, aber keinen eigenen Grunddatentyp für $\mathbb{N}$.
- Die Werte der einzelnen primitiven Typen werden intern binär dargestellt.
- Die Datentypen unterscheiden sich u.a. in der Anzahl der Bits, die für ihre Darstellung verwendet werden.
- Die Anzahl der Bits, die für die Darstellung der Werte eines primitiven Datentyps verwendet werden, heißt *Länge* des Typs.
- Die Länge eines Datentyps hat Einfluss auf den Wertebereich des Typs.
- Als *objektorientierte* Sprache bietet Java zusätzlich die Möglichkeit, benutzerdefinierte (sog. *abstrakte*) Datentypen zu definieren. Diese Möglichkeiten lernen wir im Teil über objektorientierte Modellierung genauer kennen.

Grunddatentypen (primitive Typen) in Java

Überblick

Typname	Länge	Wertebereich
`boolean`	1 Byte	Wahrheitswerte {`true`,`false`}
`char`	2 Byte	Alle Unicode-Zeichen
`byte`	1 Byte	Ganze Zahlen von -2^7 bis $2^7 - 1$
`short`	2 Byte	Ganze Zahlen von -2^{15} bis $2^{15} - 1$
`int`	4 Byte	Ganze Zahlen von -2^{31} bis $2^{31} - 1$
`long`	8 Byte	Ganze Zahlen von -2^{63} bis $2^{63} - 1$
`float`	4 Byte	Gleitkommazahlen (einfache Genauigkeit)
`double`	8 Byte	Gleitkommazahlen (doppelte Genauigkeit)

Wahrheitswerte in Java

- Java hat einen eigenen Typ **`boolean`** für Wahrheitswerte.
- Die beiden einzigen Werte (*Konstanten*) sind **`true`** für “wahr” und **`false`** für “falsch”.

Boole'sche Operatoren in Java

Operator	Bezeichnung	Bedeutung
`!`	logisches NICHT (¬)	`!a` ergibt **`false`** wenn `a` wahr ist, sonst **`true`**.
`&`	logisches UND (∧)	`a & b` ergibt **`true`**, wenn sowohl `a` als auch `b` wahr ist. Beide Teilausdrücke (`a` und `b`) werden ausgewertet.
`&&`	sequentielles UND	`a && b` ergibt **`true`**, wenn sowohl `a` als auch `b` wahr ist. Ist `a` bereits falsch, wird **`false`** zurückgegeben und `b` nicht mehr ausgewertet.
`|`	logisches ODER (∨)	`a | b` ergibt **`true`**, wenn mindestens einer der beiden Ausdrücke `a` oder `b` wahr ist. Beide Teilausdrücke (`a` und `b`) werden ausgewertet.
`||`	sequentielles ODER	`a || b` ergibt **`true`**, wenn mindestens einer der beiden Ausdrücke `a` oder `b` wahr ist. Ist bereits `a` wahr, so wird **`true`** zurückgegeben und `b` nicht mehr ausgewertet.
`^`	exklusives ODER (XOR)	`a ^ b` ergibt **`true`**, wenn beide Ausdrücke `a` und `b` einen unterschiedlichen Wahrheitswert haben.

Zeichen (Character) in Java

- Java hat einen eigenen Typ **`char`** für (Unicode-)Zeichen.
- Werte (*Konstanten*) werden in einfachen Hochkommata gesetzt, z.B. `'A'` für das Zeichen “A”.
- Einige Sonderzeichen können mit Hilfe von *Standard-Escape-Sequenzen* dargestellt werden:

Sequenz	Bedeutung
`\b`	Backspace (Rückschritt)
`\t`	Tabulator (horizontal)
`\n`	Newline (Zeilenumbruch)
`\f`	Seitenumbruch (Formfeed)
`\r`	Wagenrücklauf (Carriage return)
`\"`	doppeltes Anführungszeichen
`\'`	einfaches Anführungszeichen
`\\`	Backslash

Ganze Zahlen in Java

- Java hat vier Datentypen für ganze (vorzeichenbehaftete) Zahlen: **`byte`** (Länge: 8 Bit), **`short`** (Länge: 16 Bit), **`int`** (Länge: 32 Bit) und **`long`** (Länge: 64 Bit).
- Werte (*Konstanten*) können geschrieben werden in
  - Dezimalform: bestehen aus den Ziffern `0`, . . ., `9`
  - Oktalform: beginnen mit dem Präfix `0` und bestehen aus Ziffern `0`, . . ., `7`
  - Hexadezimalform: beginnen mit dem Präfix `0x` und bestehen aus Ziffern `0`, . . ., `9` und den Buchstaben `a`, . . ., `f` (bzw. `A`, . . ., `F`)
- Negative Zahlen erhalten ein vorangestelltes `-`.
- Gehört dieses `-` zum Literal?

Vorzeichen-Operatoren in Java

- Vorzeichen-Operatoren haben die Signatur

$$S \to S$$

  mit $S \in \{$**`byte`**, **`short`**, **`int`**, ... $\}$.

- Operationen:

Operator	Bezeichnung	Bedeutung
`+`	Positives Vorzeichen	`+n` ist gleichbedeutend mit `n`
`-`	Negatives Vorzeichen	`-n` kehrt das Vorzeichen von `n` um

- Bemerkung: Offenbar sind die Operatoren überladen!!!

Gleitkommazahlen in Java

- Java hat zwei Datentypen für Fließkommazahlen: **`float`** (Länge: 32 Bit) und **`double`** (Länge: 64 Bit).
- Werte (*Konstanten*) werden immer in Dezimalnotation geschrieben und bestehen maximal aus
  - Vorkommateil
  - Dezimalpunkt (*)
  - Nachkommateil
  - Exponent `e` oder `E` (Präfix `-` möglich) (*)
    Gehört dieses `-` zum Literal?
  - Suffix `f` oder `F` (**`float`**) oder `d` oder `D` (**`double`**) (*)

  wobei mindestens einer der mit (*) gekennzeichneten Bestandteile vorhanden sein muss.
- Negative Zahlen erhalten ein vorangestelltes `-`.
- Beispiele:
  - double: `6.23`, `623E-2`, `62.3e-1`
  - float: `6.23f`, `623E-2F`, `62.2e-1f`

Arithmetische Operationen in Java

- Arithmetische Operationen haben die Signatur

$$S \times S \to S$$

  mit $S \in \{$**`byte`**, **`short`**, **`int`**, ... $\}$.

- Operationen:

Operator	Bezeichnung	Bedeutung
`+`	Summe	`a+b` ergibt die Summe von `a` und `b`
`-`	Differenz	`a-b` ergibt die Differenz von `a` und `b`
`*`	Produkt	`a*b` ergibt das Produkt aus `a` und `b`
`/`	Quotient	`a/b` ergibt den Quotienten von `a` und `b`
`%`	Modulo	`a%b` ergibt den Rest der ganzzahligen Division von a durch b

- Bemerkung: Offenbar sind die Operatoren überladen!!!

Vergleichsoperationen in Java

- Vergleichsoperatoren haben die Signatur

$$S \times S \rightarrow \texttt{boolean}$$

  mit $S \in \{\texttt{byte}, \texttt{short}, \texttt{int}, \ldots\}$.
- Operationen:

Operator	Bezeichnung	Bedeutung
`==`	Gleich	`a==b` ergibt **`true`**, wenn `a` gleich `b` ist
`!=`	Ungleich	`a!=b` ergibt **`true`**, wenn `a` ungleich `b` ist
`<`	Kleiner	`a<b` ergibt **`true`**, wenn `a` kleiner `b` ist
`<=`	Kleiner gleich	`a<=b` ergibt **`true`**, wenn `a` kleiner oder gleich `b` ist
`>`	Größer	`a>b` ergibt **`true`**, wenn `a` größer `b` ist
`>=`	Größer gleich	`a>=b` ergibt **`true`**, wenn `a` größer oder gleich `b` ist

- Bemerkung: Offenbar sind die Operatoren ebenfalls überladen!!!

Ausdrücke in Java

- Wir können auch in Java aus Operatoren Ausdrücke (zunächst ohne Variablen) bilden, so wie im vorherigen Kapitel besprochen.
- Laut induktiver Definition von Ausdrücken ist eine Konstante ein Ausdruck.
- Interessanter ist, aus Konstanten (z.B. den **`int`** Werten `6` und `8`) und mehrstelligen Operatoren (z.B. `+`, `*`, `<`, `&&`) Ausdrücke zu bilden. Ein gültiger Ausdruck hat selbst wieder einen Wert (der über die Semantik der beteiligten Operationen definiert ist) und einen Typ (der durch die Ergebnissorte des angewendeten Operators definiert ist):
  - `6 + 8 //Wert: 14 vom Typ int`
  - `6 * 8 //Wert: 48 vom Typ int`
  - `6 < 8 //Wert: true vom Typ boolean`
  - `6 && 8 //ungültiger Ausdruck`

  Warum?

Typkonversion

- Motivation:
  - Was passiert eigentlich, wenn man verschiedene Sorten/Typen miteinander verarbeiten will?
  - Ist der Java-Ausdruck `6 + 7.3` erlaubt?
  - Eigentlich nicht: Die Operation $+$ ist zwar überladen, d.h.

    $$+ : S \times S \to S$$

    ist definiert für beliebige primitive Datentypen S, aber es gibt keine Operation

    $$+ : S \times T \to U$$

    für *verschiedene* primitive Datentypen $S \neq T$.
- Trotzdem ist der Ausdruck `6 + 7.3` in Java erlaubt.
- *Wann* passiert *was* und *wie* bei der Auswertung des Ausdrucks `6 + 7.3`?

Automatische Typkonversion in Java

- Wann:
  Während des Übersetzens des Programmcodes durch den Compiler.
- Was:
  Der Compiler kann dem Ausdruck keinen Typ (und damit auch keinen Wert) zuweisen. Solange kein *Informationsverlust* auftritt, versucht der Compiler diese Situation zu retten.
- Wie:
  Der Compiler konvertiert automatisch den Ausdruck `6` vom Typ **`int`** in einen Ausdruck vom Typ **`double`**, so dass die Operation

$$+ : \texttt{double} \times \texttt{double} \rightarrow \texttt{double}$$

  angewendet werden kann.

Automatische Typkonversion in Java

- Formal gesehen ist diese Konvertierung eine Operation `i->d` mit der Signatur

  $$\texttt{i->d} : \textbf{int} \rightarrow \textbf{double}$$

  d.h. der Compiler wandelt den Ausdruck
  `6 + 7.3`
  um in den Ausdruck
  `i->d(6) + 7.3`
- Dieser Ausdruck hat offensichtlich einen eindeutigen Typ und damit auch einen eindeutig definierten Wert.

Typkonversion in Java

- Was bedeuted “Informationsverlust”?
- Es gilt folgende “Kleiner-Beziehung” zwischen Datentypen:

```
byte < short < int < long < float < double
```

- Beispiele:
  - `1 + 1.7` ist vom Typ **`double`**
  - `1.0f + 2` ist vom Typ **`float`**
  - `1.0f + 2.0` ist vom Typ **`double`**
- Java konvertiert Ausdrücke automatisch in den allgemeineren (“größeren”) Typ, da dabei kein Informationsverlust auftritt.
  Warum?

Explizite Typkonversion in Java: Type-Casting

- Will man eine Typkonversion zum spezielleren Typ durchführen, so muss man dies in Java explizit angeben.
- Dies nennt man allgemein *Type-Casting*.
- In Java erzwingt man die Typkonversion zum spezielleren Typ `type` durch Voranstellen von `(type)`.
- Der Ausdruck
  `(type) a`
  wandelt den Ausdruck `a` in einen Ausdruck vom Typ `type` um.
- Beispiele:
  - `(byte)3` ist vom Typ **`byte`**
  - `(int)(2.0 + 5.0)` ist vom Typ **`int`**
  - `(float)1.3e-7` ist vom Typ **`float`**

Explizite Typkonversion in Java: Type-Casting

- Bei der Typkonversion in einen spezielleren Typ kann Information verloren gehen.
- Beispiele:
  - `(int)5.2` ergibt `5`
  - `(int)-5.2` ergibt `-5`

Der Type-Cast-Operator in Java

Im Ausdruck
`(type) a`
ist `(type)` ein Operator. Type-Cast-Operatoren bilden zusammen mit einem Ausdruck wieder einen Ausdruck.
Der Typ des Operators ist z.B.:

$$\texttt{(int)} : \texttt{char} \cup \texttt{byte} \cup \texttt{short} \cup \texttt{int} \cup \texttt{long} \cup \texttt{float} \cup \texttt{double} \rightarrow \texttt{int}$$

$$\texttt{(float)} : \texttt{char} \cup \texttt{byte} \cup \texttt{short} \cup \texttt{int} \cup \texttt{long} \cup \texttt{float} \cup \texttt{double} \rightarrow \texttt{float}$$

Sie können also z.B. auch **`char`** in **`int`** umwandeln.

Klingt komisch? Ist aber so! Und was passiert da?

Das Konzept Typkonversion im Allgemeinen

- Ganz allgemein nennt man das Konzept der Umwandlung eines Ausdrucks mit einem bestimmten Typ in einen Ausdruck mit einem anderen Typ *Typkonversion*.
- In vielen Programmiersprachen gibt es eine automatische Typkonversion meist vom spezielleren in den allgemeineren Typ.
- Eine Typkonversion vom allgemeineren in den spezielleren Typ muss (wenn erlaubt) sinnvollerweise immer explizit durch einen *Typecasting*-Operator herbeigeführt werden.
- Es gibt aber auch Programmiersprachen, in denen man grundsätzlich zur Typkonversion ein entsprechendes Typecasting explizit durchführen muss.
- Unabhängig davon kennen Sie jetzt das allgemeine Konzept und die Problematik solch einer Typkonversion. Unterschätzen Sie diese nicht als Fehlerquelle bei der Entwicklung von Algorithmen bzw. der Erstellung von Programmen!

Überblick

Ausdrücke mit Variablen

- Im vorherigen Kapitel haben wir Ausdrücke (in Java) nur mit Operationssymbolen (Konstanten und mehrstelligen Operatoren) gebildet.
- Warum haben wir dann die Ausdrücke vorher induktiv so definiert, dass darin auch Variablen vorkommen können?
- Antwort: Weil wir natürlich auch Variablen in (Java-)Ausdrücken zulassen wollen.

Konstanten und Variablen

- Wozu sind Variablen gut?
- Betrachten wir als Beispiel folgenden (in funktionaler Darstellung) angegebenen Algorithmus:
  - Berechne die Funktion $f(x)$ für $x \neq -1$ mit $f : \mathbb{R} \to \mathbb{R}$ gegeben durch

$$f(x) = \left(x + 1 + \frac{1}{x+1}\right)^2 \quad \text{für } x \neq -1$$

  - Eine imperative Darstellung erhält man durch Aufteilung der Funktionsdefinition in:

$$y_1 = x + 1$$
$$y_2 = y_1 + \frac{1}{y_1}$$
$$y_3 = y_2 * y_2$$
$$f(x) = y_3.$$

Konstanten und Variablen

- Intuition des Auswertungsvorgangs der imperativen Darstellung:
  - y_1, y_2 und y_3 repräsentieren drei Zettel.
  - Auf diese Zettel werden der Reihe nach Rechenergebnisse geschrieben.
  - Bei Bedarf wird der Wert auf dem Zettel “abgelesen”.
- Formal steckt hinter dieser Intuition eine Substitution.
  - x wird durch den Eingabewert substituiert.
  - y_1 wird mit dem Wert des Ausdrucks $x + 1$ substituiert wobei x bereits substituiert wurde.
  - Mit y_2 und y_3 kann man analog verfahren.
- y_1, y_2 und y_3 heißen *Konstanten*.

Konstanten und Variablen

- Bei genauerer Betrachtung:
  - Nachdem der Wert von y_1 zur Berechnung von y_2 benutzt wurde, wird er im folgenden nicht mehr benötigt.
  - Eigentlich könnte der Zettel nach dem Verwenden (Ablesen) “radiert” und für die weiteren Schritte wiederverwendet werden.
  - In diesem Fall kämen wir mit einem Zettel *y* aus.
- *y* heißt im Kontext imperativer Programmierung *Variable*.
- Im Unterschied zu Konstanten sind Variablen “radierbar”.

Konstanten und Variablen

- Die imperative Lösung zur Berechnung von $f(x)$ mit einer Variable y:

  $y = x + 1$

  $y = y + \frac{1}{y}$

  $y = y * y$

- Wenn wir im Folgenden von Konstanten und Variablen sprechen, verwenden wir immer die imperativen Konzepte (außer wir weisen explizit darauf hin), d.h. wir sprechen dann immer von Variablen ("Platzhalter") der Menge V, über die wir Ausdrücke bilden können.
- Variablen sind überschreibbare, Konstanten sind nicht überschreibbare Platzhalter.

Konstanten und Variablen

- Variablen und Konstanten können
  - *deklariert* werden, d.h. ein “leerer Zettel” (Speicherzelle) wird angelegt.
    Formal bedeutet dies, dass der Bezeichner der Menge der zur Verfügung stehenden Variablen V, die wir in Ausdrücken verwenden dürfen, hinzugefügt wird.
  - *intialisiert* werden, d.h. ein Wert wird auf den “Zettel” (in die Speicherzelle) geschrieben.
    Formal bedeutet dies, dass die Variable mit einem entsprechenden Wert belegt ist.
- Der Wert einer Variablen kann später auch noch durch eine (neue) *Wertzuweisung* verändert werden.
- Die Initialisierung enspricht einer (initialen) Wertzuweisung.
- Nach der Deklaration kann der Wert einer *Konstanten* nur noch einmalig durch eine Wertzuweisung verändert (initialisiert) werden.
- Nach der Deklaration kann der Wert einer *Variablen* beliebig oft durch eine Wertzuweisung verändert werden.

Konstanten und Variablen in Java

- Eine Variablendeklaration hat in Java die Gestalt
  `Typname variablenName;`
  Konvention: Variablennamen beginnen mit kleinen Buchstaben.
- Eine Konstantendeklaration hat in Java die Gestalt
  `final Typname KONSTANTENNAME;`
  Konvention: Konstantennamen bestehen komplett aus großen Buchstaben.
- Das bedeutet: in Java hat jede Variable/Konstante einen Typ.
- Eine Deklaration ist also das Bereitstellen eines “Platzhalters” des entsprechenden Typs.

Konstanten und Variablen in Java

- Eine Wertzuweisung (z.B. Initialisierung) hat die Gestalt
  `variablenName = NeuerWert;`
  bzw.
  `KONSTANTENNAME = Wert;` (nur als Initialisierung)
- Eine Variablen- bzw. Konstantendeklaration kann auch mit der Initialisierung verbunden sein, d.h. der ersten Wertzuweisung.
  `Typname variablenName = InitialerWert;`
  (Konstantendeklaration mit Initialisierung analog mit Zusatz **`final`**)
- Formal gesehen ist eine Wertzuweisung eine Funktion

$$=: S \to S$$

  mit beliebigem Typ S.
- Damit ist eine Wertzuweisung auch wieder ein Ausdruck.

Abkürzungen für Wertzuweisungen in Java

Für bestimmte einfache Operationen (Addition und Subtraktion mit 1 als zweitem Operanden) gibt es gängige Kurznotationen:

Operator	Bezeichnung	Bedeutung
`++`	Präinkrement	`++a` ergibt `a+1` und erhöht `a` um `1`
`++`	Postinkrement	`a++` ergibt `a` und erhöht `a` um `1`
`--`	Prädekrement	`--a` ergibt `a-1` und verringert `a` um `1`
`--`	Postdekrement	`a--` ergibt `a` und verringert `a` um `1`

Abkürzungen für Wertzuweisungen in Java

Wenn man eine Variable nicht nur um 1 erhöhen oder verringern, sondern allgemein einen neuen Wert zuweisen will, der aber vom alten Wert abhängig ist, gibt es Kurznotationen wie folgt:

Operator	Bezeichnung	Bedeutung
`+=`	Summe	`a+=b` weist `a` die Summe von `a` und `b` zu
`-=`	Differenz	`a-=b` weist `a` die Differenz von `a` und `b` zu
`*=`	Produkt	`a*=b` weist `a` das Produkt aus `a` und `b` zu
`/=`	Quotient	`a/=b` weist `a` den Quotienten von `a` und `b` zu

(Auch für weitere Operatoren möglich...)

Konstanten und Variablen in Java

- Konstanten:

```
final <typ> <NAME> = <ausdruck>;
```

```
final double Y_1 = x + 1; //Achtung: was ist x???
final double Y_2 = Y_1 + 1 / Y_1;
final double Y_3 = Y_2 * Y_2;
final char NEWLINE = '\n';
final double BESTEHENSGRENZE_PROZENT = 0.5;
final int GESAMTPUNKTZAHL = 80;
```

- Variablen:

```
<typ> <name> = <ausdruck>;
```

```
double y = x + 1; //Achtung: was ist x???
int klausurPunkte = 42;
boolean klausurBestanden =
        ((double) klausurPunkte) /
          GESAMTPUNKTZAHL >= BESTEHENSGRENZE_PROZENT;
```

Zustandsänderungen von Variablen

Beispiel:

Anweisung	x	y	z
`int x;`			
`int y = 5;`		5	
`x = 0;`	0	5	
`final int z = 3;`	0	5	| 3 |
`x += 7 + y;`	12	5	| 3 |
`y = y + z - 2;`	12	6	| 3 |
`x = x * y;`	72	6	| 3 |
`x = x/z;`	24	6	| 3 |

Legende: [ ] = radierbar [\| \|] = nicht radierbar

Überblick

Anweisungen

- In imperativen Programmen sind *Anweisungen* die elementaren Einheiten.
- Eine Anweisung steht für einen einzelnen Abarbeitungsschritt in einem Algorithmus.
- Anweisungen können unter anderem aus Ausdrücken gebildet werden.

Anweisungen in Java

- Im folgenden: Anweisungen in Java.
- Eine Anweisung wird immer durch das Semikolon begrenzt.

Die leere Anweisung

- Die einfachste Anweisung ist die leere Anweisung:
  ```
  ;
  ```
- Die leere Anweisung hat keinen Effekt auf das laufende Programm, sie bewirkt nichts.
- Oftmals benötigt man tatsächlich eine leere Anweisung, wenn von der Logik des Algorithmus *nichts* zu tun ist, die Programmsyntax aber eine Anweisung erfordert.

Der Block

- Ein Block wird gebildet von einer öffnenden geschweiften Klammer und einer schließenden geschweiften Klammer, die eine beliebige Menge von Anweisungen umschließen:

```
{
  Anweisung1;
  Anweisung2;
  ...
}
```

- Die Anweisungen im Block werden nacheinander ausgeführt.
- Der Block als ganzes gilt als eine einzige Anweisung, kann also überall da stehen, wo syntaktisch eine einzige Anweisung verlangt ist.
- Eine Anweisung in einem Block kann natürlich auch wieder ein Block sein.

Lebensdauer, Gültigkeit, Sichtbarkeit

- Die *Lebensdauer* einer Variablen ist die Zeitspanne, in der die virtuelle Maschine der Variablen einen *Speicherplatz* zu Verfügung stellt.
- Die *Gültigkeit* einer Variablen erstreckt sich auf alle Programmstellen, an denen der Name der Variablen dem Compiler durch eine Vereinbarung (*Deklaration*) bekannt ist.
- Die *Sichtbarkeit* einer Variablen erstreckt sich auf alle Programmstellen, an denen man über den Namen der Variablen auf ihren Wert zugreifen kann.

Gültigkeitsbereich von Variablen

- Eine in einem Block deklarierte (lokale) Variable ist ab ihrer Deklaration bis zum Ende des Blocks gültig und sichtbar.
- Mit Verlassen des Blocks, in dem eine lokale Variable deklariert wurde, endet auch ihre Gültigkeit und Sichtbarkeit.
- Damit oder kurz danach endet normalerweise auch die Lebensdauer der Variablen, da der Speicherplatz, auf den die Variable verwiesen hat, im Prinzip wieder freigegeben ist und für neue Variablen verwendet werden kann.
- Solange eine lokale Variable sichtbar ist, darf keine neue lokale Variable gleichen Namens angelegt werden.

Gültigkeitsbereich von Variablen

- Beispiel:

```
...
int i = 0;
{
  int i = 1;   // nicht erlaubt
  i = 1;       // erlaubt
  int j = 0;
}
j = 1;         // nicht moeglich
...
```

Ausdrucksanweisung

Aus einem Ausdruck `<ausdruck>` wird durch ein angefügtes Semikolon eine Anweisung. Allerdings spielt dabei der Wert des Ausdrucks im weiteren keine Rolle. Daher ist eine solche Ausdrucksanweisung auch nur sinnvoll (und in Java nur dann erlaubt), wenn der Ausdruck einen Nebeneffekt hat. Solche Ausdrücke sind:

- Wertzuweisung
- (Prä-/Post-)Inkrement
- (Prä-/Post-)Dekrement
- Methodenaufruf (werden wir später kennenlernen)
- Instanzerzeugung (werden wir später kennenlernen)

Überblick

Global – lokal

- Mit den bisherigen Konzepten können wir theoretisch einfache imperative Algorithmen und Programme schreiben.
- Wir werden später weitere Konzepte kennenlernen, um komplexere Algorithmen/Programme zu strukturieren:
  - Prozeduren (in Java statische Methoden genannt).
    Beispiel: die Prozedur (Methode) `main`. Ein Algorithmus ist in einer Prozedur (Methode) verpackt.
  - Module (in Java Pakete bzw. Packages genannt)
- Variablen, so wie wir sie bisher kennengelernt haben, sind *lokale* Variablen und Konstanten, d.h. sie sind nur innerhalb des Blocks (Algorithmus, Prozedur, Methode), der sie verwendet, bekannt.
- Darüberhinaus gibt es auch noch *globale* Variablen und Konstanten, die in mehreren Algorithmen (Prozeduren/Methoden) und sogar Modulen bekannt sind.
- Diese globalen Größen sind z.B. für den Datenaustausch zwischen verschiedenen Algorithmen geeignet.

Globale und lokale Größen in Java

Globale Variablen heißen in Java *Klassenvariablen*. Diese Variablen gelten in der gesamten Klasse und ggf. auch darüberhinaus.
Eine Klassenvariable definiert man üblicherweise am Beginn einer Klasse, z.B.:

```
public class HelloWorld
{
  public static String gruss = "Hello, World!";

  public static void main(String[] args)
  {
    System.out.println(gruss);
  }
}
```

Die Definition wird von den Schlüsselwörtern **`public`** und **`static`** eingeleitet. Deren Bedeutung lernen wir später kennen.

Konstante Klassenvariablen

Klassenvariablen kann man auch als Konstanten definieren. Wie bei lokalen Variablen dient hierzu das Schlüsselwort **`final`**:

```
public class HelloWorld
{
  public static final String GRUSS = "Hello, World!";

  public static void main(String[] args)
  {
    System.out.println(GRUSS);
  }
}
```

Auch bei Klassenvariablen schreibt man Namen von Konstanten in Großbuchstaben.

Klasse und Programm

- Zur Erinnerung: Ein Java-Programm besteht aus Klassen – einer oder mehreren, in größeren Projekten oft hunderten oder tausenden.
- Da eine Klasse auch einen Block bildet, ist der Gültigkeitsbereich einer Klassenvariablen klar: Sie gilt im gesamten Programmcode innerhalb der Klasse nach ihrer Deklaration.
- Darüberhinaus gilt sie aber in der gesamten Programmlaufzeit, d.h. solange das Programm ausgeführt wird, “lebt” eine Klassenvariable.
- Die Sichtbarkeit in anderen als ihrer eigenen Klasse kann man aber einschränken.
- Eine Klasse gehört immer zu einem Package. Die Klassendatei liegt in einem Verzeichnis, das genauso heißt wie das Package.
- Der Package-Name gehört zum Klassennamen.

Nutzen von Klassenvariablen

Als Klassenvariablen definiert man z.B. gerne Werte, die von universellem Nutzen sind. Die Klasse `java.lang.Math` definiert die mathematischen Konstanten *e* und π:

```
package java.lang;

public final class Math {
  ...
  /**
    * The <code>double</code> value that is closer than any other to
    * <i>e</i>, the base of the natural logarithms.
    */
  public static final double E = 2.7182818284590452354;

  /**
    * The <code>double</code> value that is closer than any other to
    * <i>pi</i>, the ratio of the circumference of a circle to its
    * diameter.
    */
  public static final double PI = 3.14159265358979323846;
  ...
}
```

Automatische Initialisierung

Im Gegensatz zu lokalen Variablen muss man Klassenvariablen nicht explizit initialisieren. Sie werden dann automatisch mit ihren Standardwerten initialisiert:

Typname	Standardwert
`boolean`	`false`
`char`	`\u0000`
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0`
`float`	`0.0`
`double`	`0.0`

Klassenkonstanten müssen dagegen explizit initialisiert werden.

Namen von lokalen Variablen und Klassenvariablen

- Lokale Variablen innerhalb einer Klasse können genauso heißen wie eine Klassenvariable.

```
public class Sichtbarkeit
{
    public static int variablenname;

    public static void main(String[] args)
    {
        boolean variablenname = true;
    }
}
```

- Das bedeutet: Während bei lokalen Variablen *Sichtbarkeit* und *Gültigkeit* zusammenfallen, muss man zwischen beiden Eigenschaften bei Klassenvariablen prinzipiell unterscheiden.

Namen von lokalen Variablen und Klassenvariablen

- Das ist aber kein Widerspruch zum Verbot, den gleichen Namen innerhalb des Gültigkeitsbereichs einer Variable nochmal zu verwenden, denn genaugenommen heißt die Klassenvariable doch anders:
- Zu ihrem Namen gehört der vollständige Klassenname (inklusive des Package-Namens).
  Der vollständige Name der Konstanten `PI` aus der `Math`-Klasse ist also:
  `java.lang.Math.PI`
- Unter dem vollständigen Namen ist eine globale Variable auch dann sichtbar, wenn der innerhalb der Klasse geltende Name (z.B. `PI`) durch den identisch gewählten Namen einer lokalen Variable verdeckt ist.

Namen von lokalen Variablen und Klassenvariablen

```
public class Sichtbarkeit
{
    public static int variablenname;

    public static void main(String[] args)
    {
        boolean variablenname = true;
        System.out.println(variablenname); // Ausgabe: true
        System.out.println(Sichtbarkeit.variablenname); // Ausgabe?
    }
}
```

Überblick

Eingebaute Datenstruktur

- In einer Programmiersprache ist üblicherweise eine einfache Datenstruktur eingebaut, die es ermöglicht, eine Reihe von Werten (gleichen Typs) zu modellieren.
- Im imperativen Paradigma ist das normalerweise das Array (Reihung, Feld).

  Vorteil: direkter Zugriff auf jedes Element möglich

  Nachteil: dynamisches Wachstum ist nicht möglich
  (In Java sind Arrays *semidynamisch*, d.h., ihre Größe kann zur Laufzeit (=dynamisch) festgesetzt werden, danach aber nicht mehr geändert werden.)

Reihung

Definition

Eine Reihung (Feld, Array) ist ein Tupel von Komponentengliedern gleichen Typs, auf die über einen Index direkt zugegriffen werden kann.

Darstellung

Eine **`char`**-Reihung `gruss` der Länge 13:

`gruss`:	’H’	’e’	’l’	’l’	’o’	’,’	’ ’	’W’	’o’	’r’	’l’	’d’	’!’
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12

Mathematische Darstellung

Allgemein

Eine Reihung mit n Komponenten vom Typ `<typ>` ist eine Abbildung von der Indexmenge I_n auf die Menge `<typ>`.

Beispiel

$$\texttt{gruss} \;:\; \{0,1,\ldots,12\} \rightarrow \textbf{char}$$

$$i \mapsto \begin{cases} \text{'H'} & \text{falls} \quad i = 0 \\ \text{'e'} & \text{falls} \quad i = 1 \\ \vdots & \\ \text{'!'} & \text{falls} \quad i = 12 \end{cases}$$

Deklaration von Arraytypen und Arrayvariablen in Java

- Der Typ eines Arrays, das den Typ
  `<typ>`
  enthält, ist in Java:
  `<typ>[]`.
- Der Wert des Ausdrucks
  `<typ>[i]`
  ist der Wert des Arrays an der Stelle `i`.

Deklaration von Arraytypen und Arrayvariablen in Java

Ein Array wird zunächst behandelt wie Variablen von primitiven Typen auch. Wir können also z.B. deklarieren:

```
char a = 'a';
char b = 'b';
char c = 'c';
char[] abc = {a, b, c};
System.out.print(abc[0]); // gibt den Character 'a' aus,
                          // den Wert des Array-Feldes
                          // mit Index 0. Allgemein: array[i]
                          // ist Zugriff auf das i-te Element
char d = 'd';
char e = 'e';
char[] de = {d, e};
abc = de;  // die Variable abc hat jetzt den gleichen
           // Wert wie die Variable de
System.out.print(abc[0]); // Ausgabe ?
```

Länge einer Reihung

Ein Array hat immer eine feste Länge, die in einer Variablen `length` festgehalten wird. Diese Variable `length` wird mit einem Array automatisch miterzeugt. Der Name der Variablen ist zusammengesetzt aus dem Namen des Arrays und dem Namen `length`:

```
char a = 'a';
char b = 'b';
char c = 'c';
char[] abc = {a, b, c};
System.out.print(abc.length); // gibt 3 aus
char[] de = {d, e};
abc = de; // gleicher Variablenname,
          // haelt aber als Wert ein
          // anderes Array
System.out.print(abc.length); // Ausgabe ?
```

Reihungen dynamisch anlegen

- Oft legt man ein Array an, bevor man die einzelnen Elemente kennt. Die Länge muß man dabei angeben:

```
char[] abc = new char[3];
```

- Dass Arrays in Java *semidynamisch* sind, bedeutet: Es ist möglich, die Länge erst zur Laufzeit festzulegen.

```
// x ist eine Variable vom Typ int
// deren Wert bei der Ausfuehrung
// feststeht, aber nicht
// beim Festlegen des Programmcodes
char[] abc = new char[x];
```

- Dann kann man das Array im weiteren Programmverlauf füllen:

```
abc[0] = 'a';
abc[1] = 'b';
abc[2] = 'c';
```

Leere Reihungen?

- Wenn man ein Array anlegt:

```
int[] zahlen = new int[10];
```

  aber nicht füllt – ist es dann leer?

- Es gibt in Java keine leeren Arrays. Ein Array wird immer mit den Standardwerten des jeweiligen Typs befüllt.
- Das spätere Belegen einzelner Array-Zellen ist also immer eine Änderung eines Wertes.

```
int[] zahlen = new int[10];
System.out.print(zahlen[3]); // gibt 0 aus
zahlen[3] = 4;
System.out.print(zahlen[3]); // gibt 4 aus
```

Array-Variablen als Konstanten

Auch Array-Variablen kann man als Konstanten deklarieren. Dann kann man der Variablen keinen neuen Wert zuweisen:

```
final char[] ABC = {'a', 'b', 'c'};
final char[] DE = {'d', 'e'};
ABC = DE; // ungueltige Anweisung: Compilerfehler
```

Aber einzelne Array-Komponenten sind normale Variablen. Man kann ihnen also einen neuen Wert zuweisen. Die Länge des Arrays ändert sich dadurch nicht:

```
ABC[0] = 'd';
ABC[1] = 'e'; // erlaubt
System.out.print(ABC.length); // gibt 3 aus
```

Array als Funktion

Rufen wir uns die abstrakte Betrachtungsweise eines Arrays als Funktion in Erinnerung:

```
final char[] ABC = {'a', 'b', 'c'};
```

$$\texttt{ABC} \quad : \quad \{0,1,2\} \rightarrow \textbf{char}$$

$$i \mapsto \begin{cases} \text{'a'} & \text{falls} \quad i = 0 \\ \text{'b'} & \text{falls} \quad i = 1 \\ \text{'c'} & \text{falls} \quad i = 2 \end{cases}$$

Was bedeutet die Neubelegung einzelner Array-Zellen?

```
ABC[0] = 'd';
ABC[1] = 'e';
```

Der Typ `String` als spezielles Array

- In der Deklaration und Initialisierung

  ```
  public static String gruss = "Hello, World!";
  ```

  stellt der Ausdruck `"Hello, World!"` eine spezielle Schreibweise für ein konstantes Array `char[13]` dar, das in einen Typ `String` gekapselt wird.
- Durch die besonderen Eigenschaften des Typs `String` können die Komponenten dieses Arrays nicht mehr (durch Neuzuweisung) geändert werden.
- Der Typ `String` ist kein *primitiver* Typ, sondern eine Klasse von Objekten. Wir untersuchen diesen Typ daher erst in einem späteren Abschnitt der Vorlesung genauer.
- Werte dieses Typs können aber – wie bei primitiven Typen – durch Literale gebildet werden.
- Literale und komplexere Ausdrücke vom Typ `String` können durch den (abermals überladenen!) Operator `+` konkateniert werden.

Mehrdimensionale Reihungen

Da auch Arrays einen bestimmten Typ haben (z.B. `gruss` : **`char`**`[]`), kann man auch Reihungen von Reihungen bilden. Mit einem Array von Arrays lassen sich z.B. Matrizen modellieren:

```
int[] m0 = {1, 2, 3};
int[] m1 = {4, 5, 6};
int[][] m = {m0, m1};
```

Überblick

Prozeduren

- Das Konzept der Prozedur dient zur Abstraktion von Algorithmen (oder von einzelnen Schritten eines Algorithmus).
- Durch Parametrisierung wird von der Identität der Daten abstrahiert: die Berechnungsvorschriften werden mit abstrakten Parametern formuliert – konkrete Eingabedaten bilden die aktuellen (Parameter-) Werte.
- Durch Spezifikation des (Ein-/Ausgabe-) Verhaltens wird von den Implementierungsdetails abstrahiert: Vorteile sind
  - **Örtliche Eingrenzung (Locality)**: Die Implementierung einer Abstraktion kann verstanden oder geschrieben werden, ohne die Implementierungen anderer Abstraktionen kennen zu müssen.
  - **Änderbarkeit (Modifiability)**: Jede Abstraktion kann reimplementiert werden, ohne dass andere Abstraktionen geändert werden müssen.
  - **Wiederverwendbarkeit (Reusability)**: Die Implementierung einer Abstraktion kann beliebig wiederverwendet werden.

Funktionen und Prozeduren

- Im funktionalen Programmierparadigma werden Algorithmen als Funktionen dargestellt.
- Das imperative Pendant dazu ist die Prozedur, die sogar ein etwas allgemeineres Konzept darstellt: Eine Funktion kann man als Prozedur bezeichnen, aber nicht jede Prozedur ist eine Funktion.
- Eine Funktion stellt nur eine Abbildung von Elementen aus dem Definitionsbereich auf Elemente aus dem Bildbereich dar.
- Es werden aber keine Werte verändert.
- Im imperativen Paradigma können Werte von Variablen verändert werden (durch Anweisungen). Dies kann Effekte auf andere Bereiche eines Programmes haben.
- Treten in einer Prozedur solche sog. Seiteneffekte (oder Nebeneffekte) auf, kann man nicht mehr von einer Funktion sprechen.
- Eine Funktion kann man also als eine Prozedur ohne Seiteneffekte auffassen.

Funktionen und Prozeduren

- Wir haben bereits Prozeduren mit Seiteneffekten verwendet:

```
public class HelloWorld
{
  public static final String GRUSS = "Hello, World!";

  public static void main(String[] args)
  {
    System.out.println(GRUSS);
  }
}
```

Funktionen und Prozeduren

- Bei einer Funktion ist der Bildbereich eine wichtige Information:

$$f : D \rightarrow B$$

- Bei einer Prozedur, die keine Funktion ist, wählt man als Bildbereich oft die leere Menge:

$$p : D \rightarrow \emptyset$$

  Dies signalisiert, dass die Seiteneffekte der Prozedur zur eigentlichen Umsetzung eines Algorithmus gehören, dagegen aber kein (bestimmtes) Element aus dem Bildbereich einer Abbildung als Ergebnis des Algorithmus angesehen werden kann.
- Sehr häufig findet man in imperativen Implementierungen von Algorithmen aber eine Mischform, in der eine Prozedur sowohl Seiteneffekte hat als auch einen nicht-leeren Bildbereich.

Prozeduren in Java

- In Java werden Prozeduren durch *Methoden* realisiert.
- Eine Methode wird definiert durch den Methodenkopf:
  `public static` `<typ> <name>(<parameterliste>)`
  und den Methodenrumpf, einen Block, der sich an den Methodenkopf anschließt.
- Als besonderer Ergebnis-Typ einer Methode ist auch **`void`** möglich. Dieser Ergebnis-Typ steht für die leere Menge als Bildbereich.
- Eine Methode mit Ergebnistyp **`void`** gibt *kein* Ergebnis zurück. Der Sinn einer solchen Methode liegt also ausschließlich in den Nebeneffekten.
- Eine Methode, die Nebeneffekte hat, die den weiteren Programmverlauf beeinflussen, sollte als Ergebnis-Typ stets **`void`** haben.
- Manchmal wählt man als Ergebnistyp auch **`boolean`** und zeigt mit dem Ergebnis*wert* an, ob der beabsichtigte Nebeneffekt erfolgreich war.
- Das Ergebnis einer Methode ist der Ausdruck nach dem Schlüsselwort **`return`**. Nach Auswertung dieses Ausdrucks endet die Ausführung der Methode.

Prozeduren in Java

```
public class Streckenberechnung
{
  /**
   * Methode zur Berechnung der zurueckgelegten Strecke
   * nach Einwirken der Kraft <code>k</code>
   * auf einen Koerper der Masse <code>m</code>
   * fuer die Zeitdauer <code>t</code>.
   *
   * @param m Masse des Koerpers
   * @param t Zeitdauer
   * @param k Kraft
   * @return zurueckgelegte Strecke
   */
  public static double strecke(double m, double t, double k)
  {
    double b = k / m;
    return 0.5 * b * (t * t);
  }
}
```

Prozeduren in Java

Beispiel: Algorithmus für die Funktion

$$f(x) = \left(x + 1 + \frac{1}{x+1}\right)^2 \quad \text{für } x \neq -1$$

```
public class Funktionsberechnung
{
  /**
   * Methode zur Berechnung der Funktion
   * <i>f(x) = ((x + 1) + 1 / (x + 1))<sup>2</sup></i>.
   *
   * @param x der Eingabewert
   * @return Wert der Funktion f an der Stelle x
   */
  public static double f(double x)
  {
    double y = x + 1;
    y = y + 1/y;
    y = y * y;
    return y;
  }
}
```

Methodenaufruf

Wie wir im Abschnitt über Anweisungen gesehen haben, bildet ein Methodenaufruf eine Anweisung. Ein Methodenaufruf kann also überall da stehen, wo eine Anweisung möglich ist.
Beispiel für einen Methodenaufruf:

```
public class HelloWorld
{
  public static void gruessen(String gruss)
  {
    System.out.println(gruss);
      // auch println(...); ist ein Methodenaufruf
  }

  public static void main(String[] args)
  {
    // Aufruf der Methode gruessen
    gruessen("Hello, World!"); // abstrakter Parameter gruss
                               // wird mit konkretem Wert belegt
  }
}
```

Die `main`-Methode

- In Programmbeispielen haben wir bereits die `main`-Methode gesehen. Die `main`-Methode ermöglicht das selbständige Ausführen eines Programmes.
- Der Aufruf `java KlassenName` führt die `main`-Methode der Klasse `KlassenName` aus.
- Die `main`-Methode hat immer einen Parameter, ein `String`-Array. Dies ermöglicht das Verarbeiten von Argumenten, die über die Kommandozeile übergeben werden.

Parameter für die `main`-Methode

Beispiel für einen Zugriff der `main`-Methode auf das Parameterarray:

```
public class Gruss
{
  public static void gruessen(String gruss)
  {
    System.out.println(gruss);
  }

  public static void main(String[] args)
  {
    gruessen(args[0]);
  }
}
```

Dadurch vielfältigere Verwendung möglich:

- `java Gruss "Hello, World!"`
- `java Gruss "Hallo, Welt!"`
- `java Gruss "Servus!"`

Verarbeitung von Parametern

Zur Verarbeitung der Parameter von Prozeduren kennen Programmiersprachen zwei grundsätzliche Möglichkeiten:

- call-by-value
  Im Aufruf `methodenName(parameter)` wird für `parameter` im Methoden-Block eine neue Variable angelegt, in die der Wert von `parameter` geschrieben wird. Auf diese Weise bleibt die ursprüngliche Variable `parameter` von Anweisungen innerhalb der Methode unberührt.
- call-by-reference
  Im Aufruf `methodenName(parameter)` wird die Variable `parameter` weiter verwendet. Wenn innerhalb der Methode der Wert von `parameter` verändert wird, hat das auch Auswirkungen außerhalb der Methode.
  call-by-reference ist daher eine potentielle Quelle unbeabsichtigter Seiteneffekte.

In Java: call-by-value

Java wertet Parameter call-by-value aus. Für den Aufruf der Methode `swap` im folgenden Beispiel werden also Kopien der Variablen `x` und `y` angelegt.

```
public class Exchange
{
  public static void swap(int i, int j)
  {
    int c = i;
    i = j;
    j = c;
  }

  public static void main(String[] args)
  {
    int x = 1;
    int y = 2;
    swap(x,y);
    System.out.println(x); // Ausgabe?
    System.out.println(y); // Ausgabe?
  }
}
```

Ausblick: call-by-reference-Effekt bei Objekten

Wenn der Parameter kein primitiver Typ ist, sondern ein Objekt (also z.B. ein Array – was genau Objekte sind, betrachten wir aber später) dann wird zwar in der Methode ebenfalls mit einer Kopie des Parameters gearbeitet, aber es handelt sich um eine Kopie der Speicheradresse, also eine zweite Zugriffsmöglichkeit für den selben Zettel.

```
public static void changeValues(int[] werte, int index, int wert)
{
  werte[index] = wert;
}

public static void main(String[] args)
{
  int[] werte = {0, 1, 2};
  changeValues(werte, 1, 3);
  System.out.println(werte[1]); // Ausgabe ?
}
```

Obwohl also auch hier die Parameterauswertung nach dem Prinzip call-by-value erfolgt, ist der Effekt der gleiche wie bei call-by-reference. Wir werden auf diesen Effekt zurückkommen.

Überblick

Bedingte Anweisungen

Motivation:

- In vielen Algorithmen benötigt man eine Fallunterscheidung zur Lösung des gegebenen Problems.
- Beispiel: Falls . . . dann . . . im Algorithmus *Wechselgeld 1*

  Führe folgende Schritte der Reihe nach aus:

  1. Setze $w = ()$.
  2. Falls die letzte Ziffer von r eine 2,4,7 oder 9 ist, dann erhöhe r um 1 und nimm 1 zu w hinzu.
  3. Falls die letzte Ziffer von r eine 1 oder 6 ist, dann erhöhe r um 2 und nimm 2 zu w hinzu.
  4. Falls die letzte Ziffer von r eine 3 oder 8 ist, dann erhöhe r um 2 und nimm 2 zu w hinzu.
  5. Solange $r < 100$: Erhöhe r um 5 und nimm 5 zu w hinzu.

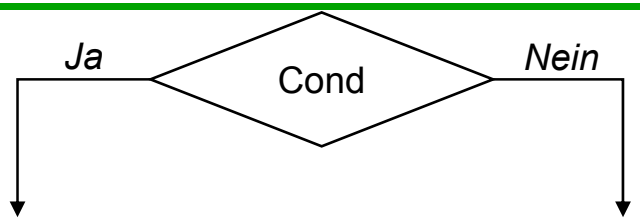
Bedingte Anweisungen

- Im einfachsten Fall (Pseudo-Code):
  `IF <Bedingung> THEN <Ausdruck1> ELSE <Ausdruck2> ENDIF`
- Dies entspricht einer echten *Fallunterscheidung*:
  - Ist `<Bedingung>` wahr, dann wird `<Ausdruck1>` ausgewertet.
  - Ist `<Bedingung>` falsch, dann wird `<Ausdruck2>` ausgewertet.
- Im funktionalen Paradigma modelliert man mit einer Fallunterscheidung eine Abbildung. Deshalb müssen dort im Allgemeinen `<Ausdruck1>` und `<Ausdruck2>` den selben Typ haben.
- In der imperativen Programmierung ist dies nicht der Fall:
  - In den verschiedenen Zweigen stehen *Anweisungen* anstelle von Ausdrücken.
  - Damit entfällt die Forderung nach gleichen Typen.

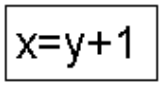
Bedingte Anweisungen

- Man spricht daher im imperativen Paradigma von *bedingten Anweisungen*.
- Die Fallunterscheidung ist ein Spezialfall der bedingten Anweisung.
- Die einfachste Form von bedingten Anweisungen ist:
  ```
  IF <Bedingung> THEN <Anweisungsfolge> ENDIF
  ```
- Bedingte Anweisungen können beliebig oft verzweigt werden:
  ```
  IF <Bedingung1> THEN <Anweisungsfolge1>
  ELSE IF <Bedingung2> THEN <Anweisungsfolge2>
  ⋮
  ELSE <AnweisungsfolgeN>
  ENDIF
  ```
- Die einzelnen Zweige nennt man auch *bewachte Anweisungen*.

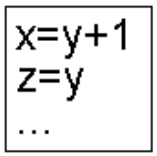
Graphische Darstellung: Kontrollflussdiagramme



Fallunterscheidung bzgl. der Bedingung `Cond`

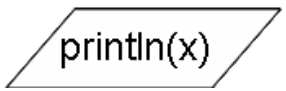


Anweisung



Block

```
{
  x = y + 1;
  z = y;
  ...
}
```

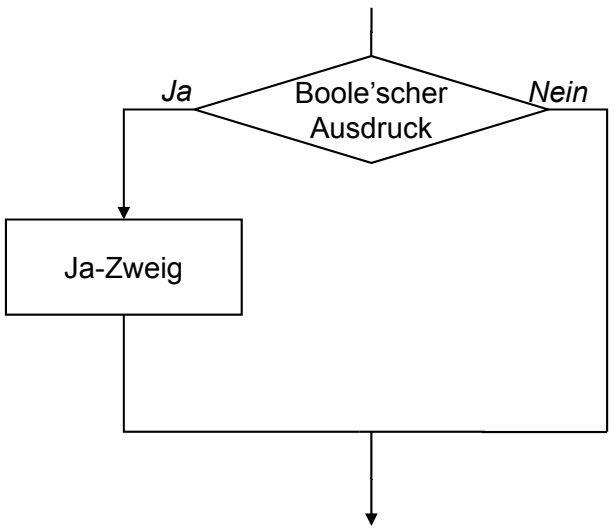


Ein- / Ausgabe

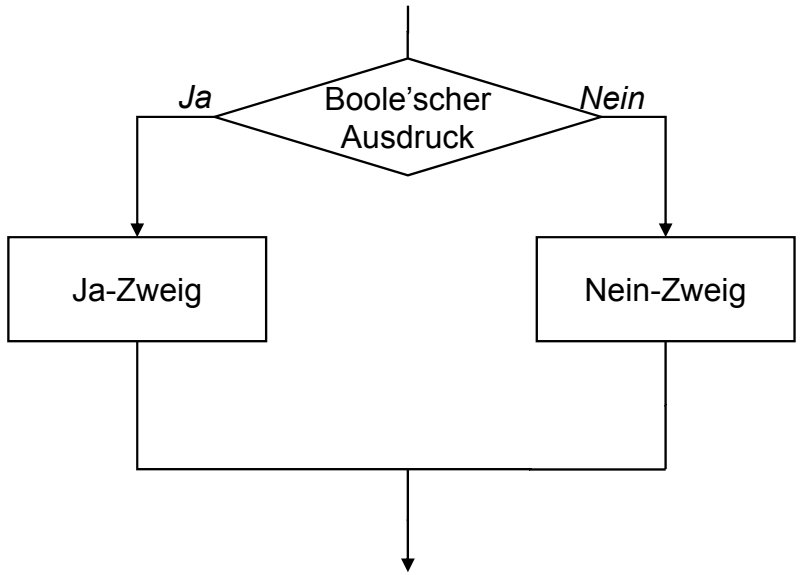
Sequentielle Abfolge

Kontrollflussdiagramme für bedingte Anweisungen

Einfache bedingte Anweisung:



Zweifache bedingte Anweisung:



Bedingte Anweisungen und Rekursion

- Rekursion ist ein nützliches und elegantes Entwurfskonzept für Algorithmen.
- Ein rekursiver Entwurf einer Methode verwendet das Ergebnis eines Aufrufs dieser Methode selbst in ihrem Rumpf wieder.
- Damit die Rekursion terminiert, benötigt man in der Regel einen oder mehrere Basisfälle (neben einem oder mehreren Rekursionsfällen).
- Diese verschiedenen Fälle werden typischerweise durch bedingte Anweisungen realisiert.

Erinnerung: Fakultäts-Funktion

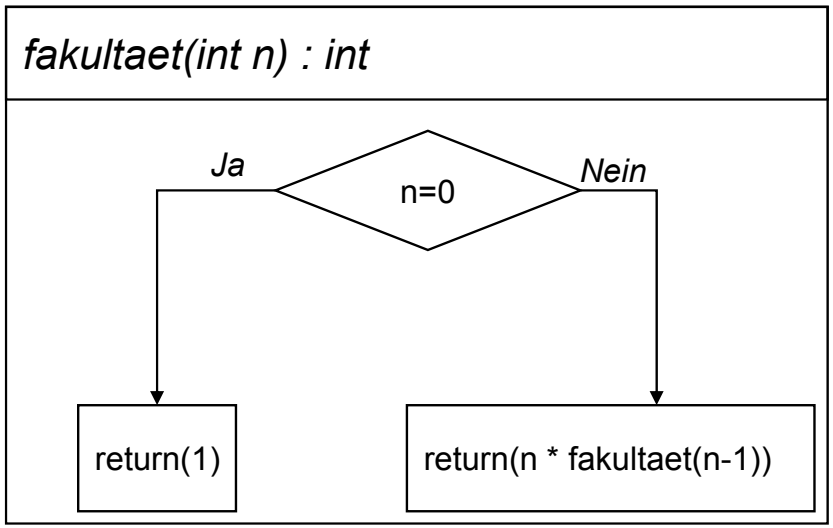
Eine rekursive Definition in der Mathematik haben wir am Beispiel der Fakultätsfunktion betrachtet:

- Die Fakultäts-Funktion $! : \mathbb{N}_0 \to \mathbb{N}_0$ ist rekursiv definiert wie folgt:
  - $0! = 1$
  - $(n+1)! = (n+1) \cdot (n!)$
- Oft wird äquivalent statt der Rückführung von $n+1$ auf n der Fall $n \neq 0$ auf $n-1$ zurückgeführt:

$$n! = \begin{cases} 1, & \text{falls } n = 0, \\ n \cdot (n-1)! & \text{sonst.} \end{cases}$$

Bedingte Anweisungen und Rekursion

- Beispiel: Algorithmus zur Berechnung der Fakultät für eine natürliche Zahl $n \in \mathbb{N}$ als Kontrollflussdiagramm:



Bedingte Anweisungen in Java

- Java erlaubt zwei Formen von bedingten Anweisungen (auch: **`if`**-*Schleifen*):
  1. Eine einfache Verzweigung:

     ```
     if (<Bedingung>)
        <Anweisung>
     ```

  2. Eine zweifache Verzweigung:

     ```
     if (<Bedingung>)
        <Anweisung1>
     else
        <Anweisung2>
     ```

  wobei

  - `<Bedingung>` ein Ausdruck vom Typ **`boolean`** ist,
  - `<Anweisung>`, `<Anweisung1>` und `<Anweisung2>` jeweils einzelne Anweisungen (also möglicherweise auch einen Block mit mehreren Anweisungen) darstellen.

Bedingte Anweisungen in Java

- Beispiel: Der Algorithmus zur Berechnung der Fakultät einer natürlichen Zahl $n \in \mathbb{N}$ kann in Java durch folgende Methode umgesetzt werden:

```
public static int fakultaet01(int n)
{
  if(n==0)
  {
    return 1;
  }
  else
  {
    return n * fakultaet01(n-1);
  }
}
```

Bedingte Anweisungen in Java

- Bedingte Anweisungen mit mehr als zwei Zweigen müssen in Java durch Schachtelung mehrerer **`if`**-Schleifen ausgedrückt werden:

```
if (<Bedingung1>)
   <Anweisung1>
else if (<Bedingung2>)
   <Anweisung2>

⋮

else if (<BedingungN>)
   <AnweisungN>
else
   <AnweisungN+1>
```

Dangling Else

- Gegeben:

```
if (a)
  if (b)
    s1;
else
  s2;
```

- Frage: Zu welchem **`if`**-Statement gehört der **`else`**-Zweig?

Dangling Else

- Antwort: Zur inneren Verzweigung `if (b)`. (Die (falsche!) Einrückung ist belanglos für den Compiler und verführt den menschlichen Leser hier, das Programm falsch zu interpretieren.)
- Tipp: Immer Blockklammern verwenden!

```
if (a)
{
   if (b)
   {
      s1;
   }
   else
   {
      s2;
   }
}
```

Mehrfachanweisungen

- Bei Mehrfachanweisungen im **`else`**-Zweig sind Blockklammern sehr wichtig, sonst kann es zu falschen Ergebnissen kommen.
- Beispiel:
  Ein Kunde einer Bank hebt einen Betrag (Variable `betrag`) von seinem Konto (Variable `kontoStand` für den aktuellen Kontostand) ab. Falls der Betrag nicht gedeckt ist, wird eine Überziehungsgebühr fällig. Die fälligen Gebühren werden über einen bestimmten Zeitraum akkumuliert (Variable `gebuehren`) und am Ende des Zeitraums in Rechnung gestellt. Was ist falsch in folgender Berechnung?

```
if (kontoStand >= betrag)
{
   double neuerStand = kontoStand - betrag;
   kontoStand = neuerStand;
}
else
   kontoStand = kontoStand - betrag;
   gebuehren  = gebuehren + UEBERZIEH_GEBUEHR;
```

Mehrfachanweisungen

- Problem im vorherigen Beispiel:
  Überziehungsgebühr wird immer verlangt, auch wenn das Konto gedeckt ist.
- Lösung: Blockklammern setzen.

```
if (kontoStand >= betrag)
{
   double neuerStand = kontoStand - betrag;
   kontoStand = neuerStand;
}
else
{
   kontoStand = kontoStand - betrag;
   gebuehren  = gebuehren + UEBERZIEH_GEBUEHR;
}
```

- Nun wird die Überziehungsgebühr nur verlangt, wenn das Konto nicht gedeckt ist.

Spezielle Mehrfachverzweigung in Java

- In Java gibt es eine weitere Möglichkeit, spezielle Mehrfachverzweigungen auszudrücken.
- Die sog. **`switch`**-Anweisung funktioniert allerdings etwas anders als bedingte Anweisungen.
- Syntax:

```
switch (ausdruck)
{
  case konstante1 : anweisung1
  case konstante2 : anweisung2

    ⋮

  default : anweisungN
}
```

Spezielle Mehrfachverzweigung in Java

- Bedeutung:
  - Abhängig vom Wert des Ausdrucks `ausdruck` wird die *Sprungmarke* angesprungen, deren Konstante mit dem Wert von `ausdruck` übereinstimmt.
  - Die Konstanten und der Ausdruck müssen den selben Typ haben.
  - Die Anweisungen nach der Sprungmarke werden ausgeführt.
  - Die optionale **`default`**-Marke wird dann angesprungen, wenn keine passende Sprungmarke gefunden wird.
  - Fehlt die **`default`**-Marke und wird keine passende Sprungmarke gefunden, so wird keine Anweisung innerhalb der **`switch`**-Anweisung ausgeführt.

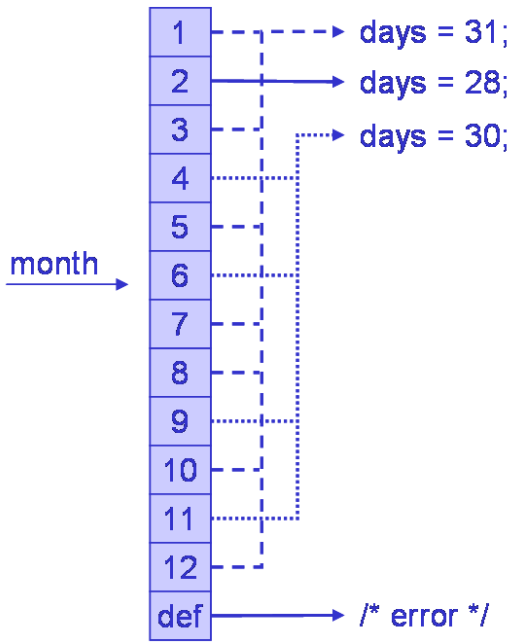
Spezielle Mehrfachverzweigung in Java

- Besonderheiten:
  - Der Ausdruck `ausdruck` darf nur vom Typ **`byte`**, **`short`**, **`int`** oder **`char`** sein.
  - Die **`case`**-Marken sollten alle verschieden sein, müssen aber nicht.
  - **Achtung:** Wird zu einer Marke gesprungen, werden alle Anweisungen hinter dieser Marke ausgeführt. Es erfolgt **keine** Unterbrechung, wenn das nächste Label erreicht wird, sondern es wird dort fortgesetzt! Dies ist eine beliebte Fehlerquelle!
  - Eine Unterbrechung kann durch die Anweisung **`break;`** erzwungen werden. Jedes **`break`** innerhalb einer **`switch`**-Anweisung verzweigt zum Ende der **`switch`**-Anweisung.
  - Nach einer Marken-Definition **`case`** muss nicht zwingend eine Anweisung stehen.

Beispiel

```
switch (month)
{
  case 1: case 3: case 5: case 7:
    case 8: case 10: case 12:
      days = 31; break;
  case 4: case 6: case 9: case 11:
      days = 30; break;
  case 2:
      if(leapYear)
      {
        days = 29;
      }
      else
      {
        days = 28;
      }
      break;
  default:
      /* error */
}
```

Sprungtabelle:



Wiederholungsanweisungen (Schleifen)

Motivation:

- Im Algorithmus *Wechselgeld 2* hatten wir eine Anweisung der Form **Solange** . . . : . . .

  Führe folgende Schritte der Reihe nach aus:

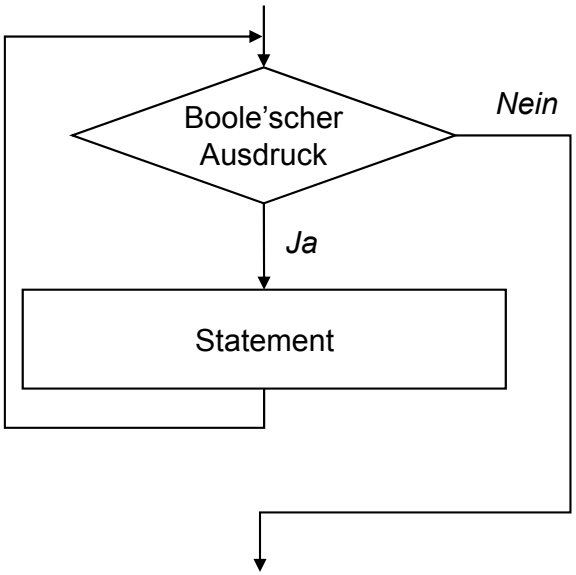
  1. Setze $w = ()$.
  2. Solange $r < 100$: Führe jeweils (wahlweise) einen der folgenden Schritte aus:
     1. Falls die letzte Ziffer von r eine 2,4,7 oder 9 ist, dann erhöhe r um 1 und nimm 1 zu w hinzu.
     2. Falls die letzte Ziffer von r eine 1,2,3,6,7 oder 8 ist, dann erhöhe r um 2 und nimm 2 zu w hinzu.
     3. Falls die letzte Ziffer von r eine 0,1,2,3,4 oder 5 ist, dann erhöhe r um 5 und nimm 5 zu w hinzu.

Wiederholungsanweisungen (Schleifen)

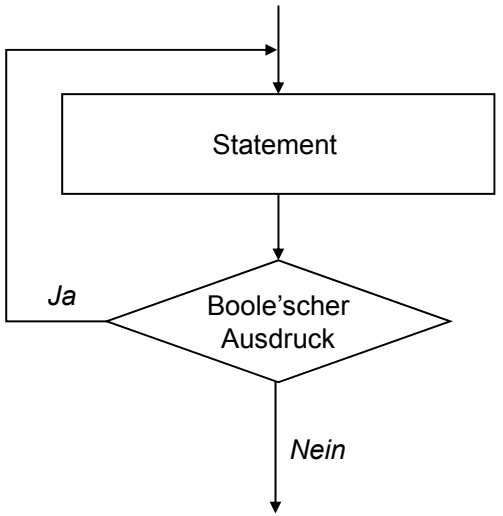
- Dabei handelt es sich um eine sog. *bedingte Wiederholungsanweisung (Schleife)*.
- Allgemeine Form:
  `WHILE` `<Bedingung>` **`DO`** `<Anweisung>` **`ENDDO`**
- `<Bedingung>` heißt Schleifenbedingung, `<Anweisung>` heißt Schleifenrumpf und kann natürlich auch wieder ein Block sein, also aus mehreren Anweisungen bestehen.
- Variante:
  `DO` `<Anweisung>` **`WHILE`** `<Bedingung>` **`ENDDO`**

Kontrollflussdiagramm für bedingte Wiederholungsanweisungen

`WHILE`-Schleife

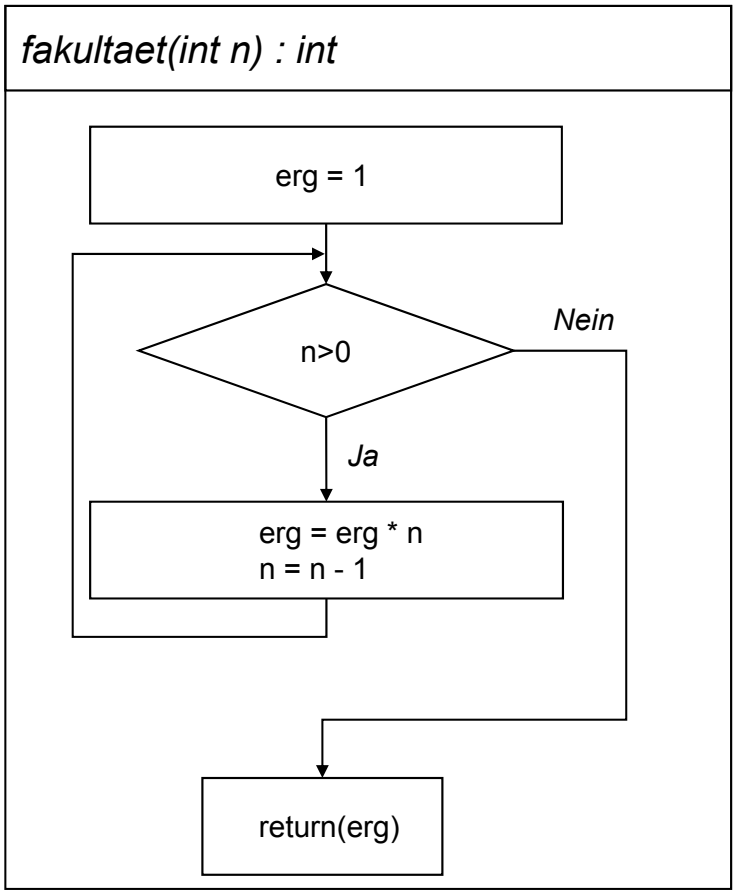


`DO`-Schleife



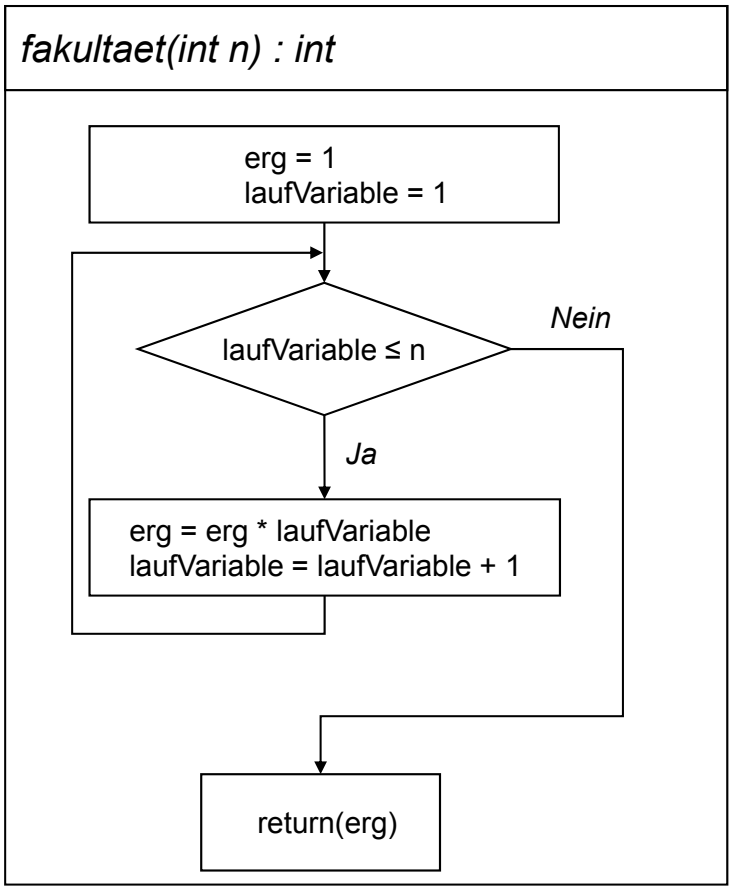
Wiederholungsanweisungen (Schleifen)

Beispiel: Fakultätsfunktion (nicht rekursiv) mit bedingter Wiederholungsanweisung



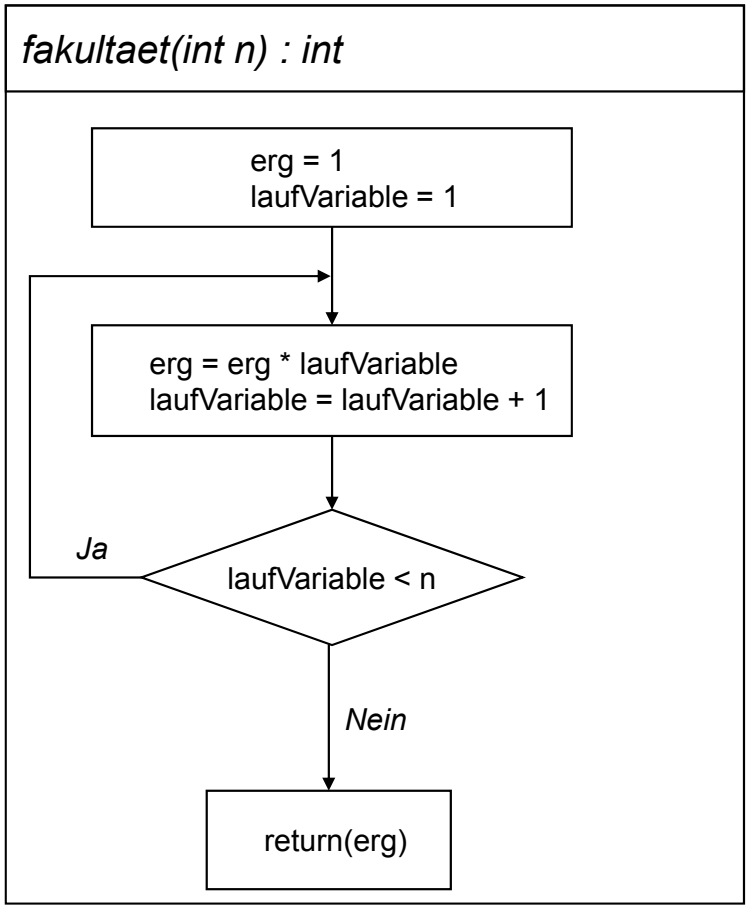
Wiederholungsanweisungen (Schleifen)

Variante: Mitzählen der durchgeführten Schritte



Wiederholungsanweisungen (Schleifen)

Variante: DO-Schleife



Wiederholungsanweisungen (Schleifen)

- Die Laufvariable (Zählen der Schritte) wurde in den letzten Beispielen zur Berechnung benutzt.
- Äquivalent zu den Formulierungen der Algorithmen zur Berechnung der Fakultätsfunktion durch die **`while`**- oder **`do`**-Schleife kann man das Ergebnis `erg` durch folgende Anweisungsfolge berechnen:

```
erg := 1;
erg := 1 * erg;
erg := 2 * erg;
⋮
erg := n * erg;
```

Wiederholungsanweisungen (Schleifen)

- Diese Folge von Zuweisungen könnte man wie folgt imperativ notieren: **Führe für** *i* = 1,. . ., *n* **nacheinander aus** : `erg := i * erg`.
- Dabei handelt es sich um eine sog. *gezählte Wiederholungsanweisung (Schleife)* (auch *Laufanweisung*).
- Allgemeine Form:
  `FOR` `<Zaehler>` **`FROM`** `<Startwert>` **`TO`** `<Endwert>`
  `BY` `<Schrittweite>` **`DO`** `<Anweisung>` **`ENDDO`**
- Analog zur bedingten Schleife heißt `<Anweisung>` Schleifenrumpf und kann wiederum aus mehreren Anweisungen (bzw. einem Block) bestehen.

Bedingte Schleifen in Java

- Java kennt mehrere Arten von bedingten Schleifen.
- Schleifen mit dem Schlüsselwort **`while`**:
  - Die klassische While-Schleife:

```
while (<Bedingung>)
  <Anweisung>
```

  - Die Do-While-Schleife:

```
do
  <Anweisung>
while (<Bedingung>);
```

- Dabei bezeichnet `<Bedingung>` ein Ausdruck vom Typ **`boolean`** und `<Anweisung>` ist entweder eine einzelne Anweisung, oder ein Block mit mehreren Anweisungen.
- Unterschied: `<Anweisung>` wird vor bzw. nach der Überprüfung von `<Bedingung>` ausgeführt.
- Hat `<Bedingung>` den Wert **`false`**, wird die Schleife verlassen.

Bedingte Schleifen in Java

Beispiel (Fakultät):

```
public static int fakultaet02(int n)
{
  int erg = 1;
  while (n > 0)
  {
    erg = erg * n;
    n--;
  }
  return erg;
}
```

Bedingte Schleifen in Java

Variante: Mitzählen der durchgeführten Schritte

```
public static int fakultaet03(int n)
{
  int erg = 1;
  int laufVariable = 1;
  while (laufVariable <= n)
  {
    erg = erg * laufVariable;
    laufVariable++;
  }
  return erg;
}
```

Bedingte Schleifen in Java

Variante: DO-Schleife

```
public static int fakultaet04(int n)
{
  int erg = 1;
  int laufVariable = 1;
  do
  {
    erg = erg * laufVariable;
    laufVariable++;
  }
  while (laufVariable < n);
  return erg;
}
```

Bedingte Schleifen in Java

- Eine weitere bedingte Schleife kann in Java mit dem Schlüsselwort **`for`** definiert werden:

```
for (<Initialisierung>; <Bedingung>; <Update>)
  <Anweisung>
```

- Alle drei Bestandteile im Schleifenkopf sind Ausdrücke (nur `<Bedingung>` muss vom Typ **`boolean`** sein).
- Vorsicht: Dieses Konstrukt ist keine klassische gezählte Schleife (auch wenn es **`for`**-Schleife genannt wird).

Bedingte Schleifen in Java

- Die Bestandteile im Einzelnen
  - Der Ausdruck `<Initialisierung>`
    - wird einmal vor dem Start der Schleife aufgerufen
    - darf Variablendeklarationen mit Initialisierung enthalten (um einen Zähler zu erzeugen); diese Variable ist nur im Schleifenkopf und innerhalb der Schleife (`<Anweisung>`) sichtbar
    - darf auch fehlen
  - Der Ausdruck `<Bedingung>`
    - ist ähnlich wie bei den While-Konstrukten die Testbedingung
    - wird zu Beginn jedes Schleifendurchlaufs überprüft
    - die Anweisung(en) `<Anweisung>` wird (werden) nur ausgeführt, wenn der Ausdruck `<Bedingung>` den Wert **`true`** hat
    - kann fehlen (gleichbedeutend mit dem Ausdruck **`true`**)
  - Der Ausdruck `<Update>`
    - verändert üblicherweise den Schleifenzähler (falls vorhanden)
    - wird am Ende jedes Schleifendurchlaufs ausgewertet
    - kann fehlen

Laufanweisungen in Java

- Eine gezählte Schleife wird in Java wie folgt mit Hilfe der **`for`**-Schleife notiert:

```
for (<Zaehler>=<Startwert>;
     <Zaehler> <= <Endwert>;
     <Zaehler> = <Zaehler> + <Schrittweite>)
  <Anweisung>
```

- Beispiel (Fakultät):

```
public static int fakultaet05(int n)
{
  int erg = 1;
  for(int i=1; i<=n; i++)
  {
    erg = erg * i;
  }
  return erg;
}
```

Sprungbefehle: `break` und `continue`

- In Java gibt es Möglichkeiten, die normale Auswertungsreihenfolge innerhalb einer **`do`**-, **`while`**- oder **`for`**-Schleife zu verändern.
- Der Befehl **`break`** beendet die Schleife sofort. Das Programm wird mit der ersten Anweisung nach der Schleife fortgesetzt.
- Der Befehl **`continue`** beendet die aktuelle Iteration und beginnt mit der nächsten Iteration, d.h. es wird an den Beginn des Schleifenrumpfes "gesprungen".
- Sind mehrere Schleifen ineinander geschachtelt, so gilt der **`break`**- bzw. **`continue`**-Befehl nur für die aktuelle (innerste) Schleife.

Sprungbefehle: `break` und `continue`

- Mit einem *Sprungbefehl* kann man an eine beliebige Stelle in einem Programm “springen”.
- Die Befehle **`break`** und **`continue`** können in Java auch für (eingeschränkte) Sprungbefehle benutzt werden.
- Der Befehl **`break <label>;`** bzw. **`continue <label>;`** muss in einem Block stehen, vor dem die Marke **`<label>`** vereinbart ist. Er bewirkt einen Sprung an das Ende dieses Anweisungsblocks.

Sprungbefehle: `break` und `continue`

- Beispiel:

```
int n = 0;
loop1:
for (int i=1; i<=10, i++)
{
  for (int j=1, j<=10, j++)
  {
    n = n + i * j;
    break loop1;
  }
}
System.out.print(n); // n = 1
```

- Der Befehl **`break`** `loop1;` erzwingt den Sprung an das Ende der äußeren **`for`**-Schleife.

Sprungbefehle: `break` und `continue`

Anmerkung:

Durch Sprungbefehle (vor allem bei Verwendung von Labeln) wird ein Programm leicht unübersichtlich und Korrektheitsüberprüfung wird schwierig. Wenn möglich, sollten Sprungbefehle vermieden werden.

Unerreichbare Befehle

- *Unerreichbare Befehle* sind Anweisungen, die (u.a.)
  - hinter einer **`break`**- oder **`continue`**-Anweisung liegen, die ohne Bedingung angesprungen werden,
  - in Schleifen stehen, deren Testausdruck zur Compile-Zeit **`false`** ist.
- Solche unerreichbaren Anweisungen sind in Java nicht erlaubt, sie werden vom Compiler nicht akzeptiert.
- Einzige Ausnahme sind Anweisungen, die hinter der Klausel

  `if (false)`

  stehen. Diese Anweisungen werden von den meisten Compilern nicht in den Bytecode übernommen, sondern einfach entfernt. Man spricht von *bedingtem Kompilieren*.

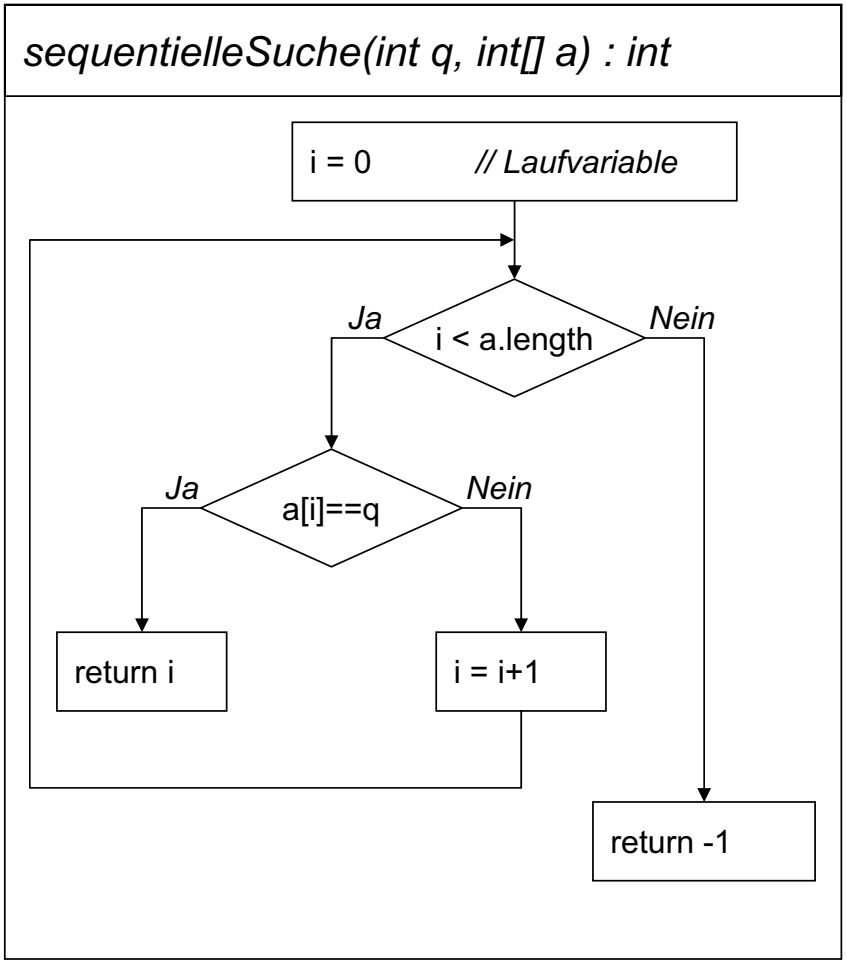
Überblick

Beispiel: Suche in einem Array

- Aufgabe:
  - Sei A ein Array der Länge n mit Werten aus $\mathbb{N}$ und $q \in \mathbb{N}$.
  - Keiner der Werte kommt mehrfach vor.
  - Suche q in A und gib, falls die Suche erfolgreich ist, die Position i mit $A[i] = q$ aus. Falls die Suche erfolglos ist, gib den Wert -1 aus.
- Einfachste Lösung: *Sequentielle Suche*
  - Durchlaufe A von Position $i = 0, \ldots, n-1$.
  - Falls $A[i] = q$, gib i aus und beende den Durchlauf.
  - Falls q nicht gefunden wird, gib -1 aus.

Beispiel: Suche in einem Array

- Algorithmus:



Beispiel: Suche in einem Array

- Java-Methode:

```
public static int sequentielleSuche(int q, int[] a)
{
  for(int i = 0; i < a.length; i++)
  {
    if(a[i]==q)
    {
      return i;
    }
  }
  return -1;
}
```

Beispiel: Suche in einem Array

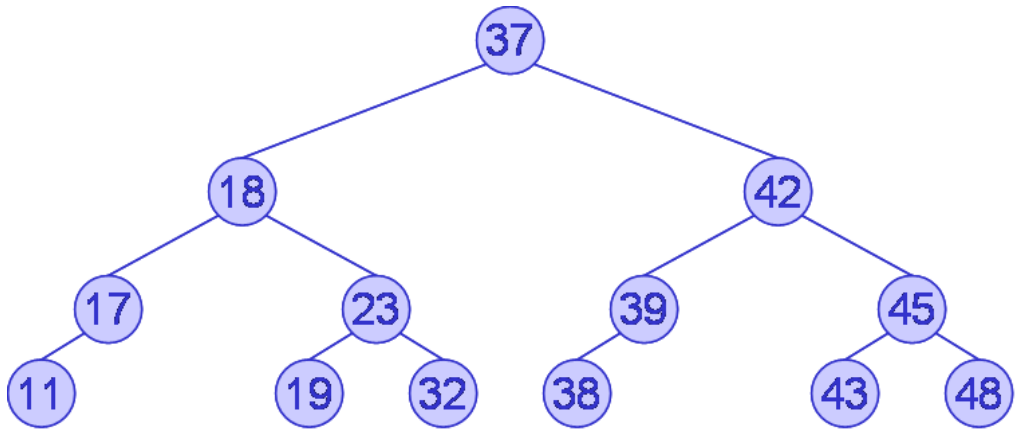
- Anzahl der Vergleiche (= Analyse der Laufzeit):
  - Erfolgreiche Suche: n Vergleiche maximal, $n/2$ im Durchschnitt.
  - Erfolglose Suche: n Vergleiche.
- Falls das Array sortiert ist, funktioniert auch folgende Lösung: *Binäre Suche*
  - Betrachte den Eintrag $A[i]$ mit $i = n/2$ in der Mitte des Arrays
    - Falls $A[i] = q$, gib i aus und beende die Suche.
    - Falls $A[i] > q$, suche in der linken Hälfte von A weiter.
    - Falls $A[i] < q$, suche in der rechten Hälfte von A weiter.
  - In der jeweiligen Hälfte wird ebenfalls mit binärer Suche gesucht.
  - Falls die neue Hälfte des Arrays leer ist, gib -1 aus.

Beispiel: Suche in einem Array

Beispiel: A:

11	17	18	19	23	32	37	38	39	42	43	45	48

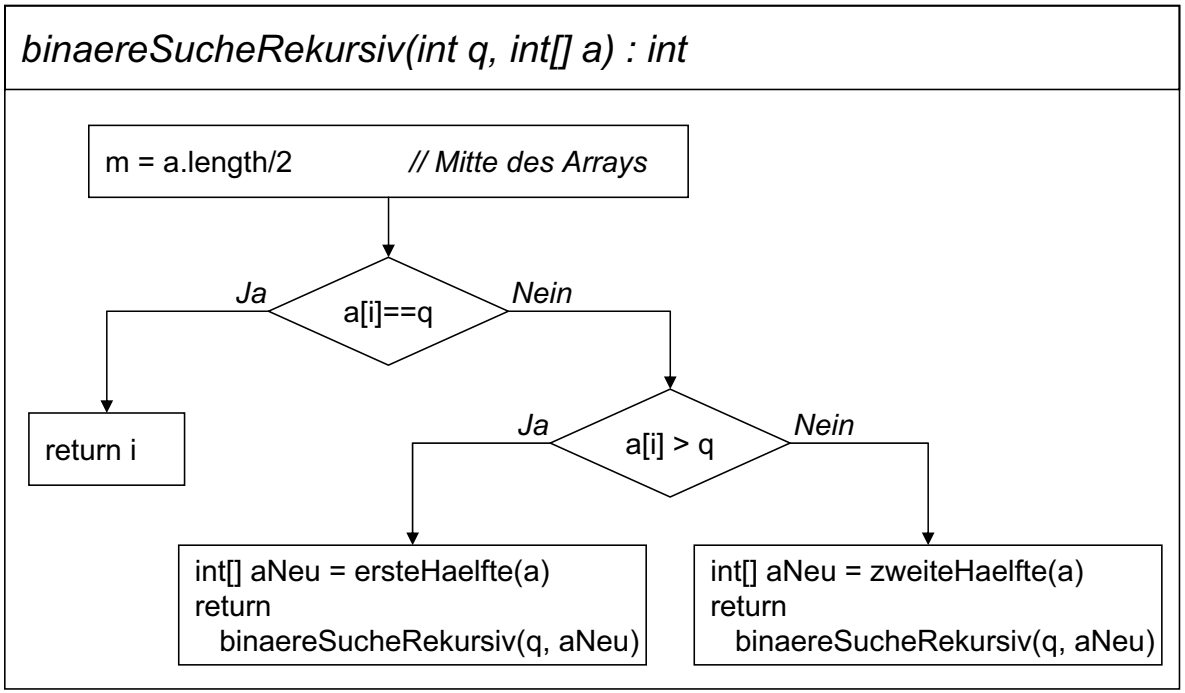
- Entscheidungsbaum:



- Analyse der Laufzeit: Maximale Anzahl von Vergleichen entspricht Höhe h des Entscheidungsbaums: $h = \lceil \log_2(n+1) \rceil$.
- Vergleich der Verfahren:
  - $n = 1.000$:
    Sequentielle Suche: 1.000 Vergleiche – Binäre Suche: 10 Vergleiche
  - $n = 1.000.000$:
    Sequentielle Suche: 1.000.000 Vergleiche – Binäre Suche: 20 Vergleiche

Beispiel: Suche in einem Array

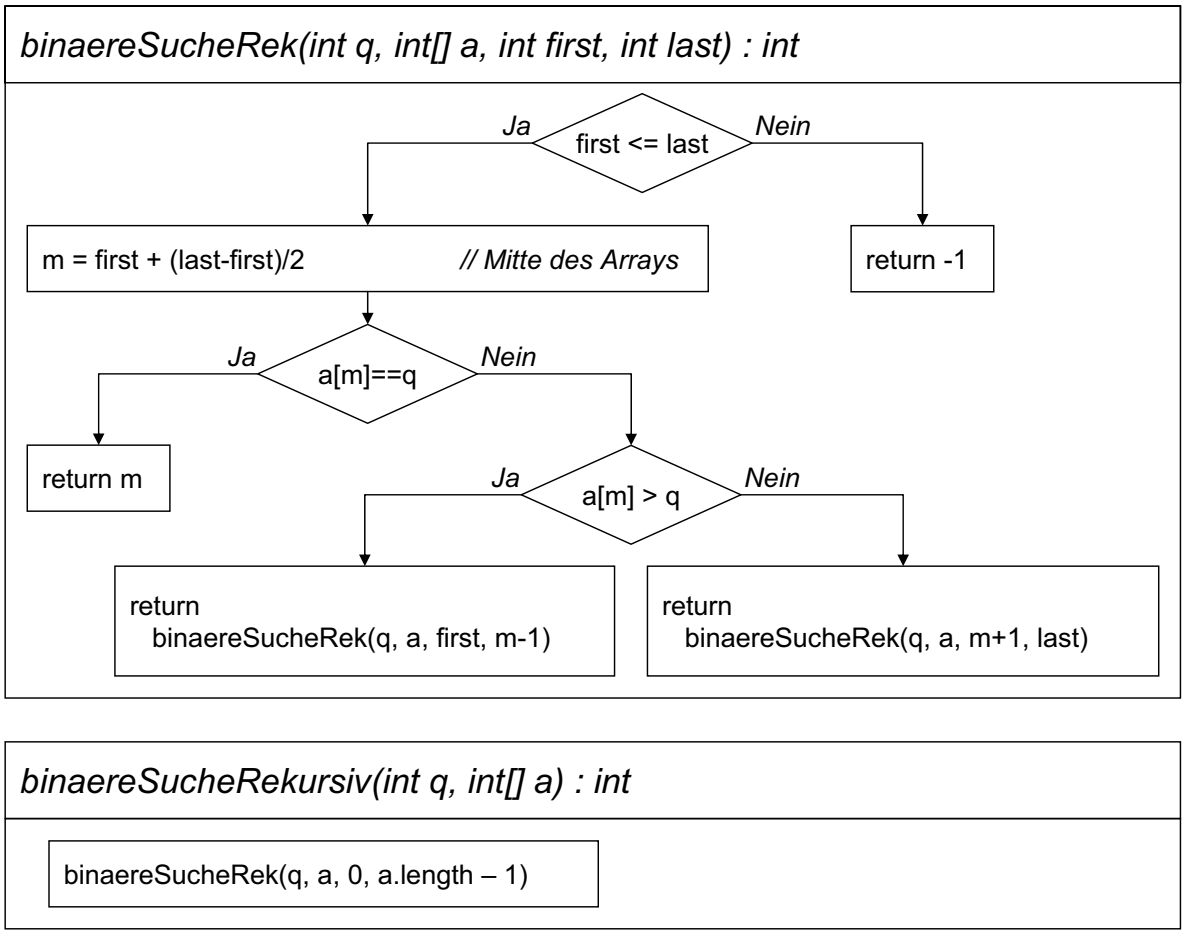
Rekursiver Algorithmus:



Probleme?

Beispiel: Suche in einem Array

Alternativer rekursiver Algorithmus mit nur einem Array:



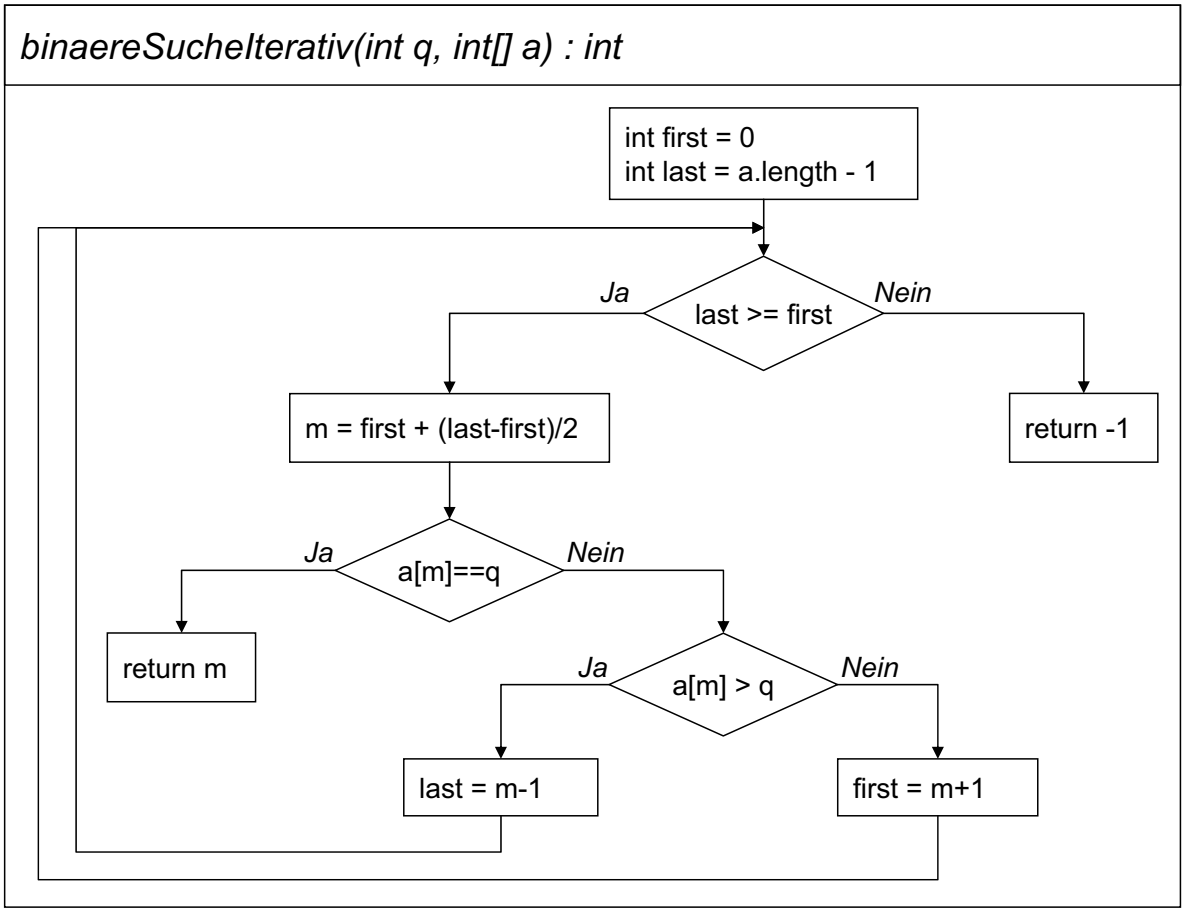
Beispiel: Suche in einem Array

Alternativer rekursiver Algorithmus in Java:

```
public static int binaereSucheRek (int q, int[] a, int first, int last)
{
  if (first > last)
  {
    return -1;  // leeres Array
  }
  int m = first + (last-first) / 2;
  if (q == a[m])
  {
    return m;   // q gefunden
  }
  else if (a[m] > q)
  {
    return binaereSucheRek(q, a, first, m-1);
  }
  else /* a[m] < q */
  {
    return binaereSucheRek(q, a, m+1, last);
  }
}
public static int binaereSucheRekursiv (int q, int[] a)
{
  return binaereSucheRek (q, a, 0, a.length - 1);
}
```

Beispiel: Suche in einem Array

Alternativer *iterativer* Algorithmus mit nur einem Array:



Beispiel: Suche in einem Array

Iterativer Algorithmus in Java:

```
public static int binaereSucheIterativ (int q, int[] a)
{
  int first = 0;
  int last = a.length - 1;
  while (last >= first)
  {
    int m = (last + first) / 2;
    if (q == a[m])
    {
      return m;
    }
    else if (q < a[m])
    {
      last = m - 1;
    }
    else /*q > a[m]*/
    {
      first = m + 1;
    }
  }
  return -1;  // not found
}
```

Zusammenfassung

Sie kennen jetzt die Grundkonzepte imperativer Programmierung:

- Grunddatentypen und ihre typischen Operationen als elementare Bestandteile der meisten Programmiersprachen,
- das Konzept von (lokalen und globalen) Variablen und Konstanten (und deren Unterschied) zur sequentiellen Verarbeitung von Information in imperativen Programmen,
- das Konzept von (sequentiellen) Anweisungen,
- Blöcke als Zusammenfassung mehrerer Anweisungen,

...

Zusammenfassung

...

- die Gültigkeitsbereiche verschiedener Arten von Variablen,
- Arrays (Reihungen) als grundlegende Datenstruktur zur Verwaltung einer Menge gleichartiger Werte im imperativen Programmierparadigma,
- Methoden (Prozeduren) als Mittel zur Abstrahierung von Algorithmen und einzelnen Berechnungsschritten sowie (durch die Verwendung formaler Parameter) von den konkreten Daten,
- typische imperative Kontrollstrukturen zur Implementierung von bedingten, kontrollierten, iterierten Anweisungen

und können somit im Prinzip einen Algorithmus imperativ in Java implementieren.

Abschnitt 5: Speicherverwaltung

Überblick

5. Speicherverwaltung

5.1 Umgebung und Speicherstrukturen

5.2 Speicherverwaltung in Java

Umgebung

- Der Wert eines Ausdruckes wird zur Laufzeit bestimmt.
- Wie wir gesehen haben, können in einem Java-Programm Ausdrücke mit Variablen vorkommen.
- Der Wert eines Ausdrucks mit Variablen hängt insbesondere von der Belegung der Variablen ab (siehe *Substitution*).
- Um während der Laufzeit Ausdrücke mit Variablen auszuwerten, muss also eine Menge von Substitutionen, d.h. Bindungen von Variablennamen an ihre Werte, verwaltet werden.
- Die Bindungen von Variablen an Werte werden im Lauf eines Programmes angesammelt und geben einer Variablen an einer bestimmten Stelle des Programmes (d.h. in einer bestimmten *Umgebung*) eine bestimmte *Bedeutung*.
- Diese Menge von Bindungen muss in einer geeigneten Datenstruktur gespeichert werden.
- Anforderungen an diese Datenstruktur werden durch ein *Umgebungsmodell* beschrieben.

Abbildung von Schachtelungen in ein Umgebungsmodell

- Was muss diese Datenstruktur berücksichtigen?
  - Im imperativen Paradigma sind Blöcke und ihre Schachtelung das wichtigste Strukturierungselement. (Klassen, Methoden, Kontrollstrukturen bilden ja stets auch einen Block.)
  - Ein Block in einem Block führt Namensbindungen ein, die zusätzlich zu den Bindungen des äußeren Blocks gelten.
  - Nach Verlassen des inneren Blocks gelten wieder nur noch jene Bindungen, die im äußeren Block bereits vor Betreten des inneren Blockes galten.
  - Die Modellierung der Umgebung sollte also dieser Anforderung Rechnung tragen:
    1. Neue Bindungen werden oft eingefügt und alte Bindungen werden oft wieder gelöscht.
    2. Die Bindungen, die zuletzt hinzukamen, werden als erste wieder entfernt.
  - Zugriffs-Prinzip: LIFO (last-in-first-out)
    Die Datenstruktur sollte also dynamisch sein und dieses LIFO-Prinzip unterstützen!

Ein einfaches Umgebungsmodell

- Welche Datenstruktur für die Modellierung der Umgebung (Verwaltung von gleichartigen Daten) steht uns bisher zur Verfügung?
  Antwort: Arrays.
- Nun stellt sich die Frage, ob Arrays den o.g. Anforderungen für die Modellierung der Umgebung gerecht werden.
- Offenbar unterstützen Arrays den wahlfreien Zugriff also insbesondere auch einen Zugriff nach dem LIFO-Prinzip (Achtung, Arrays sind aber keine klassischen LIFO-Strukturen – Warum?).
  `a[i]` ermöglicht den Zugriff auf die *i*-te Komponente eines Arrays `a`.
- Aber: Arrays sind nicht dynamisch in dem Sinn, dass sie dynamisch wachsen und schrumpfen.
  Nach der Initialisierung ist die Anzahl der Elemente, die ein Array `a` aufnehmen kann, auf `a.length` beschränkt.
- Folge: wir benötigen eine andere Datenstruktur.

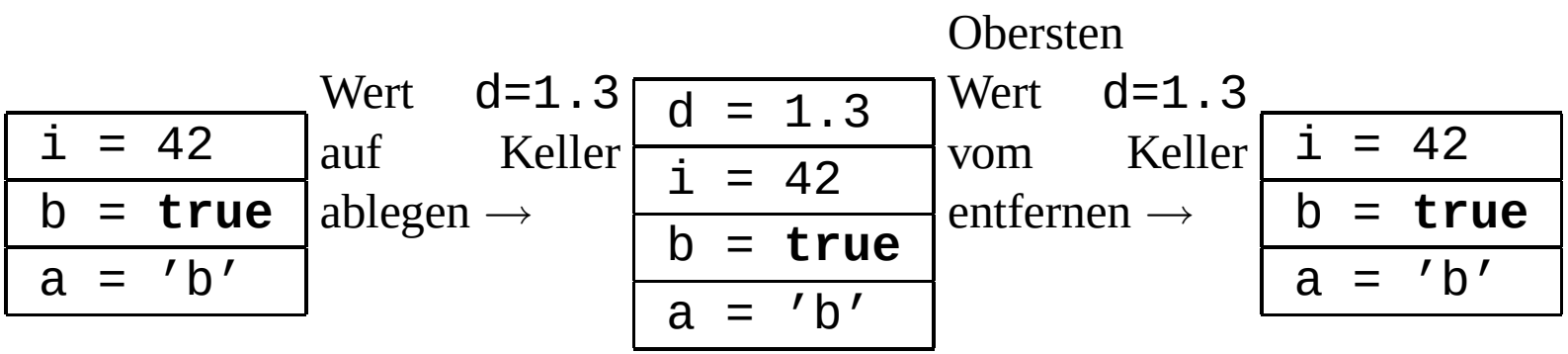
Ein verfeinertes Umgebungsmodell

- Allgemein sind *Listen* Datenstrukturen, die dynamisch wachsen und schrumpfen.
- Zwei interessante Listen sind
  - Die Schlange (*Queue*) mit einem Zugriff nach dem FIFO (first-in-first-out) Prinzip (ähnlich einer fairen Warteschlange).
  - Der Keller (*Stack*) mit dem Zugriff nach dem LIFO Prinzip.
- Die meisten imperativen Programmiersprachen benutzen zur Verwaltung der Variablensubstitutionen einen Keller.

Keller als typische LIFO-Struktur

Der Keller (auch *Stapel*) verwirklicht als Datenstruktur das LIFO-Prinzip:

- Was man ablegt, legt man wie auf einem Stapel obendrauf.
- Entfernen kann man nur, was auf dem Stapel zuoberst liegt.
- Nach dem Entfernen des obersten Items (Eintrag, Wert) liegen genau die Items vor, die vor dem Ablegen des eben entfernten Items vorlagen.
- Im Kontext der imperativen Speicherverwaltung verwendet man meist eine Spezialform des Kellers, bei der man tiefer liegende Einträge ablesen (und Werte verändern) kann.



Dynamische Speicherverwaltung

Um Werte in einem Keller (Stapel) verwalten zu können, nimmt man an, dass die Werte alle gleichviel Speicherplatz benötigen. Dinge unterschiedlicher Größe kann man ja auch nicht gut stapeln.

Oft muß man Werte sehr unterschiedlicher Größe verwalten. Hierfür braucht man eine dynamische Speicherplatzverwaltung (*dynamic storage allocation*), durch die Werte unterschiedlicher Größe in einem gemeinsamen Speicherbereich abgelegt werden können.

Halde

Ein Pool verfügbaren Speicherplatzes für dynamischen Zugriff heißt Halde, engl. *heap*.

Achtung:

Mit *heap* bezeichnet man im Zusammenhang mit effizienten Algorithmen auch eine Prioritätsliste (*priority queue*). Bitte nicht verwechseln!

Überblick

Blick hinter die Kulissen

Wir wollen im folgenden ein wenig die interne Speicherverwaltung anschauen. Wir bleiben hier aber informell und vereinfachend und betrachten die Zusammenhänge nur, soweit wir sie benötigen, um Phänomene zu verstehen, die uns auf der Ebene der Programmierung beschäftigen können.

Ein tieferes Verständnis der hier behandelten Zusammenhänge kann in anderen Vorlesungen erworben werden.

Keller für lokale Variablen

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2    int a = 5;
3    int b = 0;
4    for(int i = 0; i < a; i++)
5    {
6      int c = a-i;
7      b += c;
8    }
9    boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 2

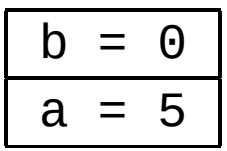
a = 5

Keller für lokale Variablen

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2    int a = 5;
3    int b = 0;
4    for(int i = 0; i < a; i++)
5    {
6      int c = a-i;
7      b += c;
8    }
9    boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 3

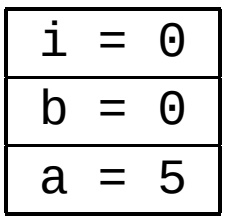


Keller für lokale Variablen

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2    int a = 5;
3    int b = 0;
4    for(int i = 0; i < a; i++)
5    {
6      int c = a-i;
7      b += c;
8    }
9    boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 4



Keller für lokale Variablen

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2    int a = 5;
3    int b = 0;
4    for(int i = 0; i < a; i++)
5    {
6      int c = a-i;
7      b += c;
8    }
9    boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 6

c = 5
i = 0
b = 0
a = 5

Keller für lokale Variablen

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2    int a = 5;
3    int b = 0;
4    for(int i = 0; i < a; i++)
5    {
6      int c = a-i;
7      b += c;
8    }
9    boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 7

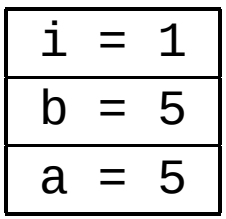
c = 5
i = 0
b = 5
a = 5

Keller für lokale Variablen

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2    int a = 5;
3    int b = 0;
4    for(int i = 0; i < a; i++)
5    {
6      int c = a-i;
7      b += c;
8    }
9    boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 4′



Keller für lokale Variablen

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2    int a = 5;
3    int b = 0;
4    for(int i = 0; i < a; i++)
5    {
6      int c = a-i;
7      b += c;
8    }
9    boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 6′

c = 4
i = 1
b = 5
a = 5

Keller für lokale Variablen

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2    int a = 5;
3    int b = 0;
4    for(int i = 0; i < a; i++)
5    {
6      int c = a-i;
7      b += c;
8    }
9    boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 7′

c = 4
i = 1
b = 9
a = 5

Keller für lokale Variablen

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2    int a = 5;
3    int b = 0;
4    for(int i = 0; i < a; i++)
5    {
6      int c = a-i;
7      b += c;
8    }
9    boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 4''

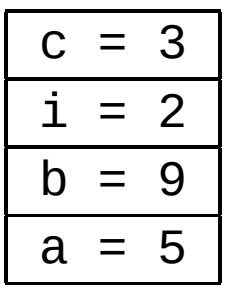
i = 2
b = 9
a = 5

Keller für lokale Variablen

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2    int a = 5;
3    int b = 0;
4    for(int i = 0; i < a; i++)
5    {
6      int c = a-i;
7      b += c;
8    }
9    boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile `6''`



Keller für lokale Variablen

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2    int a = 5;
3    int b = 0;
4    for(int i = 0; i < a; i++)
5    {
6      int c = a-i;
7      b += c;
8    }
9    boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 7''

c = 3
i = 2
b = 12
a = 5

Keller für lokale Variablen

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2    int a = 5;
3    int b = 0;
4    for(int i = 0; i < a; i++)
5    {
6      int c = a-i;
7      b += c;
8    }
9    boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile `4'''`

i = 3
b = 12
a = 5

Keller für lokale Variablen

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2    int a = 5;
3    int b = 0;
4    for(int i = 0; i < a; i++)
5    {
6      int c = a-i;
7      b += c;
8    }
9    boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile `6'''`

c = 2
i = 3
b = 12
a = 5

Keller für lokale Variablen

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2    int a = 5;
3    int b = 0;
4    for(int i = 0; i < a; i++)
5    {
6      int c = a-i;
7      b += c;
8    }
9    boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile `7'''`

c = 2
i = 3
b = 14
a = 5

Keller für lokale Variablen

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2    int a = 5;
3    int b = 0;
4    for(int i = 0; i < a; i++)
5    {
6      int c = a-i;
7      b += c;
8    }
9    boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile `4''''`

i = 4
b = 14
a = 5

Keller für lokale Variablen

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2    int a = 5;
3    int b = 0;
4    for(int i = 0; i < a; i++)
5    {
6      int c = a-i;
7      b += c;
8    }
9    boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile `6''''`

c = 1
i = 4
b = 14
a = 5

Keller für lokale Variablen

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2    int a = 5;
3    int b = 0;
4    for(int i = 0; i < a; i++)
5    {
6      int c = a-i;
7      b += c;
8    }
9    boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile `7''''`

c = 1
i = 4
b = 15
a = 5

Keller für lokale Variablen

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2    int a = 5;
3    int b = 0;
4    for(int i = 0; i < a; i++)
5    {
6      int c = a-i;
7      b += c;
8    }
9    boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile `4'''''`

i = 5
b = 15
a = 5

Keller für lokale Variablen

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2    int a = 5;
3    int b = 0;
4    for(int i = 0; i < a; i++)
5    {
6      int c = a-i;
7      b += c;
8    }
9    boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 8

b = 15
a = 5

Keller für lokale Variablen

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2    int a = 5;
3    int b = 0;
4    for(int i = 0; i < a; i++)
5    {
6      int c = a-i;
7      b += c;
8    }
9    boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 9

d = **true**
b = 15
a = 5

Globale Variablen: *Constant Pool*

Die globalen Variablen sind (zunächst) in jedem Block sichtbar, müssen also auch im Keller gesondert behandelt werden.

Es gibt daher für die globalen Variablen einen eigenen Bereich, auf den von anderen Bereichen zugegriffen werden kann: den *Constant Pool*.

Frames

- Die lokalen Variablen in einem Block sind in einer anderen Methode nicht sichtbar, auch wenn diese andere Methode im gleichen Block *aufgerufen* wird.
- Deshalb werden die lokalen Variablen innerhalb des Kellers in sog. Frames angeordnet.
- Ein Frame wird bei einem Methodenaufruf erzeugt und beinhaltet
  - die lokalen Variablen einer Methode,
  - die übergebenen (aktuellen) Parameter,
  - den Rückgabewert (und evtl. Ausnahmen),
  - einen *Operand Stack* für die Methodenausführung,
  - einen Verweis auf den Constant Pool.

Keller vs. Halde

- Der Keller sieht für jeden Eintrag nur begrenzt viel Speicherplatz vor.
- Die Werte von primitiven Typen können direkt im Keller abgelegt werden.
- Andere Typen, die Objekte (wie z.B. Strings und Arrays), können sehr viel mehr Speicherplatz in Anspruch nehmen.
- Objekte werden daher in der Halde (Heap) gespeichert, im Keller wird (als Wert des entsprechenden Bezeichners) eine Referenz (die Speicheradresse im Heap) abgelegt.

Veranschaulichung: Stack – Heap

```
char a = 'b';
String gruss1 = "Hi";
String gruss2 = "Hello";
String[] gruesse = {gruss1, gruss2};
int[] zahlen = {1, 2, 3};
boolean b = true;
int i = 42;
```

Stack:

i = 42
b = **true**
zahlen = <adr4>
gruesse = <adr3>
gruss2 = <adr2>
gruss1 = <adr1>
a = 'b'

Heap:

<adr1> : "Hi" <adr2> : "Hallo" <adr3> : { <adr1> , <adr2> } <adr4> : {1, 2, 3 }

Nochmal: call-by-value

```
1   public class Exchange
2   {
3     public static void swap(int i, int j)
4     {
5       int c = i;
6       i = j;
7       j = c;
8     }
9
10    public static void main(String[] args)
11    {
12      int x = 1;
13      int y = 2;
14      swap(x,y);
15    }
16  }
```

main-Frame nach Zeile 13

y = 2
x = 1
args = <adr1>

Nochmal: call-by-value

```
1  public class Exchange
2  {
3    public static void swap(int i, int j)
4    {
5      int c = i;
6      i = j;
7      j = c;
8    }
9
10   public static void main(String[] args)
11   {
12     int x = 1;
13     int y = 2;
14     swap(x,y);
15   }
16 }
```

`main`-Frame nach Zeile 13

y = 2
x = 1
args = <adr1>

`swap`-Frame nach Zeile 3

j = 2
i = 1

Nochmal: call-by-value

```
1   public class Exchange
2   {
3     public static void swap(int i, int j)
4     {
5       int c = i;
6       i = j;
7       j = c;
8     }
9
10    public static void main(String[] args)
11    {
12      int x = 1;
13      int y = 2;
14      swap(x,y);
15    }
16  }
```

`main`-Frame nach Zeile 13

y = 2
x = 1
args = <adr1>

`swap`-Frame nach Zeile 5

c = 1
j = 2
i = 1

Nochmal: call-by-value

```
1   public class Exchange
2   {
3     public static void swap(int i, int j)
4     {
5       int c = i;
6       i = j;
7       j = c;
8     }
9
10    public static void main(String[] args)
11    {
12      int x = 1;
13      int y = 2;
14      swap(x,y);
15    }
16  }
```

`main`-Frame nach Zeile 13

y = 2
x = 1
args = <adr1>

`swap`-Frame nach Zeile 6

c = 1
j = 2
i = 2

Nochmal: call-by-value

```
1   public class Exchange
2   {
3     public static void swap(int i, int j)
4     {
5       int c = i;
6       i = j;
7       j = c;
8     }
9
10    public static void main(String[] args)
11    {
12      int x = 1;
13      int y = 2;
14      swap(x,y);
15    }
16  }
```

`main`-Frame nach Zeile 13

y = 2
x = 1
args = <adr1>

`swap`-Frame nach Zeile 7

c = 1
j = 1
i = 2

Nochmal: call-by-value

```
1   public class Exchange
2   {
3     public static void swap(int i, int j)
4     {
5       int c = i;
6       i = j;
7       j = c;
8     }
9
10    public static void main(String[] args)
11    {
12      int x = 1;
13      int y = 2;
14      swap(x,y);
15    }
16  }
```

`main`-Frame nach Zeile 14

y = 2
x = 1
args = <adr1>

call-by-value bei Objekten: Der call-by-reference-Effekt

```
1  public static void changeValues(int[] zahlen, int index, int wert)
2  {
3    zahlen[index] = wert;
4  }
5
6  public static void main(String[] args)
7  {
8    int[] werte = {0, 1, 2};
9    changeValues(werte, 1, 3);
10 }
```

`main`-Frame nach Zeile 6

args = <adr1>

Heap nach Zeile 6

<adr1> : { }

call-by-value bei Objekten: Der call-by-reference-Effekt

```
1  public static void changeValues(int[] zahlen, int index, int wert)
2  {
3    zahlen[index] = wert;
4  }
5
6  public static void main(String[] args)
7  {
8    int[] werte = {0, 1, 2};
9    changeValues(werte, 1, 3);
10 }
```

`main`-Frame nach Zeile 8

werte = <adr2>
args = <adr1>

Heap nach Zeile 8

<adr1> : { } <adr2> : { 0, 1, 2 }

call-by-value bei Objekten: Der call-by-reference-Effekt

```
1  public static void changeValues(int[] zahlen, int index, int wert)
2  {
3    zahlen[index] = wert;
4  }
5
6  public static void main(String[] args)
7  {
8    int[] werte = {0, 1, 2};
9    changeValues(werte, 1, 3);
10 }
```

Heap nach Zeile 8

<adr1> : { } <adr2> : { 0, 1, 2 }

main-Frame nach Zeile 8

werte = <adr2>
args = <adr1>

changeValues-Frame nach Zeile 1

wert = 3
index = 1
zahlen = <adr2>

call-by-value bei Objekten: Der call-by-reference-Effekt

```
1  public static void changeValues(int[] zahlen, int index, int wert)
2  {
3    zahlen[index] = wert;
4  }
5
6  public static void main(String[] args)
7  {
8    int[] werte = {0, 1, 2};
9    changeValues(werte, 1, 3);
10 }
```

Heap nach Zeile 3

<adr1> : { } <adr2> : { 0, 3, 2 }

main-Frame nach Zeile 8

werte = <adr2>
args = <adr1>

changeValues-Frame nach Zeile 3

wert = 3
index = 1
zahlen = <adr2>

call-by-value bei Objekten: Der call-by-reference-Effekt

```
1  public static void changeValues(int[] zahlen, int index, int wert)
2  {
3    zahlen[index] = wert;
4  }
5
6  public static void main(String[] args)
7  {
8    int[] werte = {0, 1, 2};
9    changeValues(werte, 1, 3);
10 }
```

`main`-Frame nach Zeile 9

werte = <adr2>
args = <adr1>

Heap nach Zeile 9

<adr1> : { } <adr2> : { 0, 3, 2 }

Garbage Collection

Keller: Aufräumen unnötig

Die Keller werden schon von ihrer Struktur her automatisch aufgeräumt, d.h. es gibt in Kellern keine Speicherplätze, die belegt sind, obwohl die Lebensdauer des entsprechenden Namens abgelaufen ist.

Garbage Collection

Halde: Aufräumen nötig

Für die Halde gilt das nicht: Hier wird im Laufe eines Programmes Speicherplatz zugewiesen und belegt, aber nicht automatisch freigegeben, falls die Lebensdauer eines Namens, der für den gespeicherten Wert steht, abgelaufen ist.

In vielen Sprachen muß der Programmierer dafür sorgen, dass Speicherplatz, der nicht mehr gebraucht wird, freigegeben wird (explizite Speicherplatzfreigabe – Gefahr des Speicherlecks).

In vielen modernen Sprachen (auch Java) gibt es eine automatische Speicherplatzfreigabe (Garbage Collection). Der Heap wird immer wieder durchsucht nach Adressen, auf die nicht mehr zugegriffen werden kann (da kein Name diese Adresse als Wert hat).

Problem hier: der Programmierer kann nicht oder nur sehr eingeschränkt kontrollieren, wann dieser Reinigungsprozess läuft. Das ist unter Umständen (z.B. in sekundengenauen, empfindlichen Echtzeitsystemen) nicht akzeptabel.

Abschnitt 6: Korrektheit von imperativen Programmen

Überblick

Korrektheit von Algorithmen

- Algorithmen und Programme sollten das tun, wofür sie entwickelt wurden, d.h. sie sollten *korrekt* sein.
- Problem: wie kann man nachweisen, dass ein Algorithmus bzw. ein Programm korrekt ist?
- Schon bei relativ kleinen Beispielen kann man oft nicht mehr “mit dem Auge” entscheiden, ob ein Algorithmus/Programm korrekt ist.
- Im Folgenden werden wir uns auf Java-Programme konzentrieren.
- Wir werden aber allgemeine Konzepte besprechen, die auch für andere Programmiersprachen und auch ganz abstrakt für die Algorithmenentwicklung (beispielsweise mittels Pseudo-Code) gelten.

Korrektheit von Algorithmen

Beispiel: folgendes Java-Programm soll das Quadrat einer Zahl $a \in \mathbb{N}$ berechnen.

```
public static int quadrat(int a)
{
  int y;
  int z;

  // Anfang der Berechnung
  y = 0;
  z = 0;
  while (y != a)
  {
    z = z + 2*y + 1;
    y = y + 1;
  }
  // Ende der Berechnung

  return z;
}
```

Korrektheit von Algorithmen

- Bemerkung: Als Java-Programm macht der Algorithmus wenig Sinn, aber als Programm für einen Mikroprozessor, für den die Multiplikation viel aufwendiger ist als die Addition und Multiplikation mit 2, sehr wohl!
- Es ist nicht offensichtlich, dass für alle **`int`**-Werte a das Programm wirklich den Wert a^2 berechnet.
- Tatsächlich stimmt dies nur für alle $a \geq 0$.
- Wie können wir die Korrektheit dieses Java-Programms nachweisen?

Korrektheit von Algorithmen

- Zur Erinnerung: in der funktionalen Programmierung sind Algorithmen Funktionen, die, auf die Eingabewerte angewendet, spezielle Werte zurückliefern.
- Der Beweis, dass eine Funktion eine gewisse Eigenschaft hat, ist meist (relativ) einfach zu führen (z.B. durch vollständige Induktion, wenn die Funktion rekursiv definiert ist).
- Bei imperativen Programmen ist dies i.d.R. deutlich schwieriger.
  - Wir haben keinen direkten funktionalen Zusammenhang zwischen Eingabewerten und Ausgabewerten.
  - Das Resultat kommt vielmehr durch Abarbeiten verschiedener Berechnungsschritte zustande.
  - Diese Berechnungsschritte können Seiteneffekte haben.
- Der Beweis der Korrektheit imperativer Programme ist daher nicht ganz so einfach.

Korrektheit von Algorithmen

- Zunächst sollte festgelegt werden, wie man die Aussagen, die man für ein Programm machen will, formuliert: wie formuliert man z.B., dass die Methode `quadrat` das Quadrat des Eingabewerts berechnet?
- Dazu kann man eine eigene formale Sprache definieren (ist in diesem Rahmen aber nicht nötig).
- Wir verwenden Boole'sche Java Ausdrücke, um Aussagen zu machen, z.B. `0 <= a` oder `z == a*a`.
- Zusätzlich verwenden wir als gängige Abkürzung für `!a | b` die Notation `a => b`, wobei `a` und `b` beliebige Boole'sche Ausdrücke sein können.
- `a => b` drückt die logische Implikation "wenn `a`, dann `b`" aus.
- Damit hat der Ausdruck `a => b` den Wert **`true`** gdw. `b` den Wert **`true`** oder `a` den Wert **`false`** hat.

Korrektheit von Algorithmen

- Welche Art von Aussagen über ein Programm `p` will man nun beweisen?
- Typischerweise Aussagen der Art:
  "Wenn für die Eingabewerte der Programms `p` die Bedingung `PRE` gilt, dann gilt nach Ausführung von `p` für die Ausgabewerte die Bedingung `POST`".
- Dies schreibt man meist:

  ```
  (PRE) p (POST)
  ```

- `PRE` heißt *Vorbedingung* (*Precondition*) und `POST` heißt *Nachbedingung* (*Postcondition*).
- Beispiel: Für das Programm `p = quadrat(int a)` wären entsprechende Vor- und Nachbedingungen

  ```
  (0 <= a) p (z == a*a)
  ```

Korrektheit von Algorithmen

Nun unterscheiden wir zwei Arten von Korrektheit von Programmen:

- **Partielle Korrektheit:**
  Partielle Korrektheit eines Programms p heißt: falls p auf Eingabewerte, die der Vorbedingung `PRE` genügen, angewendet wird, und falls es terminiert, dann muss anschließend die Nachbedingung `POST` gelten.
- **Totale Korrektheit:**
  Totale Korrektheit eines Programms p heißt: falls p auf Eingabewerte, die der Vorbedingung `PRE` genügen, angewendet wird, dann terminiert es und danach muss die Nachbedingung `POST` gelten.
- Beobachtung: Totale Korrektheit = Partielle Korrektheit + Terminierung

Korrektheit von Algorithmen

- Wie kann man nun die Korrektheit imperativer Programme überprüfen?
- Testen: Man kann z.B. für alle möglichen Eingabewerte testen, ob das Programm die richtigen Ergebniswerte liefert.
- Offensichtlich geht dies im Allgemeinen nicht für alle möglichen Eingabewerte (primitive Datentypen haben zwar keinen unendlichen Wertebereich, die vollständige Aufzählung z.B. aller **`int`**-Werte ist allerdings wenig praktikabel).
- Testen kann dennoch sehr wichtig sein: insbesondere in der Entwicklungsphase verwendet man Tests zum Debugging.
- Zum formalen Beweis der Korrektheit imperativer Programme benötigt man stattdessen spezielle Logik-Kalküle (z.B. den Hoare-Kalkül).

Definitionsbereiche von Programmen

- Frage: Was passiert eigentlich, wenn Eingabewerte, die der Vorbedingung `PRE` nicht genügen, verarbeitet werden?
- Zunächst wiederum nur funktionale Algorithmen:
  - Der Algorithmus stellt eine Funktion dar.
  - Dies ist eine Abbildung $f : D \rightarrow B$ vom Definitionsbereich D auf/in den Bildbereich B.
  - Korrektheit bedeutet informell: für Eingaben aus D erhält man durch Auswertung von f die entsprechend gewünschten Werte aus B.
  - Für Eingabewerte, die nicht Element von D sind, erhält man durch Auswertung von f (je nach Definition von f) beliebige Werte aus B oder gar keine Werte oder f terminiert nicht, etc.

Definitionsbereiche von Programmen

- Bei imperativen Algorithmen ist die Situation ähnlich:
  - Eine Methode ist ebenfalls eine Abbildung $f : D \rightarrow B$ vom Definitionsbereich D auf/in den Bildbereich B mit möglichen Seiteneffekten.
  - Die "Vorschrift", wie diese Abbildung berechnet wird, ist nun als Folge von Anweisungen notiert, statt als mathematische Funktion.
  - Analog bedeutet Korrektheit damit informell: für Eingaben aus D erhält man durch Abarbeiten der Anweisungen die entsprechend gewünschten Werte aus B bzw. (z.B. wenn $B = \emptyset$) entsprechend gewünschte Seiteneffekte.
  - Für Eingabewerte, die nicht Element von D sind, erhält man durch Abarbeiten der Anweisungen (je nach Anweisungen) wiederum beliebige Werte oder gar keine Werte oder der Algorithmus terminiert nicht, etc.
  - Zusätzlich können bei imperativen Algorithmen für Eingabewerte, die nicht Element von D sind, (möglicherweise unerwünschte) Nebeneffekte auftreten.
- Offensichtlich sollte also die Eingabe von Werten, die nicht aus dem Definitionsbereich des Algorithmus stammen, vermieden werden.

Definitionsbereiche von Programmen

- Beispiel: Java-Methode `quadrat`
  - Definitionsbereich $D =$ **`int`**
  - Bildbereich $B =$ **`int`**
- Frage:
  Was passiert, wenn ich die Methode `quadrat` statt mit einem **`int`**-Wert mit einem **`double`**-Wert aufrufe?
- Antwort:
  - Falls dies bereits zur Kompilierzeit klar ist, wird das Programm gar nicht erst kompilieren.
  - Falls dies erst zur Laufzeit klar ist, wird die JRE eine Fehlermeldung in Form einer *Ausnahme* (*Exception*, siehe später) produzieren.
- Folgerung: Dann ist ja alles O.K., oder?

Definitionsbereiche von Programmen

- Nochmal eine Frage:
  Was passiert, wenn ich die Methode `quadrat` mit einem **`int`**-Wert `a < 0` oder einem **`char`**-Wert aufrufe?
- Antwort für **`int`**-Wert `a < 0`:
  Die Methode terminiert nicht.
- Antwort für **`char`**-Wert:
  Der **`char`**-Wert wird in einen positiven **`int`**-Wert konvertiert und damit ist der Ergebniswert wohldefiniert.
- Vermutlich ist weder diese Wohldefiniertheit, noch die Nicht-Terminierung beabsichtigt.
- In der Praxis sind solche Situationen häufig, daher ist ausführliches Kommentieren umso wichtiger!

Definitionsbereiche von Programmen

- Zusätzlich gibt es in Java Möglichkeiten, unerwünschte Ergebnisse aufgrund von unzulässigen Eingabewerten abzufangen.
- Zusicherungen (Assertions) können benutzt werden, um beliebige Bedingungen (in Form von Boole’schen Ausdrücken) an beliebigen Stellen im Programm zu plazieren (siehe nächster Unter-Abschnitt).
- Ausnahmen sind dazu da, Fehler, die während der Programmausführung auftreten können, abzufangen, z.B. einen Zugriff auf ein Arrayfeld, das außerhalb der definierten Grenzen liegt (siehe später).

Überblick

Zusicherungen

- Mit Testen kann man NICHT die Korrektheit von Programmen vollständig beweisen!
- Man kann lediglich gewisse Eigenschaften des Programmes für bestimmte Eingabewerte testen.
- In Java kann man Tests durch sog. *Zusicherungen* (*Assertions*) durchführen.
- Assertions sind Anweisungen, die Annahmen über den Zustand des Programmes zur Laufzeit verifizieren (z.B. der Wert einer Variablen).

Zusicherungen

- Die Assert-Anweisung hat in Java folgende Form:

  `assert <ausdruck1> : <ausdruck2>;`

  Wobei der Teil "`: <ausdruck2>`" optional ist.
- `<ausdruck1>` muss vom Typ **`boolean`** sein.
- `<ausdruck2>` darf von beliebigem Typ sein. Er dient als Text für eine Fehlermeldung.
- Genaugenommen wird, falls `<ausdruck1>` den Wert **`false`** hat, eine *Ausnahme* (*Exception*) ausgelöst und `<ausdruck2>` dieser Exception übergeben. Fehlt `<ausdruck2>`, wird der Exception eine "leere Fehlermeldung" übergeben. Was genau eine Exception ist, und wie man damit umgehen kann, lernen wir später kennen.
- Ist der Wert von `<ausdruck1>` **`true`**, wird das Programm normal fortgeführt.

Zusicherungen

- Beispiel:

  ```
  assert x >= 0;
  ```

  überprüft, ob die Variable `x` an dieser Stelle des Programms nicht-negativ ist.
- Unterschied zur bedingten Anweisung:
  - Kürzerer Programmcode.
  - Auf den ersten Blick ist zu erkennen, dass es sich um einen Korrektheits-Test handelt und nicht um eine Verzweigung zur Steuerung des Programmablaufs.
  - Assertions lassen sich zur Laufzeit wahlweise an- oder abschalten. Sind sie deaktiviert, verursachen sie (im Gegensatz zu einer einfachen **`if`**-Anweisung) praktisch keine Verschlechterung des Laufzeitverhaltens.
- An- und Abschalten von Assertions beim Ausführen eines Programms durch Kommandozeilenargument der JVM:
  - Anschalten: Parameter `-ea`, z.B.
    ```
    java -ea MyProgramm
    ```
  - Abschalten (default!): Parameter `-da`, z.B.
    ```
    java -da MyProgramm
    ```

Zusicherungen

- Mittels Assertions könnte man z.B. überprüfen, ob der Definitionsbereich einer Methode eingehalten wird:

```
public static int quadrat(int a)
{

  // Precondition als Zusicherung
  assert a >= 0;

  int y;
  int z;

  // Anfang der Berechnung
  y = 0;
  z = 0;
  while (y != a)
  {
    z = z + 2*y + 1;
    y = y + 1;
  }
  // Ende der Berechnung

  return z;
}
```

Zusicherungen vs. Ausnahmen

- Ist dies sinnvoll?
- Die Überprüfung, ob der Definitionsbereich einer Methode oder sogar eines gesamten Programms eingehalten wird, sollte nicht einfach zur Laufzeit abschaltbar sein!
- Vielmehr sollte die Überprüfung von Werten, die einem Programm übergeben werden, immer aktiv sein.
- Für die Überprüfung von Preconditions eignen sich daher die Ausnahmen, da diese nicht abschaltbar sind.
- Assertions sind eher geeignet, Programmfehler zu entdecken und nicht fehlerhafte Eingabedaten.
- Assertions sollten daher ausschließlich in der Debugging-Phase eingesetzt werden.
- **Folgerung:** Ein Assert-Statement wird vom Programmierer als “immer wahr” angenommen.

Zusicherungen vs. Ausnahmen

- Beispiele, bei denen der Einsatz von Assertions sinnvoll ist:
  - Überprüfung von Postconditions in der Debugging-Phase um die Korrektheit einer Methode (oder des gesamten Programms) für einige Testfälle (Eingabebeispiele) zu evaluieren.
    Achtung: Wie bereits oben diskutiert, ist Testen *kein* formaler Korrektheitsbeweis!
  - Überprüfung von *Schleifeninvarianten*. Dies sind Bedingungen, die am Anfang oder am Ende einer Schleife *bei jedem Durchlauf* erfüllt sein müssen. Dies ist besonders bei Schleifen sinnvoll, die so komplex sind, dass beim Programmieren eine Art "Restunsicherheit" bezüglich ihrer Korrektheit bestehen bleibt.
  - Die Markierung von *toten Zweigen* in `if`- oder `case`-Anweisungen, die nicht erreicht werden sollten. Anstatt hier einen Kommentar der Art "kann niemals erreicht werden" zu plazieren, könnte auch eine Assertion `assert false` gesetzt werden. Wird dieser Zweig bei einem Test während der Debugging-Phase wegen eines Programmfehlers durchlaufen, wird dies wirklich erkannt.
- Die Allgemeingültigkeit von Zusicherungen kann wiederum nur mit einem speziellen Logik-Kalkül (z.B. Hoare-Kalkül) bewiesen werden.

Überblick

Hoare-Kalkül

- Wie bereits diskutiert, benötigt man zum formalen Beweis der Korrektheit imperativer Programme (oder auch der Allgemeingültigkeit von Zusicherungen) entsprechende Logik-Kalküle.
- Wir betrachten im Folgenden den Hoare-Kalkül.
- Der Hoare-Kalkül ist ein allgemeines Konzept, das an jede Programmiersprache angepasst werden kann.
- Um das Grundprinzip zu verstehen und kleine Beispiele rechnen zu können, beschränken wir uns hier auf eine Anpassung an ein Fragment von Java, welches lediglich aus folgenden drei Arten von Anweisungen besteht:
  - Wertzuweisungen
  - **`if`** `(<bedingung>) <anweisung1>` **`else`** `<anweisung2>`
  - **`while`** `(<bedingung>) <anweisung>`
- Da der Beweis der Terminierung oft nicht einfach ist und nicht vom Hoare-Kalkül unterstützt wird, beschränken wir uns hier auf den Nachweis der partiellen Korrektheit.

Hoare-Kalkül

- Im Allgemeinen haben wir folgende Situation:

  `(PRE) {p_1; ... ; p_n;} (POST)`

  wobei
  - `(PRE)` die Precondition darstellt,
  - `p_1`, . . ., `p_n` die einzelnen Anweisungen des Programms sind,
  - `(POST)` die Postcondition darstellt.
- Es ist also für das Programm, bestehend aus der Anweisungsfolge `p_1; ... ; p_n;`, zu zeigen, dass,
  - falls das Programm auf Eingabewerte, die der Precondition `(PRE)` genügen, angewendet wird, und
  - falls es terminiert,

  anschließend die Postcondition `(POST)` gilt (partielle Korrektheit).

Hoare-Kalkül

- Das Grundprinzip des Korrektheitsbeweises mit dem Hoare-Kalkül ist wie folgt:
- **Schritt 1**
  - Finde eine Zwischenbedingung `Z_1` für `p_n` und spalte den Beweis in
    `(PRE) {p_1; ... ; p_(n-1);} (Z_1)` und
    `(Z_1) {p_n;} (POST)`
    (Der Fall `(PRE) {} (Z_1)` wird in Schritt 2 behandelt).
  - Die Weiterverarbeitung von `(Z_1) {p_n;} (POST)` hängt von der Art der Anweisung `p_n` ab. Für jede Anweisungsart benötigt man eine extra Regel.
- **Schritt 2**
  - Nach dem gleichen Schema werden die Anweisungen
    `p_1; ... ; p_(n-1);` behandelt, bis man schließlich die Situation
    `(PRE) {} (Z_n)`
    erreicht.
  - Partielle Korrektheit ist bewiesen, wenn in dieser Situation der Ausdruck `PRE => Z_n` den Wert **`true`** hat, also eine wahre Aussage darstellt.

Hoare-Kalkül

- Beispiel:
  `(x>=0 && y>=0) {} (x*y>=0)`
  führt zu
  `(x>=0 && y>=0) => (x*y>=0)`
  und dieser Ausdruck hat den Wert **`true`**.
- Die Hoare'sche Methode reduziert also die Verifikation auf rein mathematische Beweisprobleme (sog. Beweisverpflichtungen).
- Wenn alle anfallenden Beweisverpflichtungen auch tatsächlich bewiesen werden können, dann ist die partielle Korrektheit gezeigt.
- Im Folgenden schauen wir uns die einzelnen Verifikationsregeln für drei ausgewählte Arten von Anweisungen (Zuweisung, **`if`**-Anweisung, **`while`**-Schleife) und deren Beweisverpflichtungen genauer an.

Die Zuweisungsregel

- Ausgangssituation:

  `(PRE) {p_1; ...; p_(n-1); x = t;}(POST)`

  wobei `x` eine Variable und `t` ein Ausdruck ist.
- Die *Zuweisungsregel* transformiert dieses Problem in

  `(PRE) {p_1; ...; p_(n-1);} (POST[x/t])`

  wobei `POST[x/t]` bedeutet, dass alle Vorkommen der Variablen `x` in `POST` durch den Ausdruck `t` zu ersetzen sind.
- Dies spiegelt genau die Bedeutung der Zuweisung wieder: nach ihrer Ausführung sind `x` und `t` identisch.
- Die Zuweisungsregel erlaubt, die Zwischenbedingung `Z_1` explizit zu berechnen, nämlich

  `Z_1 = POST[x/t]`.

Die Zuweisungsregel

Beispiel:

Aus

```
(PRE) {p_1; ...; p_(n-1); y = x*x;} (y>=0 && y<=10)
```

wird

```
(PRE) {p_1; ...; p_(n-1)} (x*x>=0 && x*x<=10)
```

Die `if`-Regel

- Ausgangssituation:

  `(PRE) {p_1; ...; p_(n-1); if(B) p else q}(POST)`,

  wobei `B` die Testbedingung ist und `p` und `q` Programmstücke sind.
- Die **`if`-***Regel* transformiert dieses Problem in vier einzelne Probleme:
  - Finde eine geeignete Zwischenbedingung `Z_1` als neue Vorbedingung für die **`if`**-Anweisungen.
  - Beweise den **`true`**-Zweig

    `(Z_1 && B) {p} (POST)`.
  - Beweise den **`false`**-Zweig

    `(Z_1 && !B) {q} (POST)`.
  - Mache weiter mit den restlichen Anweisungen vor der **`if`**-Anweisung für die Nachbedingung `Z_1`

    `(PRE) {p_1; ...; p_(n-1);} (Z_1)`.

Die `if`-Regel

Beispiel:

```
(true)
  {
    if (x>0)
    {
      y = x;
    }
    else
    {
      y = - x;
    }
  }
(y == |x|)
```

dabei soll `|x|` den Absolutbetrag von `x` bezeichnen, d.h. das Programmstück soll den Absolutbetrag von `x` berechnen.

Die `if`-Regel

Die vier Schritte der **`if`**-Regel sind:

- Zwischenbedingung wird keine benötigt, d.h. `Z_1 = PRE = true`.
- Der **`true`**-Zweig:
  `(true && x>0) {y = x;} (y == |x|)`
  Daraus wird mit der Zuweisungsregel
  `(true && x>0) {} (x == |x|)`
  und daraus wiederum
  `(true && x>0) => (x == |x|)` und das stimmt.
- Der **`false`**-Zweig
  `(true && !(x>0)) {y = -x;} (y == |x|)`
  Daraus wird mit der Zuweisungsregel
  `(true && !(x>0)) {} (-x == |x|)`
  und daraus wiederum
  `(true && x<=0) => (-x == |x|)` und das stimmt ebenfalls.
- Der Rest vor der **`if`**-Anweisung ist leer, d.h. man wäre fertig.

Die `while`-Regel

Wir betrachten die **`while`**-Schleife ohne **`break`**- und **`continue`**-Anweisungen.

- Ausgangssituation:

  `(PRE) {p_1; ...; p_(n-1); while(B) p}(POST)`

  wobei `B` die Testbedingung ist und `p` ein Programmstück.
- Die **`while`**-*Regel* transformiert dieses Problem in fünf einzelne Probleme:

Die `while`-Regel

- Die fünf einzelnen Probleme:
  - Finde eine geeignete *Schleifeninvariante* `INV`, die bei jedem Durchgang durch das Programmstück `p` gültig (invariant) bleibt. Das Auffinden der Invariante ist oft ein kreativer Vorgang.
  - Finde eine geeignete Zwischenbedingung `Z_1` als neue Vorbedingung für die **`while`**-Anweisung, so dass gilt:

    `Z_1 => INV`.

    `Z_1` ist die spezielle Form von `INV`, die vor dem Eintritt in die Schleife gilt.
  - Verifiziere den Erhalt der Schleifeninvariante

    `(INV && B) {p} (INV)`.

    Dies bestätigt, dass `INV` solange gültig bleibt, wie `B` gilt.

Die `while`-Regel

- Die fünf einzelnen Probleme (cont.):
  - Weise nach, dass die Schleifeninvariante stark genug ist, dass sie die Nachbedingung `POST` erzwingt:

    `(INV && !B) => POST`

    d.h. nachdem `B` falsch geworden ist, und die Schleife verlassen wurde, muss `POST` folgen.
  - Mache weiter mit den restlichen Anweisungen vor der Schleife für die Nachbedingung `Z_1`

    `(PRE) {p_1; ...; p_(n-1);} (Z_1).`

Die `while`-Regel

Beispiel: Die Methode `quadrat` vom Beginn des Abschnitts.

```
(0 <= a)
  {
    y = 0;
    z = 0;
    while (y != a)
    {
      z = z + 2*y + 1;
      y = y + 1;
    }
  }
(z == a*a)
```

Die letzte Anweisung ist eine **`while`**-Schleife. Daher wird die **`while`**-Regel angewendet.

Die `while`-Regel

Die fünf Schritte der **`while`**-Regel sind:

- Schleifeninvariante `INV`:
  `y<=a && z==y*y`
- Zwischenbedingung `Z_1`:
  `0 <= a && y==0 && z==0`
  Dies impliziert offensichtlich
  `y<=a && z==y*y`

Die `while`-Regel

Die fünf Schritte der **`while`**-Regel sind (cont.):

- Erhalt der Schleifeninvariante:
  ```
  (y<=a && z==y*y && y!=a)
  {z = z + 2*y + 1; y = y + 1;}
  (y<=a && z==y*y)
  ```
  Dies zeigt man durch zweimaliges Anwenden der Zuweisungsregel:
  1. ```
     (y<=a && z==y*y && y!=a)
     {z = z + 2*y + 1;}
     ((y+1)<=a && z==(y+1)*(y+1))
     ```
  2. ```
     (y<=a && z==y*y && y!=a)
     {}
     ((y+1)<=a && (z + 2*y + 1)==(y+1)*(y+1))
     ```

Die `while`-Regel

Die fünf Schritte der **`while`**-Regel sind (cont.):

- Erhalt der Schleifeninvariante (cont.):

  Daraus wird

  ```
  (y<=a && z==y*y && y!=a)
  =>
  ((y+1)<=a && (z + 2*y + 1)==(y+1)*(y+1))
  ```

  Dies gilt, denn:

  - Aus `y<=a && y!=a` folgt `y<a` und damit `y+1<=a`
  - Aus `z==y*y` folgt
    `z+2*y+1 == y*y+2*y+1 == (y+1)*(y+1)`

Die `while`-Regel

Die fünf Schritte der **`while`**-Regel sind (cont.):

- Nachweis der Nachbedingung:
  `(INV && !B) => POST` also
  `((y<=a && z==y*y) && !(y!=a)) => z == a*a`
  dieser Ausdruck ist immer wahr, denn wenn die linke Seite der Implikation wahr ist, muss es auch die rechte sein (`!(y!=a)` entspricht `y==a`).
- Die restlichen Anweisungen vor der Schleife mit neuer Nachbedingung `Z_1` ergeben:
  `(0<=a) {y=0; z=0;} (0<=a && y==0 && z==0)`
  Zweimalige Anwendung der Zuweisungsregel ergibt:
  `(0<=a) => (0<=a && 0==0 && 0==0)`
  was offensichtlich wahr ist.

7. Komplexität von imperativen Programmen

Ressourcenbedarf von Algorithmen

- Algorithmen verbrauchen zwei Ressourcen:
  - Rechenzeit
  - Speicherplatz
- Im folgenden betrachten wir, wie wir messen können, wieviel Rechenzeit und Speicherplatz verbraucht werden.

Laufzeitanalyse

- **1. Ansatz:** Direktes *Messen der Laufzeit* (z.B. in ms).
  - Abhängig von vielen Parametern (Rechnerkonfiguration, Rechnerlast, Compiler, Betriebssystem, . . .)
  - Daher: kaum übertragbar und ungenau.
- **2. Ansatz:** Zählen der benötigten *Elementaroperationen* des Algorithmus in Abhängigkeit von der Größe *n* der Eingabe.
  - Das algorithmische Verhalten wird als Funktion der benötigten Elementaroperationen dargestellt.
  - Die Charakterisierung dieser elementaren Operationen ist abhängig von der jeweiligen Problemstellung und dem zugrunde liegenden Algorithmus.
  - Beispiele für Elementaroperationen: Zuweisungen, Vergleiche, arithmetische Operationen, Arrayzugriffe, . . .

Laufzeitanalyse

- Das Maß für die Größe *n* der Eingabe ist abhängig von der Problemstellung, z.B.
  - Suche eines Elementes in einem Array: *n* = Länge des Arrays.
  - Multiplikation zweier Matrizen: *n* = Dimension der Matrizen.
  - Sortierung einer Liste von Zahlen: *n* = Anzahl der Zahlen.
- Laufzeit:
  benötigte Elementaroperationen bei einer bestimmten Eingabelänge *n*.
- Speicherplatz:
  benötigter Speicher bei einer bestimmten Eingabelänge *n*.

Problem: Hardware-Abhängigkeit

- Laufzeit einzelner Elementar-Operation ist abhängig von der eingesetzten Rechner-Hardware.
- Frühere Rechner (incl. heutige PDAs, Mobiltelefone, . . .):
  - “billig”: Fallunterscheidungen, Wiederholungen
  - “mittel”: Rechnen mit ganzen Zahlen (Multiplikation usw.)
  - “teuer”: Rechnen mit reellen Zahlen
- Heutige Rechner (incl. z.B. Spiele-Konsolen):
  - “billig”: Rechnen mit reellen und ganzen Zahlen
  - “teuer”: Fallunterscheidungen

Asymptotisches Laufzeitverhalten

- "Kleine" Probleme (z.B. $n = 5$) sind uninteressant:
  Die Laufzeit des Programms ist eher bestimmt durch die Initialisierungskosten (Betriebssystem, Programmiersprache etc.) als durch den Algorithmus selbst.
- Interessanter:
  - Wie verhält sich der Algorithmus bei sehr großen Problemgrößen?
  - Wie verändert sich die Laufzeit, wenn ich die Problemgröße variiere (z.B. Verdopplung der Problemgröße)?
- Das bringt uns zu dem Konzept "asymptotisches Laufzeitverhalten".

Definition *O*-Notation

- Mit der *O-Notation* haben Informatiker einen Weg gefunden, die asymptotische Komplexität (bzgl. Laufzeit oder Speicherplatzbedarf) eines Algorithmus zu charakterisieren.
- Definition *O*-Notation:

  Seien $f : \mathbb{N} \to \mathbb{N}$ und $s : \mathbb{N} \to \mathbb{N}$ zwei Funktionen (s wie *Schranke*)

  Die Funktion f ist *von der Größenordnung* $O(s)$, geschrieben $f \in O(s)$, wenn es $k \in \mathbb{N}$ und $m \in \mathbb{N}$ gibt, so dass gilt:

  $$\text{Für alle } n \in \mathbb{N} \text{ mit } n \geq m \text{ ist } f(n) \leq k \cdot s(n).$$

- Man sagt auch: f wächst höchstens so schnell wie s.

Bemerkungen zur O-Notation

- k ist unabhängig von n:
  k muss dieselbe Konstante sein, die für alle $n \in \mathbb{N}$ garantiert, dass $f(n) \leq k \cdot s(n)$.
- Existiert keine solche Konstante k, ist f nicht von der Größenordnung $O(s)$.
- $O(s)$ bezeichnet die Menge aller Funktionen, die bezüglich s die Eigenschaft aus der Definition haben.
- Man findet in der Literatur häufig $f = O(s)$ statt $f \in O(s)$.

Rechnen mit der O-Notation

- Elimination von Konstanten:
  - $2 \cdot n \in O(n)$
  - $n/2 + 1 \in O(n)$
- Bei einer Summe zählt nur der am stärksten wachsende Summand (mit dem höchsten Exponenten):
  - $2n^3 + 5n^2 + 10n + 20 \in O(n^3)$
  - $O(1) \subseteq O(\log n) \subseteq O(n) \subseteq O(n \log n) \subseteq O(n^2) \subseteq O(n^3) \subseteq \ldots \subseteq O(2^n)$
- Beachte:
  $1000 \cdot n^2$ ist nach diesem Leistungsmaß immer "besser" als $0,001 \cdot n^3$, auch wenn das m, ab dem die $O(n^2)$-Funktion unter der $O(n^3)$-Funktion verläuft, in diesem Fall sehr groß ist (m=1 Million).

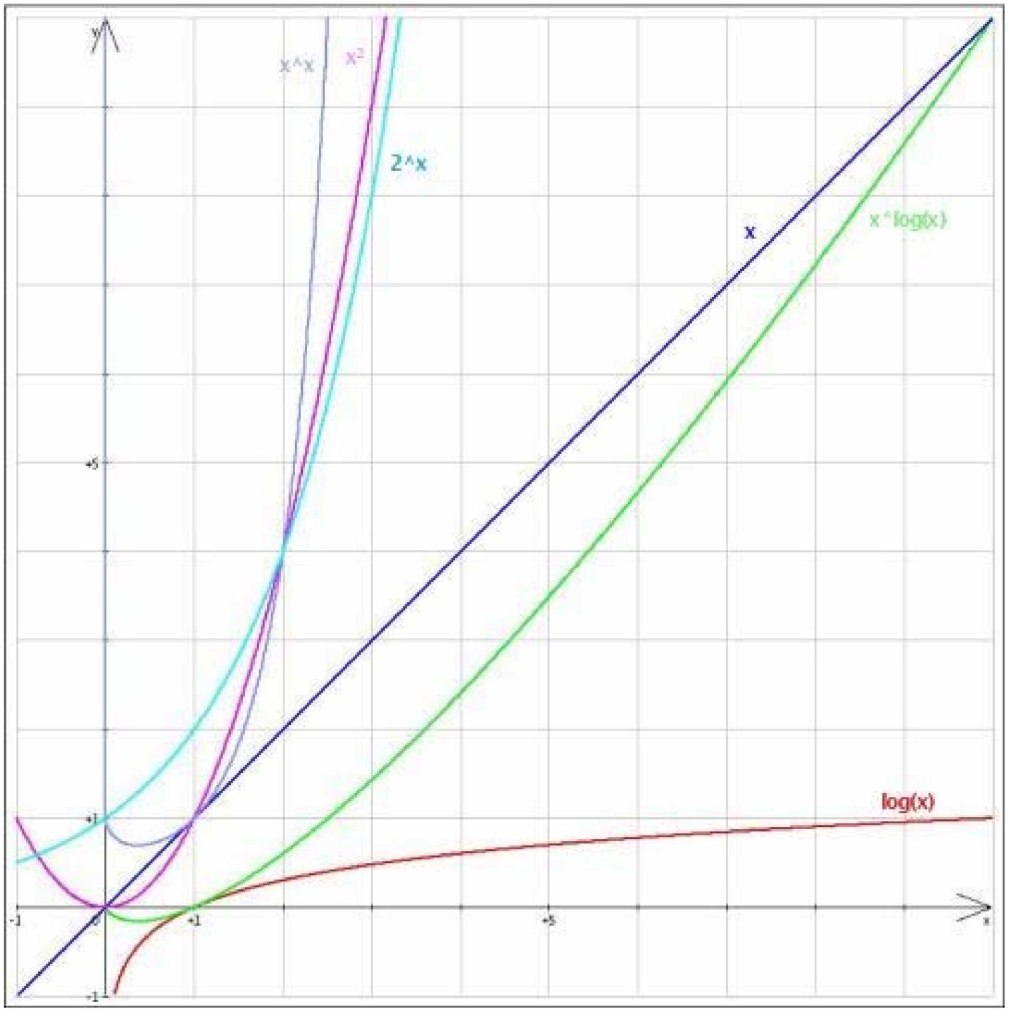
Wichtige Klassen von Funktionen

	Sprechweise	**Typische Algorithmen / Operationen**
$O(1)$	konstant	Addition, Vergleichsoperationen, rekursiver Aufruf, . . .
$O(\log n)$	logarithmisch	Suchen auf einer sortierten Menge
$O(n)$	linear	Bearbeiten jedes Elementes einer Menge
$O(n \cdot \log n)$		gute Sortierverfahren
$O(n \cdot \log^2 n)$		
	$\vdots$	
$O(n^2)$	quadratisch	primitive Sortierverfahren
$O(n^k), k \geq 2$	polynomiell	
	$\vdots$	
$O(2^n)$	exponentiell	Ausprobieren von Kombinationen

Wichtige Klassen von Funktionen

- Die *O*-Notation hilft insbesondere bei der Beurteilung, ob ein Algorithmus für großes *n* noch geeignet ist bzw. erlaubt einen Effizienz-Vergleich zwischen verschiedenen Algorithmen für große *n*.
- Schlechtere als polynomielle Laufzeit gilt als nicht effizient, kann aber für viele Probleme das best-mögliche sein.

Wichtige Klassen von Funktionen



Teil II

Konzepte objektorientierter Programmierung

Warum objektorientierte Programmierung?

- Elementarer Bestandteil der Algorithmen-Entwicklung ist die Modellierung eines Ausschnitts aus der realen Welt.
- Bisherige Modellierungsansätze:
  - Funktionale Programmierung:
    Die Lösung eines Problems wird als funktionaler (mathematischer) Zusammenhang zwischen Ein- und Ausgabedaten modelliert
  - Imperative Programmierung:
    Die Lösung eines Problems wird als Menge von sequentiellen Abarbeitungsschritten modelliert.
- Beides sind wichtige Ansätze. Mit zunehmender Komplexität der Probleme (und damit der Ausschnitte der realen Welt, die man modellieren muss) führen diese Ansätze allerdings zu immer komplexeren und unübersichtlicheren Programmen.

Warum objektorientierte Programmierung?

- Objektorientierte (oo) Programmierung ist *das* Programmierparadigma der 90er Jahre.
- OO Programmierung ist ein weiterer Ansatz, Problemstellungen der realen Welt zu modellieren.
- Der oo Ansatz orientiert sich dabei an “Dingen” (Objekte, die gewisse Eigenschaften und ein gewisses Verhalten haben), die modelliert werden müssen.
- Genau genommen ist der oo Ansatz ein Konzept, komplexe Daten und Programmzustände zu modellieren. Die tatsächlichen Algorithmen werden meist mit einer Mischung aus funktionalen und imperativen Konzepten notiert.
- Beispiel Java: die Algorithmen werden in der Regel mittels imperativer Konzepte implementiert, wogegen die Daten und deren Zustände oo modelliert sind.

Was ist objektorientierte Programmierung?

Betrachten wir als Beispiel die (fiktive) Immobilienfirma “Teuer und Häßlich”, die exklusive Einfamilienhäuser vermittelt. Wir sind damit beauftragt, eine Software zur Verwaltung dieser Häuser zu entwickeln.

- Für jedes Einfamilienhaus werden von der Firma verschiedene Daten gespeichert, z.B. Haustyp, Name des Besitzers, Adresse, Wohnfläche, Anzahl der Bäder, Verkaufspreis, etc.
- Ist ein potentieller Kunde gefunden, müssen die verschiedenen Daten für verschiedene Häuser abrufbar sein.
- Wie können wir die Daten modellieren? Was für Pobleme treten dabei auf?

Was ist objektorientierte Programmierung?

Modellierung der Daten:

- Da wir nur Arrays zur Speicherung von Mengen von Daten zur Verfügung haben und diese auch nur gleichartige Daten speichern können, müssen wir für alle Datenarten ein eigenes Array anlegen, d.h. ein Array, in dem für alle Häuser der Haustyp gespeichert ist, ein Array, in dem für alle Häuser der Name des Besitzers gespeichert ist, etc.

Problem:

- Offensichtlich ist diese Datenmodellierung sehr unübersichtlich und fehleranfällig, insbesondere wenn mehrere Programmierer an diesem Projekt arbeiten. Warum?

Lösung:

- oo Modellierung: Abstraktion und Kapselung zu Objekten.

Was ist objektorientierte Programmierung?

Die Idee der oo Programmierung (Modellierung) am Beispiel der (fiktiven) Immobilienfirma “Teuer und Häßlich”, die exklusive Einfamilienhäuser vermittelt:

- Jedes Einfamilienhaus ist ein *Objekt* (eine *Instanz* einer allgemeinen Klasse von Objekten). Die Firma vermittelt z.B. gerade ein Landhaus.
- Über dieses “Landhaus”-Objekt werden von der Firma verschiedene Daten gespeichert, z.B. Haustyp, Name des Besitzers, Adresse, Wohnfläche, Anzahl der Bäder, Verkaufspreis, etc. Diese Daten werden *Attributwerte* des Objekts genannt.
- Ist ein potentieller Kunde gefunden, muss der Kaufpreis abrufbar sein. Es wird eine Funktion benötigt, die auf das “Landhaus”-Objekt angewendet werden kann um dessen Kaufpreis zu ermitteln. Diese Funktion nennt man eine *Operation* oder *Methode*.
- Auf die Attributwerte eines Objekts kann nur über die entsprechenden Methoden zugegriffen werden. Dieses Prinzip nennt man (*Daten*-)*Kapselung*.

Was ist objektorientierte Programmierung?

- Außer dem “Landhaus”-Objekt bietet “Teuer und Häßlich” noch andere Einfamilienhäuser an:

Einfamilienhaus
Haustyp = „Landhaus“ Besitzer = „Dr. Kaiser“ Adresse = „Solln“ Wohnfläche = 400 qm Anzahl Bäder = 3 Verkaufspreis = 2 Mio. EUR
Verkaufspreis anzeigen

Einfamilienhaus
Haustyp = „Stadthaus“ Besitzer = „Herzog“ Adresse = „Laim“ Wohnfläche = 220 qm Anzahl Bäder = 2 Verkaufspreis = 1 Mio. EUR
Verkaufspreis anzeigen

Einfamilienhaus
Haustyp = „Bungalow“ Besitzer = „Miller“ Adresse = „Freimann“ Wohnfläche = 250 qm Anzahl Bäder = 2 Verkaufspreis = 1,2 Mio. EUR
Verkaufspreis anzeigen

- Alle Objekte besitzen zwar unterschiedliche Attributwerte, die aber alle vom gleichen Typ sind. Man spricht von *Attributen*.
- Zudem besitzen alle Objekte die gleiche Operation “Verkaufspreis anzeigen”.
- Die Objekte werden daher zu einer *Klasse* “Einfamilienhaus” zusammengefasst.

Was ist objektorientierte Programmierung?

- Eine Klasse definiert die Attribute und Methoden ihrer Objekte.
- Der *Zustand* eines konkreten Objekts wird durch seine Attributwerte und Verbindungen (*Links*) zu anderen konkreten Objekten bestimmt.
- Das mögliche *Verhalten* eines Objekts wird durch die Menge von Methoden beschrieben.

- Die wichtigsten Konzepte der oo Programmierung sind:
  - Abstraktion
  - Kapselung
  - Wiederverwendung
  - Beziehungen
  - Polymorphismus

Überblick

Abstraktion

- Eines der wichtigsten Konzepte der oo Programmierung ist die Trennung zwischen Konzept und Umsetzung: Wenn man z.B. seinen Führerschein gemacht hat, kann man ein Auto fahren, ohne dessen kompliziertes Innenleben zu verstehen.
- Diese *Abstraktion* manifestiert sich in der Unterscheidung zwischen Objekt und Klasse. Ein Objekt ist ein existierendes “Ding” aus der Anwendungswelt des Programms (z.B. das Landhaus von Dr. Kaiser). Die Klasse ist eine Beschreibung eines oder mehrerer ähnlicher Objekte.
- Eine Klasse beschreibt mind. drei wichtige Dinge:
  - Wie ist das Objekt zu bedienen?
  - Welche Eigenschaften hat das Objekt und wie verhält es sich?
  - Wie wird das Objekt hergestellt?

Abstraktion

- Jedes erzeugte Objekt einer Klasse hat seine eigene Identität.
- Beispiel: Lichtschalter in einem Haus
  - Alle Lichtschalter sind gleich zu bedienen.
  - Alle Lichtschalter sind gleich konstruiert.
  - Dennoch unterscheidet sich der Lichtschalter für die Flurbeleuchtung vom Lichtschalter fürs Badezimmer: Es ist nicht *derselbe* Lichtschalter.
- Abstraktion hilft, Details zu ignorieren, und reduziert damit die Komplexität des Problems. Dadurch werden komplexe Apparate und Techniken beherrschbar.

Überblick

Kapselung

- Eine Klasse ist die Zusammenfassung einer Menge von Daten (Attributen, auch: *Membervariablen* oder *Instanzvariablen*) und darauf operierender Funktionen (Methoden).
- Die Attribute werden für jedes konkrete Objekt neu angelegt bzw. belegt. Sie repräsentieren den Zustand der Objekte.
- Die Methoden sind nur einmal realisiert/definiert und operieren bei jedem Aufruf auf den Daten eines bestimmten konkreten Objekts. Sie repräsentieren das Verhalten der Objekte.
- *Kapselung* bedeutet, dass (von gewollten Ausnahmen abgesehen) die Methoden die einzige Möglichkeit darstellen, mit einem konkreten Objekt zu kommunizieren und so Informationen über dessen Zustand zu gewinnen oder diesen zu verändern.

Kapselung

- Kapselung hilft, die Komplexität der Bedienung eines Objekts zu reduzieren.
- Durch Kapselung werden die Implementierungsdetails von Objekten verborgen.
- Dadurch können Daten nicht bewußt oder versehentlich verändert werden.
- Kapselung ist daher auch ein wichtiger Sicherheitsaspekt: Ein direkter Zugriff auf die Daten wird unterbunden, der Zugriff erfolgt nur über definierte *Schnittstellen*.

Überblick

Wiederverwendung

- Durch Abstraktion und Kapselung wird die *Wiederverwendbarkeit* von Programmteilen gefördert.
- Beispiel: sog. *Collections* sind Objekte, die Sammlungen anderer Objekte aufnehmen und auf eine spezielle Art und Weise verarbeiten können.
  - Collections sind meist sehr kompliziert aufgebaut.
  - Collections haben aber meist eine einfache Art der Bedienung (Schnittstelle).
  - Werden Collections als Klasse realisiert, werden durch die Kapselung die komplexen Details “wegabstrahiert”.
  - Dies erleichtert die Wiederverwendung: Wenn ein Programm eine spezielle Collection benötigt, muss ein Objekt der passenden Klasse erzeugt werden. Das Programm kann über die einfache Schnittstelle (Methoden der Klasse) darauf zugreifen.
- Wiederverwendbarkeit hilft, effizienter und fehlerfreier zu programmieren.

Überblick

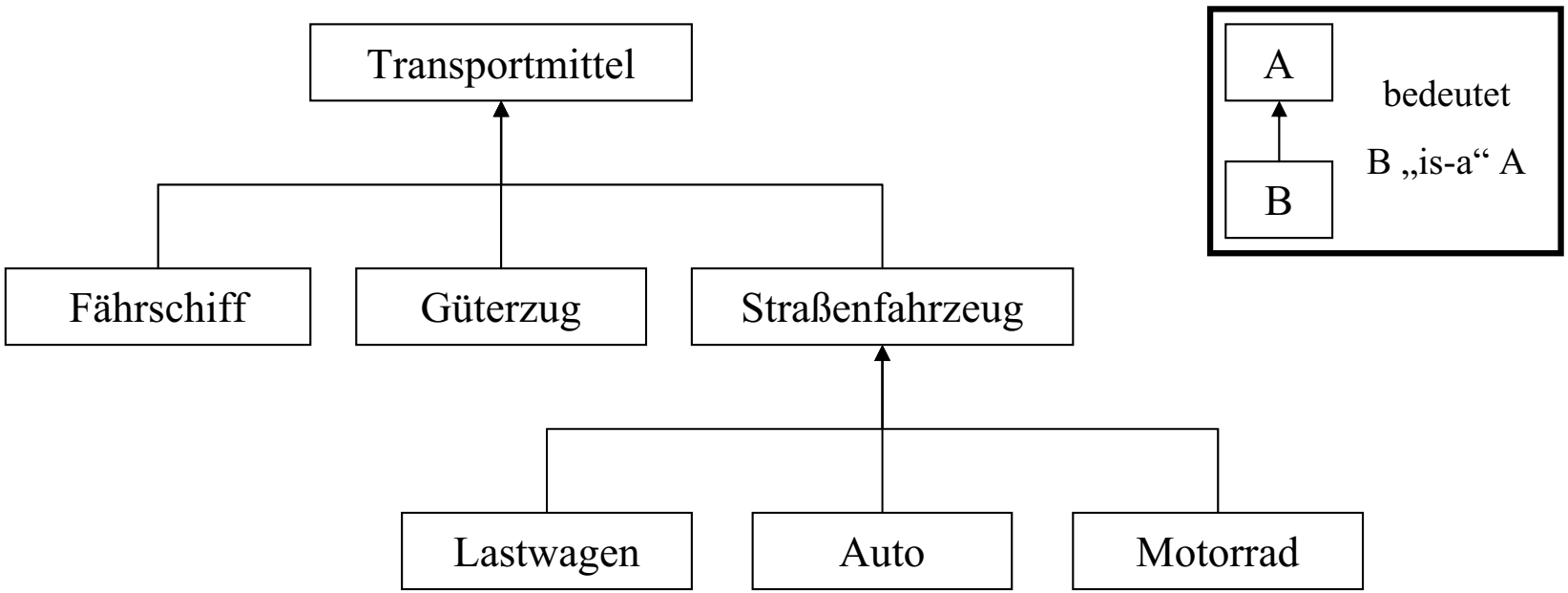
Beziehungen

- Klassen und Objekte existieren typischerweise nicht isoliert, sondern stehen in Beziehung zueinander.
- Beispiel:
  - Sowohl ein Motorrad, als auch ein Auto und ein Lastwagen sind Straßenfahrzeuge.
  - Ein Lastwagen kann möglicherweise Motorräder aufnehmen.
  - Ein Fährschiff ist ebenfalls ein Transportmittel und kann viele Autos, Lastwagen und Motorräder aufnehmen.
- Grundsätzlich gibt es in der oo Programmierung drei Arten von Beziehungen:
  - Generalisierung und Spezialisierung (“is-a”-Beziehungen),
  - Aggregation und Komposition (“part-of”-Beziehungen),
  - Verwendungs- und Aufrufbeziehungen.

Generalisierung und Spezialisierung

- Eine “is-a”-Beziehung zwischen zwei Klassen *A* und *B* sagt aus, dass “*B* ein *A* ist”, also *B* alle Eigenschaften von *A* besitzt (und darüberhinaus vermutlich noch ein paar mehr).
- *B* ist eine *Spezialisierung* von *A*.
- *A* ist eine *Generalisierung* von *B*.
- Beispiel:
  - Motorrad, Auto und Lastwagen sind zwar paarweise unterschiedlich, aber alle sind Straßenfahrzeuge.
  - Straßenfahrzeuge sind, genau wie Fährschiffe (und Güterzüge etc.), Transportmittel.
  - Die ensprechende “is-a”-Beziehung ist in der *Hierarchie* auf der folgenden Folie veranschaulicht.

Generalisierung und Spezialisierung



Generalisierung und Spezialisierung

- “is-a”-Beziehungen werden in der oo Programmierung durch *Vererbung* modelliert.
- Die speziellere Klasse wird dabei nicht komplett neu definiert, sondern von der allgemeineren Klasse *abgeleitet*.
- Die speziellere Klasse (auch: *abgeleitete Klasse*) erbt alle Eigenschaften der allgemeineren Klasse (auch: *Vaterklasse* oder *Basisklasse*). Eigene Eigenschaften können nach Belieben hinzugefügt werden.
- Vererbungen können mehrstufig sein, d.h. eine abgeleitete Klasse kann wiederum Vaterklasse für andere abgeleitete Klassen sein (siehe Beispiel auf der vorherigen Folie). Dann spricht man von *Vererbungshierarchien*.
- Grundsätzlich kann eine Klasse auch von mehreren Vaterklassen abgeleitet sein. Beispielsweise ist ein Amphibienfahrzeug (abgeleitete Klasse) sowohl ein Wasserfahrzeug (Vaterklasse) als auch ein Landfahrzeug (Vaterklasse). In diesem Fall spricht man von *Mehrfachvererbung*.

Aggregation und Komposition

- Eine “part-of”-Beziehung beschreibt die *Zusammensetzung* eines Objekts aus anderen Objekten.
- Ist diese Zusammensetzung essentiell für die Existenz des zusammengesetzten Objekts, spricht man von *Komposition*. Ein Güterzug besteht z.B. aus einer Lokomotive und mehreren Anhängern. Ohne diese Teile gibt es keine Güterzüge.
- Ist die Zusammensetzung nicht essentiell für die Existenz des zusammengesetzten Objekts, spricht man von *Aggregation*. Zwischen Straßenfahrzeug und Fährschiff besteht eine “part-of”-Beziehung, aber das Fährschiff kann auch leer fahren.
- **Bemerkung:** In den meisten oo Programmiersprachen werden Komposition und Aggregation gleich behandelt.

Verwendungs- und Aufrufbeziehungen

- Verwendungs- und Aufrufbeziehungen (*Assoziationen*) kommen dadurch zustande, dass z.B.
  - eine Methode einer Klasse *A* während ihrer Ausführung ein temporäres Objekt der Klasse *B* benutzt,
  - in der Parameterliste einer Methode einer Klasse *A* eine Variable definiert wird, die als Typ die Klasse *B* hat,
  - etc.

Überblick

Polymorphismus

- Betrachten wir noch einmal Fährschiffe: Sie können Autos, Lastwägen und Motorräder – oder ganz allgemein Straßenfahrzeuge – aufnehmen.
- Um diese Aggregation zu beschreiben, genügt es also zu definieren, dass Fährschiffe Straßenfahrzeuge aufnehmen.
- *Polymorphismus* besagt, dass nicht nur Objekte der Klasse Straßenfahrzeuge aufgenommen werden könne, sondern auch Objekte aller abgeleiteten Klassen von der Vaterklasse Straßenfahrzeuge, also auch Autos, Lastwägen und Motorräder.
- Die zusätzlichen Eigenschaften der abgeleiteten Klassen sind für das Fährschiff irrelevant.
- Andersherum funktioniert Polymorphismus nicht!

Polymorphismus

- Ein weiterer Aspekt des Polymorphismus ist, dass Objekte unterschiedlicher Klassen die gleiche Funktionalität besitzen können (die allerdings in jeder Klasse unterschiedlich realisiert ist).
- Beispiel:
  - Bei allen Straßenfahrzeugen gibt es die Funktionalität, die Anzahl der Reifen abzufragen.
  - Dies würde durch eine entsprechende Methode “anzahlReifen” realisiert.
  - Diese Methode kann in allen drei Klassen (Auto, Lastwagen, Motorrad) denselben Namen haben. In allen drei Fällen steckt aber ein unterschiedlicher Algorithmus dahinter.
  - Diese Funktionalität (Methode “anzahlReifen”) könnte bereits in der Vaterklasse zur Verfügung stehen und in den abgeleiteten Klassen *überlagert* werden.
  - Wird im Zusammenhang eines Fährschiffes die Anzahl der Reifen eines Straßenfahrzeugs benötigt, das auf diesem Schiff gerade transportiert wird, wird dynamisch ausgewählt, welche Methode ausgeführt wird, ohne dass sich das Fährschiff darum kümmern muss (ist das Straßenfahrzeug z.B. ein Auto, wird die Methode “anzahlReifen” der Klasse Auto ausgeführt).

Abschnitt 9: Objektorientierter Entwurf mit der Unified Modeling Language (UML)

Überblick

Objektorientierte Algorithmenentwicklung

- Wie schon bei der imperativen oder funktionalen Programmierung ist der Entwurf der eigentlichen Algorithmen (und damit die Modellierung der Problemstellung) die entscheidende Herausforderung.
- Den fertigen Entwurf in einer oo Programmiersprache zu implementieren ist dann wiederum relativ trivial.
- Im folgenden schauen wir uns eine Konzeptsprache an, die den oo Entwurf unterstützt: die *Unified Modelling Language* (UML).

Was ist UML?

- UML ist eine Konzeptsprache zur Modellierung von Software-Systemen (insbesondere, aber nicht zwingend: objektorientierter Programme).
- UML ist eine Art Pseudo-Code, der allerdings eine wohl-definierte Semantik besitzt und von vielen Programmen verarbeitet werden kann.
- UML bietet sogar die Möglichkeit, Code-Fragmente oder gesamte Implementierungen anzugeben (z.B. in Java-Notation). Es gibt Tools, die daraus automatisch Java-Code generieren, der (je nach Modellierungstiefe) auch ausführbar ist.
- UML-Code selbst ist nicht ausführbar. Dennoch wird UML von vielen Experten als Prototyp für die nächste Generation von Programmiersprachen betrachtet.

Bestandteile von UML

- Die UML-Notation folgt einer intuitiven Diagramm-Notation.
- Eigentlich umfasst UML ein ganzes System von Konzeptsprachen, d.h. es gibt verschiedene Diagramm-Typen.
- Jeder Diagrammtyp hat seinen eigenen Fokus, z.B.
  - die statische Klassen-Struktur (*Klassendiagramm*, *Class Diagram*),
  - die abstrakte Funktionalität des Programms (*Anwendungsfalldiagramm*, *Use Case Diagram*)
  - die möglichen Zustände und Zustandsübergänge, die ein Objekt einer bestimmten Klasse während seiner “Existenz” einnehmen bzw. ausführen kann (*Zustandsdiagramm*, *State Machine Diagram*)

Klassendiagramme

- Im Rahmen dieser Vorlesung werden wir nur kurz auf Klassendiagramme eingehen und lernen, wie die oo Konzepte in UML modelliert werden.
- Einen tieferen Einblick in UML erhalten Sie in den Vorlesungen zur Software-Entwicklung.
- Klassendiagramme (auch: *Objektdiagramme*) beschreiben die statische Struktur eines Programms. Die konzeptionelle Sicht steht dabei im Vordergrund, die Realisierungsdetails werden meist mit anderen Diagrammtypen beschrieben.
- Klassendiagramme beschreiben im Wesentlichen die Klassen, Objekte und deren Beziehungen zu einander.

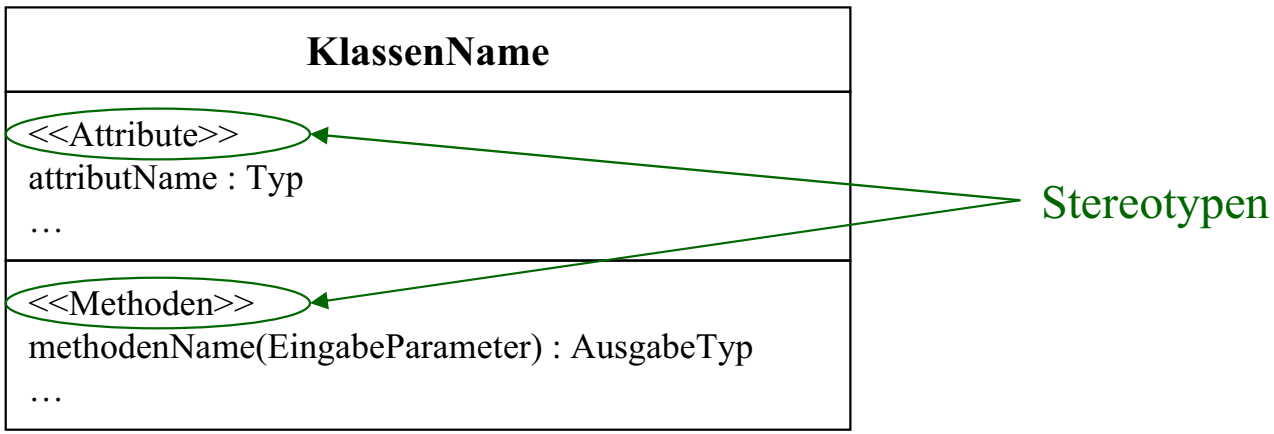
Überblick

Grundlagen

- Wie bereits erwähnt setzen die Konzepte der Klassen und Objekte die oo Grundkonzepte “Abstraktion” und “Kapselung” um.
- Klassen modellieren die gemeinsamen Eigenschaften von Objekten, insbesondere deren Attribute und Methoden.
- Attribute beschreiben den Zustand von Objekten einer Klasse und sollten für andere Benutzer (z.B. Objekte) verborgen sein.
- Methoden beschreiben das Verhalten der Objekte einer Klasse und sollten für andere Benutzer bekannt und verfügbar sein.

Klassen

Allgemeine Form einer Klassendefinition in UML:

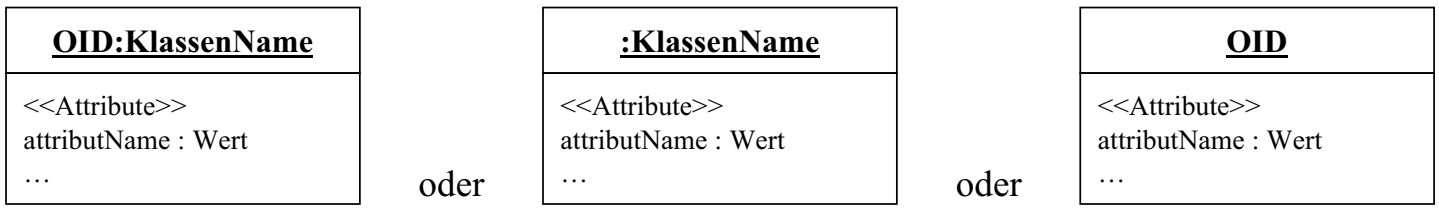


Sichtbarkeit

- Die Sichtbarkeit von Methoden und Attributen muss spezifiziert sein (beachte Kapselung/Abstraktion).
- Die Symbole zur Spezifikation der Sichtbarkeit von Klassen und deren Elementen sind unter anderem:
  - + bzw. public:
    das entsprechende Element (Attribut/Methode) ist von außen (z.B. Objekten anderer Klassen) sichtbar.
  - - bzw. private:
    das entsprechende Element (Attribut/Methode) ist von außen (z.B. Objekten anderer Klassen) NICHT sichtbar.
- Darüberhinaus gibt es weitere Möglichkeiten, die Sichtbarkeit von Elementen einer Klasse zu spezifizieren. Diese lernen wir später kennen.
- **Achtung:** Es gibt in UML (anders als z.B. in Java) *keine* Default-Spezifikation für Elemente einer Klasse, d.h. deren Sichtbarkeit muss immer explizit angegeben sein!

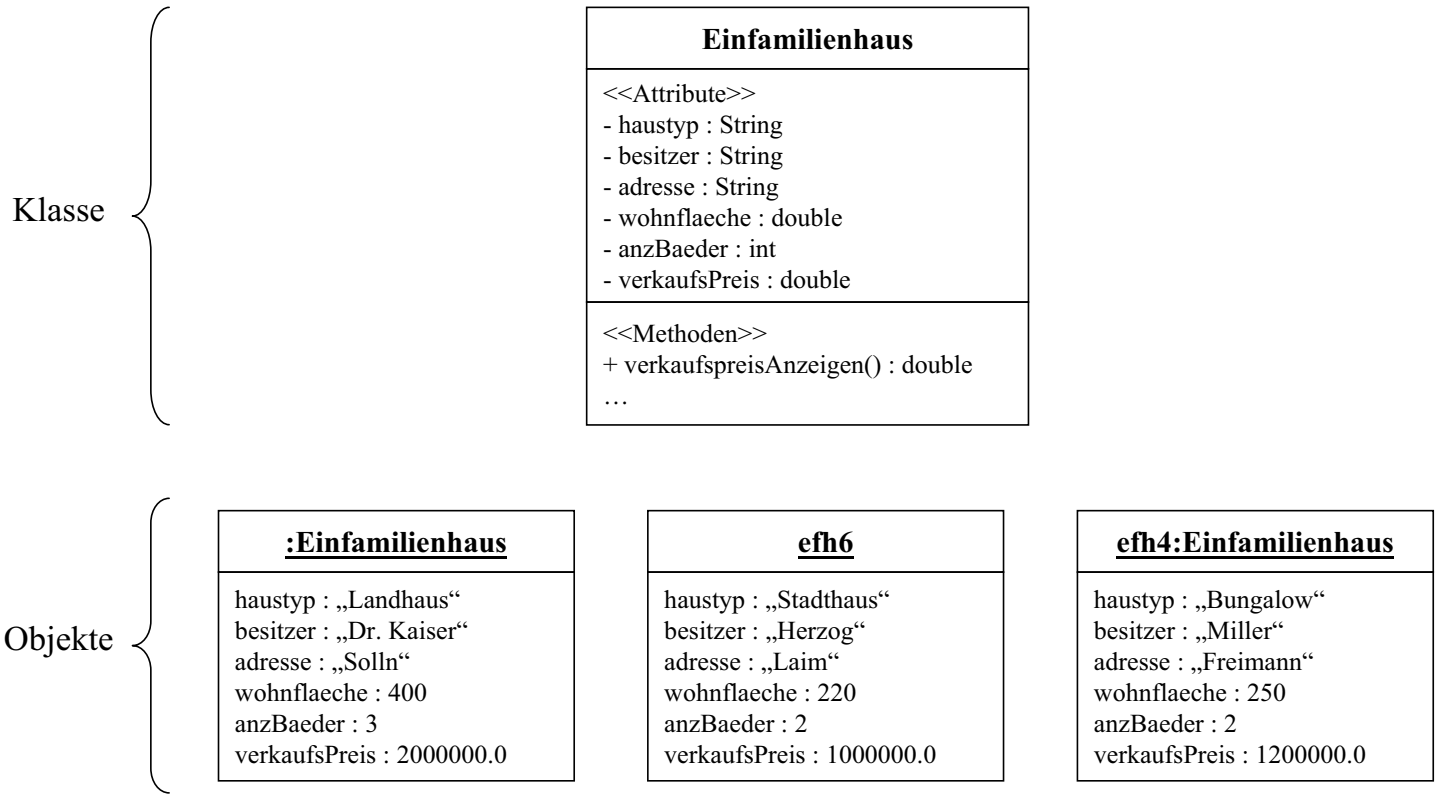
Objekte

Konkrete Objekte einer Klasse werden in UML wie folgt dargestellt:



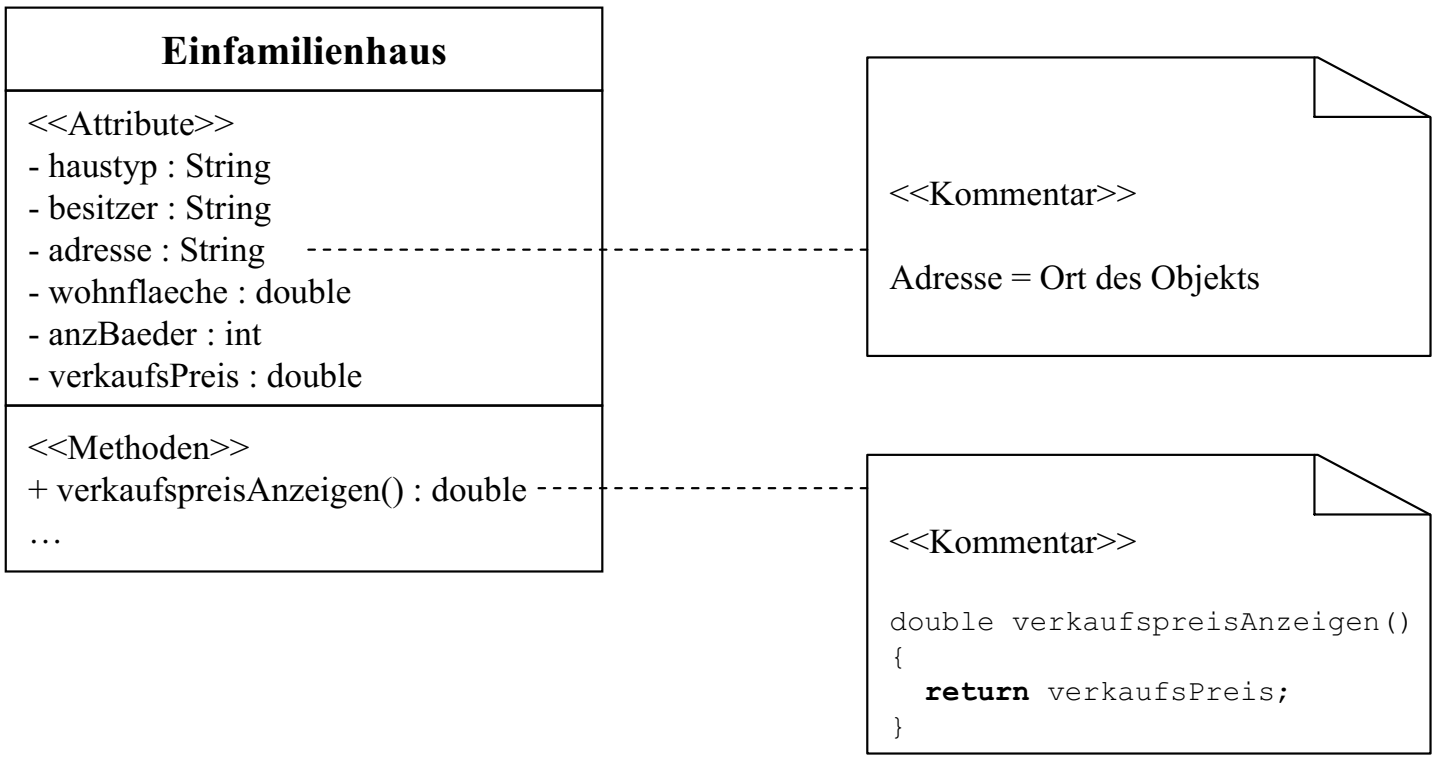
Beispiel: Klassen und Objekte

Ein Beispiel für die Klasse “Einfamilienhaus” mit drei konkreten Objekten.



Kommentare

Kommentare werden in UML wie folgt dargestellt:



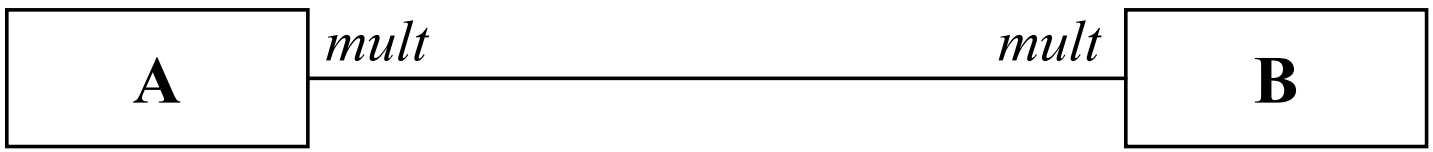
Überblick

Assoziationen

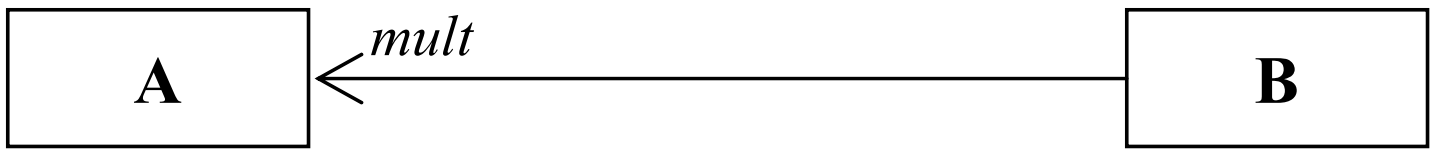
- Objekte verschiedener Klassen existieren nicht isoliert voneinander; es kann die folgenden drei Arten von Beziehungen geben:
  - Verwendungs- und Aufrufbeziehungen,
  - Aggregation und Komposition (“part-of”-Beziehungen),
  - Generalisierung und Spezialisierung (“is-a”-Beziehungen).

Allgemeine Assoziation

- Verwendungsbeziehungen sind die allgemeinste Form von Assoziationen zwischen Objekten verschiedener Klassen.
- Die Situation, dass Objekte der Klasse *A* Objekte der Klasse *B* verwenden und umgekehrt, stellt man in UML wie folgt dar:



- *mult* bezeichnet die *Multiplizität* der Assoziation.
- Die Assoziation kann auch gerichtet werden, z.B. wenn nur Objekte der Klasse *B* Objekte der Klasse *A* verwenden:

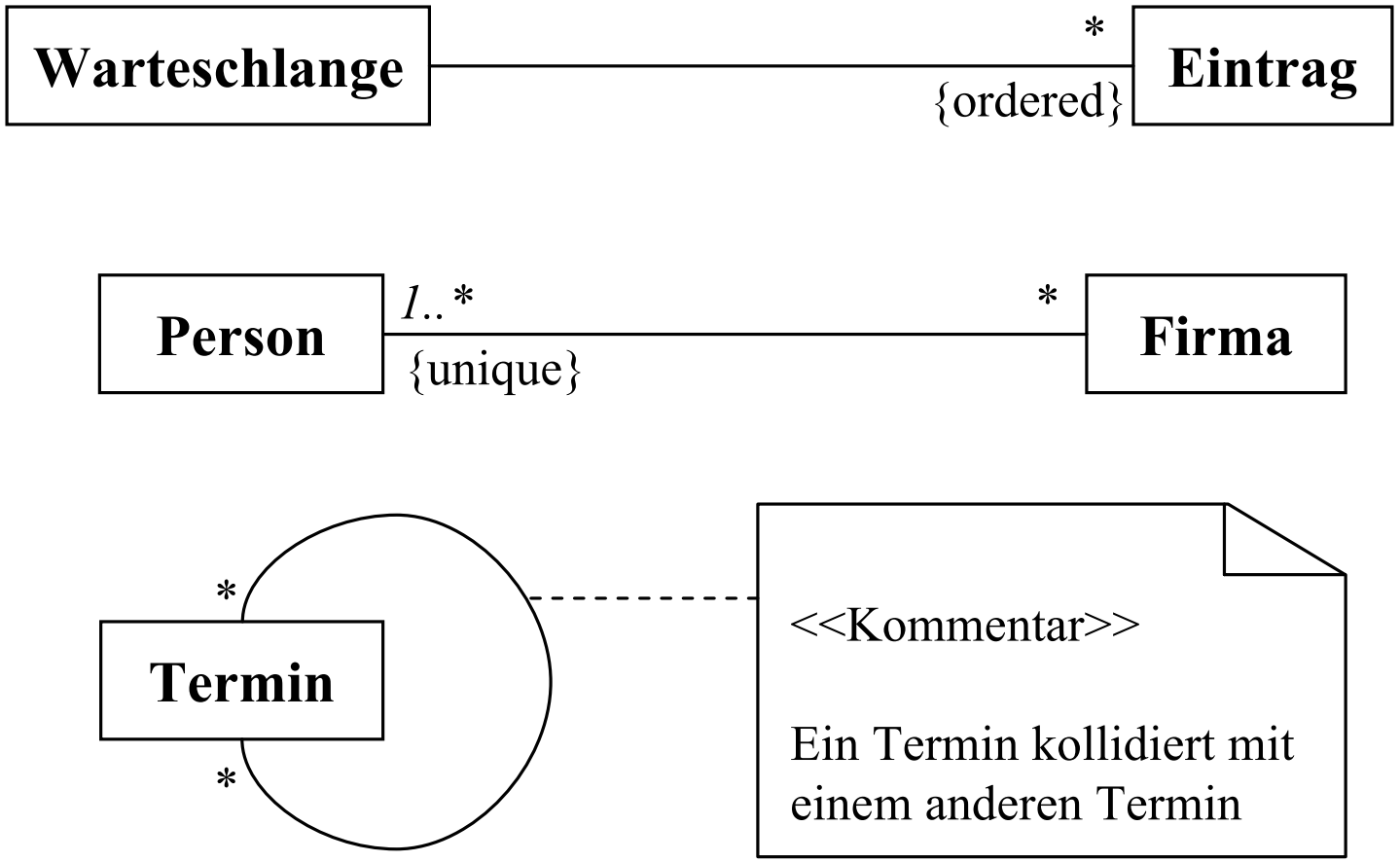


Multiplizität einer Assoziation

Für *mult* kann (u.a.) stehen:

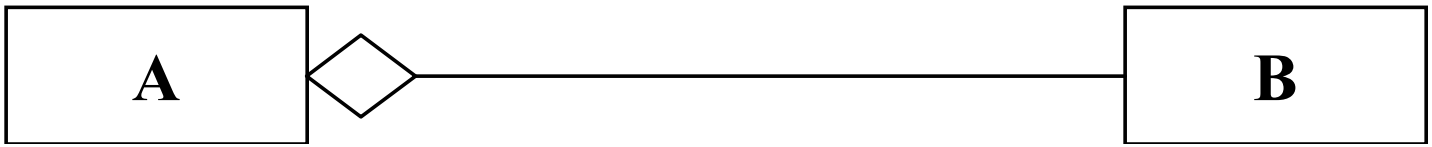
- * beliebig viele,
- *n*..*m* mindestens *n*, maximal *m*,
- Zusatz {unique}: die verwendeten Objekte sind alle paarweise verschieden,
- Zusatz {ordered}: die verwendeten Objekte sind geordnet (impliziert {unique}).

Beispiel: Assoziationen



Aggregation

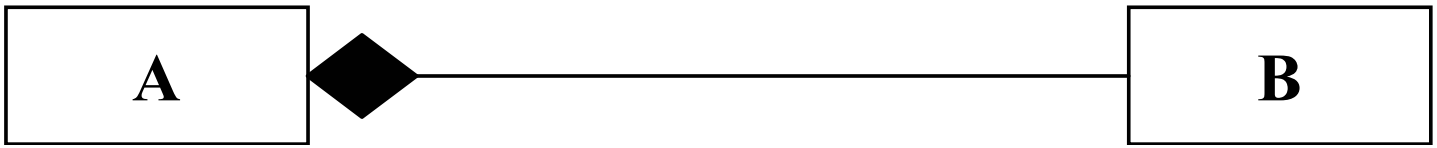
- Die Situation, dass Objekte der Klasse *A* aus Objekten der Klasse *B* zusammengesetzt sind, die Zusammensetzung aber *nicht* essentiell für die Existenz eines Objekts der Klasse *A* ist (*Aggregation*) stellt man in UML wie folgt dar:



- Auch hier kann man Multiplizitäten angeben.

Komposition

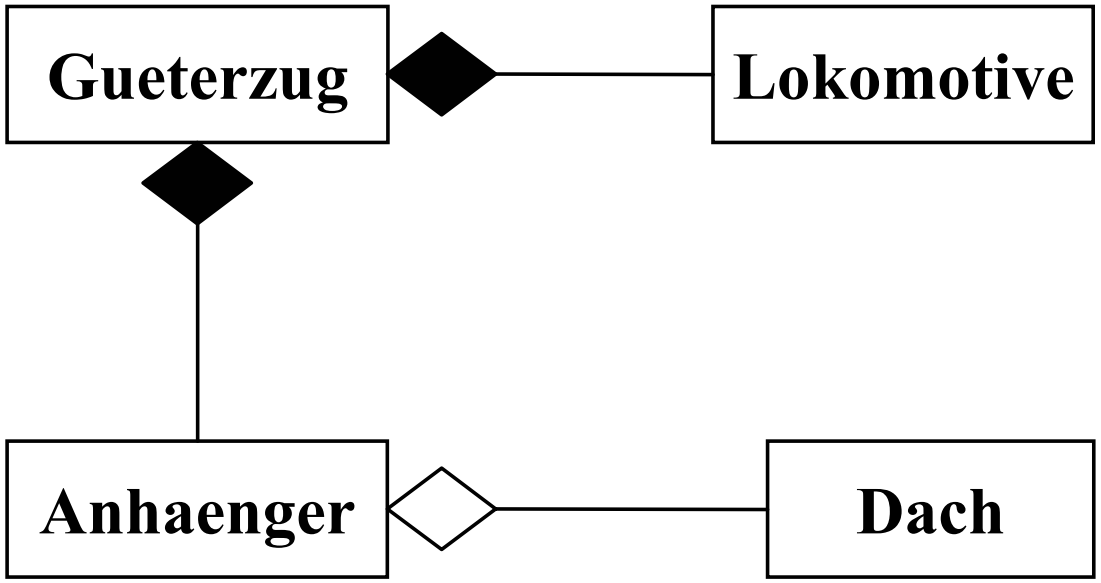
- Die Situation, dass Objekte der Klasse *A* aus Objekten der Klasse *B* zusammengesetzt sind und diese Zusammensetzung essentiell für die Existenz eines Objekts der Klasse *A* ist (*Komposition*) stellt man in UML wie folgt dar:



- Auch hier kann man Multiplizitäten angeben.

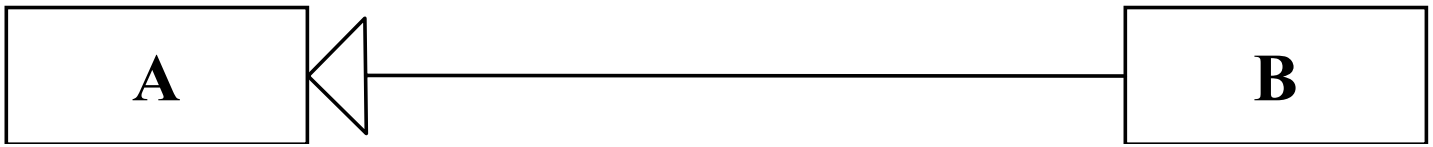
Beispiel: Aggregation und Komposition

Beispiele für Aggregation und Komposition:



Generalisierung/Spezialisierung

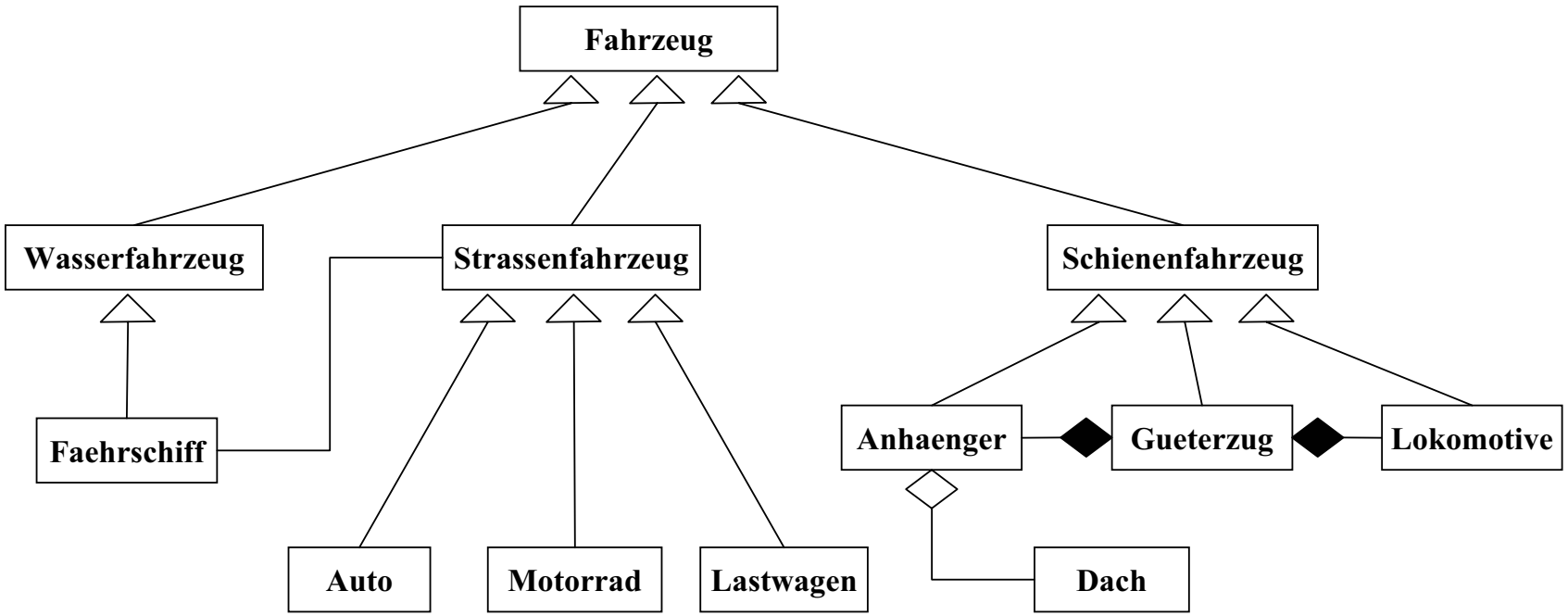
- Eine weitere wichtige Art der Beziehung zwischen Objekten verschiedener Klassen ist die Vererbungs-Relation (Generalisierung/Spezialisierung).
- In UML wird die Vererbungsbeziehung zwischen der Vaterklasse *A* und der abgeleiteten Klasse *B* dargestellt als:



- Insbesondere können nun überall dort, wo Objekte der Klasse *A* vorkommen dürfen, auch Objekte der Klasse *B* vorkommen. Welches oo Modellierungskonzept wird dabei realisiert?

Beispiel: Generalisierung/Spezialisierung

Unser Beispiel von vorher:



Welche grundlegenden oo Konzepte sind hier zu erkennen?

Überblick

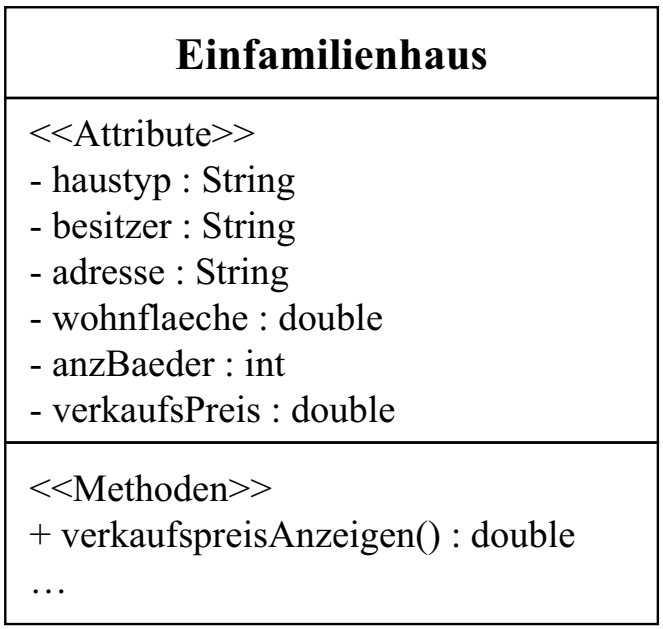
Klassendefinition

- Eine Klassendefinition in Java besteht aus dem Klassennamen und dem Klassenrumpf, der wiederum aus zwei Teilen besteht:
  - Die Attributdeklaration enthält die Liste der Attribute der Klasse. Jedes Attribut hat einen Namen und einen Typ. Dafür sind entweder primitive Typen, Reihungen oder andere Klassen (Objekttypen) erlaubt.
  - Die Methodendeklaration enthält die Liste der Methoden der Klasse. Als Typen für die Methodenparameter sind wiederum primitive Typen, Reihungen oder Objekttypen erlaubt.
- Bei allen Attributen und Methoden muss zusätzlich die *Sichtbarkeit* spezifiziert werden, d.h. ob andere Klassen das entsprechende Attribut oder die entsprechende Methode sehen und benutzen können. Dies ist für die Kapselung wichtig.
  - Das Schlüsselwort **`private`** bedeutet, dass die entsprechenden Attribute/Methoden von außerhalb nicht sichtbar sind (entspricht “-” bzw. “private” in UML).
  - Das Schlüsselwort **`public`** bedeutet, dass die entsprechenden Attribute/Methoden von außerhalb sichtbar sind (entspricht “+” bzw. “public” in UML).

Klassendefinition

- Beispiel: Die Klasse Einfamilienhaus in Java:

```
public class Einfamilienhaus
{
  // Attribute
  /** Der Haustyp des Hauses. */
  private String haustyp;

  /** Der Besitzer des Hauses. */
  private String besitzer;

  /** Die Adresse des Hauses. */
  private String adresse;

  /** Die Wohnfl&auml;che des Hauses. */
  private double wohnflaeche;

  /** Die Anzahl der B&auml;der im Haus. */
  private int anzBaeder;

  /** Der Verkaufspreis des Hauses. */
  private double verkaufsPreis;

  // Methoden
  ...
}
```



Klassendefinition

- Konventionen:
  - Klassennamen beginnen mit Großbuchstaben.
  - Attributsnamen und Methodennamen beginnen mit kleinen Buchstaben.
  - Konstantennamen bestehen komplett aus Großbuchstaben.

Überblick

Objekterzeugung

- Um ein Objekt der Klasse `Einfamilienhaus` zu erzeugen, muss man an der ensprechenden Stelle im Programm eine Variable vom Typ der Klasse deklarieren und ihr mit Hilfe des **`new`**-Operators ein neu erzeugtes Objekt zuweisen:

```
Einfamilienhaus efh4;
efh4 = new Einfamilienhaus();
```

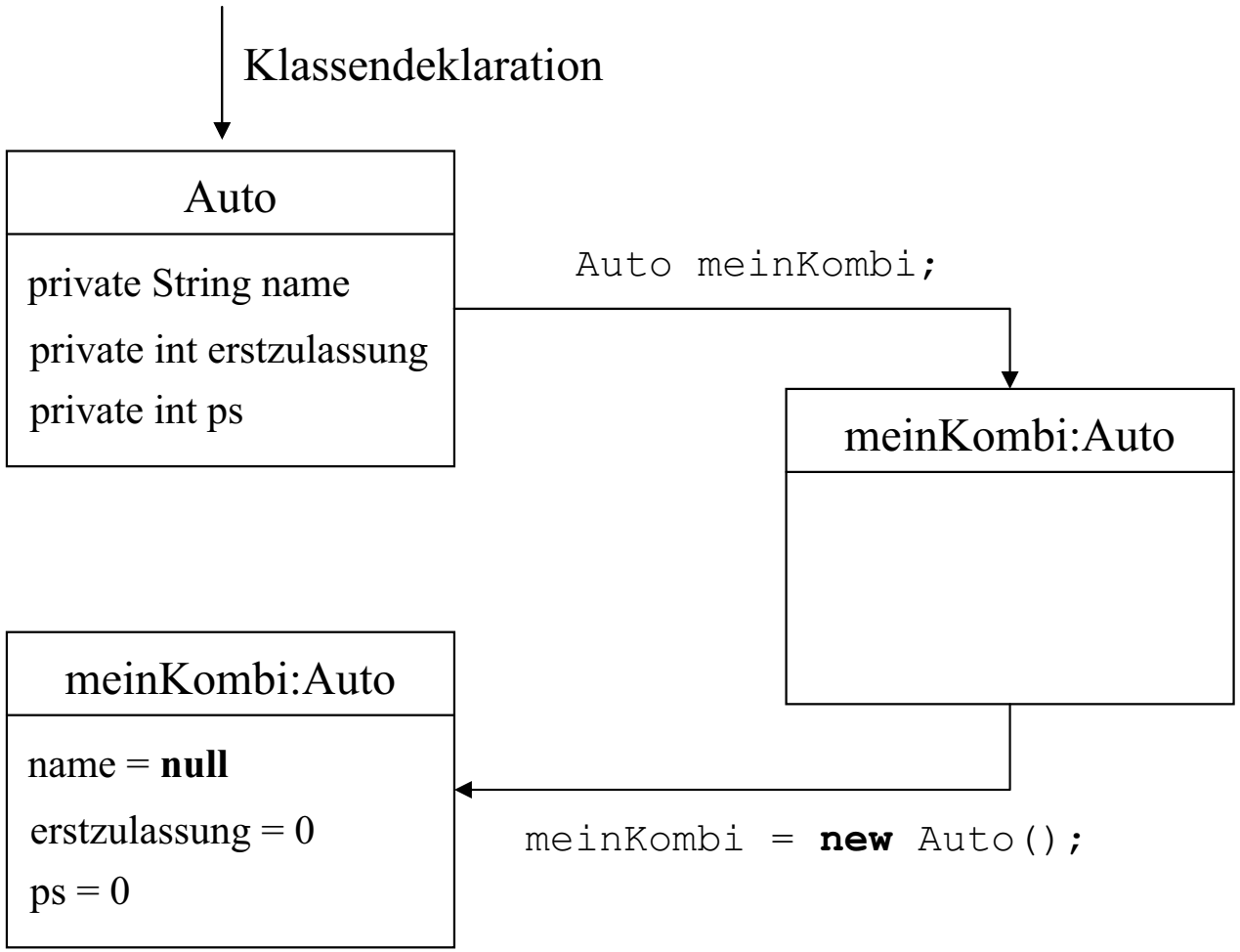
- Die erste Anweisung ist eine klassische Variablendeklaration. Anstelle eines primitiven Datentyps wird hier der Name einer zuvor definierten Klasse verwendet (Objekttyp).
- Die Variable `efh4` wird auf den Speicher gelegt mit einer Referenz auf die Halde. Dort wird später das konkrete Objekt gespeichert sein, das noch nicht generiert ist.
- Die zweite Anweisung generiert das Objekt mittels des **`new`**-Operators.
- Da Arrays auch Objekte sind, wissen Sie nun auch, warum man bei der Erzeugung eines Arrays den **`new`**-Operator benötigt (wenn man kein Literal benutzt).

Objekterzeugung

- Nach der Generierung eines Objekts haben alle Attribute mit primitiven Datentypen des Objekts zunächst ihre entsprechenden Standardwerte (siehe Arrays).
- Attribute mit Objekttypen haben den Standardwert **`null`**, die “leere Referenz”.
- Auf den Zustand und die Methoden eines Objekts kann nun wie folgt zugegriffen werden:
  - Auf alle mit **`private`** versehenen Attribute und Methoden kann außerhalb der Klasse *nicht* zugegriffen werden.
  - Auf alle mit **`public`** versehenen Attribute und Methoden kann außerhalb der Klasse über die Punktnotation zugegriffen werden: Wäre beispielsweise das Attribut **`int`** `verkaufsPreis` in der Deklaration der Klasse `Einfamilienhaus` mit der Sichtbarkeit **`public`** versehen (statt mit **`private`**), so könnte man mit `efh4.verkaufsPreis` darauf lesend/schreibend zugreifen.
- Offensichtlich widerspricht der direkte Zugriff auf den Zustand (Attribute) eines Objekts dem Prinzip der Kapselung. Daher ist es guter oo Programmierstil, Attribute grundsätzlich als **`private`** zu deklarieren.

Objekterzeugung

- Nocheinmal bildlich der Vorgang der Objekterzeugung am Beispiel einer Klasse `Auto`:



Objekterzeugung

- Nochmals zu Arrays:
  - Eigentlich (im Sinne der Kapselung) sollten Attribute als **`private`** vereinbart werden.
  - Was verbirgt sich hinter `beispielArray.length`?
  - Eigentlich ist das ein *Attribut*, auf das man von außerhalb der Klasse zugreifen kann, also **`public`** ist (Warum ist das keine Methode?).
  - Allerdings kann man (aus naheliegenden Gründen) hier nicht schreibend sondern nur lesend zugreifen.
- Tatsächlich sind Arrays besondere Objekte, zu denen es keine "echte" Klassendefinition gibt.
- Daher unterscheidet sich die "Schnittstelle" (die **`public`** Methoden/Attribute) von Arrays zu den Schnittstellen klassischer Objekte.
- Arrays werden ansonsten genauso behandelt wie "normale" Objekte.

Gleichheit von Objekten

- Was bedeutet, dass zwei Objekte gleich sind?
- Intuitiv: dass ihre Attribute dieselben Werte haben, d.h. dass sie denselben Zustand haben.

```
Einfamilienhaus villaKahn;
villaKahn = new Einfamilienhaus();
Einfamilienhaus hausBender;
villaBender = new Einfamilienhaus();

boolean vergleich = villaKahn == hausBender;  // (*)

villaKahn = hausBender;

vergleich = villaKahn == hausBender;          // (**)
```

- Was ist der Wert der Variablen `vergleich` an der Stelle `(*)`?

Gleichheit von Objekten

- Der Wert der Variablen `vergleich` an der Stelle `(*)` ist **`false`**!
- Warum? Sowohl das Objekt `villaKahn` als auch das Objekt `hausBender` haben doch als Attributwerte die Standardwerte, also haben sie insgesamt denselben Zustand.
- Richtig! Aber wir sind einem call-by-reference-Effekt bei Objekten auf den Leim gegangen.
- `villaKahn` ist eine Variable, die die Referenz zu einem speziellen Objekt speichert. Ebenso `hausBender`. Beide speichern unterschiedliche Referenzen, denn die Variablen repräsentieren ja unterschiedliche Objekte.
- Der Wert der Variablen `vergleich` an der Stelle `(**)` ist dagegen **`true`**.

Gleichheit von Objekten

- **Achtung:** Es gibt zwei “Arten” von Gleichheit von Objekten bzw. Objektvariablen (Variablen mit Objekttyp):
- Gleichheit: der Zustand der entsprechenden Objekte beider Objektvariablen ist gleich.
- Identität: beide Objektvariablen verweisen auf die gleiche Speicheradresse.
- Der Operator == prüft den zweiten Fall (Identität). Er kann also nicht dazu benutzt werden, abzufragen, ob die Objekte zweier Objektvariablen gleich bzgl. ihres Zustands sind.
- Dies wird oft übersehen und ist daher eine häufige Fehlerquelle!
- Um Gleicheit von Objekten zu prüfen, benötigt man andere Konstrukte, die wir später kennenlernen werden.

Überblick

Methoden

- Frage: Wenn der Zustand (alle Attribute) des Objekts gekapselt ist, wie kann ich dann auf den Zustand zugreifen bzw. wie kann ich ihn verändern?
- Antwort: Mit Hilfe der Methoden.
- Die Methoden einer Klasse definieren das Verhalten der Objekte der Klasse.
- Die Methoden einer Klasse haben Zugriff auf alle Attribute und Methoden der Klasse (auch auf den **`private`**-Teil).
- Methoden können wiederum von außen sichtbar (**`public`**) oder nicht sichtbar (**`private`**) sein. Die Methoden, die das Verhalten der Objekte spezifizieren, sollten alle sichtbar sein. Darüberhinaus kann es natürlich auch noch nicht sichtbare Hilfsmethoden geben.
- Beispiel: Klasse `Auto`

Methoden

```
public class Auto
{
  /** Der Modellname des Autos. */
  private String name;
  /** Das Jahr der Erstzulassung des Autos. */
  private int erstzulassung;
  /** Die Motorleistung des Autos in PS. */
  private int ps;

  /**
   * Gibt das Jahr der Erstzulassung des Auto-Objekts zur&uuml;ck.
   * @return  das Jahr der Erstzulassung des Autos.
   */
  public int getErstzulassung()
  {
    return erstzulassung;
  }

  /**
   * Ver&auml;ndert das Jahr der Erstzulassung des Auto-Objekts.
   * @param jahr  das Jahr der Erstzulassung des Autos.
   */
  public void setErstzulassung(int jahr)
  {
    erstzulassung = jahr;
  }

  /**
   * Berechnet das Alter (in Jahren) des Auto-Objekts.
   * @param aktuellesJahr  das aktuelle Jahr, z.B. 2007.
   * @return  das Alter des Autos.
   */
  public int alter(int aktuellesJahr)
  {
    return aktuellesJahr - erstzulassung;
  }
```

Methoden

- Alle Methoden sind sichtbar und haben ihrerseits Zugriff auf alle nicht sichtbaren Attribute der Klasse Auto.
- Kapselung: Das Jahr der Erstzulassung des Autos ist in diesem Beispiel nur über die Methoden `getErstzulassung` bzw. `setErstzulassung` von außerhalb lesend/schreibend zugreifbar. Dieser Zugriff ist durch die beiden Methoden wohldefiniert.

Methodenaufruf

- Die **`public`** Methoden können wie bereits erwähnt mit der Punktnotation aufgerufen werden.

```
Auto golf = new Auto();
golf.setErstzulassung(1998);
int e = golf.getErstzulassung(); // Wert von e ist 1998
int a = golf.alter(2007); // Wert von a ist 9
```

- Wie man auf der Folie davor sieht, darf eine Methode auf die Attribute (und auch auf die anderen Methoden) der Klasse ohne Punktnotation zugreifen.
- Tatsächlich bezieht der Compiler alle Variablen `x`, die nicht in Punktnotation verwendet werden, auf das Objekt **`this`**, d.h. `x` wird eigentlich als **`this`**`.x` interpretiert. Ausnahme: es gibt eine lokale Variable (z.B. in einem Methodenrumpf), die genauso heißt wie ein Klassenattribut.

Methodenaufruf

- Bei **`this`** handelt es sich um eine Art Objektvariable (also eine Referenz auf ein Objekt), die beim Anlegen eines Objekts automatisch generiert wird.
- **`this`** zeigt auf das aktuelle Objekt und wird dazu verwendet, die eigenen Methoden und Attribute anzusprechen.
- **`this`** kann auch explizit verwendet werden:

```
public int alter(int aktuellesJahr)
{
  return aktuellesJahr - this.erstzulassung;
}
```

- **`this`** muss explizit verwendet werden, wenn gleichnamige lokale Variablen (z.B. Parameter) verwendet werden.
- Die Verwendung von **`this`** auch da, wo es von der Syntax nicht verlangt wird, ist guter Stil, da die Lesbarkeit des Programmes verbessert wird.

Methodensignatur

- Die *Signatur* einer Methode setzt sich zusammen aus
  - dem Namen der Methode,
  - der Reihenfolge und Typen der Eingabeparameter,
  - dem Ausgabetyp.
- Methoden können grundsätzlich den selben Namen habe, solange sie sich in ihrer Parameterliste unterscheiden. Man spricht in diesem Fall von *Überladen*.
- Die beiden Methoden

```
public int alter(int aktuellesJahr)
{
  return aktuellesJahr - this.erstzulassung;
}
public int alter(double aktuellesJahr)
{
  return (int) aktuellesJahr - this.erstzulassung;
}
```

  sind verschieden.

Methodensignatur

- Dagegen sind die beiden Methoden

```
public int alter(int aktuellesJahr)
{
  return aktuellesJahr - this.erstzulassung;
}
public double alter(int aktuellesJahr)
{
  return (double) (aktuellesJahr - this.erstzulassung);
}
```

nicht verschieden! Warum?

Konstruktoren

- Bei unserer Beispielklasse `Auto` ist das Jahr der Erstzulassung von außen über die Methoden `setErstzulassung` veränderbar.
- Dies ist vermutlich nicht besonders glücklich: Der Wert des Attributs `erstzulassung` sollte einmal initialisiert werden können, dann aber nicht mehr verändert werden können.
- D.h., die Methode `setErstzulassung` sollte nicht zur Verfügung stehen.
- Dann kann das Attribut `erstzulassung` aber auch nicht verändert werden, schließlich ist es von außerhalb nicht sichtbar.

Konstruktoren

- Dieses Dilemma wird durch die *Konstruktoren* gelöst (auch wenn es nicht der Grund ist, warum es Konstruktoren überhaupt gibt).
- Konstruktoren sind spezielle Methoden, die zur Erzeugung und Initialisierung von Objekten aufgerufen werden können.
- Konstruktoren sind Methoden *ohne* Rückgabwert (nicht einmal **`void`**), die den Namen der Klasse erhalten.
- Konstruktoren können eine beliebige Anzahl von Eingabe-Parametern besitzen und überladen werden.
- Ein Konstruktor, der z.B. das Jahr der Erstzulassung als Parameter übergeben bekommt und das Attribut `erstzulassung` entsprechend initialisiert, löst also unser oben angesprochenes Dilemma.

Konstruktoren

```
public class Auto
{
  /** Der Modell-Name des Autos */
  private String name;
  /** Das Jahr der Erstzulassung des Autos */
  private int erstzulassung;
  /** Die Leistung des Autos in PS */
  private int ps;

  /**
   * Konstruktor. Generiert ein neues Objekt Auto.
   * @param name           der Name des Modells.
   * @param erstzulassung  das Jahr der Erstzulassung.
   * @param ps             die Leistung in PS.
   */
  public Auto(String name, int erstzulassung, int ps)
  {
    this.name = name;
    this.erstzulassung = erstzulassung;
    this.ps = ps;
  }

  // Methoden (jetzt ohne "setErstzulassung")
  ...
}
```

Konstruktoren

Da `erstzulassung` ein Attribut ist, dessen Wert sich nach der Initialisierung nicht mehr ändern soll, zeigt man das im Code am besten durch Verwendung einer Konstante:

```
public class Auto {
  /** Der Modell-Name des Autos */
  private String name;
  /** Das Jahr der Erstzulassung des Autos */
  private final int ERSTZULASSUNG;
  /** Die Leistung des Autos in PS */
  private int ps;

  /**
   * Konstruktor. Generiert ein neues Objekt Auto.
   * @param name           der Name des Modells.
   * @param erstzulassung  das Jahr der Erstzulassung.
   * @param ps             die Leistung in PS.
   */
  public Auto(String name, int erstzulassung, int ps)
  {
    this.name = name;
    this.ERSTZULASSUNG = erstzulassung;
    this.ps = ps;
  }

  // Methoden (jetzt ohne "setErstzulassung")
  ...
}
```

Konstruktoren

- Aufruf:

```
Auto golf;
golf = new Auto("Golf",1998,102);
int e = golf.getErstzulassung(); // Wert von e ist 1998
int a = golf.alter(2007); // Wert von a ist 9
```

- Wird kein expliziter Konstruktor deklariert, gibt es einen Default-Konstruktor (den wir bisher benutzt haben und der keine Eingabeparameter hat – `Auto()` in unserem Beispiel).
- Wird mindestens ein expliziter Konstruktor vereinbart, steht kein Default-Konstruktor zur Verfügung!
- **Bemerkung:** Neben den Konstruktoren gibt es noch die Möglichkeit, initiale Attributwerte in der Klassendefinition direkt zu vereinbaren (Die Attributsdefinitionen sehen dann so aus wie Variablenvereinbarungen mit Initialisierung). Offensichtlich löst dies aber nicht das vorher angesprochene Dilemma.

Überblick

Zusammenspiel von imperativen und objektorientierten Aspekten in Java

- Wie passen unsere imperativen Konstrukte aus Teil I hier dazu?
- Es gibt in Java *statische* und *nicht-statische* Elemente einer Klasse.
- Die statischen Elemente (Methoden oder Attribute) sind mit dem Schlüsselwort **`static`** gekennzeichnet.
- Fehlt **`static`**, ist das entsprechende Element (Methode/Attribut) nicht-statisch.
- Statische Elemente existieren unabhängig von Objekten, wogegen nicht-statische Elemente an die Existenz von Objekten gebunden sind.
- Werden statische Variablen von einem Objekt verändert, ist diese Veränderung auch in allen anderen Objekten der gleichen Klasse sichtbar.

Zusammenspiel von imperativen und objektorientierten Aspekten in Java

Beispiel:

- Ein imperatives Programm `Programm` in Java besteht aus einer Klassendefinition für die Klasse `Programm` mit einer statischen `main`-Methode.
- Diese Methode existiert unabhängig von Objekten der Klasse `Programm`.
- Die Klasse `Auto` in unserem vorherigen Beispiel spezifiziert ein nicht-statisches Attribut `name`.
- Dieses Attribut existiert nur dann, wenn es mindestens ein Objekt der Klasse `Auto` gibt. Für jedes existierende Objekt der Klasse `Auto` existiert ein Attribut `name`.
- Statische und nicht-statische Elemente können “nebeneinander” verwendet werden.
- Für statische Elemente stehen auch **`private`** und **`public`** zur Spezifikation der Sichtbarkeit zur Verfügung.

Zusammenspiel von imperativen und objektorientierten Aspekten in Java

Beispiel: Kontoverwaltung

- Klasse `Konto` spezifiziert die Konten einer Bank.
- Für jedes neueröffnete Konto wird eine neue, fortlaufende Kontonummer vergeben.
- Wie kann man dies realisieren?
- Z.B. mit einer statischen Variablen `aktuelleKNR`, die bei jeder neuen Kontoeröffnung gelesen wird (um die neue Kontonummer zu bestimmen) und anschließend inkrementiert wird. Dieses Attribut sollte **`private`** sein, damit nur die Objekte der Klasse darauf zugreifen können.
- Das heißt die Klasse `Konto` wird neben den nicht-statischen Attributen, die für jedes neue Objekt neu angelegt werden (z.B. für den Namen des Kontoinhabers) auch das statische Attribut `aktuelleKNR` haben, das unabhängig von den existierenden Objekten der Klasse `Konto` existiert und verwendet werden kann.

Zusammenspiel von imperativen und objektorientierten Aspekten in Java

```
public class Konto
{
  /* ** Statische (objekt-unabhaengige) Attribute ** */
  private static int aktuelleKNR = 1;

  /* ** Nicht-statische (objekt-abhaengige) Attribute ** */
  private String kundenName;
  private double kontoStand;
  private final int KONTO_NR;
  ... // weitere Attribute

  /* ** Konstruktor ** */
  public Konto(String kundenName)
  {
    this.kundenName = kundenName;
    kontoStand = 0.0;
    KONTO_NR = aktuelleKNR;
    aktuelleKNR++;
  }

  /* ** Methoden ** */
  ...
}
```

Beispiel: Wrapper-Klassen für primitive Typen

- Zu jedem primitiven Datentyp in Java gibt es eine korrespondierende (sog. *Wrapper*-) Klasse, die den primitiven Typ in einer oo Hülle kapselt.
- Es gibt Situationen, bei denen man diese Wrapper-Klassen anstelle der primitiven Typen benötigt. Z.B. werden in Java einige Klassen zur Verfügung gestellt, die eine (dynamische) Menge von beliebigen Objekttypen speichern können. Um darin auch primitive Typen ablegen zu können, benötigt man die Wrapper-Klassen.
- Zu allen numerischen Typen und zu den Typen **`char`** und **`boolean`** existieren Wrapper-Klassen.

Beispiel: Wrapper-Klassen für primitive Typen

Wrapper-Klasse	Primitiver Typ
`Byte`	**`byte`**
`Short`	**`short`**
`Integer`	**`int`**
`Long`	**`long`**
`Double`	**`double`**
`Float`	**`float`**
`Boolean`	**`boolean`**
`Character`	**`char`**
`Void`	**`void`**

Beispiel: Wrapper-Klassen für primitive Typen

- Zur Objekterzeugung stellen die Wrapper-Klassen haupsächlich zwei Konstruktoren zur Verfügung:
  - Für jeden primitiven Typ `type` stellt die Wrapperklasse `Type` einen Konstruktor zur Verfügung, der einen primitiven Wert des Typs `type` als Argument fordert:
    z.B. `public Integer(int i)`
  - Zusätzlich gibt es bei den meisten Wrapper-Klassen die Möglichkeit, einen String zu übergeben:
    z.B. `public Integer(String s)` wandelt die Zeichenkette `s` in einen Integer um, z.B. `"123"` in den `int`-Wert `123`.
- Kapselung:
  - Der Zugriff auf den Wert des Objekts erfolgt ausschließlich lesend über entsprechende Methoden:
    z.B. `public int intValue()`
  - Die interne Realisierung (wie wird der Wert gespeichert) ist dem Benutzer verborgen.
  - Insbesondere kann der Wert des Objekts nicht verändert werden.

Beispiel: Wrapper-Klassen für primitive Typen

- Statische Elemente (u.a.):
  - Wichtige Literale aus dem enstrechenden Wertebereich, z.B. Konstanten
    `public static int MAX_VALUE` bzw.
    `public static int MIN_VALUE`
    für den maximal/minimal darstellbaren **`int`**-Wert

    oder z.B. Konstanten
    `public static double NEGATIVE_INFINITY` bzw.
    `public static double POSITIVE_INFINITY`
    für $-\infty$ und $+\infty$
  - Hilfsmethoden wie z.B.
    `static double parseDouble(String s)` der Klasse `Double`
    wandelt die Zeichenkette `s` in ein primitiven **`double`**-Wert um und gibt den **`double`**-Wert aus.

Abschnitt 11: Beispiel: Die Klasse String (Teil 1)

Überblick

Zeichenketten

- Ein wichtiges Element für jede Programmiersprache sind Zeichenketten (z.B. zur Modellierung von Ein-/Ausgabe, Verarbeitung von Texten).
- Als grundlegenden Typ für (einzelne!) Zeichen kennen wir den primitiven Typ **`char`** und die Wrapper-Klasse `Character`.
- Zeichen*ketten*, also Sequenzen von mehreren Zeichen, werden durch die Klasse `String` repräsentiert.

Was sind Strings für den Compiler?

- Der Compiler kennt einige elementare Eigenschaften von Zeichenketten:
  - Er erkennt String-Literale (z.B. `"Hello, World!"`) und erzeugt daraus String-Objekte.
  - Er kann Strings verküpfen (`+`-Operator).
- Intern ist ein String eine Reihung von **`char`**s.
- Ein Großteil der Implementierung der Eigenschaften von Zeichenketten ist der Laufzeitbibliothek überlassen (z.B. Vergleich von Zeichenketten, Extraktion von Teilstrings).

Überblick

Die Klasse String

- Als Klasse hat String einige Besonderheiten:
  - Ein String kann aus Literalen erzeugt werden (s.o.).
  - Auf Strings ist ein Operator definiert (s.o.).
  - String-Objekte sind nicht dynamisch (dazu später).
- Dennoch betrachten wir Strings als erstes Beispiel für Objekte ausführlicher, denn als Modellierung von Zeichenketten ist die Klasse String von zentraler Bedeutung für die effiziente Programmentwicklung.
- Im Folgenden werfen wir einen Blick auf einige wichtige Methoden der Klasse String. Sie sollten sich mit Hilfe der Java-Dokumentation (`http://java.sun.com/javase/6/docs/api/`) weiter mit der Klasse String vertraut machen.

Statische Methoden

- Die Klasse String bietet statische Methoden an, um aus primitiven Typen Strings zu erzeugen:
  - `static String valueOf(boolean b)`
  - `static String valueOf(char c)`
  - `static String valueOf(char[] c)`
  - `static String valueOf(int i)`
  - `static String valueOf(long l)`
  - `static String valueOf(float f)`
  - `static String valueOf(double d)`
- Bei der Konkatenation (z.B. `"Note: "+1.0`) werden diese Methoden (hier: `static String valueOf(double d)`) implizit verwendet.
- Warum sind diese Methoden statisch?

Konstruktoren

Außer aus Literalen kann man Strings auch durch verschiedene Konstruktoren erzeugen. Die wichtigsten sind:

- `String()` – erzeugt ein neues String-Objekt, das den leeren String repräsentiert (eine **`char`**-Sequenz der Länge 0).
- `String(String original)` – erzeugt einen neuen String, der die gleiche **`char`**-Sequenz repräsentiert wie das angegebene Original. Es wird also eine Kopie (des Strings, nicht der Referenz!) erzeugt.
- `String(`**`char`**`[] value)` – erzeugt einen String aus dem gegebenen Array von **`char`**s. Das neue String-Objekt ist danach unabhängig vom **`char`**-Array, d.h. Änderungen am **`char`**-Array haben keinen Auswirkung auf den String.
- `String(`**`char`**`[] value, `**`int`**` offset, `**`int`**` count)` – wie `String(`**`char`**`[] value)`, wobei man aber noch einen Ausschnitt des Arrays spezifiziert.

String-Objekte sind nicht veränderbar

- Ein einmal erzeugter String kann nicht mehr verändert werden.
- Das bedeutet, String-Objekte sind nicht dynamisch, auch wenn das manche Methoden suggerieren.

Länge einer Zeichenkette

- Die Länge einer Zeichenkette entspricht der Länge des zugrundeliegenden **`char`**-Arrays.
- Ein leerer String hat die Länge 0. (Und ein String der Länge 0 ist immer leer.)
- Bei einer Länge $n > 0$ enthält ein String n Zeichen, die an den Indexpositionen 0 bis $n - 1$ liegen.
- Ein String-Objekt gibt über seine Länge Auskunft durch die Methode `int length()`.

```
String s = new String("Hello, World!");
int l = s.length();
System.out.println("Laenge von \""+s+"\": "+l);
// Ausgabe:
// Laenge von "Hello, World!": 13
```

Zeichenextraktion

- Die Methode `char charAt(int index)` liefert das Zeichen an der gegebenen Stelle des Strings.
- Das erste Element hat den Index 0.
- Das letzte Element hat den Index `length() - 1`.
- Beispiel: `"Hello, World!".charAt(12); //Wert: '!'`

Teilstring

- Die Methode `String substring(int beginIndex)` liefert den Teilstring von Stelle `beginIndex` an bis zum Ende des Strings.
- Die Methode
  `String substring(int beginIndex, int endIndex)` erlaubt zusätzlich, das Ende des zu extrahierenden Teilstrings anzugeben.

Achtung:

Ungewöhnlich ist bei dieser Methode, dass der Parameter `endIndex` die erste Stelle *nach* dem zu extrahierenden Teilstring angibt.

```
String s = "Hello, World!";
String s1 = s.substring(1,9);
// Wert: ello, Wo
String s2 = s.substring(12,13);
// Wert: !
```

- Die Methode `String trim()` gibt eine Kopie des Strings zurück, bei der führende und schließende Leerzeichen (= Code $\leq$ 32) entfernt wurden.

Ersetzen von Teilen einer Zeichenkette

- Die Methoden `String toLowerCase()` bzw. `String toUpperCase()` geben den String zurück, der entsteht, wenn man alle Großbuchstaben im aufrufenden Objekt durch Kleinbuchstaben ersetzt bzw. umgekehrt.
- Die Methode `String replace(char oldChar, char newChar)` gibt den String zurück, der durch Ersetzen aller Vorkommen von `oldChar` durch `newChar` entsteht.

Nochmal: String-Objekte sind nicht veränderbar

All diese Methoden (`substring`, `trim`, `toLowerCase`, `toUpperCase`, `replace` u.a.) verändern niemals das aufrufende Objekt (das wäre auch gar nicht möglich, weil ein String-Objekt unveränderlich ist), sondern erzeugen ein neues Objekt mit den gewünschten Eigenschaften.

```
String o1 = "Kroeger";
String o2 = o1.replace('o', 'i').substring(0, 6)+"l";
System.out.println(o2+" & "+o1);
// Ausgabe:
// Kriegel & Kroeger
```

Nochmal: String-Objekte sind nicht veränderbar

Soll nur ein String mit den neuen Eigenschaften übrigbleiben?

```
String o1 = "Kroeger";
o1 = o1.replace('o', 'i').substring(0, 6)+"l";
System.out.println(o1);
// Ausgabe:
// Kriegel
```

Was passiert hier im Speicher?

Nochmal: String-Objekte sind nicht veränderbar

```
1  String o1 = "Kroeger";
2  o1 = o1.replace('o', 'i');
3  o1 = o1.substring(0, 6);
4  o1 = o1 +"l";
```

Stack nach Zeile 1

o1 = <adr1>

Heap nach Zeile 1:

<adr1> : `"Kroeger"`

Nochmal: String-Objekte sind nicht veränderbar

```
1  String o1 = "Kroeger";
2  o1 = o1.replace('o', 'i');
3  o1 = o1.substring(0, 6);
4  o1 = o1 +"l";
```

Stack nach Zeile 2

o1 = <adr2>

Heap nach Zeile 2:

<adr1> : `"Kroeger"` <adr2> : `"Krieger"`

Nochmal: String-Objekte sind nicht veränderbar

```
1  String o1 = "Kroeger";
2  o1 = o1.replace('o', 'i');
3  o1 = o1.substring(0, 6);
4  o1 = o1 +"l";
```

Stack nach Zeile 3

o1 = <adr3>

Heap nach Zeile 3:

<adr1> : `"Kroeger"` <adr2> : `"Krieger"` <adr3> : `"Kriege"`

Nochmal: String-Objekte sind nicht veränderbar

```
1  String o1 = "Kroeger";
2  o1 = o1.replace('o', 'i');
3  o1 = o1.substring(0, 6);
4  o1 = o1 +"l";
```

Stack nach Zeile 4

o1 = <adr4>

Heap nach Zeile 4:

<adr1> : `"Kroeger"` <adr2> : `"Krieger"` <adr3> : `"Kriege"`
<adr4> : `"Kriegel"`

Nochmal: String-Objekte sind nicht veränderbar

```
1  String o1 = "Kroeger";
2  o1 = o1.replace('o', 'i');
3  o1 = o1.substring(0, 6);
4  o1 = o1 +"l";
```

Stack nach Zeile 4

o1 = <adr4>

Heap nach Zeile 4:

<adr1> : `"Kroeger"` <adr2> : `"Krieger"` <adr3> : `"Kriege"`
<adr4> : `"Kriegel"`

Es wird also immer mehr Speicher im Heap belegt. Auf die alten Zustände von `o1` kann allerdings nicht mehr zugegriffen werden (es gibt keine Referenz mehr), der entsprechende Speicherplatz kann also vom Garbage Collector bereinigt werden.

Überblick

Warum sind Strings nicht veränderbar?

- String-Objekte spielen in nahezu jedem Programm eine wichtige Rolle.
- Gefahr bei veränderlichen Objekten: Das Objekt wird an einer anderen (dem Entwickler vielleicht gar nicht bekannten) Stelle im Programm verändert.
- Als nicht veränderbares Objekt kann ein String gefahrlos von mehreren Stellen des Programmes referenziert werden.
- Außerdem erfordern nicht-veränderbare Objekte keinen Synchronisationsaufwand in nebenläufigen Programmen (*Multithreading*).
- Strings implementieren deshalb das Design-Pattern *Immutable*, das genau auf die genannten Vorteile abzielt.

Designmerkmale einer *Immutable*-Klasse

- Alle Attribute sind **`private`**.
- Schreib-Zugriffe auf Attribute erfolgen ausschließlich im Konstruktor oder in Initialisierungsmethoden.
- Lesende Zugriffe auf Attribute sind nur erlaubt, wenn das Attribut selbst immutable oder ein primitiver Typ ist. Auf veränderliche Objekte oder Arrays als Attribute darf keine Zugriffsmöglichkeit bestehen.
- Wenn veränderliche Objekte oder Arrays an einen Konstruktor übergeben werden, dann müssen sie kopiert (geklont) werden, bevor sie Attributen zugewiesen werden dürfen.

Dynamisches Arbeiten mit Strings

- Es ist also ein durchaus erwünschtes Verhalten der String-Klasse, dass ein einmal im Speicher liegendes String-Objekt seinen Wert nicht verändert.
- Nicht wünschenswert ist hingegen, dass sich der Speicher immer mehr mit “toten” String-Objekten füllt, auf die gar nicht mehr zugegriffen werden kann, denn eine Zeichenkette im Laufe eines Programmes dynamisch verändern zu können, ist durchaus auch eine häufige Anforderung.
- Hierfür gibt es eigene Klassen, deren Verwendung viele Programme deutlich beschleunigen kann, den `StringBuilder` und den `StringBuffer`.
  - Der `StringBuffer` ist seit langem gebräuchlich. Ein `StringBuffer` ist Thread-Safe.
  - Der `StringBuilder` wurde in Version 1.5 eingeführt. Wenn man nicht nebenläufig programmiert, sollte man den `StringBuilder` bevorzugen, der keinen Synchronisationsaufwand betreibt.

  Beide Klassen haben ansonsten ähnliche Methoden und zeigen ähnliches Verhalten.

Methoden von StringBuilder und StringBuffer

- Die gebräuchlichste Methode ist die `append`-Methode. Sie hängt den übergebenen Parameter an die momentante Zeichenkette an. Sie ist überladen und kann mit jedem primitiven Typ, Strings, einem StringBuilder bzw. StringBuffer und sogar jedem anderen Objekt aufgerufen werden.
- Ebenso überladen ist die `insert`-Methode, die den übergebenen Wert an einer ebenfalls als Parameter angegebenen Stelle einfügt.
- Weitere interessante Methoden sind `replace`, `reverse` und `substring`.
- Wenn die Zeichenkette schließlich feststeht, erhält man durch Aufruf der Methode `toString` den entsprechenden String.

Machen Sie sich mit diesen Klassen durch die Java-Dokumentation vertraut!

Abschnitt 12: Vererbung und Polymorphismus

Überblick

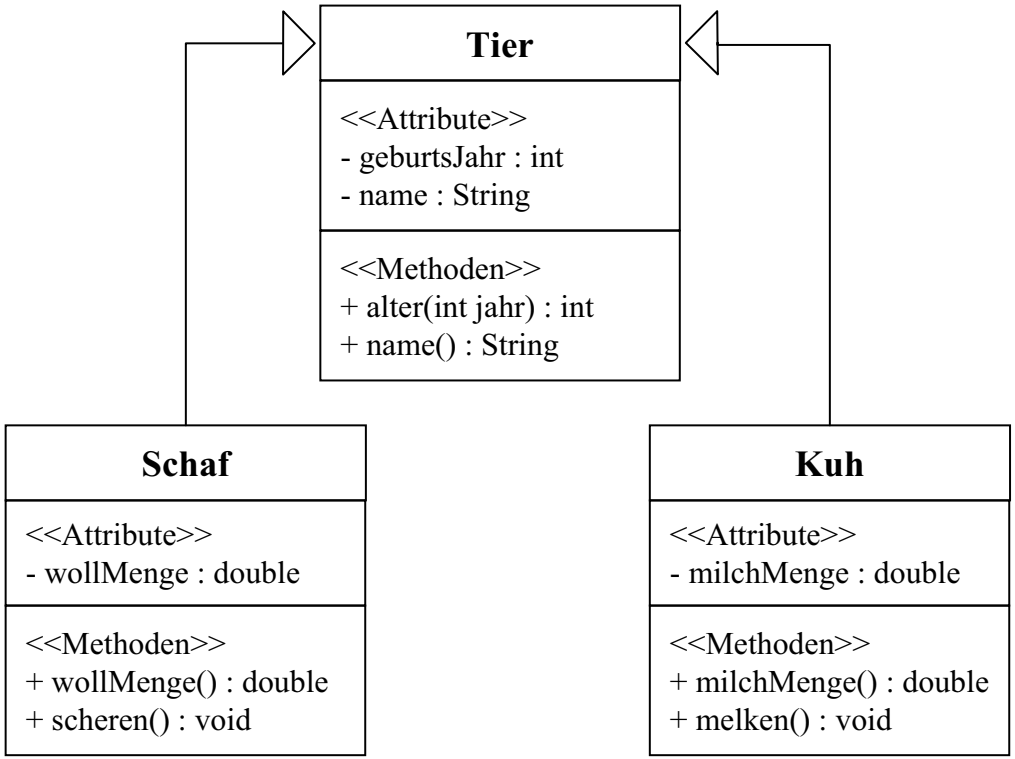
Vererbung

- Vererbung ist die Umsetzung von “is-a”-Beziehungen.
- Alle Elemente (Attribute und Methoden) der Vaterklasse sollen auf die abgeleitete Klasse vererbt werden.
- Zusätzlich kann die abgeleitete Klasse neue Elemente (Attribute und Methoden) definieren.
- Vererbung ist ein wichtiges Mittel zur Wiederverwendung von Programmteilen.

Beispiel

- Wir wollen das Tierreich (bestehend aus Schafen und Kühen) auf einem Bauernhof modellieren.
- Dazu stellen wir fest, dass beide Tierarten gemeinsame Eigenschaften haben, z.B. einen Namen, ein Geburtsjahr, etc.
- Zusätzlich haben beide Tierarten unterschiedliche Eigenschaften, bei den Kühen interessieren wir uns z.B. für die Milchmenge, die sie letztes Jahr abgegeben haben, bei den Schafen interessiert uns die Menge der Wolle, wir scheren konnten, etc.
- Dies kann man mit Vererbung entsprechend modellieren (siehe nächste Folie).

Beispiel



Was wird hier wiederverwendet?

Überblick

Ableiten einer Klasse

- In Java wird nur die einfache Vererbung (eine Klasse wird von genau einer Vaterklasse abgeleitet) direkt unterstützt.
- Mehrfachvererbung (eine Klasse wird von mehr als einer Vaterklasse abgeleitet) muss in Java mit Hilfe von *Interfaces* umgesetzt werden (siehe später).
- In Java zeigt des Schlüsselwort **`extends`** Vererbung an.

  Beispiel:

```
public class Kuh extends Tier
{
  ...
}
```

- Objekte der Klasse `Kuh` erben damit alle Attribute und Methoden der Klasse `Tier`.

Die Klasse `Object`

- Enthält eine Klasse keine **`extends`**-Klausel, so besitzt sie die implizite Vaterklasse `Object` (im Paket `java.lang`).
- Also: Jede Klasse, die keine **`extends`**-Klausel enthält, wird direkt von `Object` abgeleitet.
- Jede explizit abgeleitete Klasse ist am oberen Ende ihrer Vererbungshierarchie von einer Klasse ohne explizite Vaterklasse abgeleitet und ist damit ebenfalls von `Object` abgeleitet.
- Damit ist `Object` die Vaterklasse aller anderen Klassen.

Die Klasse `Object`

- Die Klasse `Object` definiert einige elementare Methoden, die für alle Arten von Objekten nützlich sind, u.a.:
  - **`boolean`** `equals(Object obj)`

    testet die Gleichheit zweier Objekte, d.h. ob zwei Objekte den gleichen Zustand haben.
  - `Object clone()`

    kopiert ein Objekt, d.h. legt eine neues Objekt an, das eine genaue Kopie des ursprünglichen Objekts ist.
  - `String toString()`

    erzeugt eine `String`-Repräsentation des Objekts.
  - etc.
- Damit diese Methoden in abgeleiteten Klassen sinnvoll funktionieren, müssen sie bei Bedarf überschrieben werden.

Vererbung von Attributen und Methoden

- Neben ererbten Attributen und Methoden dürfen neue Attribute und Methoden in der abgeleiteten Klasse definiert werden.
- Es dürfen aber auch Attribute und Methoden, die von der Vaterklasse geerbt wurden, neu definiert werden.
- Bei Attributen tritt dabei der Effekt der *Versteckens* auf: Das Attribut der Vaterklasse ist in der abgeleiteten Klasse nicht mehr sichtbar.
- Bei Methoden tritt zusätzlich der Effekt des *Überschreibens* (auch: *Überlagerns*) auf: Die Methode modelliert in der abgeleiteten Klasse i.d.R. ein anderes Verhalten als in der Vaterklasse.

Vererbung von Attributen und Methoden

- Ist ein Element (Attribut oder Methode) `x` der Vaterklasse in der abgeleiteten Klasse neu definiert, wird also immer, wenn `x` in der abgeleiteten Klasse aufgerufen wird, das neue Element `x` der abgeleiteten Klasse angesprochen.
- Will man auf das Element `x` der Vaterklasse zugreifen, kann dies mit dem expliziten Hinweis **`super`**`.x` erreicht werden.
- Ein kaskadierender Aufruf von Vaterklassen-Elementen (z.B. **`super.super.`**`x`) ist nicht erlaubt!
- **`super`** ist eine Art Verweis auf die Vaterklasse (Vorsicht: *kein* Zeiger auf ein ensprechendes Objekt).

Die Sichtbarkeitsklasse `protected`

- Zur Spezifikation der Sichtbarkeit von Attributen und Methoden einer Klasse hatten wir bisher kennengelernt:
  - **`public`**: das Element ist in allen Klassen sichtbar.
  - **`private`**: das Element ist nur in der aktuellen Klasse sichtbar, also auch *nicht* in abgeleiteten Klassen!
- Zusätzlich gibt es das Schlüsselwort **`protected`**.
- Elemente des Typs **`protected`** sind in der Klasse selbst und in den Methoden abgeleiteter Klassen sichtbar.

Verstecken von Attributen

- Das Verstecken von Attributen ist eine gefährliche Fehlerquelle und sollte daher grundsätzlich vermieden werden.
- Meist geschieht das Verstecken von Attributen aus Unwissenheit über die Attribute der Vaterklasse.
- Problematisch ist, wenn Methoden aus der Vaterklasse vererbt werden, die auf ein verstecktes Attribut zugreifen, deren Name aber durch den Verstecken-Effekt irreführend wird, weil es ein gleich benanntes neues Attribut in der abgeleiteten Klasse gibt.

Verstecken von Attributen: Beispiel

- Die Klasse `Tier` definiert das Attribut `name` in dem der Name des Tieres gespeichert ist.
- Zudem gibt es eine Methode `public String getName()`, die den Wert des Attributs `name` zurrückgibt.
- In der abgeleiteten Klasse `Kuh` gibt es nun ebenfalls ein Attribut `name`, in dem der Name des Besitzers gespeichert werden soll. Die Methode `getName()` aus der Vaterklasse wird nicht überschrieben.
- Wenn nun ein Objekt der Klasse `Kuh` die Methode `getName()` aufruft, wird der Name der Kuh ausgegeben und nicht der Name des Besitzers.

Überschreiben von Methoden

- Das Überschreiben von Methoden ist dagegen ein gewünschter Effekt, denn eine abgeleitete Klasse zeichnet sich eben gerade durch ein unterschiedliches Verhalten gegenüber der Vaterklasse aus.
- Late Binding bei Methodenaufrufen: Erst zur Laufzeit wird entschieden, welcher Methodenrumpf nun ausgeführt wird, d.h. von welcher Klasse das aufrufende Objekt nun ist.
- Dieses Verhalten bezeichnet man auch als *dynamisches Binden*.
- Beispiel: eine Variable vom Typ `Tier` kann Objekte vom Typ `Tier`, Objekte vom Typ `Kuh` oder Objekte vom Typ `Schaf` enthalten.
- Es kann erst zur Laufzeit entschieden werden, welchen Typ die Variable aktuell hat.
- Überschreiben von Methoden und dynamisches Binden ist ein wichtiges Merkmal von Polymorphismus.

Warum keine Mehrfachvererbung?

- Wie bereits erwähnt, ist Mehrfachvererbung in Java nicht erlaubt.
- Der Grund hierfür ist, dass bei Mehrfachvererbung verschiedene Probleme auftauchen können.
- Ein solches Problem kann im Zusammenhang mit Methoden entstehen, die aus beiden Vaterklassen vererbt werden und nicht in der abgeleiteten Klasse überschrieben werden.
- In diesem Fall ist unklar, welche Methode ausgeführt werden soll, wenn eine dieser Methoden in der abgeleiteten Klasse aufgerufen wird.

Warum keine Mehrfachvererbung?

- Beispiel: Klasse `AmphibienFahrzeug` wird von den beiden Klassen `LandFahrzeug` und `WasserFahrzeug` abgeleitet.
- Die Methode `getPS()`, die die Leistung des Fahrzeugs zurückgibt, könnte bereits in den beiden Vaterklassen implementiert sein und nicht mehr in der Klasse `AmphibienFahrzeug` überschrieben werden.
- Problem:
```
AmphibienFahrzeug a = new AmphibienFahrzeug();
int ps = a.getPS();
```
- Welche Methode `getPS()` wird ausgeführt?

Konstruktoren und Vererbung

- Grundsätzlich gilt: Konstruktoren werden *nicht* vererbt!
- Dies ist auch sinnvoll, schließlich kann ein Konstruktor der Klasse `Tier` keine Objekte der spezielleren Klasse `Kuh` erzeugen, sondern eben nur Objekte der generelleren Klasse `Tier`.
- Es müssen also (wenn dies gewünscht ist), in jeder abgeleiteten Klasse eigene explizite Konstruktoren definiert werden.

Konstruktoren und Vererbung

- Wenn ein Objekt durch Aufruf des **`new`**-Operators und eines entsprechenden Konstruktors erzeugt wird, wird grundsätzlich immer auch (explizit oder implizit) der Konstruktor der Vaterklasse aufgerufen.
- Explizit kann man dies durch Aufruf von **`super`**`(<Parameterliste>)` als ersten Befehl in einem expliziten Konstruktor erreichen. `<Parameterliste>` kann leer sein (Default-Konstruktor) oder muss zur Signatur eines expliziten Konstruktors der Vaterklasse passen.
- Steht in einem expliziten Konstruktor der abgeleiteten Klasse kein **`super`**-Aufruf an erster Stelle, wird implizit der Default-Konstruktor der Vaterklasse **`super`**`()` aufgerufen.
- **Achtung:**
  In beiden Fällen gilt: es ist nicht erlaubt, den Default-Konstruktor aufzurufen, obwohl ein expliziter Konstruktor in der Vaterklasse vorhanden ist und der Default-Konstruktor nicht existiert.

Konstruktoren und Vererbung: Beispiel

1.Fall:

In `Tier` ist *kein* expliziter Konstruktor definiert

```
public class Kuh extends Tier
{
  public Kuh(double bisherigeMilchMenge) {
    this.milchMenge = bisherigeMilchMenge;  // (*)
  }

}
```

Beim Aufruf des Konstruktors, z.B. `Kuh erni = new Kuh(35.0);` wird, bevor Zeile `(*)` ausgeführt wird, zunächst der Konstruktor `Tier()` implizit aufgerufen.

Konstruktoren und Vererbung: Beispiel

2.Fall:

In `Tier` *ist* ein expliziter Konstruktor
`Tier(String name, int geburtsJahr)`
definiert.

```
public class Kuh extends Tier
{
  public Kuh(String name, int geburtsjahr, double  bisherigeMilchMenge) {
    super(name, geburtsjahr);
    this.milchMenge = bisherigeMilchMenge;
  }
}
```

Die Lösung von Fall 1 geht hier nicht, da der (implizit vor Zeile `( * )` aufgerufene) Default-Konstruktor nicht existiert.

Konstruktoren und Vererbung

- Offenbar dient der Aufruf des Vaterklassen-Konstruktors dazu, die vererbten Attribute in der abgeleiteten Klasse zu initialisieren.
- Natürlich kann dies auch in der abgeleiteten Klasse explizit gemacht werden (ist aber meist wenig sinnvoll).
- Konstruktoren werden nicht vererbt, müssen aber (implizit oder explizit) in abgeleiteten Klassen verwendet werden (können).

Überblick

Abstrakte Methoden

- *Abstrakte* Methoden enthalten im Gegensatz zu konkreten Methoden nur die Spezifikation der Signatur.
- Abstrakte Methoden enthalten also keinen Methodenrumpf, der die Implementierung der Methode vereinbart.
- Abstrakte Methoden werden mit dem Schlüsselwort **`abstract`** versehen und anstelle der Blockklammern für den Methodenrumpf mit einem simplen Semikolon beendet.
- Beispiel:

```
public abstract <Typ> abstrakteMethode(<Parameterliste>);
```

Abstrakte Klassen

- Abstrakte Methoden können nicht aufgerufen werden, sondern definieren eine Schnittstelle: Erst durch Überschreiben in einer abgeleiteten Klasse und (dortige) Implementierung des Methodenrumpfes wird die Methode konkret und kann aufgerufen werden.
- Klassen, die mindestens eine abstrakte Methode haben, sind selbst abstrakt und müssen ebenfalls mit dem Schlüsselwort **`abstract`** gekennzeichnet werden.
- Eine von einer abstrakten Vaterklasse abgeleiteten Klassen wird konkret, wenn alle abstrakten Methoden der Vaterklasse implementiert sind. Die Konkretisierung kann auch über mehrere Vererbungsstufen erfolgen.
- Es können keine Objekte (Instanzen) von abstrakten Klassen erzeugt werden!

Ein Beispiel für Polymorphismus

- Im folgenden sehen wir uns ein Programm an, das die Mitarbeiter der LMU verwaltet, insbesondere deren brutto Monatsgehalt berechnet.
- Dazu wird zunächst die abstrakte Klasse `Mitarbeiter` definiert, die alle grundlegenden Eigenschaften eines Mitarbeiters modelliert.

  `http://www.dbs.ifi.lmu.de/Lehre/EIP/WS_2008/skript/programmbeispiele/vererbung/Mitarbeiter.java`
- In abgeleiteten Klassen werden dann die einzelnen Mitarbeitertypen `Arbeiter`, `Angestellter` und `Beamter` abgebildet und konkret implementiert.

  `http://www.dbs.ifi.lmu.de/Lehre/EIP/WS_2008/skript/programmbeispiele/vererbung/Arbeiter.java`
  `http://www.dbs.ifi.lmu.de/Lehre/EIP/WS_2008/skript/programmbeispiele/vererbung/Angestellter.java`

  `http://www.dbs.ifi.lmu.de/Lehre/EIP/WS_2008/skript/programmbeispiele/vererbung/Beamter.java`
- Die Klasse `Gehaltsberechnung` verwendet diese Klassen polymorph:

  `http://www.dbs.ifi.lmu.de/Lehre/EIP/WS_2008/skript/programmbeispiele/vererbung/Gehaltsberechnung.java`

Ein Beispiel für Polymorphismus

```
public abstract class Mitarbeiter
{
    private int persNr;
    private String name;
    private int dienstAlter;


    public Mitarbeiter(int persNr, String name)
    {
        this.persNr = persNr;
        this.name = name;
        this.dienstAlter = 0;
    }

    public abstract double monatsBrutto();
}
```

Ein Beispiel für Polymorphismus

```
public class Arbeiter extends Mitarbeiter
{
    private double stundenLohn;
    private double anzahlStunden;
    private double ueberstundenZuschlag;
    private double anzahlUeberstunden;

    public Arbeiter(int persNr, String name,
                    double sL, double aS, double uZ, double aU)
    {
        super(persNr, name);
        this.stundenLohn = sL;
        this.anzahlStunden = aS;
        this.ueberstundenZuschlag = uZ;
        this.anzahlUeberstunden = aU;
    }

    public double monatsBrutto()
    {
        return   stundenLohn * anzahlStunden +
                 (stundenLohn + ueberstundenZuschlag)
                                    * anzahlUeberstunden;
    }
}
```

Ein Beispiel für Polymorphismus

```
public class Angestellter extends Mitarbeiter
{
    private double grundGehalt;
    private double ortsZuschlag;
    private double zulage;

    public Angestellter(int persNr, String name, double gG, double oZ, double z)
    {
        super(persNr, name);
        this.grundGehalt = gG;
        this.ortsZuschlag = oZ;
        this.zulage = z;
    }

    public double monatsBrutto()
    {
        return grundGehalt + ortsZuschlag + zulage;
    }
}
```

Ein Beispiel für Polymorphismus

```
public class Beamter extends Mitarbeiter
{
    private double grundGehalt;
    private double familienZuschlag;
    private double stellenZulage;

    public Beamter(int persNr, String name, double gG, double fZ, double sZ)
    {
        super(persNr, name);
        this.grundGehalt = gG;
        this.familienZuschlag = fZ;
        this.stellenZulage = sZ;
    }

    public double monatsBrutto()
    {
        return grundGehalt + familienZuschlag + stellenZulage;
    }
}
```

Ein Beispiel für Polymorphismus

```
public class Gehaltsberechnung
{
    public static void main(String[] args)
    {
        Mitarbeiter[] ma = new Mitarbeiter[3];

        ma[0] = new Beamter(1, "Meier", 3021.37, 91.50, 10.70);
        ma[1] = new Angestellter(2, "Maier", 2303.21, 502.98, 132.65);
        ma[2] = new Arbeiter(3, "Mayr", 20.0, 113.5, 35.0, 11.0);

        double bruttoSumme = 0.0;

        for(int i=0; i<ma.length; i++)
        {
            bruttoSumme += ma[i].monatsBrutto();
        }

        System.out.println("Bruttosumme = "+bruttoSumme);
    }
}
```

Abschnitt 13: Strukturierung von Java-Programmen: Packages

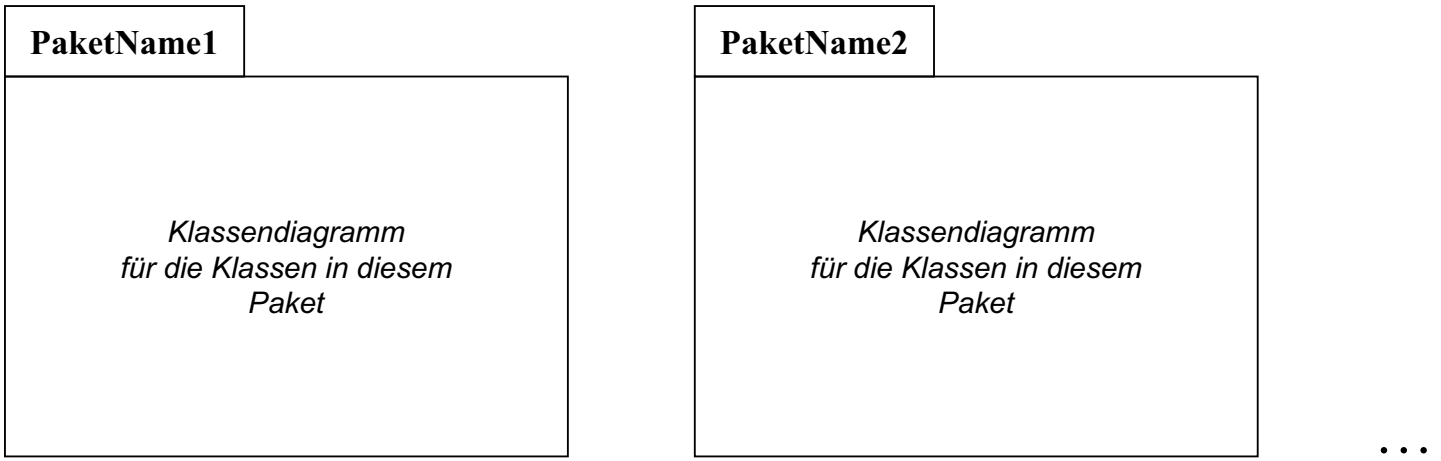
Überblick

Pakete in der Software-Entwicklung

- Bei großen Programmen entstehen viele Klassen und Schnittstellen.
- Um einen Überblick über diese Menge zu bewahren, wird ein Strukturierungskonzept benötigt, das von den Details abstrahiert und die übergeordnete Struktur verdeutlicht.
- Ein solches Strukturierungskonzept stellen die *Pakete* (*packages*) dar. Packages erlauben es, Komponenten zu größeren Einheiten zusammenzufassen.
- Die meisten Programmiersprachen bieten dieses Strukturierungskonzept (teilweise unter anderen Namen) an.

Packages in UML

- Packages gruppieren Klassen, die einen gemeinsamen Aufgabenbereich haben.
- In einem UML Klassendiagramm kann dies folgendermaßen notiert werden:



- Bemerkung: Es kann Beziehungen zwischen Klassen unterschiedlicher Pakete geben.
- Welche grundlegenden oo Modellierungsaspekte werden durch das Konzept der Pakete realisiert?

Packages in Java

- In Java können Klassen zu Packages zusammengefasst werden.
- Packages dienen in Java dazu,
  - große Gruppen von Klassen, die zu einem gemeinsamen Aufgabenbereich gehören, zu bündeln,
  - potentielle Namenskonflikte zu vermeiden,
  - Zugriffe und Sichtbarkeit zu definieren und kontrollieren,
  - eine Hierarchie von verfügbaren Komponenten aufzustellen.

Packages in Java

- Jede Klasse in Java ist Bestandteil von genau einem Package.
- Ist eine Klasse nicht explizit einem Package zugeordnet, dann gehört es implizit zu einem *Default*-Package.
- Packages sind hierarchisch gegliedert, können also Unterpackages enthalten, die selbst wieder Unterpackages enthalten, usw.
- Die Package-Hierarchie wird durch Punktnotation ausgedrückt:
  `package.unterpackage1.unterpackage2. ... .Klasse`

Import von Packages

- Der vollständige Name einer Klasse besteht aus dem Klassen-Namen *und* dem Package-Namen: `packagename.KlassenName`
- Package-Namen bestehen nach Konvention immer aus Kleinbuchstaben.
- Um eine Klasse verwenden zu können, muss angegeben werden, in welchem Package sie sich befindet. Dies kann auf zwei Arten geschehen:
  1. Die Klasse wird an der entsprechenden Stelle im Programmtext über den vollen Namen angesprochen:
     ```
     java.util.Random einZufall = new java.util.Random();
     ```
  2. Am Anfang des Programms werden die gewünschten Klassen mit Hilfe einer **`import`**-Anweisung eingebunden:
     ```
     import java.util.Random;
     ...
     Random einZufall = new Random();
     ```
- Achtung: werden zwei Klassen gleichen Namens aus unterschiedlichen Packages importiert, müssen die Klassen trotz **`import`**-Anweisung mit vollem Namen aufgerufen werden!

Import von Packages

- Klassen des Default-Packages können ohne explizite **`import`**-Anweisung bzw. ohne vollen Namen verwendet werden.
- Wird in der **`import`**-Anweisung eine Klasse angegeben, wird genau diese Klasse importiert. Alle anderen Klassen des entsprechenden Packages bleiben unsichtbar.
- Will man alle Klassen eines Packages auf einmal importieren, kann man dies mit der folgenden **`import`**-Anweisung:
  `import` `packagename.*;`
- Achtung: es werden dabei wirklich nur die Klassen aus dem Package `packagename` eingebunden und *nicht* etwa auch die Klassen aus Unter-Packages von `packagename`.

Packages aus der Java-Klassenbibliothek

- Die Java-Klassenbibliothek bietet bereits eine Vielzahl von Klassen an, die alle in Packages gegliedert sind.
- Beispiele für vordefinierte Packages:

`java.io`	Ein- und Ausgabe
`java.util`	nützliche Sprach-Werkzeuge
`java.awt`	Abstract Window Toolkit
`java.lang`	Elementare Sprachunterstützung

  usw.
- Die Klassen im Package `java.lang` sind so elementar (z.B. enthält `java.lang` die Klasse `Object`, die implizite Vaterklasse aller Java-Klassen), dass sie von jeder Klasse automatisch importiert werden. Ein expliziter Import mit **`import java.lang.*;`** ist also *nicht* erforderlich.

Deklaration eigener Packages

- Ein eigenes Package `mypackage` wird angelegt, indem man vor eine Klassendeklaration und vor den **`import`**-Anweisungen die Anweisung **`package`** `mypackage;` platziert.
- Es können beliebig viele Klassen (jeweils aber mit unterschiedlichen Namen) mit der Anweisung **`package`** `mypackage;` im selben Package gruppiert werden.
- Um Namenskollisionen bei der Verwendung von Klassenbibliotheken unterschiedlicher Hersteller zu vermeiden, ist es Konvention, für Packages die URL-Domain der Hersteller in umgekehrter Reihenfolge zu verwenden, z.B.

  `com.sun. ...` für die Firma Sun,

  `de.lmu.ifi.dbs. ...` für den DBS-Lehrstuhl an der LMU.

Deklaration eigener Packages

- Wie bereits erwähnt, muss die Deklaration einer Klasse `X` in eine Datei `X.java` geschrieben werden.
- Darüberhinaus müssen alle Klassendeklarationen (also die entsprechenden `.java`-Dateien) eines Packages `p` in einem Verzeichnis `p` liegen.
- Beispiel:
  - Die Datei `Klasse1.java` mit der Deklaration der Klasse `package1.Klasse1` liegt im Verzeichnis `package1`.
  - Die Datei `Klasse2.java` mit der Deklaration der Klasse `package1.unterpackage1.Klasse2` liegt im Verzeichnis `package1/unterpackage1`.

Überblick

13. Strukturierung von Java-Programmen: Packages

13.1 Strukturierung durch Packages

13.2 Zugriffsspezifikationen

Zugriffsrechte und Sichtbarkeit

- Wir hatten bereits verschiedene Schlüsselwörter zur Spezifikation der Sichtbarkeit von Klassen und Klassen-Elementen (Attributen/Methoden) kennengelernt und teilweise erweitert.
- Erst jetzt mit dem Konzept der Packages können wir allerdings alle Möglichkeiten kennenlernen.
- Im folgenden also nocheinmal die (nun endgültige) Möglichkeit, die Sichtbarkeit von Klassen und Elementen einer Klasse zu spezifizieren.

Zugriffsrechte und Sichtbarkeit

- Klassen und Elemente mit der Sichtbarkeit **`public`** sind von allen anderen Klassen (insbesondere auch Klassen anderer Packages) sichtbar und zugreifbar.
- Klassen und Elemente mit der Sichtbarkeit **`private`** sind nur innerhalb der eigenen Klasse (also auch *nicht* innerhalb möglicher Unterklassen oder Klassen des selben Packages) sichtbar und zugreifbar.
- Klassen und Elemente mit der Sichtbarkeit **`protected`** sind innerhalb des gesamten Packages und aller Unterklassen (auch außerhalb des Packages) sichtbar und zugreifbar.
- Klassen und Elemente, deren Sichtbarkeit *nicht* durch ein entsprechendes Schlüsselwort spezifiziert ist, erhalten per Default die sogenannte *package scoped* (*friendly*) Sichtbarkeit: diese Elemente sind nur für Klassen innerhalb des selben Packages sichtbar und zugreifbar.

Zugriffsrechte und Sichtbarkeit

Spezifikation	sichtbar in			
	allen Klassen	allen Unterklassen unabh. vom Package	Klassen im selben Package	selber Klasse
`public`	√	√	√	√
`protected`		√	√	√
			√	√
`private`				√

Abschnitt 14: Schnittstellen: Interfaces

Überblick

Schnittstellen

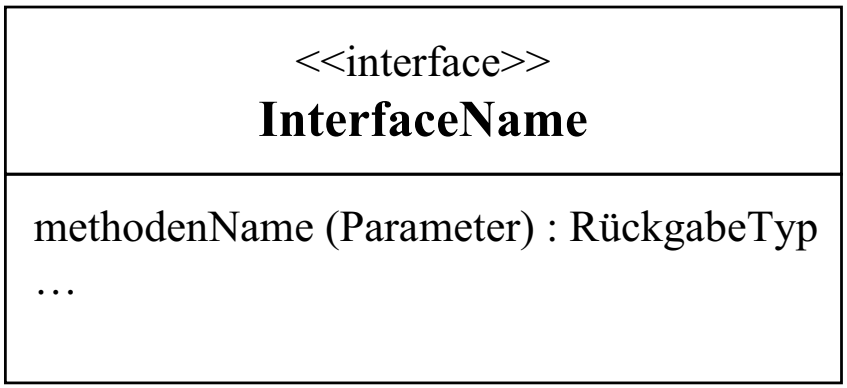
- Die Objekt-Methoden einer Klasse definieren die Verhaltensweisen von Objekten dieser Klasse.
- Aus Sicht eines Anwenders der Klasse können diese Verhaltensweisen auch als *Funktionalitäten* oder *Dienstleistungen* bezeichnet werden.
- Dabei interessiert sich der Anwender (z.B. eine andere Klasse) nicht für die Implementierungsdetails, er muss nur die Signatur einer Methode kennen, um sie verwenden zu können.
- Andererseits hängen die Implementierungsdetails nicht von der konkreten Verwendung durch einen Anwender ab.
- Der Implementierer möchte also nur wissen, welche Funktionalitäten bereitgestellt werden müssen.
- Der Anwender hingegen möchte nur wissen, welche Funktionalitäten bereitgestellt werden.
- Beide richten sich nach einer gemeinsamen “Schablone” (*Schnittstelle*, *Interface*), der Implementierer “von innen”, der Anwender “von außen”.

Schnittstellen

- Eine Schnittstelle definiert also Dienstleistungen, die für Anwender (z.B. aufrufende Klassen) zur Verfügung stehen müssen und die der Implementierer bereitstellen muss.
- Dabei werden in der Schnittstelle die Implementierungsdetails der Dienstleistungen (im Ggs. zu Klassen) *nicht* festgelegt.
- Es werden *funktionale Abstraktionen* in Form der Methodensignaturen bereitgestellt, die das WAS, aber nicht das WIE festlegen.
- Interfaces bestehen i.d.R. nur aus Methodensignaturen, d.h. sie besitzen insbesondere keine Methodenrümpfe und keine Attribute (in Java dürfen sie zusätzlich statische Konstanten spezifizieren).
- Interfaces sind daher ähnlich zu abstrakten Klassen, die ausschließlich abstrakte Methoden besitzen.
- Interfaces sind gültige Objekttypen für Variablen, es gibt aber keine Objekte (Instanzen) dieses Typs.

Interfaces in UML

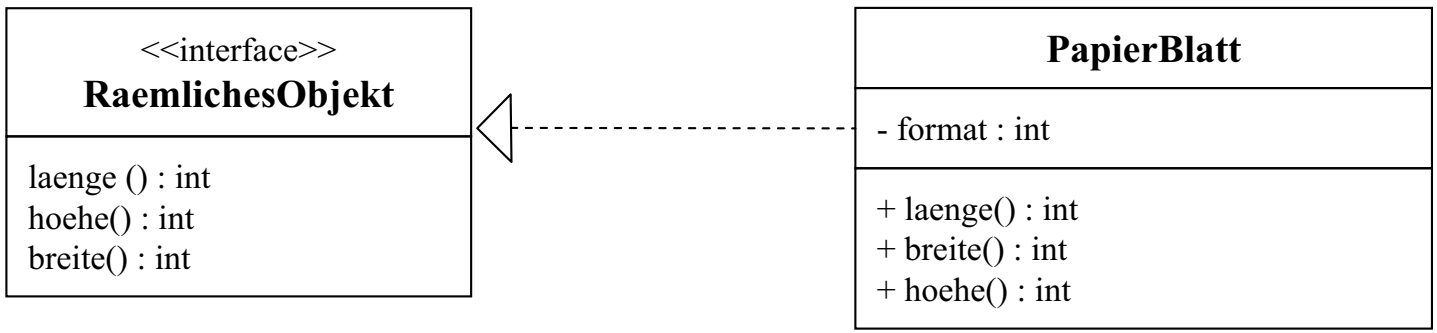
- Ein Interface spezifiziert also eine gewisse Funktionalität, ist selbst aber nicht instanziierbar.
- In UML werden Interfaces wie Klassen dargestellt, allerdings wird vor dem Interfacenamen der Stereotyp "`«interface»`" notiert:



- Welche oo Modellierungsideen werden vom Konzept der Interfaces realisiert?

Realisierungsbeziehung

- Implementiert eine (nicht-abstrakte) Klasse ein Interface, müssen alle Methoden des Interfaces in der Klasse implementiert werden.
- Eine Realisierungsbeziehung zwischen einem Interface und einer (implementierenden) Klasse wird in UML folgendermaßen umgesetzt:



Überblick

Definition von Interfaces

- In Java werden Schnittstellen mit dem Schlüsselwort **`interface`** anstelle von **`class`** vereinbart.
- Alle Eigenschaften für den Namensraum von Klassen bzgl. Packages gelten analog für Interfaces.
- Alle Methoden eines Interfaces sind grundsätzlich **`public`**, daher ist keine Sichtbarkeitsspezifikation nötig.
- Als Beispiel definieren wir ein Interface `RaeumlichesObjekt`, das den Zugriff auf die Ausdehnung eines (3-dimensionalen) räumlichen Objekts festlegt.

```
public interface RaeumlichesObjekt
{
    /** Die L&auml;nge des Objekts in mm. */
    int laenge();

    /** Die H&ouml;he des Objekts in mm. */
    int hoehe();

    /** Die Breite des Objekts in mm. */
    int breite();
}
```

Definition von Interfaces

- In einem zweiten Beispiel definieren wir ein Interface `Farbig`, das den Zugriff auf die Farbe eines Objekts festlegt und einige wichtige Farben als Konstanten definiert.
- Alle “Attribute” eines Interfaces sind grundsätzlich globale, statische Konstanten, daher werden die Zusätze **`public static final`** nicht benötigt.

```
import java.awt.*; // fuer die Klasse Color

public interface Farbig
{
    /** Die Farbe (als RGB-Zahl) des Objekts. */
    int farbe();

    /** Die Farbe "schwarz". */
    int SCHWARZ = Color.BLACK.getRGB();
    /** Die Farbe "weiss". */
    int WEISS = Color.WHITE.getRGB();
    /** Die Farbe "rot". */
    int ROT = Color.RED.getRGB();
    /** Die Farbe "gr&uuml;n". */
    int GRUEN = Color.GREEN.getRGB();
    /** Die Farbe "blau". */
    int BLAU = Color.BLUE.getRGB();
}
```

Implementierung von Interfaces

- Das Interface `RaeumlichesObjekt` beschreibt zunächst nur die gewünschte Funktionalität, stellt diese aber noch nicht zur Verfügung.
- Die Funktionalität kann nur von einer konkreten Klasse zur Verfügung gestellt werden.
- Dazu muss die entsprechende Klasse das Interface *implementieren* – dies wird mit dem Schlüsselwort **`implements`** `<InterfaceName>` angezeigt.
- Die Klasse muss dann Methodenrümpfe für alle Methoden des Interfaces definieren.
- Eine Klasse kann auch mehrere Interfaces implementieren (dabei muss sie dann *alle* Methoden *aller* Interfaces implementieren).
- Im folgenden sehen wir drei Beispielklassen, die alle das Interface `RaeumlichesObjekt` implementieren. Zwei Klassen implementieren zudem das Interface `Farbig`.

Implementierung von Interfaces

```
public class Auto2 implements RaeumlichesObjekt, Farbig
{
    private int laenge;
    private int hoehe;
    private int breite;
    private int farbe = WEISS;
    // weitere Attribute ...

    public int laenge()
    {
        return this.laenge;
    }

    public int hoehe()
    {
        return this.hoehe;
    }

    public int breite()
    {
        return this.breite;
    }

    public int farbe()
    {
        return this.farbe;
    }
}
```

Implementierung von Interfaces

```
public class FussballPlatz implements RaeumlichesObjekt, Farbig
{
    public int laenge()
    {
        return 105000;
    }

    public int hoehe()
    {
        return 0;
    }

    public int breite()
    {
        return 65000;
    }

    public int farbe()
    {
        return GRUEN;
    }
}
```

Implementierung von Interfaces

```
public class PapierBlatt implements RaeumlichesObjekt
{
    private final int FORMAT;

    public int laenge()
    {
        int erg = 0;
        switch(FORMAT)
        {
            case 0 : erg = 1189; break;
            case 1 : erg = 841; break;
            case 2 : erg = 594; break;
            case 3 : erg = 420; break;
            case 4 : erg = 297; break;
            // usw. ...
        }
        return erg;
    }

    ...
```

Implementierung von Interfaces

```
    ...

    public int hoehe()
    {
        return 0;
    }

    public int breite()
    {
        int erg = 0;
        switch(FORMAT)
        {
            case 0 : erg = 841; break;
            case 1 : erg = 594; break;
            case 2 : erg = 420; break;
            case 3 : erg = 297; break;
            case 4 : erg = 210; break;
            // usw. ...
        }
        return erg;
    }
}
```

Verwendung von Interfaces

- Nützlich sind Interfaces u.a. dann, wenn Eigenschaften einer Klasse beschrieben werden sollen, die nicht direkt in ihrer “normalen” Vererbungshierarchie abgebildet werden können.
- Hätten wir `RaeumlichesObjekt` als abstrakte Vaterklasse definiert, und `Auto2`, `FussballPlatz` und `PapierBlatt` daraus abgeleitet, ergäbe das eine etwas unnatürliche Vererbungshierarchie.
- Durch die Implementierung des Interfaces `RaeumlichesObjekt` können die drei Klassen die Methoden `laenge`, `hoehe` und `breite` dagegen *unabhängig* von ihrer Vererbungslinie garantieren.

Verwendung von Interfaces

- Definierte Interfaces können ähnlich wie abstrakte Klassen verwendet werden.
- Das folgende einfache Beispiel illustriert die Anwendung des Interfaces `RaeumlichesObjekt`:

  Eine Methode `volumen` (die z.B. eine Hilfsmethode einer `main`-Methode sein kann), die das Volumen von räumlichen Objekten berechnet, kann nun wie folgt definiert sein:

```
public static int volumen(RaeumlichesObjekt obj)
{
    return obj.laenge() * obj.hoehe * obj.breite();
}
```

Verwendung von Interfaces

- Die Methode `volumen` akzeptiert als Eingabe nun nur Objekte von Klassen, die das Interface `RaeumlichesObjekt` implementieren, also nur Objekte der Klassen `Auto2`, `FussballPlatz` und `PapierBlatt`.
- So ist sichergestellt, dass die Methoden `laenge`, `hoehe` und `breite` tatsächlich zur Verfügung stehen.
- Wie bereits erwähnt, kann man für Interfaces keine Objekte instanziieren, d.h. die Methode `volumen` kann nicht für ein Objekt vom Typ `RaeumlichesObjekt` aufgerufen werden, sondern nur für Objekte vom Typ einer Klasse, die `RaeumlichesObjekt` implementiert. Auf der anderen Seite ist ein Interface ein gültiger Objekttyp, d.h. man kann Variablen vom Typ eines Interfaces vereinbaren (wie z.B. die Parameter-Variable `obj` im obigen Beispiel).

Interfaces und Vererbung

- Eine Klasse, die ein Interface implementiert, kann auch Vaterklasse für ein oder mehrere abgeleitete Klassen sein.
- Dann erben alle abgeleiteten Klassen natürlich auch alle Methoden des Interfaces (die ja in der Vaterklasse implementiert wurden und ggf. nochmals überschrieben werden können).
- Dadurch “implementieren” auch alle abgeleiteten Klassen die Interfaces, die von der Vaterklasse implementiert werden.
- Auch Interfaces selbst können abgeleitet werden.
- Das abgeleitete Interface erbt alle Methoden des Vater-Interface.
- Eine implementierende Klasse muss damit auch alle Methoden aller Vater-Interfaces implementieren.

Beispiel: Vererbung von Interfaces

```
public interface EinDimensional
{
  int laenge();
}

public interface ZweiDimensional extends EinDimensional
{
  int breite();
}

public interface DreiDimensional extends ZweiDimensional
{
  int hoehe();
}
```

Beispiel: Vererbung von Interfaces

```
public class Auto3 implements DreiDimensional
{
  private int laenge;
  private int hoehe;
  private int breite;
  // weitere Attribute ...

  public int laenge()
  {
    return this.laenge;
  }

  public int hoehe()
  {
    return this.hoehe;
  }

  public int breite()
  {
    return this.breite;
  }
}
```

Interfaces und abstrakte Klassen: Unterschiede

- Offensichtlich haben Interfaces und abstrakte Klassen ähnliche Eigenschaften, z.B. können keine Objekte von Interfaces und abstrakten Klassen instanziiert werden.
- Im Unterschied zu Interfaces können abstrakte Klassen aber auch konkrete Methoden enthalten, d.h. Methoden mit Rumpf. Es ist sogar möglich, dass abstrakte Klassen nur konkrete Methoden spezifizieren. Mit dem Schlüsselwort **`abstract`** in der Klassendeklaration ist die Klasse dennoch abstrakt (und daher nicht instanziierbar).
- Alle Methoden eines Interfaces sind dagegen immer abstrakt.
- Abstrakte Klassen dürfen im Gegensatz zu Interfaces Attribute spezifizieren.

Interfaces und abstrakte Klassen: Verwendung

- Welches Konzept sollte man nun verwenden?
- Interfaces sind flexibler und können in unterschiedlichen Klassenhierarchien verwendet werden, da sie keinerlei Möglichkeiten bereitstellen, Implementierungsdetails festzulegen, sondern lediglich abstrakte Funktionalitäten.
- Implementierungsdetails können allerdings in der Dokumentation vorgeschrieben werden – ob diese Vorschrift eingehalten wird, kann aber der Compiler nicht überprüfen.
- Abstrakte Klassen bieten darüberhinaus die Möglichkeit, einen Teil der Implementierungsdetails bereits festzulegen und damit die Wiederverwendbarkeit von Code-Teilen zu unterstützen.

Interfaces und Mehrfachvererbung

- Wie bereits angedeutet, kann man mit Hilfe von Interfaces auch Mehrfachvererbung in Java modellieren.
- Beispiel: `AmphibienFahrzeug` wird von `WasserFahrzeug` und `Landfahrzeug` abgeleitet.
- Problem war, dass beide Vaterklassen eine Methode `getPS` implementieren, die nicht überschrieben wird.
- Falls die Methode `getPS` für ein Objekt der Klasse `AmphibienFahrzeug` aufgerufen wird, kann nicht entschieden werden, welche ererbte Version ausgeführt werden soll.

Interfaces und Mehrfachvererbung

- Lösung: Nur eine der Vaterklassen wird als Klasse realisiert, alle anderen (in unserem Fall nur eine) werden als Interfaces angegeben.
- Die (von der übriggebliebenen Vater-Klasse) abgeleitete Klasse muss nun alle Interfaces implementieren.
- Beispiel: `WasserFahrzeug` wird als Interface spezifiziert, wohingegen `Landfahrzeug` eine Klasse ist.
- Die Klasse `AmphibienFahrzeug` ist von `Landfahrzeug` abgeleitet und implementiert `WasserFahrzeug`.
- Die Methode `getPS` muss (falls sie im Interface `WasserFahrzeug` verlangt wird) in der abgeleiteten Klasse implementiert werden. Wird sie allerdings bereits von der Vaterklasse ererbt, muss `getPS` in `AmphibienFahrzeug` *nicht* implementiert werden (kann aber selbstverständlich überschrieben werden).
- In beiden Fällen ist die Methode `getPS` für Objekte der Klasse `AmphibienFahrzeug` eindeutig bestimmt.

Interfaces und Mehrfachvererbung

- Achtung: Die Realisierung von Mehrfachvererbung in Java mittels Interfaces schränkt das eigentliche Konzept der Mehrfachvererbung ein.
- Offensichtlich ist es z.B. nicht möglich, Objekte aller Vaterklassen zu erzeugen.
- In unserem Beispiel ist es nicht möglich, Objekte vom Typ `WasserFahrzeug` zu instanziieren.
- Diese Einschränkung ist allerdings nötig, um die oben angesprochenen Probleme der Mehrfachvererbung zu lösen.

Überblick

Was sind Marker-Interfaces?

- Interfaces, die weder Methoden noch Konstanten definieren, also einen leeren Rumpf haben, werden *Marker-Interfaces* (auch: *Flag-Interfaces*, *Tagging-Interfaces*) genannt.
- Marker-Interfaces sind dazu gedacht, gewisse (teilweise abstrakte) Eigenschaften von Objekten sicher zu stellen, die typischerweise im Kommentar des Interfaces spezifiziert sind.
- Implementiert eine Klasse ein Marker-Interface, sollte sich der Implementierer an diese Spezifikationen halten.
- Der Programmierer signalisiert durch **`implements`** `<MarkerInterface>`, dass seine Klasse die Eigenschaften hat, die im angegebenen Marker-Interface spezifiziert sind.
- Vorsicht bei geerbten Marker-Interfaces!

Beispiel: Das Interface `cloneable`

- Zur Erinnerung: `clone` erzeugt eine Kopie des aktuellen Objekts.
- Die ursprüngliche Fassung von `clone` erzeugt dabei eine sog. *flache Kopie* (*shallow copy*):
  - Es werden lediglich die Verweise auf die entsprechenden Attribute (d.h. die “Zettel” auf dem Keller) kopiert, nicht aber die dahinterstehenden Objekte.
  - Bei Attributen mit primitiven Typen macht das keinen Unterschied. Der Wert des neuen “Zettels” kann nicht von anderen Benutzern verändert werden, denn nur das neue Objekt hat Zugriff auf diesen “Zettel”.
  - Bei Attributen mit Objekttypen wird allerdings nur ein neuer “Zettel” angelegt, der auf das selbe Objekt auf der Halde verweist. Dieses Objekt wird auch von anderen “Zetteln” referenziert, kann also von anderen Benutzern verändert werden.
  - Damit ist nicht sichergestellt, dass die Kopie unabhängig vom ursprünglichen Objekt ist!

Beispiel: Das Interface `Cloneable`

- Die Java-API stellt das Marker-Interface `Cloneable` zur Verfügung, das für die Methode `clone` (die ja jede Klasse von der impliziten Vaterklasse `Object` erbt) eine spezielle Eigenschaft spezifiziert.
- Implementiert eine Klasse das Interface `Cloneable`, so garantiert der Implementierer, dass die Methode `clone` eine sog. *tiefe Kopie* (*deep copy*) erzeugt: für alle Attribute mit Objekttypen müssen Kopien der entsprechenden Objekte angelegt werden.
- Die Methode `clone` muss dazu entsprechend überschrieben werden.
- Achtung: Beim Erstellen einer tiefen Kopie muss man darauf achten, dass die Objekte eines Attributs mit Objekttyp selbst wieder Attribute mit Objekttypen haben können.

Beispiel: Das Interface `Cloneable`

Overview Package **Class** Use Tree Deprecated Index Help

Java™ Platform Standard Ed. 6

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

java.lang

Interface Cloneable

All Known Subinterfaces:

AclEntry, Attribute, AttributedCharacterIterator, Attributes, CertPathBuilderResult, CertPathParameters, CertPathValidatorResult, CertSelector, CertStoreParameters, CharacterIterator, CRLSelector, Descriptor, GSSCredential, Name

```
public interface Cloneable
```

A class implements the `Cloneable` interface to indicate to the `Object.clone()` method that it is legal for that method to make a field-for-field copy of instances of that class.

Invoking Object's clone method on an instance that does not implement the `Cloneable` interface results in the exception `CloneNotSupportedException` being thrown.

By convention, classes that implement this interface should override `Object.clone` (which is protected) with a public method. See `Object.clone()` for details on overriding this method.

Note that this interface does *not* contain the `clone` method. Therefore, it is not possible to clone an object merely by virtue of the fact that it implements this interface. Even if the clone method is invoked reflectively, there is no guarantee that it will succeed.

Since:

JDK1.0

See Also:

`CloneNotSupportedException`, `Object.clone()`

Beispiel: Das Interface `Cloneable`

```
public class DeepCopy implements Cloneable
{
    private int zahl;
    private int[] zahlen;

    public DeepCopy(int zahl, int[] zahlen)
    {
        this.zahl = zahl;
        this.zahlen = zahlen;
    }

    public Object clone()
    {
        int neueZahl = this.zahl;
        int[] neueZahlen = new int[this.zahlen.length];
        for(int i=0; i<this.zahlen.length; i++)
        {
            neueZahlen[i] = this.zahlen[i];
        }
        DeepCopy kopie = new DeepCopy(neueZahl, neueZahlen);
        return kopie;
    }
}
```

Was ist hier nicht optimal gelöst?

Beispiel: Das Interface `Cloneable`

```
public class DeepCopy2 implements Cloneable
{
    private int zahl;
    private int[] zahlen;

    public DeepCopy2(int zahl, int[] zahlen)
    {
        this.zahl = zahl;
        System.arraycopy(zahlen, 0, this.zahlen, 0, zahlen.length);
    }

    public Object clone()
    {
        DeepCopy2 kopie = new DeepCopy2(this.zahl, this.zahlen);
        return kopie;
    }
}
```

Überblick

Default-Implementierungen von Interfaces

- Wir hatten bisher Interfaces und abstrakte Klassen als ähnliche, aber doch unterschiedlich verwendbare Konzepte kennengelernt.
- Beide Konzepte kann man aber auch sinnvoll vereinen.
- Wird ein Interface `I` voraussichtlich sehr häufig implementiert werden und spezifiziert `I` sehr viele Methoden, dann führt das offensichtlich zu sehr hohem Aufwand.
- In diesem Fall kann es sinnvoll sein, eine Basisklasse zur Verfügung zu stellen, die das Interface `I` und alle sinnvoll realisierbaren Methoden implementiert. Diese Basisklasse wird *Default-Implementierung* von `I` genannt.
- Die Default-Implementierung muss nicht unbedingt *alle* Methoden des Interfaces implementieren. In diesem Fall ist diese Klasse abstrakt.

Default-Implementierungen von Interfaces

- Besitzt eine Klasse, die das Interface `I` implementieren muss, keine andere Vaterklasse, so kann sie von der Default-Implementierung abgeleitet werden.
- Damit erbt die Klasse bereits einen Teil der Methoden aus `I`, die implementiert werden müssen.

- Um dies zu illustrieren, ist auf den folgenden Folien ist ein beispielhaftes Interface `A`, eine Klasse `DefaultA` als Default-Implementierung von `A` und eine Klasse `B`, die `A` implementiert und von `DefaultA` abgeleitet ist, gegebenen.

Default-Implementierungen von Interfaces

```
public interface A
{
    int m1();

    int m2();

    void m3(double d);

    double m4(int i);
}
```

Default-Implementierungen von Interfaces

```
public class DefaultA implements A
{

    private int a1;
    private int a2;

    public int m1()
    {
        return a1 + a2;
    }

    public int m2()
    {
        return a2 - a1;
    }

    public void m3(double d)
    {

    }
    public double m4(int i)
    {
        return 2.0 * i;
    }
}
```

Default-Implementierungen von Interfaces

```
public class B extends DefaultA
{

    private int a3;
    private double a4;

    public void m3(double d)
    {
        a4 = d * d;
        a3 = super.m1() + super.m2();
    }

    public double m4(int i)
    {
        return a4 * (a3 + i);
    }
}
```

Delegation an die Default-Implementierung

- Wenn sich eine Klasse `C`, die `A` implementieren soll, nun nicht von `DefaultA` ableiten läßt (da `C` bereits eine andere explizite Vaterklasse besitzt), müßte sie alle Methoden aus `A` selbst implementieren.
- Da die Default-Implementierung `DefaultA` allerdings bereits nennenswerte Funktionalitäten implementiert, wäre es sehr fehlerträchtig (und auch schlechter Stil), diese ein zweites Mal in der Klasse `C` zu implementieren.
- Stattdessen ist es möglich, die Implementierung an die bereits vorhandene Klasse `DefaultA` zu *delegieren*.

Delegation an die Default-Implementierung

- Dazu muss die Klasse `C` ein Attribut vom Typ `DefaultA` anlegen, und alle Aufrufe der Interface-Methoden an dieses Objekt weiterleiten.
- Achtung: Die Delegation funktioniert nur, wenn die Default-Implementierung *keine* abstrakte Klasse ist!
- Die Klasse `C` ist auf der folgenden Folie beispielhaft dargestellt.

Default-Implementierungen von Interfaces

```
public class C extends D implements A
{
    private DefaultA defaultA = new DefaultA();

    public int m1()
    {
        return this.defaultA.m1();
    }

    public int m2()
    {
        return this.defaultA.m2();
    }

    public void m3(double d)
    {
        this.defaultA.m3(d);
    }

    public double m4(int i)
    {
        return this.defaultA.m4(i);
    }
}
```

Die leere Implementierung

- Manchmal wird ein Interface entworfen, bei dem nicht immer alle definierten Methoden benötigt werden.
- Da aber eine implementierende Klasse (wenn sie nicht abstrakt sein soll) immer alle Methoden implementieren muss, kann es sinnvoll sein, eine leere Default-Implementierung zur Verfügung zu stellen.
- Implementierende Klassen können dann ggf. von dieser abgeleitet werden und müssen dann nur noch die Methoden überschreiben, die tatsächlich benötigt werden.

Abschnitt 15: Robuste Programme durch Ausnahmebehandlung

Überblick

Robuste Programme durch Ausnahmebehandlung

- Eine *Ausnahme* ist ein Ereignis, das zu einem Laufzeitfehler führt.
- Ausnahmen können dabei unterschiedliche Ursachen haben:
  1. Programmierfehler, z.B. der Zugriff auf ein Array außerhalb der definierten Grenzen.
  2. Anwenderfehler, z.B. die Angabe eines falschen Dateinamens, was z.B. dazu führt, dass eine Datei, die geöffnet werden soll, nicht gefunden wird.
- In den älteren Programmiersprachen führen Ausnahmen meist zum Absturz des Programms, d.h. das Programm wird einfach beendet.
- Neuere Programmiersprachen wie Java erlauben die systematische Behandlung von Ausnahmen und ermöglichen dadurch robustere Programme.

Ausnahmen und deren Behandlung

- Eine Ausnahme kann also durch ein Programm zur Laufzeit verursacht werden. In Java spricht man von einer *Exception*.
- Beim *Auslösen* einer Exception spricht man in Java auch von *Werfen* oder *Throwing*.
- Das *Behandeln* einer Ausnahme, also eine explizite Reaktion auf das Eintreten einer Exception, wird im Java-Sprachgebrauch auch als *Catching* bezeichnet.

Das Grundprinzip von Exceptions in Java

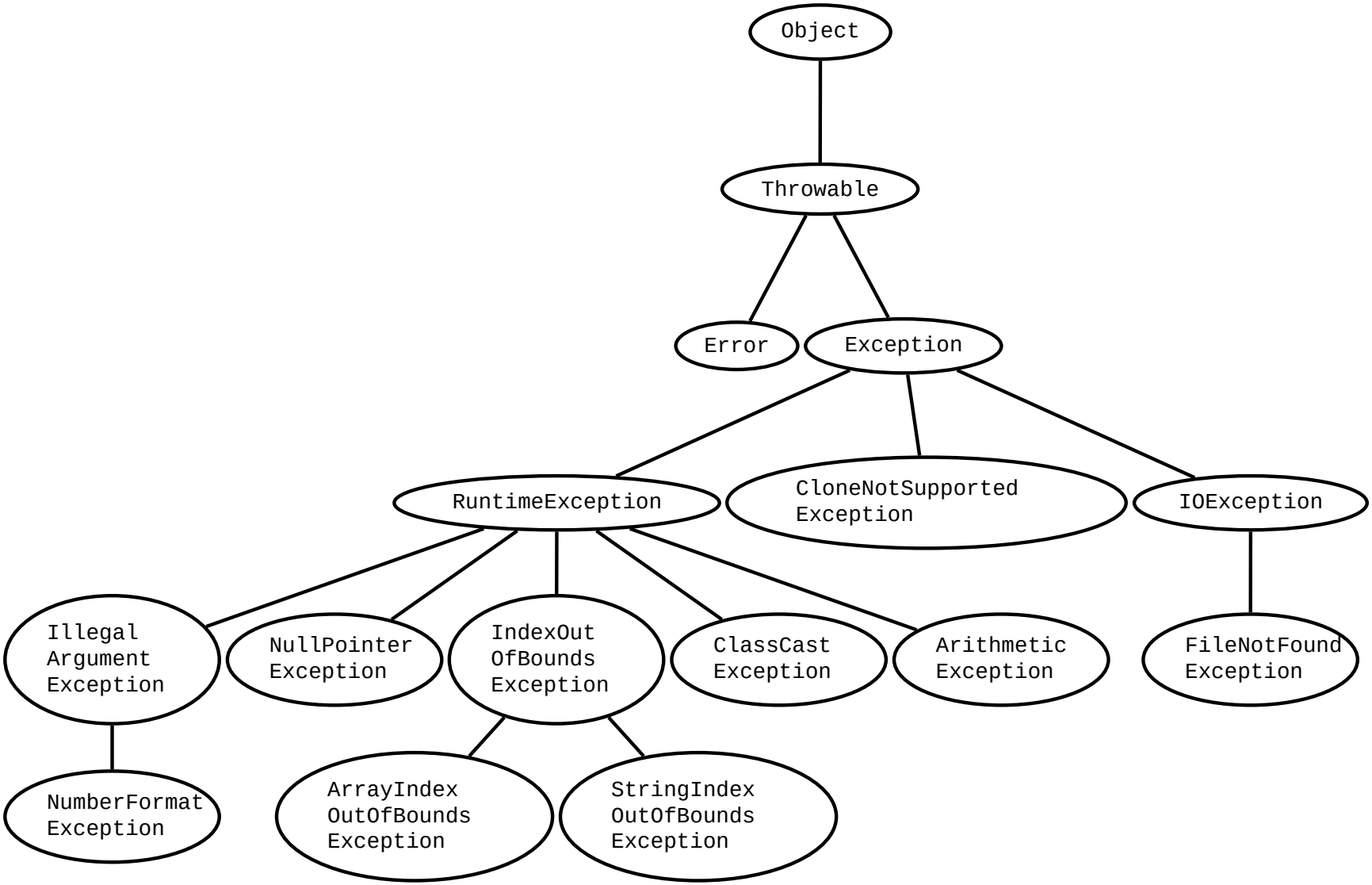
- Ein Laufzeitfehler oder eine vom Programmierer gewollte Bedingung löst eine Exception aus (Throwing).
- Diese kann entweder von dem Programmteil, in dem sie ausgelöst wurde, behandelt werden (Catching), oder sie kann weitergegeben werden (an andere Programmteile).
- Die Regel, dass alle Exceptions entweder behandelt oder weitergegeben werden müssen, bezeichnet man auch als *catch-or-throw-Regel*.
- Wird die Exception weitergegeben, so hat der Empfänger der Ausnahme erneut die Möglichkeit, sie entweder zu behandeln oder selbst weiterzugeben.
- Wird die Exception von keinem Programmteil behandelt, so führt sie zum Abbruch des Programms und zur Ausgabe einer Fehlermeldung.

Überblick

Ausnahmen in Java

- In Java sind Exceptions als normale Klassen realisiert, die instanziiert werden können. Insbesondere kann man auch eigene Exceptions definieren.
- Für die wichtigsten Exceptions gibt es bereits vordefinierte Klassen (im Package `java.lang`, d.h. sie sind ohne **`import`**-Anweisung implizit eingebunden und verwendbar).
- Oberklasse aller Exceptions ist die Klasse `Throwable`, die keine explizite Oberklasse besitzt und damit implizit von `Object` abgeleitet ist.
- Die Klasse `Throwable` stellt wichtige Konstruktoren und Methoden zum Exception-Handling bereit.

Wichtige Ausnahmen in ihrer Vererbungshierarchie



`Exception` VS. `RuntimeException`

- Java unterscheidet zwei Arten von Exceptions:
  1. “Allgemeine” Exceptions (Unterklassen von `Exception`, aber nicht von `RuntimeException`) modellieren Fehler, die im Routine-Betrieb des Programms entstehen können, z.B. auf Grund falsch eingegebener Daten. Diese Exceptions müssen *immer* behandelt oder weitergegeben werden.
  2. “Spezielle” Laufzeit-Ausnahmen (Unterklassen von `RuntimeException`) modellieren dagegen Fehler, die nicht im Routine-Betrieb des Programmes auftreten sollten, und daher auf Programmierfehler hindeuten, z.B. der Zugriff auf ein Array außerhalb der definierten Grenzen. Diese Exceptions müssen *nicht* behandelt oder explizit weitergegeben werden, bilden also eine Ausnahme zur catch-or-throw-Regel.
- Definiert man eigene Exception-Klassen, sollte man dies im Blick haben, auch wenn beide Arten von Exceptions nicht immer klar zu trennen sind.

Die Klasse `Error`

- Zusätzlich unterscheidet Java noch “schwere” Exceptions (oder: Fehler).
- Schwere Fehler modellieren hauptsächlich Probleme, die in der virtuellen Maschine ausgelöst wurden.
- Beispiel: `OutOfMemoryError` tritt auf, wenn die JVM ein neues Objekt allozieren soll, aber keinen Speicher mehr zur Verfügung hat.
- Fehler der Klasse `Error` oder deren Unterklassen müssen ebenfalls nicht behandelt oder weitergegeben werden.
- Es wird empfohlen, dies auch tatsächlich nicht zu tun.

Überblick

Die `try-catch`-Anweisung

- Das Behandeln von Exceptions erfolgt mit der **`try-catch`**-Anweisung:

```
try
{
  <Anweisung>
  ...
}
catch (<ExceptionTyp> e)
{
  <Anweisung>
  ...
}
```

Die `try-catch`-Anweisung

- Der **`try`**-Block enthält eine oder mehrere Anweisungen, bei deren Ausführung eine Exception vom Typ `ExceptionTyp` auftreten kann (dabei wird dann ein Objekt vom Typ `ExceptionTyp` erzeugt).
- Tritt bei einer der Anweisungen eine entsprechende Exception auf, wird die normale Programmausführung unterbrochen.
- Der Programmablauf fährt mit der ersten Anweisung im **`catch`**-Block fort.
- Im **`catch`**-Block kann Code untergebracht werden, der eine angemessene Reaktion auf die Exception realisiert.
- Nach einer **`try-catch`**-Anweisung wird mit der ersten Anweisung nach dem **`catch`**-Block normal fortgefahren. Dies geschieht entweder wenn das Ende des **`try`**-Blocks erreicht wurde (und dabei keine Exception aufgetreten ist) oder wenn das Ende des **`catch`**-Blocks nach einer Exception erreicht wurde.

Beispiel: Behandlung von Ausnahmen

- Im folgenden Beispiel soll der String `"40"` aus verschiedenen Zahlensystemen in ein **`int`** konvertiert und dann als Dezimalzahl ausgegeben werden.
- Dazu verwenden wir die statische Methode `Integer.parseInt(String, int)`, die einen String `s` und ein **`int`** `i` (Basis des Zahlensystems) als Parameter erwartet.

Beispiel: keine Ausnahme-Behandlung

```
public class KeineBehandlung
{
    public static void main(String[] args)
    {
        int i = 0;

        for(int base=10; base >=2; --base)
        {
            i = Integer.parseInt("40", base);
            System.out.println("40 zur Basis "+base+" = "+i);
        }
    }
}
```

Beispiel: keine Ausnahme-Behandlung

Bei Ausführung stürzt das Programm mit folgender Ausgabe ab:

```
40 zur Basis 10 = 40
40 zur Basis 9 = 36
40 zur Basis 8 = 32
40 zur Basis 7 = 28
40 zur Basis 6 = 24
40 zur Basis 5 = 20
Exception in thread "main" java.lang.NumberFormatException:
For input string: "40"
    at java.lang.NumberFormatException.forInputString
(NumberFormatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:447)
    at de.lmu.ifi.dbs.info2.ss07.skript.exceptions.KeineBehandlung.main
(KeineBehandlung.java:21)
```

Beispiel: Ausnahme-Behandlung mit `try-catch`

```
public class MitBehandlung
{
    public static void main(String[] args)
    {
        int i = 0;
        int base = 0;

        try
        {
            for(base=10; base >=2; --base)
            {
                i = Integer.parseInt("40", base);
                System.out.println("40 zur Basis "+base+" = "+i);
            }
        }
        catch (NumberFormatException e)
        {
            System.out.println("40 ist keine Zahl zur Basis "+base);
        }
    }
}
```

Beispiel: Ausnahme-Behandlung mit `try-catch`

Zwar ist 40 immer noch keine Zahl zur Basis 4, aber das Programm fängt diesen Fehler nun ab und gibt eine geeignete Fehlermeldung aus:

```
40 zur Basis 10 = 40
40 zur Basis 9 = 36
40 zur Basis 8 = 32
40 zur Basis 7 = 28
40 zur Basis 6 = 24
40 zur Basis 5 = 20
40 ist keine Zahl zur Basis 4
```

Der `catch`-Block

- Im Kopf der **`catch`**-Klausel wird die Art des abzufangenen Fehlers definiert (in unserem Beispiel `NumberFormatException`).
- Das Objekt `e` vom Typ `NumberFormatException`, das im **`try`**-Block erzeugt wurde, wird dem **`catch`**-Block übergeben.
- Da alle Exception-Klassen von der Klasse `Throwable` abgeleitet sind, ist `e` auch vom Typ `Throwable` und erbt alle Methoden aus dieser Vaterklasse.

Wichtige Methoden der Klasse `Throwable`

- Die Klasse `Throwable` definiert wichtige Methoden, die bei der Behandlung von Exceptions sehr hilfreich sind, z.B.
  - Die Methode `getMessage` liefert einen Fehlertext.
  - Die Methode `printStackTrace` druckt einen Auszug aus dem aktuellen Laufzeit-Keller.
- Wird eine Laufzeit-Exception nicht behandelt, ruft die JRE automatisch die Methode `printStackTrace` auf, bevor es das Programm beendet.

Fortfahren nach Fehlern

- Die Reaktion auf eine Exception muss nicht zwangsweise darin bestehen, das Programm zu beenden.
- Stattdessen kann versucht werden, den Fehler zu beheben oder zu umgehen um dann mit dem Programm fortzufahren.
- Im obigen Programm führt die Behandlung der Exception zum Programmende, da nach dem **`catch`**-Block keine Anweisungen mehr stehen.
- Soll nach jedem Fehler die Konvertierung mit der nächsten Basis fortgesetzt werden, muss die **`try-catch`**-Anweisung in die **`for`**-Schleife platziert werden.
- Ob es sinnvoller ist, nach einem Fehler abzubrechen oder fortzufahren, hängt von der Art des Fehlers ab.

Beispiel: Fortfahren nach Fehlern

```
public class NachFehlerFortsetzen
{
    public static void main(String[] args)
    {
        int i = 0;

        for(int base=10; base >=2; --base)
        {
            try
            {
                i = Integer.parseInt("40", base);
                System.out.println("40 zur Basis "+base+" = "+i);
            }
            catch (NumberFormatException e)
            {
                System.out.println("40 ist keine Zahl zur Basis "+base);
            }
        }
    }
}
```

Beispiel: Fortfahren nach Fehlern

Die Ausführung des Programms ergibt nun folgende Ausgabe:

```
40 zur Basis 10 = 40
40 zur Basis 9 = 36
40 zur Basis 8 = 32
40 zur Basis 7 = 28
40 zur Basis 6 = 24
40 zur Basis 5 = 20
40 ist keine Zahl zur Basis 4
40 ist keine Zahl zur Basis 3
40 ist keine Zahl zur Basis 2
```

Mehrere `catch`-Klauseln

- Innerhalb eines **`try`**-Blocks können natürlich auch mehrere Exceptions (unterschiedlichen Typs) auftreten.
- Daher ist es möglich, zu einem **`try`**-Block mehrere **`catch`**-Klauseln bzw. -Blöcke anzugeben.
- Jede **`catch`**-Klausel fängt den Fehler ab, die zum Typ des angegebenen Fehlerobjekts zuweisungskompatibel ist, d.h. alle Fehler der angegebenen Exception-Klasse und all ihrer Unterklassen.
- Die einzelnen **`catch`**-Klauseln werden in der Reihenfolge ihres Auftretens abgearbeitet.

Die `finally`-Klausel

- Die **`try-catch`**-Anweisung enthält einen optionalen Bestandteil, den wir noch nicht erläutert haben: die sogenannte **`finally`**-Klausel.
- Im Block der **`finally`**-Klausel kann Code plaziert werden, der immer dann ausgeführt wird, wenn die zugehörige **`try`**-Klausel betreten wurde.
- Dieser Code in der **`finally`**-Klausel wird grundsätzlich ausgeführt, *unabhängig* davon, welches Ereignis dazu führte, dass die **`try`**-Klausel verlassen wurde.
- Die **`finally`**-Klausel ist der ideale Ort für Aufräumarbeiten wie z.B. Dateien zu speichern/schließen, Ressourcen freizugeben, etc.

Beispiel: Die `finally`-Klausel

```
public class NachFehlerFortsetzenPlusFinally
{
    public static void main(String[] args)
    {
        int i = 0;

        for(int base=10; base >=2; --base)
        {
            try
            {
                i = Integer.parseInt("40", base);
                System.out.println("40 zur Basis "+base+" = "+i);
            }
            catch (NumberFormatException e)
            {
                System.out.println("40 ist keine Zahl zur Basis "+base);
            }
            finally
            {
                System.out.println("Ein bloedes Beispiel!");
            }
        }
    }
}
```

Beispiel: Fortfahren nach Fehlern

Die Ausführung des Programms ergibt nun folgende Ausgabe:

```
40 zur Basis 10 = 40
Ein bloedes Beispiel.
40 zur Basis 9 = 36
Ein bloedes Beispiel.
40 zur Basis 8 = 32
Ein bloedes Beispiel.
40 zur Basis 7 = 28
Ein bloedes Beispiel.
40 zur Basis 6 = 24
Ein bloedes Beispiel.
40 zur Basis 5 = 20
Ein bloedes Beispiel.
40 ist keine Zahl zur Basis 4
Ein bloedes Beispiel.
40 ist keine Zahl zur Basis 3
Ein bloedes Beispiel.
40 ist keine Zahl zur Basis 2
Ein bloedes Beispiel.
```

Überblick

Die `throws`-Klausel

- Anstelle einer **`try-catch`**-Anweisung können Exceptions auch weitergegeben werden.
- In diesem Fall muss die Methode, in der die Exception auftreten kann, gekennzeichnet werden.
- Dazu wird am Ende des Methodenkopfes das Schlüsselwort **`throws`** mit einer Liste aller Ausnahmen, die auftreten können, angehängt.
- Beispiel:

```
public static double kehrWert(int i) throws ArithmeticException
{
  return 1.0/i;
}
```

- Im Sinne der catch-or-throw-Regel müssen also alle Exceptions (Ausnahme: Unterklassen von `RuntimeException`) entweder behandelt oder weitergegeben werden.

Weitergabe von Ausnahmen

- Die **`throws`**-Klausel macht dem Compiler und auch allen Aufrufenden alle potentiellen Exceptions, die von der entsprechenden Methode verursacht werden können, bekannt.
- Dadurch kann sowohl der Compiler als auch der Aufrufende sicherstellen, dass bei jedem Aufruf dieser Methode wiederum die catch-or-throw-Regel eingehalten wird.

Behandlung und Weitergabe von Ausnahmen

Wie funktioniert die Behandlung/Weitergabe von Exceptions?

- Tritt eine Exception auf, wird zunächst nach einem umgebenden **`try-catch`**-Block gesucht, der den Fehler behandelt.
- Ist kein solcher **`try-catch`**-Block vorhanden, wird die Suche sukzessive in allen umgebenden Blöcken wiederholt.
- Ist dies auch erfolglos, wird der Fehler an den Aufrufer der Methode weitergegeben, wo wiederum blockweise weitergesucht wird.
- Enthalten alle aufrufenden Methoden inklusive der `main`-Methode keinen Code, um den Fehler zu behandeln, bricht das Programm mit einer Fehlermeldung ab.

Auslösen von Ausnahmen

- Ein wichtiger Aspekt bei der Weiterleitung von Exceptions ist, dass man auch Ausnahmen explizit auslösen kann.
- Exceptions werden mit der Anweisung
  `throw <ExceptionObject>;`
  ausgelöst.
- Objekte von Ausnahme-Typen können wie alle anderen Objekte erzeugt und verwendet werden, also z.B. auch Variablen (des entsprechenden Typs) zugewiesen werden.

Auslösen von Ausnahmen

- Die Behandlung einer ausgelösten Exception folgt den vorher skizzierten Regeln.
- Gemäß der catch-or-throw-Regel müssen diese Exceptions also entweder behandelt (mit einer **`try-catch`**-Anweisung) oder weitergeleitet (mit einer **`throws`**-Klausel) werden.

Auslösen von Ausnahmen: Beispiel

```
public static boolean istPrim(int n) throws IllegalArgumentException
{
  if(n<=0)
  {
    throw new IllegalArgumentException("Parameter > 0 erwartet. Gefunden: "+n);
  }
  if(n==1)
  {
    return false;
  }
  for(int i=2; i<=n/2; ++i)
  {
    if(n % i == 0)
    {
      return false;
    }
  }
  return true;
}
```

Abschnitt 16: Typisierte Klassen

Überblick

Polymorphie mit Typvariablen

- Wir haben das Konzept der *Polymorphie* im Zusammenhang mit Vererbung kennengelernt: Eine Methode, die Objekte der Klasse `A` als Parameter bekommen kann, kann auch Objekte jeder Unterklasse von `A` als Parameter bekommen.
- Manchmal kann es wünschenswert sein, eine Methode (oder Klasse) zwar allgemein zu definieren, beim Verwenden aber sicher zu sein, dass sie jeweils nur mit einem ganz bestimmten Typ verwendet wird.
- Lösung: Man führt eine *Typvariable* ein.

Polymorphie mit Typvariablen

- Variablen, die wir in Programmen bisher gesehen haben, sind Variablen, die verschiedene Werte annehmen können, aber nicht verschiedene Typen (außer vererbungs-polymorphe Typen).
- Eine *Typvariable* hat als Wert einen *Typ*.
- Der Typ einer *Typvariablen* ist immer der gleiche: der Typ aller Typen.
- Was ist dieser generelle Typ in Java?

Variablen vs. Typvariablen

	Typ	Wert
Variable	ein beliebiger, aber fester Typ `A`	ein beliebiger Wert des Typs (z.B. auch ein Objekt der Klasse) `A` (oder einer Unterklasse)
Typvariable	**`class`**	eine beliebige Klasse

Intuition von Typvariablen

- Zur Erinnerung: In der Definition von Eigenschaften zweistelliger Relationen wurde eine Typvariable verwendet:

$$R \subseteq M \times M$$

  Hier steht M für eine beliebige Menge. Wichtig ist aber, dass R Teilmenge des zweistelligen Kreuzprodukts *derselben* Menge ist.
- Eigenschaften von Funktionen wurden abstrakt für die Menge D als Definitionsbereich und die Menge B als Bildbereich definiert – D und B sind wiederum Typvariablen.

Intuition von Typvariablen

- Wiederum mit Typvariablen wurden die Signaturen verschiedener Funktionen angegeben, z.B.:
  - Projektion auf Folgen: $\pi : M^n \times I_n \rightarrow M$
  - Postfix: $postfix : M^* \times M \rightarrow M$
- Auch Sorten von Ausdrücken wurden mit Variablen bezeichnet, diese Variablen können wir auch als Typvariablen auffassen.

Intuition von Typvariablen

- Nicht nur Polymorphie, auch Überladung haben wir mit Typvariablen charakterisiert:
  - Vorzeichenoperator: $S \to S$ mit $S \in \{$**`byte`**,**`short`**,**`int`**,...$\}$
  - Arithmetische Operatoren: $S \times S \to S$ mit $S \in \{$**`byte`**,**`short`**,**`int`**,...$\}$
  - Vergleichsoperatoren: $S \times S \to$**`boolean`** mit $S \in \{$**`byte`**,**`short`**,**`int`**,...$\}$
  - Type-Cast-Operatoren hätten wir auch so angeben können: **`(int)`** : $A \to$ **`int`** mit $A \in \{$**`char`**,**`byte`**,**`short`**,...$\}$
- In all diesen Fällen beschreiben wir mit Hilfe der Typvariablen die abstrakte Syntax. Die Syntax legt bereits fest, dass mit jedem Vorkommen derselben Typvariablen S der selbe Typ bezeichnet wird.
- Die Semantik (Bedeutung) kommt zustande durch Belegung der Typvariable mit einem konkreten Wert (d.h., einem *bestimmten* Typ).

Überblick

Erinnerung: Überladen von Methoden

Wir haben gesehen, dass Methoden den gleichen Namen haben können, wenn sich die Parameter-Typen unterscheiden.

```
public static void methode(Object o)
{
  System.out.println("01");
}
public static void methode(Tier t)
{
  System.out.println("02");
}
public static void methode(Schaf s)
{
  System.out.println("03");
}
```

Diese drei Methoden sind unterschiedlich, obwohl sie den gleichen Namen haben. Sie sind *überladen*.

Inferenz der richtigen Methode bei überladenen Methoden

Anhand des (*deklarierten*) Typs eines Parameters kann der Compiler entscheiden, welche Methode aufgerufen wird:

```
Schaf s = new Schaf();
Object o = (Object) s;
Tier t = (Tier) s;
methode(o);
methode(t);
methode(s);
```

Ausgabe:

```
01
02
03
```

Abgrenzung: Polymorphie von Methoden

Die Polymorphie von *Objekt-Methoden* durch Überschreiben in Unterklassen ist dagegen ein anderes Phänomen. Hier wird der tatsächliche Typ des Objektes (zur Laufzeit) bestimmt und die entsprechende Methode verwendet:

```
public class Tier
{
  public String toString()
  {
    return "Tier";
  }
}
public class Schaf extends Tier
{
  public String toString()
  {
    return "Schaf";
  }
}
Schaf s = new Schaf();
Object o = (Object) s;
Tier t = (Tier) s;
System.out.println(o.toString()); // Ausgabe: Schaf
System.out.println(t.toString()); // Ausgabe: Schaf
System.out.println(s.toString()); // Ausgabe: Schaf
```

Überladene Methoden: Auswahl der spezifischsten Methode

Bei überladenen Methoden entscheidet der Compiler für eine gegebene (deklarierte) aktuelle Parametrisierung, welche Methode für diese Parametrisierung die spezifischste ist. Die spezifischste Methode wird ausgeführt.

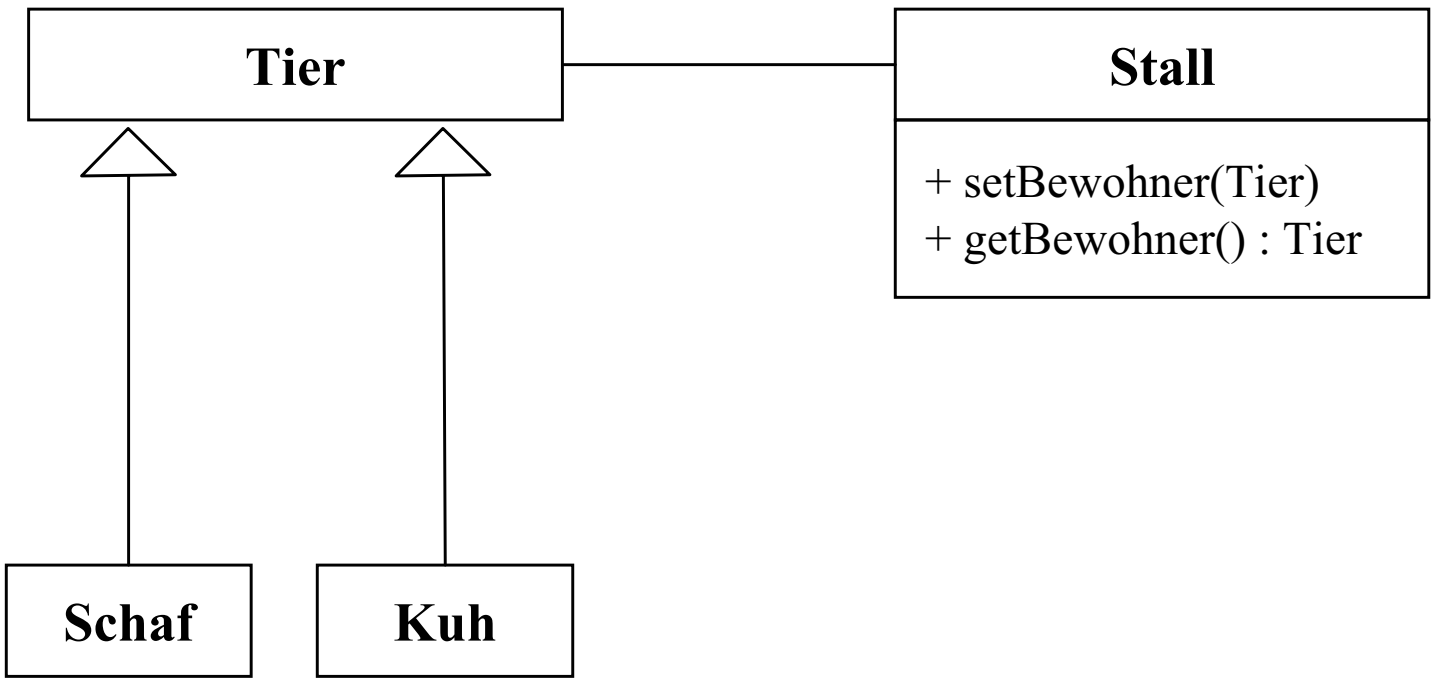
```
public static void methode(Object o)
{
  System.out.println("01");
}
public static void methode(Tier t)
{
  System.out.println("02");
}
public static void methode(Schaf s)
{
  System.out.println("03");
}
Schaf s = new Schaf();
methode(s); // Ausgabe: 03
```

Obwohl der Parameter `s` nicht nur vom Typ `Schaf`, sondern (aufgrund der Vererbung) auch vom Typ `Tier` und `Object`, wird die dritte Methode verwendet, die spezifischste für den aktuellen Parameter.

Überblick

Typ-Polymorphie durch Vererbung

- Wie wir gesehen haben, ermöglicht Vererbung Polymorphie.
- Gegeben folgendes Beispiel:



Typ-Polymorphie durch Vererbung

- Eine Klasse `Stall` wurde in diesem Beispiel für Objekte vom Typ `Tier` entworfen.

```
public class Tier
{
}
public class Stall
{
  private Tier bewohner;

  public void setBewohner(Tier tier)
  {
    this.bewohner = tier;
  }

  public Tier getBewohner()
  {
    return this.bewohner;
  }
}
```

- Welche Probleme können bei dieser Modellierung auftreten?

Typ-Polymorphie durch Vererbung

Der Stall kann jedes beliebige Objekt, das vom Typ `Tier` oder vom Typ einer Unterklasse von `Tier` ist, aufnehmen:

```
public class Schaf extends Tier
{
}
public static void main(String[] args)
{
  Stall schafstall = new Stall();
  schafstall.setBewohner(new Schaf());
}
```

Grenzen der Typ-Polymorphie durch Vererbung

Problem

Wenn man das Tier wieder aus dem Stall führt, “weiß” der Stall nicht mehr, was für ein Tier es ist:

```
public static void main(String[] args)
{
  Stall schafstall = new Stall();
  schafstall.setBewohner(new Schaf());

  Tier bewohner = schafstall.getBewohner();
}
```

Grenzen der Typ-Polymorphie durch Vererbung

Lösung ohne generische Typen

Der Programmierer merkt es sich und führt einen expliziten Typecast durch:

```
public static void main(String[] args)
{
  Stall schafstall = new Stall();
  schafstall.setBewohner(new Schaf());

  Tier bewohner = schafstall.getBewohner();
  Schaf schaf = (Schaf) schafstall.getBewohner();
}
```

Grenzen der Typ-Polymorphie durch Vererbung

- Die oben skizzierte “klassische” Lösung ist bestenfalls lästig für den Programmierer (da häufig explizite Typecasts nötig sind).
- In größeren Anwendungen (mit vielen Programmierern, die am gleichen Projekt arbeiten) ist diese Lösung auch überaus fehleranfällig und daher gefährlich (und bestenfalls teuer), da die Typüberprüfung erst zur *Laufzeit* stattfindet:

```
public class Kuh extends Tier
{
}
...
  // Code Fragment
  Stall stall = new Stall();
  stall.setBewohner(new Schaf());

  // Code an entfernter Stelle (anderer Programmierer)
  stall.setBewohner(new Kuh());

  // wiederum andere Stelle (gueltiger Code!)
  Schaf schaf = (Schaf) stall.getBewohner(); // RuntimeException:
                                             // ClassCastException
```

Überblick

Typ-Polymorphie durch typisierte Klassen

- In diesem Kapitel lernen wir eine neuere Lösung kennen, die mit Version 5.0 in Java eingeführt wurde: Polymorphie wird nicht nur durch Vererbung, sondern auch durch generische Klassen ermöglicht (*generics*). Das ist nicht nur bequemer, sondern ermöglicht die Typüberprüfung normalerweise bereits zur *Übersetzungszeit*.
- Eine *generische* Klasse ist *allgemein* für variable Typen implementiert – die Typen können bei der Verwendung der Klasse festgelegt werden, ohne dass die Implementierung verändert werden muss.
- Beide Konzepte der Polymorphie können auch gleichzeitig verwendet werden.

Generische Klassen

Beispiel: Die Klasse `Stall` wird durch Einführen einer Typvariablen generisch gehalten:

```
public class Stall<T> // Typvariable: T
{
  private T bewohner;

  public void setBewohner(T tier)
  {
    this.bewohner = tier;
  }

  public T getBewohner()
  {
    return this.bewohner;
  }
}
```

Typisierte Klassen

Im Gebrauch wird die Klasse durch Belegen der Typvariable mit einem bestimmten Typ typisiert (man spricht dann von einem parametrisierten Typ):

```
// Code Fragment
Stall<Schaf> stall = new Stall<Schaf>();
stall.setBewohner(new Schaf());

// dieses Fragment ist nicht mehr moeglich
// Kompilierfehler:
// The method setBewohner(Schaf) in the type Stall<Schaf>
// is not applicable for the arguments (Kuh)
stall.setBewohner(new Kuh());

// wiederum andere Stelle (gueltiger Code!)
Schaf schaf = stall.getBewohner();
// keine RuntimeException moeglich
```

Erinnerung: Parameter bei Methoden

```
1   public class Streckenberechnung {
2     public static double strecke(double m, double t, double k)
3     {
4       double b = k / m;
5       return 0.5 * b * (t * t);
6     }
7
8     public static void main(String[] args)
9     {
10      System.out.println(strecke(2.3, 42, 17.5));
11    }
12  }
```

Erinnerung: Parameter bei Methoden

- Im Programmbeispiel kommen Parameter der Methode `strecke` in zwei Varianten vor:
  1. als *formale* Parameter `m`, `t`, `k` (Zeile 2)
  2. als *aktuelle* Parameter `2.3, 42, 17.5` (Zeile 10)
- Die Berechnungsvorschrift wird abstrakt mit den formalen Parametern definiert, die im Methodenrumpf verwendet werden.
- Die konkrete Berechnung wird aber mit Werten durchgeführt, die den Parametern zugewiesen werden, eben den aktuellen Parametern.

Generische Klassen

Entsprechend ist der Typparameter bei Definition der Klasse ein *formaler* Typparameter.

```
public class Stall<T> // Typvariable: T (formaler Typ-Parameter)
{
  private T bewohner;

  public void setBewohner(T tier)
  {
    this.bewohner = tier;
  }

  public T getBewohner()
  {
    return this.bewohner;
  }
}
```

Typisierte Klassen

Im Gebrauch der Klasse und Belegen der Typvariable mit einem bestimmten Typ kennt der Compiler dann den *aktuellen* Typparameter und kann dadurch eine Typprüfung durchführen:

```
// Code Fragment
Stall<Schaf> stall = new Stall<Schaf>(); // aktueller Typ-Parameter: Schaf
                                         // ermoeglicht Typueberpruefung zur
                                         // Uebersetzungszeit
stall.setBewohner(new Schaf());

// dieses Fragment ist nicht mehr moeglich
// Kompilierfehler:
// The method setBewohner(Schaf) in the type Stall<Schaf>
// is not applicable for the arguments (Kuh)
stall.setBewohner(new Kuh());

// wiederum andere Stelle (gueltiger Code!)
Schaf schaf = stall.getBewohner();
// keine RuntimeException moeglich
```

Syntax für generische Typen

- Deklaration einer generischen Klasse:
  `class <Klassenname><<Typvariable(n)>>`
- Instantiierung eines generischen Typs (=Typisierung der Klasse, Parametrisierung des Typs):
  `<Klassenname><<Typausdruck/Typausdruecke>>`
- Beispiele für Instantiierungen von `Stall<T>`:
  - `Stall<Tier>`
  - `Stall<Schaf>`
  - `Stall<Kuh>`
  - `Stall<Schaf,Kuh>` *`//ungueltig: zu viele Parameter`*
  - `Stall<String>` *`//gueltig, aber unerwuenscht`*
  - `Stall<int>` *`//ungueltig (Warum???)`*

Übersetzung von generischen Typen

Zur Übersetzung von generischen Typen gibt es grundsätzlich zwei Möglichkeiten:

- **Heterogene Übersetzung**: Für jede Instantiierung (`Stall<Tier>`, `Stall<Schaf>`, `Stall<Kuh>` etc.) wird individueller Byte-Code erzeugt, also drei unterschiedliche (heterogene) Klassen.
- **Homogene Übersetzung**: Für jede parametrisierte Klasse (`Stall<T>`) wird genau eine Klasse erzeugt, die die generische Typinformation löscht ("type erasure") und die Typ-Parameter durch die Klasse `Object` ersetzt. Für jeden konkreten Typ werden zusätzlich Typanpassungen in die Anweisungen eingebaut.

Übersetzung von generischen Typen

- Java nutzt die homogene Übersetzung (C++ die heterogene).
- Der Byte-Code für eine generische Klasse entspricht also in etwa dem Byte-Code einer Klasse, die nur Typ-Polymorphie durch Vererbung benutzt.
- Gewonnen hat man aber die Typ-Sicherheit zur Übersetzungszeit.
- Vorteil/Nachteil der homogenen Übersetzung:
  - \+ weniger erzeugter Byte-Code, Rückwärtskompatibilität.
  - – geringere Ausdrucksmächtigkeit der Typüberprüfung zur Übersetzungszeit.

Verwendung generischer Typen ohne Typ-Parameter

- Generische Klassen können auch ohne Typ-Parameter verwendet werden.
- Ein generischer Typ ohne Typangabe heißt *Raw-Type*.
- Raw-Types bieten die gleiche Funktionalität wie parametrisierte Typen, allerdings werden die Parametertypen nicht zur Übersetzungszeit überprüft.
- Beispiel: Raw-Type von `Stall<T>` ist `Stall`.

```
Stall<Schaf> schafstall = new Stall<Schaf>();
Stall stall = new Stall();
stall = schafstall;

stall.setBewohner(new Kuh()); // Warnung:
    // Type safety: The method setBewohner(Tier)
    // belongs to the raw type Stall.
    // References to generic type Stall<T>
    // should be parameterized
// Die Warnung ist gerechtfertigt, denn:
Schaf poldi = schafstall.getBewohner();
// ist gueltiger Code, der aber zu einer
// RuntimeException (ClassCastException) fuehrt
```

Überblick

Unterklassen von generischen Klassen

Von generischen Klassen lassen sich in gewohnter Weise Unterklassen ableiten:

```
public class SchafStall extends Stall<Schaf>
{
}
```

Dabei kann der Typ der Oberklasse weiter festgelegt werden (hier wird die Oberklasse typisiert – andere Möglichkeiten sehen wir später).

Unterklassen von generischen Klassen

- Die Unterklasse einer generischen Klasse kann auch selbst wieder generisch sein.

```
public class GrossviehStall<T> extends Stall<T>
{
}
```

- In diesem Beispiel wird sogar ein weiterer Typ eingeführt:

```
public class DoppelStall<T, S> extends Stall<T>
{
  private S zweiterBewohner;
  ...
}
```

- Eine generische Klasse kann auch von vornherein mit mehreren Typ-Parametern definiert werden:

```
public class TripelStall<S, T, U> {
  private S ersterBewohner;
  private T zweiterBewohner;
  private U dritterBewohner;
  ...
}
```

Überblick

Platzhalter (Wildcards)

- Zur Typisierung kann man auch Wildcards verwenden:
  `public class IrgendeinStall<?>`
- Das ist so zunächst nur sinnvoll, wenn der genaue Typ keine Rolle spielt.
- Sinnvolle Verwendung finden Platzhalter zusammen mit oberen und/oder unteren Schranken.

Obere Typ-Schranke

- Wie kann die unerwünschte Typisierung `Stall<String>` verhindert werden?
- Die Typvariable kann innerhalb einer Typ-Hierarchie verortet werden, indem eine obere Schranke angegeben wird:

```
public class Stall<T extends Tier>
{
  ...
}
```

```
// Code Fragment
Stall<String> // ungueltig:
              // String ist nicht Unterklasse von Tier
```

Untere Typ-Schranke

Analog zur oberen Schranke kann man auch eine untere Schranke definieren. Das ist jedoch nur für Wildcards möglich, und nicht in Klassen-Definitionen, sondern nur in Deklarationen von Variablen:

```
Stall<? super Schaf> stall;
...
// Code Fragment
stall.setBewohner(new Kuh()) // ungueltig:
  // The method setBewohner(capture-of ? super Schaf)
  // in the type Stall<capture-of ? super Schaf> is
  // not applicable for the arguments (Kuh)
```

Untere Typ-Schranke ermöglicht breitere Nutzbarkeit

Die untere Schranke wird in der Praxis wesentlich seltener verwendet als die obere Schranke. Wir wollen aber den Sinn der unteren Schranke an einem einfachen Beispiel klar machen:
Bauer Moosgruber möchte einen Wettbewerb “Schönster Schafstall” veranstalten. Dazu bastelt er eine Klasse, die den Wettbewerb anhand eines übergebenen Vergleichsalgorithmus durchführt:

```
public interface Vergleicher<T>
{
  public int vergleiche(T t1, T t2);
}
public class SchoensterSchafstall
{
  public SchoensterSchafstall(Vergleicher<Stall<Schaf>> vergleicher)
  {
    // ...
  }
}
```

Unnötige Spezialisierung

Die unnötige Spezialisierung in diesem Beispiel wird deutlich, wenn man bedenkt, dass Moosgruber nun noch den Vergleichsalgorithmus austüfteln muss.
Bauer Huber hatte nämlich im letzten Jahr einen Wettbewerb “Schönster Stall” durchgeführt und dafür einen Vergleichsalgorithmus entworfen:

```
public class StallVergleicher implements Vergleicher<Stall<Tier>>
{
  public int vergleiche(Stall<Tier> t1, Stall<Tier> t2)
  {
    ...
  }
}
```

Bauer Moosgruber würde nun für seinen Wettbewerb gerne Hubers Klasse `StallVergleicher` verwenden. Hubers Vergleicher passt aber nicht zur Parametrisierung von Moosgrubers Konstruktor:

```
public SchoensterSchafstall(Vergleicher<Stall<Schaf>> vergleicher)
```

Wie kann Moosgruber seinen Wettbewerb so definieren, dass er Hubers `StallVergleicher` verwenden kann?

Aufhebung der unnötigen Spezialisierung

Moosgruber muß die unnötige Restriktion von seinem Konstruktor aufweichen:

```
public class SchoensterSchafstall
{
  public SchoensterSchafstall(Vergleicher<Stall<? super Schaf>> vergleicher)
  {
    ...
  }
}
```

Ein Vergleicher, der Ställe aller möglichen Oberklassen von `Schaf` vergleichen kann, kann natürlich auch Schaf-Ställe vergleichen – aber nicht umgekehrt.

Wann ist eine obere, wann eine untere Schranke sinnvoll?

- Wenn der Typ-Parameter `T` nur als Argument verwendet wird, ist oft eine untere Schranke sinnvoll (`? super T`).
- Wenn der Typ-Parameter `T` nur bei Rückgabewerten eine Rolle spielt, kann man dem Benutzer oft mehr Freiheit geben, indem man eine obere Schranke benützt (`? extends T`).

Unterklassen von generischen Klassen mit Schranken

- In der Unterklasse einer generischen Klasse können neue Schranken eingeführt werden.
- Mit dem Token & können dabei mehrere Schranken verbunden werden.

```
public interface Grossvieh
{
}
public class GrossviehStall<T extends Tier & Grossvieh> extends Stall<T>
{
}
```

- Ab der zweiten oberen Schranke kann es sich dabei natürlich nur noch um Interfaces handeln. Warum?

Überblick

Generisch typisierte Methoden

- Betrachten wir nun eine statische Methode, um ein Tier in den Stall zu treiben: Der Stall ist dabei zunächst scheinbar ein Stall für ein beliebiges Objekt.

```
public class Stall<T extends Tier>
{
}
  // statische Methode
  public static void treibeInDenStall(Tier tier, Stall<?> stall)
  {
    stall.setBewohner(tier);
  }
```

- Geht das?

Generisch typisierte Methoden

- Der Wildcard bedeutet, wir haben einen Stall von unbekanntem Typ (“Stall of unknown”).
- `Tier` ist aber ein ganz bestimmter Typ.
- Der Compiler kann nicht entscheiden, ob `Tier` (oder irgendein anderer Typ) der gleiche Typ ist wie der mit dem Wildcard bezeichnete.
- Deshalb gibt es einen Fehler zur Übersetzungszeit:

```
// The method setBewohner(capture-of ?)
// in the type Stall<capture-of ?>
// is not applicable for the arguments (Tier)
```

- Wildcards sind also nur in Verbindung mit Schranken sinnvoll oder wenn der Typ tatsächlich irrelevant (oder unbekannt) ist.

Generisch typisierte Methoden

- Die obige Methode läßt sich dennoch generisch typisieren:

```
public static <T extends Tier> void treibeInDenStall(T tier,
                                                     Stall<T> stall)
{
  stall.setBewohner(tier);
}
```

- Man definiert einen Typ-Parameter spezifisch für diese Methode. Die Definition wird vor dem Rückgabetyp der Methode eingeführt.
- Achtung: Auch falls diese Methode in der Klasse `class Stall<T extends Tier>` definiert wird, ist der Typ-Parameter `T` in der Methode nicht identisch mit dem Typ-Parameter der Klasse, da letzterer nicht-statisch, der Typ-Parameter der (statischen) Methode aber eine statische Eigenschaft ist. Als solche ist sie unabhängig von jeder möglichen Instantiierung der Klasse!

Methode mit Platzhalter

- Wenn ein Parameter ein generischer Typ ist, der genaue Typ aber irrelevant ist, kann man anstelle einer generischen Methode auch einfach eine Methode mit Platzhalter definieren:

```
public static boolean stallBesetzt(Stall<?> stall)
{
  return stall.getBewohner() != null;
}
```

- Dies ist dann möglich, wenn:
  - der Typ-Parameter T nur einmal benutzt wird,
  - der Typ des Rückgabewertes nicht von T abhängt,
  - der Typ keines anderen Argumentes der Methode von T abhängt,
  - das entsprechende Argument für Typ-Polymorphie verwendet wird (d.h., es sind verschiedene Argumenttypen erlaubt).
- Generische Methoden hingegen werden verwendet, um Abhängigkeiten zwischen mehreren Argumenttypen oder Argumenttypen und Rückgabetypen einer Methode herzustellen.

Überblick

Zuweisung von Werten spezielleren Typs an Werte allgemeineren Typs

- Wie wir aus dem Kapitel Vererbung wissen, ist folgende Zuweisung kein Problem:

```
Object objekt = new Object();
Integer integer = new Integer(1);
objekt = integer;
```

- Wie sieht es in folgendem Fall aus?

```
Stall<Schaf> schafstall = new Stall<Schaf>();
Stall<Tier> stall = new Stall<Tier>();
stall = schafstall;
```

Es besteht keine hierarchische Beziehung

- Dieser Versuch ist ungültiger Code. Er würde wieder zu folgendem altbekannten Problem führen:

```
Stall<Schaf> schafstall = new Stall<Schaf>();
Stall<Tier> stall = new Stall<Tier>();
stall = schafstall; // ungueltig (Typfehler)
stall.setBewohner(new Tier());
Schaf poldi = schafstall.getBewohner(); // ClassCastException
```

- Der Grund dafür ist, dass zwar `Schaf` eine Unterklasse von `Tier`, aber `Stall<Schaf>` *nicht* Unterklasse von `Stall<Tier>` ist.

Echte Unterklasse

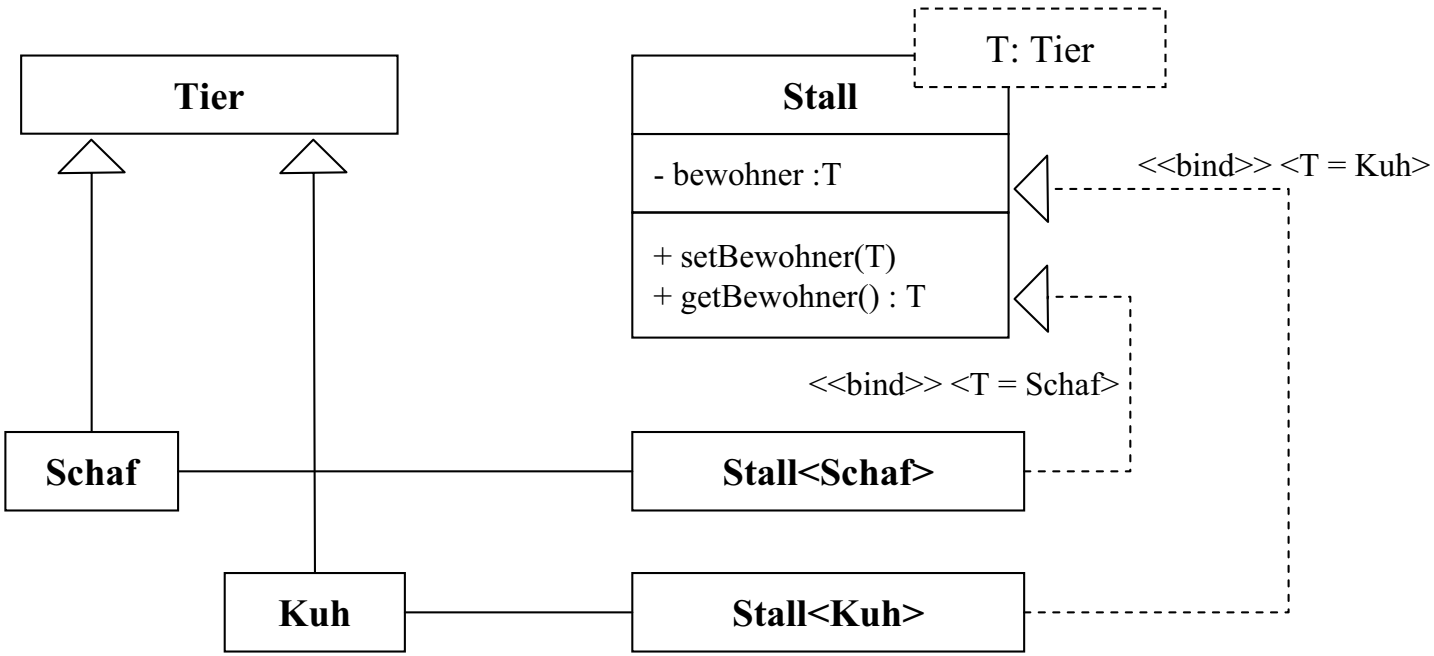
Mit der echten Unterklasse `SchafStall` **`extends`** `Stall<Schaf>` haben wir das Problem nicht. Hier kann eben auch nicht *irgendein* `Tier` angesiedelt werden, sondern nur ein `Schaf`.

```
SchafStall schafstall = new SchafStall();
Stall<Schaf> stall = new Stall<Schaf>();
stall = schafstall; // ok
stall.setBewohner(new Schaf());
// nicht moeglich:
// stall.setBewohner(new Tier());
Schaf poldi = schafstall.getBewohner();
```

Überblick

Templates

- In UML heißen typisierbare Klassen auch Templates.
- Unser Beispiel wird mit Templates in UML so dargestellt:



Abschnitt 17: Beispiel: Die Klasse String (Teil 2), das Interface Comparable<T>

Überblick

Zur Erinnerung: Eigenschaften von Relationen

Eine Relation $R \subseteq A \times A$ ist

- *reflexiv*, wenn für alle $a \in A$ gilt: $R(a, a)$
- *symmetrisch*, wenn für alle $a, b \in A$ gilt: $R(a, b) \Rightarrow R(b, a)$
- *transitiv*, wenn für alle $a, b, c \in A$ gilt: $R(a, b) \wedge R(b, c) \Rightarrow R(a, c)$.
- *antisymmetrisch*, wenn für alle $a, b \in A$ gilt: $R(a, b) \wedge R(b, a) \Rightarrow a = b$.
- *total*, wenn für alle $a, b \in A$ gilt: $R(a, b) \vee R(b, a)$.

Exakter Vergleich von Objekten

- Wie wir gesehen haben, ist in Java die Gleichheit (==) von Objekten als Überprüfung der Identität der Speicheradressen definiert.
- Die Klasse `Object` definiert aber auch eine Methode `public boolean equals(Object obj)`.
- Für Objekte vom Typ `Object` gibt `x.equals(y)` den gleichen Wert zurück wie der Vergleich `x==y`.
- Da jede Klasse eine Unterklasse von `Object` ist, verfügt auch jedes Objekt in Java über die `equals`-Methode.
- Oft wird diese Methode überschrieben, um eine allgemeinere Gleichheit zu testen.
- Wie bereits diskutiert, unterscheidet man zwischen `==` und `equals` auch als Identität vs. Gleichheit von Objekten.

Gleichheit von Objekten

Die Methode `equals` ist in der Klasse `Object` wie folgt spezifiziert:
Die `equals`-Methode implementiert eine Äquivalenz-Relation auf nicht-**`null`** Objektreferenzen mit folgenden Eigenschaften:

- *Reflexivität:* Für jede nicht-**`null`** Referenz `x` gilt: `x.equals(x)`.
- *Symmetrie:* Für nicht-**`null`** Referenzen `x` und `y` ist `x.equals(y)` **`true`** g.d.w. `y.equals(x)` **`true`** ist.
- *Transitivität:* Für nicht-**`null`** Referenzen `x`, `y` und `z` gilt: wenn `x.equals(y)` und `y.equals(z)` **`true`** sind, dann ist auch `x.equals(z)` **`true`**.
- *Konsistenz:* Für nicht-**`null`** Referenzen `x` und `y` sind mehrfache Aufrufe von `x.equals(y)` entweder immer **`true`** oder immer **`false`**, solange keine Information in den Objekten `x` oder `y` verändert wurde, die von der Methode `equals` überprüft wird.
- Außerdem gilt für eine nicht-**`null`** Referenz `x`: `x.equals(`**`null`**`)` ergibt **`false`**.

Gleichheit von Objekten

- Diese Vorschrift *soll* eingehalten werden, wenn man in einer Klasse die Methode `equals` überschreibt. Das ist aber eine *semantische* Vorschrift, die nicht vom Compiler überprüft, sondern nur vom Programmierer bewiesen werden kann.
- Die Implementierung der Methode `equals` in der Klasse `Object` selbst ist die schärfste Möglichkeit, diese Vorschrift umzusetzen.

Exakter Vergleich von Zeichenketten

- Die Klasse String überschreibt `equals` so, dass für einen String `s` und ein Objekt `o` gilt:
  `s.equals(o)` g.d.w.:
  - `o` ist nicht **`null`**
  - `o` ist vom Typ String und
  - `o` repräsentiert genau die gleiche Zeichenkette wie `s` (d.h. `s` und `o` haben gleiche Länge und an jeder Stelle steht der gleiche Character.
- Genügt die Implementierung von `equals` in der Klasse `String` der Vorschrift für die Implementierung, die in der Klasse `Object` gegeben ist?

Abgeschwächte Gleichheit von Zeichenketten

- Eine abgeschwächte Gleichheit auf Strings achtet nicht auf den Unterschied zwischen Klein- und Großbuchstaben:
  `public boolean equalsIgnoreCase(String anotherString)`.
- Gleichheit am Anfang bzw. am Ende:
  `public boolean startsWith(String prefix)` bzw.
  `public boolean endsWith(String suffix)`
- Gleichheit des Inhalts mit anderen Objekten vom Typ `CharSequence` (also z.B. ein `StringBuilder`, ein `StringBuffer` oder auch ein anderer `String`):
  `public boolean contentEquals(CharSequence cs)`

Überblick

Natürliche Ordnung auf Objekten

- `equals` implementiert eine *Äquivalenzrelation*, wie wir gesehen haben, d.h. eine Relation, die reflexiv, symmetrisch und transitiv ist.
- Eine *(partielle) Ordnung* ist eine Relation, die reflexiv, transitiv und antisymmetrisch ist.
- Eine *totale Ordnung* ist eine partielle Ordnung, die außerdem total ist.
- Das Interface `Comparable<T>` definiert eine *totale Ordnung* R auf der Klasse T durch die Methode `int compareTo(T o)` wie folgt:

$$R \subseteq T \times T \quad \text{mit} \quad (x, y) \in R \iff \texttt{x.compareTo(y) <= 0}$$

- Wenn eine Klasse das Interface `Comparable` implementiert, nennt man diese durch `compareTo` definierte totale Ordnung auch die *natürliche Ordnung* (*natural ordering*) der entsprechenden Klasse.

Natürliche Ordnung auf Objekten

Für zwei Objekte `x` und `y` vom Typ `T implements Comparable<T>` gilt:

- Der Ausdruck `x.compareTo(y)` ist vom Typ **`int`**.
- Der Wert des Ausdrucks `x.compareTo(y)` soll kleiner 0 sein, wenn `x` "kleiner" ist als `y`.
- Der Wert des Ausdrucks `x.compareTo(y)` soll größer 0 sein, wenn `x` "größer" ist als `y`.
- Der Wert des Ausdrucks `x.compareTo(y)` soll gleich 0 sein, wenn `x` "gleich" ist zu `y`.
- Diese "Gleichheit" (`x.compareTo(y)==0`) soll normalerweise die gleiche Gleichheit sein, wie die von `x.equals(y)` angegebene, dies ist aber nicht notwendigerweise so.
- **Achtung:** Falls eine Klasse `Comparable` so implementiert, dass `x.compareTo(y)==0` $\not\Leftrightarrow$ `x.equals(y)`,
  sollte in der Dokumentation deutlich darauf hingewiesen werden. Die empfohlene Bemerkung ist: *Note: this class has a natural ordering that is inconsistent with equals.*

Überblick

Lexikographischer Vergleich von Zeichenketten

- Die Klasse `String` implementiert `Comparable<String>`.
- Als natürliche Ordnung über `String`s wird dabei die lexikographische Ordnung angenommen.
- Grundlage des Vergleichs ist der `<`-Operator, angewendet auf die einzelnen **`char`**s der beiden Strings.
- Wenn zwei `String`s, `s1` und `s2`, an einem Index oder mehreren Indices aus unterschiedlichen Charactern bestehen, dann ist der Wert von `s1.compareTo(s2)` der Wert von `s1.charAt(k)-s2.charAt(k)`, wobei *k* der kleinste der unterschiedlichen Indices ist.
- Unterscheiden sie sich an keiner Stelle, dann ist der Wert von `s1.compareTo(s2)` der Wert von `s1.length()-s2.length()`.

Lexikographischer Vergleich von Zeichenketten

- Auch für den lexikographischen Vergleich gibt es die Möglichkeit, große und kleine Buchstaben als gleich anzusehen.
- `public int compareToIgnoreCase(String str)` vergleicht zwei Strings wie `compareTo`, nachdem auf jeden Character `character` die Anweisung `Character.toLowerCase(Character.toUpperCase(character))` angewendet wurde.

Überblick

Vergleich von Mustern

- Um ein Muster in einer Zeichenkette zu suchen, gibt es in der Klasse `String` z.B. die Methode
  `public String replaceAll(String regex, String replacement)`.
- Der Parameter `regex` definiert dabei einen sogenannten *regulären Ausdruck*.
- Eine andere hilfreiche Methode, die auf regulären Ausdrücken basiert, ist `public String[] split(String regex)`.

Exkurs: Reguläre Ausdrücke

- Reguläre Ausdrücke sind ein mächtiges Werkzeug, um Muster in Zeichenketten zu beschreiben.
- Reguläre Ausdrücke sind auch von theoretischer Bedeutung für die Beschreibung bestimmter Typen von Sprachen.
- Als Werkzeug in konkreten Programmierproblemen sind reguläre Ausdrücke auch von immenser praktischer Bedeutung. In dieser Hinsicht werden sie aber im Studium kaum vertieft.
- Wir geben an dieser Stelle ein paar einführende Hinweise.

Exkurs: Reguläre Ausdrücke

Definition

Reguläre Ausdrücke sind wie folgt für ein Alphabet Σ (d.h. eine Menge von Symbolen, also z.B. alle möglichen Werte vom Typ **`char`**) wie folgt *induktiv* definiert:

- $\emptyset$ (= die leere Menge) ist ein regulärer Ausdruck.
- ε (= das leere Wort) ist ein regulärer Ausdruck.
- Für jedes $a \in \Sigma$ ist a ein regulärer Ausdruck.
- Wenn a und b reguläre Ausdrücke sind, dann sind auch
  - ab (Konkatenation)
  - $a|b$ (Vereinigung)
  - $(a)^*$ (Stern-Operator), d.h. die durch den regulären Ausdruck a definierte Zeichenkette kann beliebig oft (auch 0-mal) hintereinander vorkommen.

  reguläre Ausdrücke.

Exkurs: Reguläre Ausdrücke

Konstrukte für reguläre Ausdrücke

Viele Programmiersprachen bieten Konstrukte an, um reguläre Ausdrücke zu definieren, die dann verwendet werden können, um spezielle Zeichenketten innerhalb von Zeichenketten zu finden. In Java gibt es z.B.:

- `X` für das Zeichen `X`.
- `[abc]` für die Zeichen `a`, `b` oder `c`. Mit den eckigen Klammern wird eine sogenannte *Zeichenklasse* definiert, entspricht der Vereinigung. Möglich ist z.B. auch `[a-z]` für alle Kleinbuchstaben.
- Abkürzungen für bestimmte Zeichenklassen, z.B. `.` für ein beliebiges Zeichen, `\d` für eine Ziffer (= `[0-9]`).
- Quantifier, die beschreiben, wie oft eine Zeichenkette hintereinander auftreten darf, z.B.:
  - `X*` `X` kommt beliebig oft vor (0-mal oder öfter)
  - `X+` `X` kommt einmal oder öfter vor (= `X(X)*`)
  - `X{n}` `X` kommt genau *n*-mal vor
  - `X{n,m}` `X` kommt mindestens *n*-mal, höchstens *m*-mal vor
  - `X?` `X` kommt 0 oder 1-mal vor

Exkurs: Reguläre Ausdrücke

Pattern bilden

- Ein Konstrukt, das einen regulären Ausdruck beschreibt, kann man in einem String bilden.
- Aus einem solchen String kann man ein Objekt der Klasse `Pattern` bilden (durch die statischen Methoden `Pattern.compile`).
- In der Klasse `Pattern` sind die Möglichkeiten für reguläre Ausdrücke in Java ausführlich dokumentiert.
- Worauf matchen folgende Pattern?
  - `Pattern.compile("Kr(i|o)ege(l|r)")`
  - `Pattern.compile("Info (I{1,3}|IV)")`

Pattern Matching

- Aus einem `Pattern`-Objekt `p`, das durch den String `regex` (Darstellung eines regulären Ausdrucks) definiert wurde, kann man ein `Matcher`-Objekt `mStr` spezifisch für einen String `str` erzeugen, um das Pattern `p` in `str` zu finden, zu ersetzen usw.
- Nun können Sie das Verhalten der String-Methode `public String replaceAll(String regex, String replacement)` verstehen, die eine Abkürzung für folgende Befehle ist:

```
Pattern p = Pattern.compile(regex);
Matcher mStr = p.matcher(str);
String neuerString = mStr.replaceAll(replacement);
// oder auch:
String neuerString2 =
  Pattern.compile(regex).matcher(str)
         .replaceAll(replacement);
```

- Damit ist nicht-exakte Suche mit Mustern in Strings möglich, das sogenannte *Pattern Matching*.

Überblick

String-Darstellung von Objekten

- Die Klasse `Object` definiert eine Instanzen-Methode `public String toString()`, die eine String-Repräsentation des Objektes zurückgibt.
- In `Object` ist dies eine Zeichenfolge zusammengesetzt aus dem Klassen-Namen, dem `@`-Zeichen und der Hexadezimal-Darstellung des Hash-Codes des Objektes, der für Objekte vom Typ `Object` aus der internen Speicherdarstellung erzeugt wird. (Die eigentliche Bedeutung des Hash-Codes lernen wir später kennen.)
- Da jede Klasse eine Unterklasse von `Object` ist, hat auch jedes Objekt von beliebigem Typ diese Fähigkeit, eine String-Repräsentation von sich zu geben.
- Wie wir in den Übungen bereits gesehen haben, kann man für eine bestimmte Klasse eine andere (und möglicherweise sinnvollere) Repräsentation implementieren.

String-Darstellung von Objekten

- Beispiele:
  - Ein Objekt vom Typ `StringBuilder` gibt den Inhalt der internen Zeichenkette als String zurück.
  - Ein Objekt vom Typ `Double` gibt eine String-Repräsentation der dargestellten Zahl zurück.
- Diese Methode wird oftmals implizit aufgerufen, z.B., wenn man der Methode `System.out.println(Object x)` ein Objekt übergibt.

Abschnitt 18: Beispiel: I/O-Streams

Überblick

Interaktion mit Programmen

- Eine wichtige Eigenschaft von Programmen ist die Abstraktion von konkreten Daten.
- Eine Möglichkeit der Abstraktion von konkreten Daten ist die Verwendung von *formalen Parametern*, die bei der Verwendung des Programmes für eine bestimmte Aufgabe mit *aktuellen Parametern* belegt werden und so das *abstrakte* Programm auf *konkrete* Daten anwenden.
- Bisher haben wir die Verwendung von Programmen mit Parameterübergabe kennengelernt (`main`-Methode mit Parameter-Array). Weitere Möglichkeiten sind:
  - interaktive Abfrage vom Benutzer während der Laufzeit
  - Einlesen von Dateien, Datenströmen, Sensormesswerten etc.

Interaktion mit Programmen

- Umgekehrt gibt ein Programm auch oft Information an den Benutzer zurück.
- Wir haben schon oft die einfache Ausgabe mit `System.out.println(String s)` benutzt.
- Wenn es sehr viel Information ist, sollte ein Programm direkt in eine oder mehrere Dateien schreiben können.
- Vielleicht soll ein Programm auch mal Information an ein anderes Programm übergeben.

Eingabe und Ausgabe in Java

- In Java sind die grundlegenden Klassen für Ein- und Ausgabe im Package `java.io` gesammelt.
- Wir werden nun einen Blick auf einige dieser Klassen und ihren Gebrauch werfen.
- Nach diesem Kapitel wissen Sie dann auch endlich, was bei `System.out.println(String s)` (eine Methode, die wir von Anfang an benutzt haben, ohne wirklich etwas darüber zu wissen) eigentlich passiert.

Überblick

Abstraktion von Dateien

Betrachten wir zunächst eine Abstraktion von Dateien:

- Eine Datei hat einen Namen bzw. einen (relativen oder absoluten) Pfad, der auf einem gegebenen Dateisystem zu dieser Datei führt.
- Damit ist eine Datei (relativ oder absolut) eindeutig charakterisiert.
- Eine Datei kann gelesen, geschrieben, gelöscht, neu angelegt werden.
- Eine Datei kann verborgen sein, hat einen Eintrag, wann die letzte Änderung stattfand, eine Größe.
- Eine Datei kann ein Verzeichnis sein und wiederum andere Dateien beinhalten.
- Eine Abstraktion, die Klasse `File`, bezieht sich auf diese und andere Eigenschaften, ohne vom Inhalt der Datei zu wissen.

Objekte vom Typ `File`

- Ein Objekt der Klasse `File` kann durch vier Konstruktoren erzeugt werden:
  - `File(String pathname)` – ein `File`-Objekt, das eine Datei für den angegebenen Pfad repräsentiert (*kann* existieren, *muss* aber nicht).
  - `File(String parent, String child)` – ein `File`-Objekt namens `child` innerhalb des Verzeichnisses mit Namen `parent`.
  - `File(File parent, String child)` – hier wird das Elternverzeichnis nicht durch den Namen, sondern bereits durch ein `File`-Objekt angegegeben.
  - `File(URI uri)` – die Datei wird durch ein `URI`-Objekt spezifiziert, also ein Objekt, das eine Uniform Resource Identifier (URI) Referenz repräsentiert (siehe Klasse `java.net.URI`).
- Tipp: Die Bestandteile eines Pfades werden unter Unix mit Slash (`/`) getrennt, unter Windows mit Backslash (`\`). Java ermöglicht aber, diese Eigenschaft portabel zu halten: Die statische Variable `File.separator` hält den auf dem aktuellen System gültigen Trenner.

Verzeichnisse

- Stellt ein `File`-Objekt `d` ein Verzeichnis dar (`d.isDirectory()`), kann das Objekt ein Array der Dateien in diesem Verzeichnis zu Verfügung stellen, z.B. `d.listFiles()` – oder `d.listFiles(FileFilter filter)`, um nur bestimmte Dateien (abhängig vom angegebenen Filter) in das Array aufzunehmen (z.B. nur Verzeichnisse).
- Soll für ein `File`-Objekt `d`, das auf dem Filesystem noch keinem realen File entspricht, ein Verzeichnis angelegt werden, gibt es die Methoden `d.mkdir()` bzw. `d.mkdirs()`.
- Die Methode `public File getParentFile()` gibt ein Objekt zurück, das das Verzeichnis repräsentiert, in dem dieses File-Objekt liegt.
- Lese- und Schreibzugriffe auf das Filesystem können zu Ausnahmen führen (am häufigsten `SecurityException` und `IOException`).

Beispiel: Ausgabe eines Verzeichnisbaumes

FileFilter für Verzeichnisse

```
import java.io.File;
import java.io.FileFilter;

/**
 * FileFilter, der nur Verzeichnisse akzeptiert.
 */
public class DirectoryFileFilter implements FileFilter
{
  /**
   * Akzeptiert nur Verzeichnisse.
   *
   * @param pathname File-Objekt, das dem Filter uebergeben wird
   * @return true, wenn pathname ein Verzeichnis repraesentiert, false sonst
   * @see java.io.FileFilter#accept(java.io.File)
   */
  public boolean accept(File pathname)
  {
    return pathname.isDirectory();
  }
}
```

Beispiel: Ausgabe eines Verzeichnisbaumes

Verzeichnisbaum ausgeben

```
import java.io.File;

/**
 * Klasse zur Ausgabe eines Verzeichnisbaumes.
 */
public class Verzeichnisbaum
{
  /**
   * Ein FileFilter, der nur Verzeichnisse akzeptiert.
   */
  public static final DirectoryFileFilter DIRECTORY_FILE_FILTER =
                                          new DirectoryFileFilter();

  /**
   * Die Einrueckung, um die eine tiefere Ebene
   * im Verzeichnisbaum eingerueckt wird.
   */
  public static final String EINRUECKUNG = "  ";
```

Beispiel: Ausgabe eines Verzeichnisbaumes

Verzeichnisbaum ausgeben (cont.)

```
/**
 * Gibt den Verzeichnisbaum ausgehend
 * vom angegebenen Verzeichnis rekursiv aus.
 *
 * @param verzeichnis das Verzeichnis,
 * von dem ausgehend der Verzeichnisbaum ausgegeben werden soll
 * @param einrueckung die Einrueckung fuer die aktuelle Tiefe im Baum
 * (wird fuer eine tiefere Ebene jeweils um
 * {@link #EINRUECKUNG EINRUECKUNG} erweitert).
 */
public static void gibVerzeichnisbaumAus(File verzeichnis, String einrueckung)
{
  File[] verzeichnisse = verzeichnis.listFiles(DIRECTORY_FILE_FILTER);
  for(int i = 0; i < verzeichnisse.length; i++)
  {
    System.out.println(einrueckung+verzeichnisse[i].getName());
    gibVerzeichnisbaumAus(verzeichnisse[i], einrueckung+EINRUECKUNG);
  }
}
```

Beispiel: Ausgabe eines Verzeichnisbaumes

Verzeichnisbaum ausgeben (cont.)

```
  /**
   * Gibt den Verzeichnisbaum eines Verzeichnisses aus.
   *
   * Wenn ein Verzeichnis angegeben wird, wird der Baum
   * von diesem Verzeichnis aus durchwandert,
   * andernfalls vom aktuellen Arbeitsverzeichnis ausgehend.
   *
   * @param args wenn das Array leer ist, geht die Ausgabe des
   * Verzeichnisbaumes vom aktuellen Arbeitsverzeichnis aus, andernfalls
   * vom angegebenen Verzeichnis
   */
  public static void main(String[] args)
  {
    if(args.length > 0)
    {
      gibVerzeichnisbaumAus(new File(args[0]), "");
    }
    else
    {
      gibVerzeichnisbaumAus(new File(System.getProperty("user.dir")),"");
    }
  }
}
```

Beispiel: Ausgabe eines Verzeichnisbaumes

zum Ausprobieren...

Diese Beispielklassen finden Sie unter

```
http://www.dbs.ifi.lmu.de/Lehre/EIP/WS_2008/skript/programmbeispiele/iostreams/DirectoryFileFilter.java
```

bzw.

```
http://www.dbs.ifi.lmu.de/Lehre/EIP/WS_2008/skript/programmbeispiele/iostreams/Verzeichnisbaum.java
```

Überblick

Was ist ein Stream?

- Wie wir gesehen haben, stellt ein `File`-Objekt eine Abstraktion einer Datei dar, die vom Inhalt der Datei absieht.
- Datei-Inhalte (und andere Informationen, die gelesen oder geschrieben werden) werden durch das Konzept der *Streams* modelliert.
- Ein *Stream* ist zunächst auch ein abstraktes Konstrukt, das grundsätzlich für die Fähigkeit steht, Zeichen auf ein (imaginäres) Ausgabegerät zu schreiben (`OutputStream`) oder von einem (imaginären) Eingabegerät zu lesen (`InputStream`).

Was liest oder schreibt ein Stream?

- Eigentliche Grundlage von Lese- und Schreiboperationen sind Bytes.
- Die ursprünglichen Klassen für Lese- und Schreiboperationen sind `InputStream` und `OutputStream`, die *Byte-Streams* behandeln.
- Dabei ist jede Transporteinheit 8 Bit lang, das gewährleistet Kompatibilität zu Textdateien, die mit konventionellen Programmiersprachen erstellt wurden oder gelesen werden sollen.
- Probleme gibt es allerdings mit den 16 Bit langen Unicode-Zeichen, die innerhalb von Java zur Zeichendarstellung verwendet werden.

Character-Streams

- Zur Verarbeitung von Text gibt es daher die *Character-Streams*, `Reader` bzw. `Writer`.
- Brückenklassen ermöglichen die Verbindung von *Byte-Streams* und *Character-Streams*: So erhält ein Konstruktor der Klasse `InputStreamReader` als Parameter einen `InputStream`.
- `Reader` bzw. `Writer` sind der gewöhnliche Weg, um *Text*dateien zu lesen bzw. zu schreiben. *Byte-Streams* sind eher von Bedeutung, um mit nicht-textuellen Daten umzugehen.

Überblick

Die Basis-Klasse `InputStream`

- Die (abstrakte) Klasse `InputStream`, von der alle Eingabe-Byte-Streams erben, stellt Methoden zum Lesen von der jeweiligen Quelle (die im Konstruktor übergeben wird) zu Verfügung.
- Im Gegensatz zur Klasse `File` sind bei einem `InputStream` die meisten Methoden darauf angewiesen, dass die angegebene Eingabequelle tatsächlich existiert und lesbar ist. Deshalb sind hier für die meisten Methoden `IOExceptions` (aus unterschiedlichen Gründen) möglich.
- Unterschiedliche `read`-Methoden der Klasse geben ein gelesenes Byte als **`int`** zurück oder schreiben es in ein übergebenes **`byte`**-Array.
- Wird der Eingabe-Stream nicht mehr benötigt, dann schließt man ihn mittels `close()`.

Einige von `InputStream` abgeleitete Klassen

- `FileInputStream`: Byte-Stream zum Lesen aus einer Datei.
- `SequenceInputStream`: Kann zwei oder mehr Eingabe-Streams so verbinden, dass die Daten nacheinander aus den angegebenen Quellen gelesen werden.
- `FilterInputStream`: Eine Klasse, die einen `InputStream` hält und Methoden-Aufrufe an diesen `InputStream` weitergibt. Ableitende Klassen können die Methoden dann einfach überschreiben, um besondere Aktionen beim Lesen (z.B. Filter) durchzuführen.
- Wichtige Unterklasse von `FilterInputStream`: Der `BufferedInputStream`, der den Lesevorgang puffert und damit für viele Fälle die Performanz deutlich erhöht.

Die Basis-Klasse `Reader`

- Die abstrakte Basis-Klasse aller Character-Input-Streams ist der `Reader`.
- Der `Reader` stellt (analog zum `InputStream`) Methoden zu Verfügung, um das nächste Zeichen zu lesen und zurückzugeben (als **`int`**) oder in ein bereitgestelltes **`char`**-Array zu schreiben.
- Auch hier gilt: Wird der Character-Eingabe-Strom nicht mehr benötigt, dann schließt man ihn mittels `close()`.

Abgeleitete Klassen: Auswahl der Datenquelle

- Als abgeleitete Klasse kennen wir schon den `InputStreamReader`. Hier ist die Datenquelle ein ByteStream.
- Andere Unterklassen sind z.B.:
  - `FileReader` – zum Einlesen aus einer Datei.
  - `BufferedReader` – puffert die Eingabe und ermöglicht so das Einlesen von ganzen Zeilen. Dieser Reader ist für viele Probleme der Reader der Wahl, wenn man Text zeilenweise einlesen will.
  - `LineNumberReader` – Ableitung aus dem `BufferedReader`, der zusätzlich die Zeilen zählen kann.
  - `FilterReader` – abstrakte Klasse als Basis zur Konstruktion von Eingabefiltern (benutzt als Quelle einen beliebigen weiteren `Reader`).

Beispiel: Ausgabe des Inhalts einer Textdatei

```
public static void main(String[] args) throws IOException
{
  BufferedReader reader = new BufferedReader(
                              new FileReader(
                                  new File(args[0])));
  for(String line; (line=reader.readLine())!=null;)
  {
    System.out.println(line);
  }
}
```

Der Standard-Eingabestrom: `System.in`

- Die Klasse `System` hält einen `InputStream`, `System.in`, als statische Eigenschaft, der standardmäßig auf den Konsoleninput verweist.
- Über diesen `InputStream`, der mit Beginn des Programms zunächst offen ist, können also Tastatureingaben abgefragt werden.
- Auf `System.in` kommen auch Daten an, die in ein Programm gepiped werden. Beispiel auf einer Unix-Konsole:
  ```
  programm1 | programm2
  ```
  Der Output (`System.out`) von `programm1` wird der Input (`System.in`) von `programm2`.

Beispiel: Abfrage von Benutzereingaben

```
public static void main(String[] args) throws IOException
{
  BufferedReader input = new BufferedReader(
                                 new InputStreamReader(
                                     System.in));
  System.out.println("Wie heisst Du?");
  String benutzerName = input.readLine();
  System.out.println("Hallo, " + benutzerName);
}
```

Beispiel: Verarbeiten von Konsolen-Eingabeströmen

```
public static void main(String[] args) throws IOException
{
  BufferedReader input = new BufferedReader(
                                 new InputStreamReader(
                                     System.in));
  for(String line; (line=input.readLine())!=null;)
  {
    System.out.println(line);
  }
}
```

Überblick

Die Basis-Klasse `OutputStream`

- `OutputStream` ist die Basis-Klasse für Byte-Outputstreams.
- Wie beim `InputStream` gibt es eine `close`-Methode, um einen `OutputStream` nach Gebrauch zu schließen.
- Meistens muss man außerdem explizit die `flush`-Methode aufrufen, um Daten, die gepuffert sind, physikalisch auf das Zielgerät zu schreiben.
- Analog zu den `read`-Methoden des `InputStream`s gibt es `write`-Methoden, um Bytes (als **`int`** übergeben) oder **`byte`**-Arrays zu schreiben.
- Auch hier muss man mit `IOException`s rechnen, da ja das Zielgerät vorhanden und beschreibbar sein muss, damit die Methoden ohne Fehler arbeiten können.

Einige von `OutputStream` abgeleitete Klassen

- `FileOutputStream` – um in eine Datei zu schreiben.
- `ByteArrayOutputStream` – um in ein **`byte`**-Array zu schreiben.
- `FilterOutputStream` – ermöglicht einen Filterschritt vor dem Schreiben.
- `BufferedOutputStream` – erbt von `FilterOutputStream`, schaltet einen Puffer dazwischen, was die Performanz verbessern kann.
- `PrintStream` – erweitert `FilterOutputStream` um die Fähigkeit, bequem alle möglichen Datentypen (primitive Typen, Strings und allgemein Objekte) zu schreiben (mit den `print` und `println`-Methoden). Die Besonderheit eines `PrintStream`s ist, dass keine `IOException`s geworfen werden. Stattdessen werden die `IOException`s, die der zugrundeliegende Stream wirft, als Fehler gemerkt. Der Fehler-Status des `PrintStream`s ist dann über die `checkError`-Methode abrufbar.

Die Basis-Klasse `Writer`

- Das Pendant zum `Reader` als Character-Outputstream ist der `Writer`.
- Wie beim Byte-Outputstream gibt es `close`, `flush` und `write`-Methoden, letztere erhalten aber Parameter vom Typ **`char`** bzw. **`int`** statt **`byte`**.
- Außerdem gibt es `write`-Methoden, die einen `String` erhalten können.

Abgeleitete Klassen: Auswahl des Ziels

- `OutputStreamWriter` – übergibt den Character Output an einen Byte-Outputstream.
- `FileWriter` – erweitert den `OutputStreamWriter`, um in eine Datei zu schreiben.
- `PrintWriter` – ermöglicht wie der `PrintStream` die bequeme Ausgabe aller möglichen Datentypen (primitive Typen, Strings und allgemein Objekte) in einen Text Outputstream zu schreiben (mit den `print` und `println`-Methoden).
  Wie beim `PrintStream` werden keine `IOExceptions` geworfen, sondern als Fehler gemerkt. Der Fehler-Status des `PrintWriters` ist über die `checkError`-Methode abrufbar.
- `BufferedWriter` – puffert die Ausgabe zur Verbesserung der Performanz.

Beispiel: Schreiben in eine Datei

```
public static void schreibe(String text,
                            File zieldatei)
                            throws IOException
{
  PrintWriter out = new PrintWriter(
                          new FileWriter(zieldatei));
  out.print(text);
  out.flush();
  out.close();
}
```

Standard- und Fehler-Ausgabestrom: `System.out`, `System.err`

- Mit `System.out` und `System.err` stehen zwei `PrintStreams` zu Verfügung, die standardmäßig offen sind (und keine `IOExceptions` werfen).
- Daten, die durch `System.out.println` in den Standard-`PrintStream` geschrieben werden, erscheinen (sofern `System.out` nicht umgeleitet wurde) auf der Konsole (Standard-Ausgabe).
- Daten, die durch `System.err.println` in den Error-`PrintStream` geschrieben werden, erscheinen (sofern `System.err` nicht umgeleitet wurde) auf der Konsole (Fehler-Ausgabe).

Standard- und Fehler-Ausgabestrom: `System.out`, `System.err`

- Die Methode `printStackTrace()` eines Objekts der Klasse `Throwable` (also z.B. eine `IOException`) schreibt in die Fehler-Ausgabe `System.err`.
- Beim Aufruf eines Programmes über die Konsole erscheinen beide Ausgabeströme, `System.out` und `System.err`, gemischt. Man kann sie aber unterschiedlich behandeln, z.B. durch Pipen (hier unter `bash`):
  ```
  java programm 2> error.log
  ```
  Dieser Aufruf führt dazu, dass der Error-Stream in die Datei `error.log` umgeleitet wird (die dabei neu angelegt bzw. überschrieben wird), während der Standard-Stream auf der Konsole ausgegeben wird.

Überblick

Typische I/O Ausnahmen

- Neben den Klassen zur Ein- und Ausgabeverarbeitung finden sich im Package `java.io` auch diverse Ausnahmen, die verschiedene mögliche Fehler signalisieren.
- `IOException` ist die Oberklasse für alle Ausnahmen, die etwas mit dem I/O-Prozess zu tun haben.
- Eine häufig speziell behandelte Unterklasse dieser Exception ist `FileNotFoundException`, die auftritt, wenn von einer nicht-existierenden (oder aus sonstigen Gründen nicht auffindbaren/lesbaren/schreibbaren/erstellbaren) Datei gelesen oder in sie geschrieben werden soll (also z.B. beim Konstruieren eines `FileInputStreams`).

Beispiel: Unterschiedliche Ausnahmen abfangen

```
try
{
  BufferedReader reader = new BufferedReader(
                              new FileReader(
                                  new File(args[0])));
  try
  {
    for(String line; (line=reader.readLine())!=null;)
    {
      System.out.println(line);
    }
  }
  catch(IOException e)
  {
    e.printStackTrace();
  }
}
catch(FileNotFoundException e)
{
  e.printStackTrace();
}
```

Teil III

Entwurf objektorientierter Programme

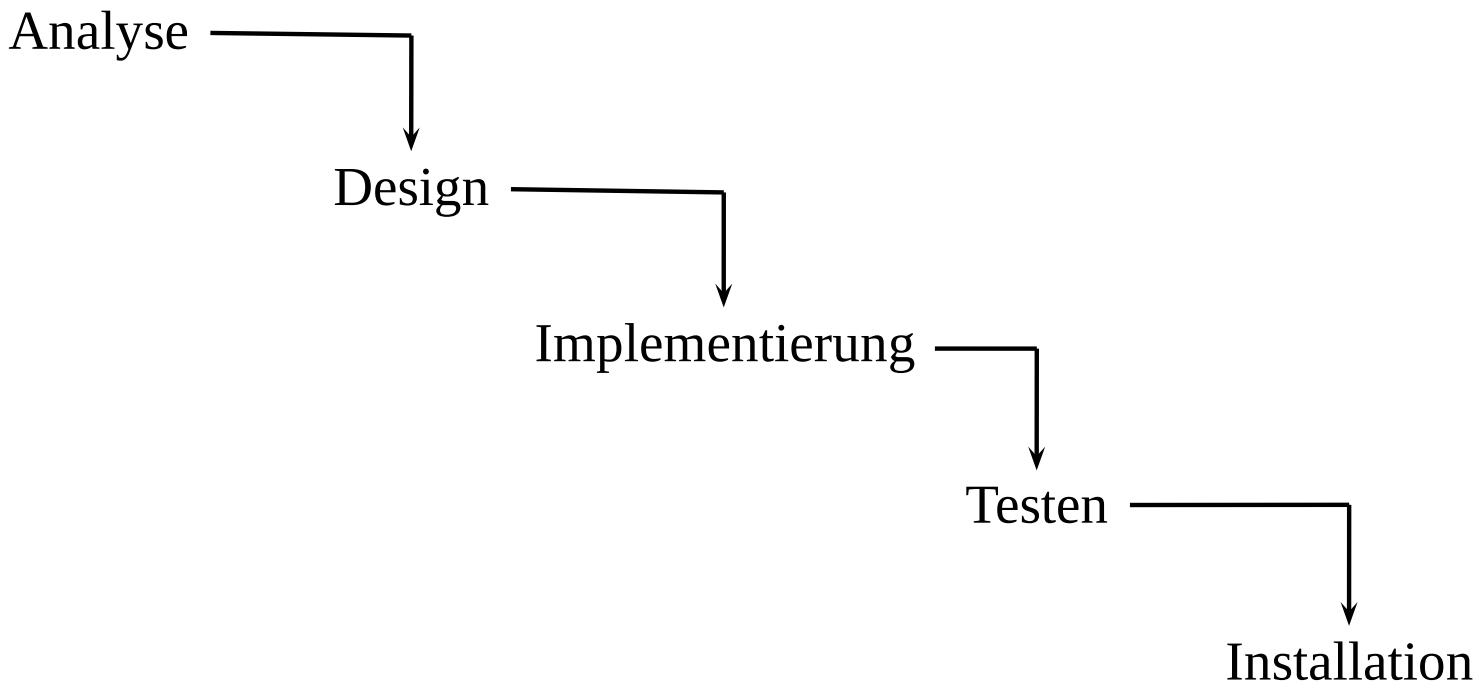
Abschnitt 19: Objektorientiertes Design

19. Objektorientiertes Design

Software-Entwicklung

- Zur Software-Entwicklung existiert eine Vielfalt von Vorgehensweisen und Modellen.
- Hier betrachten wir nur das sogenannte *Wasserfallmodell*, das den Prozess der Software-Entwicklung in fünf Phasen aufgliedert.

Das Wasserfallmodell



- Klare Trennung der verschiedenen Phasen.
- Schwierigkeiten in einer Phase verzögern das Gesamtprojekt.
- Streng sequentielles Vorgehen ist in der Praxis kaum möglich.

Wasserfallmodell: Anforderungsanalyse (requirements analysis)

- Analyse des Problembereichs
- Festlegung der (funktionalen und nicht funktionalen) Anforderungen an das Programm:
  Was soll das Programm/System leisten?
- Festlegung von organisatorischen Richtlinien (Aufwandsabschätzung, Terminplanung. . .) und Rahmenbedingungen (z.B. vorhandene Soft- und Hardware)
- Skizze der Systemarchitektur

Ergebnis der Anforderungsanalyse ist ein Anforderungskatalog (Anforderungsspezifikation).

Wasserfallmodell: Entwurf (design)

- Beschreibt die Art und Weise, in der die gestellten Aufgaben gelöst werden sollen:
  Wie lösen wir das Problem?
- Festlegung der Systemarchitektur.
- Entwurf der einzelnen Systemkomponenten (Wahl von Datenrepräsentationen und Algorithmen).
- *Objektorientierter* Entwurf: Festlegung der Klassen und Methoden.

Ergebnis der Entwurfsphase ist eine Entwurfsbeschreibung (z.B. mittels eines UML-Diagramms).

Wasserfallmodell: Entwurf (design)

Identifizierung von Klassen, Methoden und Assoziationen:

- Kandidaten für Klassen sind:
  - Personen bzw. Rollen (z.B. Student, Angestellter, Kunde, ...)
  - Organisationen (z.B. Firma, Abteilung, Uni, ...)
  - Gegenstände (z.B. Artikel, Flugzeug, Immobilien, ...)
  - begriffliche Konzepte (z.B. Bestellung, Vertrag, Vorlesung, ...)
- Möglichkeit: Durchsuchen des Anforderungskatalogs nach Substantiven, die Mengen bezeichnen
- Heuristik zum Auffinden von Methoden: Suche nach Verben
- Kandidaten für Assoziationen sind physische oder logische Verbindungen mit einer bestimmten Dauer, wie
  - konzeptionelle Verbindungen (arbeitet für, ist Kunde von, ...)
  - Besitz (hat, ist Teil von, ...)
  - (häufige) Zusammenarbeit von Objekten zur Lösung einer Aufgabe

Wasserfallmodell: Entwurf (design)

Beispiel: Identifizierung von Klassen, Methoden und Assoziationen

- Geg: Programm, welches Rechnungen für Kunden ausdruckt. Die Rechnungen sollen die Einzelposten sowie jeweils den Gesamtpreis enthalten.
- Klassen: Rechnung, Kunden, Einzelposten
- Methoden: ausdrucken, Gesamtpreis berechnen
- Assoziationen: Rechnungen *enthalten* Einzelposten, Rechnungen *für* Kunden

Wasserfallmodell: Entwurf (design)

Entwurf-Optimierung:

- Je weniger Beziehungen zwischen Klassen, desto unabhängiger kann man sie implementieren (hilft bei der Verteilung der Arbeit).
- Mögliche Schritte zur Optimierung:
  1. Reduziere Beziehungen
  2. Fasse gemeinsame Aspekte von Klassen zu Oberklassen zusammen
  3. Finde isolierte Inseln im UML-Diagramm
     → Arbeit organisieren, aufteilen

Wasserfallmodell: Implementierung, Test und Installation

- Implementierung: Codierung des Entwurfs in einer Programmiersprache, ggf. unter Wiederverwendung vorhandener Komponenten.
- Test:
  - Test der einzelnen Komponenten
  - Schrittweises Zusammenfügen einzelner Komponenten mit jeweiligem Integrationstest
  - Systemtest
  - Abnahmetest (mit “echten” Daten)
- Installation (und Wartung):
  - Installation des Systems
  - Fehlerbeseitigung nach Inbetriebnahme
  - Änderung und Erweiterung des Systems

20. Ein Grundprinzip der Software-Architektur

Modelle und Anwendungen

- Unsere Programme waren bisher oft eine Mischung aus Modellen und Anwendungen.
- Ein Modell stellt einen Ausschnitt der Welt in vereinfachter, schematisierter Form dar (z.B. die Klasse `Konto`).
- Eine Anwendung verwendet ein Modell um z.B. etwas zu berechnen, oft abhängig von Eingaben des Benutzers (z.B. eine Geschäftsabwicklung, bei der Konten angelegt und ihr Inhalt verändert wird).

Modelle und Anwendungen

- Als Anwendungen haben wir bisher `main`-Methoden kennengelernt.
- Modelle wurden bei uns bisher auf der Konsole dargestellt (`System.out.print`-Ausgaben).
- Interaktion war durch Eingabe auf die Konsole möglich (z.B. durch Start-Parameter).
- Die Konsole stellt damit eine bedeutende Schnittstelle zum Benutzer dar (Command-Line-Interface, CLI).
- Jede Klasse, die eine `main`-Methode zu Verfügung stellt, ist damit eine Anwendung (Applikation).
- Ein einziges Modell kann in sehr unterschiedlichen Anwendungen verwendet werden.

Verschiedene Arten von Anwendungen

- Eine Anwendung kann auf verschiedene Arten mit einem Benutzer interagieren.
- Neben CLI ist die bedeutenste Schnittstelle die GUI (Graphical User Interface).
- GUIs können eigenständige Programme sein oder auch auf Interaktionsmöglichkeiten über Web-Oberflächen – z.B. als Applets oder als Java-Server-Pages (JSP) – basieren.
- Wir werden im Softwareentwicklungspraktikum GUIs als sehr zentrale und Applets als java-typische, sehr schlanke Anwendungen kennenlernen.

Trennung von Modell und Benutzer-Interaktion

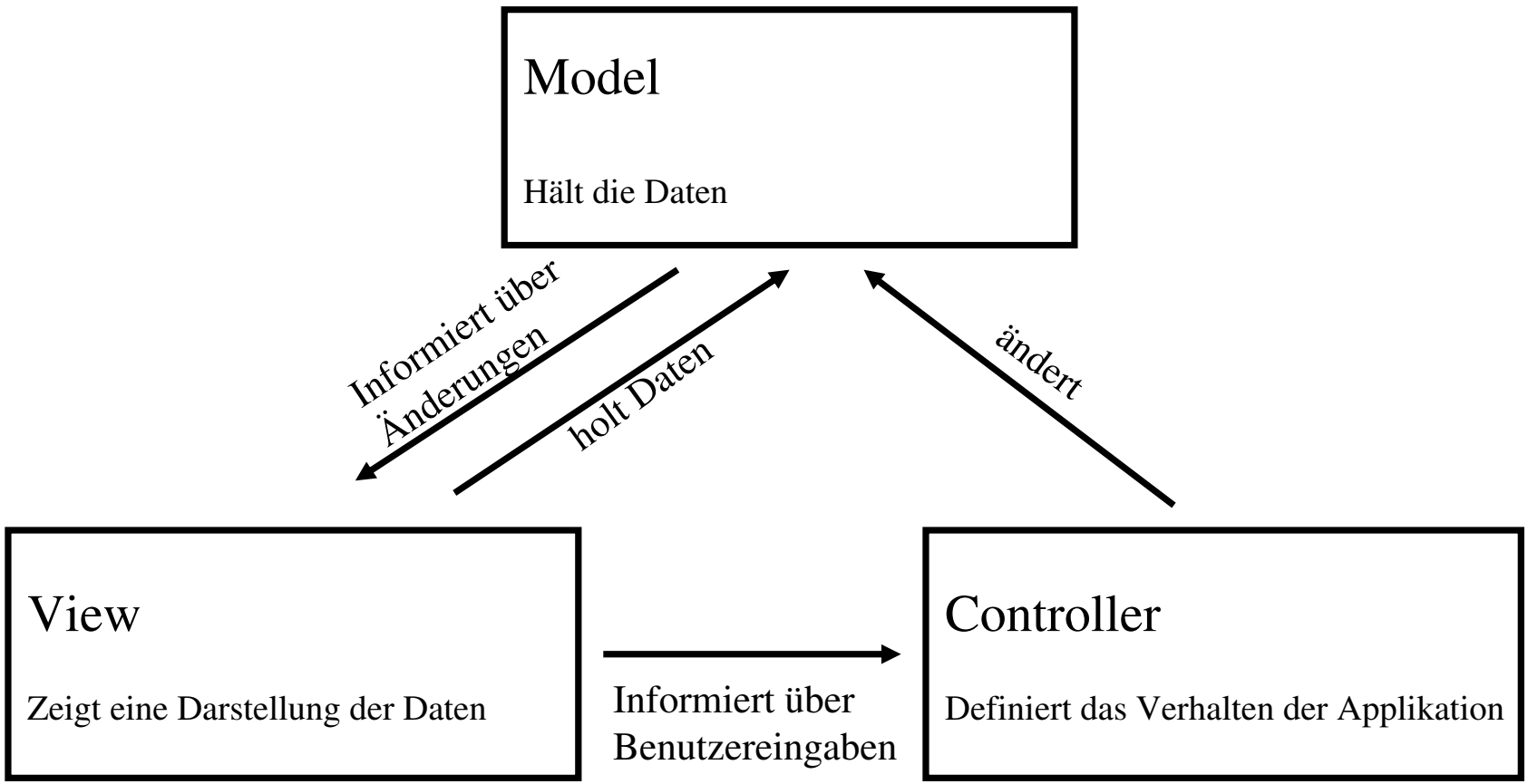
- Es ist eine wichtige Design-Hilfe für gute Programme, die Modellierung eines Ausschnitts der Wirklichkeit (das *Modell*) von den Möglichkeiten der Benutzer-Interaktion getrennt zu halten.
- In der Software-Entwicklung hat man für diese grundlegende Trennung das *Model-View-Controller* (*MVC*) Konzept geprägt.

Model-View-Controller

- Drei getrennte Programm-Komponenten sind für
  - das Modell (*Model*),
  - die Darstellung des Modells (*View*) und
  - die Beeinflussung des Modells (*Controller*)

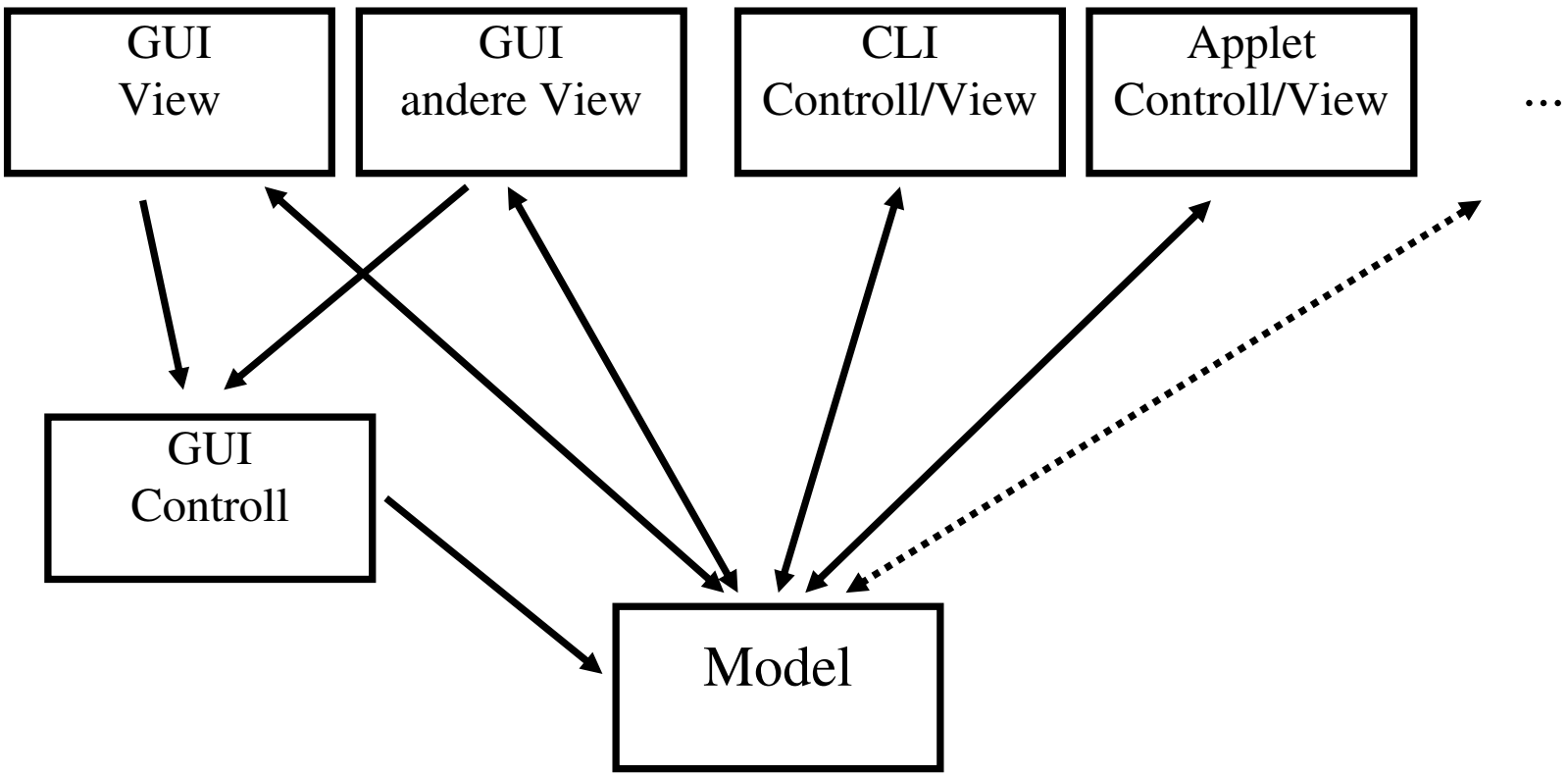
  zuständig.
- *View* und *Controller* hängen meist enger zusammen, da beide für eine bestimmte Art der Applikation (CLI, GUI, Applet, JSP, ...) bestimmt sind.
- Aber auch innerhalb des Anwendungsszenarios GUI ist es sinnvoll, View und Controller getrennt zu halten. Das Design der Darstellung (“Look-And-Feel”) kann dadurch leicht ausgetauscht werden, ohne die Kontroll-Ebene zu beeinflussen.

Model-View-Controller



Model-View-Controller

Das Modell steht dem View-Controller-Paar eher unabhängig gegenüber und könnte auch von einem ganz anderen View-Controller-Paar verwendet werden.



Applikationen

- In unseren bisherigen Programmen haben wir implizit eine CLI-View-Controll verwendet.
- Um das MVC-Prinzip bei einfachen Aufgaben mit CLI einzuhalten, achtet man z.B. darauf, dass die Programmlogik (Methoden, die etwas berechnen) getrennt ist von der View (Methoden, die etwas ausgeben). Auf diese Weise könnte die Programmlogik auch von anderen View-Controller-Paaren verwendet werden.
- Beispiele aus den Übungsaufgaben u.a.: 9-2, 10-2, 10-3, 11-1, 11-3, 11-6

Teil IV

Datenstrukturen und Algorithmen

Abschnitt 21: Datenstrukturen

Überblick

Was sind Datenstrukturen?

- Viele Computer-Programme sind in erster Linie dazu da, Daten zu verarbeiten.
- Eine Datenmenge muß dazu intern organisiert und verwaltet werden.
- Als einfache Datenstruktur zur Verwaltung gleichartiger Elemente haben wir für imperative Sprachen das Array kennengelernt.
- Im ersten Teil der Vorlesung haben wir bei den Wechselgeldalgorithmen *Folgen* als Datenstruktur verwendet.
- Ein Äquivalent zum mathematischen Konzept der *Folge* findet sich in vielen Programmiersprachen als *Liste*.
- Auch eine Klasse dient zunächst der Darstellung von Objekten, die einen Ausschnitt der Wirklichkeit abstrahiert repräsentieren.
- Als spezielle Datenstruktur können wir auch die Strings (und verwandte Klassen) betrachten, die für eine Menge von Zeichen stehen.

Die Bedeutung von Datenstrukturen

- Bei vielen Anwendungen besteht die wichtigste Entscheidung in Bezug auf die Implementierung darin, die passende Datenstruktur zu wählen.
- Verschiedene Datenstrukturen erfordern für dieselben Daten mehr oder weniger Speicherplatz als andere.
- Für dieselben Operationen auf den Daten führen verschiedene Datenstrukturen zu mehr oder weniger effizienten Algorithmen.
- Die Auswahlmöglichkeiten für Algorithmus und Datenstruktur sind eng miteinander verflochten. Durch eine geeignete Wahl möchte man Zeit und Platz sparen.

Datenstrukturen als Objekte

- Eine Datenstruktur können wir auch wieder als Objekt auffassen und entsprechend modellieren.
- Das bedeutet, dass eine Datenstruktur Eigenschaften und Fähigkeiten hat, also z.B. typische Operationen ausführen kann.
- Für Arrays haben wir z.B. die typischen Operationen:
  - Setze das *i*-te Element von `a` auf Wert `x`: `a[i]=x;`
  - Gib mir das *j*-te Element von `a`: `a[j];`
  - Gib mir die Anzahl der Elemente in `a`: `a.length`

Arrays vs. Listen

- Wie wir gesehen haben, erlauben Arrays effizient den sogenannten "wahlfreien Zugriff", d.h. wir können auf ein beliebiges Element in $O(1)$ zugreifen.
- Bei der Spezifikation von Folgen aus Kapitel 2+3 (und entsprechenden Listen-Implementierungen) gilt das nicht. Der Zugriff auf das n-te Element erfordert einen Aufwand in $O(n)$.
  Zur Erinnerung: Definition der Projektion

$$\pi(x,i) = \begin{cases} first(x), & \text{falls } i = 1, \\ \pi(rest(x), i-1) & \text{sonst.} \end{cases}$$

- Dafür können Folgen (Listen) beliebig wachsen, während wir die Größe eines Arrays von vornherein festlegen müssen.
- Betrachten wir nun im folgenden die Implementierung von Listen etwas genauer, wozu wir uns jetzt natürlich objektorientierte Implementierungen in Java vornehmen wollen.

Überblick

Erinnerung: Folgen

- Konkatenation einer Folge $x \in M^*$ an ein Element $a \in M$:

$$prefix : M \times M^* \to M^*$$

$$\text{mit } prefix(a, x) = (a) \circ x$$

- Induktive Definition von M^*:
  1. $() \in M^*$
  2. Ist $a \in M$ und $x \in M^*$, dann ist $prefix(a, x) \in M^*$.
- Zugriff auf das erste Element:

$$first : M^+ \to M \text{ mit } first(prefix(a, x)) = a$$

- Liste nach Entfernen des ersten Elementes:

$$rest : M^+ \to M^* \text{ mit } rest(prefix(a, x)) = x$$

Einfaches Listenmodell

Erstellen wir zunächst eine einfache Listenimplementierung analog zu dem Verhalten von Folgen wie in Kapitel 3 definiert:

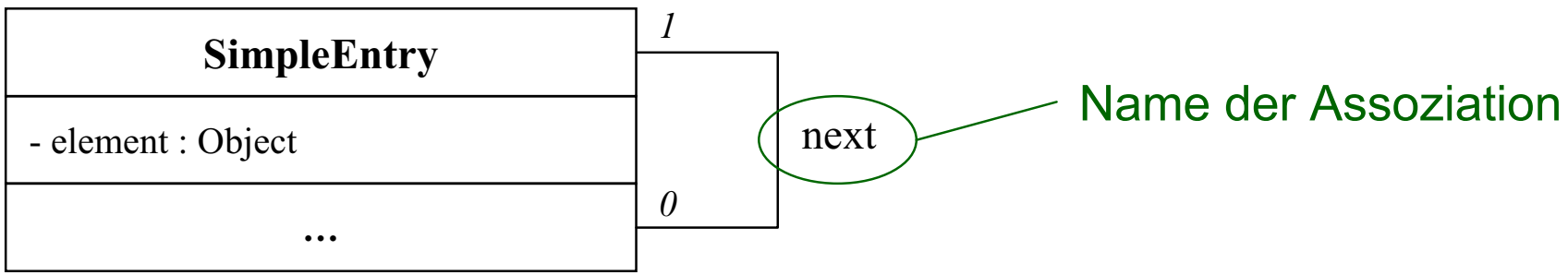
- Eine Liste kann leer sein.
- Ein Element wird vorne an eine Liste angehängt.
- Wir können nur das erste Element einer Liste entfernen.
- Zusätzlich: Eine Liste kann Auskunft über ihre Länge (= Anzahl der Elemente) geben.
  Die Länge einer Folge ist rekursiv definiert als:

$$size(x) = \begin{cases} 0, & \text{falls } x = (), \\ 1 + size(rest(x)) & \text{sonst.} \end{cases}$$

imperative, objektorientierte Modellierung?

Element der Liste als besonderes Objekt

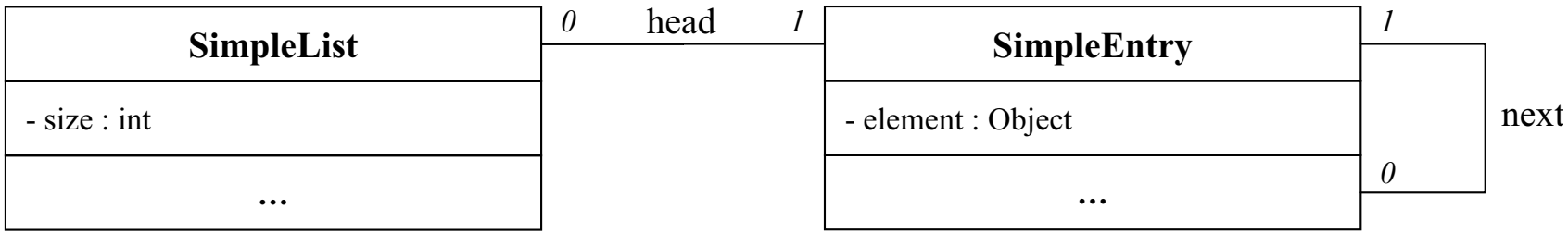
- Von der Objektorientierung herkommend, können wir ein Element der Liste zunächst als eigenständiges Objekt ansehen.
- Dieses Objekt hält das eigentlich gespeicherte Element.
- Andererseits hält das Element-Objekt einen Verweis auf das nächste Element, den Nachfolger.



Element der Liste als besonderes Objekt

```
public class SimpleEntry
{
  private Object element;
  private SimpleEntry next;

  public SimpleEntry(Object o, SimpleEntry next)
  {
    this.element = o;
    this.next = next;
  }

  public Object getElement()
  {
    return this.element;
  }

  public SimpleEntry getNext()
  {
    return this.next;
  }

  public void setNext(SimpleEntry next)
  {
    this.next = next;
  }
}
```

Liste und Elemente (UML)

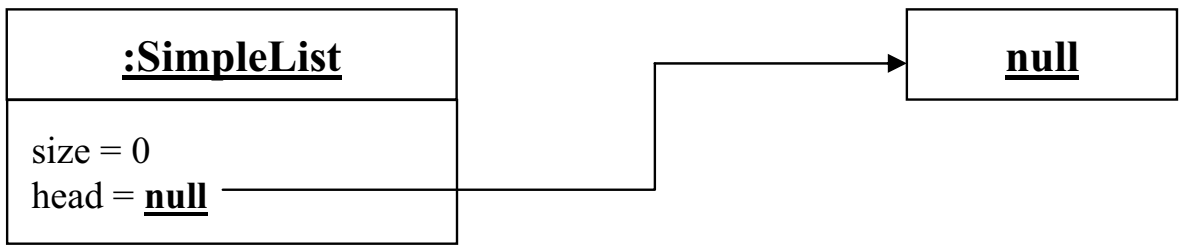


- Ermöglicht Zugriff auf das erste Element.
- Wie kommt man an den Rest der Liste?
- Wie sieht die leere Liste aus?

Liste hält Verweis auf erstes Element

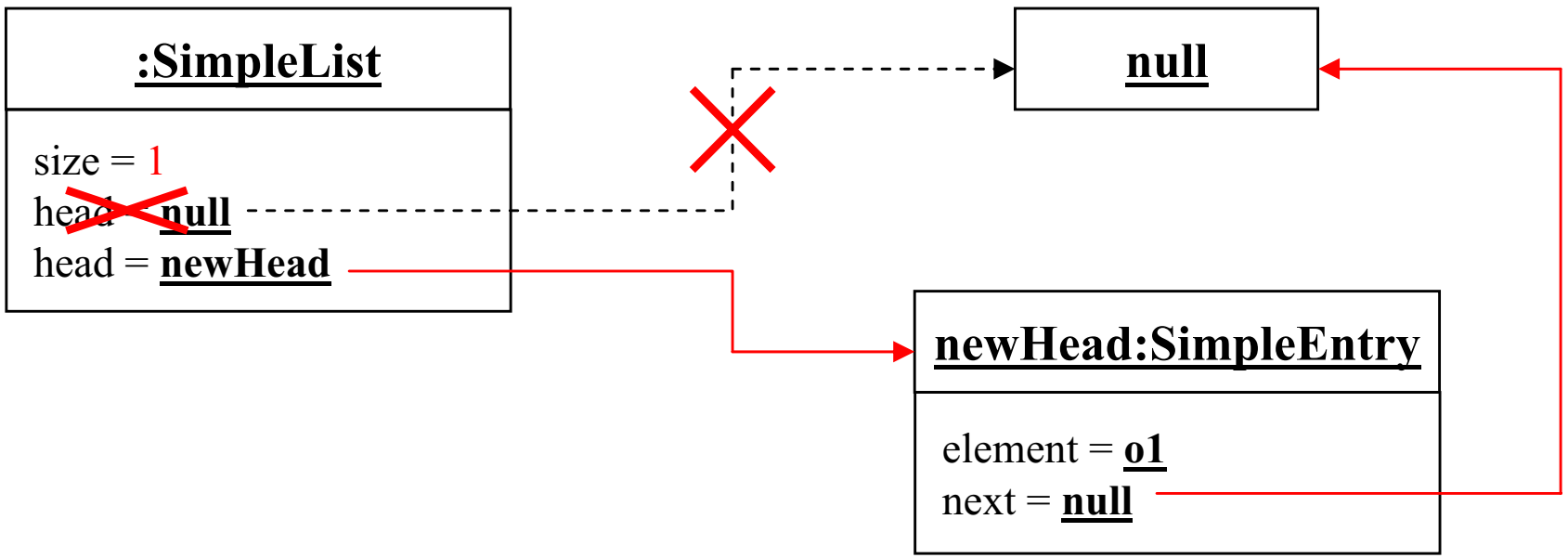
- Die Liste muss nur das erste Element kennen, über dessen Zeiger zum nächsten Element können alle Nachfolger erreicht werden.
- In der leeren Liste ist das erste Element **`null`**.

```
public class SimpleList
{
  private int size;

  private SimpleEntry head;

  public SimpleList()
  {
    this.size = 0;
    this.head = null;
  }

  public int length()
  {
    return this.size;
  }

  ...

}
```

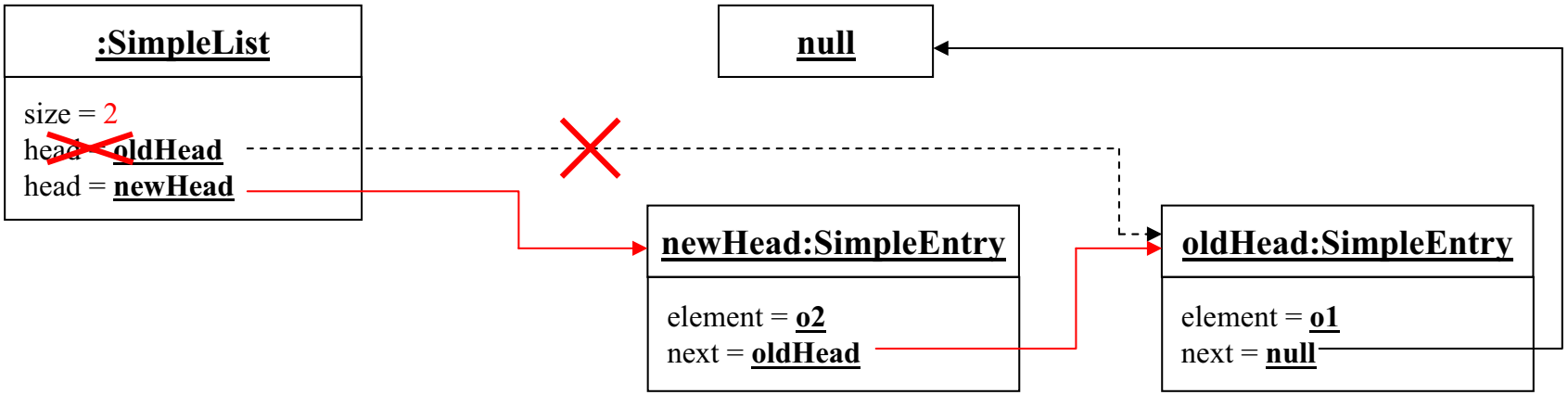


Liste: Hinzufügen eines Elementes

- Um ein neues Element hizuzufügen, wird ein neues `SimpleEntry`-Element erzeugt und als neues erstes Element gesetzt.
- Dessen Nachfolger ist das alte erste Element.
- Die Länge der Liste erhöht sich um 1.



Liste: Hinzufügen eines Elementes




```
...
public void add(Object o)
{
  SimpleEntry newHead = new SimpleEntry(o, this.head);
  this.head = newHead;
  this.size++;
}
...
```

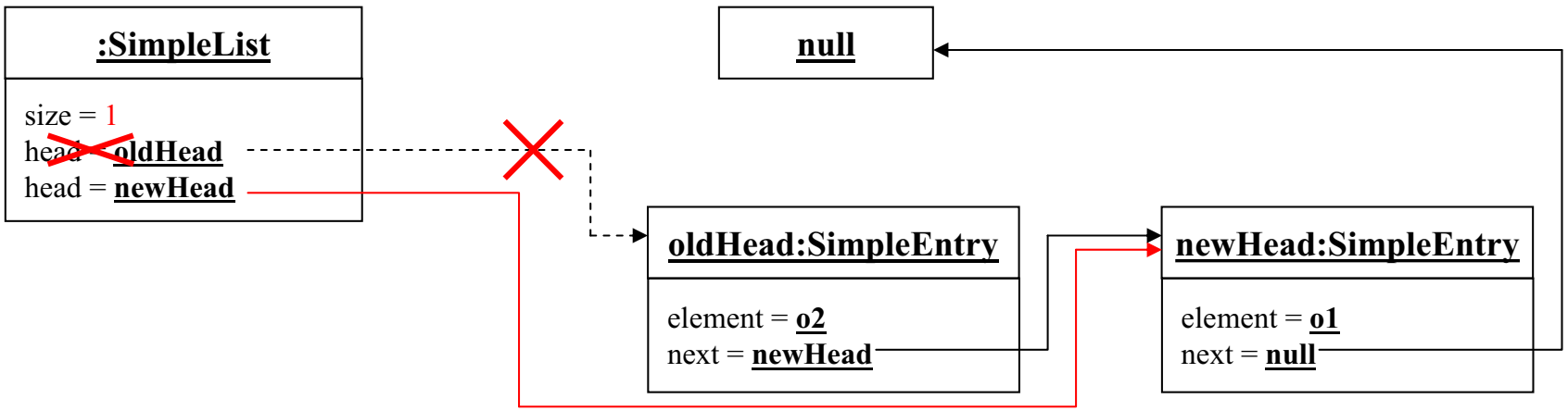
Liste: Wert des ersten Elementes

- Nur den Wert des ersten Elementes zu bekommen, ist einfach.
- Probleme können bei der leeren Liste auftreten.

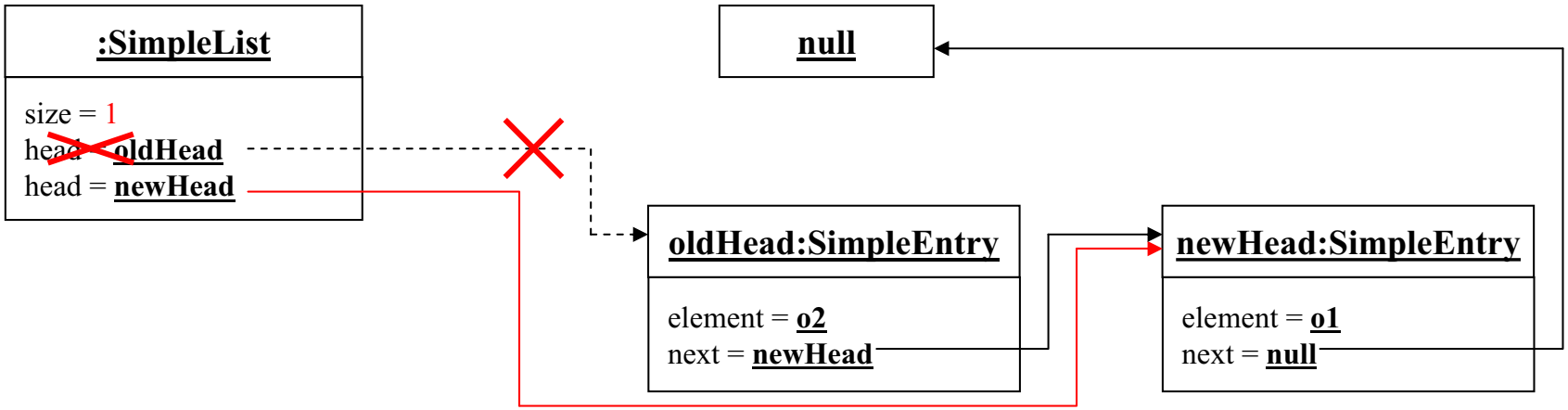
```
...

public Object head()
{
  if(this.head==null)
  {
    throw new NullPointerException("Empty List - no head element available.");
  }
  return this.head.getElement();
}

...
```

Liste: Entfernen des ersten Elementes

- Um das erste Element zu entfernen, muss der Head-Zeiger der Liste auf den Nachfolger des ersten Elementes zeigen.
- Die Länge der Liste wird erniedrigt.
- Danach gibt es keine Zugriffsmöglichkeit mehr für das erste Element.
- Wiederum können Probleme bei der leeren Liste auftreten.



Liste: Entfernen des ersten Elementes




```
public void removeHead()
{
  if(this.head==null)
  {
    throw new NullPointerException("Empty List - no head element available.");
  }
  this.head = this.head.getNext();
  this.size--;
}
```

Verbesserung: Kapselung der Wrapper-Klasse für Einträge

- Die Klasse `SimpleEntry` wird niemals außerhalb der Liste gebraucht, sondern dient nur als Wrapper für das eigentliche Element und die Verkettung zum nächsten Element.
- Java bietet eine Möglichkeit, um diese enge Beziehung auszudrücken: `SimpleEntry` kann man als innere Klasse der Liste definieren.

```
public class SimpleList2
{
  private int size;
  ...

  private static class SimpleEntry
  {
    private Object element;
    ...
  }
}
```

- Innere Klassen sollte man nur sehr zurückhaltend verwenden, aber in diesem Fall ist die Verwendung sinnvoll.

Problem: Unbekannter Typ der Listen-Elemente

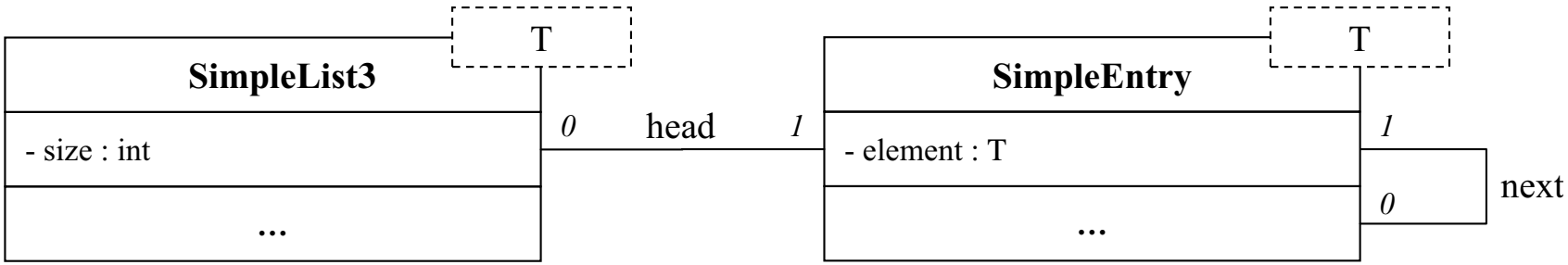
- Bisher können wir jedes beliebige Objekt in unserer Liste ablegen.

```
public class SimpleEntry {
  private Object element;
  private SimpleEntry next;

  public SimpleEntry(Object o, SimpleEntry next){
    this.element = o;
    this.next = next;
  }

  public Object getElement(){
    return this.element;
  }
  ...
}
```

- Was ist hieran problematisch?

Verbesserung 2: Typisierung der Liste

- Durch die Verwendung von `Object` als Typ des Listeneintrags sind in der Verwendung dem Auftreten von Typfehlern Tür und Tor geöffnet.
- Wir kennen schon eine Möglichkeit, die Liste typsicher zu machen: Typisierung der Klasse.



Verbesserung 2: Typisierung der Liste

```
public class SimpleList3<T>
{
  private int size;

  private SimpleEntry<T> head;

  public SimpleList3()
  {
    this.size = 0;
    this.head = null;
  }

  public int length()
  {
    return this.size;
  }
  ...
```

Verbesserung 2: Typisierung der Liste

```
...
public void add(T o)
{
  SimpleEntry<T> newHead = new SimpleEntry<T>(o, this.head);
  this.head = newHead;
  this.size++;
}

public T head()
{
  if(this.head==null)
  {
    throw new NullPointerException("Empty List - no head element available.");
  }
  return this.head.getElement();
}

public void removeHead()
{
  if(this.head==null)
  {
    throw new NullPointerException("Empty List - no head element available.");
  }
  this.head = this.head.getNext();
  this.size--;
}
...
```

Verbesserung 2: Typisierung der Liste

```
...
private static class SimpleEntry<T>
{
  private T element;

  private SimpleEntry<T> next;

  public SimpleEntry(T o, SimpleEntry<T> next)
  {
    this.element = o;
    this.next = next;
  }
  ...
```

Verbesserung 2: Typisierung der Liste

```
    ...
    public T getElement()
    {
      return this.element;
    }

    public SimpleEntry<T> getNext()
    {
      return this.next;
    }

    public void setNext(SimpleEntry<T> next)
    {
      this.next = next;
    }
  }
}
```

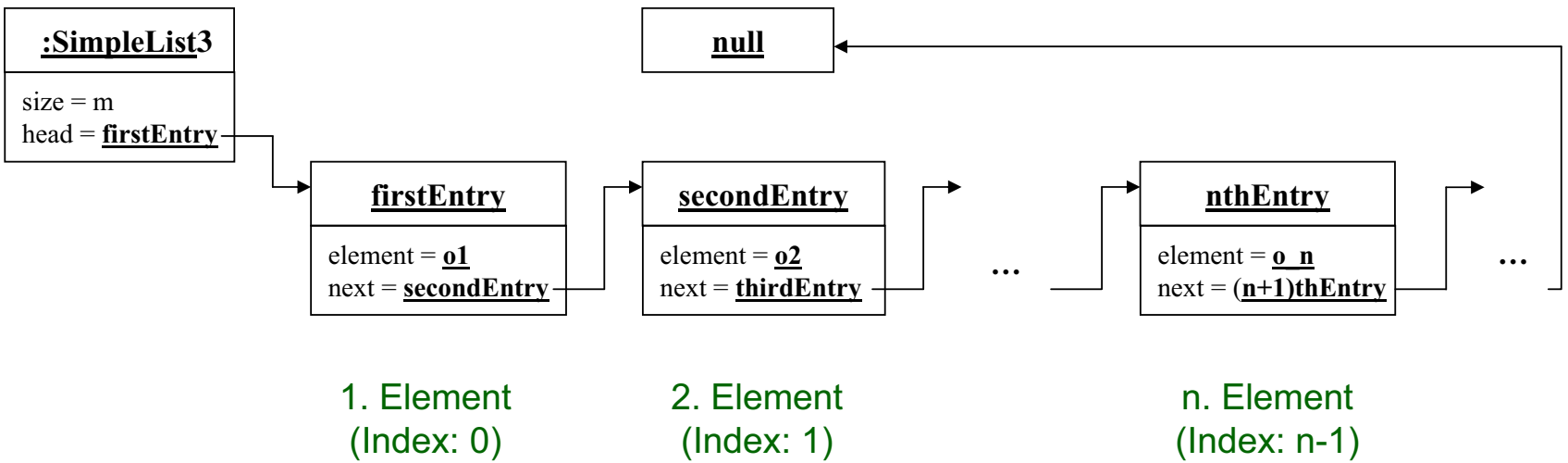
Zugriff auf das *n*-te Element?

Erweiterungswünsche für die Datenstruktur Liste:

- Wie können wir auf das *n*-te Element der Liste zugreifen?
- Wie können wir das *n*-te Element aus der Liste entfernen?
- Wie können wir ein Element an der *n*-ten Stelle in die Liste einfügen?

n-tes Element (mit Index $n - 1$)

Um das *n*-te Element einer Liste zu bekommen, müssen die ersten $n - 1$ Elemente durchlaufen werden:

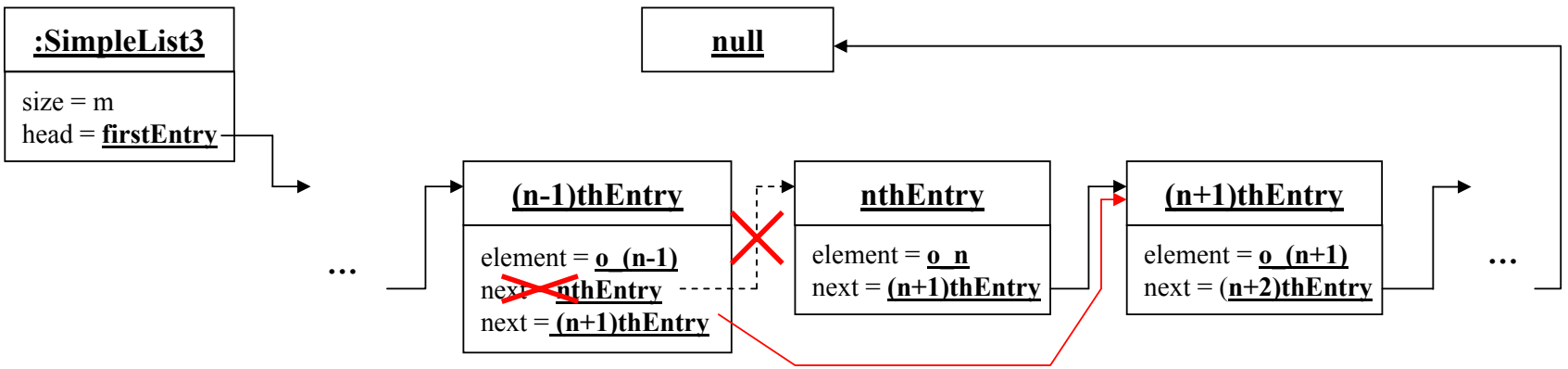


n-tes Element (mit Index $n - 1$)

```
...
public T get(int index)
{
  if(this.head==null)
  {
    throw new NullPointerException("Empty List.");
  }
  if(index>=this.length())
  {
    throw new IllegalArgumentException(index+" exceeds length of list.");
  }
  SimpleEntry<T> currentEntry = this.head;
  while(index>0)
  {
    currentEntry = currentEntry.getNext();
    index--;
  }
  return currentEntry.getElement();
}
...
```

n-tes Element entfernen

- Um das *n*-te Element aus einer Liste zu entfernen, müssen ebenfalls die ersten $n - 1$ Elemente durchlaufen werden.
- Dann muss der Verweis des $n - 1$-ten-Elementes auf den neuen Nachfolger “umgebogen” werden:

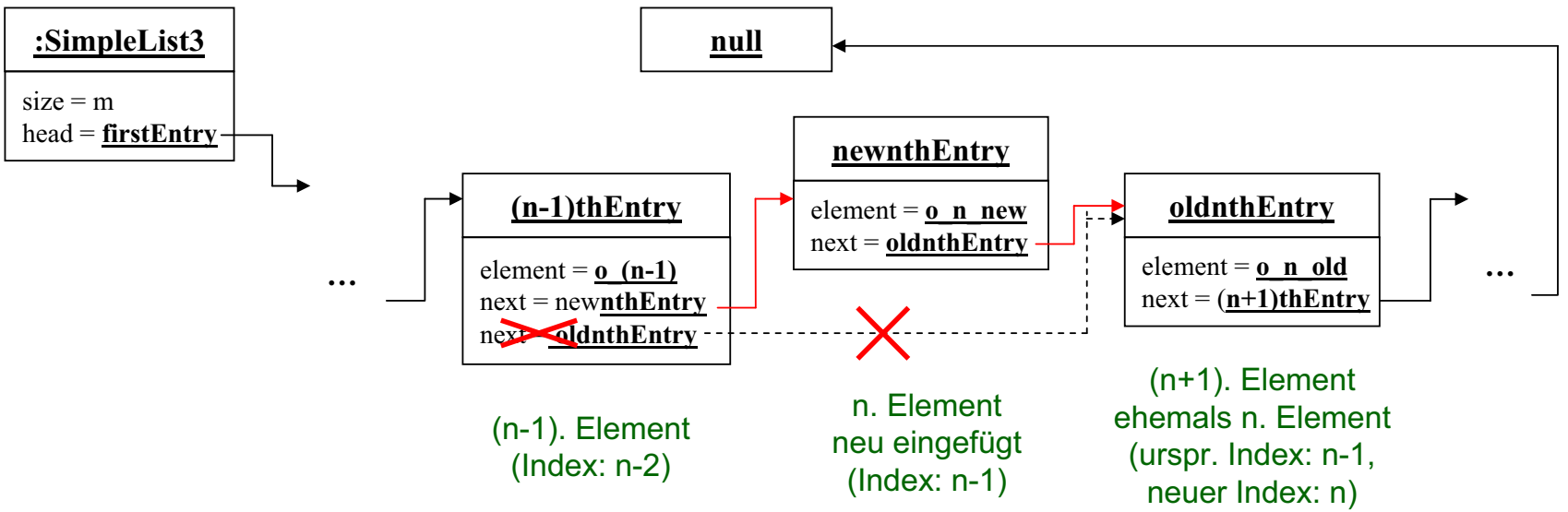


n-tes Element entfernen

```
...
public void delete(int index)
{
  if(this.head==null)
  {
    throw new NullPointerException("Empty List.");
  }
  if(index>=this.length())
  {
    throw new IllegalArgumentException(index+" exceeds length of list.");
  }
  SimpleEntry<T> currentEntry = this.head;
  while(index>1)
  {
    currentEntry = currentEntry.getNext();
    index--;
  }
  currentEntry.setNext(currentEntry.getNext().getNext());
  this.size--;
}
...
```

Element an Stelle *n* einfügen

- Um ein Element an einer bestimmten Stelle *n* einzufügen, müssen wiederum die ersten $n - 1$ Elemente durchlaufen werden.
- Dann muss der Verweis des $n - 1$-Elementes auf den neuen Nachfolger "umgebogen" werden.
- Zuvor brauchen wir aber den alten Verweis, weil der neue Nachfolger des $n - 1$-ten Elements als Nachfolger den alten Nachfolger des $n - 1$-ten Elements haben muss.



Element an Stelle *n* einfügen

```
...
public void insert(T o, int index)
{
  if(index > this.length())
  {
    throw new IllegalArgumentException(index+" exceeds length of list.");
  }
  if(this.head==null)
  {
    this.head = new SimpleEntry<T>(o,null);
  }
  else
  {
    SimpleEntry<T> currentEntry = this.head;
    while(index > 1)
    {
      currentEntry = currentEntry.getNext();
      index--;
    }
    SimpleEntry<T> newEntry = new SimpleEntry<T>(o,currentEntry.getNext());
    currentEntry.setNext(newEntry);
  }
  this.size++;
}
...
```

Code-Fragment

Was passiert in diesem Code-Fragment?

```
...
  SimpleEntry<T> currentEntry = this.head;
  while(currentEntry!=null && !currentEntry.getElement().equals(o))
  {
    currentEntry = currentEntry.getNext();
  }
  return currentEntry!=null;
...
```

Suche nach einem Element

Häufige Anforderung: Stelle fest, ob die Liste ein Objekt bestimmter Art enthält.

```
...
public boolean contains(T o)
{
  SimpleEntry<T> currentEntry = this.head;
  while(currentEntry!=null && !currentEntry.getElement().equals(o))
  {
    currentEntry = currentEntry.getNext();
  }
  return currentEntry!=null;
}
...
```

Darstellung der Liste

Um sich den aktuellen Zustand der Liste anzuschauen, überschreibt man am besten die `toString`-Methode in geeigneter Weise.

```
...
public String toString()
{
  StringBuilder builder = new StringBuilder();
  builder.append("[");
  for(SimpleEntry<T> entry = this.head; entry!=null; entry=entry.getNext())
  {
    builder.append(entry.getElement().toString());
    if(entry.getNext()!=null)
    {
      builder.append(", ");
    }
  }
  builder.append("]");
  return builder.toString();
}
...
```

Liste in ein Array umwandeln

Wenn sich die Länge einer Liste nicht mehr ändert, ist es oft praktischer, mit einem Array weiterzuarbeiten. Warum?

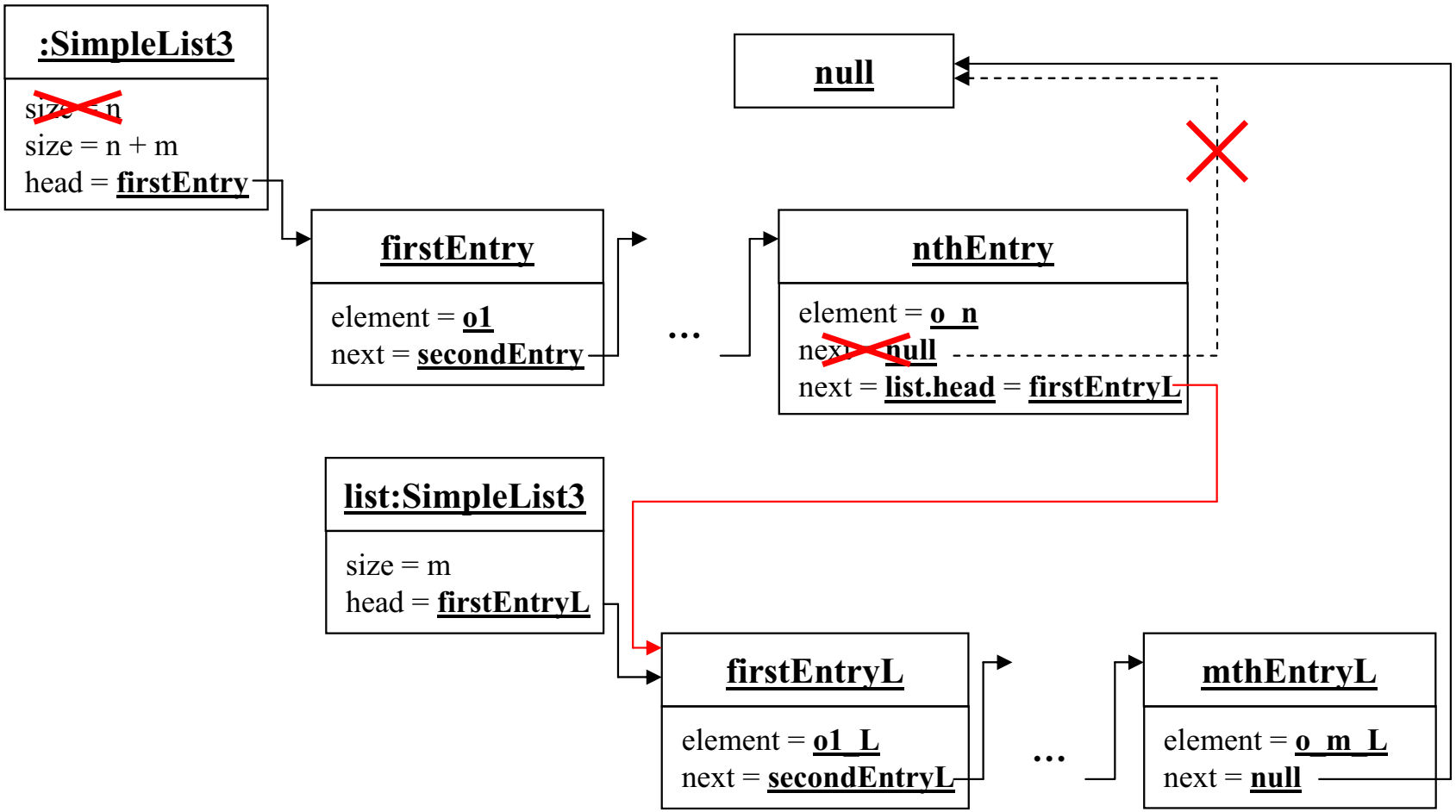
```
...
public T[] asArray()
{
  // hier wird sich der Compiler besorgt zeigen:
  // Type safety: The cast from Object[] to T[]
  // is actually checking against the erased type Object[]
  // Erinnerung: wir koennen keine generisch typisierten Arrays erzeugen
  T[] array = (T[]) new Object[this.size];
  int index = 0;
  for(SimpleEntry<T> entry = this.head; entry!=null; entry=entry.getNext())
  {
    array[index++] = entry.getElement();
  }
  return array;
}
...
```

Append: Eine Liste anhängen

- Die Methode append soll eine andere Liste an diese Liste anhängen.
- Wie kann diese Funktionalität implementiert werden?

Append: Eine Liste anhängen

- Das letzte Element dieser Liste soll nicht mehr auf **`null`** verweisen, sondern auf das erste Element der anderen Liste:



Append: Eine Liste anhängen

```
public void append(SimpleList3<T> list)
{
  SimpleEntry<T> last = this.head;
  while(last.getNext()!=null)
  {
    last = last.getNext();
  }
  last.setNext(list.head);
  this.size += list.length();
}
```

Zeitbedarf für append?

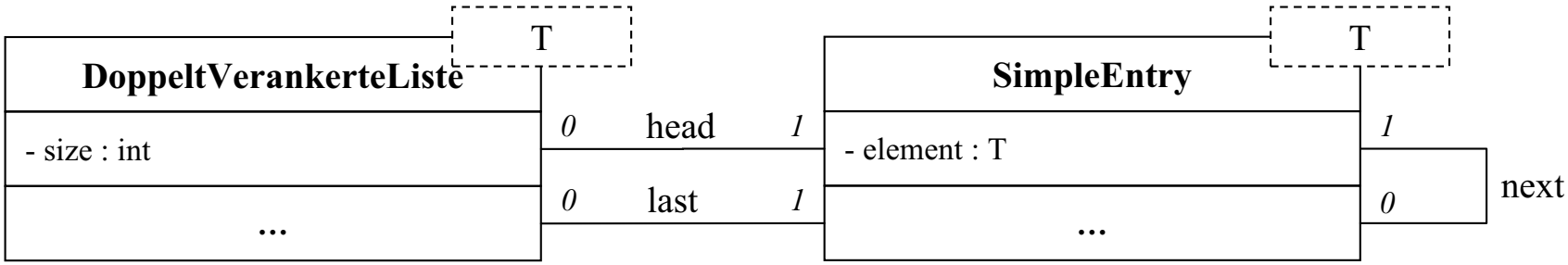
Append: Eine Liste anhängen

```
public void append(SimpleList3<T> list)
{
  SimpleEntry<T> last = this.head;
  while(last.getNext()!=null)
  {
    last = last.getNext();
  }
  last.setNext(list.head);
  this.size += list.length();
}
```

- In der bisherigen Implementierung muss man für die append-Methode die erste Liste ganz durchwandern ($O(n)$).
- Man kann die Liste aber auch so implementieren, dass sich der Zeitbedarf für append auf $O(1)$ reduziert.

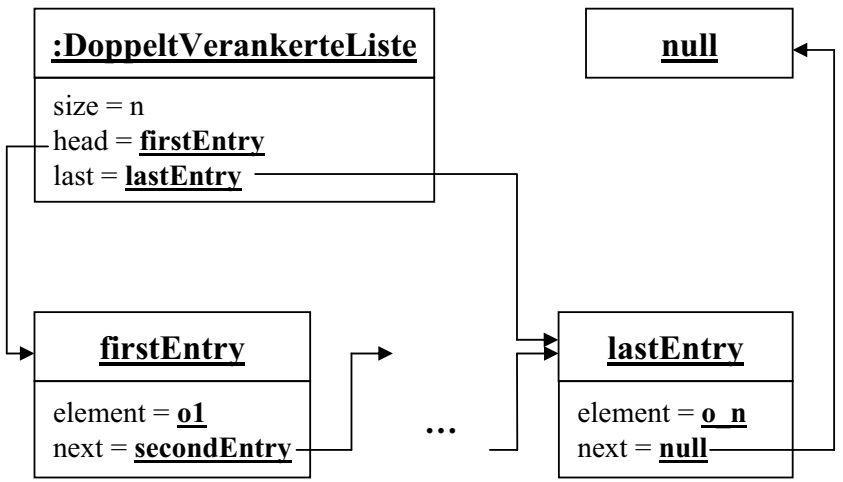
Effizienzverbesserung: Doppelt-verankerte Liste

- Lösung: halte den Verweis auf das letzte Element als Attribut der Liste:

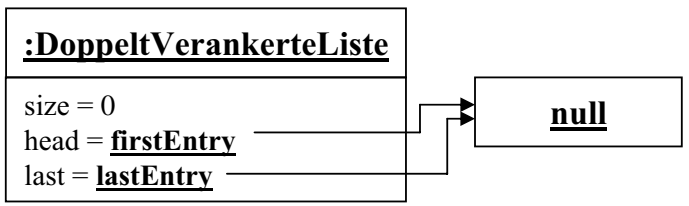


Effizienzverbesserung: Doppelt-verankerte Liste

Allgemeines Schema:



Nach Instanziierung:

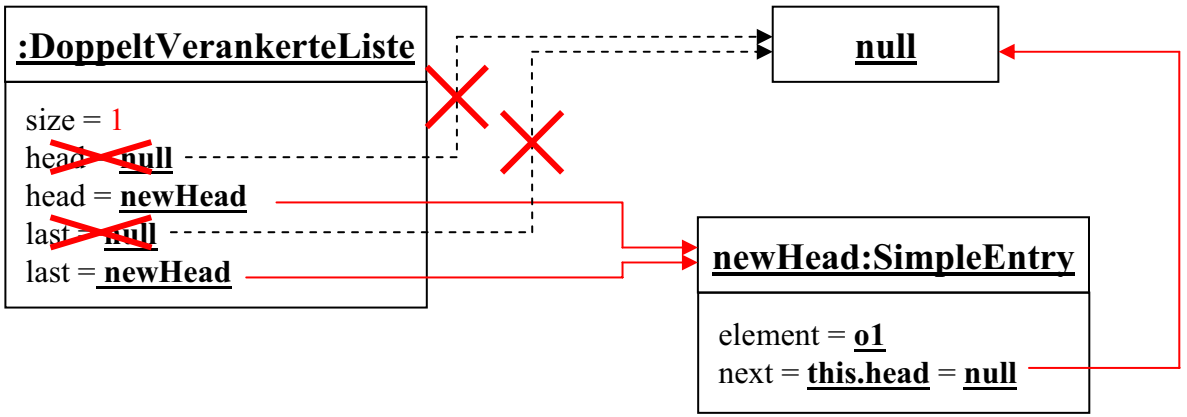


Effizienzverbesserung: Doppelt-verankerte Liste

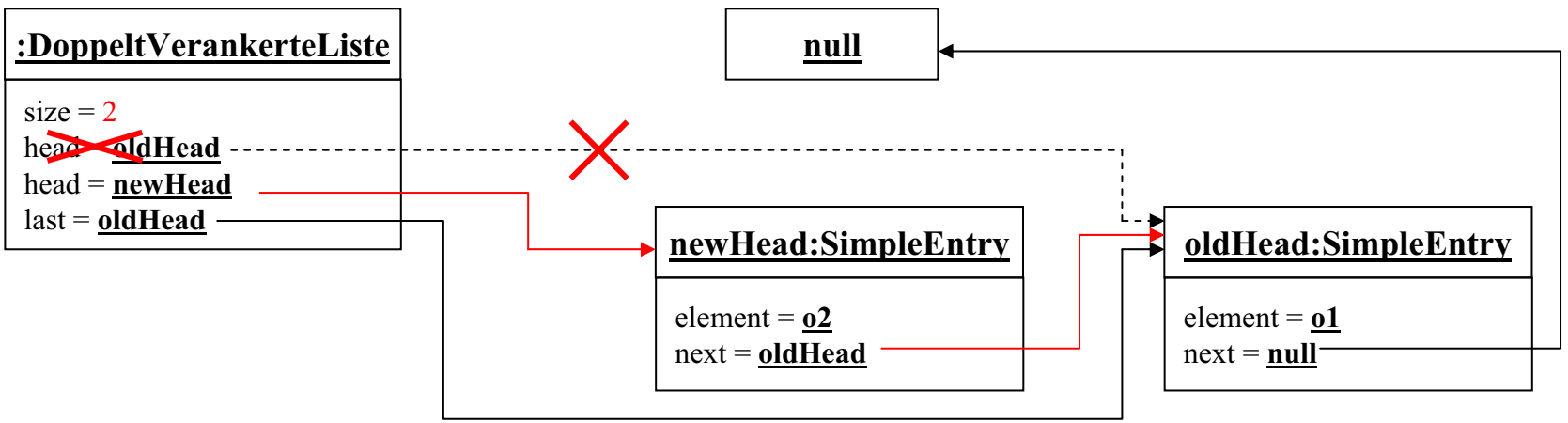
```
public class DoppeltVerankerteListe<T>
{
  private int size;

  private SimpleEntry<T> head;

  private SimpleEntry<T> last;

  public DoppeltVerankerteListe()
  {
    this.size = 0;
    this.head = null;
    this.last = null;
  }
  ...
```

Effizienzverbesserung: Doppelt-verankerte Liste

Erstes Einfügen:



Zweites Einfügen:

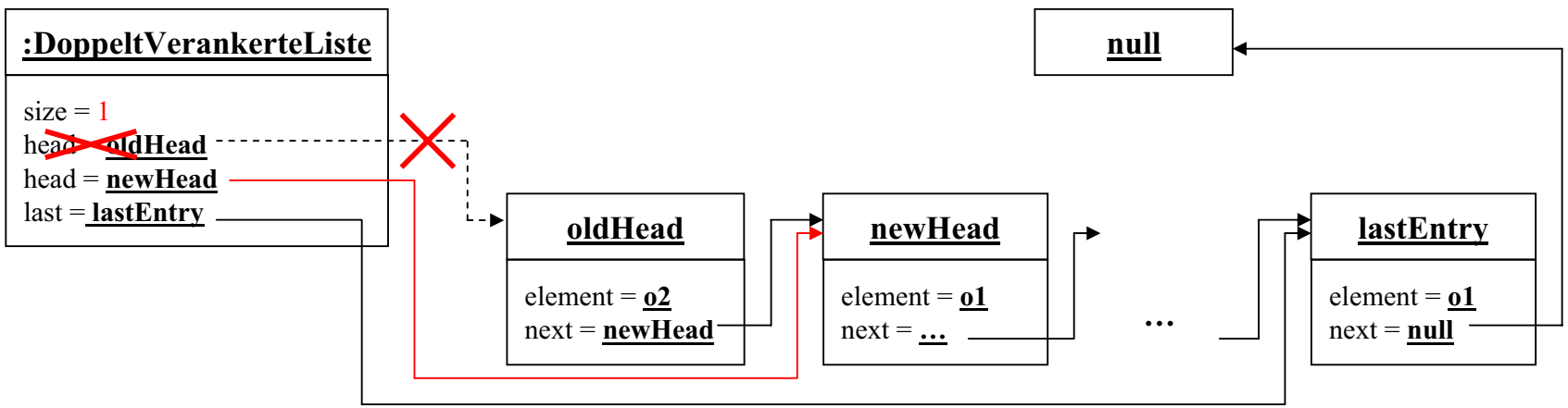


Effizienzverbesserung: Doppelt-verankerte Liste

```
public void add(T o)
{
  SimpleEntry<T> newHead = new SimpleEntry<T>(o, this.head);
  this.head = newHead;
  if(this.last==null)
  {
    this.last = newHead;
  }
  this.size++;
}
```

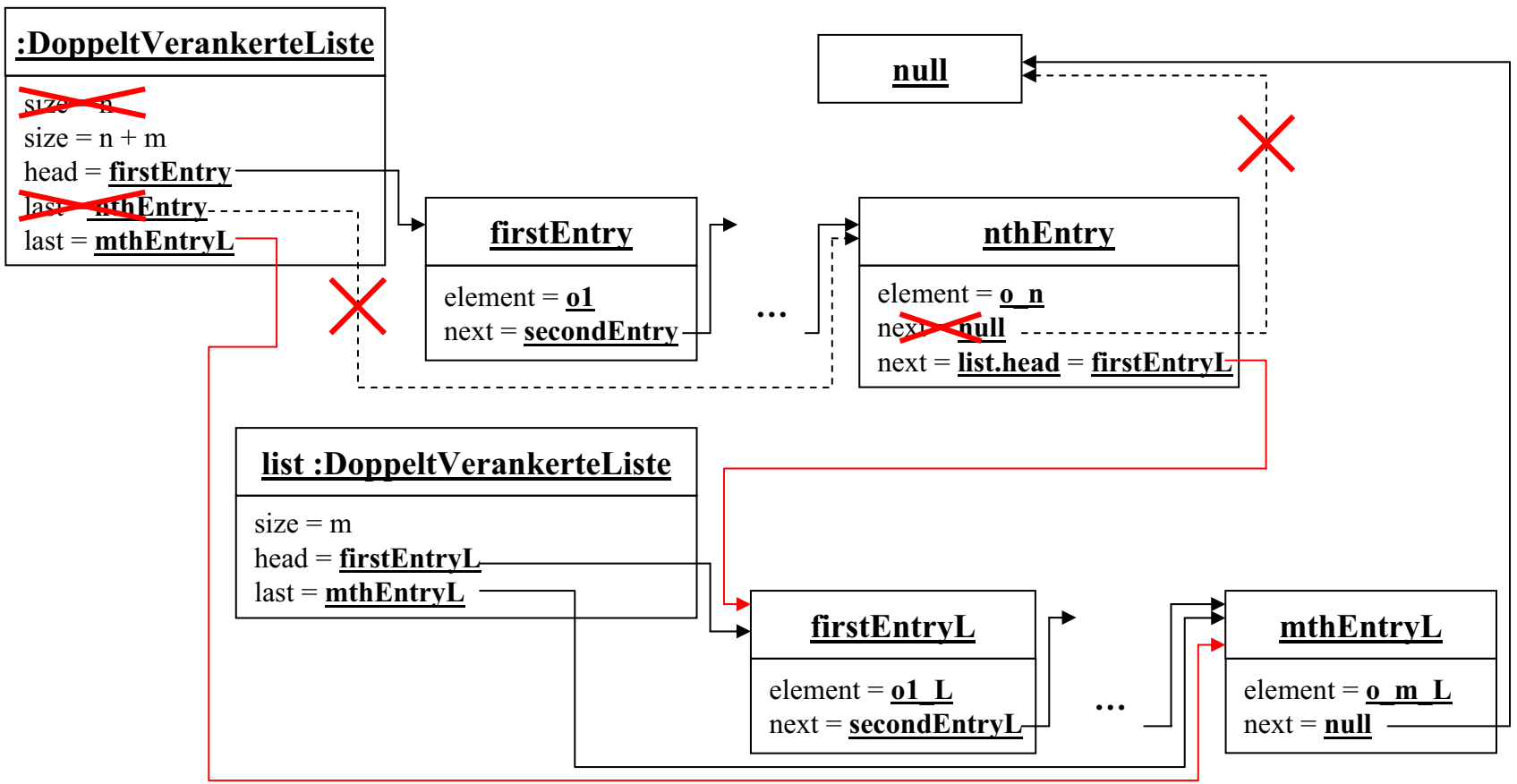
Effizienzverbesserung: Doppelt-verankerte Liste

```
public void removeHead()
{
  if(this.head==null)
  {
    throw new NullPointerException("Empty List - no head element available.");
  }
  this.head = this.head.getNext();
  if(this.head==null)
  {
    this.last = null;
  }
  this.size--;
}
```



Effizienzverbesserung: Doppelt-verankerte Liste

```
public void append(DoppeltVerankerteListe<T> list)
{
  this.last.setNext(list.head);
  this.last = list.last;
  this.size += list.length();
}
```



Effizienzverbesserung: Doppelt-verankerte Liste

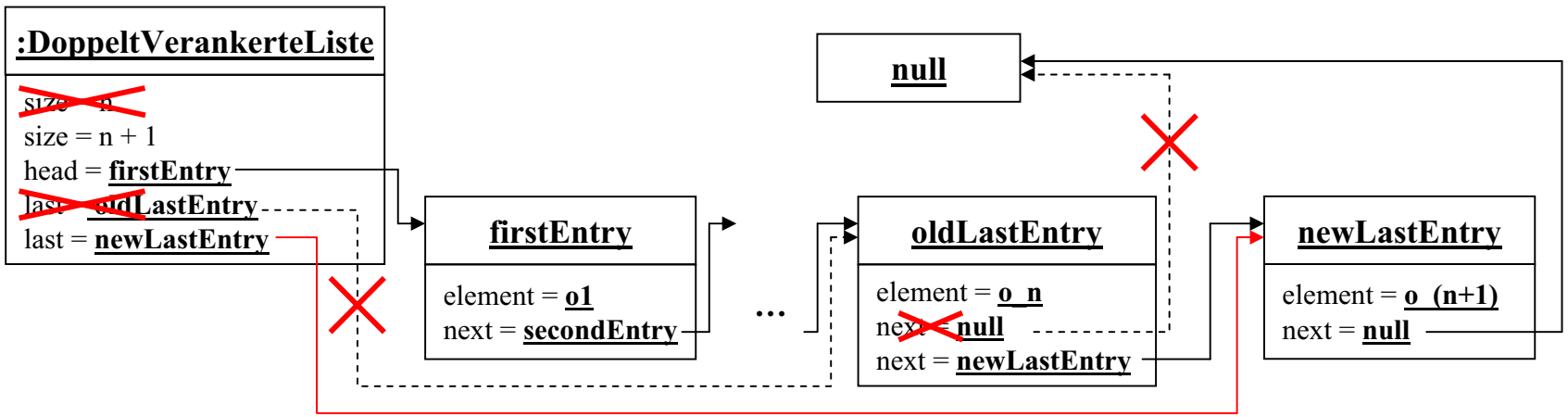
```
public void delete(int index)
{
  if(this.head==null)
  {
    throw new NullPointerException("Empty List.");
  }
  if(index>=this.length())
  {
    throw new IllegalArgumentException(index+" exceeds length of list.");
  }
  SimpleEntry<T> currentEntry = this.head;
  while(index>1)
  {
    currentEntry = currentEntry.getNext();
    index--;
  }
  if(currentEntry.getNext()==this.last)
  {
    this.last = currentEntry;
    this.last.setNext(null);
  }
  else
  {
    currentEntry.setNext(currentEntry.getNext().getNext());
  }
  this.size--;
}
```

Effizienzverbesserung: Doppelt-verankerte Liste

```
...
public void insert(T o, int index){
  if(index > this.length()){
    throw new IllegalArgumentException(index+" exceeds length of list.");
  }
  if(this.head==null){
    this.head = new SimpleEntry<T>(o,null);
    this.last = this.head;
  }
  else{
    SimpleEntry<T> currentEntry = this.head;
    while(index>1){
      currentEntry = currentEntry.getNext();
      index--;
    }
    SimpleEntry<T> newEntry = new SimpleEntry<T>(o,currentEntry.getNext());
    if(currentEntry==this.last){
      this.last=newEntry;
    }
    currentEntry.setNext(newEntry);
  }
  this.size++;
}
...
```

Einfügen am Ende der Liste

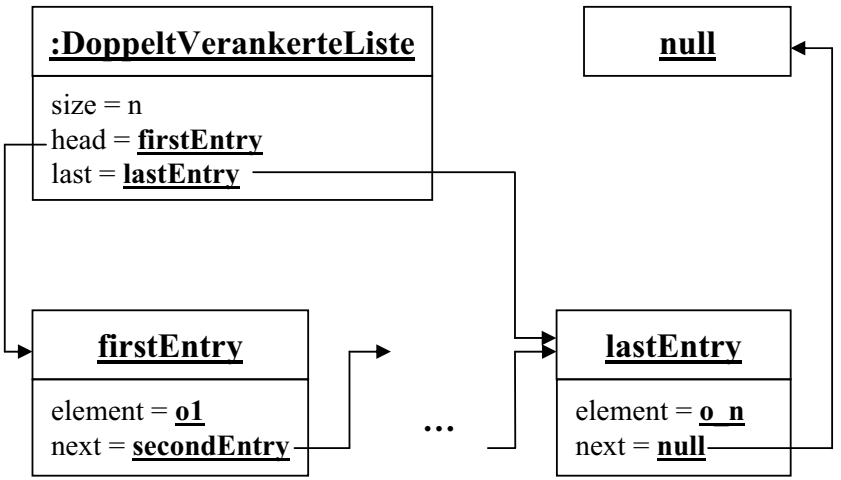
- Damit haben wir gleichzeitig eine effiziente Möglichkeit gewonnen, ein Element am Ende der Liste einzufügen.



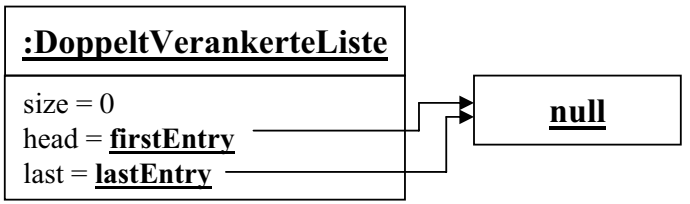

```
public void append(T o)
{
  SimpleEntry<T> newLast = new SimpleEntry<T>(o, null);
  this.last.setNext(newLast);
  this.last = newLast;
  this.size++;
}
```

Entfernen am Ende der Liste

Allgemeines Schema:



Nach Instanziierung:



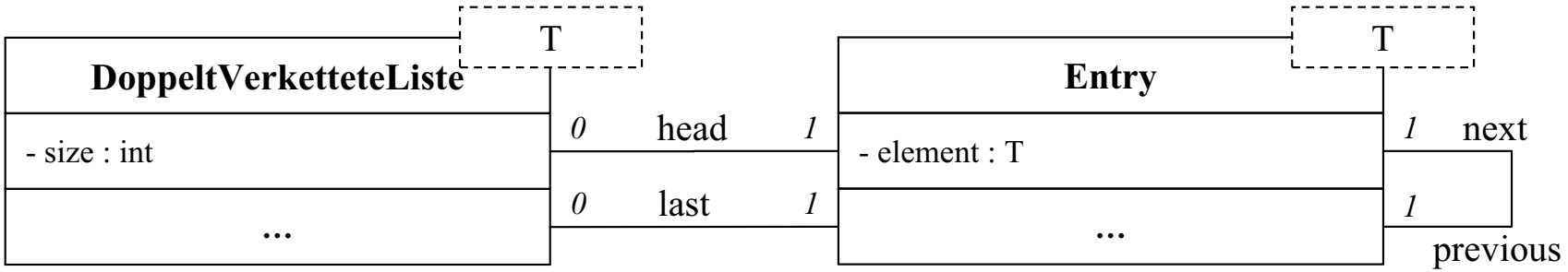
- Ist auch das Entfernen am Ende der Liste möglich?

Entfernen am Ende der Liste

- Der Zeiger für das letzte Element muss auf das vorletzte Element “umgebogen” werden.
- In der bisherigen Implementierung müssten wir dazu wieder von vorne die Liste durchlaufen.
- Lösung: Doppelt-*verkettete* Liste.

Doppelt-verkettete Liste

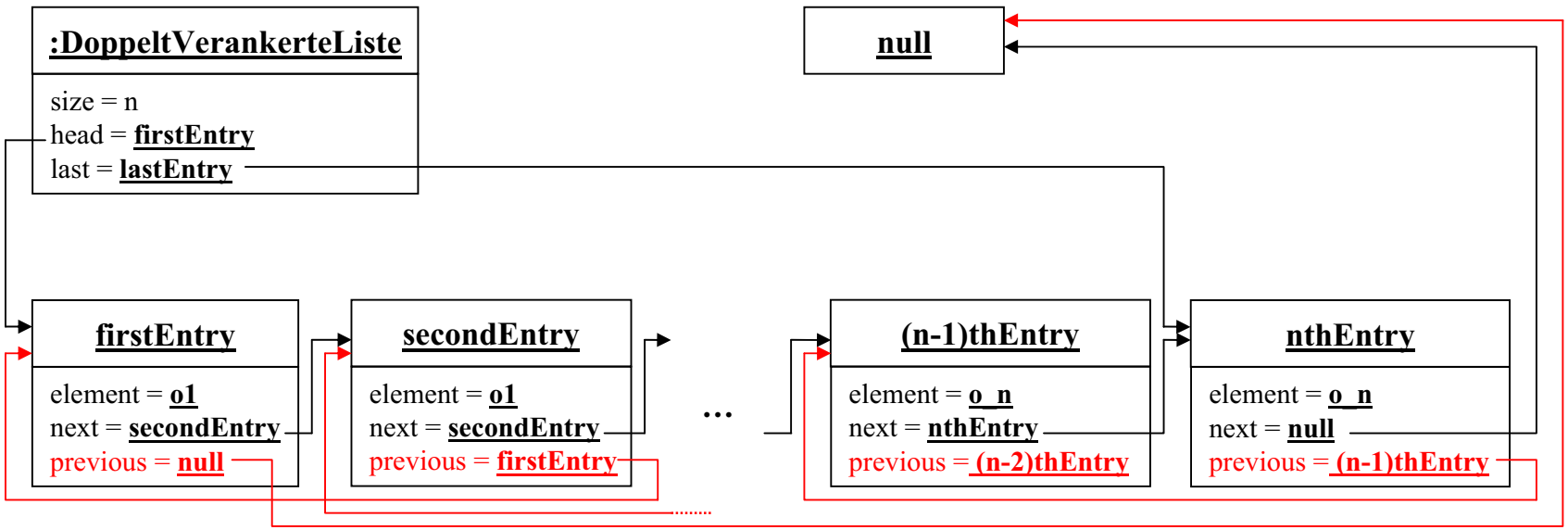
- Jeder Eintrag `Entry` enthält auch einen Verweis auf seinen Vorgänger.
- Dadurch kann man die Liste auch von hinten nach vorne durchlaufen.
- Entfernen des letzten Elementes: $O(1)$.



Doppelt-verkettete Liste

```
...
private static class Entry<T>
{
  private T element;

  private Entry<T> next;

  private Entry<T> previous;

  public Entry(T o, Entry<T> next, Entry<T> previous)
  {
    this.element = o;
    this.next = next;
    this.previous = previous;
  }
  ...
}
```

Doppelt-verkettete Liste



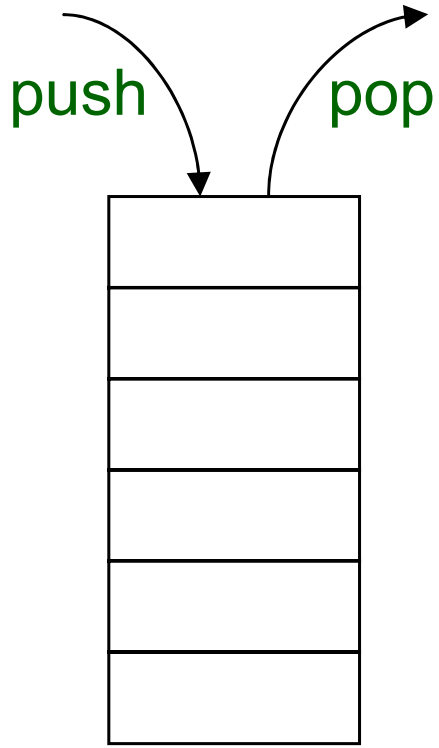
- Zur Übung: Welche Änderungen werden in den bisherigen Methoden dadurch nötig?

Doppelt-verkettete Liste als Standard

- Im imperativen Paradigma wächst eine Liste normalerweise nicht nach vorne, sondern nach hinten.
- Die einfache Anfüge-Operation hängt ein neues Element hinten an.
- Einfügen und Löschen ist auch an beliebiger Stelle möglich.
- Standard-Implementierungen einer verketteten Liste sind oft doppelt-verkettete Listen (z.B. `java.util.LinkedList`).

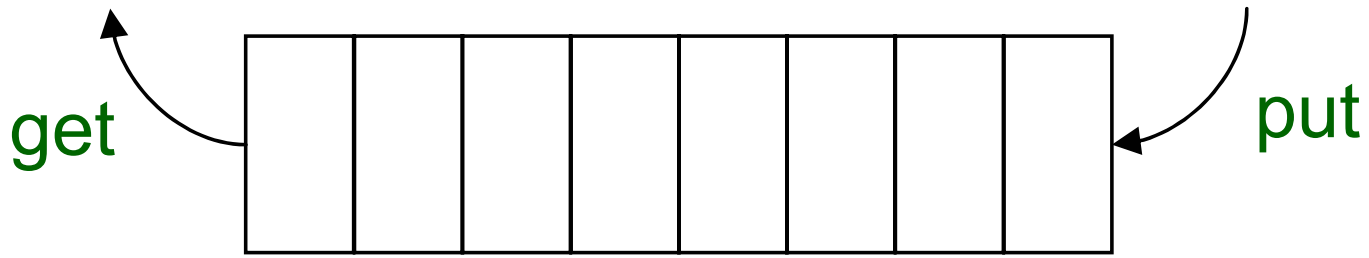
Beispiel: Keller (Stapel, Stack)

- Den Keller als LIFO-Datenstruktur haben wir bereits kennengelernt.
- Wir wollen einen Kellerspeicher mit zwei Operationen:
  - `void push(o)` – legt Objekt `o` auf dem Stapel ab.
  - `E pop()` – entfernt oberstes Element (vom Typ `E`) und gibt es zurück.
- Welche Mittel benötigen wir *minimal* zur Implementierung:
  - Einfach verkettete Liste?
  - Doppelt verankerte Liste?
  - Doppelt verkettete Liste?



Beispiel: Warteschlange (Queue)

- Die Warteschlange (Queue) ist eine häufig benötigte FIFO-Datenstruktur.
- Typischerweise zwei Operationen:
  - `void put(o)` – fügt das Objekt `o` an die Schlange an.
  - `E get()` – entfernt vorderstes Element (vom Typ `E`) und gibt es zurück.



- Welche Mittel sind zur Implementierung am besten geeignet:
  - Einfach verkettete Liste?
  - Doppelt verankerte Liste?
  - Doppelt verkettete Liste?

Flexible Datenstrukturen mit effizientem wahlfreiem Zugriff

- Den Vorteil der flexiblen Länge haben wir in den bisherigen Implementierungen dadurch erkauft, dass der Zugriff auf das n-te Element eine Zeitkomplexität von $O(n)$ hat.
- Andere Implementierungen verwirklichen die flexible Länge durch ein internes Array.
- Wird das interne Array zu kurz, wird es durch ein längeres ersetzt und der Inhalt des alten Arrays in das neue kopiert.
- Beispiele für diese Implementierung kennen Sie bereits mit `StringBuilder` und `StringBuffer`.
- Allgemeine, array-basierte Listen-Implementierungen: `java.util.ArrayList` und `java.util.Vector`.
- Der Vorteil durch zeiteffizienten wahlfreien Zugriff ($O(1)$) wird erkauft durch höheren Speicherplatzbedarf und gelegentlichen Kopieraufwand beim Wachsen der Liste.
- Weitere Vor- und Nachteile der beiden grundsätzlichen Arten der Implementierung (Array-basiert vs. verkettet)?

Überblick

Was ist ein assoziativer Speicher?

- Ein assoziativer Speicher ist eine materialisierte Abbildung, die einen Schlüssel auf einen Wert abbildet.
- Als einfachen assoziativen Speicher haben wir die Arrays kennengelernt: Ein Schlüssel (der Index) wird auf einen Wert (der an der Stelle `index` im Array gespeichert ist) abgebildet:

Beispiel

Eine **`char`**-Reihung `gruss` der Länge 13:

`gruss`:	'H'	'e'	'l'	'l'	'o'	','	' '	'W'	'o'	'r'	'l'	'd'	'!'
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12

$$\texttt{gruss} : \{0, 1, \ldots, 12\} \to \textbf{char}$$

$$i \mapsto \begin{cases} \text{'H'} & \text{falls} \quad i = 0 \\ \text{'e'} & \text{falls} \quad i = 1 \\ \vdots & \\ \text{'!'} & \text{falls} \quad i = 12 \end{cases}$$

Verallgemeinerung

- Allgemein ist ein assoziativer Speicher denkbar als Abbildung aus einer beliebigen Domäne in eine andere (oder auch die gleiche).
- Statt einer Indexmenge ist der Definitionsbereich der Abbildung also irgendeine Domäne, aus der die Schlüssel stammen.
- Beispiel:
  Wörterbuch deutsch – englisch
  - `String` → `String`
  - “hallo” ↦ “hello”

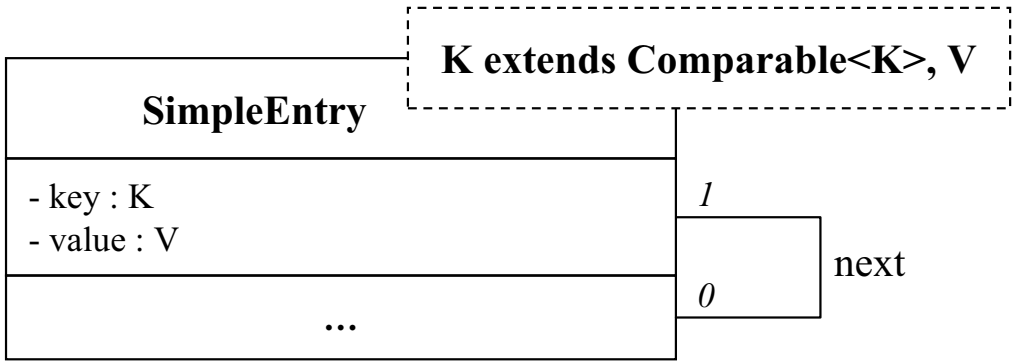
  Platznummer für Passagier auf einem bestimmten Flug
  - `String` → **`int`**
  - “Hans-Peter Kriegel” ↦ 17
- Einen assoziativen Speicher nennt man auch *Map*, *Dictionary* oder *Symboltabelle*.

Computerbasierte assoziative Speicher

- Wörterbücher, Telefonbücher, Lexika und ähnliche gedruckte Nachschlagewerke (also statische assoziative Speicher) unterstützen durch alphabetische Sortierung der Schlüssel (Stichwörter, Namen, ...) effiziente Suche nach einem Eintrag zu einem bestimmten Schlüssel.
- Computerbasierte assoziative Speicher sind dagegen auch dynamisch. Sie können wachsen und schrumpfen (ähnlich wie Listen gegenüber Arrays).
- Im Folgenden betrachten wir grundlegende Ideen, um dieses dynamische Verhalten für das Einfügen neuer Schlüssel und Werte und ihr Löschen effizient zu ermöglichen.

1. Versuch: sortierte lineare Liste

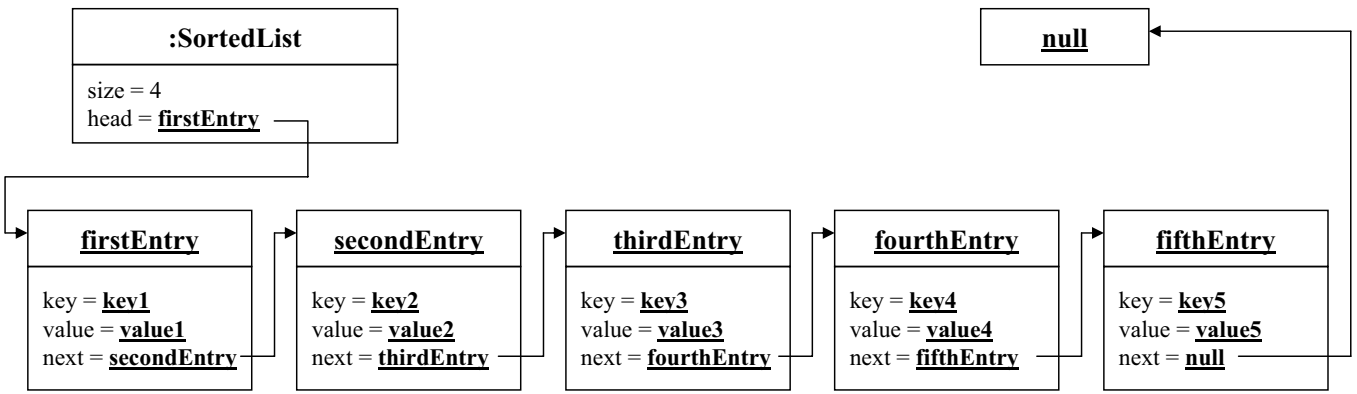
- Als ersten Ansatz versuchen wir einen assoziativen Speicher als *sortierte* lineare Liste zu realisieren.
- Die Klasse `SimpleEntry` muss daher angepasst werden, um nun Schlüssel *und* Eintrag (Wert) zu verwalten.



- Frage: Warum muss der Datentyp `K` für den Schlüssel `key` das Interface `Comparable` implementieren?

1. Versuch: Einfügen/Löschen/Zugriff

- Beim Einfügen wird zunächst die Stelle gesucht, an die der neue Eintrag eingefügt werden muss, um die Sortierung weiterhin zu gewährleisten.
- Dazu muss die Liste vom Beginn an solange durchlaufen werden, bis der aktuelle Eintrag einen Schlüssel besitzt, dessen Wert größer als der Schlüsselwert des neuen Eintrags ist.
- Analog muss beim Löschen der Eintrag gesucht werden, der zum angegebenen Schlüssel passt (durch Suchen vom Beginn der Liste).
- Frage: Wie lässt sich der Zugriff auf den Wert eines bestimmten Schlüssels realisieren?



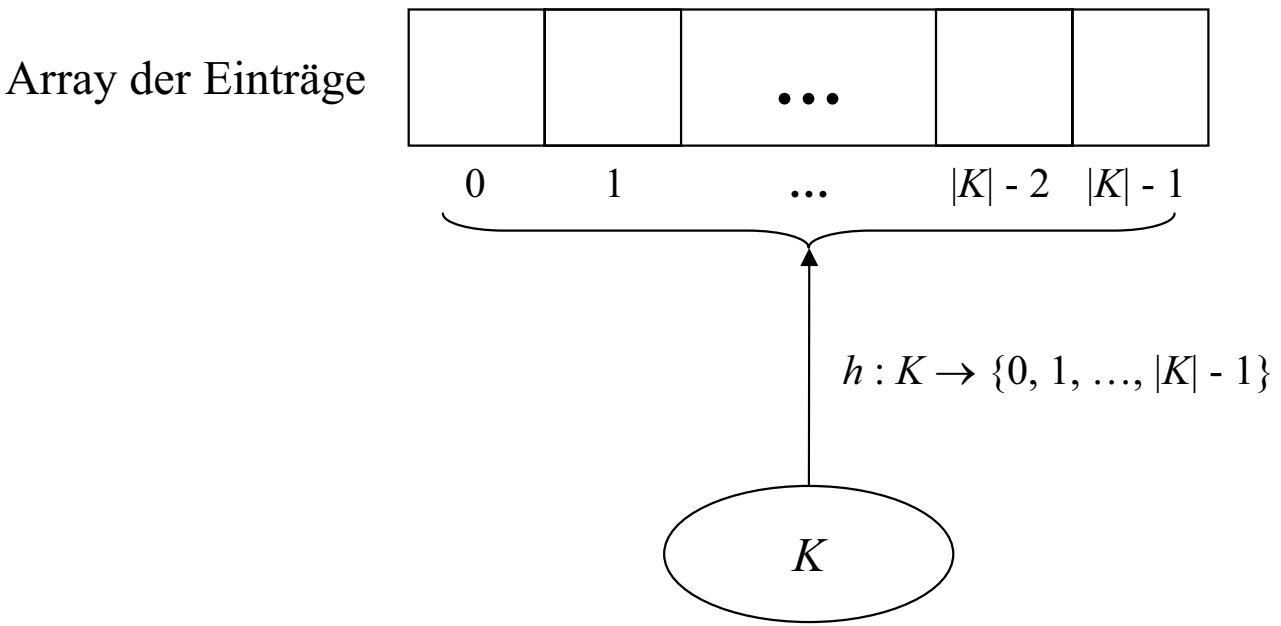
1. Versuch: Analyse

- Effizienzanalyse (worst case):
  - **Einfügen:** Sequentielle Suche nach der richtigen Einfügestelle $\Rightarrow O(n)$.
  - **Löschen:** Sequentielle Suche nach dem zu löschenden Schlüssel $\Rightarrow O(n)$.
  - **Zugriff auf Eintrag mit bestimmten Schlüssel:** Sequentielle Suche nach dem angefragten Schlüssel $\Rightarrow O(n)$.
- Dynamik:
  Die Implementierung mittels sortierter linearer Liste ist dynamisch.
- Fazit:
  Implementierung ist dynamisch, aber nicht wirklich effizient.

2. Versuch: Array

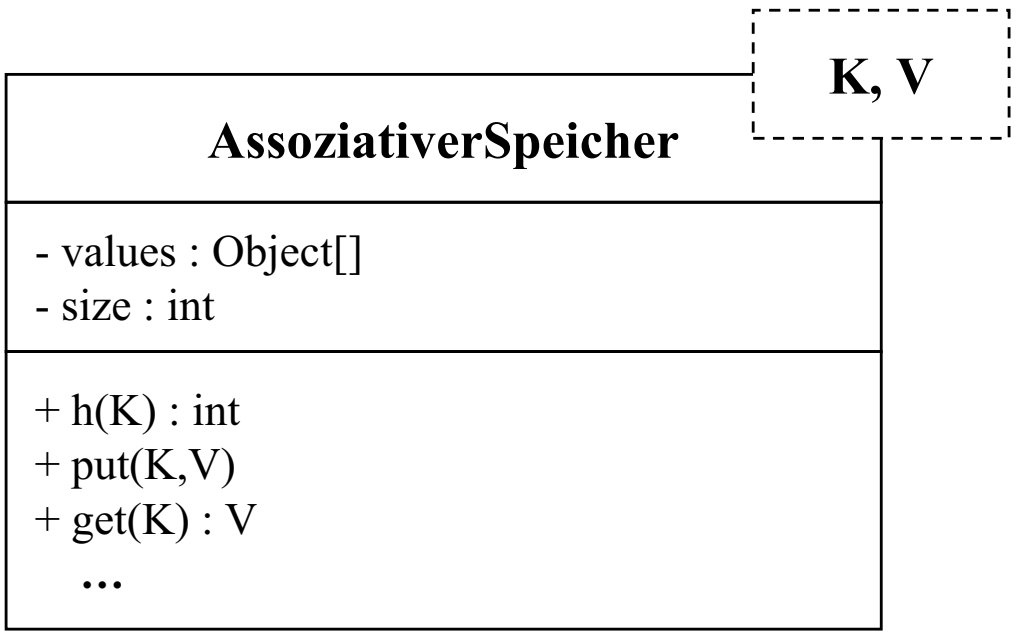
- Idee: Verwende ein Array um die Einträge zu verwalten.
- Wenn wir eine eindeutige Abbildung aller Schlüsselwerte auf einen Array Index definieren können, hätten wir konstanten Zugriff auf die Einträge (im best case)!
- Für die Menge der Schlüssel K benötigen wir also eine Abbildung

  $h : K \rightarrow \{0, \ldots |K|-1\}$ ($|K|$ bezeichnet die Anzahl der Schlüssel in $|K|$).



2. Versuch: Realisierung

- Die Funktion *h* bezeichnet man als *Hashfunktion* oder auch als *Adressfunktion*.
- Assoziative Speicher werden meist mittels einer Hashfunktion realisiert (*Hashverfahren*).
- Verwendete Datenstruktur: Array von Objekten des Typs der Einträge.



2. Versuch: Realisierung (Einfügen/Löschen/Suchen)

Theoretisch:

- Einfügen eines Eintrags *v* mit Schlüssel *k*:
  - Berechne den Hashwert des Schlüssels *k*: Auswertung von $h(k)$.
  - Füge Eintrag *v* in `values[h(k)]` ein (falls bisher leer).
  - Aufwand: $O(1)$ bzw. abhängig von der Auswertung von $h(k)$.
- Löschen eines Eintrags mit Schlüssel *k*:
  - Berechne den Hashwert des Schlüssels *k*: Auswertung von $h(k)$.
  - Lösche Eintrag in `values[h(k)]` (setze `values[h(k)] = null;`).
  - Aufwand: $O(1)$ bzw. abhängig von der Auswertung von $h(k)$.
- Suchen eines Eintrags mit Schlüssel *k*:
  - Berechne den Hashwert des Schlüssels *k*: Auswertung von $h(k)$.
  - Gib Eintrag in `values[h(k)]` zurück (`return values[h(k)];`).
  - Aufwand: $O(1)$ bzw. abhängig von der Auswertung von $h(k)$.

2. Versuch: Hashfunktionen

- Damit das Einfügen, Löschen und Suchen so funktioniert, wie auf der vorherigen Folie skizziert, muss $h(K)$ für jeden Schlüssel aus K einen (eindeutigen) Array-Index zwischen 0 und $(|K| - 1)$ zurückgeben (man sagt dann, h ist eine *perfekte Hashfunktion*).
- Beispiele (Annahme: $|K| = n$):
  - Für ganzzahlige Schlüssel:

    $h : \texttt{int} \to \{0, \ldots, n-1\}$ mit

    $$h(k) = k \,\%\, n \qquad \text{(Modulo-Operator)}.$$

  - Für String-Schlüssel:

    Idee: wandle den `String` in ein **`int`** um, indem die **`int`**-Werte aller Zeichen des `String`s aufsummiert werden.

    $h : \texttt{String} \to \{0, \ldots, n-1\}$ mit

    $$h(k) = \Big(\sum_{i=0}^{k.\texttt{length()}-1} k.\texttt{charAt}(i) \Big) \,\%\, n.$$

2. Versuch: Hashfunktionen

- Grundsätzliches Vorgehen für $|K| = n$:
  - Wandle den Schlüssel in eine ganze Zahl um.
  - Berechne Hash-Adresse durch Modulo-Operator (Modulo n).
- In Java gibt es die Methode `hashCode()`, die von der Klasse `Object` an alle Java-Objekte vererbt wird.
- `hashCode()` erzeugt für das aufrufende Objekt einen ganzzahligen Wert.
- Beispiel: Die Methode `hashCode()` wurde in der Klasse `String` überschrieben mit:

$$s.\texttt{hashCode()} = \sum_{i=0}^{s.\texttt{length()}-1} s.\texttt{charAt}(i) \cdot 31^{(s.\texttt{length()}-1-i)}.$$

- Eine Hashfunktion für allgemeine Objekt-Schlüssel ist also:

$h : \texttt{Object} \to \{0, \ldots, n-1\}$ mit

$$h(k) = (k.\texttt{hashCode()})\,\%\,n.$$

2. Versuch: Problem mit Hashfunktionen

- Frage: Ist eine perfekte Hashfunktion bei sehr großer Schlüsselmenge K sinnvoll?
- Antwort: in der Regel nicht, daher verwendet man in der Praxis meistens keine perfekten Hashverfahren, sondern arbeitet mit mehreren Einträgen pro Bucket (realisierbar z.B. durch ein weiteres Array pro Bucket). Daraus ergeben sich allerdings weitere Probleme, die wir hier nicht näher betrachten wollen.
- Frage: Was passiert, wenn sich die Schlüsselmenge K ändert (z.B. neue Schlüssel hinzukommen)?
- Antwort: im Falle einer perfekten Hashfunktion benötigen wir plötzlich mehr Buckets als vorhanden, d.h. das Array hat weniger Einträge zur Verfügung als benötigt!
- In diesem Fall muss das zugrundeliegende Array erweitert und alle bisherigen Einträge entsprechend kopiert werden. Meist muss auch die verwendete Hashfunktion h an die neue Kardinalität der Schlüsselmenge angepasst werden.

2. Versuch: Analyse

- Effizienzanalyse (best case):
  - **Einfügen:** Auswerten der Hashfunktion $\Rightarrow O(1)$.
  - **Löschen:** Auswerten der Hashfunktion $\Rightarrow O(1)$.
  - **Zugriff auf Eintrag mit bestimmten Schlüssel:** Auswerten der Hashfunktion $\Rightarrow O(1)$.
- Effizienz im worst case?
- Dynamik:
  Die Implementierung mittels Hashfunktion und Array ist nicht dynamisch. Falls die Schlüsselmenge erweitert wird, muss das Array mit den Einträgen entsprechend erweitert und kopiert werden. Die Hashfunktion muss ggf. angepasst werden.
- Fazit:
  Implementierung ist sehr effizient, aber nicht dynamisch (erfordert erheblichen Mehraufwand bei Erweiterung der Schlüsselmenge).
- Standard-Implementierungen eines assoziativen Speichers in Java basierend auf Hashfunktionen sind `java.util.Hashtable` und `java.util.HashMap`.

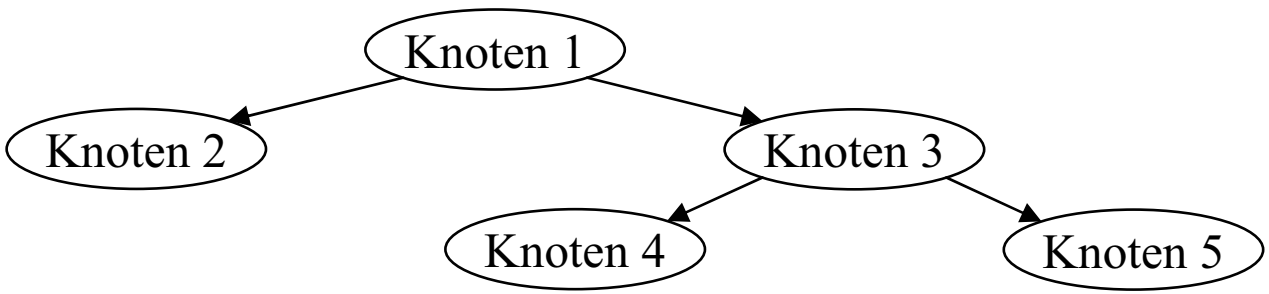
Überblick

Motivation

- Bisher hatten wir zwei konträre Ansätze zur Verwaltung von Mengen von Objekten kennen gelernt:
  - Die linearen Listen waren im worst case ineffizient beim Einfügen, Löschen und Suchen, dafür aber dynamisch.
  - Die assoziativen Speicherverfahren (Hashverfahren) waren im best case sehr effizient beim Einfügen, Löschen und Suchen, dafür aber nicht dynamisch.
- Bäume als Datenstruktur stellen nun einen Kompromiss aus den Vor- und Nachteilen beider Konzepte dar: sie sind dynamisch und bieten effizienteres Einfügen, Löschen und Suchen als Listen, allerdings weniger effizienteres als Hashverfahren im best case, aber besser als Hashverfahren im worst case.

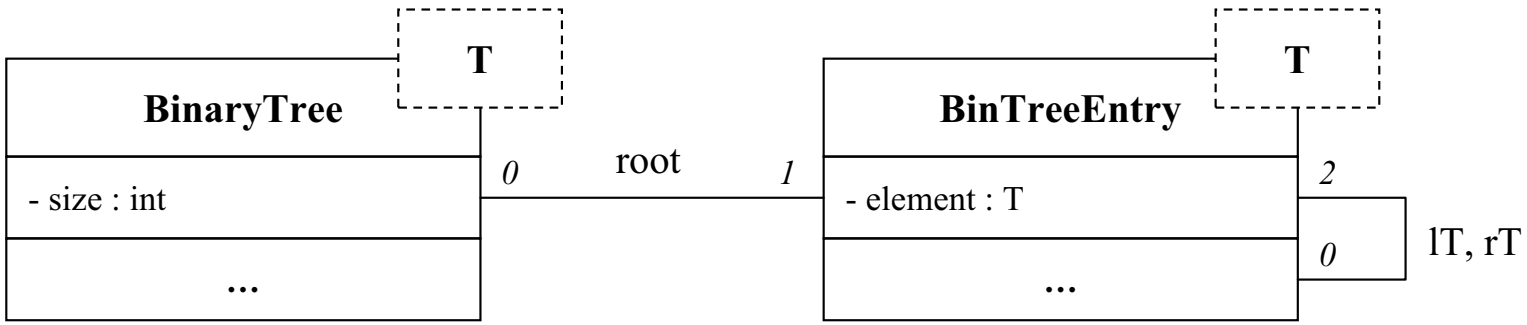
Aufbau

- Bäume organisieren Einträge (im folgenden *Knoten*) nicht mehr linear, sondern *hierarchisch*.
- Induktive Definition eines *binären Baums* über eine Knotenmenege K (siehe Kap. 3.2):
  - der leere Baum ε ist ein binärer Baum
  - sind lT und rT binäre Bäume und $k \in K$ ein Knoten (Eintrag), so ist (k, lT, rT) ebenfalls ein binärer Baum.
- Verallgemeinerung: m-äre Bäume: statt 2 Teilbäumen hat jeder Knoten m Teilbäume.
- Beispiel für einen Binärbaum:



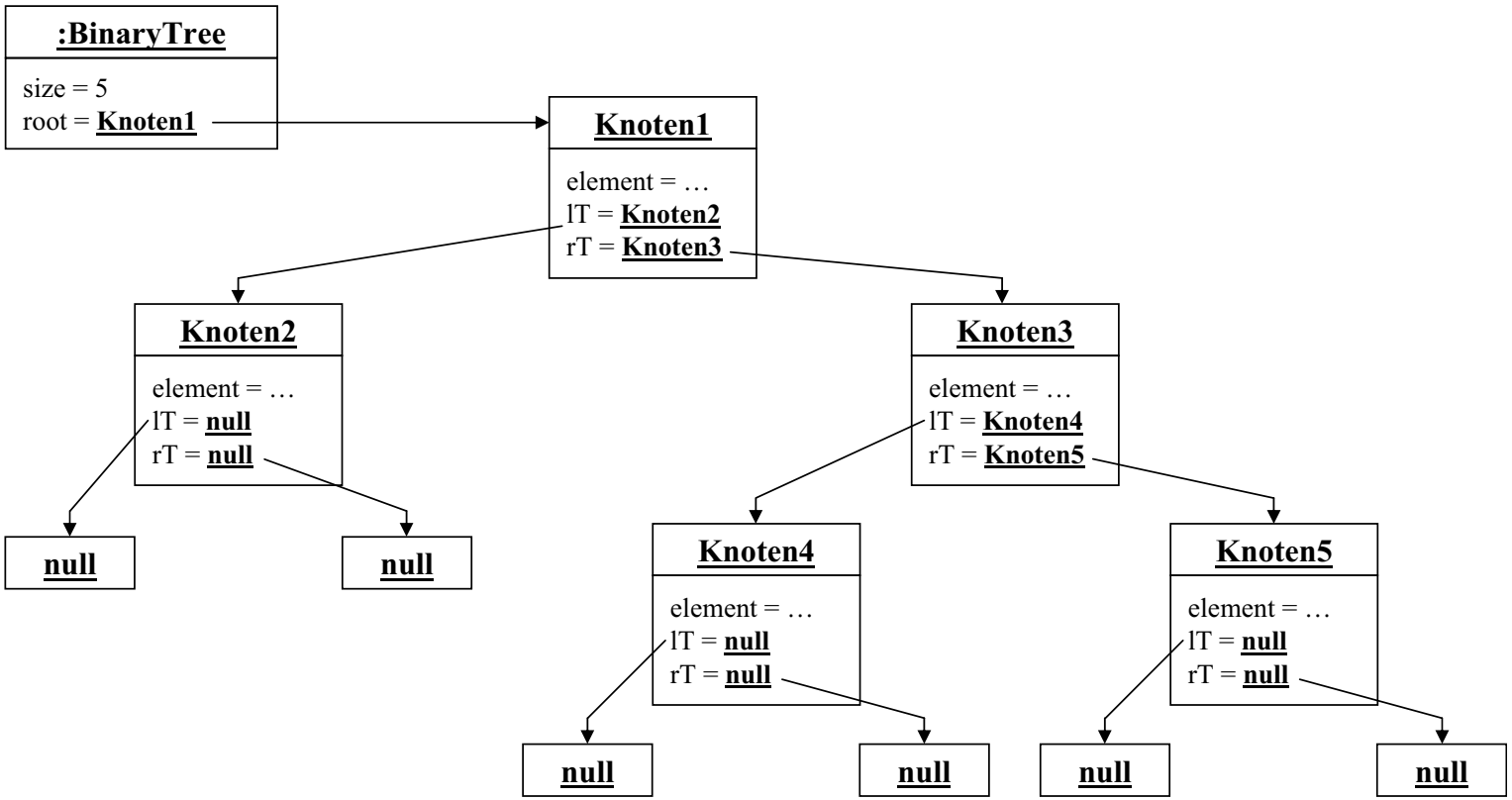
Realisierung

- *lT* und *rT* werden auch linker bzw. rechter *Teilbaum* genannt.
- Jeder Knoten ist der *Vaterknoten* (*Wurzel*) seiner Teilbäume.
- Knoten, deren linker und rechter Teilbaum leer sind, heißen *Blätter*.
- Der Vaterknoten des gesamten Baumes ist die Wurzel des Baumes.
- Bäume werden ähnlich wie Listen verkettet gespeichert.

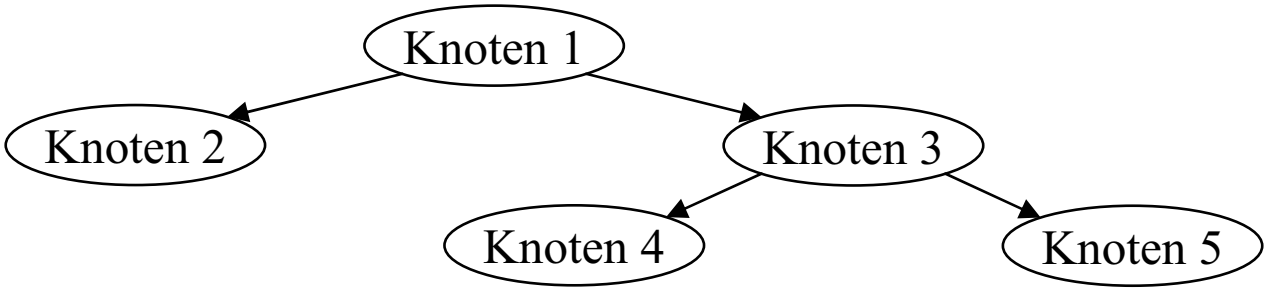


Beispiel

- Der Beispiel-Baum von der vorvorhergehenden Folie:



Baumdurchlauf



- Ein Baum kann in verschiedenen Reihenfolgen durchlaufen werden (z.B. um einen bestimmten Eintrag zu suchen) – vgl. Notation von Ausdrücken (Kapitel 3).
- *Präorder*-Reihenfolge: Wurzel – linker Teilbaum – rechter Teilbaum
  Im Beispiel: Knoten 1, Knoten 2, Knoten 3, Knoten 4, Knoten 5
- *Inorder*-Reihenfolge: linker Teilbaum – Wurzel – rechter Teilbaum
  Im Beispiel: Knoten 2, Knoten 1, Knoten 4, Knoten 3, Knoten 5
- *Postorder*-Reihenfolge: linker Teilbaum – rechter Teilbaum – Wurzel
  Im Beispiel: Knoten 2, Knoten 4, Knoten 5, Knoten 3, Knoten 1

Beispiel Präorder-Durchlauf

```
private static void preOrder(BinTreeEntry root)
{
  if(root != null)
  {
    System.out.println(root.getElement());
    preOrder(root.getlT());
    preOrder(root.getrT());
}

public void printTreeInPrOrder()
{
  preOrder(this.root);
}
```

Einfügen, Löschen, Suchen

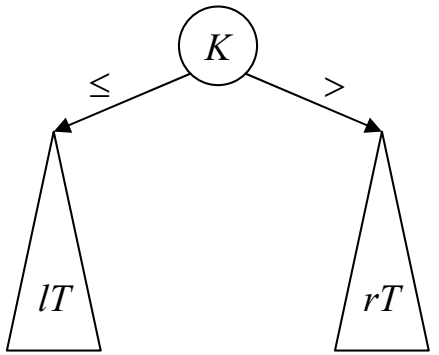
- Bisher unterscheidet sich der Aufbau eines Baumes nicht wirklich von einer Liste:
- Zum Einfügen, Löschen oder Suchen muss im schlechtesten Fall wiederum der gesamte Baum in einer bestimmten Reihenfolge durchlaufen werden $\Rightarrow O(n)$.
- Wie kann man die hierarchische Struktur des Baumes nutzen, um die Effizienz dieser Operationen zu verbessern?
- Idee: Sortiere die Einträge im Baum so, dass eine Art binäre Suche ermöglicht wird, d.h. in jedem Knoten muss eindeutig entscheidbar sein, in welchem Teilbaum die Suche fortgesetzt werden muss.

  $\Rightarrow$ **Suchbäume**

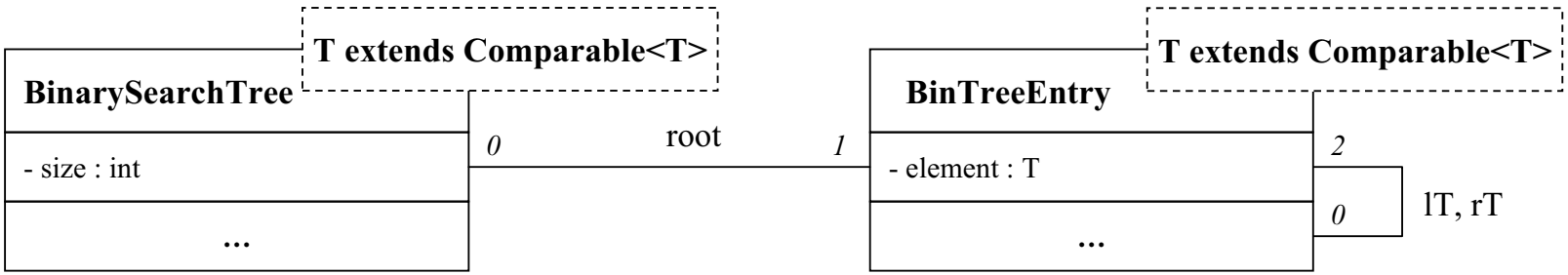
Suchbäume

- Ein (binärer) *Suchbaum* ist ein Binärbaum für dessen Knoten gilt:

  Für jeden Knoten *K* gilt: alle Schlüssel in *lT* sind kleiner oder gleich dem Schlüssel in *K* und alle Schlüssel in *rT* sind größer als der Schlüssel in *K*.

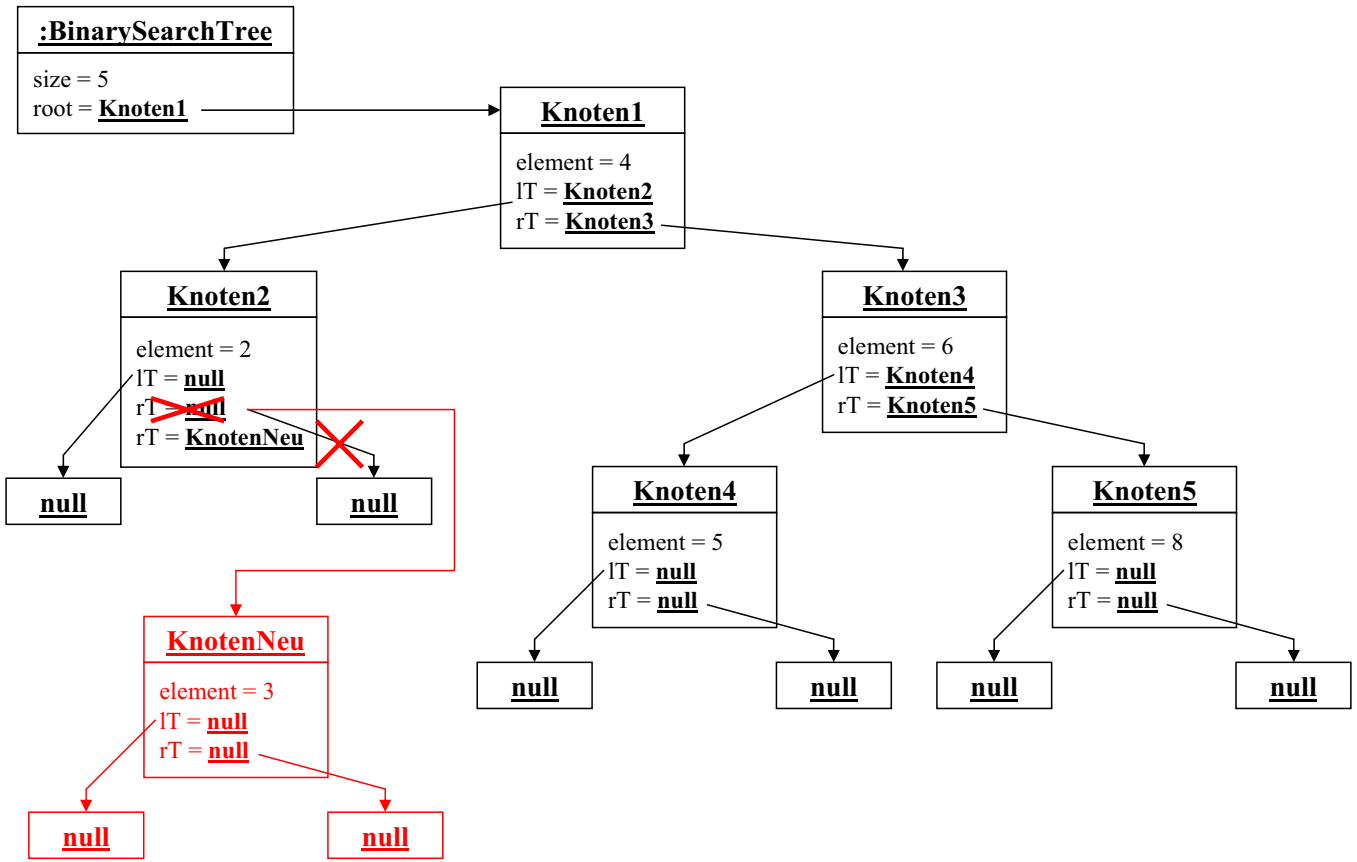


- An der Datenstruktur ändert sich nichts, außer dass der Typ `T` nun natürlich das Interface `Comparable<T>` implementieren muss.



Suchbäume: Einfügen

- Einfügen eines Schlüssels *k* in einen binären Suchbaum (Annahme: Schlüssel *k* ist noch nicht im Baum vorhanden):
  - Suche die richtige Einfügestelle (Blatt).
  - Füge neuen Knoten *k* mit Schlüssel *k* ein.



Suchbäume: Einfügen (Teil 1)

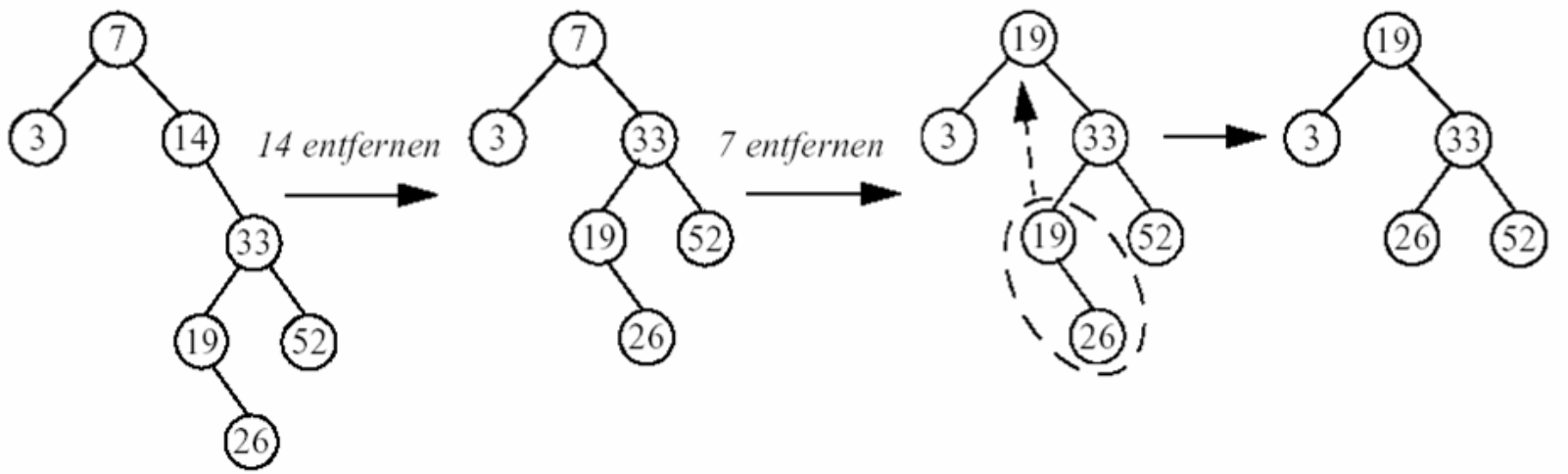
```
public void insert(T k)

  BinTreeEntry newNode = new BinTreeEntry(k);
  if(this.root == null)
  {
    this.root = newNode;
  }

  // Suche Einfuegestelle
  BinTreeEntry currNode = this.root;
  BinTreeEntry father = null;
  while(currNode != null)
  {
    father = currNode;
    if(newNode.getElement().compareTo(currNode.getElement()) < 0)
    {
      currNode = currNode.getlT();
    }
    if(newNode.getElement().compareTo(currNode.getElement()) > 0)
    {
      currNode = currNode.getrT();
    }
    else
    {
      throw new Exception("Schluessel "+k+" ist bereits vorhanden.");
  }
```

Suchbäume: Einfügen (Teil 2)

```
    // Fuege ein
    if(newNode.getElement().compareTo(father.getElement()) < 0)
    {
      father.setlT(newNode);
    }
    else
    {
      father.setrT(newNode);
    }
    // lT und rT von newNode sind null

}
```

Suchbäume: Löschen

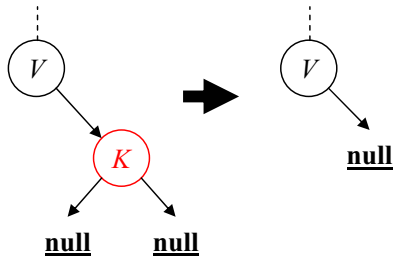
- Löschen eines Knoten *K* mit Schlüssel *k* in einem binären Suchbaum:
  - Suche Knoten *K*.
  - Lösche *K*.
- Löschen ist etwas komplizierter, da nun auch *innere Knoten* (Knoten, die nicht Blätter sind) entfernt werden können:



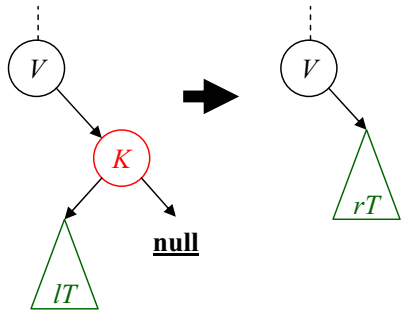
Suchbäume: Löschen

- Fall 1: Knoten *K* besitzt höchstens einen Sohn (Blatt oder Halbblatt)

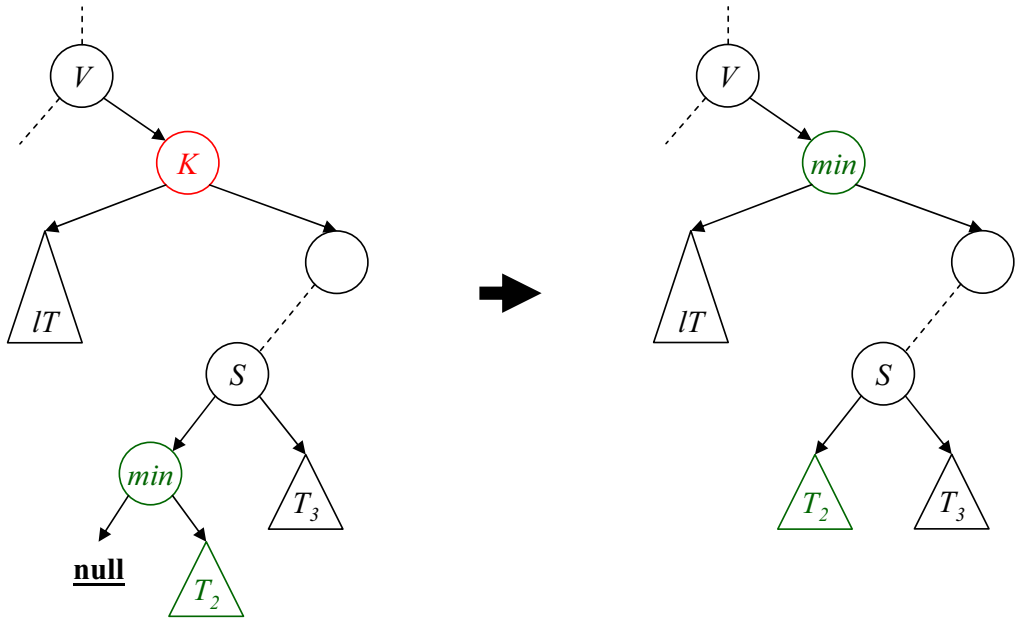
Fall 1.1 *K* ist Blatt



Fall 1.2 *K* hat linken Teilbaum
(rechts: symmetrisch)



- Fall 2: Knoten *K* besitzt zwei Söhne (innerer Knoten)



Suchbäume: Suchen

- Suchen eines Schlüssels k in einen binären Suchbaum:
  - Wenn die Wurzel R leer ist: Suche erfolglos beendet.
  - Vergleiche k mit dem Schlüssel der Wurzel R.
  - Bei Gleichheit: Suche erfolgreich beendet.
  - Wenn $k < R$.Schlüssel: suche rekursiv in lT nach k.
  - Wenn $k > R$.Schlüssel: suche rekursiv in rT nach k.
- Laufzeit: $O(h)$, wobei h die Höhe des Baumes ist.
- Höhe eines binären Suchbaums:
  - Worst-Case: Baum entartet zur linearen Liste $\Rightarrow O(n)$.
  - Im Durchschnitt (alle Permutationen der Einfüge-Reihenfolge sind gleichwahrscheinlich): $O(\log n)$.

Suchbäume: Suchen

```
public BinTreeEntry suche(T k)
{
  return suche(this.root, k);
}

private BinTreeEntry suche(BinarySearchTree tree, T key)
{
  if(tree.root == null)
  {
    return null;
  }
  if(key.compareTo(tree.root.getElement()) == 0)
  {
    return tree.root;
  }
  if(key.compareTo(tree.root.getElement()) < 0)
  {
    return suche(tree.root.getlT(), key);
  }
  else
  {
    return suche(tree.root.getrT(), key);
  }
}
```

Suchbäume: Analyse

- Effizienzanalyse:
  - **Einfügen:** Suche nach Einfügestelle $\Rightarrow$ im Durchschnitt $O(\log n)$, im worst-case $O(n)$.
  - **Löschen:** Suche nach zu löschendem Eintrag $\Rightarrow$ im Durchschnitt $O(\log n)$, im worst-case $O(n)$.
  - **Zugriff auf Eintrag mit bestimmten Schlüssel:** Suche im Durchschnitt $O(\log n)$, im worst-case $O(n)$.
- Dynamik:
  Die Implementierung ist dynamisch wie verkettete Listen.
- Fazit:
  Kompromiss zwischen (zumindest theoretisch) effizientem Hashing und dynamisch verketteter Liste.
- Eine Standard-Implementierung eines Suchbaumes als assoziativer Speicher ist `java.util.TreeMap`.

Überblick

Mengen und Multimengen

- Eine *Menge* ist eine (ungeordnete) Zusammenfassung verschiedener Objekte. Insbesondere enthält eine Menge keine Duplikate.
- Eine *Multimenge* ist eine Menge, die Duplikate enthalten kann.
- Mengen und Multimengen sind wichtige Konzepte, u.a. um Assoziationen zwischen Objekten verschiedener Klassen umzusetzen.
- Im folgenden wiederholen wir grundlegende Funktionalitäten von Mengen. Eine Implementierung sollte entsprechende Methoden für diese Funktionalitäten anbieten. Diese gelten natürlich genauso für Multimengen.

Grundlegende Operationen auf Mengen

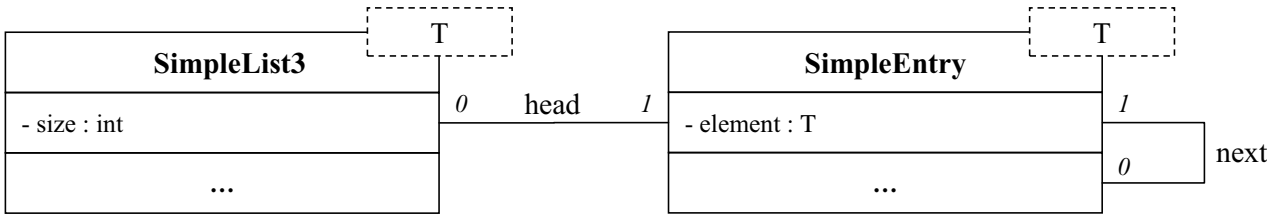
- Leere Menge:
  $\emptyset :\rightarrow Set(T)$
- Elementbeziehung:
  $\in: T \times Set(T) \rightarrow \mathbb{B}$
- Teilmengenbeziehung:
  $\subseteq: Set(T) \times Set(T) \rightarrow \mathbb{B}$
- Vereinigung:
  $\cup : Set(T) \times Set(T) \rightarrow Set(T)$
- Durchschnitt:
  $\cap : Set(T) \times Set(T) \rightarrow Set(T)$
- Differenz:
  $\setminus : Set(T) \times Set(T) \rightarrow Set(T)$
- Kardinalität:
  $|.| : Set(T) \rightarrow \mathbb{N}$

Realisierung von Mengen

- Zur effizienten Realisierung von Mengen spielen folgende Gedanken eine Rolle:
  - Die grundlegenden Operationen sollten effizient sein.
  - Das Einfügen eines neuen Elementes sollte effizient sein. Insbesondere ist darauf zu achten, dass keine Duplikate eingefügt werden.
  - Das Löschen eines Elementes sollte effizient sein.
  - Die Suche nach einem Element sollte effizient sein.
- Für die Realisierung von Multimengen spielen natürlich die selben Gedanken eine Rolle. Allerdings muss nun auf Duplikate keine Rücksicht genommen werden.

Realisierung von Mengen: Beispiel

- Als Beispiel schauen wir uns eine Implementierung einer Menge auf Basis einer typisierten (einfach) verketteten Liste an.
- Zur Erinnerung:



Realisierung von Mengen: Beispiel

```
public class Menge<T>
{
  private SimpleList3<T> elemente;

  public int cardinalitaet()
  {
    return this.elemente.length();
  }

  public T[] getElemente()
  {
    return this.elemente.asArray();
  }

  public void einfuegen(T element)
  {
    if(!this.enthaelt(element))
    {
      this.elemente.add(element);
    }
  }
  ...
}
```

Realisierung von Mengen: Beispiel ($\in$, $\cup$)

```
public boolean enthaelt(T t)
{
  return this.elemente.contains(t);
}


public void vereinigung(Menge<T> andereMenge)
{
  T[] neueElemente = andereMenge.getElemente();
  for(T t : neueElemente)
  {
    this.einfuegen(t);
  }
}
```

Realisierung von Mengen: Beispiel ($\subseteq$)

```
public boolean teilmengeVon(Menge<T> superMenge)
{
  boolean istTeilmenge = true;
  T[] elemente = this.getElemente();
  for(T t : elemente)
  {
    istTeilmenge = istTeilmenge && superMenge.enthaelt(t);
  }
  return istTeilmenge;
}
```

Zusammenfassung

Vergleich der durchschnittlichen Laufzeiten

	einfaches Array	einfache Liste	doppelt verankerte Liste	doppelt verkettete Liste	perfekte Hashverfahren (assoz. Array)	binärer Suchbaum
dynamisch	nein	ja	ja	ja	nein	ja
Einfügen (Schlüssel)	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$ bzw. $O(n)$	$O(\log n)$
Löschen (Schlüssel)	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(\log n)$
Suchen (Schlüssel)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(\log n)$

- Eine Standard-Implementierung einer Menge in Java ist z.B. die Klasse `java.util.HashSet`, die auf einem assoziativen Speicher beruht. Dabei wird intern die Methode `hashCode()` für die Objekte der Menge verwendet. Achtung: Die Laufzeit dieser Klasse degeneriert, falls die Hashfunktion (die auf der Methode `hashCode()` basiert), nicht optimal ist!

Überblick

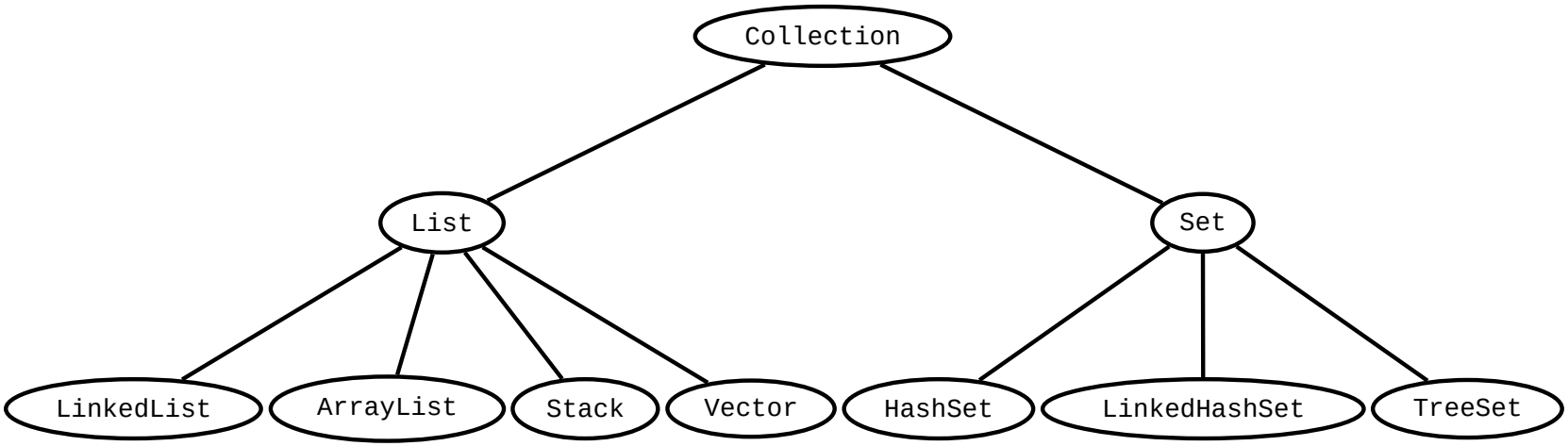
Collections in Java

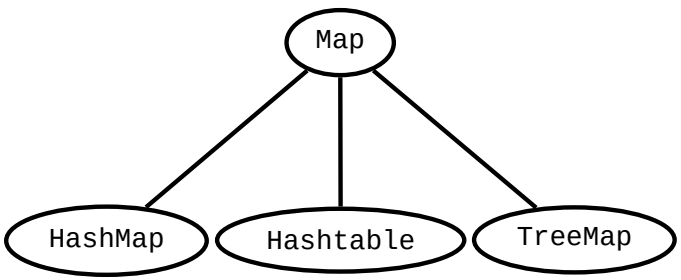
- In Java gibt es eine große Menge an vordefinierten Klassen für mengenartige Datenstrukturen.
- Im folgenden geben wir einen kurzen Überblick über das Collections-Framework von Java.
- Zur Vertiefung empfehlen wir das intensive Studium der Dokumentationen der entsprechenden Klassen.

Das Collections-Framework

- Alle Klassen mengenartiger, assoziativer Datenstrukturen implementieren das Interface `java.util.Map`. Das Interface stellt Basis-Funktionalitäten wie Einfügen (`put`), Löschen (`remove`) und verschiedene Formen der Suche (`get`, `containsKey`, `containsValue`) zur Verfügung.
- Alle Klassen mengenartiger, nicht-assoziativer Datenstrukturen implementieren das Interface `java.util.Collection`. Das Interface stellt Basis-Funktionalitäten wie Einfügen (`add`), Löschen (`remove`) und Suchen (`contains`) zur Verfügung.
- Vom Interface `java.util.Collection` abgeleitete Interfaces sind u.a.:
  - `java.util.Set`: Klassen, die Mengen ohne Duplikate modellieren, implementieren dieses Interface.
  - `java.util.List`: Klassen, die Multimengen modellieren, implementieren dieses Interface.

Hierarchie


Collection
List
Set
LinkedList
ArrayList
Stack
Vector
HashSet
LinkedHashSet
TreeSet


Map
HashMap
Hashtable
TreeMap

Wiederholung: Iteration über Arrays

```
public static void accessFor(Integer[] intArray)
{
  Integer currentInteger;
  for(int i = 0; i < intArray.length; i++)
  {
    currentInteger = intArray[i];
  }
}
```

Laufzeit?

Iteration über Collections

Um iterative Aktionen auf allen Elementen einer Collection (oder einer bestimmten Teilmenge davon) durchzuführen, sind nicht alle Schleifenkonstrukte gleichwertig.
Beispiel:

```
public static void accessFor(List<Integer> list)
{
  Integer currentInteger;
  for(int i = 0; i < list.size(); i++)
  {
    currentInteger = list.get(i);
  }
}
```

Laufzeit?

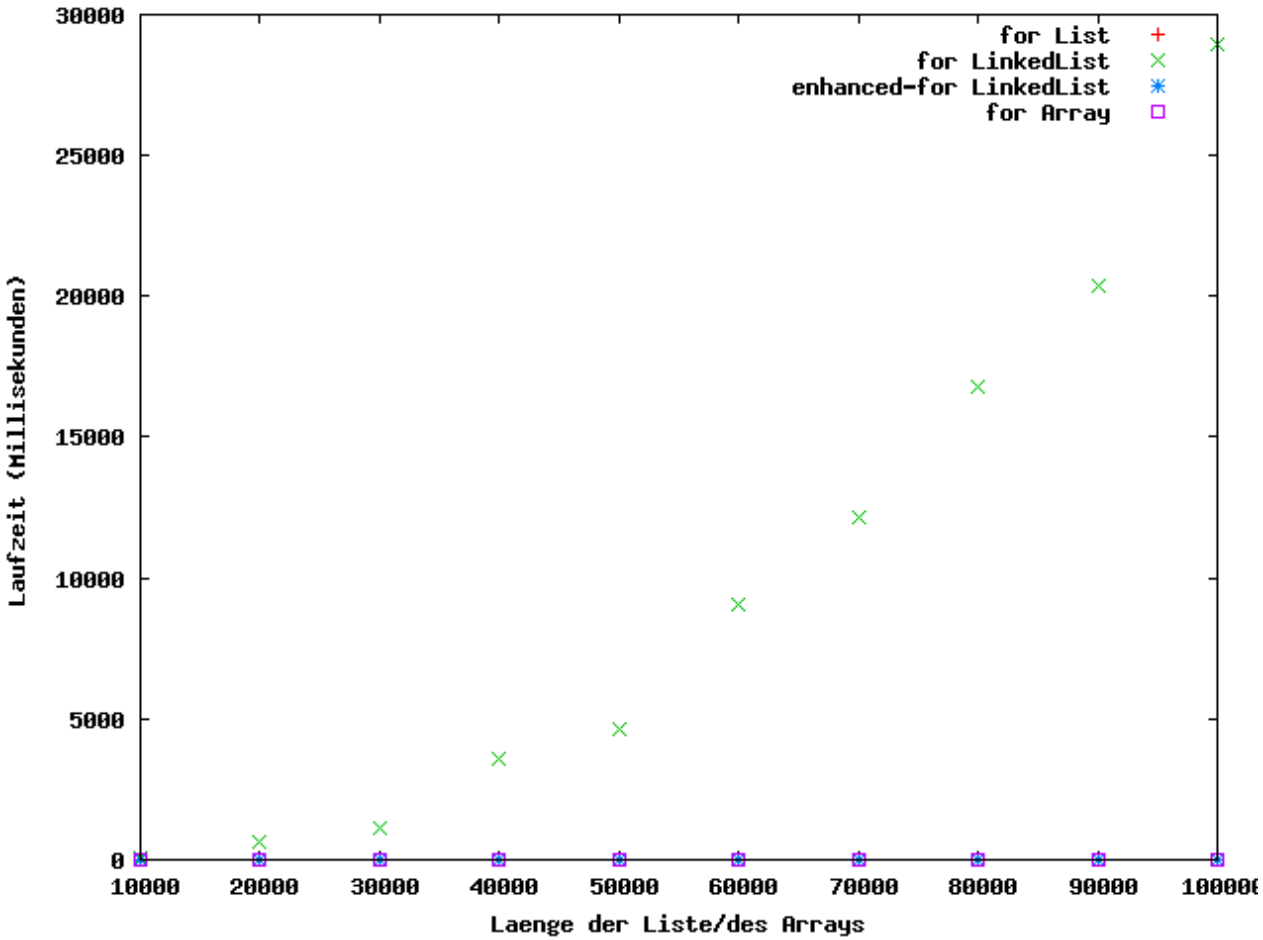
Die `Iterator`- und `Iterable`-Interfaces und *enhanced for*

Eine `Collection` implementiert das Interface `Iterable`, das nur vorschreibt, dass ein `Iterator` zurückgegeben werden kann, der es ermöglicht, die Collection zu durchwandern. Dieser `Iterator` wird auch implizit im sogenannten *enhanced for* verwendet:

```
public static void accessIterable(
            Iterable<Integer> intIterable)
{
  Integer currentInteger;
  for(Integer currentInt : intIterable)
  {
    currentInteger = currentInt;
  }
}
```

Laufzeit für eine verkettete Liste?

Test für die verschiedenen Iterationen auf verschiedenen Listen und einem Array



Test-Klasse:

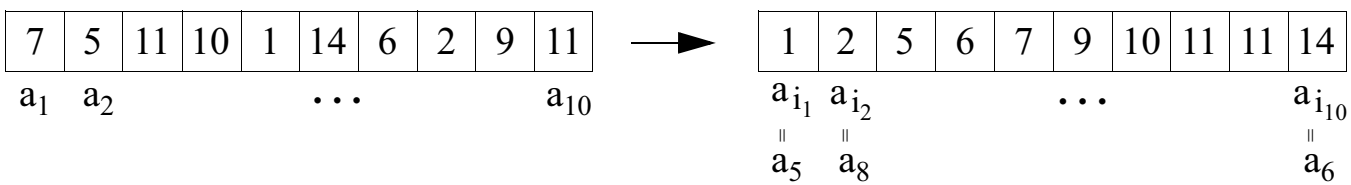
`http://www.dbs.ifi.lmu.de/Lehre/EIP/WS_2008/skript/programmbeispiele/datenstrukturen/IterationTest.java`

Abschnitt 22: Sortierverfahren

Überblick

Motivation

- Sortieren ist eine wichtige Operation auf Mengen von Objekten.
- Sortierte Mengen erlauben deutlich effizienteren Zugriff auf einzelne Elemente (z.B. durch binäre Suche).
- Problemstellung:
  - Gegeben: n Objekte $a_1, \ldots, a_n$ mit ihren Schlüsseln $k_1, \ldots, k_n$, anhand derer die Objekte sortiert werden sollen.
  - Gesucht: Eine Anordnung $a_{i_1}, \ldots, a_{i_n}$ mit
    - $(i_1, \ldots, i_n)$ ist eine Permutation von $(1, \ldots, n)$
    - $k_{i_1} \leq k_{i_2} \leq \ldots \leq k_{i_n}$ (*Sortierkriterium*).
- Beispiel:



Allgemeines

- Der Typ der Schlüssel k_i muss entweder primitiv sein, oder das Interface `Comparable` implementieren.
- Im folgenden betrachten wir folgende Vereinfachungen:
  - Wir verwenden die Schreibweisen $k_i < k_j$, $k_i > k_j$, usw., bzw. $k_i = k_j$ unabhängig davon, ob k_l einen primitiven Datentyp oder einen Objekttyp besitzt (im letzteren Fall müssten wir eigentlich mit den Methoden `compareTo` bzw. `equals` arbeiten).
  - Wir nehmen an, dass die Datenstruktur zur Verwaltung der Menge ein einfaches Array ist.
- Allgemeiner Sortieralgorithmus:
  while $\exists(i,j) : (i < j) \wedge (k_i > k_j)$ **do** vertausche a_i und a_j;
- Problem: Algorithmus nicht deterministisch.
- Daher: speziellere Algorithmen nötig.

Stabilität von Sortierverfahren

Definition (Stabilität von Sortierverfahren)

Ein Sortierverfahren heisst *stabil*, wenn die relative Ordnung von Elementen mit gleichen Schlüsselwerten beim Sortieren erhalten bleibt, d.h. wenn für die sortierte Menge $k_{i_1}, \ldots k_{i_n}$ gilt: $k_{i_j} = k_{i_l}$ und $j < l \Rightarrow i_j < i_l$.

Definition (in situ Sortierverfahren)

Ein Sortierverfahren heisst *in situ*, wenn zusätzlich zu dem zu sortierenden Array *kein* weiterer Speicherplatz benötigt wird. Dabei wird angenommen, dass die zu sortierenden Objekte die Indexpositionen $1, \ldots n$ belegen und das Feld mit Index 0 für Vertauschungen genutzt werden kann.

Klassifikation von Sortierverfahren

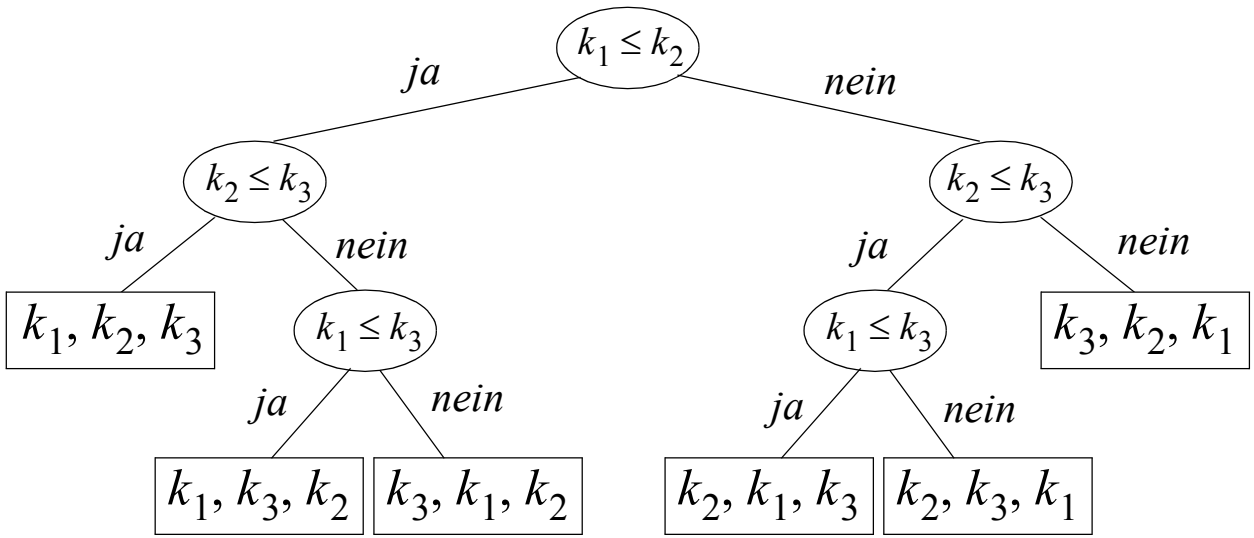
- Es gibt verschiedene Kriterien, um Sortieralgorithmen zu klassifizieren, z.B. ihre Stabilität oder ihren Speicherplatzbedarf.
- Das in dieser Vorlesung wesentliche Kriterium ist das Laufzeitverhalten.
- Um das Laufzeitverhalten unterschiedlicher Verfahren miteinander zu vergleichen, zählen wir die Anzahl der bei einer Sortierung von n Objekten durchzuführenden Operationen (in Abhängigkeit von n).

Lauzeitverhalten von Sortierverfahren

- Die beiden wesentlichen Operationen der meisten Sortierverfahren sind:
  - *Vergleiche* von Schlüsselwerten um Informationen über die vorliegende Ordnung zu erhalten. Im folgenden bezeichnen wir die Anzahl dieser Schlüssel-Vergleiche mit $C(n)$.
  - *Zuweisungsoperationen*, z.B. Vertauschungoperationen oder Tranpositionen von Objekten (i.a. innerhalb eines Arrays) zur Herstellung der Sortierordnung. Im folgenden bezeichnen wir die Anzahl dieser Zuweisungsoperationen mit $M(n)$.
- Wir beschränken uns hier auf Algorithmen, die nur Schlüsselvergleiche und Transpositionen verwenden.

Wie schnell kann man sortieren?

- Gesucht: eine untere Schranke für die Anzahl $C_{max}(n)$ von Schlüsselvergleichen, die im schlechtesten Fall notwendig sind, um n Objekte zu sortieren.
- Entscheidungsbaum für 3 Schlüssel k_1, k_2 und k_3:



- In einem Entscheidungsbaum ohne redundante Vergleiche entspricht jedes Blatt einer der $n!$ verschiedenen Permutationen.
- Da der Entscheidungsbaum ein binärer Baum ist, hat er für n Elemente eine minimale Höhe von $\lceil \log(n!) \rceil$.

Wie schnell kann man sortieren?

- Damit gilt für einen beliebigen Sortieralgorithmus:

$$C_{max}(n) \geq O(n \cdot \log n), \text{ denn}$$

$$C_{max}(n) \geq \lceil \log(n!) \rceil \quad \text{und}$$

$$n! \geq n \cdot (n-1) \cdot \ldots \cdot (\lceil n/2 \rceil) \geq (\frac{n}{2})^{\frac{n}{2}}$$

$$\Rightarrow \log_2(n!) \geq \frac{n}{2} \cdot \log_2(\frac{n}{2}) = O(n \cdot \log n).$$

- Resultat: Sortierverfahren haben mindestens eine Laufzeit von $O(n \cdot \log n)$.

Überblick

Überblick

- Einfache Sortierverfahren besitzen meist eine Laufzeit von $O(n^2)$.
- Für wenige Objekte (kleine Werte von n) ist dies meist noch akzeptabel, für große Mengen (z.B. $n > 100$) allerdings nicht mehr.
- Im folgenden besprechen wir einige einfache Sortierverfahren informell. Diese Algorithmen wurden in den Übungen für den primitiven Datentyp **`int`** implementiert. Nun sollte es Ihnen nicht mehr allzu schwer fallen, diese Algorithmen für beliebige Typen in einem Java-Programm zu implementieren.

Sortieren durch Abzählen: Count Sort

- Prinzip:
  Der j-te Schlüssel der sortierten Folge ist größer als $j-1$ der übrigen Schlüssel. Die Position eines Schlüssels in der sortierten Folge kann damit durch Abzählen der kleineren Schlüssel ermittelt werden.
- Vorgehen:
  Für jedes Objekt o im ursprünglichen Array: zähle die Objekte o' mit $o' < o$.
- Laufzeit:
  $C(n) = (n-1) + (n-2) + \ldots + 1 = \frac{n \cdot (n-1)}{2} = O(n^2)$
  $M(n) = C(n) = O(n^2)$
- Das Verfahren ist nicht in situ.
- Das Verfahren ist stabil (wird instabil falls Test $o' \leq o$).

Sortieren durch direktes Auswählen: Selection Sort

- Prinzip/Vorgehen:
  Suche aus allen n Elementen das kleinste und setze es an die erste Stelle. Wiederhole dieses Verfahren für die verbleibenden Elemente. Nach $n-1$ Durchläufen ist die Folge sortiert.
- Laufzeit:
  $C(n) = (n-1) + (n-2) + \ldots + 1 = \frac{n \cdot (n-1)}{2} = O(n^2)$
  $M_{\emptyset}(n) = O(n \log n)$ (ohne Beweis)
- Das Verfahren ist in situ.
- Je nach Implementierung ist das Verfahren stabil.

Sortieren durch direktes Austauschen: Bubble Sort

- Prinzip:
  Vertausche die relative Reihenfolge benachbarter Elemente, sodass kleine Elemente nach vorne und größere Elemente nach hinten wandern.
- Vorgehen:
  Im ersten Durchlauf werden die Paare (a_{n-1}, a_n), (a_{n-2}, a_{n-1}), ... bearbeitet. Dadurch wandert das kleinste Element an die erste Position im Array. Nach $n-1$ Durchläufen ist die Sortierung abgeschlossen.
- Laufzeit:
  $C(n) = (n-1) + (n-2) + \ldots + 1 = \frac{n \cdot (n-1)}{2} = O(n^2)$
  $M_\emptyset(n) = O(n^2)$ (ohne Beweis)
- Das Verfahren ist in situ.
- Je nach Implementierung ist das Verfahren stabil.

Überblick

Das Divide-and-Conquer-Prinzip

- Analyse: einfache Sortierverfahren reduzieren die Größe des noch zu sortierenden Arrays in jedem Schritt lediglich um eins.
- Idee: Reduziere das noch zu sortierende Array in jedem Schritt um die Hälfte. Dadurch wird eine Laufzeit von $O(n \log n)$ ermöglicht.
- Das allgemeine Algorithmus-Prinzip von Quicksort heißt *Divide-and-Conquer* (teile-und-beherrsche).
- Divide-and-Conquer-Algorithmen zerlegen ein gegebenes Problem solange in kleinere Probleme, bis diese beherrschbar sind. Die globale Lösung ergibt sich durch Verschmelzen der einzelnen (Teil-)Lösungen.

Divide-and-Conquer-Sortieralgorithmen

- Allgemeines Schema eines Divide-and-Conquer-Sortieralgorithmus:
  - Wenn die Objektmenge klein genug ist, löse das Problem direkt.
  - Ansonsten:
    DIVIDE: Zerlege die Menge in möglichst gleich große Teile.
    CONQUER: Löse das Problem für jede Teilmenge.
    MERGE: Berechne aus den Teil- die Gesamt-Lösung.
- Wichtige Eigenschaft: Jedes Divide-and-Conquer-Sortierverfahren, dessen *Divide*- und *Merge*-Schritt in $O(n)$ Zeit durchgeführt werden können und das eine balancierte Unterteilung des Problems garantiert, besitzt eine Laufzeit von $O(n \log n)$.

Quicksort: Prinzip

- Ein (prinzipiell beliebiger) Schlüssel x aus dem Array wird ausgewählt (das sog. *Pivot-Element*).
- *Divide*-Schritt: das Array wird in Schlüssel $\geq x$ und Schlüssel $< x$ zerlegt.
- *Conquer*-Schritt: die beiden resultierenden Teilarrays werden rekursiv bis auf Elementebene in gleicher Weise behandelt.
- *Merge*-Schritt: entfällt, durch entsprechende Speicherung der Teilarrays innerhalb des Arrays.

Quicksort: Algorithmus

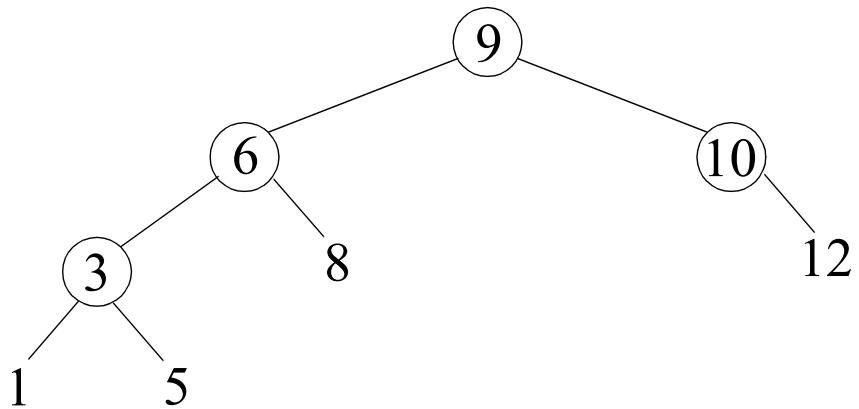
```
ALGORITHM QuickSort(ARRAY S)
  IF S.length = 1
  THEN RETURN S;
  ELSE
    // DIVIDE:
    Waehle ein Pivot-Element x aus S;
    Initialisiere zwei Teilfolgen S1 und S2;
    FOR EACH y IN S DO
      IF y < x
      THEN fuege x zu S1 hinzu;
      ELSE fuege x zu S2 hinzu;
      ENDIF
    ENDFOR
    // CONQUER:
    S1' = QuickSort(S1);
    S2' = QuickSort(S2);
    // MERGE:
    RETURN Konkatenation aus S1' und S2';
  ENDIF
END
```

- Arbeit im *Divide*-Schritt: $O(n)$.
- Arbeit im *Merge*-Schritt: $O(1)$.

Quicksort: Beispiel

9	5	1	12	8	10	6	3

- Rekursive Unterteilung des Array bzgl. des Pivot-Elements ergibt einen Binärbaum über den Array-Elementen (innere Knoten entsprechen den einzelnen Pivot-Elementen).



- Beobachtung: Höhe des Baumes entspricht Rekursionstiefe des Verfahrens.
- Erwünscht: balancierter binärer Baum $\Rightarrow$ Rekursionstiefe $O(\log n)$.

Quicksort: Laufzeit

- Im besten Fall:
  In jedem Divide-Schritt wird immer in zwei gleich große Teilarrays geteilt: $O(n \log n)$.
- Im Durchschnitt: $O(n \log n)$.
- Im schlechtesten Fall:
  Degenerierung zur linearen Liste: stets wird das größte/kleinste Element als Pivot gewählt: $O(n^2)$

Mehr dazu in der Vorlesung “Effiziente Algorithmen”.

Quicksort: Schlussbemerkungen

- Austausch von Schlüsseln über große Distanzen $\Rightarrow$ Array ist schnell "nahezu" sortiert.
- Wie alle komplexen Sortierverfahren ist Quicksort schlecht für kleine n.
- Es gibt verschiedene Methoden, die Wahrscheinlichkeit des schlechtesten Falls zu vermindern; dennoch bleibt Quicksort immer eine Art "Glückspiel".
- Quicksort war lange Zeit das im Durchschnittsverhalten beste bekannte Sortierverfahren.
- Die statischen Methoden `java.util.Arrays.sort` implementieren Quicksort.