Programmierung I

Bernd Müller

Fakultät Informatik
Ostfalia
Hochschule Braunschweig/Wolfenbüttel

Sommersemester 2012

Vorwort

Zur Veranstaltung

- Name: Grundlagen des Programmierens
- Aus Informatik-Duden: „*Programmierung: Unter Programmierung versteht man zum einen den Vorgang der Programmerstellung und zum anderen das Teilgebiet der Informatik, das die Methoden und Denkweisen beim Entwickeln von Programmen umfasst.*“
- Für die Veranstaltung gilt:
  - Ich setze *nichts* voraus, wir beginnen bei Null
  - Sie lernen Grundlagen der Programmierung / Software-Entwicklung
  - Die Software-Entwicklung geht weit über die Programmierung hinaus. Programmierung ist aber zentral.
  - In Stellenanzeigen lesen Sie daher *Software-Entwickler*, *Java-Entwickler*, . . . und nicht *Programmierer*

Materialien

- Dieses Skript
- Das Buch *Sprechen Sie Java?* von Hanspeter Mössenböck. Anschaffung dringend empfohlen: Im Skript steht *nicht* alles!
- Die Kapitelnummerierung des Skripts entspricht diesem Buch, die Inhalte – mit Erweiterungen und Änderungen – auch
- Alternativ praktisch jedes Buch über Programmierung und Java
- Ein *sehr* ausführliches Java-Buch gibt es auch online: Java ist auch eine Insel. Hat aber über 1300 Seiten und geht weit über ein einführendes Lehrbuch hinaus. Evtl. aber für Sie als Nachschlagewerk geeignet.
- Ebenfalls online sind die Java Tutorials von Sun
- Sehr gute englische Bücher sind *Core Java – Fundamentals* und *Core Java – Advanced Features*

Übungen

- Das Buch von Herrn Mössenböck enthält am Ende eines jeden Kapitels Übungsaufgaben, die Sie unbedingt bearbeiten sollten
- Ich verteile wöchentlich Aufgabenblätter, die Sie ebenfalls bearbeiten und die Lösungen abgeben (Details in Vorlesung)
- Es finden betreute Übungen statt (Details in Vorlesung)
- Ihre Abgaben werden mit Punkten versehen, die zu 30 % in die Gesamtnote eingehen. 70 % der Punkte können durch die Klausur erzielt werden

Grundlagen

Allgemeines

- Programmieren ist eine *sehr* kreative Tätigkeit
- Programmieren ist eine Ingenieurtätigkeit: systematisch, fundiert, umsetzungsorientiert, qualitätsbewusst, ...
- Definition Programmieren: Eine Problemlösung in einer Programmiersprache so genau beschreiben, dass sie sogar ein dummer Computer durchführen kann

Daten und Befehle

Grundelemente von Software

- Daten
  - Text, Zahlen, Bilder, Videos, ...
  - „Bernd Müller“, 1962, ...
- Befehle
  - Addiere, Vergleiche, Springe zu, ...
  - Um das Alter zu berechnen, subtrahiere das Geburtsdatum vom Tagesdatum

Daten

- In unseren heutigen Rechnern werden Daten im Speicher abgelegt
- Speicher besteht aus Zellen (kleine Schachteln)
- Jede Zelle enthält ein Datenelement
- Jede Zelle hat einen Namen oder Adresse
- Die Werte in den Zellen sind *binär codiert*. Ein *Bit*: 0, 1
- Anstatt 1962 steht 10101...
- Größe der Zellen:
  - 1 Byte = 8 Bit
  - 1 Wort = 4 oder 8 Byte

Befehle

- Rechner kennt nur *sehr* einfache Befehle
- Beispiele:

Maschinensprache		Hochsprache
ACC ← x	Lade Zelle x in Akkumulator	z = x + y;
ACC ← ACC + y	Addiere Zelle y	
z ← ACC	Speichere Ergebnis in Zelle z	

- Befehle werden auch binär codiert im Speicher abgelegt
- Man spricht vom *von-Neumann-Rechner*
- Wir wollen nicht mit solch primitiven Befehlen programmieren, sondern mit Hochsprachen (höheren Programmiersprachen)

Programmiersprachen

- Höhere Abstraktionsebene (je höher desto besser)
- Muss vom Compiler auf Maschinenbefehle zurückgeführt werden (bei Java noch etwas anders)
- Typische Programmentwicklung:
  - Spezifikation des Problems
  - Lösungsidee
  - Algorithmus als Lösungsverfahren
  - Programmierung in einer Programmiersprache
  - Übersetzen durch einen Compiler
  - Ausführen

Algorithmen

Definition und Beispiel

- Definition: Ein Algorithmus ist ein schrittweises, präzises Verfahren zur Lösung eines Problems.
- Beispiel: Summe der Zahlen von 1 bis max

Summiere Zahlen ($\downarrow$max, $\uparrow$sum)

1. Setze sum $\leftarrow$ 0
2. Setze zahl $\leftarrow$ 1
3. Wiederhole Schritt 3, solange zahl $\leq$ max
   3.1 Setze sum $\leftarrow$ sum + zahl
   3.2 Setze zahl $\leftarrow$ zahl + 1

Algorithmus und Programm

Ein Algorithmus besteht aus drei Teilen:

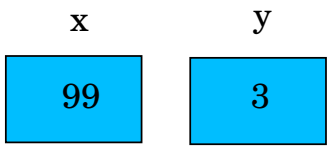
1. Name (Summiere Zahlen)
2. Eingangswerte (max) und Ausgangswerte (sum), Flussrichtung durch Pfeile symbolisiert
3. Folge von Schritten, die Operationen ausführen

- Ein Programm ist die Beschreibung eines Algorithmus in einer bestimmten Programmiersprache
- Der Algorithmus ist also das universellere Konstrukt, ein Programm eine von vielen möglichen Implementierungen

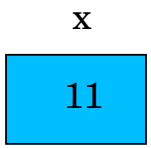
Variablen

Variablen

- Variablen sind benannte Behälter für Werte



- Variable können ihren Wert ändern



- Variable haben einen Datentyp (Menge erlaubter Werte). Es gibt auch Sprachen, die nicht getypte Variablen erlauben
  - Beispiele sind etwa Zeichenketten, ganze oder gebrochene Zahlen

Vorsicht

- Variable gibt es auch in der Mathematik
- Dort versteht man unter einer Variablen *Werte*
- In der Informatik ist eine Variable ein *Behälter* für Werte

Anweisungen

Alle Sprachen sind verschieden ...

- haben aber z.T. sehr ähnliche Grundmuster/Konzepte
- *Anweisungen* greifen auf Werte von Variablen zu und erlauben bestimmte Kontrollmuster
- Wir stellen diese abstrakt, d.h. mit keiner realen (z.B. Java-) Syntax vor

Wertzuweisung

- Berechnet Wert eines Ausdrucks und legt ihn in Variable ab

$$y \leftarrow x + 1$$

- berechnet x+1
- und speichert Ergebnis in y ab

Wertzuweisung (II)

- Links muss immer eine Variable stehen
- Rechts ein Ausdruck aus Variablen und Konstanten (später noch Methodenaufrufen)

Folge (Sequenz)

- Mehrere Anweisungen hintereinander
- Beispiel für *Ablaufdiagramm* siehe Buch

Zusicherung (Assertion)

- Eine Zusicherung ist eine Aussage über den Zustand eines Algorithmus oder eines Programms an einer bestimmten Stelle
- Es hat kommentatorischen Charakter, kann aber auch seit Java 1.4 als Code verwendet werden (Syntax kommt später)

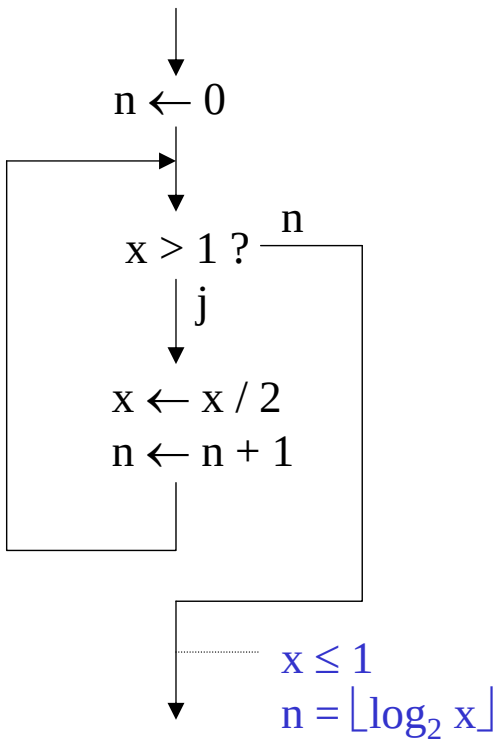
Verzweigung (Selektion, Auswahl)

- Eine (mehrere) Anweisungen sollen nur unter einer bestimmten *Bedingung* ausgeführt werden

```
if (x < y)
    then
        min ← x
    else
        /* x >= y */
        min ← y
```

Schleife (Iteration, Wiederholung)

- Folge von Anweisungen so lange wiederholt ausführen, bis *Abbruchbedingung* eintritt



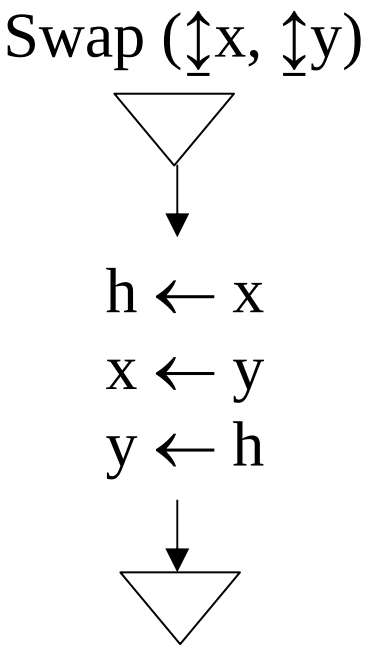
	x	n
1. Besuch	4	0
2. Besuch	2	1
3. Besuch	1	2

Schleife (II)

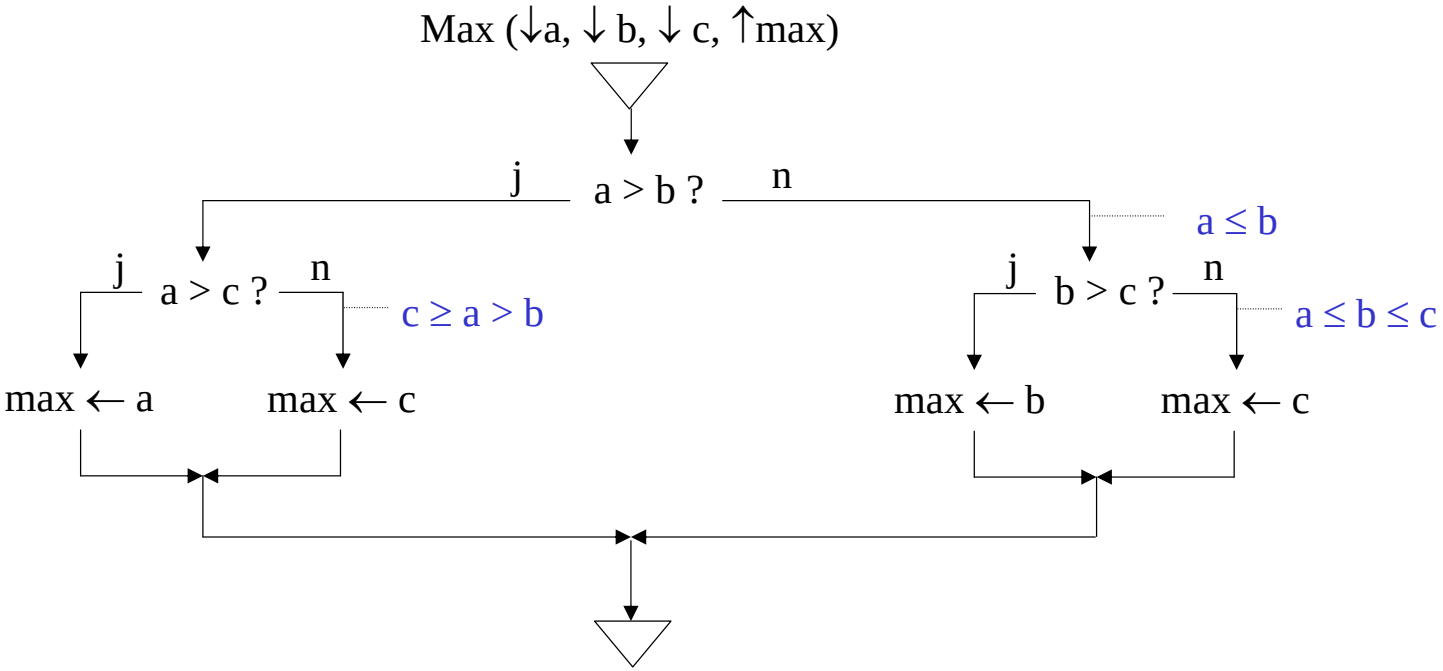
- Schleifen sind für Programmieranfänger sicher eine der schwierigsten Anweisungsarten
- Machen Sie sich unbedingt die Funktionsweise klar und spielen Sie eigene Beispiele durch!!

Beispiele für Algorithmen

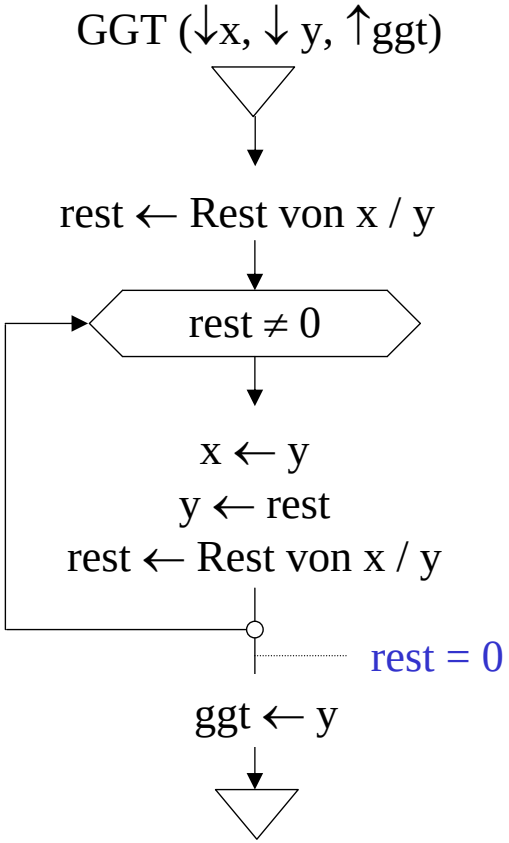
Vertauschen zweier Variableninhalte



Maximum dreier Zahlen



Größter gemeinsamer Teiler



- 2300 Jahre alt. Geht auf griechischen Mathematiker Euklid zurück

Beschreibung von Programmiersprachen

Warum überhaupt beschreiben?

- Computer sind fleißig aber doof
- Man muss ein Programm daher *GAAAANZ* genau beschreiben, damit keine Zweideutigkeiten bestehen
- Dies geschieht in einer bestimmten *Syntax*, die durch eine Grammatik beschrieben wird (Es gibt verschiedene Grammatikformalismen)
- Was das so beschriebene Programm dann tut, beschreibt die *Semantik*
- Im Fall von Java wird Syntax und Semantik in der sogenannten *Language Specification* beschrieben
- Bitte lesen Sie auch die Ausführungen im Buch und bearbeiten Sie die entsprechenden Übungsaufgaben!

Einfache Programme

- Bisher haben wir Algorithmen in sprachunabhängiger Form betrachtet
- Jetzt wollen wir eine konkrete Sprache verwenden und wählen Java, da für die Lehre gut geeignet
- Java wurde 1995 von Sun definiert
- Die damals definierten Ziele
  - Internet-Sprache (auf Applet-Basis)
  - und Programmierung von Embedded Devices

  wurden nicht erreicht.
- Java ist aber *sehr* etabliert und verbreitet und *die* Server-Sprache für Unternehmensanwendungen
- Man unterscheidet
  - Java SE (Standard Edition), die wir nutzen
  - Java EE (Enterprise Edition)
  - Java ME (Micro Edition)
  - und (inoffiziell und je nach Sichtweise) Java für Android

Grundsymbole

Namen

- Namen bezeichnen in einem Programm Dinge wie Variablen, Konstanten, Typen oder Methoden
- Sie bestehen aus Buchstaben, Ziffern, Unterstrich und Dollarzeichen
- Das erste Zeichen darf keine Ziffer sein
- Beispiele sind
  - `x`
  - `x12`
  - `totalSum`
  - `total_Sum`

Namen (II)

- Bemerkung: Jede Sprache hat neben den festen syntaktischen Regeln zur Bildung von Namen Konventionen, die Sie ebenfalls beachten sollten, aber nicht müssen (Gibt aber Abzug in der Klausur ;-) und evtl. Probleme im Entwickler-Team)
- In Java werden zusammengesetzte Namen nach der Kamelnotation (die mit den Höckern) gebildet, also z.B. `totalSum` oder `DasIstEinGanzLangerNameDerNichtWirklichSinnvollIst`.
- Wann ein Name mit einem Groß- oder Kleinbuchstaben beginnt, werden wir noch sehen

Schlüsselwörter

- Schlüsselwörter sind spezielle Namen, die verwendet werden, um Programmteile einzuleiten oder hervorzuheben
- Sie dürfen daher von Ihnen nicht zur Benennung von Variablen, Methoden, ... (als Bezeichner) verwendet werden
- Die nächste Seite führt alle Schlüsselwörter auf
- Die Schlüsselwörter `const` und `goto` werden in Java nicht verwendet, verhindern aber durch ihre Definition als Schlüsselwort eine Verwendung als Bezeichner
- `true`, `false` und `null` sind keine Schlüsselworte, sondern Literale

Schlüsselwörter: komplette Übersicht

abstract	continue	for	new	switch
assert	default	if	package	synchronized
boolean	do	goto	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Zahlen

- Es wird zwischen ganzen Zahlen und Gleitkommazahlen unterschieden
- Ganze Zahlen können dezimal, hexadezimal oder binär (seit Java 7) beschrieben werden (zur Unterscheidung Präfix `0x` für hex, `0b` für binär)
- Beispiele: 127, 0x1A1, 0b10101
- Gleitkommazahlen kommen erst in Kapitel 5

Zeichen und Zeichenketten (Strings)

- Zeichenkonstanten werden in Hochkommas eingeschlossen (`'x'`, `'+'`, `'3'`)
- Achtung: Die Zahl 3 und das Zeichen `'3'` ist etwas *ganz* anderes
- Da Java als Internet-Sprache konzipiert wurde, unterstützt es den Unicode-Zeichensatz und kann sehr viel mehr Zeichen darstellen, als etwa im Deutschen verwendet werden, z.B. $\alpha\beta$ als `\u03b1\03b2` (ausführlicher in Kapitel 8)
- Zeichenkettenkonstante ist Folge von Zeichen durch Anführungszeichen eingeschlossen (`"Ein String"`)
- Metazeichen mit Backslash begonnen: `\n`, `\"`, `\\` (ausführlicher ebenfalls Kapitel 8)

Variablendeklarationen

Variablendeklarationen

- In den meisten Sprachen (nicht Skriptsprachen) müssen Variablen vor ihrer ersten Verwendung *deklariert* werden

```
int x;
short y, z;
```

- Deklaration führt Namen ein und reserviert dem Datentyp entsprechend Speicherplatz für die Variable
- Ein Datentyp definiert eine Menge von Werten. Z.B. bei `int` die Menge der ganzen Zahlen
- Allerdings nur eine endliche Teilmenge der ganzen Zahlen, da im Gegensatz zur Mathematik auf einem Rechner alles endlich ist (außer die Zeit)

Datentypen

- Definiert Menge von Werten und
- Menge von Operationen auf diesem Datentyp
- Der Compiler kann prüfen, ob Operanden und Operationen „zusammen passen“
- Man spricht in diesem Zusammenhang von *statisch getypten* Sprachen, die bestimmte Fehler bereits zur Compile- und nicht erst zur Laufzeit erkennen lassen, wie die meisten Skriptsprachen
- Neben den vordefinierten Typen kann man in Java eigene Typen definieren. Diese nennt man Klassen und Aufzählungstypen (Kapitel 10, 11, 14)

Standardtypen für ganze Zahlen

Typ	Bitanzahl	Bereich (binär)	Bereich (dez.)
`byte`	8	$-2^7 \ldots 2^7 - 1$	-128 ... 127
`short`	16	$-2^{15} \ldots 2^{15} - 1$	-32768 ... 32767
`int`	32	$-2^{31} \ldots 2^{31} - 1$	-2147483648 ... 2147483647
`long`	64	$-2^{63} \ldots 2^{63} - 1$	-9223372036854775808 ... 9223372036854775807

Initialisierung von Variablen

- Variablen können bei der Deklaration gleich *initialisiert*, d.h. mit einem Anfangswert versehen werden:

```
int x = 100;
```

- Falls dies nicht erfolgt, muss vor der ersten (lesenden) Verwendung explizit durch eine Zuweisung ein Wert zugewiesen werden
- mehrere Initialisierungen sind erlaubt:

```
int x = 100, y = 999;
```

Kommentare

- Kommentare sollen den Programmtext erläutern
- Sie gehen von „//“ bis zum Zeilenende:

```
int length; // Laenge in Metern
```

- oder sie gehen von „/*“ bis zu „*/“

```
/* Laenge in Metern.
   Dabei wird immer auf den naechsten vollen
   Meter aufgerundet */
int length;
```

- Es gibt noch Java-Doc-Kommentare (/** ... */) TODO

Tipps

- Wie Sie gesehen haben, vermeide ich Umlaute im Quell-Code
- Ich rate Ihnen, dies auch zu tun
- Außerdem keine Leerzeichen und Umlaute in Datei- und Verzeichnisnamen !
- Englisch ist meist kürzer und prägnanter als Deutsch

Zuweisungen

Zuweisungen

- Sinn: Wert in Variablen speichern
- Wird mit Gleichheitszeichen ausgedrückt, ist aber keine Gleichung im mathematischen Sinne
- Daher nicht „gleich“ gesprochen, sondern
  - ergibt sich aus
  - wird zu
- Zuweisungen werden mit einem Strichpunkt abgeschlossen

Zuweisungskompatibilität

- Typ des Wert einer Zuweisung muss immer gleich oder kleiner dem Typ der Variablen sein

```
int i = 1, j = 2;
short s = 3;
byte b = 4;

i = j;
i = s;
// s = i; // Compiler-Fehler
i = 300;
b = 1;
//b = 300; // Compiler-Fehler
i = s + b; // widening conversion
s = (short) i; // narrowing conversion
```

Konvertierungen

- Im letzten Beispiel *widening* und *narrowing* Konvertierungen
- Sind uns im Augenblick zu schwierig, machen wir später
- Werden im Kapitel 5 der Spezifikation gemacht und sind im PDF 36 Seiten, also nicht ganz trivial
- Falls wir nicht explizit konvertieren, findet der Compiler Typ-Fehler und hilft uns damit, etliche Arten von Fehlern zu vermeiden

Arithmetische Ausdrücke

Binäre Operatoren

+	Addition
-	Subtraktion
*	Multiplikation
/	Division (Ganzzahl/Gleitkomma!)
%	Modulo (Divisionsrest)

Binäre Operatoren (II)

- Es gilt „Punkt vor Strich“
- Klammern möglich
- Gleichrangige Operatoren werden von links nach rechts ausgewertet
- Operanden eines ganzzahligen arithmetischen Ausdrucks müssen `long`, `int`, `short` oder `byte` sein
- Wenn mindestens ein Operand `long` ist, so ist Erbebnistyp `long`, sonst `int`
- Zahlkonstanten sind vom Typ `int`
- Longkonstanten durch anhängen von `l` oder `L`

Beispiele und Cast

```
short s;
int i;
long l;
... l + 1 ... // long
... s + 1 ... // int
s = (short) (s + 1); // type cast
```

- Das „Hochkonvertieren“ geht automatisch
- Das „Runterkonvertieren“ (Bytes abschneiden) nicht, da gefährlich!
- Man spricht von widening and narrowing conversion
- Bei narrowing conversion kann Information verloren gehen
- Z.B. ist (short)(„kleinste Int“) gleich 0 wg Vorzeichen/Zweierkomplement
- Ausführlich in Kapitel 5.1.3 Narrowing Primitive Conversion der Sprachdefinition

Unäre Operatoren

`+` Identitätsoperator: `+x` entspricht `x`

`-` Vorzeichenumkehr

- Unäre Operatoren binden stärker als binäre Operatoren

Inkrement- und Dekrement-Operatoren

- Generell ist `x++;` und `++x` identisch zu `x = x + 1;`
- Bei der Verwendung in Ausdrücken wird es aber kompliziert!
- Wert des Ausdrucks wird vor (`x++;`) oder nach (`++x`) Inkrement/Dekrement ermittelt

```
int i = 5;    // 5
i = i++;      // 5
i = ++i;      // 6
i = i--;      // 6
i = --i;      // 5
```

- Hier relativ sinnlos.
- Es gibt aber sinnvolle Verwendungsmöglichkeiten!

Bit-Shifts

- Die Operatoren << , >> und >>> schieben die Bits einer Ganzzahl um die angegebene Anzahl Bits nach links oder rechts
- Da Ganzzahlen intern als Binärzahlen dargestellt werden, bedeutet dies eine Multiplikation bzw. Division mit 2
- Bei >> wird vorzeichengerecht nachgeschoben, bei >>> werden Nullen nachgeschoben

```
3  <<  2      // 12
3  <<  3      // 24
-3  >>  2     // -1
-3  >>>  2    // 1073741823
```

- Verwendung ist kritisch, da komplizierte Semantik

Zuweisungsoperator

- Bei der Verwendung arithmetischer Operatoren bei einer einfachen Zuweisung, etwa

```
x = x + y;
```

- Kann man auch schreiben

```
x += y;
```

- Diese abkürzende Schreibweise ist für `+`, `-`, `*`, `/`, `%` definiert

Ein-/Ausgabe

Ein-/Ausgabe für sinnvolle Programme nötig, aber komplex

- Bisher Variablen nur durch Zuweisungen oder Initialisierungen Werte erhalten
- Für sinnvolle Programme müssen Daten von anderen Programmen oder von Benutzern erfragt werden können und Daten an andere Programme oder Benutzer ausgegeben werden können
- Man spricht von Ein-/Ausgabe oder von I/O
- Der Sprachkern von Java ist sehr klein. Die Ein-/Ausgabe ist nicht enthalten, sondern in eine Bibliothek ausgelagert
- Um eine generelle Verwendbarkeit zu bekommen, wurde relativ viel Aufwand getrieben und die Verwendung ist aufwändig
- Achtung: Wir verwenden nicht die Bibliothek von Herrn Mössenböck aus dem Buch

Klassen und Objekte für I/O

- Klassen und Objekte kommen erst in Kapitel 10 und 11
- Hier jetzt nur die Verwendung von `System.in` und `System.out` und `Scanner`:

```
System.out.println("Bitte eine Zahl eingeben: ");
Scanner in = new Scanner(System.in);
int i = in.nextInt();
System.out.println("Eingegeben wurde " + i);
```

- Der +-Operator addiert nicht nur, sondern verbindet auch zwei Strings
- Ist einer der beiden Operanden ein String, der andere aber nicht, wird er in einen String konvertiert
- Man sagt, der +-Operator ist *überladen*

Grundstruktur von Java-Programmen

Eingeschränke Struktur

- Eine *sehr* einfache Struktur eines vollständigen Java-Programms

```
class Programmname {

    public static void main(String[] args) {
        ... /* Deklarationen  */ ...
        ... /* Anweisungen */ ...
    }

}
```

- Das blau geschriebene sind Namen und können frei gewählt werden
- „`main`“ ist allerdings ein vordefinierter Name

Unser erstes Java-Programm

```
import java.util.Scanner;

public class PrintSum {

  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    int i, j;

    System.out.println("Bitte zwei Zahlen eingeben: ");
    i = scanner.nextInt();
    j = scanner.nextInt();
    System.out.println("Die Summe ist " + (i + j));
  }

}
```

- Jetzt Demo mit Eclipse

Addendum: Methoden

Methoden

- Einschub: Damit wir Übungsaufgaben machen können ;-)
- Programme in der Praxis viel zu groß, um in einem einzigen Stück geschrieben zu werden
- Daher: in kleinere Anweisungsfolgen zerlegen
- Diese gehören logisch zusammen und erledigen bestimmte Teilaufgaben
- Diese Anweisungsfolge bekommt einen Namen und kann dann unter diesem Namen aufgerufen werden
- Man nennt dies eine *Methode*
- Machen wir in Kapitel 6 Methoden

Beispiel Methoden main() und addiere()

```
package programmierung;

public class Methoden {

  public static void main(String[] args) {
    System.out.println(addiere(5, 9));
  }

  static int addiere(int summand1, int summand2) {
    return summand1 + summand2;
  }

}
```

Beispiel Methoden `main()` und `addiere()`

- `programmierung` ist das Package (Kapitel 18)
- `main()` ist eine besondere Methode, der Einstiegspunkt für das Programm
- `addiere()` ist eine weitere Methode
- Statt `void` `int`: Methode gibt etwas zurück, ist also mathematisch gesehen eine Funktion
- `public` oder nicht: Von außerhalb aufrufbar oder nicht
- `static`: gehört zur Klasse nicht zu Objekt (Abschnitt 11.3)

Konstantendeklarationen

Benannte bzw. symbolische Konstanten

- Bereits Ganzzahl- und String-Konstanten verwendet. Man sagt auch *Literale*
- Besser sind *benannte* Konstanten
- Diese können an *mehrere* Stellen verwendet werden, müssen jedoch bei Bedarf nur an einer Stelle geändert werden
- Per Konvention alle Zeichen groß und durch Unterstrich aneinandergehängt
- `static`: später ($\approx$ gehört zur Klasse)
- `final`: darf nicht verändert werden, also Konstante

Beispiel Konstante

```
public class Brutto {

  static final int MEHRWERTSTEUER_IN_PROZENT = 19;

  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Geben Sie den Netto-Preis ein"
        + " (ganze Euro): ");
    int preis = scanner.nextInt();
    System.out.println("Der Mehrwertsteuersatz betraegt "
        + MEHRWERTSTEUER_IN_PROZENT + " Prozent.");
    System.out.println("Der Brutto-Preis ist "
        + (preis * (100 + MEHRWERTSTEUER_IN_PROZENT) / 100)
        + " Euro.");
  }

}
```

Merke

- Verwende symbolische Konstanten in der der Konvention entsprechenden Schreibung
- Dies erleichtert spätere Änderungen (sogenannte *Wartung*), vor allem bei Änderungen durch andere Entwickler

Namenswahl

Namenswahl (Bezeichner)

- Gute Namenswahl zentral für lesbare Programme
- Namen sollten kurz und aussagekräftig sein
- Hilfsvarible (z.B. Laufvariablen in Schleifen) dürfen kurze Namen, wie `i` oder `x` haben
- Häufig englische Wörter prägnanter. Aber: Fachlichkeit schwierig zu übersetzen
- Variablennamen mit Kleinbuchstaben beginnen und mit „Kamelschreibweise“ fortführen
- Beispiele:
  - `totalSum`
  - `numberOfValues`
  - `alleMeineEntchen`

Verzweigungen

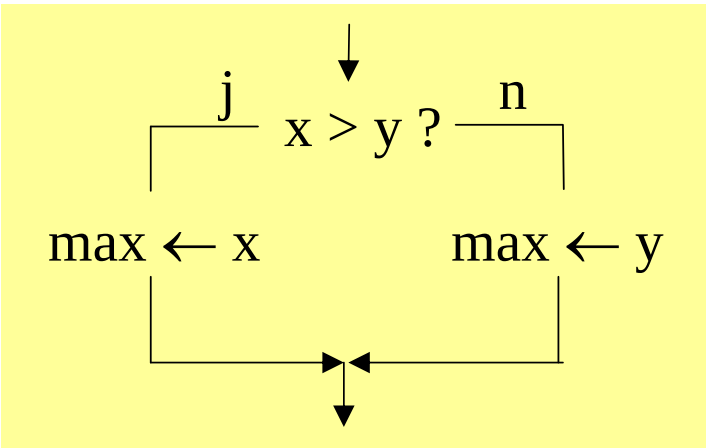
Fallunterscheidung bzgl. verschiedenen Bedingungen

- Problemlösung meist aus mehreren Teillösungen
- Diese werden in Abhängigkeit bestimmter *Bedingungen* ausgewählt
- Java (und viele andere Sprachen) bietet zwei Arten von Anweisungen an, um anhand solcher Bedingungen Verzweigungen für den Programmablauf zu realisieren:
  - *If-Anweisung* als Zweiweg-Verzweigung
  - *Switch-Anweisung* als Mehrweg-Verzweigung

If-Anweisung

If-Anweisung

- Prüft Bedingung. Ausgang: wahr oder falsch
- Dann Ausführung des Then-Zweigs (bei wahr) oder Else-Zweigs (bei falsch)
- In der Literatur und vielen Sprachen als *If-Then-Else* bekannt




```
if (x > y) {
    max = x;
} else {
    max = y;
}
```

- Syntax:

```
IfStatement = "if" "(" Expression ")" Statement
              ["else" Statement].
```

If-Anweisung (II)

- Der Ausdruck muss ein Ergebniss vom Typ `boolean` liefern
- Es gibt *kein* Schlüsselwort `then`, trotzdem wird es auch *if-then-else* genannt
- Die geschweiften Klammern sind optional, jedoch *sehr* zu empfehlen
- Die Schreibung ebenfalls (ist Java-Standard, siehe auch Java-Code-Conventions und *The Elements of Style*)

Dangling Else

- Da If-Anweisungen geschachtelt werden können, kann die folgende Situation entstehen

```
if (a > b)
    if (a > 0)
        max = a;
else
    max = b;
```

- Die Zuordnung des `else` ist nun nicht eindeutig
- Dieses Problem entsteht bei allen Sprachen mit if-then-else und ist in der Literatur als *Dangling-Else*-Problem bekannt
- Es wird durch Definition gelöst: Ein `else` gehört immer zum unmittelbar vorhergehenden `if`

Dangling Else (cont'd)

- Obige Einrückung suggeriert also falsche Zugehörigkeit des else-Zweigs
- Richtig wäre

```
if (a > b)
    if (a > 0)
        max = a;
    else
        max = b;
```

Dangling Else (cont'd)

- Will man die suggerierte Lösung tatsächlich haben, muss man schreiben (gleich mit richtigen Klammern):

```
if (a > b) {
    if (a > 0) {
        max = a;
    }
} else {
    max = b;
}
```

Boolesche Ausdrücke

Vergleichsoperatoren

- Java kennt 6 Vergleichsoperatoren

Operator	**Bedeutung**	**Beispiel**
==	gleich	`x == 3`
!=	ungleich	`x != y`
>	größer	`x > 3`
<	kleiner	`x < y`
>=	größer oder gleich	`x >= 3`
<=	kleiner oder gleich	`x <= y`

Datentyp `boolean`

- Die auf den Mathematiker George Boole zurückgehenden Wahrheitswerte bezeichnet man in Java mit den beiden Literalen `true` und `false`
- Beispiele:

```
boolean p, q;
p = false;
q = 0 < x;
p = (q || p) && x < 10;
```

- Letztes Beispiel verwendet logische Operatoren für zusammengesetzte Vergleiche

Zusammengesetzte Vergleiche

Operanden		**Und-Verknüpfung**	**Oder-Verknüpfung**	**Negation**
`x`	`y`	`x && y`	`x || y`	`!x`
`true`	`true`	`true`	`true`	`false`
`true`	`false`	`false`	`true`	`false`
`false`	`true`	`false`	`true`	`true`
`false`	`false`	`false`	`false`	`true`

- Beispiel:

```
if (0 <= x && x <= 10 || 100 <= x && x <= 110) {
    y = x;
}
```

- Achtung Vorrangregeln (Präzedenz)! Kommt noch

Tipp

- Die Anweisung

```
if (!(x > 0)) {
    x = -x;
}
```

- Lässt sich vereinfachen zu

```
if (x <= 0) {
    x = -x;
}
```

- Negationen sind häufig schwerer zu verstehen

Kurzschlussauswertung

- Die Auswertung von booleschen Ausdrücken erfolgt den Präzendenzregeln (Vorrangregeln) entsprechend solange, bis der Wert des Gesamtausdrucks feststeht. Dann wird abgebrochen.
- Wenn z.B. in

```
if (y != 0 && x/y > 10) { ...
```

- `y` den Wert 0 hat, ist der linke Teilausdruck `false`.
- Damit steht der Wert des Gesamtausdrucks fest und `x/y > 10` wird nicht mehr ausgewertet
- Im Beispiel erspart man sich so einen *Laufzeitfehler*, die Division durch 0
- Beim logischen Oder wird daher abgebrochen, sobald ein Teilausdruck wahr ist

Bitweise logische Verknüpfungen

- Die Operatoren `&`, `|`, `^` und `~` sind für ganze Zahlen und für Wahrheitswerte definiert
- Bei ganzen Zahlen bedeuten sie die bitweise logische Verknüpfung (und, oder, exklusiv-oder, Komplement). Machen wir nicht.
- Bei Wahrheitswerten sind es die normalen Booleschen Verknüpfungen, allerdings ohne Kurzschlussauswertung

Vorrangregeln (Präzendenzregeln)

- Negation bindet stärker als Und, Und stärker als Oder

```
!(y == 0) || 0 < x && x < 10
```

- Zunächst wird `!(y == 0)` berechnet,
- dann `0 < x && x < 10`,
- dann `||` der beiden Ergebnisse
- Falls Sie eine andere Reihenfolge benötigen, mussen Sie explizit klammern, also z.B.

```
(!(y == 0) || 0 < x) && x < 10
```

Switch-Anweisung

`switch`

- Wertet einen Ausdruck vom Typ `int`, `short`, `byte`, `char`, `String` aus und schlägt entsprechend dem Wert des Ausdrucks einen von mehreren möglichen Wegen ein
- Der Ausdruck kann auch eine sogenannte Enum-Variable (oder Enum-Wert, nicht sinnvoll) sein (Kapitel 14)
- Beispiel: Abhängig vom Monat (eine Zahl zwischen 1 und 12) soll die Anzahl der Tage des Monats berechnet werden
- Der Monat ist in der Variablen `month` enthalten, die Anzahl Tage werden in die Variable `days` geschrieben

```
switch (month) {
  case 1:
  case 3:
  case 5:
  case 7:
  case 8:
  case 10:
  case 12:
    days = 31;
    break;
  case 4:
  case 6:
  case 9:
  case 11:
    days = 30;
    break;
  case 2:
    days = 28;
    break;
  default:
    System.out.println("error");
}
```

Switch (II)

- Ausführliche Syntax im Buch oder in Spec 14.11
- `default` ist optional
- `break` ebenfalls
- Aber vorsicht: Ein vergessenes `break` ist eine *ganz* häufige Fehlerquelle.
- Ohne `break` wird beim nächsten `case` weitergemacht

Assertionen bei Verzweigungen

Tipp

- Assertionen dokumentieren den Code
- Bei Verzweigungen ist die Assertion des Then-Zweigs offensichtlich, da ja direkt in der If-Bedingung sichtbar
- Im Else-Zweig, der evtl. viele Zeilen nach dem If kommt, ist die Zusicherung nicht offensichtlich
- Es ist ratsam, sie explizit als Kommentar aufzuschreiben:

```
if (x > y) {
   ...
} else { // 1000 Zeilen weiter:
   // x <= y
   ...
}
```

Echte Assertions in Java

- Seit Java 1.4 gibt es Assertions in Java
- Diese können über das Schlüsselwort `assert` definiert werden.
- Hinter dem `assert` wird ein boolescher Wert erwartet, der zur Laufzeit `true` sein muss
- Ist der Ausdruck `false` wird ein `AssertionError` geworfen
- Optional kann hinter dem Ausdruck durch Doppelpunkt getrennt die Fehlermeldung definiert werden

Echte Assertions in Java (cont'd)

- Ein sinnvoller Einsatz von Assertions ist die Überprüfung der Parameterwerte einer Methode, die wir allerdings erst in Kapitel 6 einführen. Hier ein Beispiel als Vorgriff:

```
void eineMethode(int prozentwert) {
  assert 0 <= prozentwert && prozentwert <= 100:
       "Prozentwert kann nur zwischen 0 und 100 sein";
  ...
}
```

- Assertions sind während der Programmentwicklung sehr hilfreich, da Sie helfen, Fehler zu finden

Echte Assertions in Java (cont'd)

- Ein auf Laufzeitminimum getrimmtes Programm möchte evtl. auf Assertions bzw. deren Überpüfung verzichten. Dies ist der Standardfall bei der JVM
- Man muss Java mit der Option `-enableassertions` bzw. `-ea` aufrufen, um die Überprüfung von Assertions zu erzwingen

Effizienzüberlegungen

Meist mehrere Lösungen für ein Problem

- In der Regel gibt es für ein Problem immer mehrere Lösungen
- Beispiel Maximum dreier Zahlen

```
if (a > b) {
   if (a > c) {
      max = a;
   } else {
      max = c;
   }
} else {
   if (b > c) {
      max = b;
   } else {
      max = c;
   }
}
```

Meist mehrere Lösungen für ein Problem (cont'd)

- Oder ist das besser ?

```
max = a;
if (b > max) {
  max = b;
}
if (c > max) {
  max = c;
}
```

Allgemeines

- Zunächst gilt: „Schön“ programmieren und sich bei Bedarf über Effizienz Gedanken machen
- Wenn Sie immer nur überlegen, wie Sie etwas am effizientesten hinbekommen, werden Sie sehr schlechten Code schreiben
- Was ist Programmqualität? Mögliche Parameter?
  - Statische Kürze (Größe des Programmtextes)
  - Dynamische Kürze (Laufzeit)
  - Lesbarkeit, damit Wartbarkeit

Schleifen

Warum Schleifen?

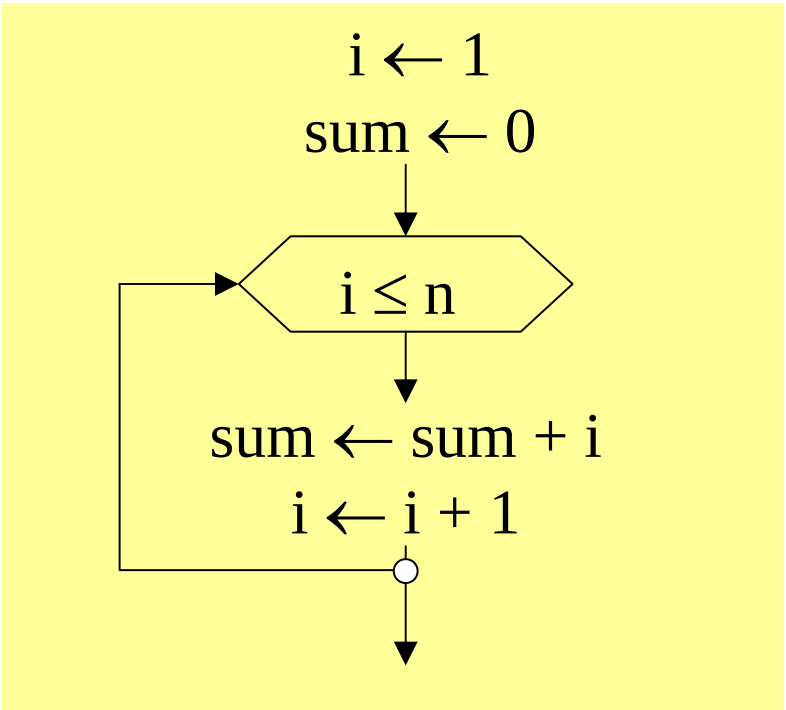
- Praktisch *jeder* Algorithmus benötigt wiederholte Berechnungen
- In der Regel wird wiederholt, bis eine bestimmte Bedingung eintritt, bzw. nicht mehr gilt
- Java (und die meisten anderen Sprachen) kennen drei Arten von Schleifen:
  - Test der Abbruchbedingung am Schleifenbeginn
  - Test der Abbruchbedingung am Schleifenende
  - Vordefinierter Schleifenumfang

While-Schleife

Motivation

- Aufgabe: $\sum_{i=1}^{n} i$ berechnen
- Für bekanntes und festes n kein Problem:
  `sum = 1 + 2 + 3 + ... + n;`
- Aber für beliebiges n ?
- Lösung: Summenbildung durch Wiederholung von `sum = sum + i;`, wobei `i` die Werte 1, 2, 3, ..., n annehmen muss

Lösung mit While-Schleife




```
i = 1;
sum = 0;
while (i <= n) {
   sum = sum + i;
   i++;
}
```

Allgemeines zur While-Schleife

- Die Abbruchbedingung wird *vor* jedem Schleifendurchlauf geprüft
- Sie lesen daher manchmal (seltener) auch *Abweisschleife*
- Es kann sein, dass der Schleifenrumpf *nie* ausgeführt wird
- Die Syntax lautet
  `WhileStatement = "while" "(" Expression ")" Statement.`
- Zu Übungszwecken sollten Sie das Beispiel schrittweise für verschiedene n ausführen

Tipps für das Vorgehen

- Ziel: Einlesen einer Zahlenfolge und diese als Histogramm ausgeben
- Also z.B. für 3 2 4

  ***

  **

  ****
- Aufteilen in Teilaufgaben (Divide and Conquer)

Tipps für das Vorgehen (Divide and Conquer)

- Zunächst Zahlen einlesen!
- Solange etwas zu lesen ist, lies es

```
while (scanner.hasNext()) {
   int i = scanner.nextInt();
   ...
}
```

- Aus API-Doc `hasNext()`: *Returns true if this scanner has another token in its input.*

Typisches Muster (neudeutsch Pattern)

```
while (nächstes Element existiert) {
        lies nächstes Element;
        verarbeite Element;
}
```

Weiteres Vorgehen

- Zahlen sind gelesen
- Jetzt: Die i Sterne ausgeben
- Machen wir mit zweiter Schleife, die i-mal durchlaufen wird und jeweils einen Stern ausgibt
- Wir verwenden eine Hilfsvariable `j`

```
int j = 1;
while (j <= i) {
    System.out.print('*');
    j++;
}
System.out.println();
```

- Dies ist das bereits bekannte Muster !

Kompletter Code-Ausschnitt

```
Scanner scanner = new Scanner(System.in);
while (scanner.hasNext()) {
    int i = scanner.nextInt();
    int j = 1;
    while (j <= i) {
        System.out.print('*');
        j++;
    }
    System.out.println();
}
```

Assertionen bei Schleifen

Offensichtliche Zusicherungen

- Bei einer While-Schleife der Art
  `while (i <= n)` gilt offensichtlich:
  - *In* der Schleife: `i <= n`
  - *Nach* der Schleife: `i > n`

Schleifeninvariante

- Aussage, die in jedem Schleifendurchlauf gleich bleibt
- Und damit Auskunft über den Berechnungsfortschritt gibt

```
i = 1; sum = 0;
while (i <= n) {
  // sum == Summe(1 .. i-1)
  sum = sum + i;
  // sum == Summe(1 .. i)
  i = i + 1;
  // sum == Summe(1 .. i-1)
```

- Am Anfang und am Ende der Schleife gilt dieselbe Assertion
- Schleifenrumpf lässt Invariante unverändert!

Korrektheitsbeweise und Testen

- Wir haben *bewiesen*, dass unser Programm korrekt ist
- Das geht aber nur für kleine Spiel-Programme, nicht für wirkliche Programme
- Daher werden Programme getestet, d.h. für einen (sehr kleinen) Ausschnitt von Eingabedaten wird die Ausgabe auf Korrektheit geprüft

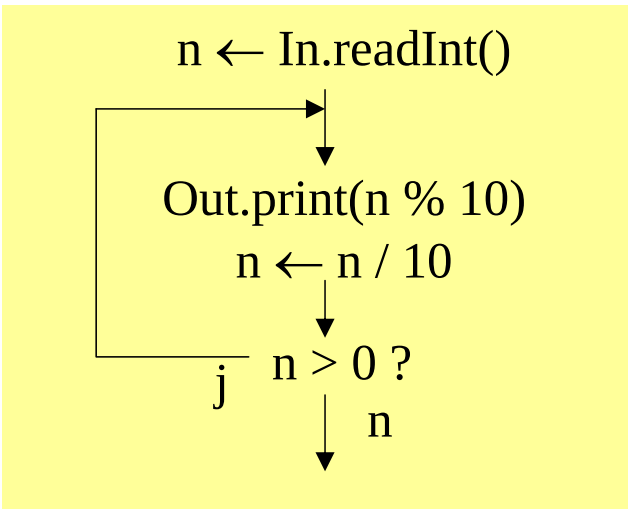
Do-While-Schleife

Motivation

- Manche Algorithmen benötigen Schleifen, die immer mindestens einmal durchlaufen werden
- Man kann das mit einer While-Schleife hinbekommen, es sieht aber evtl. etwas „gekünstelt“ aus
- Sprachen der Pascal-Famile kennen hierfür ein Repeat-Until, Java (C, C++, C#) die Do-While-Anweisung

Beispiel

- Ausgabe der Ziffern einer Zahl in umgekehrter Reihenfolge




```
int n = In.readInt();
do {
   Out.print(n % 10);
   n = n / 10;
} while (n > 0);
```

- `In` und `Out` aus Mössenböcks Bibliothek
- Syntax:

```
DoStatement = "do" Statement "while" "(" Expr ")" ";".
```

For-Schleife

For-Schleife

- Wird verwendet, wenn Anzahl Schleifendurchläufe im Vorhinein feststeht
- For-Schleife besteht aus 4 Teilen:
  - Initialisierungsteil
  - Abbruchbedingung
  - Inkrementierungsteil
  - Schleifenrumpf

Beispiel: Summe der ersten *n* natürlichen Zahlen

```
int sum = 0;
int i;
for (i = 1; i <= n; i++) {
   sum = sum + i;
}
```

- Initialisierung: `i = 1`
- Abbruchbedingung: `i <= n`
- Inkrementierungsteil: `i++`
- Schleifenrumpf: `sum = sum + i`

Äquvalenz zur While-Schleife

- Die For-Schleife ist eine etwas kürzere Schreibweise der While-Schleife
- Transformation als Übung an der Tafel
- Sie sollten bei Iterationen, deren Anzahl im voraus bekannt ist, die For-Schleife verwenden. Das macht den Charakter der Schleife offensichtlich

For-Syntax

```
ForStatement = "for" "(" [ForInit] ";" [Expr]";" [ForUpdate] ")"
ForInit      = Expr {"," Expr}
             | ["final"] Type Var {"," Var}.
ForUpdate    = Expr { "," Expr}.
```

- Man erkennt die Möglichkeit der Deklaration einer Variablen *in* der For-Schleife. Dies ist die vorzuziehende Verwendungsmöglichkeit!

```
int sum = 0;
for (int i = 1; i <= n; i++) {
   sum = sum + i;
}
```

Beispiel: nxn-Multiplikationstabelle

- Aufgabe: Für gegebenes n Ausgabe einer Tabelle mit n Zeilen und n Spalten, in der das Element in Zeile i und Spalte j den Wert $i * j$ enthält
- Offensichtlich benötigt man zwei Schleifen:
  - Eine Schleife über alle Zeilen
  - Eine Schleife (geschachtelt) über alle Spalten
- Da n bekannt ist, nimmt man eine For-Schleife

Beispiel: nxn-Multiplikationstabelle (Code)

```
int n = scanner.nextInt();
for (int i = 1; i <= n; i++) {
  for (int j = 1; j <= n; j++) {
    System.out.print(i * j + " ");
  }
  System.out.println();
}
```

Zählschleife rückwärts

- Für den Inkrementierungsteil sind beliebige Ausdrücke zugelassen, also z.B. auch
  - `i = i - 2`
  - `i--`
  - `...`
- Man kann daher in beliebigen Schrittweiten hoch- oder runterzählen

Abbruch von Schleifen

Beenden von Schleifen

- Die While- und Do-While-Schleife können immer durch geschickte Formulierung der Abbruchbedingung anwendungsbezogen beendet werden
- Der Trick dabei ist die Einführung einer booleschen Variablen, die den Abbruch symbolisiert
- Dieses Muster kann aber zu schlechter lesbaren Strukturen führen und kann in Java durch die Break-Anweisung vermieden werden
- Andererseits kann aber auch die Verwendung eines `break` (oder sogar mehrerer) ebenfalls zu schlechter lesbaren Strukturen führen
- Sie müssen also den richtigen Mittelweg finden

Mit oder ohne break ?!?

```
int sum = 0;
int x = In.readInt();
while (In.done()) {
  sum = sum +x;
  if (sum > 1000) {
    Out.println("Fehler");
    break;
  }
  x = In.readInt();
}
```

```
int sum = 0;
int x = In.readInt();
while (In.done()
       && sum + x <= 1000)
  sum = sum +x;
  x = In.readInt();
}
```

Verlassen mehrerer Schleifen

- Im Standardfall verlässt `break` die unmittelbar umgebende Schleife (`while`, `do`, `for`, aber auch `switch`)
- Man kann auch mehrer Schleifen verlassen, benötigt dann jedoch eine *Marke* (engl. Label), damit klar ist, wo weitergemacht wird

```
myLabel:
  for (;;) {
    ...
    for (;;) {
      ...
      if (...)
        break; //inner loop
      else
        break myLabel; // labeled loop
    }
  }
```

Anmerkungen

- Der Konstrukt `for (;;)` bildet genauso wie der Konstrukt `while (true)` eine *Endlosschleife*
- Das `break` ist eine abgeschwächte Form des `goto`, einem Sprungbefehl. Dieser ist in Java nicht vorhanden (aber reserviertes Schlüsselwort), weil es allgemein als sehr unschönes Konstrukt gesehen wird

Vergleich der Schleifenarten

Schleifenarten

- Alle Schleifenarten sind gleichmächtig und lassen sich ineinander überführen
- Wenn Sie ein wenig Programmiererfahrung haben, werden Sie ein Gefühl dafür entwickeln, welche Schleifenart Sie für welches Problem am besten nutzen

Gleitkommazahlen

Einführung

- Zur Darstellung von Kommazahlen wird in vielen Programmiersprachen das *Gleitkommaformat* verwendet
- Dazu wird eine Kommazahl mit einer Zehnerpotenz multipliziert
- `0.314E1` bedeutet $0,314 * 10^1$, also 3,14
- Exponent kann negativ sein: `31.41E-1` bedeutet $31,4 * 10^{-1}$, also ebenfalls 3,14
- Das Komma gleitet, daher Gleitkommazahl
- Der Exponent 0 kann weggelassen werden

Datentypen float und double

- Zwei Datentypen für Gleitkommazahlen in Java: `float` und `double`
- Man spricht auch von einfacher und doppelter Genauigkeit
- Es werden 32 bzw. 64 Bits zur Darstellung verwendet
- Schreibweise:

```
float x, y;
double z;
```

- Bitte beachten Sie: Es gibt unendlich viele reele und rationale Zahlen aber nur endlich viele floats und doubles, daher Darstellungs- und Rundungsfehler

Anwendungsbeispiel

- Harmonische Reihe für gegebenes n ist definiert als $1/1 + 1/2 + 1/3 + \ldots + 1/n$
- Für eine Reihe liegt die Berechnung mit einer Schleife nahe:

```
int n = scanner.nextInt();
float sum= 0;
for (int i = n; i > 0; i--) {
  sum = sum + (float) 1 / i;  // warum cast?
}
System.out.println("sum = " + sum);
```

Vergleiche von Gleitkommazahlen

- Alle bekannten Vergleichsoperatoren erlaubt: ==, !=, <, <=, >, >=
- Aber Vorsicht: Durch Darstellungs- und Rechenungenauigkeiten können mathematisch gleiche Werte durchaus verschiedene Float- oder Double-Werte haben
- Beispiel Harmonische Reihe von 1 bis n oder von n bis 1
- Daher besser als Prüfung auf Gleichheit Prüfung, ob Differenz im Verhältnis zu Operanden sehr klein
- Gute Nachricht: Man kann in Java auch exakt rechnen: Klasse `BigDecimal`.

Zuweisungskompatibilität

- Es gilt die Zuweisungskompatibilität bzgl. folgender Mengenbeziehung:
  `double` ⊃ `float` ⊃ `long` ⊃ `int` ⊃ `short` ⊃ `byte`
- Es kann bei Zuweisungen allerdings zu einem Genauigkeitsverlust kommen

Beispiele

```
double d;
float f;
int i;

f = i;          // ok
i = f;          // falsch
i = (int) f;    // ok
d = 3.14;       // ok
f = 3.14;       // falsch
f = 3.14f;      // ok, Float-Konstante
```

Nur weil eine Zuweisung erlaubt ist, heißt das noch lange nicht, dass sie exakt ist, wie das folgende Beispiel zeigt.

Beispiele (cont'd)

```
float x = 2147483647;
DecimalFormat format = new DecimalFormat("##########");
System.out.println(x);
System.out.printf("%10.0f\n", x);
System.out.println("x:       " + format.format(x));
System.out.println("x - 64: " + format.format(x - 64));
System.out.println("x - 65: " + format.format(x - 65));
// Zwischenraum zwischen zwei benachbarten Floats
// in diesem Bereich ist 128 !
```

Ausgabe?

```
2.14748365E9
2147483648
x:       2147483648
x - 64: 2147483648
x - 65: 2147483520
```

Beispiele (cont'd)

Das ist allerdings kein Java-Problem, sondern gilt für alle Sprachen, die die Spezifikation IEEE 754 verwenden.
Das folgende C-Programm liefert dieselbe Ausgabe

```
float x = 2147483647;
printf("%10.0f\n", x);
printf("%10.0f\n", x - 64);
printf("%10.0f\n", x - 65);
```

Gleitkommakonstanten

- Im Buch finden Sie eine vereinfachte Darstellung von Gleitkommakonstanten
- In der Sprachdefinition finden Sie die exakte Darstellung von *Floating-Point Literals*

Typregeln in Ausdrücken

- Was passiert bei Ausdrücken, deren Operanden verschiedene Typen haben?
- Regel:

  Der kleinere Operandentyp wird vor Ausführung der Operation in den größeren konvertiert. Der Ausdruck bekommt dann den gleichen Typ wie seine Operanden, zumindest aber den Typ `int`*.*

```
double d; float f; int i; short s;
... f + i ...                 // float
... d * (f + i) ...           // double
... s + s ...                 // int
... f / 3 ...                 // float
... i / 3 ...                 // int
... (float) i / 3 ...         // float
```

Geschwindigkeitsüberlegungen

- . . . sollten Sie sich (zur Zeit) keine machen !!!

Methoden

Motivation

- Komplette Programme sind viel zu groß, um in einem einzigen Stück geschrieben zu werden
- Man zerlegt ein Programm in logisch zusammengehörende kleine Anweisungsfolgen, die Teilaufgaben erledigen
- Diese Anweisungsfolgen bekommen einen Namen und können unter diesem Namen an beliebigen Stellen aufgerufen werden

Parameterlose Methoden

Methode ohne Parameter

- Einfachste Form einer Methode

```
static void printHeader() {
    System.out.println("Artikelliste");
    System.out.println("------------");
}
```

- Jeder Aufruf von `printHeader()` führt die Anweisungsfolge aus
- `static`: Statisch, D.h. Klassenmethode (später mehr)
- `void`: Keine Rückgabe eines Wertes
- `printHeader`: Name der Methode
- `()`: Keine Parameter
- `{ ... }`: Rumpf der Methode

Verwendung

```
class Programm {
  static void printHeader() {
    System.out.println("Artikelliste");
    System.out.println("-------------");
  }
  public static void main(String[] args) {
    ...
    printHeader();
    ...
  }
}
```

- `printHeader()` *Methode* der Klasse `Programm`
- `main()` ebenfalls (das „Hauptprogramm“)
- Eine Klasse kann beliebig viele Methoden haben, die sich gegenseitig aufrufen können

Methodenaufrufe

- Im Rumpf einer Methode kann eine (andere) Methode aufgerufen werden
- In dieser kann ebenfalls wieder eine Methode aufgerufen werden
- Es entwickelt sich eine ganze Folge von Aufrufen, die in der richtigen Reihenfolge abgearbeitet werden
- Wird eine Methode beendete (ihr Ende erreicht), wird bei der nächsten Anweisung nach dem ursprünglichen Methodenaufruf weiter gemacht

Namenskonventionen

- Allgemein: sprechende Namen
- Häufig mit Verb beginnend (`berechneUmsatz()`, `speichern()`, `loescheKunden()`, ...)
- Beginnt mit Kleinbuchstaben
- Zusammengesetzt nach „Kamelschreibweise“

Parameter

Sinn

- Übergabe von Werten zur Verwendung in Berechnungen
- Beispiel: Ausgabe des Maximums zweier Zahlen muss offensichtlich zwei Zahlen übergeben bekommen:

```
static void printMax(int x, int y) {
  if (x > y) {
    System.out.print(x);
  } else {
    System.out.print(y);
  }
}
```

Erläuterungen

- Methode `printMax()` hat zwei Parameter `x` und `y`
- Diese werden wie Variablendeklarationen zwischen den Parameterklammern geschrieben
- Methodenkopf mit Parameterliste wird *Signatur* genannt
- Beim Aufruf müssen zwei Werte mitgegeben werden: `printMax(100, 2 * i);`
- `x` bekommt den Wert 100, `y` den Wert des Ausdrucks `2 * i` zugewiesen
- Dann wird der Rumpf der Methode ausgeführt
- So kann der Code der Methode an unterschiedlichen Stellen mit verschiedenen Parametern aufgerufen werden

Formale und aktuelle Parameter

- Formaler Parameter: Variablendeklaration bei der Deklaration der Methode
- Aktueller Parameter: Wert, der beim Aufruf der Methode übergeben wird. Kann Ausdruck sein, der zu einem Wert evaluiert
- Formale Parameter sind Variablen, die *lokal* zur Methode sind. Sie können nur *in* der Methode verwendet werden
- Falls außerhalb der Methode Variable mit demselben Namen existiert, wird dieser in der Methode „überdeckt“
- Die der Variablen zugeordnete Speicherzelle existiert nur so lange die Methode läuft (genaueres in Abschnitt 6.6)

Parameterübergabe

- Beim Aufruf einer Methode werden aktuelle an formale Parameter übergeben
- Dazu:
  - Ausdrücke der aktuellen Parameter berechnen
  - Wert der Ausdrücke den formalen Parametern zuweisen
- Da Zuweisung gelten Kompatibilitätsregeln (Faustregel: Typ des aktuellen Parameters kleiner oder gleich Typ des formalen Parameters)
- Durch Zuweisung erhält formaler Parameter eine *Kopie* des Werts des aktuellen Parameters
- Änderung der Kopie ändert Original nicht!

Methoden mit Rückgabewerten

Funktionen

- Bezeichnung Funktion hat mathematischen Ursprung
- In nicht objektorientierten Sprachen bezeichnet man Prozeduren, die Werte zurückliefern als Funktionen
- Dies ist bei objektorientierten Sprachen *nicht* der Fall, man spricht von Methoden unabhängig von der Möglichkeit einer Rückgabe
- In der Java-Sprachdefinition wird in diesem Zusammenhang *nie* von Funktionen gesprochen
- Bitte verwenden auch Sie diesen Begriff *nicht*, auch wenn er im Buch verwendet wird!

Beispiel Rückgabewert

- Bestimmung des Maximums ohne Ausgabe sondern als Rückgabe:

```
static int max(int x, int y) {
  if (x > y) {
    return x;
  } else {
    return y;
  }
}
```

- Der Rückgabetyp muss in der Signatur angegeben werden (hier `int`)
- Der Wert wird mit der Return-Anweisung zurückgegeben
- Es muss eine solche Return-Anweisung geben (Compiler/Eclipse achtet darauf)

Verwendung von Rückgabewerten

- Der Rückgabe*wert* ist ein Wert
- Der Methodenaufruf evaluiert zu diesem Wert. Er kann daher überall vewendet werden, wo ein Wert des entsprechenden Typs erlaubt ist
- Für das Maximum von vier Zahlen können wir schreiben:

```
int max = max(max(a,b), max(c,d));
```

- Wir können auch eine weitere Methode schreiben:

```
static int max(int a, int b, int c, int d) {
  return max(max(a,b), max(c,d));
}
```

- Der Rückgabewert einer Methode kann ignoriert werden, d.h. man kann den Methodenaufruf wie eine Anweisung verwenden: `max(a,b)`

Beispiel: Ganzzahliger Zweierlogarithmus

```
static int log2(int x) {
  int result = 0;
  while (x > 1) {
    x = x / 2;
    result++;
  }
  return result;
}
```

Return-Anweisung ohne Rückgabewert

- Kann verwendet werden, um Methode vorzeitig zu beenden
- Beispiel: Ausgabe einer Zahl, falls diese prim ist:

```
static void printIfPrime(int n) {
  if (n > 2 && n % 2 == 0) {
    return;
  }
  for (int i = 3; i * i <= n; i = i + 2) {
    if (n % i == 0) {
      return;
    }
  }
  System.out.println(n + " is prime");
}
```

Alternative I: Ohne return

```
static void printIfPrime(int n) {
  if (n > 2 && n % 2 != 0) {
    int i;
    for (i = 3; (i * i <= n) && (n % i != 0);
         i = i + 2) ; // leerer For-Rumpf
    if (i * i > n) {
      System.out.println(n + " is prime");
    }
  }
}
```

Alternative II: Test, d.h. mit boolescher Rückgabe

```
static boolean isPrime(int n) {
  if (n > 2 && n % 2 == 0) {
    return false;
  }
  for (int i = 3; i * i <= n; i = i + 2) {
    if (n % i == 0) {
      return false;
    }
  }
  return true;
}
```

- Aufgabe: Probieren Sie dies in Eclipse und mit `javac` und lassen abwechselnd eine der drei Return-Anweisungen weg. Was fällt Ihnen auf?

Lokale und globale Namen

Lokal versus Global

- Methode kann auch Deklarationen enthalten
- Alle in einer Methode deklarierten Namen (auch formale Parameter) sind *lokal* zu dieser Methode
- Alle außerhalb einer Methode deklarierten Namen sind *global*

Lokale Variablen

```
static void myMethod(int x) {
  int y;
  float z;
  ...
}
```

- Deklariert drei lokale Variablen `x`, `y` und `z`
- Sie sind nur in der Methode `myMethod` sichtbar, nicht außerhalb
- Ihr Speicherplatz wird jedesmal neu angelegt, wenn `myMethod` aufgerufen wird
- Am Ende der Methode wird ihr Speicherplatz wieder freigegeben

Globale Variablen

- Variable ist global, wenn sie außerhalb einer Methode (auf Klassenebene) deklariert wurde

```
class Program {
  static int a;
  static float b;

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

- `static` im Augenblick nicht weiter betrachtet (Kapitel 11)
- Globale Variable in allen Methoden sichtbar
- Globale Variable in allen Initialisierungsausdrücken danach sichtbar

Globale Variablen (cont'd)

- `a` und `b` in `myMethod()` und `main()` sichtbar
- Speicherplatz einer globalen Variable wird bei Programmstart angelegt und am Programmende wieder freigegeben
- Also überleben Werte globaler Variablen Methodenaufrufe, während dies lokale Variablen nicht tun
- Falls lokale Variable gleichen Namens globale Variable überdeckt, mit Klassennamen gefolgt von Punkt als Präfix eindeutig machen (siehe Abschnitt 6.5)

Lokale Konstanten

- Konstanten werden mit dem *Modifier* `final` deklariert
- Sie dürfen global oder lokal deklariert werden
- Die lokale Verwendung ist unüblich, da Konstanten typischerweise in einem größeren Bereich verwendet werden

```
static void myMethod() {
  final boolean DEBUG = true;
  ...
  if (DEBUG) {
    Systme.out.println(...);
  }
}
```

Beispiel: Summe einer Zahlenfolge

- Ziel: Methode `add(int x)`, die Parameter `x` zu Variablen `sum` addiert
- Versuch:

```
class Program {
  static void add(int x) {
    int sum = 0;
    sum = sum + x;
  }
  public static void main(String[] args) {
    add(3);
    add(17);
    add(5);
    System.out.println(sum);
  }
}
```

Fehler?

- `main()` kann nicht auf `sum` zugreifen
- `sum` wird bei *jedem* Aufruf angelegt, initialisiert und dann wieder freigegeben

Neuer Versuch

```
class Program {

  static int sum = 0;

  static void add(int x) {
    sum = sum + x;
  }

  public static void main(String[] args) {
    add(3);
    add(17);
    add(5);
    System.out.println(sum);
  }
}
```

Nebeneffekte (oder Seiteneffekte)

- Methoden sollen Dinge berechnen und damit bestimmte Effekte haben (Variablen schreiben, Dateien schreiben, . . .)
- Methoden mit Rückgabewerten werden funktional genutzt (daher „Funktion“ im Buch). Eine mathematische Funktion hat keine Nebeneffekte
- Wir sollten daher auch in der Programmierung versuchen, dies zu erreichen
- Falls wir dies (auch welchen Gründen auch immer) nicht erreichen können, so ist dies zu dokumentieren!

Beispiel Nebeneffekt

- Schlecht:

```
static int n = 0;
static int getN() {
  return n++;
}
```

- Gut (weil nebeneffektfrei):

```
static int n = 0;
static int getN() {
  return n;
}
static void incN() {
  n++;
}
```

Wann lokale, wann globale Variablen?

- Regel: möglichst lokal
- Vorteil: Man kann in verschiedenen Methoden Variablen mit gleichem Namen verwenden, ohne dass sie sich stören
- Vorteil: Man muss sich keine Gedanken machen, welche Namen schon verwendet werden
- Vorteil: Deklaration und Verwendung nahe zusammen, daher besser verständlich
- Vorteil: Variable kann durch andere Methoden nicht verändert werden. Methode daher verstehbar und verifizierbar, ohne auf andere Methoden schauen zu müssen
- Zum Schluss: Machen Sie sich keine Sorgen über die Effizienz (Anlegen, Freigeben sind vernachlässigbar)

Sichtbarkeitsbereich von Variablen

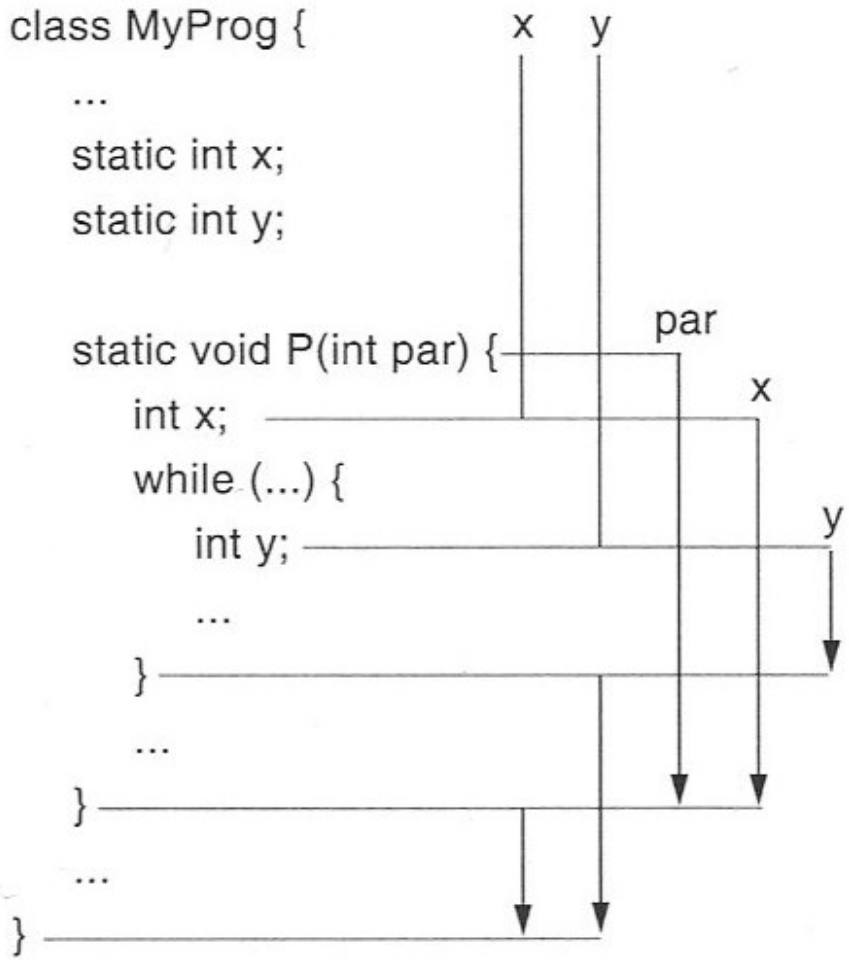
Definition

- Sichtbarkeitsbereich oder Gültigkeitsbereich einer Variablen ist der Bereich, in der die Variable verwendbar (sichtbar) ist
- Er erstreckt sich von der Deklaration bis an das Ende des Blocks (in der Regel geschweifte Klammer zu)
- Bei Methoden also bis zum Ende der Methode
- Bei Klassen bis zum Ende der Klasse
- Bei anderen Blocken bis zum Blockende (for, while, . . .)

Überlagerung von Sichtbarkeitsbereichen

- Die Deklaration einer Variablen in einer Methode überdeckt bereits gültige Deklarationen mit demselben Namen
- Da Methoden nicht geschachtelt werden dürfen, ist dies nur auf einer Ebene möglich
- Allgemeine Blöcke können jedoch beliebig tief geschachtelt werden
- Dort deklarierte Variablen sind bis ans Ende des Blockes sichtbar, dürfen jedoch nicht denselben Namen wie lokale Variablen haben

Beispiel Sichtbarkeitsbereiche



Zugriff auf überlagerte globale Variable

- Zugriff auf überlagerte globale Variable durch Verwendung des Klassennamen als Präfix (getrennt durch Punkt)

```
class Klasse {
  static int x;

  static void foo(int x) {
    int x; // Alternative zu Parameter !
    ...
    x = ... // die lokale Variable
    ...
    Klasse.x = ... // die globale Variable
  }
}
```

Lebensdauer von Variablen

Lebensdauer und Sichtbarkeit

- Sichtbarkeit ist eine statische Eigenschaft (zur Compile-Zeit)
- Lebensdauer ist eine dynamische Eigenschaft (zur Laufzeit)
- Wenn eine Variable nicht sichtbar ist, heißt das nicht, dass sie zur Laufzeit nicht existiert
- Globale Variablen leben während des gesamten Programmlaufs
- Lokale Variablen werden beim Betreten des Blocks angelegt und beim Verlassen wieder gelöscht
- Da Methoden Methoden aufrufen können, müssen lokale Variablen nach dem Last-In-First-Out-Prinzip (LIFO), als Datenstruktur auch Keller oder Stack genannt, verwaltet werden. Dazu ein Beispiel

Demo Lebensdauer mit Stack für lokale Variablen

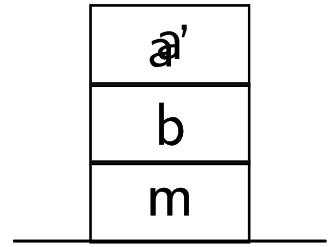
```
class Program {

  static int g;

  static void meth1() {
    int a;
    ...
  }

  static void meth2() {
    int b;
    ... meth1(); ... ... meth1(); ...
  }

  public static void main(String[] args) {
    int m;
    ... meth2(); ...
  }
}
```



Überladen von Methoden

Eindeutigkeit von Namen

- Prinzipiell alle in einem Block deklarierten Namen verschieden
- Ausnahme bei Methoden. Hier wird die Signatur als Unterscheidungsmerkmal hinzugenommen
- Also Anzahl und Typ von Parametern

Beispiel: Überladen

```
static void print(int i) { ...}
static void print(float f) { ...}
static void print(int i, int width) { ...}
```

- Der Methodenname `print` ist *überladen*
- Bei einem Aufruf wird die „passende“ Methode automatisch ausgewählt:

```
print(100);      // print(int i)
print(3.14f);    // print(float f)
print(100, 5);   // print(int i, int width)
```

Überladen (II)

- Bei einem Aufruf `short s; print(s);` würde `print(int i)` aufgerufen werden, da diese „am nächsten“ liegt
- Wir haben schon häufig überladenen Methoden verwendet, nämlich `System.out.print()`

Beispiele für Methoden

Größter gemeinsamer Teiler

- Algorithmus in Kapitel 1, Sie erinnern sich ;-)

```
static int ggt(int x, int y) {
    int rest = x % y;
    while (rest != 0) {
        x = y;
        y = rest;
        rest = x % y;
    }
    return y;
}
```

GGT, um Bruch zu kürzen

```
static void reduce(int zaehler, int nenner) {
  int ggt = ggt(zaehler, nenner);
  System.out.println(
    zaehler + "/" + nenner + " kann man kuerzen zu
    + (zaehler / ggt) + "/" + (nenner / ggt));
}
```

- Bezeichner sinnvoll angepasst! (Sprechende Namen!)

Potenzieren

```
static long power(int n, int e) {
    long res = 1;
    for (int i = 1; i <= e; i++) {
        res = res * n;
    }
    return res;
}
```

Anwendungsgebiete von Methoden

Wiederverwendung von Code

- Mehrere (fast) identische Code-Blöcke zu *einer* Methode zusammenfassen und dann diese an den ursprünglichen Stellen aufrufen. Wird mittlerweile von modernen IDEs unterstützt (Stichwort Refactoring, Demo!)
- Dadurch weniger Code
- Der evtl. in Bibliotheken ausgelagert werden kann und dann zu robusteren Programmen führt, da Fehler über die Zeit entfernt wurden

Definition benutzerspezifischer Operationen und Datentypen

- Erweiterung einer Programmiersprache um neue Typen, die auch Operatoren auf diesen Typen enthalten (in Java Operatoren nicht möglich, daher Methoden). Beispiel: BigDecimal

Strukturierung von Programmen

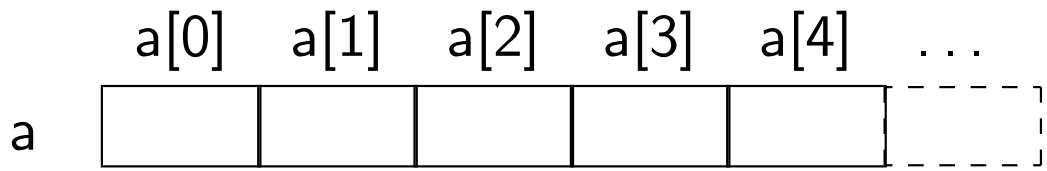
- Kleinere Code-Segmente, die logisch zusammen gehören und einen Namen haben
- Durch die geringere Größe/Komplexität einfacher zu verstehen
- Ermöglicht Abstraktion! Wir mussten die Methode `println()` des `System.out` nicht in ihrer Implementierung verstehen, um sie verwenden zu können
- Nur auf dieser Basis ist es heute überhaupt möglich, große Systeme zu bauen

Arrays

Eindimensionale Arrays

Motivation

- Bisher Einzelwerte mit eingenem Namen
- Häufig aber auch Bedarf an Mengen von Werten (Wertefolgen): Liste aller Kunden, alle Rechnungen des Monats Oktober, ...
- Ein Array ist eine Tabelle gleichartiger Elemente
- Die Tabelle hat einen Namen, die einzelnen Elemente nicht:



- Ein Element verhält sich wie eine namenlose Variable
- Achtung: Die Zählung beginnt bei 0

Deklaration von Arrays

- Arrays werden durch ein eckiges Klammernpaar symbolisiert:

```
int[] a;
float[] vector;
```

- Erste Zeile: Array mit Namen `a`. Elemente vom Typ `int`
- Zweite Zeile: Array mit Namen `vector`. Elemente vom Typ `float`
- Die Deklaration legt noch keinen Speicherbereich für das Array an. In den Beispielen wüsste man auch nicht, wie viel!
- „Gemischte“ Arrays sind nicht möglich
- Statt `int[] a;` kann auch `int a[];` geschrieben werden (ist aber nicht empfehlenswert)

Erzeugung von Arrays

- Bei der Erzeugung muss Größe angegeben werden:

```
a = new int[5];
```

- Man kann Deklaration und Erzeugung auch verbinden:

```
int[] a = new int[5];
```

- Die Variable a ist eine *Referenz* auf den kompletten Speicherbereich des Arrays
- Man Spricht daher auch von einem `Referenztyp` (Klassen in Kapitel 10 auch)
- Einzelne Elemente spricht man durch *Indizierung* an:

```
sum = a[2] + 27 + a[0];
```

Benutzung von Arrays

- Arrays können einander zugewiesen werden:

```
int[] a;
int[] b;
...
a = b;
a = new int[10000000];
```

- Ihre Länge erhält man durch `a.length`

Benutzung von Arrays II

- Als Index kann ein beliebiger ganzzahliger Ausdruck verwendet werden

```
a[3] = 0;
a[2 * i + 1] = a[j];
a[max(i, j)] = 100;
```

- Ein Zugriff auf ein nicht existierenden Index erzeugt einen Laufzeitfehler, die `ArrayIndexOutOfBoundsException`
- Bitte versuchen Sie, diese Exception in einem Beispiel zu provozieren!
- Java hilft uns hier, Fehler zu erkennen!

Arrays und Schleifen

- Das folgende Beispiel liest Werte ein und weist sie den Array-Elementen zu:

```
for (int i = 0; i < a.length; i++) {
  a[i] = scanner.readNext();
}
```

- Bitte beachten Sie das Muster für die Index-Obergrenze!

Arrays und Schleifen II

- Man kann die Array-Elemente einfach aufsummieren:

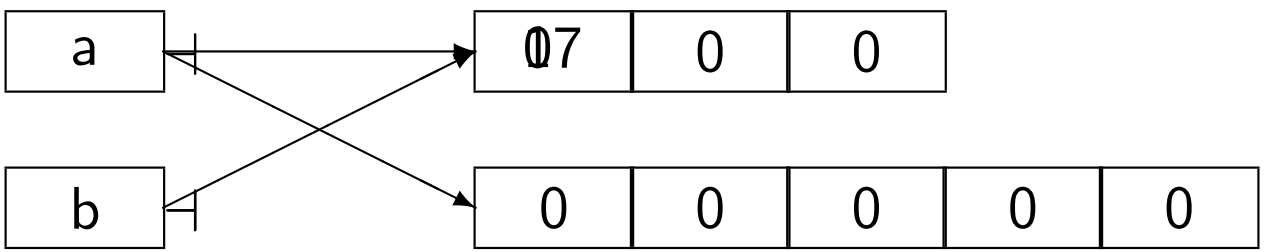
```
int sum = 0;
for (int i = 0; i < a.length; i++) {
   sum = sum + a[i];
}
```

Array-Zuweisung

- Zuweisung an Array-Variablen folgen den normalen Typregeln, d.h. Typ muss derselbe sein
- Der Wert der Array-Variablen ist die Referenz
- Es können mehrere Referenzen auf denselben Speicherbereich verweisen
- Dies ist für den Programmieranfänger häufig schwer zu verstehen, darum jetzt ein Beispiel

Array-Zuweisung II

```
int[] a, b;
a = new int[3];
b = a;
a[0] = 17; // b[0] == 17
a = new int[5];
b = null;
```



Freigabe von Arrayspeicher

- Sie müssen nur den Speicher anfordern, freigegeben wird er automatisch
- Wenn es keine Referenz auf den Speicher mehr gibt (wie im letzten Beispiel) *kann* das Java-Laufzeitsystem ihn freigeben
- Die automatische Speicherbereinigung heißt im Englischen *Garbage Collection*, das bereinigende System *Garbage Collector*
- Es gelten die Regeln zur Sichtbarkeit und Lebensdauer, allerdings nur für die Array-Variable und nicht den referenzierten Speicherbereich!

Initialisieren von Arrays

- Arrays können bei der Deklaration erzeugt und gleich initialisiert werden
- Da Java zählen kann, muss die Größe nicht explizit angegeben werden:

```
int[] primes = {2, 3, 5, 7, 11};
```

- Die Werte (vom korrekten Typ) werden durch Komma getrennt in geschweifte Klammern eingeschlossen
- Alternativ kann das auch bei einer späteren Erzeugung geschehen:

```
primes = new int[] {2, 3, 5, 7, 11};
```

Beispiel: Sequentielles Suchen

- Häufiges Problem ist die Frage, ob Element in einer Menge vorhanden ist oder nicht
- Hier ein Array von Integer-Zahlen und Rückgabe des Index, falls vorhanden, -1 falls nicht
- Herleitung: Am Schleifenende muss für Index-Variable `pos` und gesuchtes `x` gelten:

```
pos == -1 || a[pos] == x
```

- Durch Negation dieser Assertion erhält man die Schleifenbedingung:

```
pos != -1 && a[pos] != x
```

- Wenn man jetzt noch runter statt rauf zählt, bekommt man die -1 geschenkt

Sequentielles Suchen II

```
static int search(int[] a, int x) {
  int pos = a.length -1;
  while (pos >= 0 && a[pos] != x) {
    pos--;
  }
  // pos == -1 || a[pos] == x
  return pos;
}
```

- Assertionen sind bei der Programmentwicklung hilfreich
- Die Array-Variable (Referenz) wird zwar als Wertekopie übergeben, der Speicherbereich des Arrays aber nicht. Änderungen an Array-Elementen würden also den Methodenaufruf überleben

Beispiel: Binäres Suchen

- Letztes Beispiel muss im schlimmsten Fall (worst case) alle Elemente des Arrays anschauen
- Falls die Werte der einzelnen Elemente auf- oder absteigend sortiert sind, kann man effizienter Suchen, nämlich binär:
  - Wir halbieren das Array und wissen dann, dass in der linken Hälfte nur kleinere (bei aufsteigend sortiert), in der rechten Hälfte nur größere Werte stecken
  - Wenn das mittlere Array-Element nicht den gesuchten Wert enthält, wissen wir daher, in welcher Hälfte wir weiter suchen müssen
  - Wir müssen *logarithmischen* Aufwand betreiben, d.h. die Anzahl der Schleifendurchläufe benötigt im Worst-Case $log_2(n)$ bei n Array-Elementen
- Bemerkung: Man bekommt aber nichts geschenkt. Wir mussten vorher Sortieren, was auch Aufwand kostet

Beispiel: Binäres Suchen II

```
static int binarySearch(int[] a, int val) {
  int low = 0;
  int high = a.length - 1;
  while (low <= high) {
    int m = (low + high) / 2;
    if (a[m] == val) {
      return m;
    } else if (val > a[m]) {
      low = m + 1;
    } else {
      // val < a[m]
      high = m - 1;
    }
  }
  return -1;
}
```

- Bei 16 Elementen max. 4 Schleifendurchläufe, bei 1024 Elementen max. 10 Schleifendurchläufe (statt 1024)

Beispiel: Sieb des Erathostenes

- Algorithmus zur Berechnung von Primzahlen bis zu Obergrenze n
- Geht auf griechischen Mathematiker Erathostenes (284 - 202 v. Chr.) zurück
- Idee: Vielfache einer Primzahl können keine Primzahl sein
- Wenn wir die Menge der ersten n natürlichen Zahlen haben, müssen wir lediglich die Vielfachen der Primzahlen streichen
- Wenn dieses Streichen aufsteigend gemacht wird, ist die nächstgrößere übriggebliebene Zahl gerade die nächste Primzahl
- Beispiel:
  - 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...
  - 3, 5, 7, 9, 11, 13, 15, 17, 19, ... (2 gemerkt)
  - 5, 7, 11, 13, 17, 19, ... (3 gemerkt)
  - 7, 11, 13, 17, 19, ... (5 gemerkt)
  - ...

Allgemeines Problem der Software-Entwicklung

- Wie repräsentiere ich bestimmte Daten?
- Hier: Menge von natürlichen Zahlen
- Menge von Zahlen als boolesches Array. `true` für Zahl (die durch Index repräsentiert wird) ist enthalten, `false` für Zahl ist nicht enthalten

Beispiel: Sieb des Erathostenes (Code)

```
static void printPrimes(int max) {
  boolean[] sieve = new boolean[max+1]; // mit fals
  int i, j;
  for (i = 2; i <= max; i++) {
    sieve[i] = true; // Alle Zahlen > 1 enthalten
  }
  i = 2;
  while (i <= max) {
    System.out.println(i + ""); // i ist Primzahl
    for (j = i; j <= max; j = j + i) {
      sieve[j] = false;
    }
    while (i <= max && ! sieve[i]) {
      i++; // auf naechste Primzahl positionieren
    }
    // i > max || sieve[i] == true
  }
}
```

Beispiel: Monatstage berechnen

- Hatten wir schon in Kapitel 3 mit `switch(month)`
- geht viel einfacher:

```
int[] dayTab =
  {0,31,28,31,30,31,30,31,31,30,31,30,31};
int days = dayTab[month];
```

- Beachten Sie den „Nicht-Monat“ für Index 0!

Beispiel: Parameter-Array der Main-Methode

- Obwohl wir Strings erst in Kapitel 9 einführen, schauen wir uns noch einmal die Main-Methode an:

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

  for (int i=0; i < args.length; i++) {
    System.out.println("args[" + i + "]: "
                       + args[i]);
  }

}
```

- Demo mit Java und Eclipse

Mehrdimensionale Arrays

Java und Tabellen

- Ein Array von Arrays (von Arrays) . . . möglich
- Konzeptionell dann ein mehrdimensionales Array
- In Java jedes Array mit verschiedener Größe möglich, aber in der Regel nicht empfehlenswert

Deklaration und Benutzung

- Deklaration eines zweidimensionalen Arrays:

```
int[][] a;
```

- Erzeugen eines zweidimensionalen Arrays:

```
a = new int[4][3];
```

- Zugriff auf Element in Zeile i und Spalte j:

```
int x = a[i][j];
```

- Anzahl Zeilen, Anzahl Spalten:

```
a.length
a[0].length
```

Beispiel: Array initialisieren und ausgeben

```
int[][] a = {{1,2,3}, {4,5,6}, {7,8,9},
             {10,11,12}};

for (int i = 0; i < a.length; i++) {
    for (int j = 0; j < a[0].length; j++) {
        System.out.print(a[i][j] + " ");
    }
    System.out.println();
}
```

Beispiel Matrixmultiplikation

- Ziel ist eine Methode `matMult(a, b)`, die die Matrizen a und b multipliziert und als Ergebnis zurückliefert
- Das Element c[0,0] der Ergebnismatrix ist das Skalarprodukt der Zeile 0 von a mit der Spalte 0 von b, also:
  a[0][0] * b[0][0] + a[0][1] * b[1][0] + a[0][2] * b[2][0]

Code der Matrixmultiplikation

```
static float[][] matMult(float[][] a, float[][] b) {
    float[][] c =   new float[a.length][b[0].length];
    for (int i = 0; i < a.length; i++) {
        for (int j = 0; j < b[0].length; j++) {
            float sum = 0;
            for (int k = 0; k < b.length; k++) {
                sum = sum + a[i][k] * b[k][j];
            }
            c[i][j] = sum;
        }
    }
    return c;
}
```

Iterator-Form der For-Anweisung

Erweiterte For-Schleife

- Seit Java 5 gibt es eine erweiterte For-Schleife, die deutlich an Schreibarbeit spart
- Um das Array

```
int[] primes = {2, 3, 5, 7, 11, 13, 17, 19};
```

- auszugeben, musste man vor Java 5 schreiben:

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

- Jetzt genügt:

```
for (int p : primes) {
  System.out.println(p);
}
```

Methoden mit variabler Parameteranzahl

Motivation

- Anforderung: Summe von n Zahlen berechnen
- Einfache Realisierung: Methodenparameter vom Typ `int[]`
- Seit Java 5 gibt es die Alternative von Varargs-Parametern, bei der die unbekannte Anzahl von Parametern als Typ gefolgt von drei Punkten deklariert wird
- Bemerkung: intern wird der Parameter als Array gehandhabt und der Compiler erzeugt für den Aufruf ein solches Array

Varargs-Methoden

- Die Methode

```
static int sum(int... values) {
    int result = 0;
    for (int i : values) {
        result += i;
    }
    return result;
}
```

- Kann dann aufgerufen werden als:

```
System.out.println(sum(1, 5));
System.out.println(sum(3, 1, 17));
System.out.println(sum(13, 11, 1, 17, 99));
```

Regeln für Varargs

- Es darf nur ein Varargs-Parameter verwendet werden
- Dieser muss der letzte Parameter sein
- Da intern auf Array abgebildet, sind die folgenden beiden Signaturen identisch und damit nicht erlaubt

```
public static void foo(int... values) { ... }

public static void foo(int[] values) { ... }
```

Überladen mit Varargs

- Wenn die Signaturen hinreichend verschieden sind, können Varargs-Methoden nach den Regeln aus Kapitel 6 überladen werden
- In der Verwendung evtl. nicht eindeutig und damit in der Praxis die überladene Definition sinnlos:

```
// erlaubt 0, 1, 2, ... int-Parameter:
public static void foo(int... values) {...}
// erlaubt 1, 2, 3, ... int-Parameter:
public static void foo(int i, int... values) {...}
```

damit

```
foo();          // eindeutig, ok
foo(1);         // nicht eindeutig, also Fehler
foo(1, 2);      // nicht eindeutig, also Fehler
foo(1, 2, 3); // nicht eindeutig, also Fehler
```

Zeichen

Zeichenkonstanten und Zeichen-Codes

Zeichenkonstanten

- Einfache Zeichenkonstanten werden durch Zeichen, eingeschlossen in Apostrophe erzeugt:
  `'a', 'A', '?', ...`
- Da Zeichen intern ebenfalls binär dargestellt werden müssen, benötigt man eine *Codierung* der Zeichen

ASCII-Code

- Der American Standard Code of Information Interchange war früher die Standard-Codierung für Zeichen
- Er verwendet 7 Bits und enthält z.B. keine Umlaute
- Nicht alle Zeichen sind druckbar, d.h. haben eine sichtbare Darstellung
- Sie werden z.B. als Steuerzeichen verwendet: CR (carriage return), LF (line feed), FF (form feed), ...

ASCII-Code Darstellung

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

- Darstellung als hexadezimaler Wert: Linke Spalte ist 1. Potenz, obere Zeile 0. Potenz
- Beispiel: 'A' hat den Hexadezimalwert 0x41, dezimal 65

Unicode

- ASCII ist nicht mehr zeitgemäß, da nur 127 verschiedene Zeichen darstellbar
- Java verwendet Unicode
- Achtung! Fehler im Buch: Java verwendet *nicht* zwei Bytes pro Zeichen. Fehler habe ich bereits im Juli 2008 an Herrn Mössenböck gemeldet. Leider wurde 4. Auflage nicht korrigiert
- Java verwendet seit Version 5 Unicode 4.0, und damit 2 oder 4 Bytes zur Codierung eines Zeichens
- Der Datentyp `char` umfasst jedoch in der Tat 16 Bits und ist für europäische Zeichen völlig ausreichend
- Wenn Sie *sehr* großes Interesse daran haben, können sie dies nachlesen: Unicode 4.0 support in J2SE 1.5 und Supplementary Characters in the Java Platform
- Welche Codierung verwendet wird, ist im API-Doc der Klasse `Character` nachzulesen. Für Java 7 wird Unicode 6.0 verwendet

Unicode II

- Ersten 128 Zeichennummern identisch zu ASCII
- Höhere Nummern für Umlaute, griechische, arabische, chinesische, . . . Zeichen
- In der Regel nicht über Tastatur eingebbar, daher Schreibweise `\unnnn`, wobei `nnnn` vierstellige Hexadezimalzahl bedeutet
- Darstellung von Vier-Byte-Unicodes betrachten wir nicht
- Beispiele: `\u0041` für das Zeichen A, `\u03c0` für das Zeichen π
- Derartige Unicode-Werte als Zeichen, in Zeichenketten und in Namen verwendbar

Unicode III

- Einige Steuerzeichen können auch symbolisch, als sogenannte *Escape-Sequenzen* geschrieben werden:

`'\n'`	new line (LF)	`\u000a`
`'\r'`	return (CR)	`\u000d`
`'\t'`	Tabulator	`\u0009`
`'\\'`	Backslash	`\u005c`
`'\''`	Apostroph	`\u0027`

Umlaute

ä	\u00e4
Ä	\u00c4
ö	\u00f6
Ö	\u00d6
ü	\u00fc
Ü	\u00dc
ß	\u00df

Unicode in Namen und Schlüsselwörtern

- Kann man machen, muss man nicht ;-)

```
int sum = 1234;
System.out.println(\u0073\u0075\u006D);

public static void \u0073\u0075\u006D() {
   ...
}

public \u0073tatic void foo() {
   ...
}
```

Zeichenvariablen

Deklaration

- Deklaration mit Typ `char`:

```
char ch;
```

- Zuweisung:

```
ch = 'a';
ch = '\u0061';
```

- Der Datentyp `char` gehört zu den Ganzzahlentypen und ist eine Teilmenge von `int`
- `char` ist allerdings vorzeichenlos und damit Konvertierungen problematisch

Rechnen mit Zeichen

- Arithmetische Operationen und Vergleiche erlaubt
- Aber Vorsicht walten lassen! Arithmetik muss begründet sein!

```
char ch1, ch2 = 'a';
ch1 = ch2;
int i = ch1;
ch2 = (char) (i + 1);
System.out.println("ch2: " + ch2);

char ch = ...
if ('a' <= ch && ch <= 'z') {
    // ist Kleinbuchstabe
}
```

Ein-/Ausgabe von Zeichen

- Mit unserer Bibliothek möglich:

```
char ch = ...
if ('0' <= ch && ch <= '9') {
  System.out.println(ch + " ist eine Ziffer");
} else  if ('a' <= ch && ch <= 'z') {
  System.out.println(ch + " ist ein Kleinbuchstabe
} else if ('A' <= ch && ch <= 'Z') {
  System.out.println(ch + " ist ein Grossbuchstabe
} else {
  System.out.println(ch + " ist was anderes");
}
```

Zeichenarrays

- Arrays über alle Typen möglich, also auch über Zeichen
- Damit Texte möglich, aber besser mit Klasse `String`, im nächsten Kapitel
- Deklaration:

```
char[] s;
```

- Erzeugung:

```
s = new char[20];
```

- Deklarieren, Erzeugen, Initialisieren:

```
char[] s1 = new char[20]; // mit '\u0000' init
char[] s2 = {'a', 'b', 'c'};
```

Beispiel: Zeichenkettensuche

- Aufgabe: Suchen einer Zeichenkette in einer anderen Zeichenkette

```
static int stringPos(char[] t, char[] pat) {
  int i, j;
  int last = t.length - pat.length; // last possibl
  for (i = 0; i <= last; i++) {
    if (t[i] == pat[0]) {
      j = 1;
      while (j < pat.length && pat[j] == t[i+j]) {
        j++;
      }
      if (j == pat.length) {
        return i;
      }
    }
  }
  return -1;
}
```

Bibliotheksmethoden

Methoden der Klasse `Character`

- Nachdem Sie das API-Doc installiert haben, selektieren Sie im Frame links oben `java.lang`, danach im Frame links unten `Character`
- Studieren Sie nun die Beschreibung der Klasse im rechten Frame und insbesondere Methoden wie
  - `isLetter(char)`
  - `isDigit(char)`
  - `isLetterOrDigit(char)`
  - `toUpperCase(char)`
  - `toLowerCase(char)`

Strings

Motivation

- Zeichenketten (Strings) sehr häufig verwendet
- In jeder Programmiersprache Möglichkeit zur Darstellung von Strings
- In Java: Klasse `String`
- Compiler erzeugt direkt String-Operationen

String-Konstanten

Konstanten

- Zeichenfolgen in doppelten Hochkommas eingeschlossen
- Die in Kapitel 8 definierten Escape-Sequenzen sowie Unicode-Konstanten dürfen verwendet werden
- `"x"` und `'x'` sind etwas grundverschiedenes

Datentyp String

Deklaration und Zuweisung

- Deklaration wie gewohnt:

```
String a, b;
```

- Zuweisungen ebenfalls:

```
a = "Hello";
b = a;
```

- String-Variable beinhalten Objekte (Instanzen) der Klasse `String`
- Objektvariable sind Referenzen (wie bei Arrays)
- `a` und `b` verweisen nach den obigen Zuweisungen auf denselben Speicherbereich, der den Text `Hallo` enthält

Strings sind *immutable*

- Strings sind nach ihrer Erzeugung nicht mehr änderbar. Im Englischen (und auch Deutschen) spricht man von *immutable*
- Bekommt a einen neuen Wert zugewiesen

```
a = a + " World";
```

- so zeigt b immer noch auf `Hello`, a aber auf `Hello World`
- Der Plus-Operator ist für Strings als Verketten (Konkatenation) der Parameter definiert
- Typkonvertierungsregel: Wenn einer der Operanden von + ein String ist, wird der andere Operand automatisch in einen String konvertiert (das geht immer, wie wir in Kapitel 10/11 sehen werden)

String-Vergleiche

Identität und Gleichheit

- Der ==-Operator definiert die Identität eines Java-Objekts im Speicher (seine Adresse)
- Zwei Objektvariablen sind identisch, wenn sie auf dasselbe Objekt verweisen
- Bei String-Vergleichen daher nicht auf Identität prüfen!!!
- Beispiel:

```
String s = In.readString();
if (s == "Hallo") {
    System.out.println("s == 'Hallo'");
} else {
    System.out.println("s != 'Hallo'");
}
```

- liefert ungleich, auch wenn Sie Hallo eingeben

Es kommt sogar noch schlimmer

```
String a = "huhu";
String b = "huhu";

if (a == b) {
    System.out.println("'==' ergibt true");
} else {
    System.out.println("'==' ergibt false");
}
a = a + " haha";
b = b + " haha";
if (a == b) {
    System.out.println("'==' ergibt true");
} else {
    System.out.println("'==' ergibt false");
}
```

- Da der Compiler schlau ist (und es in der JLS so steht), optimiert er und legt nur ein `"huhu"` an. Dann allerdings zur Laufzeit zwei `"huhu haha"`
- Fazit: == nicht verwenden!

Die equals()-Methode

- Wie vergleicht man dann, ob zwei Strings denselben *Wert* haben, also gleich sind?

```
String s = In.readString();
if (s.equals("Hallo")) {
  Systme.out.println("s ist equals zu 'Hallo'");
} else {
  System.out.println("s ist nicht equals zu 'Hallo'");
}
```

- equals ist kommutativ. Sie können also auch schreiben:

```
String s = In.readString();
if ("Hallo".equals(s)) {
  System.out.println("s ist equals zu 'Hallo'");
} else {
  System.out.println("s ist nicht equals zu 'Hallo'");
}
```

String-Operationen

String-Methoden

- Die Klasse String besitzt ein paar nützliche Methoden, die Sie verwenden dürfen und sollen
- Haben Sie noch immer nicht das API-Doc installiert?
- Beispiele:

```
i = s.length(); // Array: length
ch = s.charAt(i);
i = s.indexOf("huhu");
i = s.indexOf("huhu", 5);
i = s.lastIndexOf("huhu");
s2 = s.substring(2);
s2 = s.substring(2, 6);
if (s.startsWith("huhu")) ...
if (s.endsWith("huhu")) ...
s2 = s.trim();
...
```

Aufbauen von Strings

Die Klassen `StringBuffer` und `StringBuilder`

- Zur Erinnerung: Strings sind in Java immutable
- Man benötigt etwas, um mit Strings direkt arbeiten zu können, d.h. um Strings verändern zu können
- Dies wird mit den Klassen `StringBuffer` und `StringBuilder` erreicht
- Faustregel für unsere Zwecke: Nehmen Sie `StringBuilder`

Schnittstelle (API) von StringBuilder

```
StringBuilder str = new StringBuilder();

i = b.length();
str.append(otherString);   // Ueberladen: String, ch
str.insert(pos, x);        // ebenfalls ueberladen
str.delete(from, to);
str.replace(from, to, "abc");
str.substring(from, to);   // wie bei String
ch = b.charAt(i);          // wie bei String
str.setChar(i, 'x');
String s = str.toString(); // liefert String
str.append(x).append(y).delete(5,10); // da Rueckga
```

String-Konversionen

Strings in Zahlen wandeln (und umgekehrt)

```
int i = Integer.parseInt("123");
float f = Float.parseFloat(var);
String s = String.valueOf(x);
```

- Letzte Methode überladen, siehe API-Doc!!!

Beispiele

Beispiel: Manipulation von Dateipfaden

```
static String className(String path) {
  StringBuffer b = new StringBuffer(path);
  if (path.endsWith(".java")) {
    int len = path.length();
    b.replace(len - 5, len, ".class");
  }
  //int i = path.lastIndexOf('\\');
  int i = path.lastIndexOf(
      System.getProperties()
            .getProperty("file.separator"));
  if (i >= 0) {
    b.delete(0, i + 1);
  }
  return b.toString();
}
// Aufruf:
  className("C:\\Programme\\MyProg.java");
  className("/home/bernd/MyProg.java");
```

Beispiel: Wörter aus einem Text herauslösen

```
static void printWords(String text) {
  int i = 0;
  int last = text.length() - 1;
  while (i <= last) {
    while (i <= last
           && !Character.isLetter(text.charAt(i))) {
      i++;
    }
    int beg = i;
    while (i <= last
           && Character.isLetter(text.charAt(i))) {
      i++;
    }
    if (i > beg) {
      System.out.println(text.substring(beg, i));
    }
  }
}
// Aufruf:
printWords("eins, zwei und drei");
```

Beispiel: Zahl in einen String konvertieren

```
static String valueOf(int n) {
    assert n >= 0;
    char[] a = new char[10];
    int i = 0;
    do {
        a[i++] = (char) (n % 10 + '0');
        n = n / 10;
    } while (n > 0);
    StringBuffer b = new StringBuffer();
    do {
        b.append(a[--i]);
    } while (i > 0);
    return b.toString();
}
// Aufruf:
    valueOf(123456);
    valueOf(987654321);
```

Klassen

Objektorientierung

- Java und andere moderne Sprachen sind objektorientiert
- Um das zu erreichen, werden Klassen definiert, die Daten und Methoden zu einer Einheit zusammenfassen
- Objekte sind dann *Instanzen* solcher Klassen

Deklaration und Verwendung

Einführendes Beispiel

- Ziel: Repräsentation eines Datums. Man benötigt Tag, Monat und Jahr

```
int day;
String month;
int year;
```

- Ist ungeschickt, da bei mehreren Datumsangaben mehrere solcher Variablen deklariert werden müssen, die nichts miteinander zu tun haben
- Besser: Deklaration einer Klasse

Deklaration von Klassen

- Klasse `Date` in Java:

```
class Date {
  int day;
  String month;
  int year;
}
```

- Variablen einer Klasse werden Felder, Attribute oder Instanzvariablen genannt
- Eine Klasse befindet sich in einer Datei mit demselben Namen
- Klassenname beginnt mit Großbuchstaben
- Klassen können Methoden enthalten (schon häufiger gemacht). Machen wir später, jetzt nur Daten

Verwendung von Klassen

- Deklaration von Variablen wie gehabt:
- Klasse `Date` definiert.

```
Date x, y;
```

- Variablen `x` und `y` sind vom Typ `Date` und bestehen jeweils aus den Variablen (Feldern) `day`, `month` und `year`
- Auf diese kann duch *qualifizierte* Benennung zugegriffen werden:

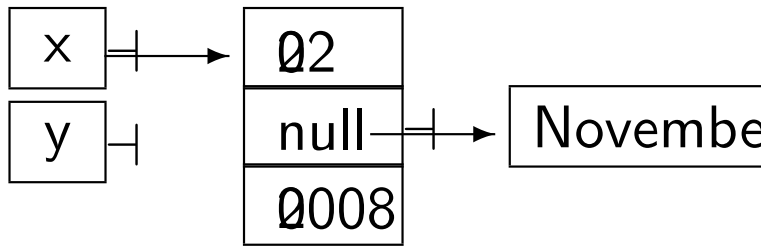
```
x.day = 22;
x.month = "November";
x.year = 2008;
```

- Variablen, deren Typ eine Klasse ist, enthalten Referenzen
- Die Werte solcher Variablen nennt man *Objekte*

Erzeugung von Objekten

- Objekte müssen mit `new` erzeugt werden (wie Arrays)

```
Date x, y;
```



- Objekte werden mit `null` initialisiert
- Jetzt Objekt erzeugen und Referenz in Variable schreiben

```
x = new Date();
```

- Integer werden mit 0 initialisiert, Strings sind Objekte
- Jetzt Werte zuweisen:

```
x.day = 22;
x.month = "November";
x.year = 2008;
```

Freigabe von Objekten

- Wenn ein Objekt nicht mehr referenziert wird, kann auf seine Werte nicht mehr zugegriffen werden, es ist „Garbage“
- Der *Garbage Collector* kann es dann aufsammeln und die JVM kann den Speicherbereich neu verwenden

Zuweisungen zwischen Objektvariablen

- Objektvariablen können einander zugewiesen werden
- Sie zeigen dann auf denselben Speicherbereich
- Das Schreiben des einen Objekts ändert auch das andere

Zuweisungskompatibilität

- Zuweisungen sind nur für dieselben Typen erlaubt (ab Kapitel 13 auch noch für sogenannte Unterklassen)
- Selbst wenn zwei Klassen identische Strukturen haben, sind sie nicht typkompatibel
- Der Wert `null` kann jeder Objektvariablen zugewiesen werden und steht für „*verweist auf nichts*“

Vergleiche

- Werden zwei Objektvariablen mit == oder != verglichen, so werden die Referenzwerte auf Gleichheit bzw. Ungleichheit verglichen
- Will man wissen, ob zwei Objekte desselben Typs dieselben Werte haben, so muss dies explizit programmiert werden:

```
static boolean isEqual(Date x, Date y) {
  return x.day == y.day
    && x.month.equals(y.month) // ist Objekt
    && x.year == y.year;
}
...
if (isEquals(x, y)) {
  ...
```

Klassen versus Arrays

- Arrays bestehen aus *gleichartigen* Elementen, Klassen aus *verschiedenartigen* Feldern
- Elemente eines Arrays haben keinen Namen und werden indiziert, Instanzvariablen haben einen Namen und werden über diesen angesprochen
- Anzahl der Elemente eines Arrays bei Erzeugung angegeben, Instanzvariablen einer Klasse bei Deklaration der Klasse

Beispiel: Datenstruktur für Linien

- Linien im zweidimensionalen Raum
- Linie besteht aus zwei Endpunkten
- Punkt hat Koordinaten x und y

Beispiel: Datenstruktur für Linien (II)

```
class Point {
  int x, y;
}
class Line {
  Point p1, p2;
}

Line line = new Line(); // Konstruktor fuer Linie
line.p1 = new Point();  // Konstruktor fuer Punkt
line.p1.x = 10;
line.p1.y = 20;
line.p2 = new Point();
line.p2.x = 30;
line.p2.y = 50;
```

Methoden mit mehreren Rückgabewerten

Typ von Rückgabewerten

- Bisher haben unserer Methoden meist `int`- oder `boolean`-Werte zurückgegeben
- Man kann als Rückgabetyp auch eine Klasse verwenden, d.h. die Methode muss ein Objekt dieser Klasse zurückgeben

Beispiel: Umrechnung von Sekunden auf Stunden, Minuten, Sekunden

- Rückgabetyp der Methode:

```
class Time {
  int h; // hours
  int m, // minutes
  int s; // seconds
}
```

- Methode `convert()` liefert Time-Objekt zurück:

```
static Time convert(int seconds) {
  Time t = new Time();
  t.h = seconds / 3600;
  t.m = (seconds % 3600) / 60;
  t.s = seconds % 60;
  return t;
}
```

Komplettes Programm

```
class Time {  // Datei Time.java
  int h, m, s;
}
class TimeMain { // Datei TimeMain.java
  private static Time convert(int seconds) {
    Time time = new Time();
    time.h = seconds / 3600;
    time.m = (seconds % 3600) / 60;
    time.s = seconds % 60;
    return time;
  }
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter seconds> ");
    int seconds = scanner.nextInt();
    while (true) {
      Time t = convert(seconds);
      System.out.println(t.h + ":" + t.m + ":" + t.s);
      System.out.print("Enter seconds> ");
      seconds = scanner.nextInt();
    }
  }
```

Kombination von Klassen und Arrays

Motivation

- Arrays von Klassen häufig anzutreffen (später evtl. besser/allgemeiner Collections von Klassen)
- Beispiel Telefonbuch: Für einen Namen eine Telefonnummer finden
- Geht nicht als zweidimensionales Array, da verschiedene Typen
- Zwei Alternativen:
  - Ein Array aus Objekten mit Name/Nummer
  - Ein Objekt mit zwei Arrays

Alternative I

- Klasse mit Name und Nummer

```
class Entry {
  String name;
  int phone;
}
```

- Telefonbuch ist Array von Entries:

```
Entry[] book = new Entry[100000];
```

- Verwendung:

```
book[i] = new Entry();
book[i].name = "Meier";
book[i].phone = 1234567;
```

Alternative II

- Klasse mit zwei Arrays

```
class PhoneBook {
  String[] name;
  int[] phone;
}
```

- Telefonbuch ist Instanz dieser Klasse

```
PhoneBook book = new PhoneBook();
book.name = new String[100000];
book.phone = new int[100000];
```

- Verwendung:

```
book.name[i] = "Meier";
book.phone[i] = 1234567;
```

- Was ist besser?

Beispiel: Eintragen und Suchen im Telefonbuch

- Die Klasse

```
class Entry {
    String name;
    int phone;
}
```

- Das Eintragen

```
static void enter(String name, int phone) {
  if (nEntries >= book.length) {
    System.out.println("Keine weiteren Eintraege m
  } else {
    Entry e = new Entry();
    e.name = name;
    e.phone = phone;
    book[nEntries++] = e;
  }
}
```

Suchen

```
static int lookup(String name) {
    int i = 0;
    while (i < nEntries && !name.equals(book[i].name))
      i++;
    }
    // i == nEntries || name.equals(book[i].name
    if (i == nEntries) {
        return -1;
    } else {
        return book[i].phone;
    }
}
```

Ein einfacher Test

```
public static void main(String[] args) {
  book = new Entry[100];
  enter("Mueller", 123456);
  enter("Meier",   234567);
  enter("Schulze", 345678);
  enter("Schmidt", 456789);

  System.out.println("Nummer von 'Mueller': "
      + lookup("Mueller"));
  System.out.println("Nummer von 'Schulze': "
      + lookup("Schulze"));
  System.out.println("Nummer von 'Haumichtot': "
      + lookup("Haumichtot"));
}
```

- Der Konstruktor von `Scanner` kann auch mit einer Datei (Instanz der Klasse `File`) aufgerufen werden anstatt mit `System.in`. Sie können dann mit `next()` und `nextInt()` aus der Datei lesen. Schreiben Sie ein Programm, das ein Telefonbuch aus einer Datei einliest.

Objektorientierung

Motivation

- Bisher Klassen zur Sammlung von Daten verwendet
- Konzeptioneller Zweck aber ein anderer: Daten *und* Methoden zu einer Einheit zusammenzufassen
- Damit „zusammen ist, was zusammen gehört“
- Beispiel Telefonbuch: Es ist sehr viel besser, die Einträge, das Einfügen und das Suchen in eine Klasse zusammenzufassen
- Wenn jemand diese Klasse verwendet, muss er nicht wissen, ob das Telefonbuch als Array von Einträgen oder mit zwei Arrays implementiert ist, da die beiden Methoden zum Einfügen und Suchen dies verstecken (kapseln)
- Ein Objekt ist eine Instanz einer Klasse und kann „eigenständig“ leben

Methoden in Klassen

Bruchzahlen als Objekte

- Wir benötigen ein Konstrukt zur Repräsentation von Brüchen
- Bekanntlich aus Zähler und Nenner bestehend
- Sinnvolle Operationen: Addition, Subtraktion, Multiplikation, Division
- Wir schreiben eine Klasse `Bruch` mit `zaehler` und `nenner` und den entsprechenden Methoden !

Die Klasse Bruch

```
public class Bruch {
  int zaehler;
  int nenner;

  void addiere(Bruch b) {
    zaehler = zaehler * b.nenner + b.zaehler * nenner;
    nenner = nenner * b.nenner;
  }
  void subtrahiere(Bruch b) {
    zaehler = zaehler * b.nenner - b.zaehler * nenner;
    nenner = nenner * b.nenner;
  }
  void multipliziere(Bruch b) {
    zaehler = zaehler * b.zaehler;
    nenner = nenner * b.nenner;
  }
  void dividiere(Bruch b) {
    int tmp = nenner * b.zaehler;
    zaehler = zaehler * b.nenner;
    nenner = tmp;
  }
}
```

Achtung!

- Die Variablen sind jetzt *ohne* `static` deklariert
- Die Methoden sind jetzt *ohne* `static` definiert
- Sie gehören nicht zur Klasse sondern zu *Instanzen* der Klasse, die man auch Objekte nennt
- Was bedeutet dies für die Methoden? Gibt es sie pro Klasse oder pro Instanz?

Beispiel

- Objekte erzeugen:

```
Bruch a, b;
a = new Bruch(); // geht auch bei Deklaration
b = new Bruch();
```

- Daten (Instanzvariablen) initialisieren:

```
a.zaehler = 1;
a.nenner = 2;
b.zaehler = 3;
b.nenner = 4;
```

- Methode aufrufen

```
a.addiere(b);
```

Zur Schreibweise

- Der Punkt ist ein Separator/Operator
- Links steht ein Objekt
- Rechts steht eine Variable oder eine Methode
- Man sagt auch, dass `a`, der Empfänger, die Nachricht `addiere(b)` geschickt bekommt
- `a` führt als Empfänger den Methodenaufruf durch (ausprobieren!)

UML-Klassendiagramme

- Die OMG (Object Management Group) ist eine Vereinigung praktisch aller namhafter großer Firmen, die OO betreiben
- Die UML (Unified Modeling Language) ist die Sprache (Notation), um objektorientierte Systeme zu beschreiben
- In einem UML-Klassendiagramm besteht eine Klasse aus
  - Einem dreigeteilten Kästchen
  - Im ersten Teil: Der Name
  - Im zweiten Teil: Die Variablen
  - Im dritten Teil: Die Methoden
- Moderne Systeme erzeugen daraus direkt Java (und alle andere Sprachen) und Datenbankschemata

Konstruktoren

Sinnvolle Initialisierung von Objekten

- Bei der Erzeugung eines Objekts werden alle Variablen initialisiert
- Zahlen mit 0, Referenzen mit `null`, Booleans mit `false`
- Falls dies nicht ausreicht (wie in unserem Beispiel), kann mit einem *Konstruktor* eine abkürzende Schreibweise eingeführt werden
- Ein Konstruktor hat denselben Namen wie die Klasse und kann Parameter haben (analog zu Methoden)

Beispiel Konstruktoren

```
public class Bruch {

  int zaehler;
  int nenner;

  public Bruch() {
  }
  public Bruch(int zaehler, int nenner) {
    this.zaehler = zaehler;
    this.nenner = nenner;
  }
```

- Für Konstruktoren mit Parametern hat sich das obige Namensschema etabliert. Bitte folgen Sie diesem
- Oder Sie lassen es Eclipse machen (rechte Maus - Source - Generate Constructor using Fields...

Konstruktoren

- Sinnvolle Default-Initialisierungen macht man im sogenannten Default-Konstruktor (der ohne Parameter)

```
public Bruch() {
  zahler = 0;
  nenner = 1;
}
```

- Konstruktoren dürfen wie Methoden überladen werden:

```
public Bruch(int zaehler, int nenner) {
  this.zaehler = zaehler;
  this.nenner = nenner;
}
```

- Der Default-Konstruktor wird automatisch erzeugt, falls Sie ihn nicht schreiben und die Klasse keinen anderen Konstrukor enthält (Beispiel)

Statische und objektbezogene Felder und Methoden

Klassen versus Objekte

- Manche Eigenschaften sind einer Klasse zugeordnet, manche einem Objekt
- Man spricht von
  - Klassenvariable vs Objektvariable (Instanzvariable)
  - Klassenmethode vs Objektmethode (Instanzmethode)
- Der *Modifier* `static` steht für Klassenvariable/-methode, sein Nichtvorhandensein (der Regelfall) für Objektvariable/-methode

Beispiel

- In Java gibt es keine Möglichkeit, das System zu fragen, wieviele Instanzen einer Klasse existieren. Wir müssen dies explizit programmieren
- Wie? Am besten mit einer Klassenvariable, die die bereits erzeugte Anzahl von Objekte enthält
- Das Zählen muss im Konstruktor erfolgen
- Falls man diesen Zähler zurücksetzen können muss, benötigen wir hierfür eine Klassenmethode

Code für Zählen von Instanzen

```
public class Bruch {

  static int anzahlInstanzen;

  int zaehler;
  int nenner;

  public Bruch() {
    anzahlInstanzen++;
  }
  public Bruch(int zaehler, int nenner) {
    this.zaehler = zaehler;
    this.nenner = nenner;
    anzahlInstanzen++;
  }
  static void zuruecksetzen() {
    anzahlInstanzen = 0;
  }
```

Static Initializer

- Das Initialisieren von Instanzvariablen kennen wir schon
- Gibt es so etwas auch für Klassenvariable?
- Ja, der sogenannte *Static Initializer* (JLS 8.4), der durch das Schlüsselwort `static` vor einem Block dargestellt wird:

```
public class Bruch {
  static int anzahlInstanzen;
  ...
  static {
    anzahlInstanzen = 0;
  }
```

- Es kann dort auch Code stehen (Öffnen einer Verbindung zur Datenbank, o.ä.)
- Da es keinen Namen und damit kein Überladen gibt, kann es nur einen geben (Highlander ;-))

Verwendung

- Objektmethoden können auf Klassenvariablen zugreifen
- Klassenmethoden aber nicht auf Objektvariablen (auf welches Objekt auch?)
- Objektmethoden können Klassenmethoden aufrufen aber nicht umgekehrt

Zugriff auf Klassenvariablen und Klassenmethoden

- Klassenvariable werden von außerhalb mit dem Klassennamen qualifiziert (genaue Sichtbarkeitsregeln folgen):

```
Bruch.anzahlInstanzen = 0;
```

- Klassenmethoden ebenfalls:

```
Bruch.zuruecksetzen();
```

- Klassenmethoden kennen kein `this`, da dies ja das Empfängerobjekt ist und hier keine Objekte im Spiel sind

Klassen als Hauptprogramme

- Wiederholung aus Kapitel 7
- Übung: Ersetzen Sie den Aufruf von `System.out.println` durch den Aufruf der selbstgeschriebenen Methode `print(String)`.

Zusammenfassung: Klasse vs Objekt

	Objekt	**Klasse**
Deklaration	ohne `static`	mit `static`
existierten	in jedem Objekt	nur einmal pro Klasse
Felder anlegen	Objekterzeugung	Ladezeitpunkt
Felder freigeben	GC	Entladezeitpunkt
Konstruktoraufruf	mit `new`	Ladezeitpunkt
referenzieren	obj.field this.field	Klasse.field
Methodenaufruf	obj.meth() this.meth()	Klasse.meth()

Beispiel: Klasse PhoneBook

Telefonbuch

- Hatten wir in Kapitel 10 schon realisiert, dort aber *nicht* objektorientiert
- Deshalb jetzt noch einmal und zwar objektorientiert

Beispiel Telefonbuch

```
class Entry {

    String name;
    int phone;

    public Entry(String name, int phone) {
        this.name = name;
        this.phone = phone;
    }

}
```

Beispiel Telefonbuch (II)

```
public class PhoneBook {
  Entry[] entries;
  int nEntries;
  public PhoneBook(int size) {
    entries = new Entry[size];
    nEntries = 0;
  } // Sie schreiben bitte Leerzeilen zw. Methoden!
  void enter(String name, int phone) {
    if (nEntries < entries.length && lookup(name) == -1)
      entries[nEntries++] = new Entry(name, phone);
    }
  }
  private int lookup(String name) {
    int i;
    for (i = 0; i < nEntries
          && !name.equals(entries[i].name); i++);
    if (i == nEntries) { return -1;
    } else {
      return entries[i].phone;
    }
  }
}
```

Beispiel Telefonbuch (III)

```
public static void main(String[] args) {
  PhoneBook book = new PhoneBook(10);
  book.enter("Mueller",  12345);
  book.enter("Meier",    23456);
  book.enter("Schulze", 34567);
  book.enter("Schmidt", 45678);

  System.out.println("Nummer von Hr/Fr Mueller: " +
  System.out.println("Nummer von Hr/Fr Schulze: " +
  System.out.println("Nummer von Hr/Fr Bohlen: " + 
}
```

- Kann in selben oder anderen Klasse stehen
- Im Buch mit lesen einer Datei
- Versuchen Sie dies auch

Beispiel: Klasse Stack

Machen Sie zuhause oder
nächstes Semester in *Algorithmen und Datenstrukturen*

Beispiel: Klasse Queue

Machen Sie zuhause oder
nächstes Semester in *Algorithmen und Datenstrukturen*

Dynamische Datenstrukturen

- Das Kapitel 12 *Dynamische Datenstrukturen* behandeln wir in dieser Veranstaltung nicht
- Die Inhalte des Kapitels 12 — und noch viel mehr — lernen Sie in der Veranstaltung *Algorithmen und Datenstrukturen* im nächsten Semester kennen

Vererbung

Grundsätzliches

- Klassen modellieren Dinge der realen Welt (Personen, Konten, Rechnungen, Verträge . . .)
- Diese Dinge öfter in verschiedenen Varianten, die man durch Klassifikation (über einen Diskriminator) hierarchisch gliedern kann
- Beispiel
  - Kunde mit Adresse
  - Firmenkunde mit Firmenname, Ansprechpartner, UStIdNr
  - Privatkunde mit Namen

Anforderungen an objektorientierte Programmiersprache

- Mit Varianten arbeiten können
- Mal Varianten unterscheiden, mal nicht
- Z.B. Wieviele Kunden habe ich (Firmen- und Privatkunden)
- Gib mir den Privatkunden Müller

Klassifikation

Beispiel

- Buchhandlung mit Büchern und CDs

Buch	**CD**
Artikelnummer	Artikelnummer
Verfasser	Interpret
Titel	Liste von Musikstücken
Verlag	Preis
Erscheinungsjahr	
Preis	

- Was fällt auf?

Abstraktion

- Web-Anwendung für Einkauf von Büchern und CDs
- Warenkorb: Pro Artikel Name und Preis
- Evtl. noch weitere Informationen, aber ohne Unterscheidung
- Also abstrakt Info über einen *Artikel*

Die Klasse Article

- Artikelklasse enthält alle Variablen und Methoden, die Bücher und CDs *gemeinsam* haben

```
class Article {
  int articleNumber;
  String description;
  BigDecimal price;
  void showInfo() {...}
  String getArticleLine() {...}
  Article(int articleNumber, String description,
          BigDecimal price) {...}
}
```

- `getArticleLine()`: Zeile aus `articleNumber`, `description` und `price`

Die Klasse Book

- Bücher sind nun *Spezialfälle* von Artikeln
- Sie besitzen dieselben Variablen und Methoden wie Artikel
- Und noch weitere !

```
class Book extends Article {
  String author, title, publisher;
  int year;
  void showinfo() {...}
}
```

- `extends` bedeutet *Erweiterung*
- Man spricht auch von *Oberklasse* und *Unterklasse* (engl. super class, sub class)

Die Klasse CD

- CDs ebenfalls *Spezialfälle* von Artikeln

```
class CD extends Article {
  String artist;
  String[] songs;
  void showinfo() {...}
}
```

- Achtung Code-Conventions: Pluralize Collections !

Klassenhierarchie

- Weitere Unterklassen möglich
  - Z.B. Hardcover, Softcover
- Unterklassen erben Variablen und Methoden
- Und können neue Variablen und Methoden hinzufügen
- Und können geerbte Methoden *überschreiben*, Beispiel `showInfo()`
- In Java sind alle Klassen, die nicht explizit erben, von der Klasse `Object` abgeleitet
- In der Klasse `Object` werden so wichtige Methoden wie etwa
  - `toString()`
  - `equals()`

  definiert. Studieren Sie das API-Doc!

Aufruf geerbter Methoden

- Wie kann eine geerbte, aber überschriebene Methode aufgerufen werden ?
- Die überschreibende Methode überdeckt ja die geerbte Methode (siehe Gültigkeitsbereich/Sichtbarkeit von Variablen im Kapitel 6)
- Neues Schlüsselwort: `super`

```
class Book extends Article {
  ...
  void showInfo() {
    super.showInfo();
    ...
  }
}
```

Aufruf geerbter Konstruktoren

- Geerbte Konstruktoren werden analog aufgerufen
- Da der Name klar ist, wird auf ihn verzichtet

```
class Book extends Article {
  Book( ... ) {
    super( ... );
    ...
  }
}
```

- Der Aufruf des Oberklassenkonstruktors muss die erste Anweisung des Unterklassenkonstruktors sein
- Fehlt der Aufruf wird automatisch der Default-Konstruktor aufgerufen
- Machen Sie sich die Aufrufreihenfolge klar !

Kompatibilität zwischen Ober- und Unterklasse

Warum das Ganze?

- Jedes Programm, das in der Lage ist, mit Objekten der Oberklasse zu arbeiten, kann auch mit Objekten der Unterklasse arbeiten
- Man nennt das das *Substitutionsprinzip*
- Im Beispiel:
  - Allgemein Algorithmus für Artikel formulieren
  - Funktioniert für Bücher und CDs
  - Wenn später Videos oder Computer-Spiele dazu kommen, muss nichts geändert werden
- Zwischen Unterklasse und Oberklase besteht eine *ist-Beziehung*

Zuweisungskompatibilität

```
Article a1 = new Article();
Article a2 = new Book();
Article a3 = new CD();
```

- Compiler weiß, dass Article-Objekte drin sind, also kein Zugriff auf Book- oder CD-Felder
- Man kann den Typ prüfen und „kleiner“ machen (Type-Cast, kurz Cast/Casting):

```
if (a instanceof Book) {
  Book b = (Book) a2;
  ... // jetzt Book-Methoden moeglich
}
```

Statischer vs dynamischer Typ

```
Article a = new Book();
```

- a hat den *statischen* Typ `Article` (Deklaration)
- Aber den *dynamischen* Typ `Book`
- Dies kann im Allgemeinen *nicht* berechnet werden:

```
Article a;
if ( Random/Benutzereingabe ) {
   a  = new Book();
} else {
   a  = new CD();
}
// hier zur Compile-Zeit nicht bekannt !
```

Finale Klassen

- Soll verhindert werden, dass eine Klassen Unterklassen haben kann, so ist in der Klassendeklaration das Schlüsselwort `final` zu verwenden
- Die Klasse ist dann die finale Klasse in der Hierarchie
- Prominente Vertreter im SDK sind die Klassen `System` und `String`

```
public final class KannNichtVererben {
   ...
}
```

Dynamische Bindung

Analyse eines Methodenaufrufs

- Was passiert bei `a.showInfo()` ?
- Variable a ist *polymorph*
- Zur Laufzeit muss der Typ von a bestimmt werden und dann die richtige Methode (Article, Book oder CD) aufgerufen werden
- Dies nennt man *dynamische Bindung*
- Wie gezeigt, kann dies *nicht* für den allgemeinen Fall zur Compile-Zeit bestimmt werden !
- Diese „Vielgestaltigkeit“ wird auch *Polymorphie* genannt (Substantivierung von polymorph)

Abstrakte Klassen

Wenn mal was nicht bekannt ist . . .

- Wenn bestimmte Informationen für eine Klasse nicht bekannt sind oder nicht bestimmt werden können, kann man die Klasse *abstrakt* machen
- Dies geschieht mit dem Schlüsselwort *abstract* auf Klassenebene
- Von einer solchen Klasse können keine Instanzen erzeugt werden
- Eine Methode wird mit diesem Schlüsselwort ebenfalls abstrakt (es fehlt dann der Rumpf)
- Besitzt eine Klasse eine abstrakte Methode, so ist sie selbst abstrakt (es würde bei einer Instanziierung ja etwas fehlen)
- Instanzen von Unterklassen können nur erzeugt werden, wenn nichts abstrakt ist, also alle Methoden ausformuliert sind
- Bitte ausprobieren ! Beispiele im SDK

Interfaces

Zentraler Ingenieur-Gedanke

- Man definiert Schnittstellen, damit es besser zusammen passt
- Beispiele: Metrisches Gewinde, Euro/Schuko-Stecker, Leuchtmittel, Bus-Systeme im PC, ...
- Machen Sie sich die Auswirkungen bei Nichtexistenz klar !

Java-Interfaces

- Reine Schnittstelle eines Typs
- Also quasi vollständig abstrakt
- Darf also keine Implementierungsdetails enthalten
- Übrig bleiben:
  - Abstrakte Methoden
  - Konstanten
- Keine
  - Konstruktoren
  - Variablen

Beispiel

```
public interface ArticleInterface {
   final int MEHRWERTSTEUERSATZ = 19;
   void showInfo();
}
```

- Alle Methoden sind öffentlich und abstrakt (`public abstract`)
- Alle Variablen sind Konstanten, öffentlich und abstrakt (`public static final`)

```
public class Book implements ArticleInterface {
   @Override // optionale Annotation
   public void showInfo() {
      ...
   }
}
```

- Eine Klasse *implementiert* ein Interface

Interfaces vs Vererbung

- Vererbung vererbt tatsächlich Code
- Interfaces vererben nichts, sondern beschreiben Schnittstelle
- Erben von mehreren Oberklassen verboten
- Implementieren mehrere Interfaces erlaubt
- Interfaces haben nichts mit Vererbung zu tun

Wozu Interfaces?

- Prominentes Beispiel: JDBC (Java Database Connectivity)
- Package `java.sql`: 7 Klassen, 22 Interfaces !!
- Warum?
- Schnittstelle zu relationalen Datenbanksystemem: Oracle, DB/2, SQL Server, Postgres, MaxDB, Informix, MySQL, Cloudscape (Derby, Java DB), HSQLDB, ...
- Sie schreiben Java-Code und es geht mit *allen* Datenbanksystemen ohne Code-Änderungen !
- Na gut, mit fast allen ;-)
- Noch extremer: JPA 2.0 (Package `javax.persistence`): 1 Klasse, 11 Interfaces, 76 Annotationen (besondere Interfaces, Kapitel 21)

Wrapper-Klassen und Boxing

Motivation

```
Article[] articles;
```

- In `articles` können Sie jetzt Bücher, CDs, ... reinstecken

```
Object[] objects;
```

- Hier können Sie alle Objekte reinstecken, aber nicht `int`, `double`, ...

Wrapper

- Für alle primitiven Datentypen gibt es Wrapper
- Im Package `java.lang`: `Boolean`, `Character`, `Double`, `Float`, `Byte`, `Short`, `Integer`, `Long`
- Wrapper sind wie Strings immutable
- Numerische Typen sind von der Oberklasse `Number` abgeleitet
- Enthalten noch sinnvolle Daten/Methoden: Min/Max, von/nach String, Unendlich, NaN, ... Hatten wir schon in Übungsaufgaben verwendet

Wrapper cont'd

- Konstuktoren für primitive Typen und Strings
- Zugriff auf Werte über gleichartige Methoden

```
Integer i1 = new Integer(27);
Integer i2 = new Integer("7391");

int i = i1.intValue();
Double dObject = new Double(1.14152);
double d = dObject.doubleValue();
```

Auto-Boxing / Unboxing

- Seit Java 5 macht der Compiler das Ein- und Auspacken automatisch
- Man kann jetzt also `int`, `double`, ... in ein `Object[]`-Array stecken
- Aber vorsicht: Sie dürfen nicht überall statt primitive Typen Wrapper schreiben:

```
Integer i1 = new Integer(1);
Integer i2 = new Integer(1);
...  i1 == i2 ... // false!
```

Weitere Themen der objektorientierten Programmierung

Was es sonst noch so gibt . . .

- Frameworks
- Patterns (Proxy, Factory, Singleton, . . .
- Komponenten
- Dependency Injection
- . . .

Enumerationstypen / Aufzählungstypen

Vor Java 5

- Häufig können Variable nur Werte einer kleinen Menge annehmen
  - Farben: rot, grün, blau
  - Himmelrichtungen: Nord, Süd, Ost, West
  - Familienstand: ledig, verheiratet, verwitwet, geschieden
  - Tic-Tac-Toe: Kreuz, Kreis, leer

```
static final int RED = 0;
static final int BLUE = 1;
static final int GREEN = 2;
```

- Nachteil: keine Typsicherheit

```
int color = RED;
color = 279431;
```

Seit Java 5: Aufzählungstypen

- Statt `class` wird `enum` verwendet

```
public enum Color {
  RED, BLUE, GREEN
}
```

- Compiler verhindert falsche Verwendung

```
Color color = Color.RED;   // ok
color = 279431;            // Fehler
```

Aufzählungstypen (cont'd)

- Enums in `switch` erlaubt:

```
Color color;
...
switch (color1) {
  case RED:
  ...
  break;
  default:
  ...
  break;
}
```

Enums sind Klassen

- Alle Enums erben von der Klasse `Enum`
- Und damit deren Methoden:
  - `compareTo()`
  - `ordinal()`, fängt bei 0 an
  - `toString()`
  - ...
- Methode `values()` kann im For-Iterator verwendet werden:

```
for (Color c: Color.values()) {
  System.out.println(c + " = " + c.ordinal());
}
```

Enums sind Klassen (cont'd)

```
public enum Roman {
  I(1), V(5), L(50), C(100), D(500), M(1000);
  private int value;
  Roman(int value) {
    this.value = value;
  }
  public int getValue() {
    return value;
  }
}

Roman roman = Roman.V;
System.out.println(roman.ordinal());
System.out.println(roman.getValue());
```

- Wird aber eher selten (bis nie) verwendet

Literatur I

- Hanspeter Mössenböck.
  Sprechen Sie Java?
  Dpunkt, 4 Auflage, 2011.
- Cay S. Horstmann and Gary Cornell.
  Core Java, Volume I — Fundamentals.
  Prentice Hall, 8. Auflage, 2008.
- Cay S. Horstmann and Gary Cornell.
  Core Java, Volume II — Advanced Features.
  Prentice Hall, 8. Auflage, 2008.

Literatur II

- Al Vermeulen, Scott W. Ambler, Greg Bumgardner, Eldon Metz, Trevor Misfeldt, Jim Shur, and Patrick Thompson. *The Elements of Java Style.* Cambridge University Press, 2000.