Programmierung

Hochschule Furtwangen
Studiengang Medieninformatik
Prof. Dr. Dirk Eisenbiegler

Stand: Wintersemester 2022/23

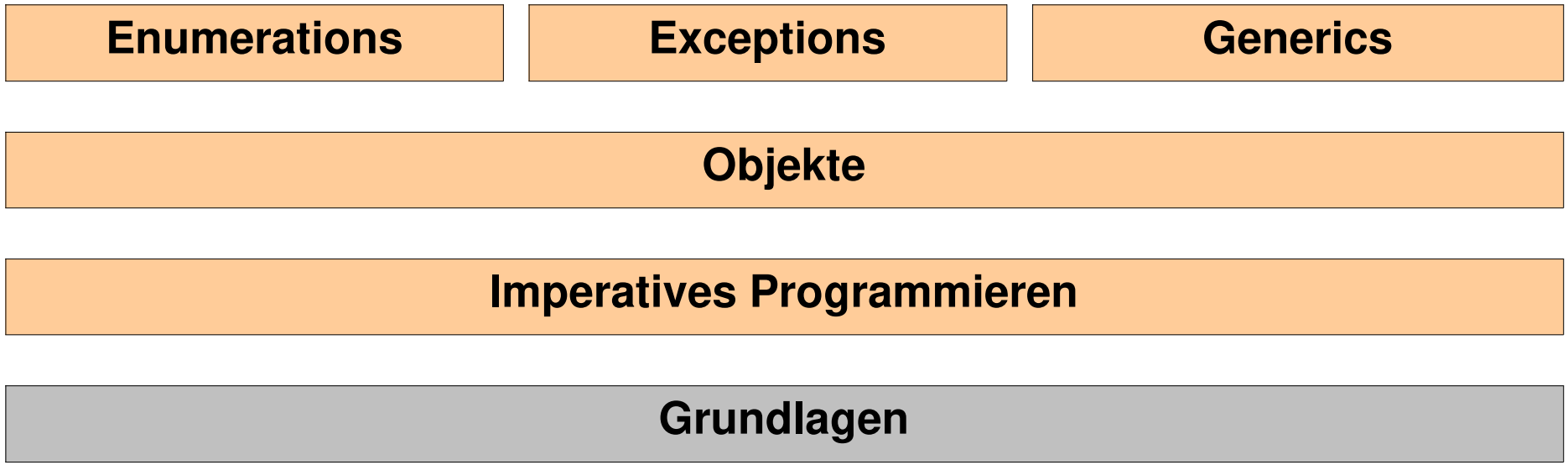
Organisatorisches zum Praktikum

- Der Praktikumsschein und die Klausur zur Vorlesung sind zwei eigenständige Prüfungsleistung, die Sie unabhängig voneinander erbringen müssen.
- Wenn Sie einen eigenen Notebook besitzen, so können Sie diesen gerne im Praktikum einsetzen. Die verwendete Software ist durchgängig frei erhältlich. Beachten Sie die Hinweise zur Software-Installation im Praktikum.

Empfehlungen

- ✗ Bearbeiten Sie alle Programmieraufgaben unbedingt selbstständig.
- ✗ Zum besseren Verständnis der in der Vorlesung beschriebenen Konzepte kann es sinnvoll sein, die dort beschriebenen Programme und Programmstücke Schritt für Schritt durchlaufen zu lassen.
  - ⇨ Tipp: Installieren Sie dazu die Java-Entwicklungsumgebung auf Ihrem eigenen Rechner und gehen Sie die Programmstück mit dem Debugger durch.
- ✗ Berücksichtigen Sie, dass der Aufwand bei dieser Veranstaltung für die Studenten sehr unterschiedlich sein kann. Es hat sich gezeigt, dass sich vor allem die Studenten schwer tun, die bisher noch wenig oder gar keine Programmiererfahrung haben.

Übersicht


Enumerations
Exceptions
Generics
Objekte
Imperatives Programmieren
Grundlagen

Kapitel 1

Grundlagen

1.1 Einleitung

Programm

Fahrradschlauch flicken

1. den Schlauch herausnehmen
2. das Ventil wieder anschrauben
3. den Schlauch aufpumpen und aufgepumpt in einen Wassereimer halten
4. die Stelle, aus der Luft austritt, mit einem Flicken abdichten
5. den Schlauch einbauen
6. den Schlauch aufpumpen

Programm

Programm

Joh. Seb. Bach Italienisches Konzert, BWV 971
1685-1750 Arr. Rainer Seidel

Leopold Mozart Frosch Parthia
1719-1787 Arr. Frédéric Schwenk

Georg Chr. Wagenseil Sonata VI in G, Andante-Andante-Allegro
1715-1777

Joseph Haydn Klaviertrio G-Dur, Presto-Rondo all'ongarese
1732-1809 Arr. Frédéric Schwenk

****** P A U S E ******

Gaetano Donizetti Ouv. zu der Oper "Don Pasquale"
1797-1848 Arr.Thomas Horch

Adolf Bergt Trio für 3 Fagotte
1822-1862

Programm

Käsespätzle für 4 Personen

Zutaten: 4 Eier, 1/8 l Wasser, 300 g Käse, 200 g Sahne, Röstzwiebeln, Pfeffer, Salz

Mehl, Eier, Salz und Wasser zu einem glatten Teig verrühren. Teig ca. 15 Minuten ruhen lassen, damit das Mehl quellen kann. Den Teig durch eine Spätzlepresse in kochendes Salzwasser drücken. Solange darin ziehen lassen, bis sie an der Oberfläche schwimmen. Mit der Schaumkelle Herausheben und in ein Sieb zum Abtropfen geben. Den Käse fein reiben. Abwechselnd Spätzle und Käse in eine feuerfeste, gefettete Auflaufform geben. Mit Käse abschließen. Mit Pfeffer würzen und die Sahne darübergießen. Im Ofen bei 200 Grad ca. 15 - 25 Minuten überbacken (bis der Käse zerlaufen ist). Auf vorgewärmte Teller geben und mit Röstzwiebeln bestreuen.

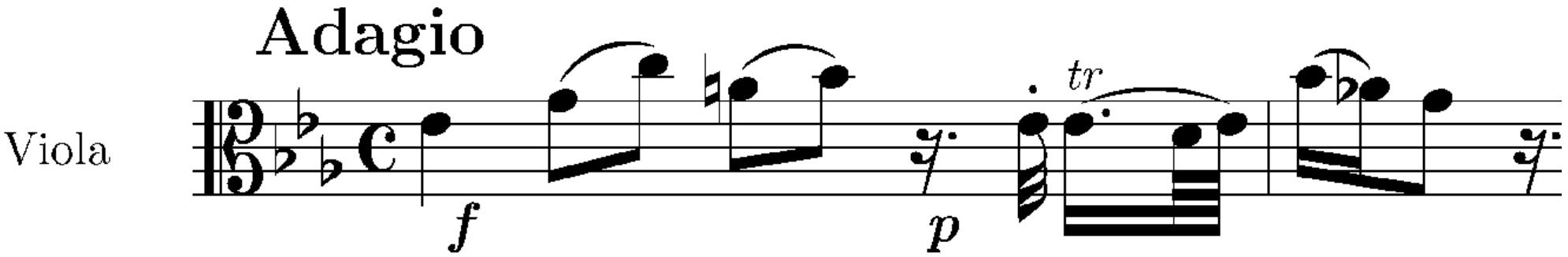
Programm

resep nasi goreng (ukuran porsi: 6 orang)

Bahan-bahan: 600 gram nasi matang, 4 sendok makan kecap manis, 2 sendok makan saus tomat, 5 sendok makan minyak, 5 buah bawang merah, 2 siung bawang putih, 5 buah cabai merah, 1/2 sendok teh terasi matang, 1 buah timun, potong bulat tipis, 6 butir telur, diceplok, 100 gram emping goreng, 2 sendok makan bawang merah goreng

Cara membuat Haluskan bawang merah, bawang putih, cabai merah, dan terasi matang, lalu sisihkan. Panaskan minyak di atas api sedang, lalu masukkan bahan yang telah dihaluskan, tumis hingga harum. Masukkan nasi matang, aduk rata, lalu tambahkan kecap manis dan saus tomat, aduk rata, bagi menjadi 6 bagian. Letakkan 1 bagian nasi goreng di atas sebuah piring, tambahkan sebagian timun, telur ceplok, dan emping goreng di atasnya, lalu taburi dengan sebagian bawang merah goreng di atasnya, ulangi hingga bahan habis.

Programm



Programm

... Wenden Sie das Schnitzel und braten Sie es zwei Minuten von der anderen Seite. Nehmen Sie das Schnitzel aus der Pfanne und blanchieren Sie den Spinat.

Blanchieren?

Programm

Tim!

Wenn Du morgen aufwachst, werden wir schon auf dem Weg in den Urlaub sein. Denk bitte daran, den Rasen zu mähen, wenn die Sonne scheint oder wenn die Nachbarn nicht da sind. Und füttere die Katze!

Dein Vater

Was tun, wenn die Sonne scheint und die Nachbarn nicht da sind?

Programm

Arithmetisches Mittel

Dividieren Sie jede Zahl durch n und addieren Sie die Ergebnisse.

Arithmetisches Mittel

Addieren Sie alle Zahlen und teilen Sie das Ergebnis durch n.

Programm

... heben Sie den Deckel vorsichtig ab. Entfernen Sie die Dichtungsringe. Unter dem Deckel finden Sie drei Sechskantmuttern. Schrauben Sie diese vorsichtig ab. Verwenden Sie dazu unbedingt einen Xfsalö.

Definition Programm

Ein Programm beschreibt einen Ablauf, der sich aus elementaren Schritten zusammensetzt.

Programme

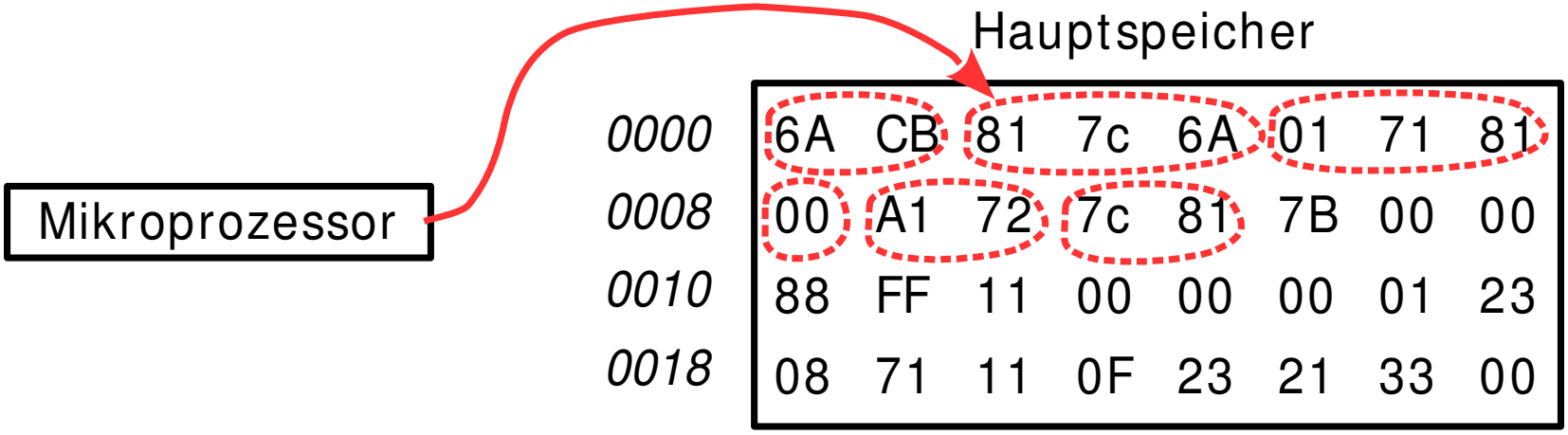
- ✗ Ein Programm ist für einen bestimmten Adressaten bestimmt.
- ✗ Das Programm muss in einer für den Adressaten verständlichen Sprache verfasst sein. Gegebenenfalls ist es notwendig, das Programm in die Sprache des Adressaten zu übersetzen.
- ✗ Dem Adressaten muss bekannt sein, wie die elementaren Schritte auszuführen sind.
- ✗ Das Programm muss frei von syntaktischen Fehlern sein.

Programme

- Das Programm soll eindeutig sein.
- Natürliche Sprachen (Deutsch, Englisch, etc.) bieten Spielräume für Mehrdeutigkeiten und Widersprüche.
- Um das gleiche Ergebnis zu erzielen, gibt es oft mehrere äquivalente Programme. Diese können zu einem unterschiedlich großen Aufwand führen.

1.2 Computerprogramme

Maschinenbefehle



- Die Ausführung von Programmen geschieht im Mikroprozessor.
- Der Mikroprozessor liest die Anweisungen in Form von binären Daten aus dem Hauptspeicher.
- Die Anweisungen, die ein Mikroprozessor bearbeiten kann, werden als Maschinenbefehle bezeichnet.

Ausführung von Maschinenbefehlen

Der Mikroprozessor arbeitet in Zyklen, die als Maschinenzyklen bezeichnet werden. In jedem Maschinenzyklus arbeitet er einen Maschinenbefehl ab.

Arbeitsweise eines Mikroprozessors:

(1) einen Maschinenbefehl aus dem Hauptspeicher laden

(2) aus dem Befehl herauslesen, welche Operation ausgeführt werden soll und welche Daten dafür benötigt werden

(3) die benötigten Daten aus dem Hauptspeicher laden

(4) die Operation ausführen

(5) die Ergebnisse zurück in den Hauptspeicher schreiben

(6) mit dem nächsten Befehl im Hauptspeicher bei (1) weitermachen

Anmerkung: Die Arbeitsweise des Mikroprozessors ist grob vereinfacht dargestellt.

Maschinensprache

- Die Menge aller Maschinenbefehle, die ein Mikroprozessor ausführen kann, wird als die Maschinensprache des Mikroprozessors bezeichnet.

- Es gibt zahlreiche unterschiedliche Mikroprozessoren, bei denen sich die Maschinensprache in der Regel unterscheidet.

- Maschinenbefehle werden in sehr kurzer Zeit ausgeführt (wenige ns). Moderne Mikroprozessoren können viele Milliarden Maschinenbefehle pro Sekunde abarbeiten.

Maschinensprache

- Jeder Maschinenbefehl besteht aus einer kleinen Anzahl von Datenworten, deren Anzahl sich von Befehl zu Befehl unterscheiden kann.
  Typische Werte:
  Datenwortgröße: 64 Bit
  Maschinenbefehl: besteht aus einem oder einigen wenigen Datenworten

- Maschinenbefehle beschreiben sehr einfache Vorgänge. Beispiele:
  - ⇨ zwei Zahlen addieren
  - ⇨ ein Datenwort von einer Stelle im Hauptspeicher an eine andere kopieren

- Üblicherweise steht im ersten Datenwort des Maschinenbefehls der Befehl selbst und in den weiteren Datenworten stehen Operanden.

Maschinensprache

- Um eine Maschinensprache zu erlernen, muss man sich zunächst mit dem groben Aufbau des Mikroprozessors vertraut machen. Die Maschinenbefehle beziehen sich auf Strukturen im Mikroprozessor, die sich von Mikroprozessor zu Mikroprozessor unterscheiden.

- Maschinensprachen enthalten oft eine sehr große Anzahl unterschiedlicher Maschinenbefehle. In der Regel handelt es sich um wenige Grundbefehle, von denen es jeweils zahlreiche Varianten mit unterschiedlichen Adressierungsarten für die Daten gibt.

Programmieren in Maschinensprache

Anwendungen direkt in Maschinensprache zu programmieren ist mühsam, denn

- ✗ Maschinensprachen sind schwer zu erlernen.
- ✗ Maschinensprache ist schwer zu lesen: rein binäre Daten
- ✗ Bei einem Wechsel zu einem anderen Mikroprozessor muss neu programmiert werden.
- ✗ Die Maschinenbefehle realisieren sehr einfache Operationen.
- ✗ Es gibt nur eine sehr einfache Datenstruktur: das Datenwort.

Assembler-Sprachen

Eine kleine Erleichterung bei der Programmierung ergibt sich durch die Verwendung von Assemblersprachen anstelle von Maschinensprachen.

- ✗ Zu jeder Maschinensprache gibt es eine Assemblersprache. Ein einfaches Programm, der Assembler, übersetzt Assemblerprogramme (eine Textdatei) in Maschinensprache (Binärcode).

- ✗ Im Assembler werden Befehle durch Kurzzeichen (Mnemonics) dargestellt, die sich leichter lesen lassen als die Binärzahl.
  Beispiel: ein Addierbefehl wird mit ADD statt mit 0100111000000000 bezeichnet

- ✗ Die Operanden der Befehle können in unterschiedlichen Zahlendarstellungen geschrieben werden (Binär, Dezimal, Hexadezimal etc.).

- ✗ Das Programm kann unabhängig von der Position im Speicher beschrieben werden.

Beispiel

```
pop     eax             ; store EFLAGS in EAX
mov     ebx, eax        ; save in EBX for later testing
xor     eax, 00200000h  ; toggle bit 21
push    eax             ; push to stack
popfd                   ; save changed EAX to EFLAGS
pushfd                  ; push EFLAGS to TOS
pop     eax             ; store EFLAGS in EAX
cmp     eax, ebx        ; see if bit 21 has changed
jz      NO_CPUID        ; if no no
```

Erläuterung: Am Anfang jeder Zeile steht der Maschinenbefehl. Getrennt durch ein Leerzeichen folgen die Operanden des Maschinenbefehls, die untereinander durch Kommata getrennt werden. Hinter dem Semikolon steht ein Kommentar, der bei der Übersetzung in Maschinencode ignoriert wird.
Quelle: AMD, AMD x86-64 Architecture, Programmers Manual

1.3 Höhere Programmiersprachen - Konzept

Problem

Will man eine Anwendung programmieren, so liegt die Aufgabenstellung zunächst in einer natürlichen Sprache wie etwa in Deutsch vor.

- ✗ Maschinensprachen und Assembler sind von der Abstraktionsebene, die man in der natürlichen Sprache verwendet weit entfernt.

- ✗ Programmieren bedeutet, dass man aus der Aufgabenstellung in natürlicher Sprache ein auf dem Computer ausführbares Programm in Maschinensprache erstellt.

Konzept: Höhere Programmiersprache

- Man programmiert die Anwendung nicht direkt in Maschinensprache oder Assembler, sondern beschreibt das Programm in einer so genannten „höheren Programmiersprache“.

- Mit höhere Programmiersprachen können Programme auf einer Abstraktionsebene formuliert werden, die der natürlichen Sprache näher kommt.

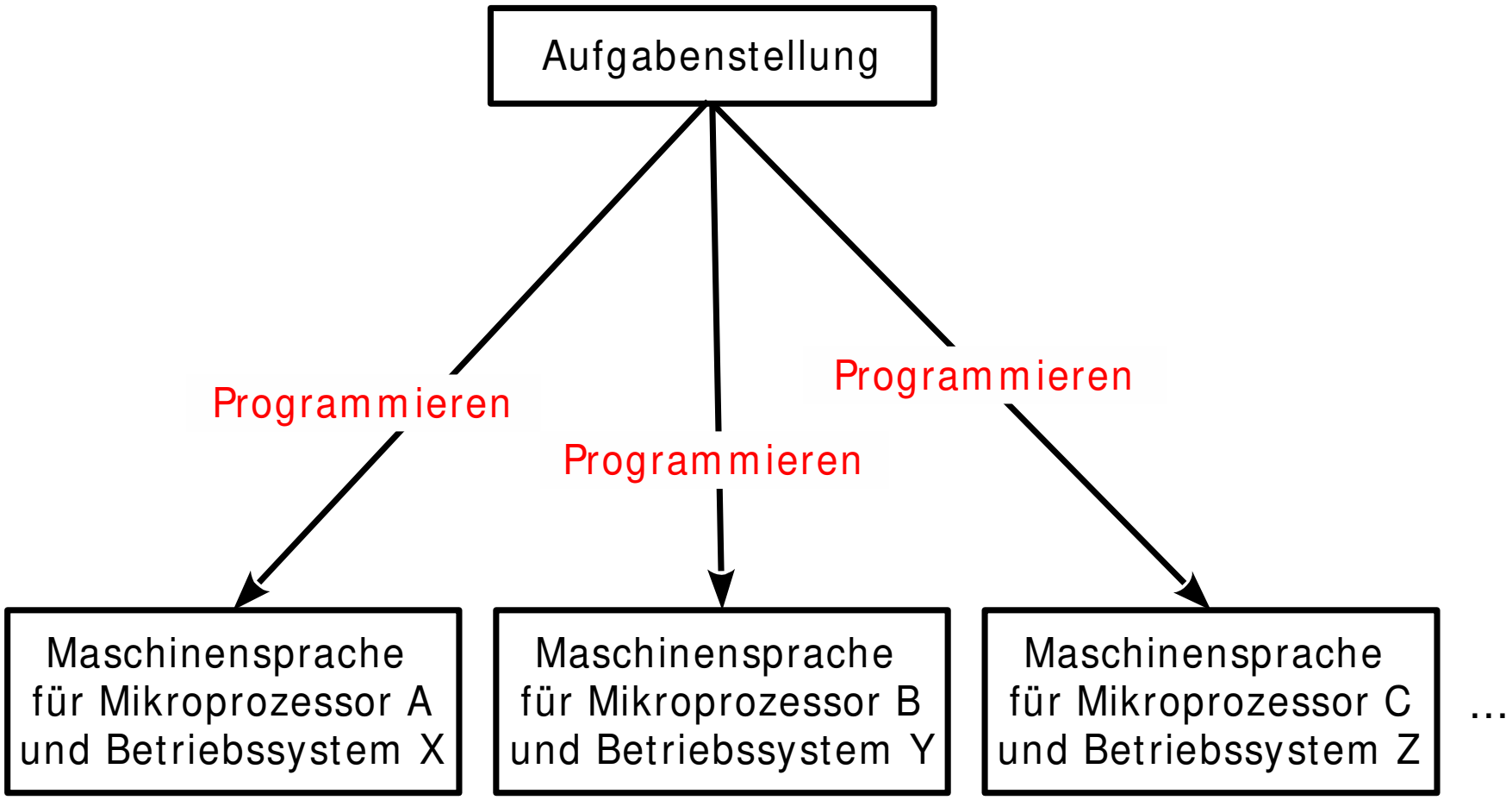
- Zu der höheren Programmiersprache gibt es Übersetzungsprogramme, die das Programm in Maschinensprache konvertieren. Diese Übersetzungsprogramme bezeichnet man als Compiler.

Vorteil höherer Programmiersprachen

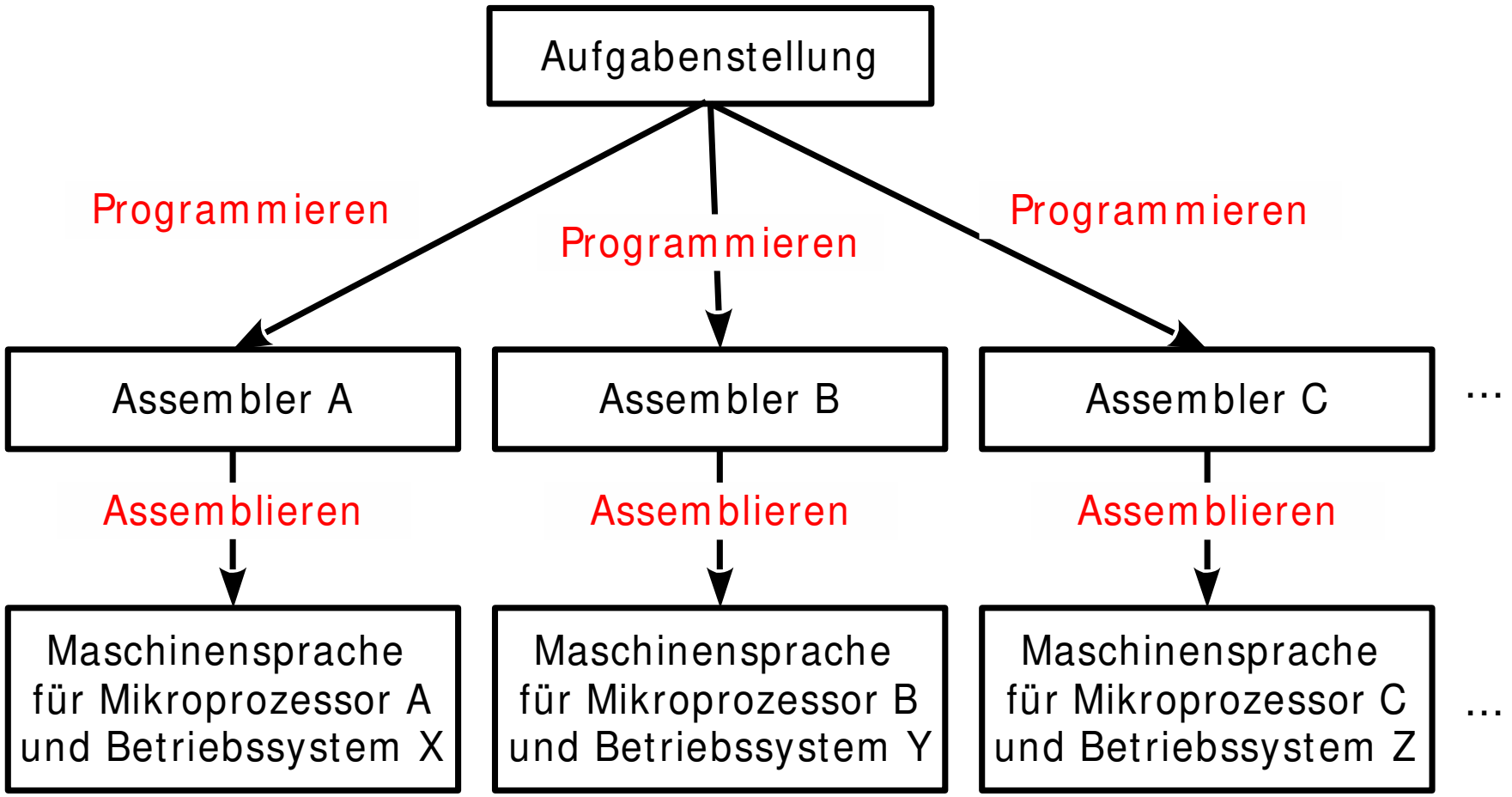
- In der Regel gibt es zu einer Programmiersprache verschiedene Compiler, die für bestimmte Mikroprozessoren und für bestimmte Betriebssysteme bestimmt sind. Hat man einmal ein Programm in einer höheren Programmiersprache geschrieben, so kann man dieses Programm auf unterschiedlichen Mikroprozessoren und Betriebssystemen ausführen lassen. Man benötigt dann für den jeweiligen Mikroprozessor und das jeweilige Betriebssystem den passenden Compiler.

- Auf diese Weise macht man sich bei der Programmierung unabhängig von Mikroprozessoren und Betriebssystemen. Die Programmierung von Software ist oft mit erheblichen Kosten verbunden. Ziel ist es, diese Investitionen nachhaltig zu sichern. Das einmal geschriebene Programm soll „zukunftssicher“ sein und auf allen aktuellen und zukünftigen Computersystemen ausgeführt werden können („write once, run everywhere“).

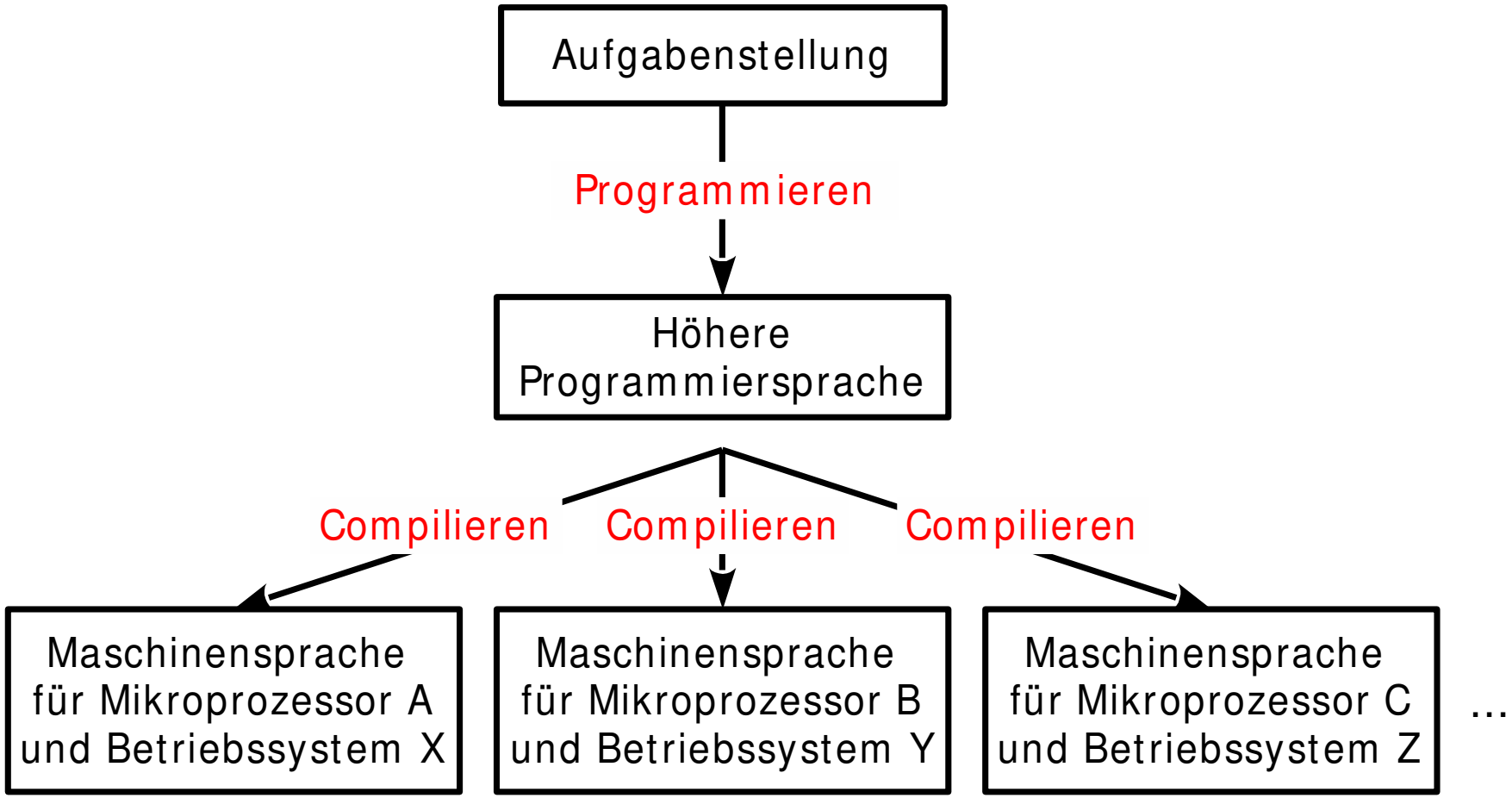
Programmieren mit Maschinensprachen



Programmieren mit Assembler



Programmieren mit höheren Programmiersprachen



1.4 Höhere Programmiersprachen - Überblick

Höhere Programmiersprachen

- Es wurden seit den 60er Jahren über 2000 verschiedene Programmiersprachen entwickelt und jeweils Compiler dafür implementiert.

- Programmiersprachen wurden seit den 60er Jahren zunehmend verbessert unter dem Aspekt, dass heute Programme in einer zunehmend abstrakten, mathematischen Denkweise beschrieben werden können - weg von der Nähe zur Maschinensprache.

- Messlatte für moderne Programmiersprachen:

  Der Programmcode großer, komplexer Anwendungen
  muss möglichst schnell und kostengünstig
  implementiert und gewartet werden können!

Klassifikation von Programmiersprachen nach Programmierkonzept

- ✗ imperative Programmiersprachen (befehlsorientiert)
  - ⇨ nicht objektorientiert
    Fortran, Algol, Basic, PL/I, Cobol, Pascal, C, Modula-2, ADA
  - ⇨ objektorientiert
    Smalltalk, C++, Java, Ruby, Python, Ruby
- ✗ funktionale Programmiersprachen
  Lisp, Miranda, ML
- ✗ prädikatenlogische Programmiersprachen
  Prolog

Heute dominieren imperative, objektorientierte Programmiersprachen und zunehmend enthalten diese auch funktionale Anteile

Klassifikation nach Anwendungsgebiet

Oft wurden Programmiersprachen für ganz bestimmte Anwendungsgebiete konzipiert oder sie haben sich vor allem in bestimmten Anwendungsbereichen etabliert:

- Assemblernahe Programmiersprachen: Betriebssysteme, Treiber
  C
- Mathematisch-Naturwissenschaftliche Anwendungen
  Fortran, Algol
- Betriebswirtschaftliche Anwendungen
  Cobol, PL/I

Moderne Programmiersprachen sind in der Regel „universelle Programmiersprachen“ und sind nicht auf bestimmte Anwendungsgebiete beschränkt.

Typisierte und untypisierte Programmiersprachen

Maschinensprachen kennen nur einen einzigen Datentyp: das Datenwort des Mikroprozessors - also in der Regel 64 Bit. In höheren Programmiersprachen können bei weitem vielfältigere und komplexere Datenstrukturen mit unterschiedlicher Größe ausgedrückt werden: Zahlen, Zeichenketten, Listen, Baumstrukturen, etc.

Man unterscheidet Programmiersprachen danach, wie sie mit Datenstrukturen umgehen:

- ✗ untypisiert: Den Daten wird nicht unbedingt ein bestimmter Typ zugeordnet, oder es wird nicht strikt darauf geachtet, dass immer der passende Datentyp verwendet wird.
  C, Lisp
- ✗ typisiert, streng typisiert: Alle Daten haben einen Typ, und es wird durch die Programmiersprache ausgeschlossen, dass ein falscher Typ verwendet wird.
  Pascal, Miranda, ML, Prolog, Modula-2, Java, Python, Ruby

1.5 Die Programmiersprache Java

Entstehung von Java

- 1995 entwickelt von Sun Microsystems (heute Oracle)
- Einordnung:
  - ⇨ imperativ
  - ⇨ objektorientiert
  - ⇨ streng typisiert
  - ⇨ mit einfachen funktionalen Elementen
- In diesem Vorlesung verwenden wir Java in der Version 8.

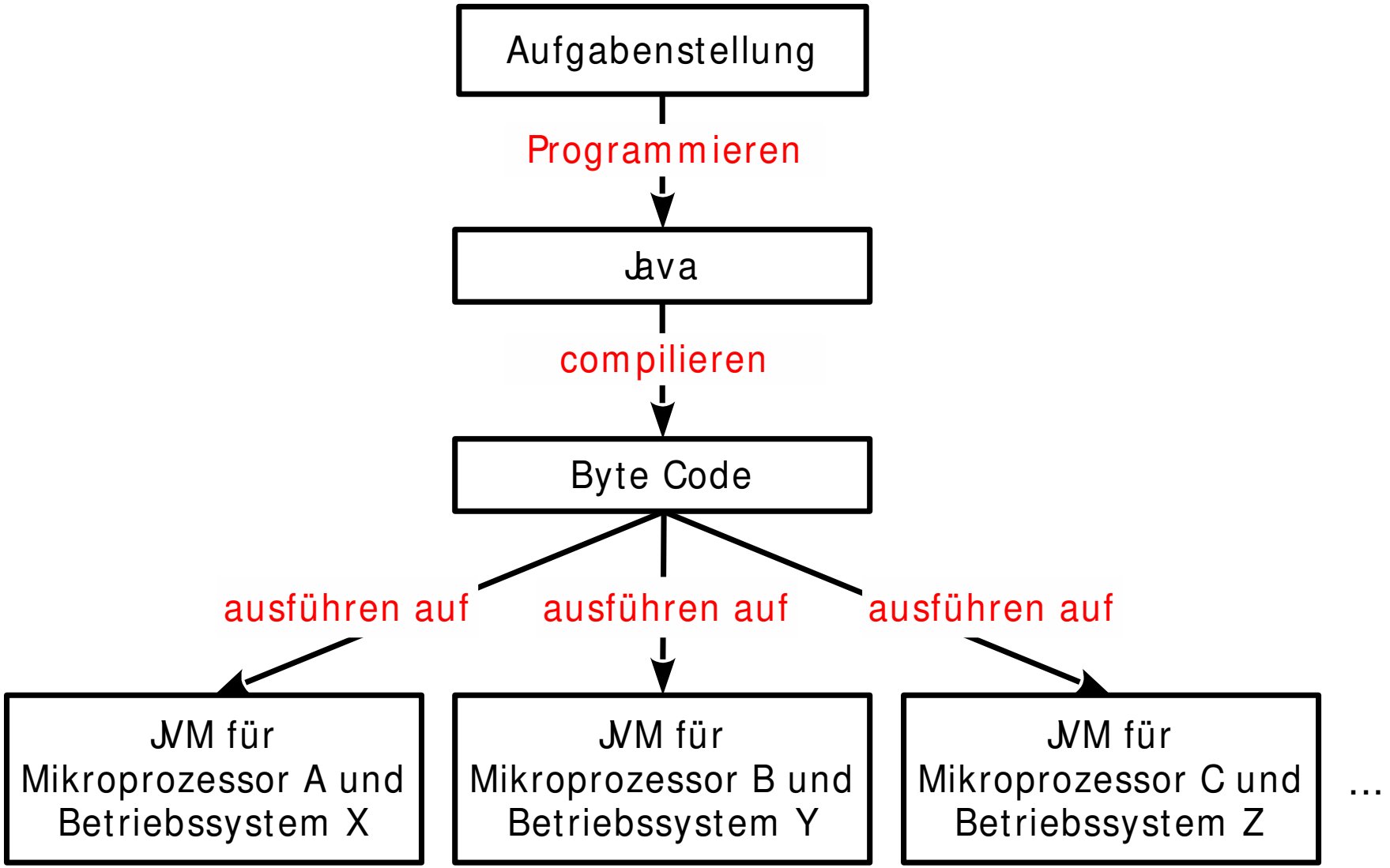
Java Virtual Machine und Java Bytecode

- Die Java-Programmiersprache wird vom Compiler nicht sofort in eine konkrete Maschinensprache, sondern in einen so genannten „Java Bytecode“ übersetzt.

- Java Bytecode ähnelt einer Maschinensprache. Java Bytecode wird jedoch nicht auf einem bestimmten Mikroprozessor und auf einem bestimmten Betriebssystem ausgeführt, sondern auf einer so genannten Java Virtual Machine (JVM).

- Die JVM ist ein Programm, das es für alle gängigen Mikroprozessoren und Betriebssyteme gibt. Die JVM stellt ein virtuelles Betriebssystem mit einem virtuellen Prozessor dar.

Java, Java Byte Code, JVM



Software Development Kit (SDK)

- Compiler und JVMs für Java werden zusammen als SDK (software development kit) bezeichnet.
- Es gibt verschiedene Anbieter, die diese für alle gängigen Betriebssystem anbieten.
- Die SDKs können frei im Internet heruntergeladen werden.
- Java Compiler und JVM werden über die Kommandokonsole des Betriebssystems gestartet. Die Entwicklung von Programmen mit dem SDK ist wenig komfortabel.

Entwicklungsumgebungen für Java

- Entwicklungsumgebungen sind komplexe Werkzeuge zur effizienten und komfortablen Programmentwicklung. Sie bieten zahlreiche Funktionen, die über das reine Compilieren hinausgehen
  - ⇨ Editoren
  - ⇨ Debugging (Schritt für Schritt durch Programme gehen)
  - ⇨ Profiling (Leistungsuntersuchung des Codes)
  - ⇨ Versionierung des Codes
  - ⇨ ...
- Es gibt für Java zahlreiche populäre Entwicklungsumgebungen. Beispiele:
  - ⇨ Eclipse (IBM)
  - ⇨ NetBeans (Oracle)
  - ⇨ IntelliJ IDEA (Open Source)
  - ⇨ ...

Eclipse

- In den Praktika verwenden wir die Entwicklungsumgebung Eclipse.
- Eclipse ist für alle gängigen Betriebssysteme frei erhältlich.

Kapitel 2

Imperatives Programmieren

2.1 Sequenzen von Anweisungen

Ein erstes Programm

```
public class ErstesProgramm {
    public static void main(String[] args) {
        System.out.println("Herzlich willkommen zum Praktikum!");
        System.out.println("Eine Zufallszahl: " + Math.random());
        System.out.println("Ende des Programms.");
    }
}
```

- Die Anweisungen befinden sich in einer Methode mit dem Namen main.
- Die main-Methode befindet sich in einer Klasse mit einem vom Benutzer festgelegten Namen (hier: ErstesProgramm).
- Der gesamte Text befindet sich in einer Textdatei mit dem Namen ErstesProgramm.java.

... Ein erstes Programm

```
System.out.println("Herzlich willkommen zum Praktikum!");
System.out.println("Eine Zufallszahl: " + Math.random());
System.out.println("Ende des Programms.");
```

- Die Anweisung werden in der angegebenen Reihenfolge hintereinander ausgeführt.
- Jede Anweisung endet mit einem Semikolon. Das Semikolon dient als Trennzeichen zwischen den Anweisungen.

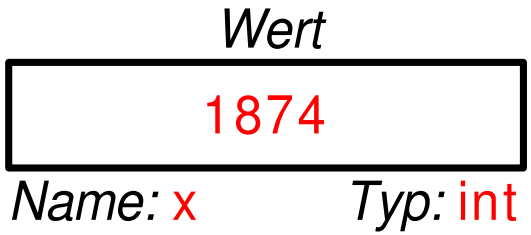
Methoden, Klassenmethoden

```
public class ErstesProgramm {
    public static void main(String[] args) {
        System.out.println("Herzlich willkommen zum Praktikum!");
        System.out.println("Eine Zufallszahl: " + Math.random());
        System.out.println("Ende des Programms.");
    }
}
```

- Da die Methode main das Schlüsselwort static hat, bezeichnet man sie auch genauer als Klassenmethode - eine der Klasse zugeordnete Methode.
- In diesem Kapitel werden nur Klassenmethoden betrachtet.
- Später wird der Begriff Klassenmethode gezielt zur Abgrenzung zu Objektmethoden verwendet (ohne static).

2.2 Variablen

Variablen



- Variablen dienen der Aufbewahrung von Daten.
- Variablen haben
  - ⇨ einen Wert
  - ⇨ einen Namen
  - ⇨ einen Typ

Variablen

- In einer imperativen Programmiersprache wie Java dient eine Variable als Speicherplatz.
- In der Variable wird ein Wert gespeichert. Der Wert kann sich während des Programmablaufs ändern.
- Der Name dient der eindeutigen Bezeichnung. Im Programmcode wird auf die Variablen mit ihrem Namen Bezug genommen.
- Jede Variable hat einen vorgegebenen Typ: nur Werte von diesem Typ können in der Variablen gespeichert werden.
- Jeder Variablen entspricht ein Bereich im Hauptspeicher, in dem der Variablenwert abgespeichert wird.

Variablen-Deklarationen

```
int x;
```

- Eine Variablen-Deklaration definiert eine neue Variable durch
  - ⇨ einen Namen
    hier: x
  - ⇨ einen Typ
    hier: int
- Fortan kann diese Variable im Programmcode verwendet werden.
- Alle Variablen, die im Programm verwendet werden sollen, müssen zuvor deklariert werden.

Zuweisungen

```
int x;
x = 3;
int y;
y = 3*x+4;
```

- Eine Zuweisung ist eine Anweisung, die den Wert einer Variablen ändert.
- Die Variable, deren Wert geändert werden soll, steht links vom Gleichheitszeichen.
- Durch die Ausführung der Zuweisung nimmt die Variable den Wert an, der sich aus dem Ausdruck rechts des Gleichheitszeichens ergibt.
- Der Wert, den man der Variablen zuweist, muss von dem Typ sein, der in der Variablendeklaration festgelegt wurde.

Zuweisungen

```
x = 3;
```

- Im einfachsten Fall ist der Ausdruck auf der rechten Seite einfach eine Konstante (der Wert selbst).

```
y = 3*x+4;
```

- Alternativ kann sich der Ausdruck auf der rechten Seite auch aus einer oder mehreren Operatoren (hier: +, *), Variablen (hier: x) und Konstanten (hier: 3, 4) zusammensetzen. In diesem Fall wird bei der Ausführung der Zuweisung zunächst der Ausdruck ausgewertet (d.h. der Wert berechnet) und dann wir das Ergebnis dieser Berechnung der Variablen zugewiesen.

Zuweisungen

```
x = x+1;
```

- Die Variable, der ein Wert zugewiesen wird, darf auch im Ausdruck auf der rechten Seite vorkommen. In diesem Fall bezieht sich die Variable auf der rechten Seite auf den Wert vor Ausführung der Zuweisung: Zuerst wird der Ausdruck auf der rechten Seite berechnet, dann wird er der Variablen x zugewiesen.

- In obigem Beispielcode wird x um 1 erhöht.

Anmerkung: Für den speziellen Fall, dass eine Variable um eins erhöht werden soll, gibt es eine verkürzte Schreibweise, die oft verwendet wird:

```
x++;
```

Initialisierung

Variablen sollen immer einen definierten Wert haben. Aus diesem Grund wird den Variablen gern unmittelbar nach der Deklaration ein Anfangswert zugewiesen. Eine solche Zuweisung wird als Initialisierung bezeichnet.

Man kann die Deklaration und die Initialisierung in Java in einen Befehl zusammenfassen.

```
int x = 3;
```

ist gleichbedeutend mit

```
int x;
x = 3;
```

Einfache Datentypen

Typ	Bedeutung	Werte	Speicher-bedarf
byte	ganze Zahlen in Zweierkomplement-darstellung	-128,... 127	8 Bit
short		-32768, ... 32767	16 Bit
int		-2147483648, ... 2147483647	32 Bit
long		-9223372036854775808, ... 9223372036854775807	64 Bit
float	Fließkommazahlen	$-3.40282 \cdot 10^{38}$, ... $3.40282 \cdot 10^{38}$ Mantisse: 8 Dezimalstellen Exponent: -38, … 38	32 Bit
double		-1.79769e+308, ... 1.79769e+308 Mantisse: 15 Dezimalstellen Exponent: -308, … 308	64 Bit
char	Buchstaben in Unicode	´a´, ´A´, ´b´, ´B´, ... ´$´, ´?´, ...	32 Bit
boolean	Boolescher Wert	true, false	1 Bit

Operationen für einfachen Datentypen

Arithmetische Operationen

+ * - /	Grundrechenarten für byte, short, int, long, float, double
a ^ b	Potenz a^b
a / b	Ganzzahlige Division ohne Rest
a % b	Rest der Division von a/b

Vergleiche: zwei Zahlen werden verglichen, Ergebnis vom Typ boolean

< > <= >=	kleiner als, größer als, kleiner gleich, größer gleich
==	Gleichheit
!=	Ungleichheit

… Operationen für einfachen Datentypen

Boolesche Operationen

a && b	logisches Und: a und b
a || b	logische Oder: a oder b
! a	Negation: nicht

Strings

Strings sind Zeichenketten, die sich aus mehreren Zeichen zusammensetzen.

- ✗ String-Konstanten werden in Anführungszeichen eingeschlossen.
- ✗ Strings können durch + miteinander verbunden werden.
- ✗ Strings können mit einfachen Datentypen durch + zu einem großen langen String verbunden werden. Dabei werden die einfachen Datentypen zunächst in Strings umgewandelt.
- ✗ Strings können mit dem Befehlt System.out.println(x) ausgegeben werden.

```
String vorname = “Oliver“;
int alter = 21;
System.out.println(“Vorname: “ + vorname + “  Alter: “ + alter);
```

Einfache Datentypen in Strings umwandeln

„Addiert“ man einen String mit einem einfachen Datentyp, so entsteht ein String. Um einen einfachen Datentyp in einen String zu konvertieren, addiert man einfach den leeren String ““.

```
int x = 257;
String s = ““ + x;
```

Strings in Zahlen umwandeln

Zur Konvertierung von Strings in Zahlen gibt es parse-Funktionen:

```
String s1 = "117";
byte b = Byte.parseByte(s1);
short s = Short.parseShort(s1);
int i = Integer.parseInt(s1);
long l = Long.parseLong(s1);

String s2 = "1.23";
float r = Float.parseFloat(s2);
double d = Double.parseDouble(s2);
```

Arrays

Ein Array ist eine Variable, die sich aus n einzelnen Variablen gleichen Typs zusammensetzt.

```
int[] x = {5, 6, 7};
x[0] = 4;
x[2] = x[0] + x[1];
int i=1;
x[i] = x[i+1] + 2;
```

- Ist x eine Array-Variable, so sind x[0], ... x[n-1] die einzelnen Variablen des Arrays.
- In einem Ausdruck der Form x[i] wird i als Index bezeichnet.

Initialisierung von Arrays

- Durch die Initialisierung erhält ein Array eine Länge und jeder Einzelvariablen wird ein Wert zugewiesen. Eine Array-Variable kann man auf zweierlei Weise initialisieren:

```
int[] x = {3, 8, 17, 25};
```

```
int[] x = new int [552];
```

- Im ersten Fall soll der Array initial aus den Zahlenwerten bestehen, die der Programmierer explizit aufzählt. In dem Beispielcode soll der Array vier Speicherplätze haben, die initial mit die Werte 3, 8, 17 und 25 haben sollen.

- Im zweiten Fall legt der Programmierer fest, dass der Array aus einer vorgegebenen Anzahl von Zahlen bestehen soll. Die Inhalte des einzelnen Variablen sollen alle einen Defaultwert haben (bei int-Werten ist der Defaultwert 0). Somit erzeugt der Beispielcode einen Array aus 552 Nullen.

Zwei Schreibweisen bei der Variablendeklaration

Kurioserweise gibt es in Java zwei unterschiedliche Schreibweisen bei der Variablen, die einen Array enthält.

Die Schreibweise

```
int x[];
```

ist äquivalent zu

```
int[] x;
```

In beiden Fällen wird *x* als eine Variable deklariert, die einen Array von int-Werten enthält. Ob das eckige Klammernpaar [] vor oder hinter dem Variablennamen steht, macht in Java keinen Unterschied.

Arrays haben eine feste Länge

- ✗ Sobald ein Array einmal erzeugt wurde, hat er eine feste Länge. Die Inhalte der Einzelvariablen können danach verändert werden, die Länge jedoch nicht.

- ✗ Die Länge des Arrays kann im Programm mit x.length() bestimmt werden. x.length() ist eine Zahl vom Typ int.

- ✗ Es dürfen im Programm nur die Indizes angesprochen, die kleiner als die Länge des Arrays sind. Ansonsten kommt es zu einem Fehler.
  - ⇨ Hat ein Array *x* also beispielsweise die Länge 7, so besteht er aus den Einzelvariablen x[0], x[1], x[2], x[3], x[4], x[5] und x[6]. Der Aufruf von x[8] oder der Aufruf von x[-3] würde jeweils zu einem Fehler führen.

Arrays unterschiedlicher Typen

- Arrays gibt es von beliebigen Datentypen. Hier ein Beispiel für einen String-Array:

```
String[] s = {"Auto", "Katze", "Hammer"};
```

2.3 Ein- und Ausgabe von Daten

Ein- und Ausgabe

Es gibt zahlreiche Möglichkeiten, wie einem Programm Daten zugeführt werden können (Eingabe) und wie das Programm diese ausgeben kann (Ausgabe).

Beispiel:

- ✗ Eingabe über Tastatur
- ✗ Ausgabe über Textbildschirm
- ✗ Ausgabe auf Drucker
- ✗ Eingabedaten aus Videokamera
- ✗ Ein- und Ausgabe über graphische Oberfläche (Fenster, Button, Menüs, ...).
- ✗ Eingabedaten aus Datei, Ausgabe der Daten in eine Datei
- ✗ Eingabedaten aus einem Netzwerk (z.B. Internet) - Ausgabe der Daten über ein Netzwerk
- ✗ ...

Ein- und Ausgabe über die Konsole

- Es gibt in Java verschiedene Programmbibliotheken für die Ein- und Ausgabe von Daten, mit deren Hilfe ein Programm auf sehr unterschiedliche Weise mit der Außenwelt kommunizieren kann.

- Der Einfachheit halber beschränkt sich diese Vorlesung auf eine Ein- und Ausgabe über die Konsole. Ein- und Ausgabe-Operationen sind textbasiert und erfolgen alleine mit Hilfe des Monitors und der Tastatur.

Eingabebefehl readString()

- Der Befehl *readString()* fordert den Benutzer auf einen String einzugeben. Der Befehl wartet solange, bis der Benutzer eine Zeichenkette eingegeben und diesen String mit der Return-Taste abgeschlossen hat.

- Das Ergebnis dieses Befehls ist eine Zeichenkette. Die Zeichenkette enthält die vom Benutzer eingegebenen Zeichen – ohne das Return-Zeichen.

readString() mit Prompt

- Der Befehl *readString()* wartet, bis der Anwender etwas eingegeben hat. Damit der Benutzer versteht, was er einzugeben hat, kann man dem Befehl *readString()* einen String als Parameter mitgeben. Dieser String wird dann ausgegeben, bevor auf die Eingabe des Benutzers gewartet wird.

- Möchte man beispielsweise, dass der Benutzer einen Namen eingibt, so kann man *readString("Name")* aufrufen. Auf dem Bildschirm erscheint dann
  Name:
  Hinter dem Doppelpunkt wartet der Cursor jetzt auf eine Eingabe des Benutzers.

Beispiele;

```
String s = readString();
String name = readString("Name");
```

Eingabebefehle readInt() und readDouble()

- Die Befehle *readInt* und *readDouble* dienen der Eingabe von Int- beziehungsweise von double-Werten.
- Die Funktionsweise entspricht weitgehend der des Befehls *readString*. Die Befehle *readInt* und *readDouble* lesen zunächst eine Textzeile ein, die der Benutzer eingibt. Dieser String wird dann automatisch in einen int- beziehungsweise in einen double-Wert konvertiert.
- Auch bei den Befehlen *readInt* und *readDouble* kann jeweils ein Prompt angegeben werden.

Beispiele;

```
int x = readInt("x");
double y = readDouble("y");
```

Eingabebefehle readIntArray() und readDoubleArray()

- Die Befehle *readIntArray* und *readDoubleArray* dienen der Eingabe von jeweils mehreren Int- beziehungsweise von double-Werten. Sie funktionieren wir die Befehle *readInt* und *readDouble* mit dem Unterschied, dass der Anwender mit jeder Eingabe nicht nur eine, sondern mehrere Zahlen angeben kann. Die Zahlen tippt der Anwender durch Kommata getrennt in eine Textzeile ein. Ergebnis ist dann jeweils ein Array von int- beziehungsweise double-Werten.

Beispiele;

```
int[] x = readIntArray(“x“);
double[] y = readDoubleArray(“y“);
```

Ausgabebefehle System.out.print und System.out.println

- Der Befehlt *System.out.print* schreibt einen String auf einen Textbildschirm. Mehrere aufeinanderfolgende Aufrufe dieses Befehls führen dazu, dass die Ausgaben hintereinander auf der Konsole erscheinen.

- Der Befehl *System.out.println* funktioniert im Prinzip genau so wie der Befehl *System.out.print*, mit dem Unterschied, dass nach der Ausgabe automatisch ein Zeilenumbruch (Return) durchgeführt wird.

  ⇨ Anmerkung: *println* steht für *print line*

Beispielprogramm Multiplizierer

```
import static prog.ConsoleReader.*;

public class Multiplizierer {

    public static void main(String[] args) {
        int x = readInt("x");
        int y = readInt("y");
        int z = x * y;
        System.out.println(x + " multipliziert mit " + y + " ergibt " + z);
    }
}
```

Beispielablauf des Programms

```
x: 12
y: 4
12 multipliziert mit 4 ergibt 48
```

Die vom Benutzer eingetippten Zeichen sind unterstrichen dargestellt. Die restlichen Zeichen hat das Programm ausgegeben.

Anmerkungen zu readString, readInt,...

- Anders als *System.out.println()* und *System.out.print()* sind die Befehle *readString(), readInt(), readDouble(), readIntArray()* und *readDoubleArray()* nicht standardmäßig in der Programmiersprache Java enthalten.

- Damit die Befehle *readString(), readInt(), readDouble(), readIntArray()* und *readDoubleArray()* verwendet werden können, sind vorab zwei Schritte erforderlich:
  - ⇨ Die Bibliothek *prog.jar* muss eingebunden werden. Wie dies geht, erfahren Sie im Praktikum.
  - ⇨ Vor dem Beginn der Klassendeklaration benötigt man die folgende Textzeile (siehe Programmcode von Multiplizierer).

```
import static prog.ConsoleReader.*;
```

  - ⇨ Aus Platzgründen und wegen der besseren Übersichtlichkeit wird diese import-Zeile in den nachfolgenden Beispielen bisweilen weggelassen.

2.4 Kommentare

Kommentare

- Kommentare dienen der Dokumentation des Programms.
- Kommentare können vom Programmierer beliebigen Stellen in den Programmtext eingefügt werden.
- Kommentare werden beim Compilieren ignoriert und haben keinen Einfluss auf das übersetzte Programm.
- Was vernünftigerweise in Kommentaren stehen kann:
  - ⇨ Was macht das Programm?
  - ⇨ Wer hat dieses Programm geschrieben?
  - ⇨ Wieso wurde das hier so programmiert?
  - ⇨ Welche Bedeutung hat diese Variable?
  - ⇨ ...

Kommentare

- Das Zeichen // legen fest, dass der Rest der Zeile ein Kommentar ist.
- Mehrzeilige Kommentare werden mit /* und */ geklammert

```
/* Programm zur Multiplikation zweier Zahlen.
   Eingabewerte über zwei Programmparameter.
   Autor: Dirk Eisenbiegler  */
public class Multiplizierer {
    public static void main(String[] args) {
        int x = readInt("x");
        int y = readInt("y");
        int z = x * y;  // hier findet die Multiplikation statt!
        System.out.println(x + " mal " + y + " ergibt " + z + ".");
    }
}
```

Tipp: Auskommentieren

- Programmcode, den man vorübergehend nicht braucht, den man aber zunächst noch nicht löschen möchte, kann man mit /* und */ „auskommentieren“.
- Der Programmcode wird dann bei der Übersetzung und bei der Ausführung ignoriert, da er als Kommentar betrachtet wird.
- Entfernt man die Kommentarklammern /* und */ wieder, dann wird der Programmcode wieder "aktiviert".

```
int[] x = {5, 6, 7};
/*
x[0] = 4;
x[2] = x[0] + x[1];
*/
int i=1;
```

2.5 Einfache Kontrollstrukturen

Kontrollstrukturen

- Bisher war der Ablauf der Programme (der Kontrollfluss) sehr einfach:
  - ⇨ Die Anweisungen stehen durch Semikolon getrennt untereinander.
  - ⇨ Das Programm durchläuft die Anweisungen von oben nach unten.
  - ⇨ Jede Anweisung wird genau einmal ausgeführt.

- Mit Hilfe von Kontrollstrukturen lassen sich komplexere Abläufe beschreiben:
  - ⇨ Der Kontrollfluss führt nicht nur von oben nach unten.
  - ⇨ Die im Programm enthaltenen Anweisungen werden mehrfach oder gar nicht ausgeführt.
  - ⇨ Der Kontrollfluss kann von Variablenwerten abhängig gemacht werden.

if

Mit der Kontrollstruktur *if* wird ein Programmteil nur dann ausgeführt, wenn eine Bedingung erfüllt ist.

```
if (x<3)
    System.out.println("Vorsicht, x ist kleiner als 3.");
```

Bezieht sich die Bedingung nicht nur auf einen Befehl, sondern auf mehrere, so werden diese mit Mengenklammern eingefasst.

```
if (x>y) {
    int z = y;
    y = x;
    x = z;
}
```

if-else

Mit der Kontrollstruktur if-else wird der Programmteil nach dem if nur dann ausgeführt, wenn eine Bedingung erfüllt ist und ansonsten der Programmteil nach dem else.

```
public class Dividierer {
    public static void main(String[] args) {
        int x = readInt("x");
        int y = readInt("y");
        if (y == 0) {
            System.out.println("Es kann nicht durch 0 geteilt werden.");
        } else {
            int z = x / y;
            System.out.println(x + " durch " + y + " ergibt " + z + ".");
        }
    }
}
```

while

Mit der Kontrollstruktur *while* wird ein Programmteil solange ausgeführt, wie die Bedingung erfüllt ist.

```
int x = 1;
while (x < 10000) {
    System.out.println("x : " + x);
     x = x*2;
}
```

for

Mit der Kontrollstruktur *for* wird der Rumpf der for-Schleife mehrfach ausgeführt. Dabei durchläuft eine Schleifenvariable einen bestimmten Bereich.
Im Beispiel wird der Rumpf zehnmal ausgeführt, wobei die Schleifenvariable i beim ersten Durchlauf den Wert 0, beim zweiten Durchlauf den Wert 1 ... und beim letzten Durchlauf den Wert 9 annimmt.

```
int sum = 0;
for (int i=0;  i<10;  i++)
    sum = sum + i;
```

- ✗ `int i=0;` Der erste Abschnitt zwischen den runden Klammern der for-Anweisung dient der Deklaration und Initialisierung der Schleifenvariablen.
- ✗ `i<10;` Der zweite Abschnitt definiert eine Schleifenbedingung. Solange diese erfüllt ist, soll die Ausführung des Rumpfes wiederholt werden.
- ✗ `i++` Der dritte Abschnitt verändert die Schleifenvariable nach jedem Durchlauf.

2.6 Klassenmethoden

Klassenmethoden

Mit Klassenmethoden werden mehrere Anweisungen zu einer größeren Anweisung zusammengefasst.

```
public class Test {

    public static void main(String[] args) {
        System.out.println("Hallo!");
        liesUndQuadriere("a");
        liesUndQuadriere("b");
        liesUndQuadriere("c");
        System.out.println("Ende!");
    }

    public static void liesUndQuadriere(String v) {
        int n = readInt(v);
        System.out.println("Die Quadratzahl von " + v + " ist " + (n*n));
    }
}
```

main und andere Klassenmethoden

- Eine Klasse, die ausgeführt werden soll, muss eine main-Klassenmethode enthalten.

```
public static void main(String[] args) {
    ...
}
```

- Die main-Klassenmethode wird beim Start der Klasse ausgeführt.
  Aufruf: java Klassenname

- Es dürfen in einer Klasse weitere Klassenmethoden mit beliebigem Namen deklariert werden.

  ⇨ In diesem Beispiel die Methode *liesUndQuadriere*

- Von jeder Klassenmethode aus können andere Klassenmethoden aufgerufen werden.

- Klassenmethoden dürfen auch sich selbst aufrufen (Rekursion).

Methoden-Deklaration und Methoden-Aufruf

- ✗ Eine Methode muss deklariert werden, damit sie verwendet werden kann.
- ✗ Methoden können Parameter haben.
  - ⇨ Die Parameter werden der Methode beim Aufruf übergeben.
  - ⇨ Sie müssen genau die Typen haben, die in der Deklaration der Methode vorgegeben wurden.

Warum Methoden?

- ✗ Wiederverwendung:
  Programmstücke mit gleichem oder ähnlichem Inhalt werden nur einmal geschrieben und können mehrfach verwendet werden.
  - ⇨ kompakterer Programmcode
    Das Programmstück erscheint im Text nur einmal statt mehrfach.
  - ⇨ leichtere Wartung
    Änderungen an diesem Programmcode nur an einer zentralen Stelle.

- ✗ Strukturierung:
  Dadurch, dass man Programmteile in sinnvoller Weise zu Methoden zusammenfasst und man diesen Programmteilen einen Namen gibt (den Namen der Methode), wird der Programmcode besser strukturiert und leichter lesbar.

Rückgabewerte

```
public class Test{
    public static void main(String[] args) {
        int x = readInt("x");
        int q = average(x,2) + 3;
        System.out.println(q);
    }
    public static int average(int a, int b) {
        int sum = a+b;
        return sum/2;
    }
}
```

- Methoden können Werte zurückgeben. Der Typ des Rückgabewertes wird in der Methoden-Deklaration festgelegt (statt void).
- Der Wert wird mit dem Befehl *return* an den Aufrufer zurückgegeben.

Rückgabewerte

- Beispiele: Eine Methode ohne Rückgabewert und eine Methode mit einem Rückgabewert vom Typ int.

```
public static void main(String[] args) {
```

```
public static int average(int a, int b) {
```

- Durch den Aufruf des Befehls *return* endet die Methode sofort. Nachfolgende Befehle werden, so vorhanden, nicht mehr ausgeführt.

Rückgabewerte

- Methoden mit Rückgabewert dürfen
  - ⇨ wie eine Funktion in einem Ausdruck verwendet werden

```
int q = average(x,2) + 3;
```

  - ⇨ wie eine eigene Anweisung verwendet werden

```
average(x,2);
```

- Werden Methoden mit Rückgabewert im Stil einer Anweisung aufgerufen, so wird der Rückgabewert ignoriert.

- Methoden ohne Rückgabewert dürfen nur im Stil von Anweisungen verwendet werden.

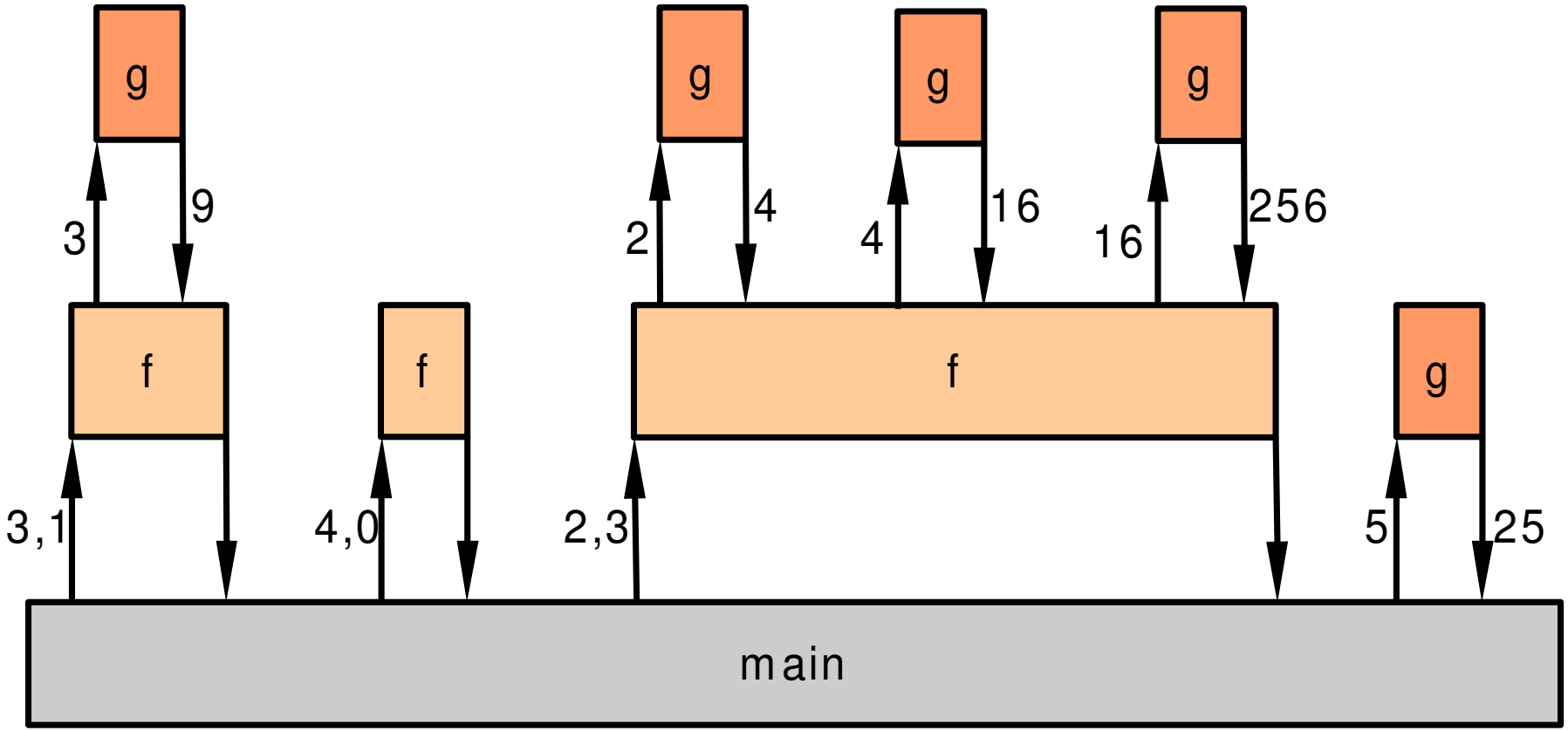
Geschachtelte Aufrufe von Methoden

```
public static void main (String[] args) {
        f(3,1);
        f(4,0);
        f(2,3);
        System.out.println(g(5));
}

public static void f (int a, int n) {
        int x=a;
        for (int i=0; i<n; i++) x=g(x);
        System.out.println(x);
}

public static int g (int x) {
        return x *x;
}
```

Geschachtelte Aufrufe von Methoden

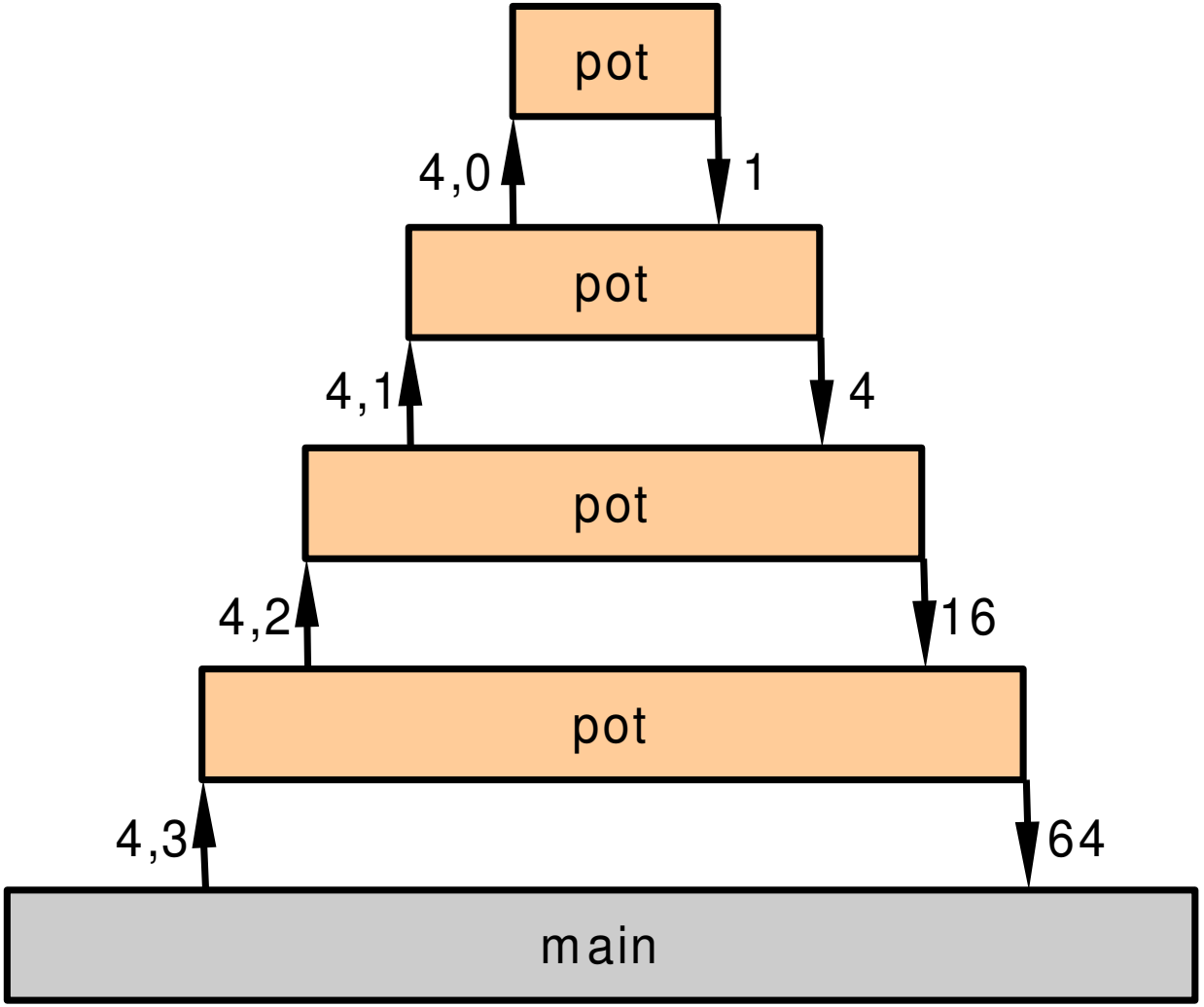


Rekursion

```
public static void main (String[] args) {
        System.out.println(pot(4,3));
}

public static int pot (int x, int y) {
        if (y <= 0)
                return 1;
        else
                return x * pot(x,y-1);
}
```

Rekursion



Rekursion

Direkte Rekursion: Aufruf einer Methode aus der Methode selbst heraus.

Indirekte Rekursion: Eine Methode ruft eine andere Methode durch diese wird wiederum (direkt oder indirekt) die erste Methode aufgerufen.

Lokale Variablen

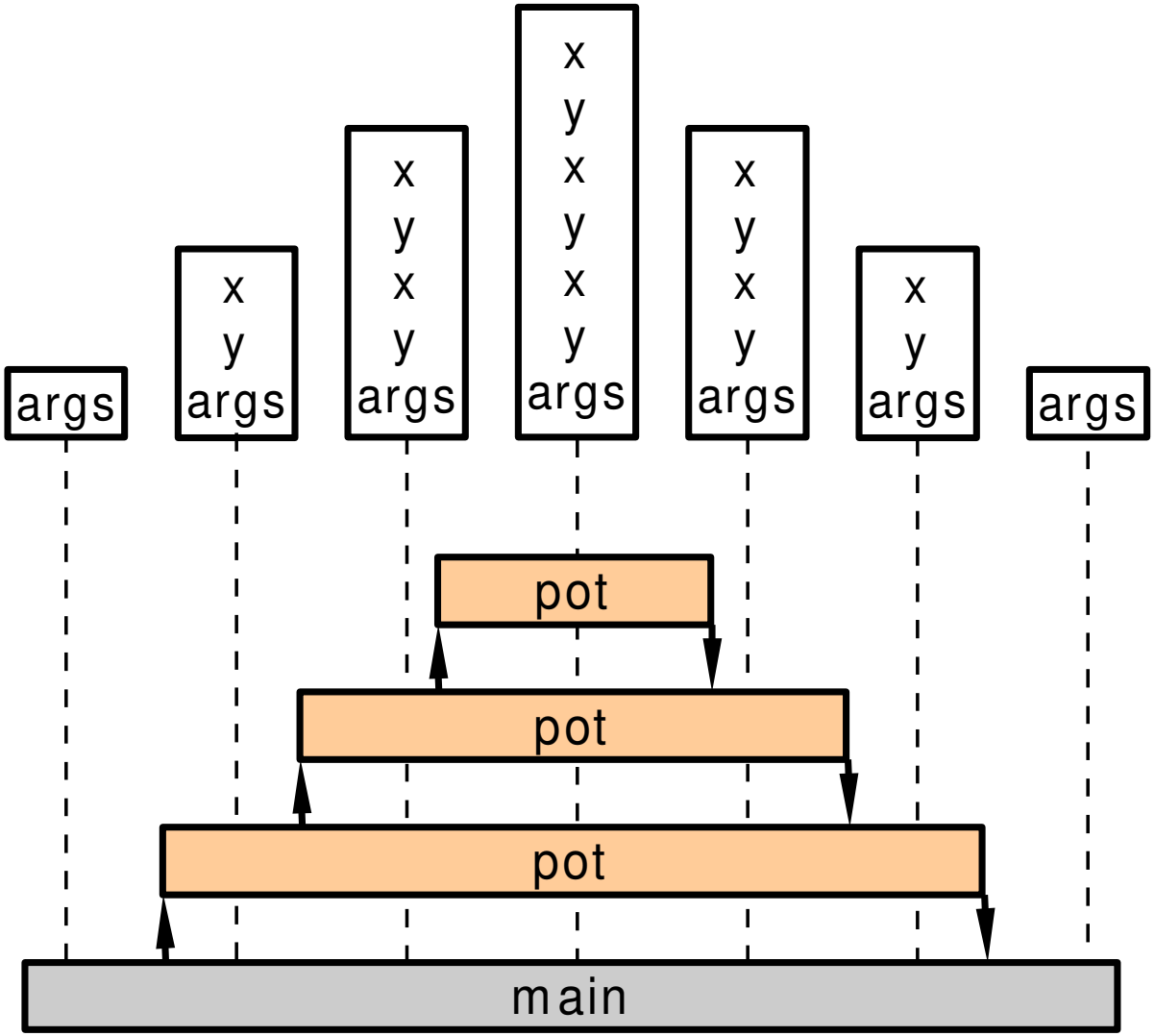
Als lokale Variablen einer Methode bezeichnet man

- ⇨ die Variablen, die in einer Methode deklariert wurden und
- ⇨ die Variablen, die einer Methode als Parameter übergeben werden.

Lokale Variablen

- Lokale Variablen beziehen sich immer auf den Aufruf einer Methode: eine lokale Variable lebt genau so lange wie der Aufruf der Methode.
- Bei rekursiven Methoden wird jede lokale Variable einer Methode gleichzeitig mehrfach benötigt.
- Bei dem Aufruf einer Methode kann aus der Methode heraus immer nur auf die eigenen (lokalen) Variablen zugreifen.
- Auf die lokalen Variablen anderer Methoden und auf die lokalen Variablen anderer Aufrufe der selben Methode kann nicht zugegriffen werden.

Lokale Variablen und der Stack (Stapel)



Lokale Variablen und der Stack

- Lokale Variablen werden durch einen Stack organisiert.
- Beim Aufruf einer Methode werden deren lokale Variablen auf den Stack gelegt.
- Nach der Beendigung des Aufrufs werden sie vom Stack genommen.
- Es gilt: Last-In, First-Out
  Variablen, die zuletzt auf den Speicher gelegt werden werden als erstes wieder vom Stack genommen.
- Ein Stack ist ein Bereich im Hauptspeicher mit einer festen unteren Adresse und einer variablen oberen Adresse. Der benutzte Speicherbereich wächst mit jedem Aufruf an und schrumpft, wenn Methoden verlassen werden.

2.7 Aufruf von Klassenmethoden anderer Klassen

Aufruf

Soll aus einer beliebigen Klassenmethode einer Klasse A die Klassenmethode f einer anderen Klasse B aufgerufen werden, so müssen der Klassenname B und ein Punkt "." vorangestellt werden: B.f()

```
public class A {
    public static void main(String[] args) {
        System.out.println( B.f(Integer.parseInt(args[0])) );
    }
}
```

```
public class B {
    public static int f(int x) {
        return x*x;
    }
}
```

Tipp: Testerklassen

- In der Regel entwickelt man beim Programmieren nicht ganze Programme auf einen Schlag, sondern man entwickelt die Methoden einzeln Schritt für Schritt.
- Um Methoden einzeln zu testen, empfiehlt es sich, eine separate Testerklasse anzulegen, von der aus diese Methode aufgerufen wird.
- Die Testerklasse hat eine main-Methode. Von der main-Methode der Testerklasse aus wird die zu testenden Methoden aufgerufen:
  - ⇨ Der Methode werden die Parameterwerte übergeben.
  - ⇨ Die von der Methode erzeugten Rückgabewerte werden ausgegeben (so vorhanden).
- In der Testerklasse können auch mehrere Aufrufe hintereinander durchgeführt werden (mit unterschiedlichen Parameterwerten).

Tipp: Testerklasse

```
public class A {
    public static void main(String[] args) {
        System.out.println( "f(3) = " + B.f(3) );
        System.out.println( "f(8) = " + B.f(8) );
        System.out.println( "f(0) = " + B.f(0) );
    }
}
```

```
public class B {
    public static int f(int x) {
        return x*x;
    }
}
```

public - private

- Bisher haben wir bei allen Deklarationen von Methoden das Schlüsselwort *public* verwendet.

```
public static int f(int x) { ...
```

- *public* steht für: Von jeder Methode aus darf diese Methode aufgerufen werden.

- Anstelle von *public* darf auch das Schlüsselwort *private* verwendet werden:

```
private static int f(int x) { ...
```

- *private*: Nur aus den Methoden der eigenen Klasse heraus darf diese Methode aufgerufen werden. Der Versuch, *private*-Methoden von anderen Klassen aus aufzurufen, scheitert am Compiler. Es wird ein Syntaxfehler angezeigt.

Klasse als Sammlung von Methoden

- Eine Klasse kann als eine Sammlung von Methoden betrachtet werden.
- Beispiel: Die Klasse Math, die in Java standardmäßig enthalten ist, enthält zahlreiche arithmetischer Methoden.
- Will man eine solche Sammlung von Methoden implementieren, so wird man zur Realisierung der Methoden eventuell andere Hilfsmethoden schreiben wollen. Diese Hilfsmethoden sollen nicht zur Sammlung gehören. Sie sollen deshalb nicht von außen aus aufrufbar sein (private).
- public: alle Methoden, die die Klasse nach außen zur Verfügung stellen soll
- private: alle Methoden, die innerhalb der Klasse verwendet werden, die aber nicht nach außen zur Verfügung gestellt werden sollen

Warum überhaupt Methoden mit private verstecken?

- Die Klasse für den Anwender übersichtlicher machen!
  - ⇨ Jemand, der die Klasse verwenden will, erkennt sofort, welche Methoden er verwenden soll (public) und welche nur interne Hilfskonstrukte sind (private).
- Die Wartbarkeit des Programmcodes verbessern!
  - ⇨ public-Methoden können in anderen Klassen verwendet werden. Wird die public-Methode geändert, so kann dies Auswirkungen auf alle Stellen im Programmcode haben, in denen die Methode verwendet wird.

2.8 Klassenattribute

Beispielprogramm

```
public class X {
    public static int zaehler;
    public static void dekrementieren() {
        if (zaehler > 0)
            zaehler = zaehler -1;
    }
}
```

```
public class Y {
    public static void main(String[] args) {
        X.zaehler = 2;
        X.dekrementieren();
        X.dekrementieren();
    }
}
```

Klassenattribute

- Klassenattribute sind Variablen.
- Anders als lokale Variablen, werden Sie direkt in der Klasse deklariert. Die Deklaration steht also neben den Methodendeklarationen und nicht wie bei lokalen Variablen im Rumpf von Methodendeklarationen.
- andere Bezeichnung: statisches Attribut
- Die Lebensdauer eines Klassenattributs beginnt mit dem Programmstart und endet mit dem Programmende. Zum Vergleich: Lokale Variablen existieren nur für einen Methodenaufruf lang.
- Klassenattribute können innerhalb der Methoden der gleichen Klasse verwendet werden.

... Klassenattribute

- Sollen Klassenattribute von einer anderen Klasse aus verwendet werden, so muss der Klassennamen vorangestellt werden:
  Klassenname.Attributname

2.9 Packages

Problem

- Bisher liegen alle Klassen, die zu einem Programm gehören, im gleichen Verzeichnis. Jeder Klasse X entsprechen eine X.java-Datei und eine X.class-Datei.
- Jeder Klassenname kann in einem Programm nur einmal verwendet werden.
- Schreiben mehrere Personen an unterschiedlichen Teilen eines Programmes, so könnte es leicht passieren, dass versucht wird, einen Klassennamen mehrfach zu verwenden.
- Bei einer großen Anzahl von Programmierern müssten sich diese in aufwendiger Weise abstimmen, um eine Namens-Kollision zu vermeiden.

Idee

- Es werden Unterverzeichnisse gebildet.
- In jedem Unterverzeichnis darf jeder Klassenname verwendet werden.
- Insbesondere darf der gleiche Klassenname in unterschiedlichen Verzeichnissen verwendet werden.

Packages

- Das Verzeichnis, in dem bisher alle Klassen lagen, wird als "Default Package" bezeichnet.
  - ⇨ Wir nehmen an, das Verzeichnis des Default-Packages sei def.
- Unterhalb des Verzeichnisses des "Default Package" werden Unterverzeichnisse und Unter-Unterverzeichnisse etc. gebildet.
  - ⇨ Beispiel: Zwei Unterverzeichnisse def\geometry\triangles und def\encryption
- Das Hauptverzeichnis def und die Unterverzeichnisse werden als Packages bezeichnet. Der Package-Name entspricht dem Pfadnamen ab dem Verzeichnis def, es wird jedoch der Punkt "." als Trennzeichen verwendet anstelle des Back-Slash "\".
  - ⇨ Beispiel: Die obigen Packages heißen geometry.triangles und encryption.

... Packages

- Jede Klasse, die nicht im Default-Paket ist, muss am Anfang des Programmcodes ein Package-Statemanet haben, das angibt, zu welchem Package sie gehört.
  - ⇨ Beispiel:
    Die Klasse A sei im Package geometry.triangles.
    Dann liegen die beiden Dateien A.java und A.class beide im Unterverzeichnis def\geometry\triangles.
    Außerdem muss der Programmcode von A.java mit einem Package-Statement beginnen

```
package geometry.triangles;
public static A {
...
}
```

... Packages

- Das Verzeichnis, in dem die Klasse steht, und das Package-Statement müssen konsistent sein!
- Soll eine Klasse eine Klasse aus einem anderen Paket verwenden, so muss der Paketname vorangestellt werden.
  - ⇨ geometry.triangles.A
- Der gleiche Klassenname darf in unterschiedlichen Packages verwendet werden. Zur Unterscheidung dient der Paketname.
  - ⇨ geometry.triangles.A
  - ⇨ encryption.A

Verwendung von Packages

- Packages gruppieren die Klassen in hierarchischer Weise.
- Diese Hierarchie sollte eine systematische Gliederung des Codes darstellen.
- Diese Gliederung des Codes kann in einem Softwareprojekt gleich oder ähnlich sein zur Aufgabenaufteilung auf die Entwickler oder Teilgruppen des Entwicklungsteams.
- Programmieren in einem Softwareprojekt mehrere Entwickler an Klassen im gleichen Package, so müssen sich die Entwickler bzgl. der Namensgebung bei den Klassen abstimmen.
- Beschränkt sich hingegen jeder Programmierer auf seine eigenen Packages, so ist diese Abstimmung nicht notwendig.

import

- ✗ Programmcode, in dem viele Klassen mit langen Pakage-Namen verwendet werden, wirkt unübersichtlich.
- ✗ Abhilfe: import-Anweisungen

import

- Import-Anweisungen können im Programmcode unterhalb der package-Deklaration und noch vor der Klassendeklaration stehen.

```
package geometry.triangles;
import java.lang.Math;
public class A {
    public static void f () {
        double d = Math.random();
        System.out.println(d);
    }
}
```

- Import-Anweisungen beschreiben Packagename und Klassennamen von Klassen anderer Packages.

- Die in import-Anweisungen aufgezählten Klassen, dürfen ohne Angabe des Package-Namen im Programmcode verwendet werden.

... import

- Es können auch alle Klassen eines Packages auf einmal importiert werden. Die Import-Anweisung hat dann ein "*" anstelle eines Klassennamens.
  - ⇨ Beispiel:

```
import java.util.*;
```

Irgendeine Klassenmethode aufrufen

Ein eindeutiger Aufruf einer Klassenmethode setzt sich aus Paketname, Klassenname und Klassenmethodenname zusammen:

```
geometry.triangles.A.f();
```

geometry.triangles	Der Packagename sollte immer nur aus kleinen Buchstaben bestehen. Im allgemeinen setzt er sich aus mehreren durch . getrennte Abschnitte zusammen. Darf weggelassen werden, wenn die Klasse im gleichen Package liegt.
A	Klassennamen sollten immer mit meinem großen Buchstaben beginnten. Darf weggelassen werden, wenn sich die Klassenmethode in der gleichen Klasse befindet.
f	Methodennamen sollten immer mit einem Kleinbuchstaben beginnen.

2.10 Classpath, Libraries

Der CLASSPATH

- Als CLASSPATH bezeichnet man eine Einstellung für den Java-Compiler und den Java-Interpreter.
- Im CLASSPATH steht der Pfad zu dem Verzeichnis, in dem sich das Default-Package befindet.
- Von diesem Verzeichnis aus können Compiler und Interpreter alle Packages finden:
  - ⇨ Der Pfad zu dem Verzeichnis, indem sich das Default-Package befindet, steht direkt im CLASSPATH.
  - ⇨ In den Unterverzeichnissen dieses Verzeichnisses befinden sich die anderen Packages.
- In Entwicklungsumgebungen (Eclipse, JBuilder,...) kann CLASSPATH in der Konfiguration des Projekts angezeigt und verändert werden.

Mehrere Pfade im CLASSPATH

- In der CLASSPATH-Einstellung können auch mehrere Verzeichnis-Pfade angegeben werden.
- Um eine Klasse zu finden, werden nacheinander von links nach rechts alle im CLASSPATH angegebenen Pfade durchsucht.

Libraries

- Anstelle eines Verzeichnis-Pfads kann auch eine Datei mit der Endung .jar oder eine .zip angegeben werden.
- Dateien mit der Endung .jar und .zip bezeichnet man als Libraries.
- Libraries sind komprimierte Verzeichnisstrukturen ("gezippt"), in denen sich Packages (Verzeichnisse) mit Klassen (.class-Dateien) befinden.

Die Standard-Library

- Die Library rt.jar ist immer im Pfad enthalten.
- Die Library rt.jar enthält alle Standardklassen von Java (circa 8000 Klassen).
- Beim Programmieren können die Klassen von rt.jar problemlos verwendet werden. Es kann vorausgesetzt werden, dass diese Klassen auf allen Java-Systemen zur Verfügung stehen.
- Werden bei der Programmierung einer Anwendung andere Libraries verwendet, so muss sichergestellt werden, dass diese auf der Zielplattform vorhanden ist. Gegebenenfalls müssen diese mit der Anwendung ausgeliefert werden.

Kapitel 3

Objekte

Motivation

- Die bisher vorgestellten Sprachkonstrukte von Java:
  - ⇨ skalare Datentypen (int, char, boolean,...)
  - ⇨ Klassenmethoden
  - ⇨ Klassen als "Sammelbehälter" von Klassenmethoden
  - ⇨ lokale Variablen
  - ⇨ Parameter von Methoden
  - ⇨ Rückgabewerte von Methoden

- Die bisher vorgestellte Teilsprache von Java ist hinreichend, um beliebige Programme schreiben zu können.

- Das in diesem Kapitel vorgestellte Konzepte der Objekte erweitert die Sprache Java um Konstrukte im Sinne einer besseren Strukturierung des Codes und einer besseren Wiederverwendbarkeit.

Irrtümer

- ✗ In Java kommt man auch ohne Objektorientierung aus.
  - ⇨ Die Standard-Bibliothek von Java und auch andere Bibliotheken sind objektorientiert programmiert worden. Um sie verwenden zu können, muss man die objektorientierten Konzepte verstehen.
  - ⇨ Es gibt weitere Sprachkonstrukte in Java (z.B. Exceptions und Threads), die ebenfalls auf der Objektorientierung aufbauen.
- ✗ Objektorientiertes Programmieren ist leichter zu erlernen, objektorientierte Programme sind schneller, verbrauchen bei der Ausführung weniger Ressourcen und haben automatisch weniger Fehler.
  - ⇨ Das ist leider nicht so.

... Irrtümer

- ✗ Graphische Oberflächen kann man nur objektorientiert programmieren. Gleiches gilt für Programme die auf Datenbanken zugreifen sollen.
  - ⇨ Man kann beliebige Programme mit und ohne Objektorientierung programmieren.
- ✗ Programme, die in objektorientierten Programmiersprachen geschrieben wurden, laufen langsamer.
- ✗ Durch die Umstellung auf Objektorientierung werden Programme "automatisch" besser strukturiert.
- ✗ Objektorientiert ist objektorientiert.
  - ⇨ Es gibt große Unterschiede zwischen objektorientierten Sprachen und es gibt viele Gestaltungsmöglichkeiten bei der objektorientierten Umsetzung einer gegebenen Aufgabenstellung.

3.1 Objektattribute, Instanzen

Beispiel: Koordinate

```
public class Koordinate {
    public int x;
    public int y;
}
```

```
public class KoordinateTester {
    public static void main(String[] args) {
        Koordinate k = new Koordinate();
        k.x = 3;
        k.y = 4;
    }
}
```

Objekte = Instanzen von Klassen

- Bisher wurden Klassen als Sammelbehälter für Klassenmethoden und Klassenattribute verwendet.

- Jetzt werden durch Klassen neue Datentypen deklariert.

  ⇨ Im Beispiel: Die lokale Variable k hat den Typ Koordinate.

- Die Inhalte von skalaren Variablen werden als Werte bezeichnet. Die Menge der zulässigen Werte ist durch den Datentyp vorgegeben.

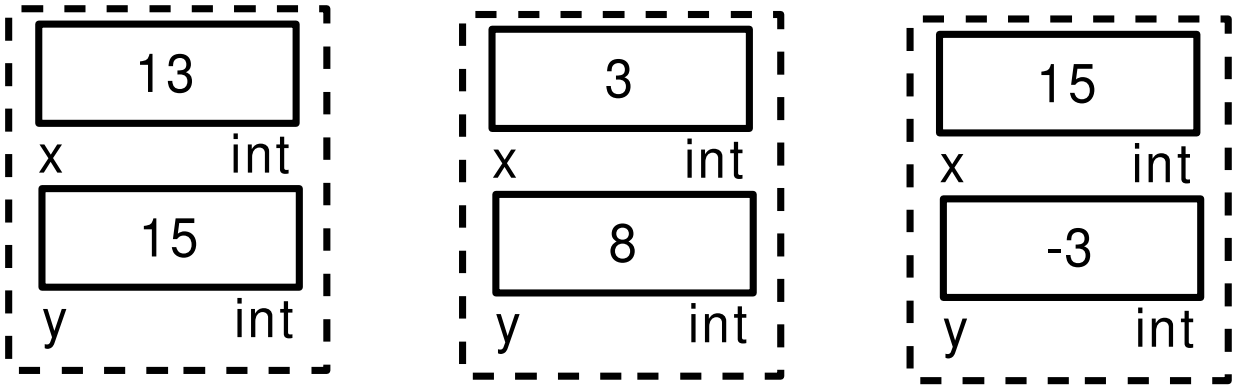
- Klassen stehen für einen Datentyp dessen Elemente als Objekte oder als Instanzen der Klasse bezeichnet werden.

skalarer Datentyp	Klasse
Werte	Objekte (Instanzen der Klasse)

Objekte und Objektattribute

```
public class Koordinate {
    public int x;
    public int y;
}
```

- Die in dieser Klasse deklarierten Variablen bezeichnet man als Objektattribute. Ein Objekt ist eine Datenstruktur, die sich aus mehreren Variablen, seinen Objektattributen, zusammensetzt.



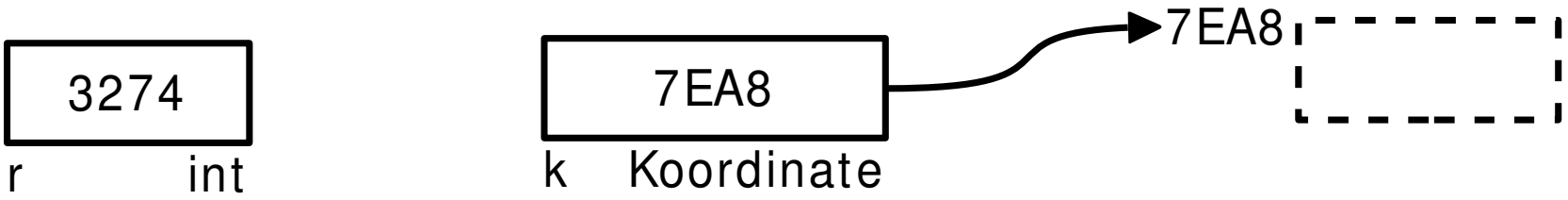
Instanzen von Klassen

- Durch die Deklaration einer skalaren Variable wird im Hauptspeicher ein Bereich zur Speicherung des Variablenwerts bereitgestellt.

- Durch die Deklaration einer Variablen, deren Typ eine Klasse ist, wird im Hauptspeicher nur ein Bereich zur Speicherung einer Referenz auf eine Instanz einer Klasse bereitgestellt.

```
Koordinate k;
```

- Die Referenz ist eine Adresse, die angibt, wo im Hauptspeicher sich das Objekt befindet. Das Objekt selbst existiert noch nicht.

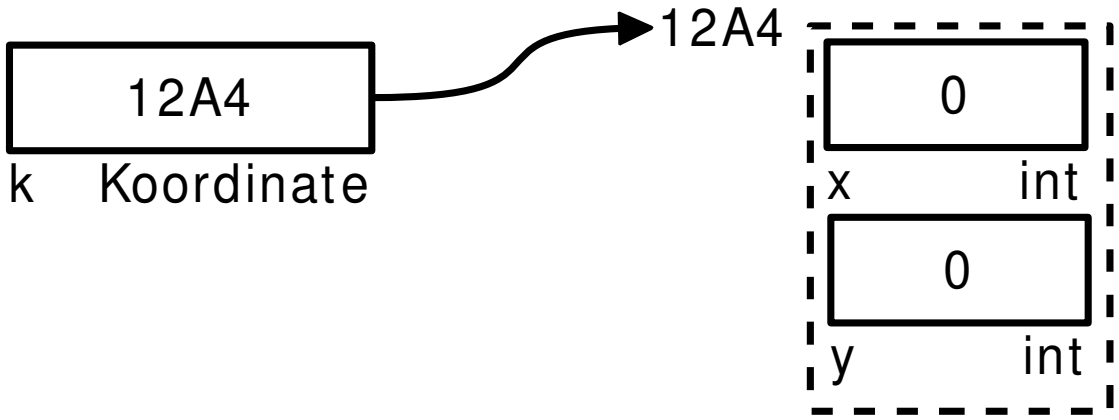


Erzeugen von Instanzen

- Zu Klassen können mit Hilfe des new-Operators Instanzen gebildet werden.

```
k = new Koordinate();
```

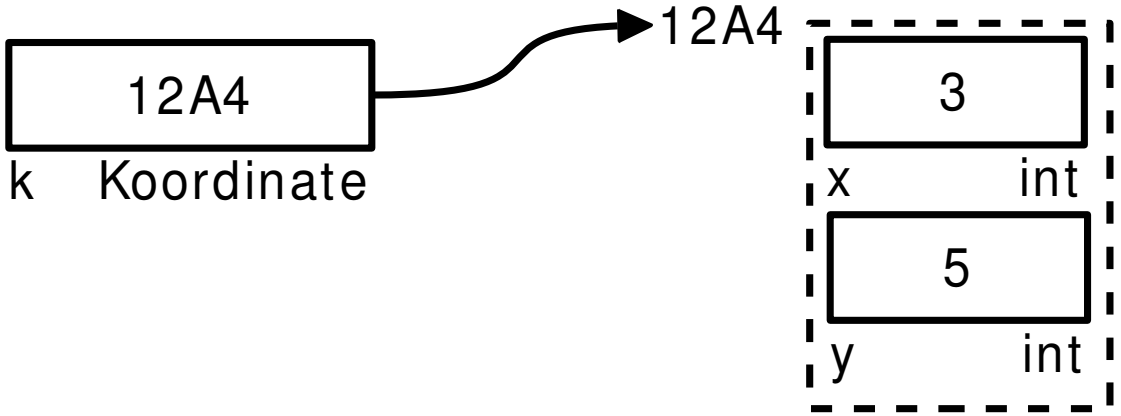
- Beim Aufruf des new-Operators geschieht folgendes
  - ⇨ Ein Stück Hauptspeicher wird für das Objekt reserviert.
  - ⇨ Die Referenz auf das Objekt (die Anfangsadresse des Hauptspeicherstücks) wird zurückgegeben.



Zugriff auf Objektattribute

- Sobald ein Objekt erzeugt wurde, kann auf die Objektattribute dieses Objekts mit *Objektname.Attributname* zugegriffen werden.

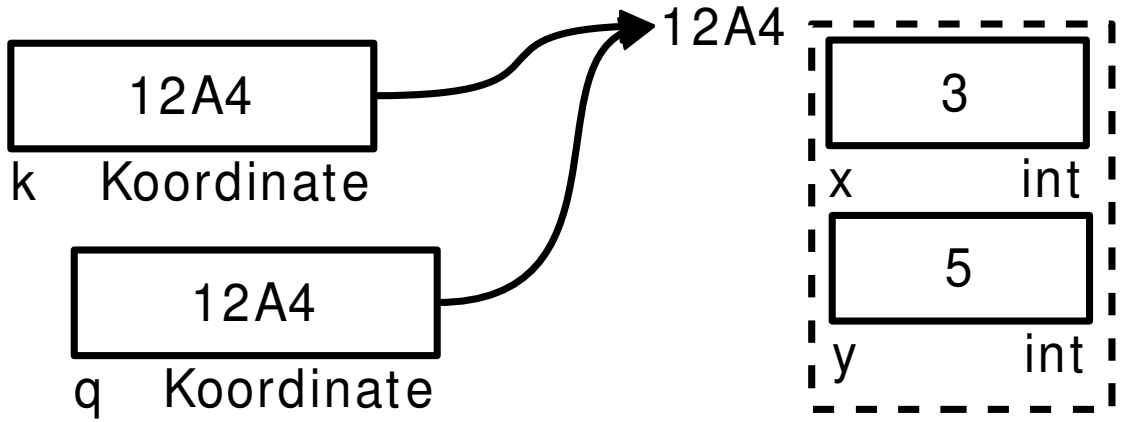
```
k.x = 3;
k.y = k.x * k.x - 4;
```



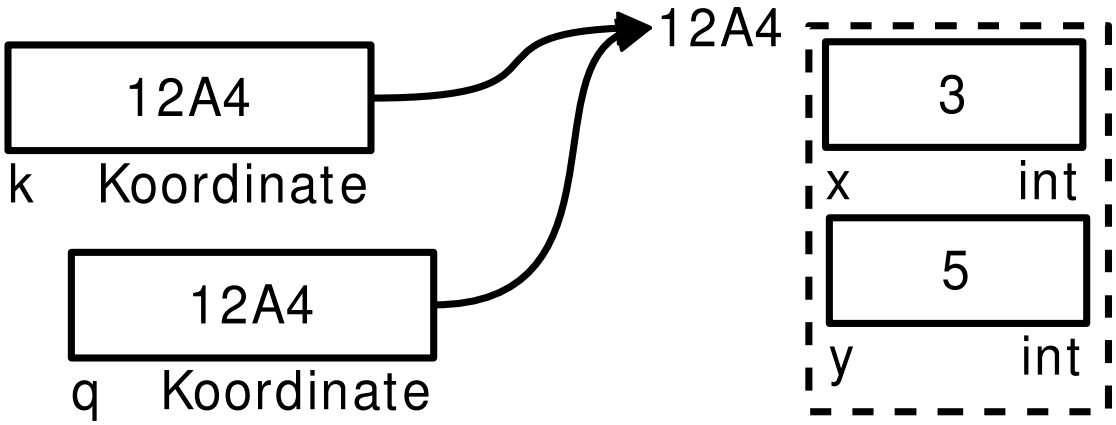
Zuweisung von Objektreferenzen

- Variablen, deren Typ eine Klasse ist, können Variablen vom gleichen Typ, also Instanzen der gleichen Klasse, zugewiesen werden.
- Dabei entsteht kein neues Objekt, es wird lediglich die Referenz kopiert. Beide Variablen zeigen anschließend auf das gleiche Objekt, das nur einmal existiert.

```
Koordinate q;
q = k;
```



Konsequenz daraus



- In dieser Situation kann auf die gleichen Speicherinhalte über die k und q zugegriffen werden.
- In dem folgenden Beispiel wird die Zahl 7 ausgegeben.

```
k.x = 7;
System.out.println(q.x);
```

Objektreferenzen als Parameter

- Wird für den Typ eines Methodenparameters eine Klasse angegeben, so wird an die Methode eine Objektreferenz übergeben.
- Die Methode kann das Objekt verändern.

  ⇨ Im folgenden Beispiel werden -1 und -2 ausgegeben.

```
public static void spiegeln(Koordinate p) {
    p.x = -p.x;
    p.y = -p.y;
}
public static void main(String[] args) {
    Koordinate k = new Koordinate();
    k.x = 1;
    k.y = 2;
    spiegeln(k);
    System.out.println(k.x + "  " + k.y);
}
```

Lebensdauer eines Objekts

- Objekte werden durch den new-Operator erzeugt.
- Dabei wird die Referenz zurückgegeben.
- Durch Zuweisungen kann die Referenz in verschiedenen Variablen gespeichert werden.
- Dadurch erhöht sich die Anzahl der Referenzen, die auf ein Objekt zeigen.
- Dadurch, dass einer Variable, die auf das Objekt zeigt, eine andere Referenz zugewiesen wird, verringert sich die Anzahl der Referenzen auf das Objekt um eins.
- Das "Leben" eines Objekts endet, wenn es keine Variable mehr gibt, die auf das Objekt zeigt.

Garbage Collection

- Ein Objekt entsteht explizit durch den Aufruf des new-Operators.
- Die Java-Virtual-Machine organisiert dann Speicherplatz für das Objekt.
- Das "Ableben" eines Objekts geschieht implizit: keine Referenz zeigt mehr auf das Objekt.
- Von Zeit zu Zeit durchforstet die Java-Virtual-Machine den Hauptspeicher nach Objekten, "die keiner mehr braucht".
- Dieser Vorgang wird als "Garbage Collection" bezeichnet.
- Der Speicher nicht mehr benötigter Objekte wird freigegeben.

Der Heap

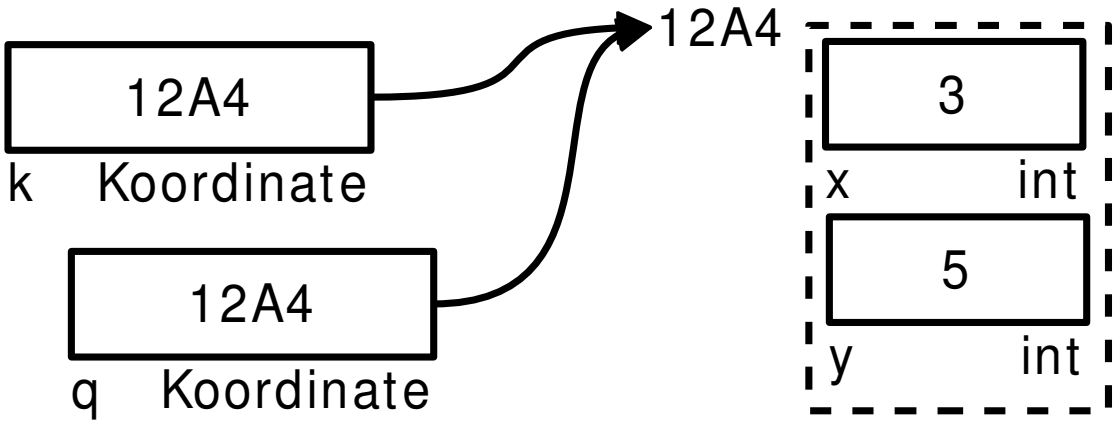
- Der Speicher, den die JVM (Java-Virtual-Machine) für Objekte verwendet, wird als Heap bezeichnet.
- Der Java-Programmierer weiß,
  - ⇨ dass die JVM beim new-Aufruf Speicher auf dem Heap reserviert
  - ⇨ dass die JVM die Speicherstücke verwaltet und zu jedem Zeitpunkt den Objekten Speicherplätze im Hauptspeicher zuordnen kann
  - ⇨ dass die JVM zu jedem Zeitpunkt weiß, welcher Teil des Heap belegt ist und welcher nicht
  - ⇨ dass die JVM Objekte nicht mehr referenzierte Objekte früher oder später aus dem Heap entfernt (wenngleich er auch nicht weiß, wann dies genau geschieht und es kann ihm auch gleichgültig sein)
- Wie die JVM den Heap verwaltet, ist für den Anwender der Programmiersprache nicht erkennbar und auch nicht erheblich.

null

- Allen Variablen, deren Typ eine Klasse ist, kann die Konstante null zugewiesen werden.
- null steht für: "kein Verweis auf ein Objekt"

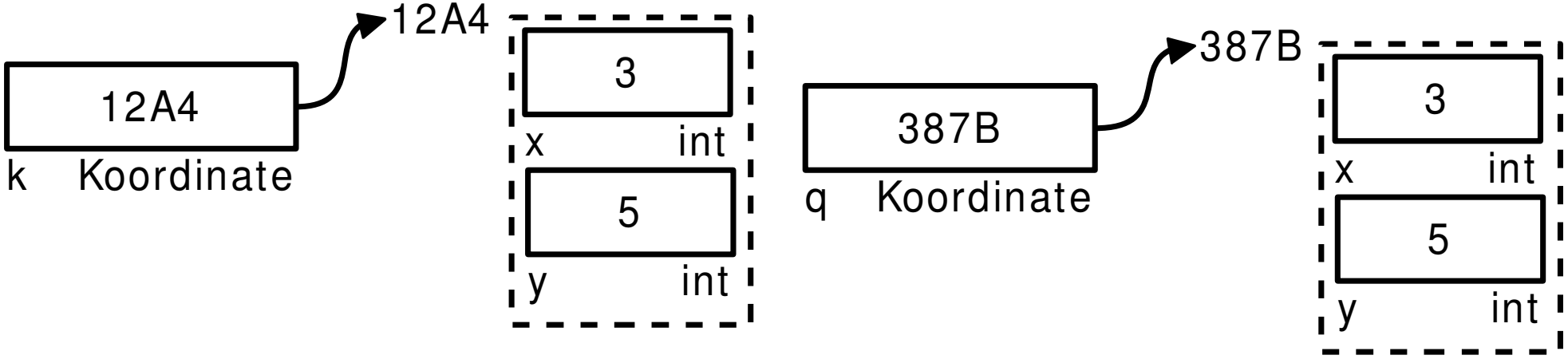
Vergleich von Objektreferenzen

- Objektreferenzen können miteinander mit == verglichen werden.
- Der Vergleich ergibt true genau dann, wenn die Objektreferenzen auf das selbe Objekt zeigen.
  - ⇨ Beispiel:
    Der Vergleich k==q ergibt im folgenden Fall true.



... Vergleich von Objektreferenzen

- Auch dann, wenn zwei unterschiedliche Objekte den gleichen Inhalt haben, ergibt der Vergleich der zugehörigen Objektreferenzen false.
  - ⇨ Beispiel:
    Der Vergleich k==q ergibt im folgenden Fall false.



- Haben beiden Objektreferenzen den Wert *null*, so ergibt der Vergleich true. Ist eine Objektreferenz *null* und die andere zeigt auf ein Objekt, so ergibt der Vergleich false.

Anmerkung

In Methodenrümpfen können lokale Variablen deklariert und verwendet werden. Es stellt sich die Frage, ob man im Programmcode einer Klassenmethode auch auf die Objektvariablen der gleichen Klasse zugreifen kann. Dem ist nicht so.

Erläuterung:

- ✗ Der Aufruf einer Klassenmethode steht nicht in Bezug zu einer bestimmten Instanz der Klasse. Will man auf ein Objektattribut zugreifen, so benötigt man jedoch ein Objekt, eine konkrete Instanz. Jedes Objektattribut existiert als Speicherplatz einmal pro Instanz. Instanzen könnte es zu einem Zeitpunkt mehrere geben oder auch gar keine. Es wäre unklar, welcher Speicherplatz gemeint ist.

3.2 Lokale Variablen, Klassenattribute, Objektattribute

Variablen

- Es wurden drei Arten von Variablen vorgestellt:
  - ⇨ lokale Variablen
  - ⇨ Klassenattribute
  - ⇨ Objektattribute

- Die Variablen-Arten unterscheiden sich in der Deklaration.
  - ⇨ Lokale Variablen werden innerhalb von Methodenrümpfen deklariert.
  - ⇨ Klassenattribute und Objektattribute werden innerhalb von Klassen und neben Methoden deklariert. Klassenattribute haben im Gegensatz zu Objektattributen das Schlüsselwort static in der Deklaration.

... Variablen

- Allen Variablen ist gemein, dass Sie der Aufbewahrung von Daten dienen. Sie haben einen festen Typ und einen festen Namen, die beide in der Deklaration festgelegt werden. Die Werte der Variablen können durch Zuweisungen verändert werden.

- Jede Variable - also jede lokale Variable, jedes Klassenattribut und jedes Objektattribut - kann unabhängig von seiner Art einen beliebigen Java-Typ haben. Java kennt folgende Typen:
  - ⇨ skalare Typen
    - byte, short, int, long, float, double, char oder boolean
  - ⇨ Klassen
    - jede in Java bekannte Klasse eignet sich als Datentyp
    - die in den Variablen gespeicherten Werte sind Referenzen auf Instanzen der jeweiligen Klasse

Variableninhalte

- Jede Variable benötigt in Java eine feste Anzahl von Bytes. Die benötigte Anzahl von Bytes hängt nur vom Typ der Variablen ab.

- Werte skalarer Datentypen:

  ⇨ Jeder der acht skalaren Datentypen wird durch eine feste Anzahl von Bytes kodiert. Beispiel: Der Typ int ist eine Zweikomplementdarstellung mit 4 Bytes.

- Objektereferenzen:

  ⇨ Eine Objektreferenz zeigt auf ein Objekt im Hauptspeicher. Gespeichert wird also eine Adresse im Hauptspeicher. Wieviel Speicherplatz für eine Hauptspeicheradresse benötigt wird, hängt von der JVM ab.

... Variableninhalte

- Um die Sprache Java richtig zu verstehen, muss man strikt zwischen skalaren Variablen und Objektreferenzen unterscheiden.
- Eine Variable, deren Datentyp skalar ist, speichert einen Wert. Eine Variable, deren Datentyp eine Klasse ist, speichert die Referenz auf ein Objekt.
- Folglich werden bei Zuweisungen einmal Werte und einmal Referenzen auf Objekte kopiert. Objekte werden durch eine Zuweisung nicht kopiert, sondern es wird in der Variablen lediglich ein weiterer Verweis auf das Objekt abgelegt.

Lokale Variablen, Klassenattribute, Objektattribute

- Die verschiedenen Arten von Variablen dürfen nicht verwechselt werden. Sie unterscheiden sich in der Lebenszyklus.
  - ⇨ Wenn beginnt die Existenz der Variablen?
  - ⇨ Wann endet die Existenz der Variablen?
- Verwechslungen führen zu fehlerhaftem oder zumindest ineffizientem Programmcode.

Speicherplätze: Lokale Variablen, Klassenattribute, Objektattribute, Objekte

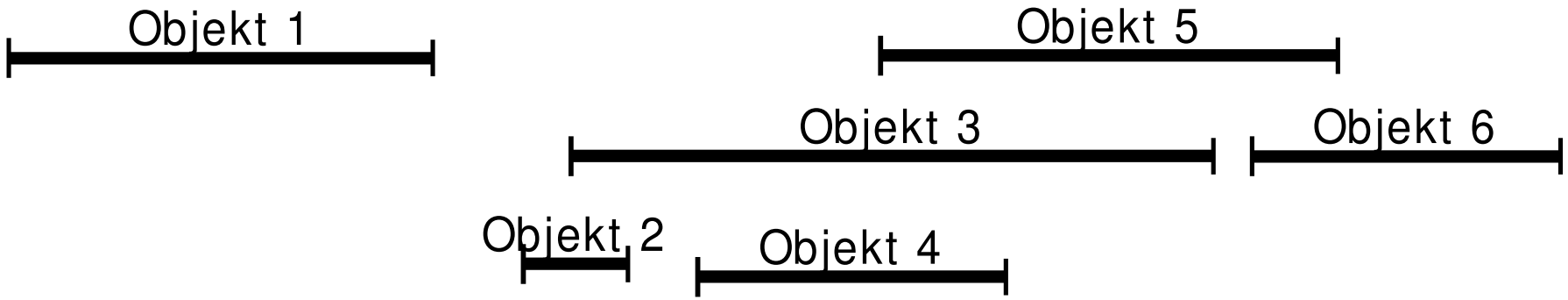
	Lokale Variable	*Klassenattribut*	*Objektattribut*	*Objekt*
Inhalt	skalarer Wert oder Objektreferenz	skalarer Wert oder Objektreferenz	skalarer Wert oder Objektreferenz	mehrere Attribute
Beginn Lebenszyklus	Aufruf einer Methode	Erstmalige Benutzung der Klasse im Programm	Entstehung des zugehörigen Objekts	new-Aufruf
Ende Lebenszyklus	Rückkehr von der Methode	Programmende	Ende des zugehörigen Objekts	keine Referenz zeigt mehr auf das Objekt
Speicherung	Stack	Heap	Heap (als Teil des Objekts)	Heap

Heap - Stack

- Variablen auf dem Stack werden in entgegengesetzter Reihenfolge angelegt und wieder entfernt.
  - ⇨ Zusammenhang: Aufrufhierarchie der Methoden.
  - ⇨ LIFO: Last In First Out
  - ⇨ einfache, schnelle Verwaltung auf dem Stapel

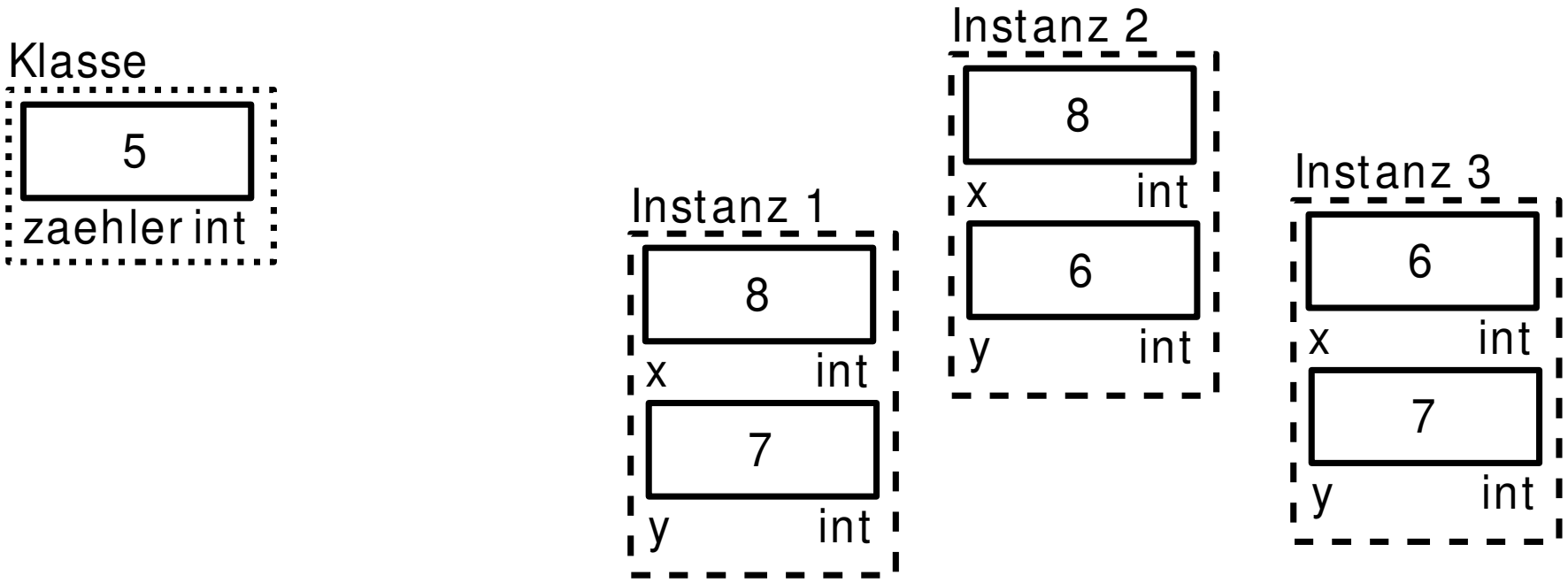
... Heap - Stack

- Objekte auf dem Heap werden zu "beliebigen" Zeitpunkten angelegt und wieder entfernt.
  - ⇨ Die Verwaltung des Heap ist aufwändiger.
  - ⇨ Instanzenbildung ist teuer:
    Aufwand für Suche nach freiem Speicherplatz, Verwaltung, Garbage Collection



Klassenattribute - Objektattribute

```
public class Koordinate {
    public static int zaehler;
    public int x;
    public int y;
}
```



Klassenattribute - Objektattribute

Klassenattribut	*Objektattribut*
statisch = Klassen-bezogen	nicht-statisch = Instanz-bezogen
Lebensdauer während der gesamten Programmlaufzeit	Lebenszeitraum des Objektattributs = Lebenszeitraum des Objekts
existiert genau einmal	existiert einmal pro Instanz
Verwendung in anderen Klassen: *Klassenname.Attributname*	Verwendung in anderen Klassen: *Objektname.Attributname*

3.3 Objektmethoden

Beispielprogramm

```
public class Koordinate {
    public double x;
    public double y;

    public void spiegeln() {
        x = -x;
        y = -y;
    }

    public double radius() {
        return Math.sqrt(x*x + y*y);
    }
}
```

... Beispielprogramm

```
public class Tester {
    public static void main(String[] args) {
        Koordinate k = new Koordinate();
        k.x = 3.0;
        k.y = 4.0;
        k.spiegeln();
        System.out.println(k.radius());
    }
}
```

Objektmethoden

- Die Deklaration einer Objektmethode unterscheidet sich von der Deklaration einer Klassenmethode nur durch das fehlende Schlüsselwort static.

- Unterschiede:
  - ⇨ Objektmethoden beziehen sich auf eine Objekt, eine Instanz der Klasse. Klassenmethoden beziehen sich auf eine Klasse.
  - ⇨ Aufruf einer Objektmethode eines Objekts, zu dem man einen Verweis hat:
    VerweisAufObjekt.Methodenname()
  - ⇨ Aufruf einer Klassenmethode aus einer anderen Klassen mit:
    Klassenname.Methodenname()
  - ⇨ Innerhalb der Objektmethode dürfen die Objektattribute des gleichen Objekts verwendet werden.

3.4 Klassen, Objekte, Attribute, Methoden

Übersicht

Als Bestandteile einer Klassendeklaration wurden bisher vorgestellt:

Objektmethoden	`public void f() { ... }`
Klassenmethoden	`public static void f() { ... }`
Objektattribute	`public int x;`
Klassenattribute	`public static int x;`

Anmerkung:
Die Deklaration einer lokalen Variablen sieht aus wie die Deklaration eines Objektattributs, befindet sich jedoch im Rumpf einer Objektmethode oder einer Klassenmethode und nicht direkt in der Klassendeklaration.

Andere Bezeichnungen

Für die vier Begriffe Objektmethoden, Klassenmethoden, Objektattribute und Klassenattribute gibt es leider eine Vielzahl von alternativen Bezeichnungen, die ebenfalls häufig gebraucht werden. Die nachfolgenden Ausführungen sollen nur der Orientierung dienen. Innerhalb der Vorlesung sollen die obigen vier Begriffe verwendet werden.

- Oft werden, wie auch in dieser Vorlesung, Objektattribute und Klassenattribute zusammenfassend als Attribute bezeichnet. Analog werden Objektmethoden und Klassenmethoden oft zusammenfassend als Methoden bezeichnet.

- Manchmal werden die Begriffe Methode und Attribut jedoch auch implizit als Synonym für Objektmethoden und Objektattribute gebraucht. Dies soll in dieser Vorlesung nicht geschehen.

... Andere Bezeichnungen

- In Java werden Attribute auch als Datenfelder (fields) bezeichnet, in C++ auch als data members.
- Anstelle der Bezeichnung Objektattribut ist auch die Bezeichnung Objektvariable oder Instanzvariable gebräuchlich.
- Klassenattribute werden auch als statische Attribute und Klassenmethoden als statische Methoden bezeichnet. Analog werden dann Objektattribute und Objektmethoden als nichstatische Attribute bzw. nichtstatische Methoden bezeichnet.

Regeln

- ✗ In einer Klassenmethode darf nur auf die Klassenattribute der gleichen Klasse zugegriffen werden.
- ✗ In einer Objektmethode darf auf alle Attribute der gleichen Klasse zugegriffen werden.

Beispiel für ein nicht zulässiges Programm:

```
public class X {
    public double x;
    public static int zaehler;
    public static void dekrementieren() {
        zaehler = zaehler-1;
        x = -x;                          ✗ Fehler!
    }
}
```

... Regeln

Aus einer Objektmethode heraus dürfen aufgerufen werden:

- ✗ Objektmethoden der gleichen Klasse
- ✗ Klassenmethoden der gleichen Klasse

```
public class X {
    ...
    public void f() { ...
    }
    public static void g() { ...
    }
    public void h() {
        f();
        g();
    }
}
```

... Regeln

- ✗ Aus einer Klassenmethode heraus dürfen Klassenmethoden der gleichen Klasse aufgerufen werden.
- ✗ Der Aufruf von Objektmethoden ist nicht erlaubt.

```
public class X {
    ...
    public void f() { ...
    }
    public static void g() { ...
    }
    public static void h() {
        f();                    ✗ Fehler
        g();
    }
}
```

3.5 Objektorientierung

Beispielprogramm - objektorientierter Stil

```
public class Koordinate {
    public double x;
    public double y;

    public void spiegeln() {
        x = -x;
        y = -y;
    }
    public double radius() {
        return Math.sqrt(x*x + y*y);
    }
}
```

Beispielprogramm - "klassischer" Stil

```
public class Koordinate {
    public double x;
    public double y;
}
```

```
public class KoordinatenOperationen {
    public static void spiegeln(Koordinate k) {
        k.x = -k.x;
        k.y = -k.y;
    }
    public static double radius(Koordinate k) {
        return Math.sqrt(k.x*k.x + k.y*k.y);
    }
}
```

Klasse = Datenstruktur + Programm

- Prinzip von Klassen:
  Kapselung einer Datenstruktur mit dem dazugehörigen Programmcode als eine Einheit

- Dieses Grundkonzept erweist sich als sehr erfolgreich bei der Strukturierung großer Programme.

Nicht-objektorientierte Programmiersprachen

- ✗ Nicht-Objektorientierte Programmiersprachen wie Pascal und C verfügen über ein solches Sprachkonstrukt nicht.
- ✗ In nicht-objektorientierten Programmiersprachen gibt es Sprachkonstrukte, mit denen neuen Datenstrukturen durch Bündelung bekannter Datenstrukturen definiert werden können (Verbunde, Records).
- ✗ Außerdem gibt es Sprachkonstrukte zur Bündelung von Programmteilen (Module).
- ✗ Beide Sprachkonstrukte werden durch das Konzept der Objektorientierung als Spezialfälle abgedeckt.
- ✗ Jedoch:
  In nicht-objektorientierten Programmiersprachen gibt es keine Kapselung einer Datenstruktur mit dem dazugehörigen Programmcode als eine Einheit.

Konstrukte nicht-objektorientierter Programmiersprachen

Verbund (Record)

- Aufgabe: Definition einer neuen Datenstruktur durch Bündelung bekannter Datenstrukturen
- entspricht in objektorientierten Sprachen einer Klasse, die nur aus Objektattributen besteht

```
public class Koordinate {
    public double x;
    public double y;
}
```

Konstrukte nicht-objektorientierter Programmiersprachen

Modul

- Aufgabe: Bündelung von Programmteilen (Prozeduren)
- entspricht in objektorientierten Sprachen einer Klasse, die ausschließlich aus Klassenmethoden besteht

```
public class KoordinatenOperationen {
    public static void spiegeln(Koordinate k) {
        k.x = -k.x;
        k.y = -k.y;
    }
    public static double radius(Koordinate k) {
        return Math.sqrt(k.x*k.x + k.y*k.y);
    }
}
```

Konsequenz

- In nicht-objektorientierten Programmiersprachen werden die Datenstrukturen und die Programmteile, die mit ihnen arbeiten, durch unterschiedliche Sprachkonstrukte gebündelt.
- Dieser Unterschied zu objektorientierten Sprachen mag auf den ersten Blick unwesentlich erscheinen.
- Solange man nur mit kleinen Programmen arbeitet, wird man den Vorteil der Objektorientierung kaum erkennen.
- Der Vorteil der besseren Strukturierung durch das objektorientierte Sprachkonzept wird mit zunehmender Komplexität der Programme augenfällig.
- In der Praxis verdrängen objektorientierte Programmiersprachen zunehmend nicht-objektorientierte Programmiersprachen.

3.6 Konstruktoren

Initialisierung von Objekten

Die Erzeugung von Objekten geschah bisher in zwei Schritten

1. Schritt: Aufruf des new-Operators mit dem Default-Konstruktor
   - ⇨ Ein Speicherstück für die Attribute wird reserviert.
   - ⇨ Die Attribute erhalten automatisch initiale Werte:
     alle Objektreferenzen: null
     alle int-Attribute: 0
     alle boolean-Attribute: false
     etc.
2. Schritt: Den Attributen werden eigene Initialwerte zugeordnet.

```
Koordinate k = new Koordinate();
k.x = 3;
k.y = 4;
```

Beispiel: Selbstdefinierte Konstruktoren

```
public class Koordinate {
    public double x;
    public double y;
    public Koordinate (double a, double b) {
        x = a;
        y = b;
    }
    public Koordinate (double a) {
        x = a;
        y = 0;
    }
}
```

```
Koordinate k1 = new Koordinate(3,4);
Koordinate k2 = new Koordinate(7);
```

Selbstdefinierte Konstruktoren

- ✗ Selbstdefinierte Konstruktoren ähneln im Aufbau einer Methode.
- ✗ Der Name ist immer gleich dem Klassennamen.
- ✗ Die Namen von Klassen werden üblicherweise mit großem Anfangsbuchstaben geschrieben, Konstruktoren auch.
- ✗ Konstruktoren können - wie Methoden - beliebige Parameter haben.
- ✗ Konstruktoren haben keinen Rückgabewert. Es wird weder ein Rückgabetyp noch void angegeben.
- ✗ Im Rumpf der Konstruktoren befindet sich Programmcode, der die Aufgabe hat, die Attribute des Objekts zu initialisieren.
- ✗ Klassen können mehrere Konstruktoren haben, die sich in den Parametern unterscheiden.

Anmerkungen

- ✗ Alle Klassen, die keinen selbstdefinierten Konstruktor enthalten, haben automatisch den Default-Konstruktor.

- ✗ Enthält eine Klasse mindestens einen selbstdefinierter Konstruktor, so steht der Default-Konstruktor nicht zu Verfügung.

Bestandteile einer Klassendeklaration

Objektmethoden	`public void f() { ... }`
Klassenmethoden	`public static void f() { ... }`
Objektattribute	`public int x;`
Klassenattribute	`public static int x;`
Konstruktoren	`public A() { ... }`

Anmerkung:
Die Deklaration einer lokalen Variablen sieht aus wie die Deklaration eines Objektattributs, befindet sich jedoch im Rumpf einer Objektmethode oder einer Klassenmethode und nicht direkt in der Klassendeklaration.

3.7 Vererbung

Beispiel

```
public class Koordinate {
    public double x;
    public double y;
    public void spiegeln() {
        x = -x;
        y = -y;
    }
    public double radius() {
        return Math.sqrt(x*x + y*y);
    }
}
```

```
public class Koordinate3d extends Koordinate {
    public double z;
    public void normalisieren() {
        int r = radius();
        x = x/r;
        y = y/r;
        z = z/r;
    }
    public double radius() {
        return Math.sqrt(x*x + y*y + z*z);
    }
}
```

Vererbung

- Idee:
  Deklaration einer neuen Klasse als eine Erweiterung einer bestehenden Klasse.

- Die neue Klasse wird als Sohn-Klasse bezeichnet, die bestehende Klasse als Vater-Klasse.

  ⇨ Deklaration:

```
public class Sohnklasse extends Vaterklasse {
    ...
}
```

... Vererbung

- Die neue Klasse "erbt" alle Attribute und Methoden von der Vaterklasse.
  - ⇨ Dadurch, dass "extends Vaterklasse" angegeben wird, verfügt die Sohn-Klasse schon über alle Attribute und Methoden der Vater-Klasse.
- Die Sohn-Klasse wird um Attribute und Methoden erweitert.
  - ⇨ zusätzliche Attribute und Methoden im Rumpf der Sohn-Klasse, die es in der Vater-Klasse noch nicht gibt
- In der Sohn-Klasse werden Methoden der Vater-Klasse überschrieben.
  - ⇨ überschreiben = umdefinieren, neu implementieren
  - ⇨ Methoden mit genau der gleichen Schnittstelle (Name, Parameter, Rückgabewert) wie die Vaterklasse.

... Vererbung

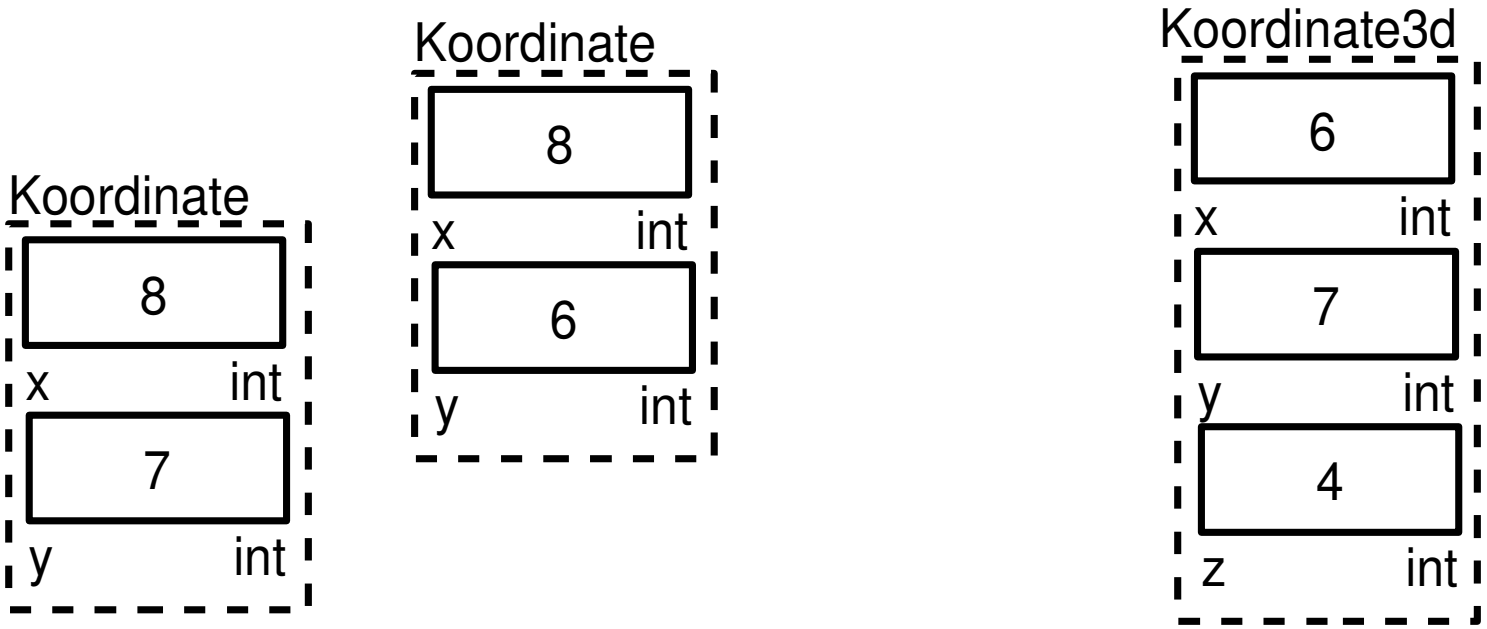
- Die in der Vater-Klasse verfügbaren Attribute und Methoden sind in der Sohn-Klasse verfügbar.
- Auch die Vater-Klasse kann wiederum durch Vererbung entstanden sein. Die in der Vater-Klasse verfügbaren Attributen und Methoden sind ererbt oder im Rumpf deklariert worden.
- Eine Klasse erbt, so vorhanden, von
  - ⇨ der Vater-Klasse
  - ⇨ der Vater-Klasse der Vater-Klasse ("Großvater-Klasse")
  - ⇨ der Vater-Klasse der Vater-Klasse der Vater-Klasse ("Urgroßvater-Klasse")
  - ⇨ ...

Prinzip der Einfachvererbung

- In Java kann eine Klasse nur von einer Klasse erben.
  - ⇨ In anderen Worten:
    Es ist nicht erlaubt, dass eine Klasse mehrere Vater-Klassen hat.
  - ⇨ Prinzip der Einfachvererbung
- Hinweis: Es ist in Java durchaus erlaubt, dass mehrere Klassen von einer Klasse erben.
  - ⇨ mehrere Klassen haben die gleiche Vater-Klasse
- Es gibt andere Programmiersprachen, bei denen Mehrfachvererbung erlaubt ist.
  - ⇨ Mehrfachvererbung: eine Klasse hat mehrere Vater-Klassen
  - ⇨ Mehrfachvererbung gibt es z.B. in der Programmiersprache C++

Zusätzliche Objektattribute

- Wird die Sohn-Klasse um Objektattribute erweitert, so bedeutet dies, dass jede Instanz alle Objektattribute der Vaterklasse und alle Objektattribute der Sohn-Klasse hat.
- Konsequenz: Eine Instanz der Sohn-Klasse benötigt mehr Speicherplatz für die Attribute als eine Instanz der Vater-Klasse.



Zusätzliche Objektmethoden

- In den zusätzlichen Objektmethoden der Sohn-Klasse darf zugegriffen werden auf
  - ⇨ die Methoden der Vater-Klasse
  - ⇨ die zusätzlichen Methoden der Sohn-Klasse

⇨ Beispiel für eine zusätzlichen Objektmethode:
Im Beispielcode die Methode *normalisieren* der Klasse *Koordinate3d*.

Das Überschreiben von Objektmethoden

- Die Situation ist ähnlich wie bei zusätzlichen Objektmethoden mit dem Unterschied, dass in der Vaterklasse bereits eine Objektmethode mit
  - ⇨ dem gleichem Namen,
  - ⇨ den gleichen Parametern und
  - ⇨ dem gleichen Rückgabewert

  existiert.

- Vater-Klasse und Sohn-Klasse haben somit eine Objektmethode, die in gleicher Weise benutzt werden kann, die sich jedoch im Methodenrumpf unterscheidet.

⇨ Beispiel für eine überschriebene Objektmethode:
Im Beispielcode die Methode *radius* der Klasse *Koordinate3d*.

super

- Gegeben:
  eine Objektmethode wird in der Sohn-Klasse überschrieben
  ⇨ Beispiel: die Methode *spiegeln* in der Klasse *Koordinate3d*

- Wird in einer Objektmethode der Sohn-Klasse diese Methode aufgerufen, so bezieht sich der Aufruf automatisch auf den Programmcode im Rumpf der Sohn-Klasse.

- Soll hingegen gezielt die gleichnamige Objektmethode der Vaterklasse angesprochen werden, so muss dem Methodennamen *super.* vorangestellt werden.

super - Beispiel

```
public class Koordinate {
    public double x;
    public double y;
    public void spiegeln() {
        x = -x;
        y = -y;
    }
    public double radius() {
        return
            Math.sqrt(x*x+y*y);
    }
}
```

```
public class Koordinate3d extends Koordinate {
    public double z;
    public void normalisieren() {
        x = x/radius();
        y = y/super.radius();
        z = z/radius();
    }
    public double radius() {
        return Math.sqrt(x*x + y*y + z*z);
    }
}
```

- In der Methode *normalisieren* wird mit *radius()* der Programmcode der Methode *radius()* im Rumpf der Klasse *Koordinate3d* aufgerufen
- In der Methode normalisieren wird mit *super.radius()* der Programmcode der Methode radius() im Rumpf der Klasse *Koordinate* aufgerufen.

Konstruktoren und Vererbung

- Wurde bei einer Vater-Klasse ein Konstruktor deklariert, so ist für alle Söhne der Default-Konstruktor nicht mehr verfügbar und es muss im Sohn ein eigener Konstruktor deklariert werden.

- Der selbstdefinierte Konstruktor des Sohnes kann dann in der ersten Programmzeile mit

```
super(parameter1, parameter2,...);
```

  den Konstruktor des Vaters aufrufen.

Zusätzliche Klassenattribute

- Wird die Sohn-Klasse um Klassenattribute erweitert, so bedeutet dies, dass nur für die neuen Klassenattribute zusätzlicher Speicherplatz benötigt wird.

- Die Speicherplätze für die ererbten Klassenattribute sind identisch mit denen der Vaterklasse.

  ⇨ In nachfolgendem Beispiel verweisen A.x und B.x auf die gleiche Variable:

```
public class A {
    public static double x;
}
```

```
public class B extends A {
    public static double y;
}
```

Klassenmethoden in der Sohnklasse

- In den zusätzlichen Klassenmethoden der Sohn-Klasse darf zugegriffen werden auf alle Klassenattribute und alle Klassenmethoden der Sohn- und der Vaterklasse.

- Haben Vater- und Sohnklasse Methoden mit gleichem Namen und gleichen Parametern und Rückgabewerten, so sind diese doch klar unterscheidbar, indem beim Aufruf der Klassenname vorangestellt wird. Innerhalb der Vaterklasse bzw. innerhalb der Sohnklasse können die jeweils eigenen Methoden auch ohne das Voranstellen des Klassennamens aufgerufen werden.

3.8 Die Klasse Object

Die Klasse Object

- In Java gibt es eine bereits definierte Klasse mit dem Namen *Object* (genauer: *java.lang.Object)*.

- Die Klasse Object hat in Java eine besondere Stellung:

  ⇨ Jede Klasse ist direkt oder indirekt Sohn der Klasse *Object*.

Mit und ohne "extends"

- Durch "*extends Vaterklasse*" kann bei einer Klassendeklaration explizit eine Vaterklasse angegeben werden.

```
public class Sohnklasse extends Vaterklasse { ...
}
```

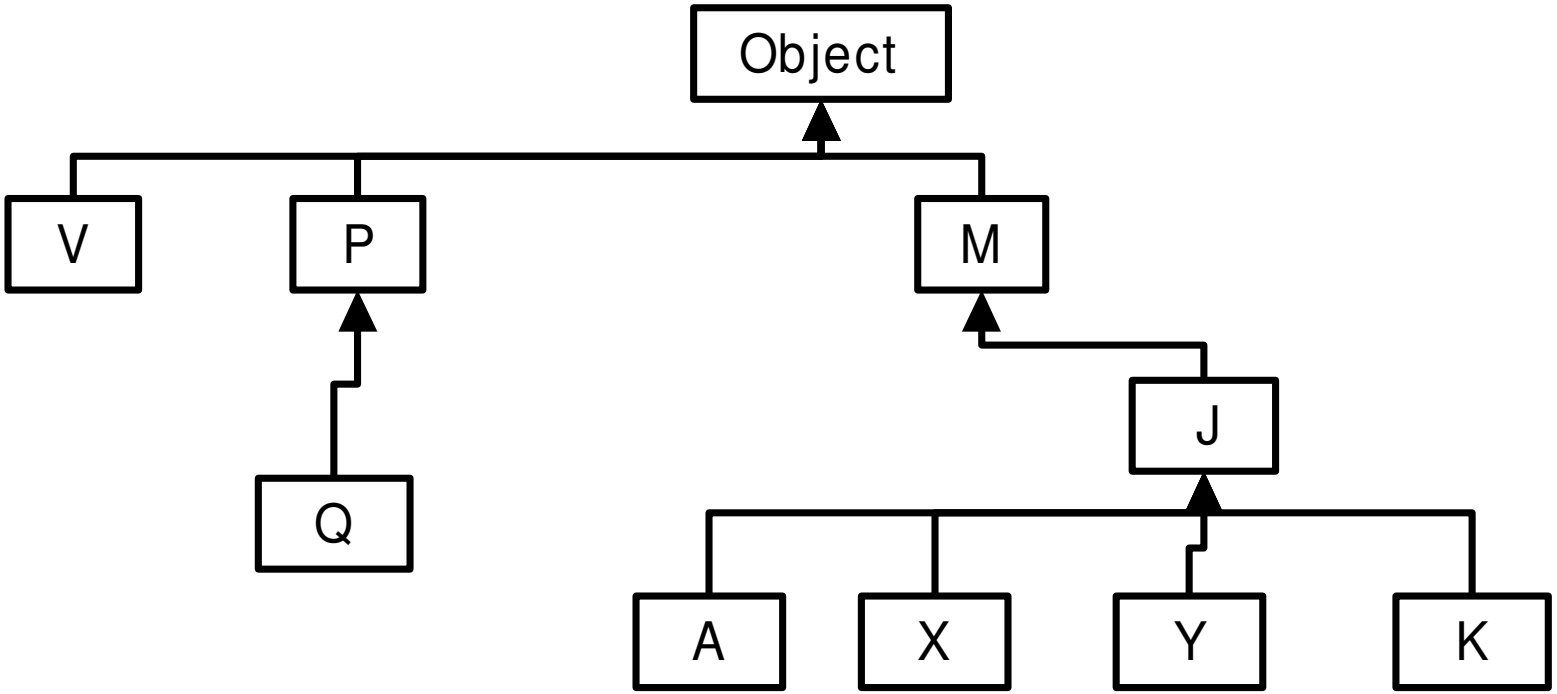
- Enthält die Klassendeklaration kein *extends*, so bedeutet dies nicht, dass sie keine Vaterklasse hätte, sondern vielmehr, dass die Vaterklasse *Object* ist.

```
public class Sohnklasse { ...
}
```

steht für

```
public class Sohnklasse extends java.lang.Object { ...
}
```

Object und die Klassenhierarchie in Java



Die Pfeile stehen für die Relation "ist Sohnklasse von".

In Java bilden alle Klassen bezüglich der Vererbung eine große Baumstruktur, an deren Wurzel die Klasse *Object* steht.

Folgerungen

- Die einzige Klasse in Java, die keine Vaterklasse hat, ist die Klasse *Object*.
- Alle anderen Klassen haben eine Vaterklasse: entweder die im extends-Abschnitt angegebene oder *Object*.
- Dies gilt sowohl für die vom Benutzer definierten Klassen als auch für die bereits standardmäßig in der Runtime-Library von Java enthaltenen Klassen.

Methoden der Klasse Object

Die Klasse *Object* enthält bereits elf Methoden. Diese Methoden werden an alle Klassen in Java vererbt und werden dort zum Teil überschrieben.
Hinweis: Die Methoden von *Object* beziehen sich zum Teil auf Konzepte, die in der Vorlesung noch nicht vorgestellt wurden (Hash-Tabelle, Introspection, Nebenläufigkeit).

toString()	Umwandeln des Objekts in einen String
equals(Object obj)	Vergleich mit einem anderen Objekt
clone()	Kopieren des Objekts
finalize()	wird vor Garbage-Collection ausgeführt
getClass()	Informationen über die eigene Klasse (Introspection)
hashcode()	Schlüssel für Hash-Tabellen
notify(), notifyAll(), wait(), wait(long t), wait(long t, long n)	Synchronisation von Threads (Nebenläufigkeit)

3.9 Polymorphie

Polymorphie

Jede Instanz der Sohnklasse kann wie eine Instanz der Vaterklasse verwendet werden.

Anmerkung:
Das Prinzip der Vererbung stellt sicher, dass die Methoden und Attribute, die es in einem Objekt der Vater-Klasse gibt, auch in einem Objekt der Sohn-Klasse existieren.

... wenngleich durch Überschreiben einige der Methoden im Sohn anders implementiert sein können

Polymorphie - genauere Betrachtung

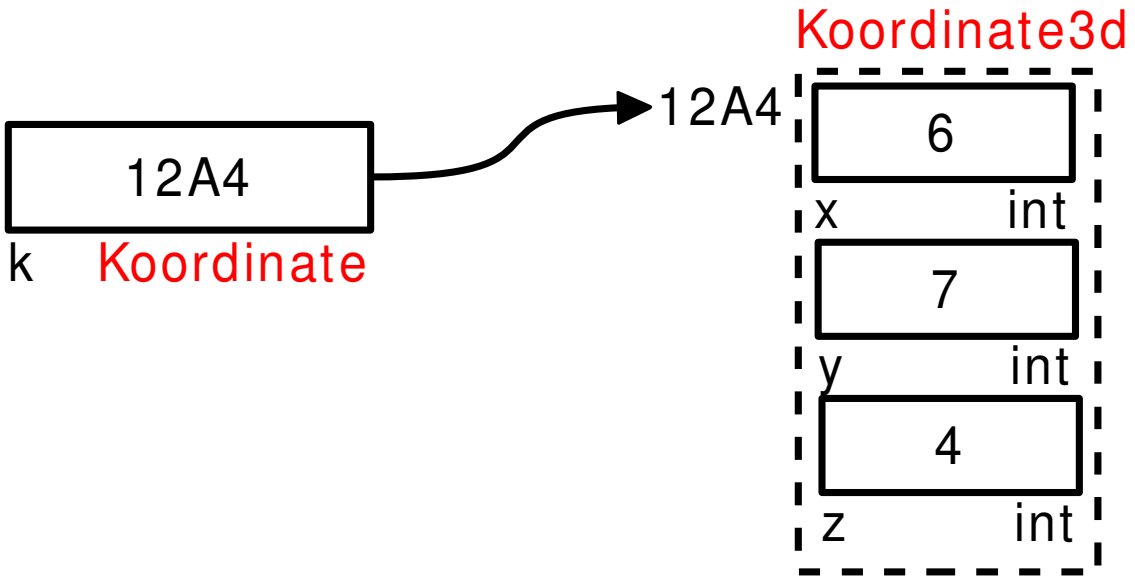
- In Variablen, die den Typ einer Klasse haben, wurden bisher immer Referenzen auf Instanzen dieser Klasse gespeichert.
  - ⇨ Beispiel:
    Die Variable k hat den Typ Koordinate. Es können in ihr Referenzen auf Instanzen vom Typ Koordinate gespeichert werden

```
Koordinate k;
```

- Jetzt neu: Man kann in diesen Variablen auch Referenzen auf Instanzen von Sohnklassen speichern. Gleiches gilt für Sohnes-Söhne etc..
  - ⇨ Beispiel:

```
Koordinate k;
k = new Koordinate3d();
```

... Polymorphie - genauere Betrachtung

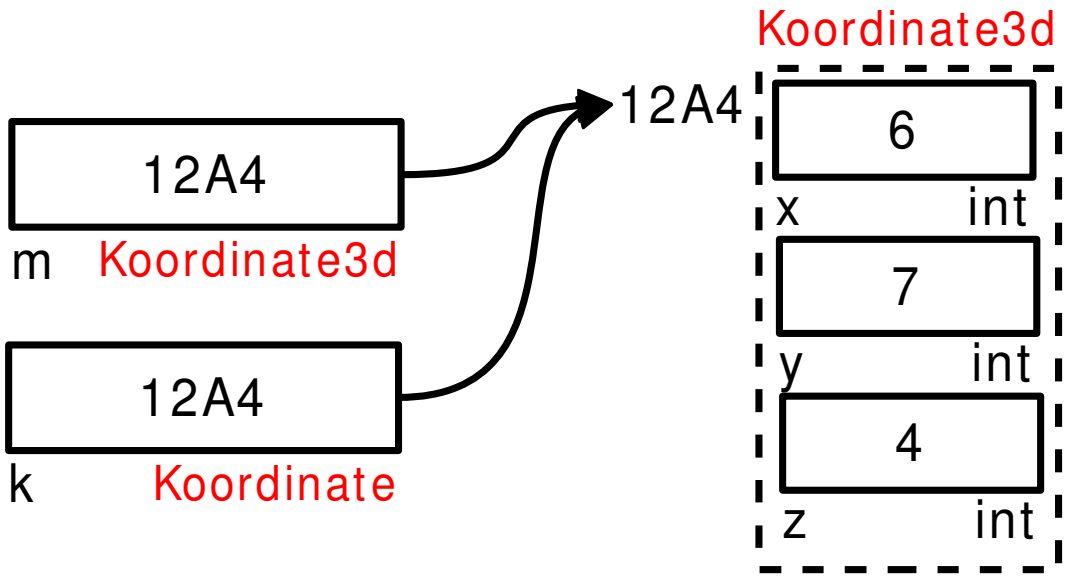


- Auf welche Attribute und Methoden über eine Referenz zugegriffen werden kann, ergibt sich aus dem Typ der Referenz und nicht aus dem Typ der Instanz.
  - ⇨ Beispiel: Die Instanz von Koordinate3d hat die drei Attribute x, y und z. Über die Referenz k kann nur auf die Attribute x und y zugegriffen werden.

Folgerungen

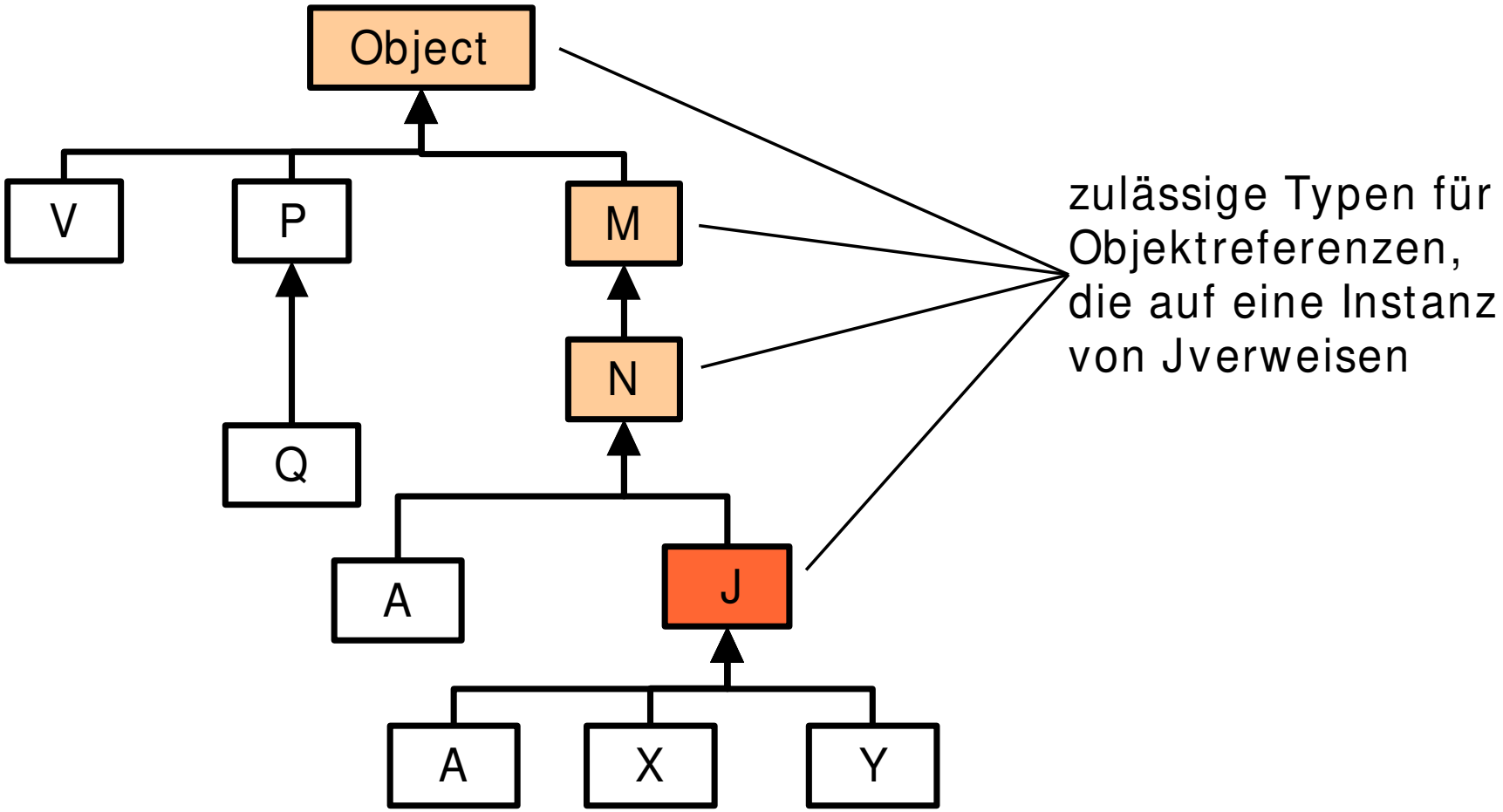
Auf eine Instanz einer Klasse können zu einem Zeitpunkt mehrere Referenzen verweisen, deren Typ unterschiedlich sein kann: die Klasse selbst oder eine Vaterklasse.

```
Koordinate3d m;
Koordinate k;
m = new Koordinate3d();
k = m;
```



Erläuterung: Über *m* kann auf alle drei Attribute des Objekts zugegriffen werden, über *k* nur auf die Attribute *x* und *y*.

... Folgerungen



... Folgerungen

- Hat in einer Methode ein Parameter den Typ einer Klasse, so dürfen auch Referenzen auf Söhne dieser Klasse übergeben werden.
- Gleiches gilt für Rückgabewerte von Methoden.

Polymorphie und Überschreiben

```
public class X {
    public static double f(Koordinate a, Koordinate b) {
        return a.radius() + b.radius();
    }
}
```

- ✗ Der Methode *f* werden gemäß Deklaration zwei Parameter übergeben, die Referenzen der Klasse *Koordinate* sind.
- ✗ Anstelle dessen dürfen jeweils auch Referenzen auf Söhne und Sohnes-Söhne von *Koordinate* übergeben werden.

⇨ Beispiel:

```
double z = X.f(new Koordinate(), new Koordinate3d());
```

... Polymorphie und Überschreiben

- Polymorphie im Zusammenhang mit dem Überschreiben von Objektmethoden führt dazu, dass bei einem Aufruf dieser Objektmethode unterschiedlicher Programmcode ausgeführt wird - je nachdem von welchem Sohn oder Sohnes-Sohn das Objekt erzeugt wurde.

- Im allgemeinen kann erst zur Laufzeit entschieden werden, welcher Programmcode ausgeführt werden muss.

- Beispiel:

```
Koordinate z;
if (p<28)
    z = new Koordinate3d();
else
    z = new Koordinate();
System.out.println(z.radius());
```

Polymorphie und die Klasse Object

- Da jede Klasse direkt oder indirekt Sohn der Klasse *Object* ist, kann in einer Variable vom Typ Object eine Referenz auf ein beliebiges Objekt gespeichert werden.

  ⇨ Variablen vom Typ *Object* als Container beliebiger Objekte.

3.10 Casts

Casts

- Ein Cast steht für eine Typumwandlung:
  Zu einem Wert x vom Typ *typa* wird ein "analoger" Wert vom *typb* berechnet.

- Schreibweise: Es wir einem Ausdruck ein Cast-Operator vorangestellt, ein mit runden Klammern umklammerter Java-Typ.

  ⇨ Beispiel: In dem folgenden Ausdruck habe *x* den Typ *typa*, der gesamt Ausdruck hat den Typ *typb*.

```
(typb) x
```

- Casts gibt es sowohl für skalare Werte als auch für Objektreferenzen. Den Casts für skalare Werte und den Casts für Objektreferenzen liegen zwei grundlegend unterschiedliche Konzepte zugrunde. Durch Casts können nur Objektreferenzen in Objektreferenzen und skalare Werte in skalare Werte überführt werden - aber nicht Objekte in skalare Werte oder umgekehrt.

Casts mit skalaren Werten

```
int x = 3;
double y = 4.4;
x = (int)y;
```

- Beim Casting von skalaren Werten wird zu einem Wert eines skalaren Datentyps ein "analoger Werte" eines anderen skalaren Typs bestimmt.
  - ⇨ Aus der int-Zahl 3 wird durch Casting mit *(double)3* die double-Zahl 3.0.
- Es gibt in Java genau definierte Regeln, die für die acht verschiedenen skalaren Datentypen von Java beschreiben
  - ⇨ ob von einem Typ zum anderen gecastet werden darf
  - ⇨ wie gegebenenfalls diese Überführung erfolgt

Casts mit Objektreferenzen

- In diesem Abschnitt soll im weiteren nur noch das Casting von Objektreferenzen betrachtet werden.

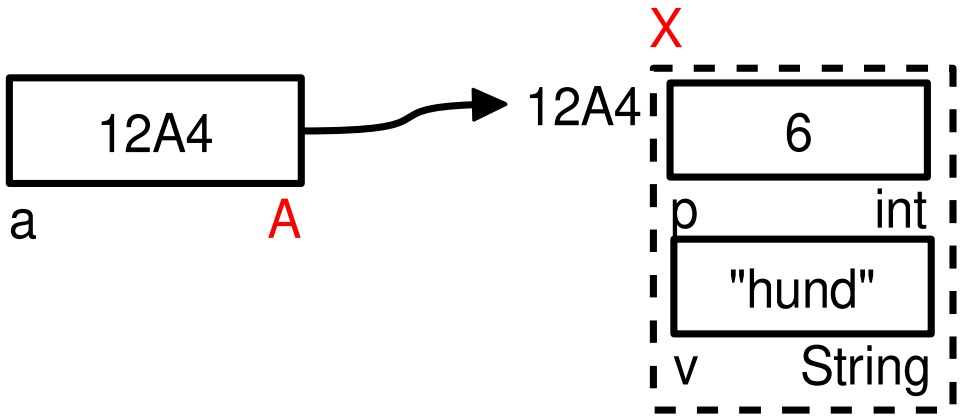
- Beim Casting von Objektreferenzen wird zu einer Objektreferenz auf ein Objekt eine Objektreferenz auf das gleiche Objekt bestimmt, die einen anderen Typ hat.
  - ⇨ Erinnerung: Auf eine Instanz einer Klasse dürfen Objektreferenzen verweisen, deren Typ eine Vaterklasse der Instanz ist.

- Das Objekt selbst wird durch den Cast nicht verändert!

... Casts mit Objektreferenzen

Es soll allgemein ein Cast von Klasse *A* zu Klasse *B* beschrieben werden.

- Ausgangssituation: Es existiere eine Variable *a* vom Typ *A*, die auf eine Instanz der Klasse *X* verweist. Dies ist nur dann möglich, wenn die Klasse *A* eine Vaterklasse von *X* ist oder wenn sie identisch sind.

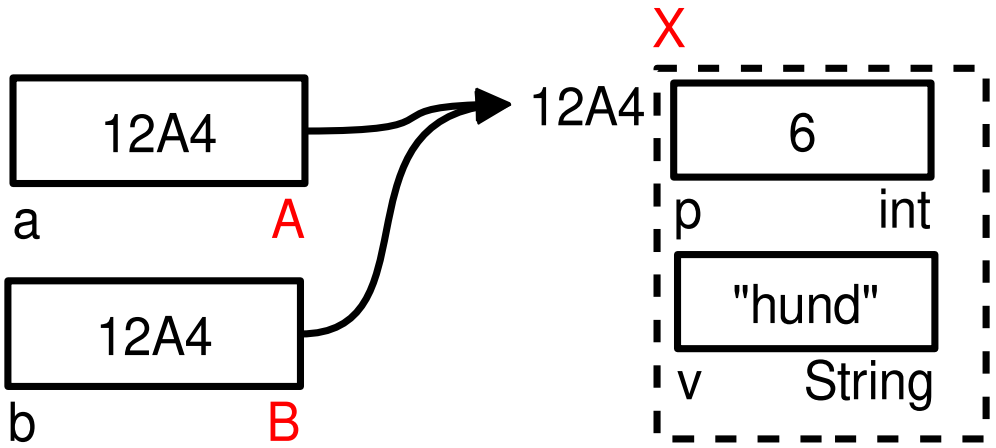


... Casts mit Objektreferenzen

- Der Cast: Es sei *b* eine Variable vom Typ *B*. Der nachfolgende Cast ist nur dann möglich, wenn *B* ebenfalls eine Vaterklasse oder Vater-Vater-Klasse von *X* ist oder wenn sie identisch sind.

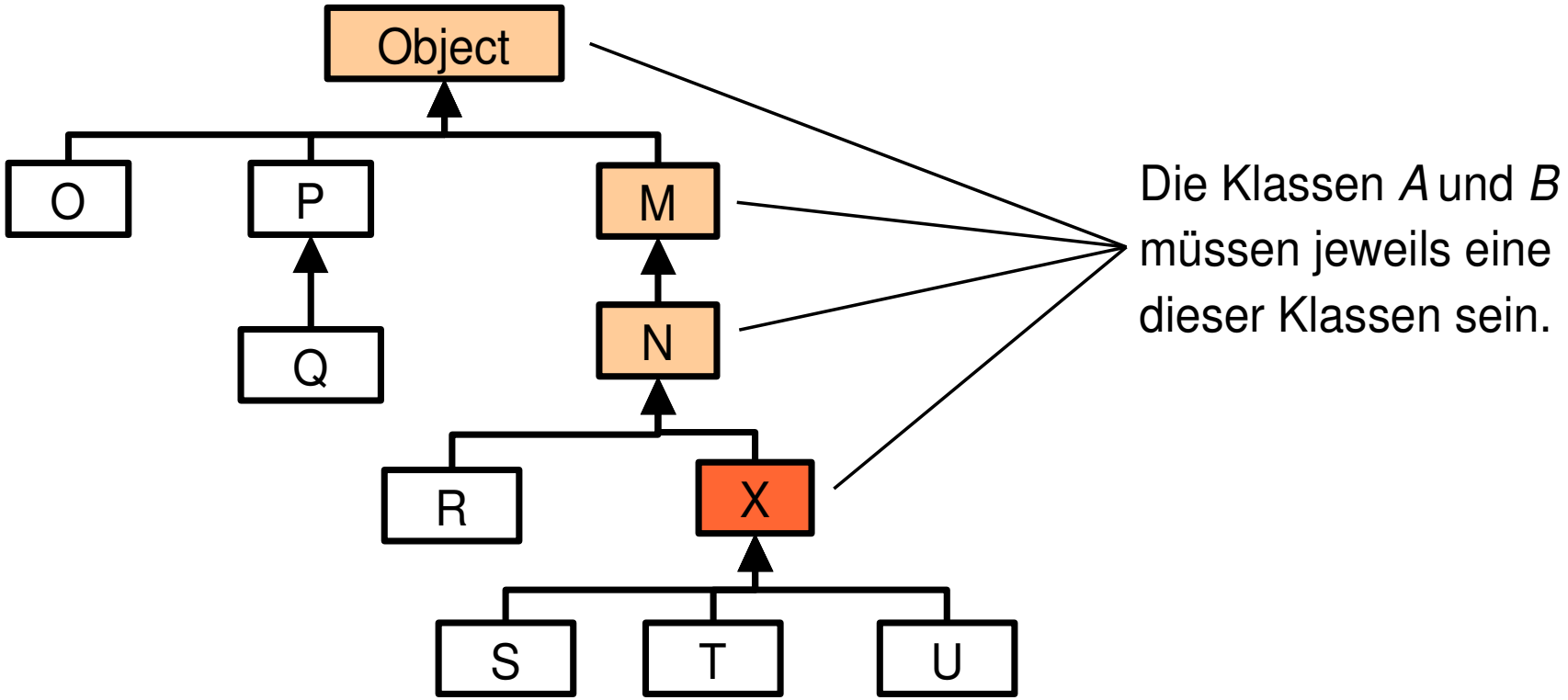
```
b = (B) a;
```

- Situation nach dem Cast:



Voraussetzung für den Cast von Objektreferenzen

Sowohl die Objektreferenz von der aus gecastet wird (A) als auch die Objektreferenz zu der gecastet wird (B) müssen jeweils einen Typ haben, der entweder identisch mit der Klasse der referenzierten Instanz (X) ist oder eine Vaterklasse der Klasse der Instanz ist (bzw. Vater-Vater-Klasse etc.).



Upcast - Downcast

Casts mit Objektreferenzen beziehen sich auf die Vererbungsbeziehung der Klassen.
Es gibt zwei Arten von Casts mit Objektreferenzen:

- Upcast: Cast in Richtung Vater-Klasse
  - ⇨ *B* ist Vaterklasse von *A* (oder Vater-Vater etc.)
- Downcast: Cast in Richtung Sohn-Klasse
  - ⇨ *B* ist Sohnklasse von *A* (oder Sohnes-Sohn etc.)

Implizite Upcasts

Ergibt sich bei Upcasts der Zieltyp aus dem Kontext des Programms, so darf der Cast-Operator weggelassen werden. Man spricht dann von impliziten Upcasts. Diese Form von Casts kam bereits im Abschnitt zum Thema Polymorphie zum Einsatz.

⇨ Beispiel:

```
Koordinate3d m = new Koordinate3d();
Koordinate k;
k = m;                    //    Impliziter Upcast
```

Probleme mit Casts

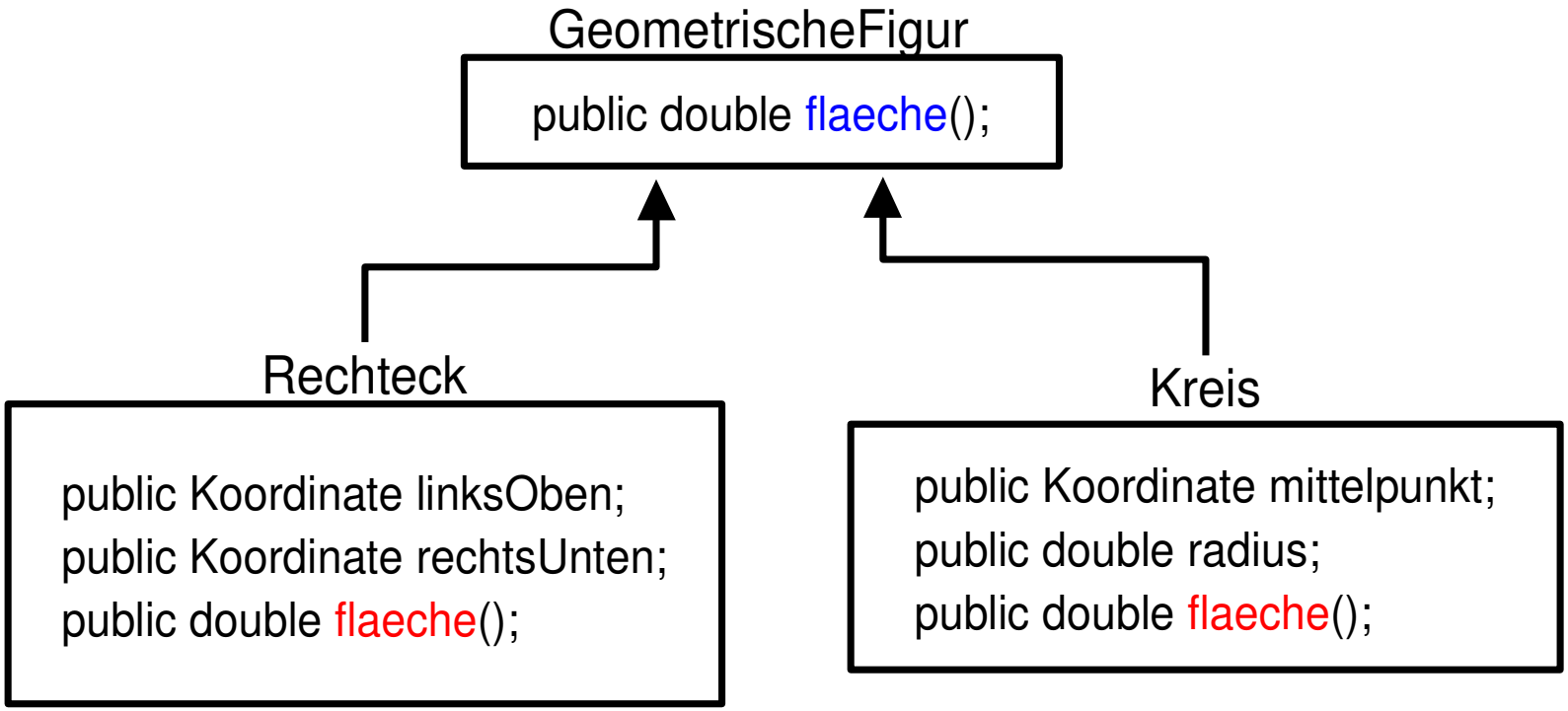
Zu einer ClassCastException kann es nur bei einem Downcast kommen, bei Upcasts ist dies ausgeschlossen.

Vorsicht bei Downcasts!

- ✗ Abhängig von dem Programmablauf kann die zu castende Referenz auf irgendeine Instanz einer Klasse verweisen, die Sohn des Typs der Referenz ist.
- ✗ Entscheidende Frage: Ist bei dem Downcast immer gewährleistet, dass diese Instanz Sohn der Klasse ist, auf die gecastet werden soll?
- ✗ Andernfalls kommt es zu einer ClassCastExceptions.

3.11 Abstrakte Methoden, Abstrakte Klassen

Motivation



- Die Methode *flaeche* soll für alle Instanzen von *GeometrischeFigur* zur Verfügung stehen.
- Von der Klasse *GeometrischeFigur* selbst sollen keine Instanzen gebildet werden - nur von den Sohnklassen, die die Methoden *flaeche* überschreiben.

Abstrakte Methoden

```
public abstract double flaeche();
```

- Eine abstrakte Methode ist eine Methode ohne Methodenrumpf (also ohne Programmcode).
- In der Methodendeklaration einer abstrakten Klasse muss das Schlüsselwort "abstract" verwendet werden.
- Typen der Parameter und des Rückgabewertes einer abstrakten Methode sind eindeutig festgelegt - nur der Programmcode fehlt.
- Abstrakte Klassenmethoden sind nicht zulässig. Abstrakte Methoden sind somit immer Objektmethoden.

Abstrakte Klassen

```
public abstract class GeometrischeFigur {
    public abstract double flaeche();
}
```

- Eine Klasse mit mindestens einer abstrakten Methode wird als eine abstrakte Klasse bezeichnet.
- Abstrakte Klassen müssen durch das Schlüsselwort "abstract" gekennzeichnet werden - andernfalls wird ein Syntaxfehler angezeigt.
- Von abstrakten Klassen können keine Instanzen gebildet werden.
  - ⇨ Damit wird es auch unmöglich die abstrakten Methoden aus anderen Klassen aufzurufen. Da abstrakte Methoden Objektmethoden sind, müsste zunächst eine Instanz gebildet werden, bevor die Methode aufgerufen werden könnte.

Beispiele

```
public class Kreis extends GeometrischeFigur {
    public Koordinate mittelpunkt;
    public double radius;
    public double flaeche() {
        return radius * radius * Math.PI;
    }
}
```

```
public class Rechteck extends GeometrischeFigur {
    Koordinate linksOben;
    Koordinate rechtsUnten;
    public double flaeche () {
        return Math.abs(
            (linksOben.x - rechtsUnten.x) * (linksOben.y - rechtsUnten.y) );
    }
}
```

Aufruf abstrakter Methoden

```
public abstract class GeometrischeFigur {
      public double dicke;
      public abstract double flaeche();
      public double doppelteFlaeche() {
            return flaeche() * 2.0;
      }
}
```

- Aus den nichtabstrakten Methoden der abstrakten Klassen dürfen auch abstrakte Methoden aufgerufen werden - und zwar sowohl
  - ⇨ neu deklarierte abstrakte Methoden
  - ⇨ als auch ererbte abstrakte Methoden
- Welcher Programmcode an dieser Stelle tatsächlich aufgerufen wird, ergibt sich erst durch die Sohnklassen.

Verwendung abstrakter Klassen als Typ

```
public static double flaechenSumme (GeometrischeFigur a, GeometrischeFigur b) {
    return a.flaeche() + b.flaeche();
}
```

- Abstrakte Klassen dürfen als Typen von Variablen verwendet werden.
  - ⇨ In diesen Variablen werden zur Laufzeit nie die Instanzen der Klasse selbst gespeichert, sondern immer nur Instanzen nicht-abstrakter Sohn-Klassen.
  - ⇨ Erinnerung: Instanzen von abstrakten Klassen gibt es nicht.

Von abstrakten Vaterklassen zu nicht-abstrakten Sohnklassen

- ✗ Abstrakte Klassen haben ohne nicht-abstrakte Sohnklassen keinen Nutzen.
- ✗ Erst wenn durch Vererbung und Überschreiben aller abstrakten Methoden nicht-abstrakte Sohnklassen gebildet werden, können Instanzen gebildet werden.
- ✗ Erst wenn zu den nicht-abstrakten Sohnklassen Instanzen gebildet werden, können die abstrakten Methoden, die von der Vaterklasse geerbt werden, ausgeführt werden.
- ✗ Vorgehensweise:
  - ⇨ normale Vererbung: Sohnklasse extends Vaterklasse
  - ⇨ alle abstrakten Methoden der Vaterklassen werden durch nicht-abstrakte Methoden überschrieben

Beispiel

```
public abstract class GeometrischeFigurMitDicke extends GeometrischeFigur {
    public double dicke;
    public double volumen() {
        return flaeche() * dicke;
    }
}
```

Von abstrakten Vaterklassen zu abstrakten Sohnklassen

- Vorgehensweise:
  - ⇨ normale Vererbung: Sohnklasse extends Vaterklasse
  - ⇨ nicht alle abstrakten Methoden der Vaterklassen werden überschrieben oder
    es werden zusätzliche abstrakte Methoden deklariert

- Auch für die abstrakten Sohnklassen gilt wiederum:
  Abstrakte Klassen sind ohne nicht-abstrakte Sohnklassen nicht sinnvoll.

3.12 Interfaces, Schnittstellenvererbung

Interface - Beispiel

```
public interface BegrenzteFigur {
    public abstract void breite();
    public abstract void hoehe();
}
```

Interface

Interfaces sind Klassen, bei denen alle Methoden abstrakt sind.

Nichtabstrakte Klassen, Abstrakte Klassen, Interfaces

	Welche der Objektmethoden sind abstrakt?
Nicht-abstrakte Klasse	keine
Abstrakte Klasse	mindestens eine
Interface	alle

Interfaces - "abstract" weglassen

```
public interface BegrenzteFigur {
    public void breite();
    public void hoehe();
}
```

- ✗ Da Interfaces nur abstrakte Methoden enthalten, darf das Schlüsselwort "abstract" bei den Methoden weggelassen werden. Es ergibt sich implizit durch das Schlüsselwort interface.

Interface

Weitere Eigenschaften von Interfaces:

- ✗ Klassenmethoden sind bei Interfaces nicht erlaubt.

- ✗ Attribute im herkömmlichen Sinne gibt es nicht.
  Anmerkung für den interessierten Leser:
  Ausnahmen sind Klassenattribute und Objektattribute, denen unmittelbar ein fester Wert zugewiesen werden muss, der nicht geändert werden kann.

Abstrakte Klassen - Interfaces

- Interfaces sind Spezialfälle von abstrakten Klassen.

```
public abstract class BegrenzteFigur {
    public abstract double breite();
    public abstract double hoehe();
}
```

- Auch dann, wenn alle Objektmethoden einer Klasse abstrakt sind, unterscheidet sich die abstrakte Klasse von einem Interface syntaktisch:
  - ⇨ Interfaces

```
public interface BegrenzteFigur { ...
```

  - ⇨ abstrakte Klassen

```
public abstract class BegrenzteFigur { ...
```

Schnittstellenvererbung - Beispiel

```
public class Rechteck extends GeometrischeFigur
                implements BegrenzteFigur
{
    Koordinate linksOben;
    Koordinate rechtsUnten;
    public double flaeche () {
        return Math.abs(
            (linksOben.x - rechtsUnten.x) * (linksOben.y - rechtsUnten.y) );
    }
    public double breite() {
        return Math.abs(linksOben.x - rechtsUnten.x);
    }
    public double hoehe() {
        return Math.abs(linksOben.y - rechtsUnten.y);
    }
}
```

Schnittstellenvererbung - Implementierungsvererbung

- Die bisher betrachtete Vererbung wird zur Abgrenzung gern auch genauer als Implementierungsvererbung bezeichnet
  - ⇨ Schreibweise: extends Vaterklasse
  - ⇨ Das Prinzip der Einfachvererbung "Jede Klasse darf nur von einer Klasse erben" bezieht sich auf Implementierungsvererbung!
- Schnittstellenvererbung bezieht sich auf ein oder mehrere Interfaces
  - ⇨ Schreibweise: implements Interface
  - ⇨ Jede Klasse darf von beliebig vielen Interfaces erben (Schnittstellenvererbung)
  - ⇨ Schreibweise: implements Interface1, Interface2, Interface3

Verwendung von Interfaces als Typ

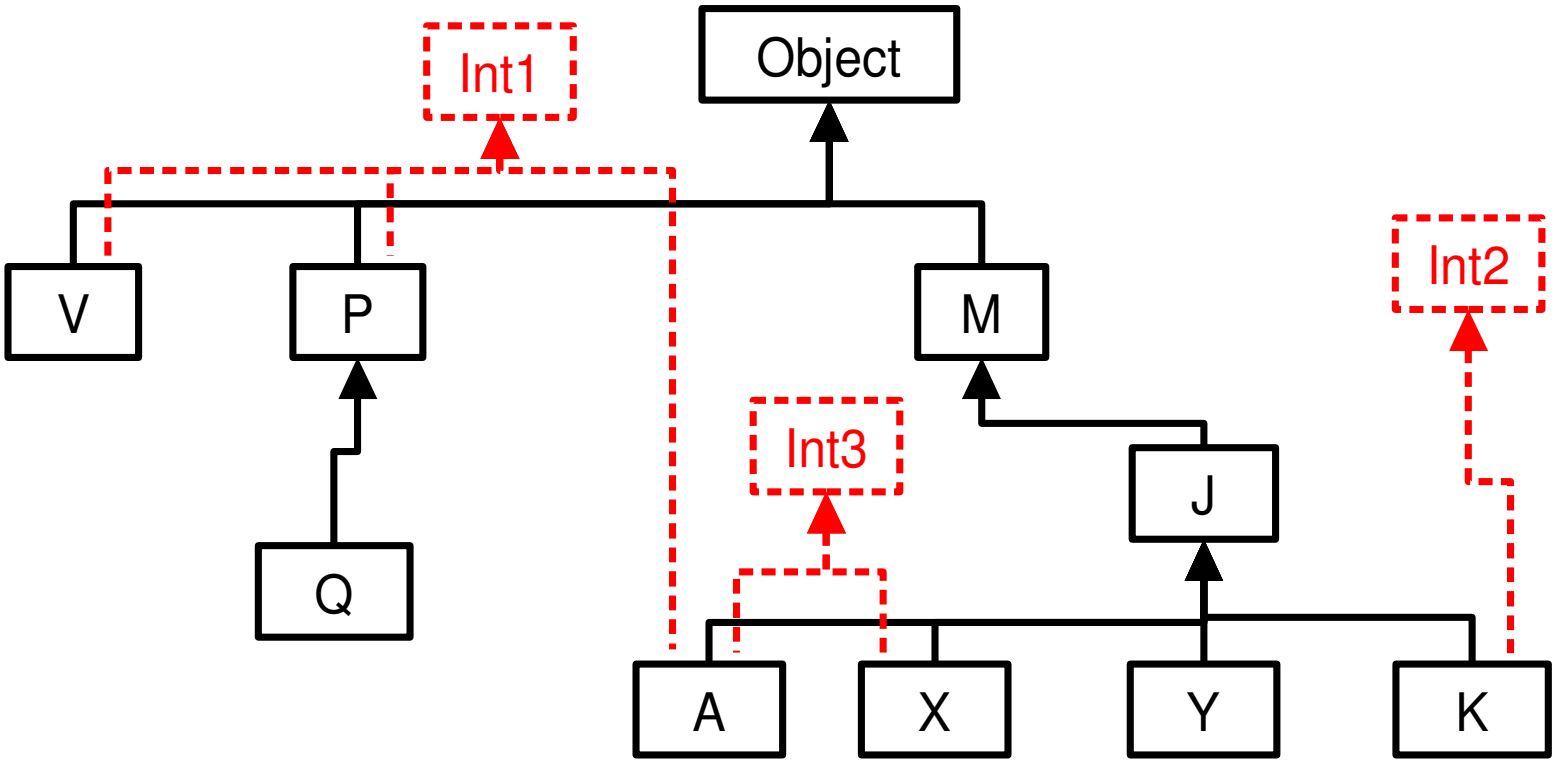
```
public static double flaecheUmgebendesRechteck(BegrenzteFigur a) {
    return a.breite() * a.hoehe();
}
```

- Interfaces dürfen als Typen von Variablen verwendet werden.
  - ⇨ In diesen Variablen werden zur Laufzeit nie die Instanzen der Interfaces selbst gespeichert, sondern immer nur Instanzen nicht-abstrakter Sohn-Klassen.
  - ⇨ Erinnerung: Instanzen von abstrakten Klassen und Interfaces gibt es nicht.

Anmerkungen

- Eine Klasse darf von einer Klasse erben (Implementierungsvererbung) und/oder von einer oder mehreren Schnittstellen (Schnittstellenvererbung).
- Über Schnittstellenvererbung werden nur abstrakte Methoden vererbt (Schnittstellen).
- Programmcode wird nur über die Implementierungsvererbung vererbt.

Implementierungsvererbung, Schnittstellenvererbung



Erläuterung:
Int1, Int2 und Int3 seien Schnittstellen.
Alle anderen Rechtecke seien Klassen (abstrakte oder nicht-abstrakte).

3.13 public, privat, protected

public, private

- *public* bedeutet, dass jede andere Klasse auf diese Attribute und Methoden zugreifen darf.

- Wird stattdessen das Schlüsselwort *private* verwendet, so dürfen andere Klassen nicht auf diese Attribute und Methoden und zugreifen
  - ⇨ Methoden der gleichen Klasse dürfen *private*-Attribute und *private*-Methoden weiterhin verwenden.

Wieso verstecken?

- ✗ Änderungen an *public*-Attributen und *public*-Methoden können dazu führen, dass alle Programmteile überarbeitet werden müssen, die diese verwenden.
  - ⇨ Beispiele für Änderungen, die andere Programmteile "beeinflussen":
    - Ändern des Typs eines *public*-Attributs,
    - Ändern der Parameter einer *public*-Methode,
    - Entfernen einer *public*-Methode, ...

- ✗ Ob ein *public*-Attribut oder eine *public*-Methode benutzt wird, lässt sich bei größeren Programmen nur schwer überprüfen.

... Wieso verstecken?

- ✗ Gibt man Programmcode an andere Entwickler weiter, so muss prinzipiell davon ausgegangen werden, dass alle public-Konstrukte benutzt werden. Bei der Überarbeitung und Pflege des Programmcodes muss dann beachtet werden, dass Änderungen an public-Konstrukten im allgemeinen zu nicht übersetzbarem oder nicht funktionstüchtigen Programmen bei den anderen Entwicklern führt.

- ✗ Bei privaten Attributen und private Methoden sind Änderungen nach außen nicht sichtbar.

  ⇨ Bei der Überarbeitung und Optimierung einer Klasse dürfen private Attribute und private Methoden beliebig verändert werden. Nur die Schnittstelle der public-Methoden muss erhalten bleiben.

Beispiel

```
public class Punkt {
    private double x;
    private double y;
    public Punkt (double a, double b) {
        x = a;
        y = b;
    }
    public double x() {
        return x;
    }
    public double y() {
        return y;
    }
    public double radius() {
        return Math.sqrt(y*y + x*x);
    }
    public double winkel() {
        return Math.asin(y / radius());
    } }
```

Beispiel - Veränderte Implementierung

```
public class Punkt {
    private double radius;
    private double winkel;
    public Punkt (double a, double b) {
        radius = Math.sqrt(a*a + b*b);
        winkel = Math.asin(b / radius);
    }
    public double x() {
        return Math.cos(winkel) * radius;
    }
    public double y() {
        return Math.sin(winkel) * radius;
    }
    public double radius() {
        return radius;
    }
    public double winkel() {
        return winkel;
    } }
```

Anmerkungen zu den beiden Implementierungen

- Beide Implementierungen der Klasse haben nach außen die gleiche Funktion:
  - ⇨ Sie erlauben das erstellen eines Punktes durch Angabe der kartesischen Koordinaten.
  - ⇨ Sie erlauben die Abfrage der Koordinaten des Punktes in kartesischer Darstellung (x,y) und als Polarkoordinaten (radius, winkel).

- Bei der ersten Implementierung wird die Position als kartesiche Koordinaten gespeichert, in der zweiten Implementierung als Polarkoordinaten.

- Die beiden Implementierungen unterscheiden sich in der Performance: Je nachdem welche Koordinatenform öfter benötigt wird, ist die eine oder die andere Implementierung der Klasse effizienter.

Anmerkung

Die beiden Implementierungen der Klasse Punkt liefern nur für positive x- und y-Werte mathematisch sinnvolle Ergebnisse.

Konzept der Kapselung

Systematische Trennung zwischen:

- ✗ Schnittstelle der Klasse nach außen
  - ⇨ die von einer Klasse nach außen zur Verfügung gestellten Attributen und Methoden
  - ⇨ gekennzeichnet mit *public*
- ✗ Interna der Klasse
  - ⇨ Attribute und Methoden, die die Klasse zur Realisierung der nach außen angebotenen Schnittstelle benötigt
  - ⇨ gekennzeichnet mit *private*

protected

- Wird für Attribute oder Methoden *protected* als Sichtbarkeit verwendet, so sind diese sichtbar für:
  - ⇨ alle Sohnklassen (und deren Sohnklassen etc.)

    und
  - ⇨ alle Klassen des gleichen Pakets

Wann soll protected verwendet werden?

- *protected* schränkt die Sichtbarkeit stärker ein als *public* und weniger als *private*.
- Prinzipiell sollte die Sichtbarkeit immer so weit wie möglich eingeschränkt werden, damit bei Änderungen möglichst wenige Klassen überarbeitet werden müssen.
- Ist *private* nicht möglich und das nur, weil Klassen des gleichen Pakets, die Methode bzw. das Attribut auch verwenden müssen, so ist *protected* besser als *public*.

Vergleich

Zugreifbar von Methoden ...	***public***	***protected***	***private***
der eigenen Klasse	x	x	x
einer Klasse des gleichen Pakets	x	x	
einer Sohnklasse	x	x	
einer Klasse eines anderen Pakets, die nicht Sohnklasse ist	x		

Sichtbarkeit von Klassen

- Ähnlich wie bei Attributen und Methoden kann auch bei Klassen die Sichtbarkeit unterschiedlich deklariert werden.

- Von Bedeutung sind dabei nur *public* und *protected*:
  - ⇨ *public*:
    die Klasse kann von allen anderen Klassen verwendet werden
  - ⇨ *protected*:
    die Klasse kann nur innerhalb des eigenen Pakets verwendet werden

- *private*-Klassen sind nur für "Inner-Classes" von Bedeutung - ein Konzept das hier noch nicht vorgestellt werden soll.

3.14 StringBuffer

StringBuffer

Problemstellung:

- Mehrere Strings sollen in einem String aneinander gehängt werden.

Mögliche Lösung:

```
String s = "";
s = s + "Das ";
s = s + "wird ";
s = s + "ein ";
s = s + "langer ";
s = s + "String.";
```

Problem:

- Für jedes Teilergebnis muss eine neue String-Instanz erzeugt werden:

  "", "Das ", "Das wird ", "Das wird ein ", "Das wird ein langer ",
  "Das wird ein langer String"

StringBuffer

Effiziente Alternative:

```
StringBuffer sb = new StringBuffer();
sb.append("Das ");
sb.append("wird ");
sb.append("ein ");
sb.append("langer ");
sb.append("String.");
String s = sb.toString();
```

StringBuffer

String	*StringBuffer*
In jeder Instanz wird eine Zeichenkette gespeichert.	In jeder Instanz wird eine Zeichenkette gespeichert.
Die Zeichenkette, die in einer Instanz gespeichert wird, kann nicht verändert werden.	An die Zeichenkette, die in einer Instanz gespeichert wird, können mit *append* weitere Zeichen angehängt werden.

3.15 Objekte vergleichen

Objekte vergleichen

- Oft wird beim Vergleich von Ausdrücken, deren Typ eine Klasse ist, der Fehler gemacht, dass versehentlich die Referenzen der Objekte und nicht die Inhalte der Objekte verglichen werden (Vergleich mit ==).

- Um den Inhalt der Objekte zu vergleichen, verwendet man die Methode *equals.*

- Die Methode equals ist eine Objektmethode der Klasse Object und steht somit in allen Klassen zur Verfügung.

- In der Klasse Object ist die Methode equals noch so implementiert, dass sie mit == die Referenzen der Objekte vergleicht. Bei Sohnklassen von Object soll diese Methode so überschrieben werden, dass die Inhalte verglichen werden. Bei den meisten Klassen der Standardbibliothek wurde dies bereits in "sinnvoller Weise" realisiert.

Was bedeutet Gleichheit?

- Es ist oft nicht ganz eindeutig, was unter "Gleichheit" zu verstehen ist.
  - ⇨ Bei String-Objekten bedeutet Gleichheit (equals) beispielsweise, dass die Zeichenketten gleich lang und die Zeichen paarweise gleich sein müssen.
  - ⇨ Im einfachsten Fall werden einfach die Attributwerte paarweise miteinander verglichen.
  - ⇨ Aber was ist, wenn die Attribute selbst Objekte sind? Werden dann nur deren Referenzen verglichen (==) oder deren Inhalte (equals)? Und was, wenn die Objekte des Attributs von der gleichen Klasse sind wie das Objekt selbst? Rekursive Strukturen würden ggf. zu einer nicht-terminierenden Implementierung von equals führen.
- Wie sie für eine bestimmte Klasse genau überschrieben wurde, sollte man gegebenenfalls in der Dokumentation nachschlagen.

Objekte vergleichen

```
String a;
String b;
```

Referenzen vergleichen:

```
if (a == b) ...
```

⇨ Zeigen a und b auf das gleiche Objekt?

Inhalte vergleichen:

```
if (a.equals(b)) ...
```

⇨ Enthalten die Objekte, auf die a und b zeigen die gleiche Zeichenkette?

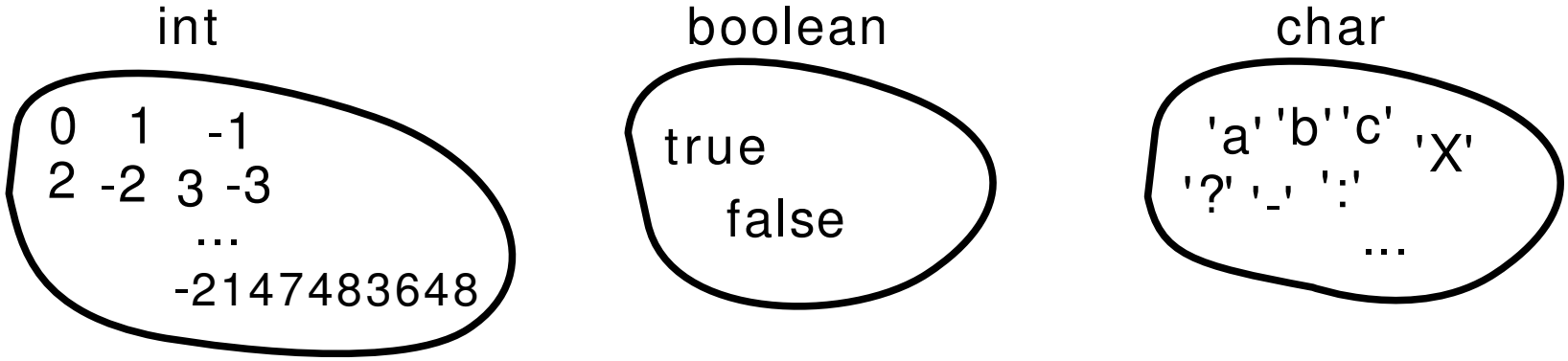
⇨

Kapitel 4

Enumerations

Motivation

Die acht skalaren Datentypen sind vordefinierte Datentypen, die jeweils eine feste, endliche Menge von Elementen repräsentieren. Für jedes Element gibt es in Java eine Konstante.

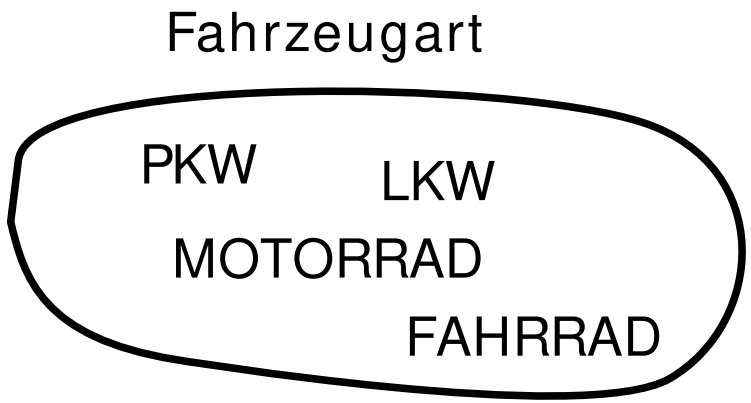


Ziel: Einen "neuen skalaren Datentyp" selbst definieren. Angepasst auf die eigene Problemstellung.

Aufzählungsdatentyp - Konzept

Englischer Begriff: "enumeration type", oft auch kurz "enumeration"

Ein neuer Datentyp wird dadurch deklariert, dass eine endliche Menge von Elementen angegeben wird. Die Namen der Elemente werden aufgezählt.



Enumeration deklarieren

```
public enum Fahrzeugart {
    PKW, LKW, MOTORRAD, FAHRRAD
}
```

Dieser Programmcode wird in einer Datei mit dem Namen Fahrzeugart.java gespeichert. Die Deklaration einer Enumeration ähnelt einer gewöhnlichen Klassendeklaration mit dem Unterschied, dass anstelle des Schlüsselwortes class das Schlüsselwort enum steht.

Auch durch eine enum-Deklaration entsteht eine Klasse. Anders als bei gewöhnlichen Klassen, gibt es bei Enumerations keine Konstruktoren. Es können keine Instanzen erzeugt werden. Die einzigen Werte, die in einer Variable mit dem Typ einer Enumeration gespeichert werden können, sind die Werte, die in der Deklaration aufgezählt werden. Hier: PKW, LWK, MOTORRAD, FAHRRAD

Anmerkungen

- Eine Enum-Deklaration ist der Deklaration einer Klasse sehr ähnlich:
  - ⇨ Sie steht in einer eigenen Datei mit dem Namen Fahrzeugart.java.
  - ⇨ Anstelle des Schlüsselworts class steht das Schlüsselwort enum.
  - ⇨ Mit dem Java-Compiler wird sie in eine Byte-Code-Datei übersetzt.
- Fahrzeugart kann künftig als Typ verwendet werden.
- Die einzigen Werte, die in einer Variable vom Type einer Enumeration gespeichert werden können, sind die Werte, die in der Deklaration aufgezählt werden. Hier: PKW, LWK, MOTORRAD und FAHRRAD.

… Anmerkungen

- Die einzigen möglichen Werte, die in einer Variable vom Typ Enum gespeichert werden können sind die Werte, die in der Deklaration aufgezählt werden.
- Die in der Deklaration aufgezählten Werte sind Instanzen der Klasse. Es gibt zu einer Enum nur diese Instanzen. Enums enthalten keinen von außen zugänglichen Konstruktor. Es können keine weiteren Instanzen erzeugt werden.
- Ob zwei Werte vom Type Enumeration den gleichen Wert habe oder nicht, kann mit dem Operator == ermittelt werden.
- Konvention: Die Namen der Elemente von Enumerations bestehen nur aus Großbuchstaben.

Enumeration verwenden

```
public class FahrzeugartTester {
    public static void main(String[] args) {
        Fahrzeugart x = Fahrzeugart.PKW;
        Fahrzeugart y = Fahrzeugart.LKW;
        Fahrzeugart z = x;
        if (y==Fahrzeugart.PKW)
            System.out.println(x);
    }
}
```

Methoden von Enumerations

Ohne, dass sie deklariert werden müssten, enthalten Enumeration-Klassen einen Satz von Methoden.

Diese Methoden werden anhand des Beispiels Fahrzeugart vorgestellt. Bein anderen Enumeration-Deklarationen sehen diese Methoden fast genauso aus – mit dem kleinen Unterschied, dass überall dort, wo in den Methodendeklarationen der Typ Fahrzeugart steht, jeweils der Name der anderen Enumeration steht.

Enumeration-Methoden

```
public static Fahrzeugart[] values();
```

Als Rückgabewert des Aufrufs von *values()* erhält man einen Array mit den Elementen des Aufzählungsdatentypen.

```
public int ordinal();
```

Jedem Element des Aufzählungsdatentypen ist eine fortlaufende Nummer zugeordnet: int-Zahlen beginnend mit 0. Von einer Instanz der Klasse kann man die Objektmethode *ordinal()* aufrufen, und erfährt so die fortlaufende Nummer des Elements. Die fortlaufende Nummer entspricht genau der Position des Elements in dem Array, den man über den Methodenaufruf *values()* erhält.

… Enumeration-Methoden

```
public String toString();
```

Ganz allgemein liefert die Objektmethode *toString()* eine Stringdarstellung eines Objekts. Sie ist eine Objektmethode der Klasse Object. In Ansatz 2 wurde die Methode so überschrieben, dass man mit ihr den Namen des Elements erhält. Beispiel: Der Aufruf von *Fahrzeug.PKW.toString()* ergibt den Wert "PKW".

```
public static Fahrzeugart valueOf(String s);
```

Zu jeder Enumeration-Klasse gibt es eine Klassenmethode mit dem Namen *valueOf.* Mit dieser Methode wird zu einem String das Element (Instanz der Enumeration-Klasse) bestimmt, das diesen Namen hat.
Beispiel: Mit *Fahrzeugart.valueOf("LKW")* erhält man das Objekt *Fahrzeugart.LKW*.

4.1 Sprachkonstrukte für Enumerations

Static Imports

```
import static Fahrzeugart.*;

public class FahrzeugartTester {
    public static void main(String[] args) {
        Fahrzeugart x = PKW;
        Fahrzeugart y = LKW;
        Fahrzeugart z = x;
        if (y==PKW)
            System.out.println(x);
    }
}
```

Static imports

- Mit einem Static Import können alle statischen Attribute und alle statischen Methoden importiert werden. Die Attribute und Methoden dürfen dann ohne vorangestellten Klassennamen verwendet werden.

- Vergleich: Import von Klassen. Eine oder mehrere Klassen zu importieren hat zur Folge, dass diese Klassen ohne vorangestellten Paketnamen verwendet werden dürfen.

- Static imports können auch für einzelne statische Attribute erfolgen. Beispiel:

```
import static Fahrzeugart.LKW;
```

Iterationen über Collections

```
for (Fahrzeugart f : Fahrzeugart.values())
                System.out.println(f);
```

Die Variable f durchläuft nacheinander die Werte, die in einer Java-Datenstruktur gespeichert sind. Bei der Datenstruktur kann es sich wie hier um einen Array handeln. Java erlaubt an Stelle des Arrays auch andere Datenstrukturen wie etwa Vector oder Set.

Switch

```
public static int faehrpreis (Fahrzeugart fahrzeugart, int personen) {
    switch (fahrzeugart) {
        case LKW : return 80 + 3 * personen;
        case PKW : return (personen < 5 ? 20 : 20 + (personen -4)*3 );
        case MOTORRAD : return 10 + 3 * personen;
        case FAHRRAD : return 2 + 3 * personen;
    }
    return 0;
}
```

Fallunterscheidung über verschiedene Werte, die eine Variable annehmen kann. Entspricht einer gestaffelten Fallunterscheidung mit if-else. Dieses Konstrukt eignet sich besonders für skalare Datentypen und Enums.

4.2 Zusätzlich Objektattribute und Methoden

Attribute und Methoden in Enums

```
public enum Fahrzeugart {
    PKW (true, 80),
    LKW (true, 70),
    MOTORRAD (true, 80),
    FAHRRAD (false, 15);

    private boolean motor;
    private int geschwindigkeit;
    private Fahrzeugart (boolean motor, int geschwindigkeit) {
        this.motor = motor;
        this.geschwindigkeit = geschwindigkeit;
    }

    public double reisedauer (double entfernung) {
        return entfernung / geschwindigkeit ;
    }
}
```

... Attribute und Methoden in Enums

- Da Enums implizit Sohnklassen der Klasse Enum sind, erben sie bereits Methoden und Attribute von dieser Klasse.
- In Enums können zusätzliche Attribute, Methoden und Konstruktoren deklariert werden.

Elementspezifische Konstruktoraufrufe

- Es ist möglich, die Attribute der Enum-Objekte mit für das Objekt spezifischen Werten zu initialisieren.
- Dazu muss zunächst ein geeigneter Konstruktor deklariert werden.
- Bei der Deklaration eines Enums werden die Elementnamen aufgezählt. Hinter einem Elementnamen kann eine Parameterliste stehen (in runden Klammern). Die Parameterliste steht für einen Konstruktoraufruf. Die Instanz, die in dem statischen Attribut gespeichert werden soll, entsteht durch einen Konstruktoraufruf mit dieser Parameterliste.
- Beispiel: Nachfolgendes Sprachkonstrukt aus obiger Enum-Deklaration legt fest, dass in dem PKW-Objekt das Objektattribut *motor* auf *true* und die *geschwindigkeit* auf 80 gesetzt wird.

```
PKW (true, 80),
```

Methoden in Enumerations

In Rumpf von Enums dürfen sowohl Klassenmethoden als auch Objektmethoden deklariert werden.

Beispiele:

```
public double reisedauer (double entfernung) {
    return entfernung / geschwindigkeit;
}
```

```
public static double durchschnittsgeschwindigkeit () {
    double s = 0;
    for (Fahrzeugart f : values())
        s = s + f.geschwindigkeit;
    return s / values().length;
}
```

4.3 Alternativen zu Enums

Alternative aus der nicht-objektorientierten Welt

```
public class Fahrzeugart {
    public static final int PKW = 0;
    public static final int LKW = 1;
    public static final int MOTORRAD = 2;
    public static final int FAHRRAD = 3;
}
```

Erläuterungen

- ✗ Werte werden bei diesem Ansatz immer in int-Variablen gespeichert. Es gibt keine Unterscheidung zwischen unterschiedlichen Enumeration-Typen. Einer Variablen, die zur Speicherung der Fahrzeugart vorgesehen ist, kann auch ein Wert eines anderen Enumeration-Typen (Farbe: ROT, GELB, GRUEN) zugewiesen werden.
  - ⇨ nicht typsicher. schlechter geeignet für komplexe Softwareprojekte
  - ⇨ Java-Enums sind hingegen typsicher

- ✗ Ferner ist zu beachten, dass es bei Java-Enums die Möglichkeit gibt, für die Elemente Objektmethoden zu implementieren. Auf diese Weise können den Elementen des Aufzählungsdatentypen "Funktionen" zugeordnet werden. Dies ist im Sinne einer konsequent objektorientierten Modellierung von Vorteil.

Lösungen bei anderen Programmiersprachen

- ✗ Programmiersprachen ohne ein Sprachkonstrukt Enumerations
  - ⇨ z.B. in C
- ✗ Programmiersprachen mit einem Sprachkonstrukt für Enumerations, bei dem neue skalare Datentypen entstehen.
  - ⇨ typsicher, nicht objektorientiert
  - ⇨ z.B. in Pascal und C++

Kapitel 5

Exceptions

5.1 Exception, throw, try-catch

Exceptions auslösen

- Exceptions können an jeder Stelle im Programmcode mit dem Befehl throw ausgelöst werden:

```
throw new Exception();
```

- Hinter dem throw-Befehl folgt ein Objekt der Klasse *Exception*.

Exceptions abfangen

- Jedes Stück Programmcode kann durch einen try-catch-Block umklammert werden.

```
try {
    Programmcode
} catch (Exception e) {
    Ausnahmebehandlung
}
```

- Der mit *Ausnahmebehandlung* gekennzeichnete Bereich enthält ein Programmstück, das ausgeführt werden soll, wenn in dem als *Programmcode* gekennzeichneten Programmstück eine Exception ausgelöst wird.

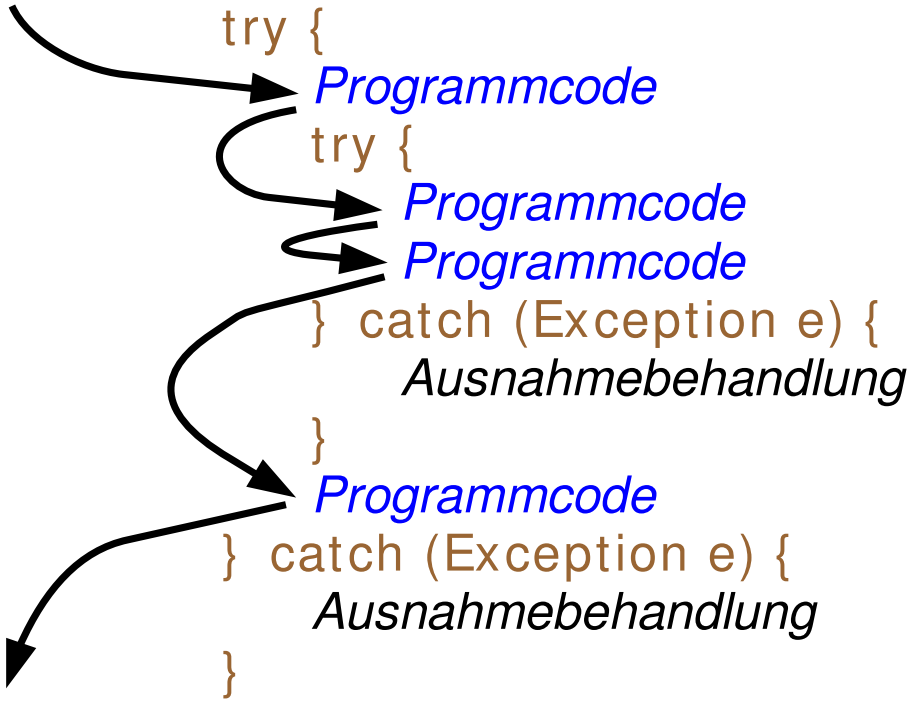
... Exceptions abfangen

- Try-catch-Blöcke können auch Programmstücke umklammern, die selbst wieder try-catch-Blöcke enthalten.

```
try {
    Programmcode
    try {
        Programmcode
        Programmcode
    } catch (Exception e) {
        Ausnahmebehandlung
    }
    Programmcode
} catch (Exception e) {
    Ausnahmebehandlung
}
```

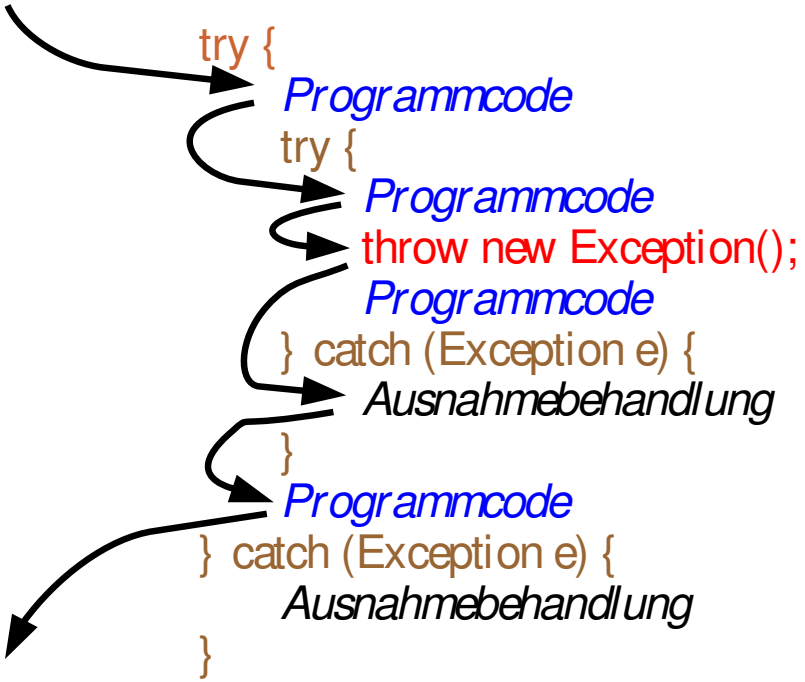
Ablauf ohne Exception

Solange im Programmverlauf keine Exceptions ausgelöst werden, werden nur die try-Blöcke ausgeführt.



Ablauf beim Auslösen einer Exception

Wird eine Exception ausgelöst (throw), so springt das Programm in die Ausnahmebehandlung des try-catch-Blocks, dessen try-Block das Programm zuletzt betreten und noch nicht verlassen hat. Von dort aus wird das Programm normal fortgesetzt.



Ablauf beim Auslösen einer Exception

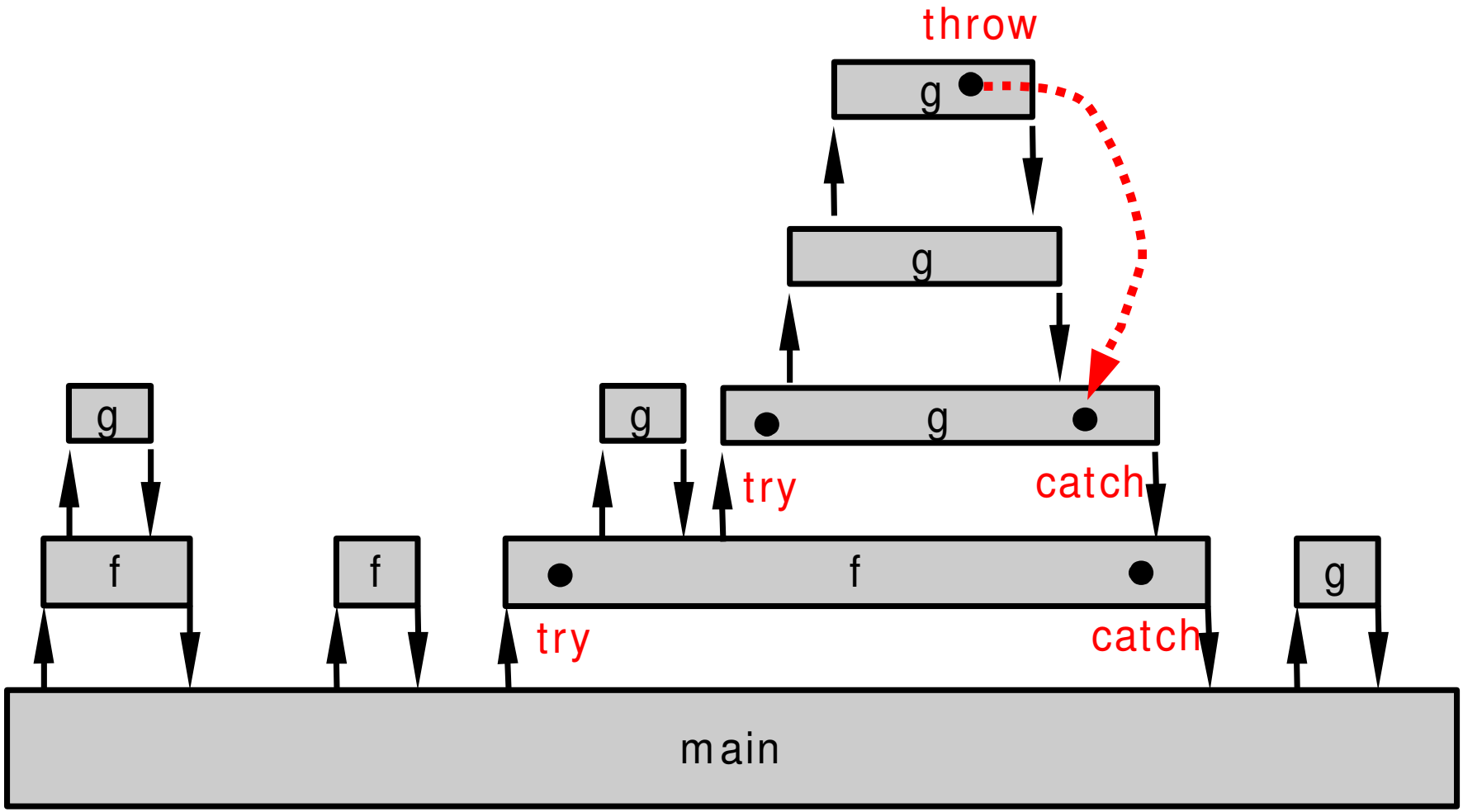
Programmabläufe können vielfältig sein:

- Es können Schleifen und If-Konstrukte durchlaufen werden.
- Es können Methoden aufgerufen werden.
- Methoden können wiederum andere Methoden aufrufen.
- etc.

Ganz egal wie der Programmablauf aussieht, es gilt immer:

- Wird eine Exception ausgelöst (throw), so springt das Programm in die Ausnahmebehandlung des try-catch-Blocks, dessen try-Block das Programm zuletzt betreten und noch nicht verlassen hat. Von dort aus wird das Programm normal fortgesetzt.

Methodenaufrufe und Exceptions



Nicht abgefangene Exceptions

- Wird eine Exception ausgelöst, und das Programm befindet sich gerade nicht in einem try-Block, so beendet die JVM die Ausführung des gesamten Java-Programms.

Irrtümer

- ✗ Exceptions sollen sparsam verwendet werden (nur ausnahmsweise).
  - ⇨ Für eine derartige Einschränkung gibt es keinen Grund.
- ✗ Exceptions müssen unbedingt etwas mit "Fehlern" und "Fehlerbehandlung" zu tun haben.
  - ⇨ Eine derartige Beschränkung gibt es nicht.
- ✗ Exceptions sind langsam.
  - ⇨ Falsch!

... Irrtümer

- ✗ Exceptions müssen möglichst nah an der Stelle abgefangen werden, an der sie ausgelöst wurden. Am besten behandelt man die abgefangene Exception damit, dass man sofort eine neue Exception auslöst.
  - ⇨ Genau so wird gern von Programmierern gearbeitet, die das Prinzip der Exceptions nicht richtig verstanden haben.
  - ⇨ Ergebnis: Ein durch die Exception-Behandlung aufgeblähter Code, der bei Exceptions zu einem "Gewitter" von nachfolgenden Exceptions führt. Programmabläufe sind nur noch schwer nachzuvollziehen.

... Irrtümer

- ✗ Exceptions bringen nichts. Man reicht Informationen über Ausnahmesituationen besser als Teil des Rückgabewertes einer Methode zurück.
  - ⇨ Dieser Ansatz ist prinzipiell möglich. Eine Implementierung mit Exceptions ist jedoch im Allgemeinen bei weitem kompakter, übersichtlicher und effizienter in der Ausführung. Dies gilt besonders bei komplexen Programmen mit einer tiefen Hierarchie von Aufrufbeziehungen.

5.2 Sohnklassen von Exception

Sohnklassen von Exception

- Von der Klasse Exception können Sohnklassen abgeleitet werden.
  - ⇨ Beispiel:

```
public class MeineException extends Exception {
    public int x;
    public MeineException (int a) {
        x = a;
    }
}
```

- In Java gibt es schon zahlreiche Sohnklassen von Exception.

... Sohnklassen von Exception

- In throw-Anweisungen und in catch-Anweisungen können anstelle der Klasse Exception auch Sohnklassen von Exception verwendet werden.
  - ⇨ Beispiel:

```
try {
    ...
    throw new MeineException(3);
    ...
} catch (MeineException e) {
    System.out.println(e.x);
}
```

- Mit dem Exception-Objekt können von der Stelle, an der die Exception ausgelöst wird, Daten an die Stelle transportiert werden, an der die Exception abgefangen wird.

Die catch-Anweisung und Sohnklassen

- In catch-Anweisungen werden immer nur die Exceptions abgefangen, die Instanzen der dort angegebenen Klasse oder Instanzen von Sohnklassen dieser Klasse sind.
  - ⇨ Beispiel 1: Die Exception wird abgefangen

```
try {
    ...
    throw new MeineException(3);
    ...
} catch (MeineException e) {
    ...
}
```

Die catch-Anweisung und Sohnklassen

⇨ Beispiel 2: Die Exception wird abgefangen

```
try {
    ...
    throw new MeineException(3);
    ...
} catch (Exception e) {
    ...
}
```

⇨ Beispiel 3: Die Exception wird nicht abgefangen

```
try {
    ...
    throw new Exception();
    ...
} catch (MeineException e) {
    ...
}
```

Mehrere catch-Blöcke

Zu einem try-Block können auch mehrere catch-Blöcke angegeben werden. In dem Fall, dass in dem try-Block eine Exception ausgelöst wird, springt das Programm dann an die entsprechende Behandlungsroutine.

```
try {
    Programmcode
} catch (Exception1 e) {
    Ausnahmebehandlung1
} catch (Exception2 e) {
    Ausnahmebehandlung2
}
```

5.3 finally

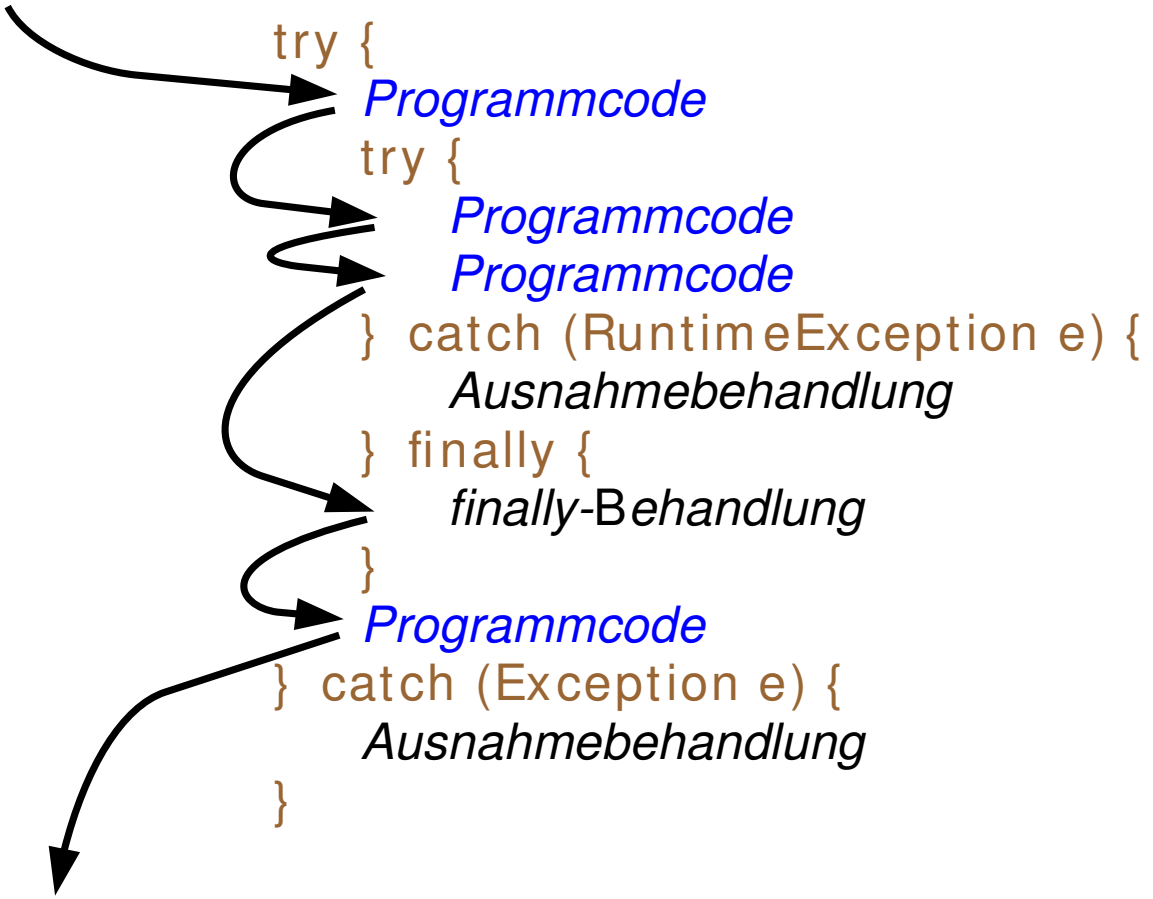
finally

Am Ende eines try-Blocks kann ein finally-Block stehen. Der finally-Block steht hinter dem try-Block und den catch-Blöcken (so vorhanden).

```
try {
    Programmcode
} catch (RuntimeException e) {
    Ausnahmebehandlung
} finally {
    finally-Behandlung
}
```

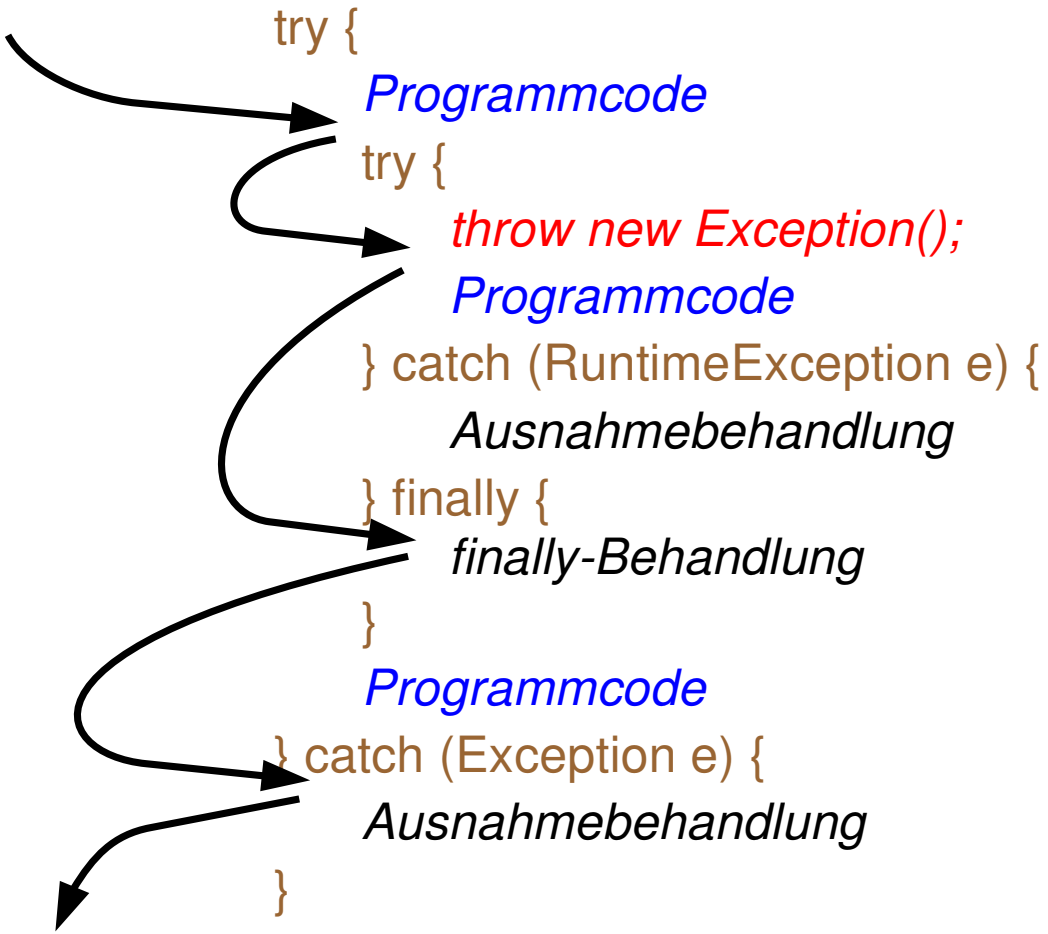
Programmablauf ohne Exceptions

Am Ende des try-Blocks fährt das Programm mit dem finally-Block fort.



Ablauf beim Auslösen einer Exception

Kommt es in einem try-Block zu einer Exception, die in diesem try-Block nicht abgefangen wird, so führt das Programm die entsprechende finally-Behandlungen aus und springt anschließend zum catch-Block.



finally

- Betritt ein Programm einen try-Block, der einen finally-Block hat, so ist gewährleistet, dass das Programm beim Verlassen des try-Blocks immer den finally-Block ausführt.
  Dies gilt in beiden Fällen:
  - ⇨ Fall 1:
    Das Programm erreicht das Ende des try-Blocks.
  - ⇨ Fall 2:
    Das Programm verlässt den try-Block, indem eine Exception ausgelöst wird, die innerhalb des try-Blocks nicht abgefangen wird.

Ressourcen freigeben

- In Programmen werden oft Ressourcen temporär verwendet.
  - ⇨ Beispiele:
    - Zugriff auf eine Datei
    - Datenbankverbindung
- Die Ressourcen werden durch spezifische Befehle angefordert.
- Nach ihrer Benutzung müssen diese Ressourcen wieder freigegeben werden.
- Schema:

```
Ressource Anfordern
Mit der Ressource arbeiten
Ressource freigeben
```

... Ressourcen freigeben

```
Ressource Anfordern
Mit der Ressource arbeiten
Ressource freigeben
```

- Problem: Kommt es bei dem oben angedeuteten Programm beim Arbeiten mit der Ressource zu einer Exception, so wird die Ressource nicht freigegeben.

- Lösung:

```
try {
    Ressource Anfordern
    Mit der Ressource arbeiten
} finally {
    Ressource freigeben
}
```

5.4 throws, RuntimeException

throws

- Methoden müssen eine throws-Deklaration enthalten, wenn es in der Methode zu einer Exceptions kommen kann und diese innerhalb der Methode nicht abgefangen wird.
  - ⇨ Beispiel:

```
public static double durchschnitt (double[] x) throws Exception {
    if (x.length == 0)
        throw new Exception();
    double s = 0;
    for (int i=0; i<x.length; i++)
        s = s + x[i];
    return s/x.length;
}
```

throws

- Werden innerhalb der Methode an verschiedenen Stellen Exceptions ausgelöst, die nicht abgefangen werden, so müssen diese im throws-Abschnitt der Methodendeklaration durch Kommata getrennt aufgezählt werden.

- Ist die Vaterklasse einer Exception im throws-Abschnitt bereits aufgeführt, so muss eine Sohnklasse dieser Exception nicht mehr aufgeführt werden.

- Es gibt im Programmcode von Methoden zwei "Quellen" für Exceptions:
  - ⇨ explizite throw-Aufrufe
  - ⇨ Aufrufe von Methoden, in denen Exceptions nicht abgefangen werden (mit throws-Abschnitt)

Ergebnis eines Methodenaufrufs

Methoden können prinzipiell auf zweierlei Weise beendet werden:

- ✗ Reguläre Beendigung
  - ⇨ Das Programm erreicht das Ende der Methode oder eine return-Anweisung.
  - ⇨ Ist der Rückgabewert der Methode nicht *void*, so wird in der Methode ein Rückgabewert erzeugt, der an die aufrufende Stelle zurückgegeben wird.
- ✗ Exception
  - ⇨ Das Programm stößt auf eine throw-Anweisung, die in der Methode nicht abgefangen wird. Die Exception wird also erst außerhalb der Methode abgefangen oder gar nicht.
  - ⇨ Über das Exception-Objekt können Daten, die in der Methode erzeugt wurden, an die Stelle übergeben werden, an der die Methode abgefangen wird.

RuntimeException

- Die Klasse RuntimeException ist eine Sohnklasse von Exception.
- Besonderheit: Exceptions der Klasse RuntimeException und deren Söhne müssen nicht im throws-Abschnitt aufgeführt werden.
  - ⇨ Beispiel:

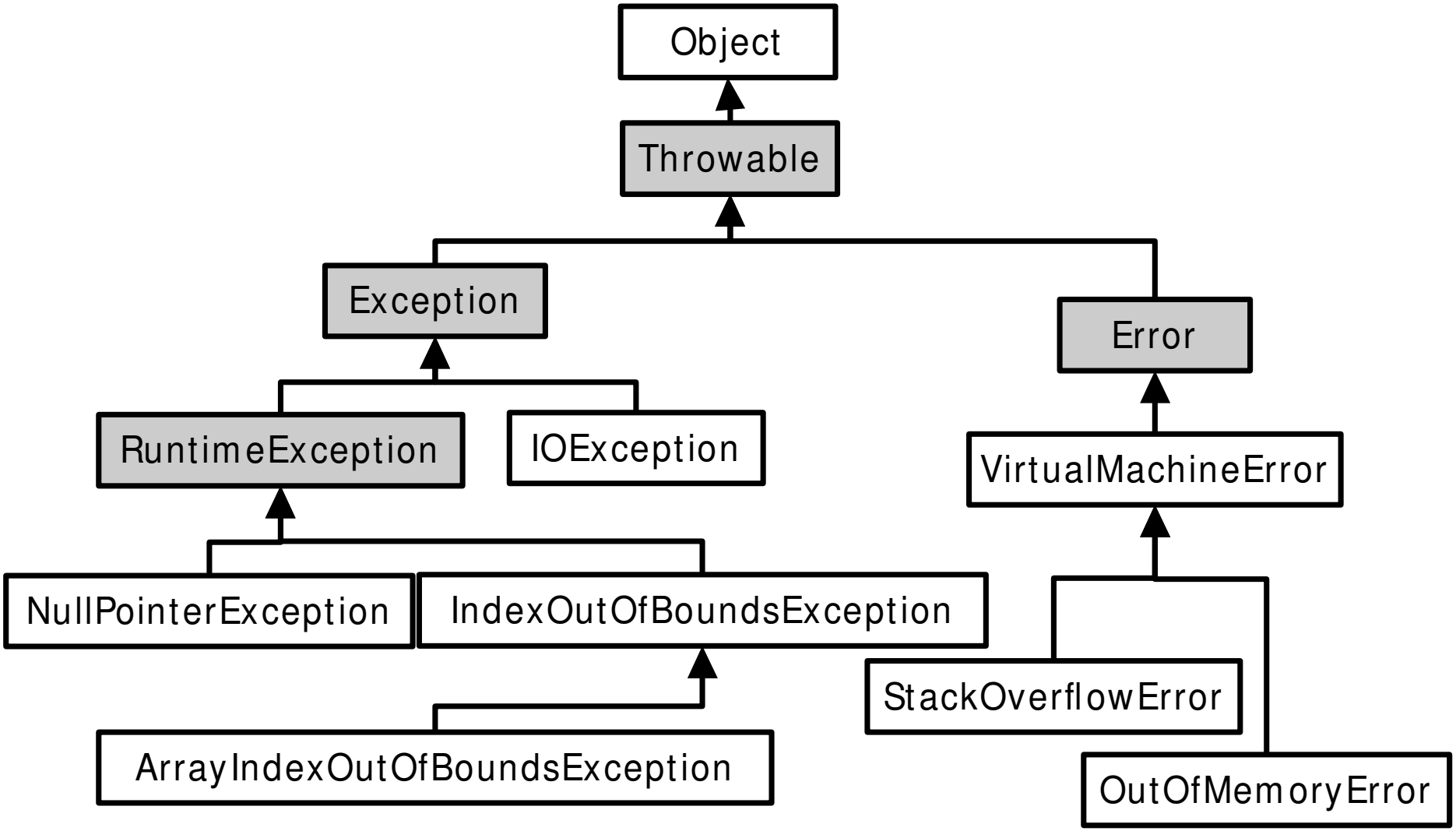
```
public static double durchschnitt (double[] x) {
    if (x.length == 0)
        throw new RuntimeException();
    double s = 0;
    for (int i=0; i<x.length; i++)
        s = s + x[i];
    return s/x.length;
}
```

5.5 Throwable, Error

Throwable, Error

- Mit der Klasse *Error* werden Fehler des Systems signalisiert.
  - ⇨ Beispiele: *StackOverflowError*, *OutOfMemoryError*
- Abgrenzung:
  - ⇨ Exception: Ausnahmesituationen des Programms
  - ⇨ Error: Ausnahmesituationen der JVM
- Die beiden Klassen *Exception* und *Error* haben eine gemeinsame Vaterklasse: *Throwable*.
- Alles, was bisher zu Exceptions gesagt wurde, gilt allgemeiner auch für die Klasse *Throwable*:
  - ⇨ sie können mit *throw* ausgelöst werden
  - ⇨ sie können mit *try-catch* abgefangen werden

Hierarchie


Object
Throwable
Exception
Error
RuntimeException
IOException
VirtualMachineError
NullPointerException
IndexOutOfBoundsException
StackOverflowError
ArrayIndexOutOfBoundsException
OutOfMemoryError

Kapitel 6

Generics

Aufgabenstellung

Es soll eine Klasse programmiert werden, die Binärbäume realisiert.

Binärbäume eignen sich prinzipiell zur strukturierten Aufbewahrung von beliebigen Daten. Es soll eine Binärbaum-Klasse programmiert werden, bei der die Daten in den Knoten einen beliebigen Datentyp haben können. Von Binärbaum zu Binärbaum sollen in den Knoten unterschiedliche Datentypen möglich sein. Innerhalb eines Binärbaums sollen die Daten aller Knoten jedoch den gleichen Datentyp haben.

Generic deklarieren und verwenden

```
public class Binärbaum <L> {
    public L daten;
    public Binärbaum <L> links;
    public Binärbaum <L> rechts;
}
```

```
Binärbaum <String> b = new Binärbaum <String> ();
b.daten = "Bär";
b.links = new Binärbaum <String> ();
b.links.daten = "Maus";
b.rechts = new Binärbaum <String> ();
b.rechts.daten = "Hund";
b.rechts.links = new Binärbaum <String> ();
b.rechts.links.daten = "Pferd";
```

Erläuterungen

- Der Binärbaum wird als eine Generic-Klasse (kurz: Generic) deklariert, also als eine eine Klasse mit einem variablen Datentyp *L* deklariert.

- Im unteren Abschnitt des Beispielcodes wird eine bestimmte Ausprägung des Binärbaums verwendet: Ein Binärbaum mit Daten vom Typ *String*.

- Man bezeichnet
  Binärbaum <String>
  als einen Subtypen von
  Binärbaum <L>

... Erläuterungen

- Im Beispielcode hat b den Typ *Binärbaum <String>*, also den Typ, der entsteht, wenn man in der Klasse Binärbaum den Typ *L* durch *String* ersetzt. Damit hat das Attribut b.daten den Typ *String* und die Attribute *b.links* und *b.rechts* haben den Typ *Binärbaum <String>*. Also müssen auch deren Daten wieder den Typ *String* und deren Teilbäume wieder den Typ *Binärbaum <String>* haben usw.

- Zugriff auf Dateninhalte:

```
System.out.println(b.links.daten.length());
```

- Bei einem Subtyp wird ein variabler Typ durch einen konkreten Typ ersetzt. Nur dieser konkrete Typ kann dann bei dem Subtyp verwendet werden.

6.1 Generics, Grundlagen

Deklaration von Generic-Klassen

- Generic-Klassen sind Klassen, bei denen eine oder mehrere der darin verwendeten Datentypen variabel sind. Andere Bezeichnung: Generics.

- Bei Generics werden in der Klassendeklaration unmittelbar nach dem Klassennamen ein oder mehrere Typvariablen deklarariert. Beispiel:

```
public class Binärbaum <L> {
    ...
}
```

- Im Rumpf der Klasse Binärbaum darf L anstelle eines „normalen Typs“ verwendet werden.

... Deklaration von Generic-Klassen

- Beispiele:
  - ⇨ der Typ eines Attributs hat den Typ L

```
public L daten;
```

  - ⇨ der Typ eines Attributs hat den Typ Binärbaum <L>

```
public Binärbaum <L> links;
```

  - ⇨ eine Objektmethode in der Klasse hat einen Parameter vom Typ L

```
public void linksEinfügen(L x) {
    if (links==null) {
        links = new Binärbaum <L> ();
        links.daten = x;
    } else
        links.linksEinfügen(x);
}
```

Subtyping

- Soll die Generics-Klasse in einer konkreten Ausprägung verwendet werden, so bildet man zu der Generics-Klasse einen Subtypen. Man erhält einen Subtypen von einer Generics-Klasse, indem man die Typvariable durch einen konkreten Typ ersetzt.

```
public class Tester {
    public static void main(String[] args) {
        Binärbaum <String> b = new Binärbaum <String>();
        ...
        Binärbaum <Integer> c = new Binärbaum <Integer>();
        ...
    }
}
```

... Subtyping

- Ein Subtyp einer Generics-Klasse hat die selbe Funktion wie die Klasse, die entstehen würde, wenn man im Rumpf der Generics-Klasse die Typvariablen durch die im Subtyp vorgegeben Typen ersetzt.
  - ⇨ Beispiel: Nachfolgend wird die Klasse *BinärbaumFürStrings* deklariert. Diese hat die gleiche Funktionalität wie der Subtyp *Binärbaum <String>*.

```
public class BinärbaumFürStrings {
    public String daten;
    public BinärbaumFürStrings links;
    public BinärbaumFürStrings rechts;
}
```

Mehrere Typvariablen

- Bei Generic-Klassen mit mehreren Typvariablen werden die Typvariablen innerhalb der spitzen Klammern durch Kommata getrennt aufgezählt. Beispiel:

```
public class Z <R,S> {
    ...
}
```

- Beim Subtyping müssen dann alle Platzhalter durch konkrete Typen belegt werden.

```
Z <Integer,String> x;
```

6.2 Generic-Methoden

Aufgabenstellung

Es soll eine Methode geschrieben werden, bei der einer oder mehrere der Parameter und/oder der Rückgabewert den Typ einer Generic-Klasse hat. Diese Methode kann in irgendeiner Klasse stehen, also nicht unbedingt in einer Generic-Klasse selbst. Die Methode soll so flexibel sein, dass sie mit beliebigen Subtypen der Generic-Klasse verwendet werden kann.

Beispiel: Es soll eine Methode geschrieben werden, der als Parameter ein Binärbaum und Objekt übergeben werden, und die bestimmt, ob das Objekt in einem der Knoten des Baumes enthalten ist. Der Rückgabewert soll vom Typ boolean sein.

Beispiel

```
public class X {
    public static <A> boolean f(Binärbaum <A> b, A x) {
        if (b==null)
            return false;
        A y = b.daten;
        if (x.equals(y))
            return true;
        return f(b.links,x) || f(b.rechts,x);
    }
}
```

Erläuterungen

- Bei Generic-Methoden werden in der Methodendeklaration unmittelbar vor dem Rückgabetyp ein oder mehrere Typvariablen deklarariert. Beispiel:

```
public static <A> boolean f(...) {
    ...
}
```

- In der Methodendeklaration kann nachfolgend diese Typvariable im Rückgabetyp, in Parametertypen und im Rumpf der Methodendeklaration anstelle eines beliebigen Typs erwendet werden.

```
public static <A> boolean f(Binärbaum <A> b, A x) {
    ...
    A y = b.daten;
    ...
}
```

Aufruf von Generic-Methoden

- Der Aufruf von Generic-Methoden geschieht wie der gewöhnlicher Methoden. Je nach Aufruf ergeben sich aufgrund der Typen der bei einem Aufruf übergebenen Parameter bestimmte konkrete Typen für die Typvariablen der Generic-Methode.

- Wird die Methode f in obigem Beispiel mit zwei Parametern vom Typ *Binärbaum <String>* bzw. *String* aufgerufen, so ergibt sich für die Typvariable *A* der Typ *String*.

- Nicht zulässig ist beispielsweise ein Aufruf der Methode *f* mit zwei Parametern vom Typ *Binärbaum <String>* bzw. *Integer.*

6.3 Gebundene Typvariablen

Konzept

Typvariablen sind im allgemeinen Platzhalter für beliebige Datentypen. Diese Freiheit kann gezielt eingeschränkt werden. Eine an eine bestimmte Klasse gebundene Typvariable ist ein Platzhalter für all jene Datentypen die direkt oder indirekt Sohnklasse dieser Klasse sind – oder die Klasse selbst.

Beispiel

Die folgende Generic-Methode erwartet als Parameter einen Binärbaum, bei dem die Elemente Koordinaten oder Instanzen von Söhnen der Koordinate-Klasse sind. Nur damit lässt sie sich aufrufen. Weil dieser Zusammenhang bekannt ist, kann auf auf die spezifischen Methoden und Attribute der Klasse Koordinate zugegriffen werden. Hier: Attribute x und y.

```
public static <A extends Koordinate> void g(Binärbaum <A> b) {
    if (b == null)
        return;
    g(b.links);
    System.out.println(b.daten.x + "," + b.daten.y);
    g(b.rechts);
}
```

6.4 Wildcards

Wildcards

- Das Wildcard-Symbol ? repräsentiert eine Typvariable, der kein Name zugeordnet ist. Das Symbol darf an allen Stellen verwendet werden, wo auf eine Generic-Klasse mit einem variablen Typ Bezug genommen werden soll.

- Typvariablen müssen nicht deklariert werden.
  - ⇨ Wildcards stellen eine Alternative zu methodenbezogenen Typvariablen dar. Methodenbezogene Typvariablen müssen deklariert werden.

- Alle Wildcard-Symbole stehen für unterschiedliche Typvariablen.
  - ⇨ Soll ausgedrückt werden, dass an zwei Stellen im Programmcode ein Typ variabel ist und dass der Typ an beiden Stellen gleich sein soll, so muss hierfür eine klassische Typvariable verwendet werden. Wildcards eignen sich hierfür nicht.

Beispiel

```
public static void ausgeben(Binärbaum <?> b){
    if (b==null)
        return;
    ausgeben(b.links);
    System.out.println(b.daten.toString());
    ausgeben(b.rechts);
}
```

Dies ist gleichbedeutend mit:

```
public <A> static void ausgeben(Binärbaum <A> b){
    if (b==null)
        return;
    ausgeben(b.links);
    System.out.println(b.daten.toString());
    ausgeben(b.rechts);
}
```

Beispiel

In der folgenden Methode werden an zwei Stellen Typvariablen benötigt. Die Typvariablen stehen nicht in Bezug zueinander.

```
public static void ausgeben2(Binärbaum <?> x, Binärbaum <?> y) {
    ausgeben(x);
    System.out.println("---");
    ausgeben(y);
}
```

Würde man in diesem Beispiel anstelle der Wildcards gewöhnliche Typvariablen verwenden, so müssten zwei verschiedene Typvariablen verwendet werden.

Gegenbeispiel

Hier wird die Typvariable an verschiedenen Stellen verwendet. An all diesen Stellen soll der Typ der gleiche sein.

```
public class X {
    public static <A> boolean f(Binärbaum <A> b, A x) {
        if (b==null)
            return false;
        A y = b.daten;
        if (x.equals(y))
            return true;
        return f(b.links,x) || f(b.rechts,x);
    }
}
```

Gebundene Wildcards

- Auch Wildcards dürfen analog zu Typvariablen gebunden sein.
- Die Schreibweise ist analog zu der mit Typvariablen, nur dass anstelle des Namens der Typvariablen das Wildcard-Symbol ? steht. Außerdem findet das Binden nicht dort statt, wo die Typvariable deklariert wird (vor dem Typ des Rückgabewertes), sondern an der Stelle, wo sie verwendet wird.
- Wildcards dürfen nur als Typ-Parameter von Generic-Klassen verwendet werden.

Beispiel

Die bereits vorgestellte Methode g verwendet Subtyping mit einer Typvariablen.

```
public static <A extends Koordinate> void g(Binärbaum <A> b) {
    g(b.links);
    System.out.println(b.daten.x + "," + b.daten.y);
    g(b.rechts);
}
```

Da die Typvariable nur an einer Stelle verwendet wird, kann sie alternativ auch als eine gebundene Wildcard realisiert werden.

```
public static void h(Binärbaum <? extends Koordinate> b) {
    g(b.links);
    System.out.println(b.daten.x + "," + b.daten.y);
    g(b.rechts);
}
```

6.5 Wrapper-Klassen

Skalare Werte – Objekten

Eine Typvariable stellt einen Platzhalter für eine beliebige Klasse dar. Kann man in einem Subtype eine solche Typvariablen auch durch einen skalaren Typ ersetzen? Die Antwort lautet: Nein!

Skalare Wert und Objekte sind in Java strikt voneinander getrennt. Man kann weder einer Variablen, die einen skalaren Typ hat, einen Objektverweis zuweisen noch einer Variablen, die den Typ einer Klasse hat, einen skalaren Wert zuweisen.

Workaround: Wrapper-Klassen

Zu jedem skalaren Datentyp gibt es in der Java-Standard-Library eine Wrapper-Klasse. Eine Wrapper-Klasse enthält ein einziges Objektattribut, in dem der skalare Wert gespeichert wird. Überall dort, wo man ein eine Typvariable durch einen skalaren Typ ersetzen möchte, verwendet man stattdessen die dem skalaren Typ zugeordnete Wrapper-Klasse.

Wrapper-Klassen

Skalarer Typ	Wrapper-Klasse
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Verwendung der Wrapper-Klassen

```
int i;
Integer x;
```

Konvertierug von int nach Integer:

```
x = new Integer(i);
```

Konvertierung von Integer nach int:

```
i = x.intValue();
```

„Autoboxing“

```
int i;
Integer x;
```

Konvertierug von int nach Integer:

```
x = i;
```

Bemerkung: Das ist eine Kurzschreibweise für *x = new Integer(i);*

Konvertierung von Integer nach int:

```
i = x;
```

Bemerkung: Das ist eine Kurzschreibweise für *i = x.intValue();*

Beispiel: Hashtable<String,Integer>

```
Hashtable<String,Integer> h = new Hashtable<String,Integer>();
h.put("Schaufeln", 3);
h.put("Eimer", 7);
int i = h.get("Schaufeln");
```

Erläuterungen:

- In der Hashtable werden Key-Value-Pärchen gespeichert. Der Typ des Keys und der Typ des Values ist bei Hashtable beliebig festlegbar. In dem Beispiel wird der Typ des Keys zu String und der Typ der Values zu Integer. Integer ist ein Ersatz für int.
- In obigem Beispiel hat die Methode *put* als zweiten Parameter den Typ Integer. Anstelle von *Integer*-Objekten werden der Methode direkt *int*-Werte übergeben (Autoboxing).
- In obigem Beispiel hat der Rückgabewert von *get* den Typ *Integer*. Das Ergebnis des Rückgabewerts kann direkt einer *int*-Variablen zugewiesen werden (Autoboxing).