Abschnitt 1: Digitalisierung und Computersysteme

Grundsätzliche Regel:

- Informationen können genau dann in Computer eingespeist werden, wenn sie adäquat und vollständig als $0/1$–Daten (*Binärdaten*) darstellbar sind.
- Verarbeitungsschritte auf Binärdaten sind genau dann durch Computer ausführbar, wenn sie sich auf elementare Logikoperationen zurückführen lassen.

Abschnitt 1.1: Binärcode

Elementare Logikoperationen:

- Binärwert 1 wird als „wahr“, Binärwert 0 als „falsch“ interpretiert.
- *Logisches Und*: Für zwei Binärwerte a und b ist $a \wedge b = 1$ genau dann, wenn $a = 1$ und $b = 1$ ist.
- *Exklusiv–Oder*: $a \leftrightarrow b = 1$ genau dann, wenn $a = 1$ oder $b = 1$ gilt, aber nicht beides zugleich.
- *Inklusiv–Oder*: $a \vee b = 1$ genau dann, wenn $a = 1$ oder $b = 1$ oder beides zugleich gilt.
- *Negation*: $\neg a = 1$ genau dann, wenn $a = 0$.

Wahrheitstafel

für diese logischen Operationen:

a	b	$a \wedge b$	$a \leftrightarrow b$	$a \vee b$	$\neg a$	$\neg b$
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	1
1	1	1	0	1	0	0

Erinnerung aus der Schule:

- Unser Zahlensystem („Zehnersystem“) ist nicht *das* Zahlensystem schlechthin,
- sondern eben „nur“ das Zahlensystem zur Basis „zehn“.
- Jede andere Zahl $2, 3, 4, 5, \ldots$ ließe sich ebenso als Basis hernehmen.
- *Beispiel*: Die Zahl $28\,130$ im Zehnersystem lässt sich auch so schreiben:

  $1{\cdot}4^7 + 2{\cdot}4^6 + 3{\cdot}4^5 + 1{\cdot}4^4 + 3{\cdot}4^3 + 2{\cdot}4^2 + 0{\cdot}4^1 + 2{\cdot}4^0$

  $\longrightarrow$ Darstellung im Vierersystem: $12\,313\,202$

Arithmetische Operationen:

- *Erinnerung*: Bekanntlich arbeiten Computer auf dem *Binärsystem*, das heißt dem Zahlensystem mit Basis 2.
- *Wichtige Erkenntnis*: Arithmetische Operationen im Binärsystem lassen sich aus rein logischen Operationen synthetisieren.

  $\longrightarrow$ Sind daher durch Computer ausführbar.
- *Einfachstes, grundlegendes Beispiel*: Zwei einstellige Binärzahlen a und b sind zu addieren.

  $\longrightarrow$ Das Ergebnis ist ein- oder zweistellig.
- Wenn man zulässt, dass Zahlen mit Nullen beginnen, kann man das Ergebnis in jedem Fall zweistellig aufschreiben.

Konkret als Auswertungstabelle:

a	b	$a+b$
0	0	0 0
0	1	0 1
1	0	0 1
1	1	1 0

Vergleich mit den Wahrheitstafeln auf Folie 3:

- Die zweite Stelle von $a+b$ ist gerade $a \leftrightarrow b$.
- Die erste Stelle von $a+b$ ist gerade $a \wedge b$.

$\longrightarrow$ Arithmetik auf Logik zurückgeführt.

Etwas komplizierter: Addition mit zwei Stellen:

a	b	$a+b$
00	00	0 0 0
00	01	0 0 1
00	10	0 1 0
00	11	0 1 1
01	00	0 0 1
01	01	0 1 0
01	10	0 1 1
01	11	1 0 0
10	00	0 1 0
10	01	0 1 1
10	10	1 0 0
10	11	1 0 1
11	00	0 1 1
11	01	1 0 0
11	10	1 0 1
11	11	1 1 0

Addition mit zwei Stellen in Logik umgesetzt:

- *Notation*:
  - a_1: die erste Stelle von a (von links),
  - a_2: die zweite Stelle von a (von links),
  - b_1: die erste Stelle von b (von links),
  - b_2: die zweite Stelle von b (von links),
  - c_1: die erste Stelle von $a+b$ (von links),
  - c_2: die zweite Stelle von $a+b$ (von links),
  - c_3: die dritte Stelle von $a+b$ (von links).
- Dann gilt:
  - $c_3 = a_2 \leftrightarrow b_2$,
  - $c_2 = (a_1 \wedge b_1 \wedge a_2 \wedge b_2) \vee (a_1 \wedge \neg b_1 \wedge \neg(a_2 \wedge b_2)) \vee (\neg a_1 \wedge b_1 \wedge \neg(a_2 \wedge b_2)) \vee (\neg a_1 \wedge \neg b_1 \wedge (a_2 \wedge b_2))$,
  - $c_1 = (a_1 \wedge b_1) \vee (a_1 \wedge a_2 \wedge (b_1 \vee b_2)) \vee (b_1 \wedge b_2 \wedge (a_1 \vee a_2))$.

$\longrightarrow$ Prüfen Sie es nach!

Wie kommt man auf diese Formeln:

- $c_3 = a_2 \leftrightarrow b_2$: wie bei einstelliger Addition.
- $c_2 = (a_1 \wedge b_1 \wedge a_2 \wedge b_2) \vee (a_1 \wedge \neg b_1 \wedge \neg(a_2 \wedge b_2)) \vee (\neg a_1 \wedge b_1 \wedge \neg(a_2 \wedge b_2)) \vee (\neg a_1 \wedge \neg b_1 \wedge (a_2 \wedge b_2))$:
  - ◇ Bei der einstelligen Addition haben wir gesehen, dass $a_2 \wedge b_2$ gerade der Übertrag der dritten Stelle auf die zweite Stelle ist.
  - ◇ Es gilt $c_2 = 1$ genau dann, wenn $a_1 + b_1 + (a_2 \wedge b_2)$ ungerade ist.
  - ◇ Das ist einerseits der Fall, wenn alle drei Summanden 1 sind.

    $\longrightarrow a_1 \wedge b_1 \wedge (a_2 \wedge b_2)$ bzw. $a_1 \wedge b_1 \wedge a_2 \wedge b_2$.
  - ◇ Andererseits ist das der Fall, wenn genau einer der drei Summanden gleich 1 ist.

    $\longrightarrow (a_1 \wedge \neg b_1 \wedge \neg(a_2 \wedge b_2)) \vee (\neg a_1 \wedge b_1 \wedge \neg(a_2 \wedge b_2)) \vee (\neg a_1 \wedge \neg b_1 \wedge (a_2 \wedge b_2))$.

Fortsetzung: Wie kommt man auf diese Formeln

- $c_1 = (a_1 \wedge b_1) \vee (a_1 \wedge a_2 \wedge (b_1 \vee b_2)) \vee (b_1 \wedge b_2 \wedge (a_1 \vee a_2))$:
  - ◇ $(a_1 \wedge b_1)$ bedeutet $(a \geq 2 \wedge b \geq 2)$,
  - ◇ $(a_1 \wedge a_2 \wedge (b_1 \vee b_2))$ bedeutet $(a = 3 \wedge b > 0)$,
  - ◇ $(b_1 \wedge b_2 \wedge (a_1 \vee a_2))$ bedeutet $(a > 0 \wedge b = 3)$.

Multiplikation mit zwei Stellen:

a	b	$a * b$
00	00	0000
00	01	0000
00	10	0000
00	11	0000
01	00	0000
01	01	0001
01	10	0010
01	11	0011
10	00	0000
10	01	0010
10	10	0100
10	11	0110
11	00	0000
11	01	0011
11	10	0110
11	11	1001

Umsetzung in Logik:

- $c_4 = a_2 \wedge b_2$,
- $c_3 = \neg(a_1 \wedge a_2 \wedge b_1 \wedge b_2) \wedge ((a_1 \wedge b_2) \vee (a_2 \wedge b_1))$,
- $c_2 = a_1 \wedge b_1 \wedge (\neg a_2 \vee \neg b_2)$,
- $c_1 = a_1 \wedge b_1 \wedge a_2 \wedge b_2$.

$\longrightarrow$ Prüfen Sie es wieder nach!

Fundamentale Einsicht

Alle anderen arithmetischen Operationen auf Zahlen in Binärdarstellung lassen sich ebenfalls auf die elementaren logischen Operationen zurückführen.

$\longrightarrow$ Hier nicht weiter ausgeführt.

Abschnitt 1.2: Digitale Daten

- **Konsequenz** aus der letzten Folie: Alle Datenmanipulationen,
  - ◇ die sich als arithmetische Operationen auf natürlichen Zahlen formulieren lassen,
  - ◇ lassen sich auch allein mit Hilfe von logischen Operationen auf binären Wahrheitswerten formulieren
  - ◇ und lassen sich daher durch Computer erledigen (gemäß der grundsätzlichen Regel auf Folie 1).
- In den fünf Unterabschnitten von Abschnitt 1.2 werden wir vier verschiedene Arten von Daten beispielhaft betrachten.

Beispiel 1: Ganze Zahlen

- In der Regel werden alle ganzen Zahlen mit einer festen Stellenzahl m abgespeichert.

  $\longrightarrow$ Von links ggf. mit Nullen aufgefüllt wie auf Folie 6 ff.
- Einfache Möglichkeit zur Unterscheidung von positiven und negativen ganzen Zahlen: Das 1. Bit speichert das Vorzeichen:
  - ◦ 0 ≡ „+“
  - ◦ 1 ≡ „−“

  (Ist in realen Computern aus gewissen Gründen nicht so simpel realisiert.)
- *Konsequenz*: Die Zahlenmenge

  $$[-2^{m-1} \ldots +2^{m-1}]$$

  kann dargestellt werden.

Konkrete Umsetzung:

111...111	≡	$-2^{m-1}+1$	
111...110	≡	$-2^{m-1}+2$	
...		...	
100...010	≡	-2	
100...001	≡	-1	
100...000	≡	-0	⟵ !!!
000...000	≡	$+0$	⟵ !!!
000...001	≡	$+1$	
000...010	≡	$+2$	
...		...	
011...110	≡	$+2^{m-1}-2$	
011...111	≡	$+2^{m-1}-1$	

Beispiel 2: Zeichen

- Klein- und Großbuchstaben
- Ziffern
- Interpunktionszeichen
- Sonstige

Grundsätzlicher Ansatz:

Jedem Zeichen wird derart eine natürliche Zahl zugeordnet, dass je zwei Zeichen unterschiedliche Zahlen zugeordnet sind.

Konkretisierung:

- Auf den allermeisten Computern (auch bei uns) werden Zeichen nach dem *ASCII–Standard* kodiert.
- *ASCII*: American National Standard Code for Information Interchange (sprich „Aas-kih“).
- Jedem Zeichen wird eine Bitfolge aus sieben Bits zugeordnet.

  $\longrightarrow$ Zahlen $0 \ldots 2^7 - 1 = 127$, jeweils analog zu Folie 6 ff. und 15 ff. von vorne mit Nullen auf 7 Bits aufgefüllt.
- Beispiele:

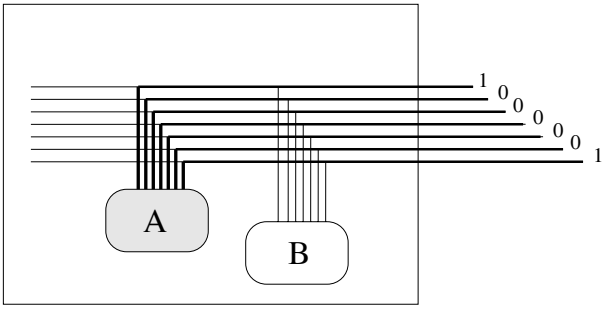
'a'...'z'	$\equiv$	97...122
'A'...'Z'	$\equiv$	65...90
'0'...'9'	$\equiv$	48...57
'?'	$\equiv$	63
'%'	$\equiv$	37
'/'	$\equiv$	47
'&'	$\equiv$	38

Einsicht:

Innerhalb eines Computers werden Zeichen praktisch ausschließlich als ihre ASCII–Bitmuster gespeichert und verarbeitet.

Frage: Wie kommt ein Zeichen bei der Eingabe zu seinem ASCII–Wert?

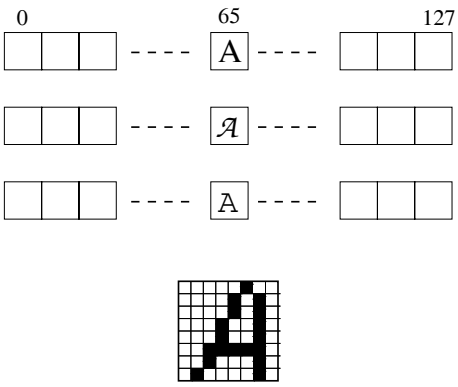
Antwort: Zum Beispiel bei Eingabe des Zeichens 'A' per Tastatur durch Stromfluss in Leitung Nr. 7 und 1 ('A' $\equiv 65 \equiv$ 1000001).



Tastatur

Rückwandlung eines ASCII–Wertes in eine bildliche Darstellung des Zeichens (vereinfacht):

- Für diverse Schriftarten (*Fonts*) sind Tabellen gespeichert, in denen jedem ASCII–Wert ein Bild zugeordnet ist.
- Üblicherweise (aber nicht ausschließlich) sind die Bilder als matrixartig angeordnete Sequenz von Bits abgelegt (1=gehört zum Zeichen, 0=Hintergrund).



Beachte:

Der ASCII–Wert einer Dezimalziffer ist nicht identisch mit ihrem dezimalen Zahlenwert.

- *Beispiel*: Die Zeichenfolge „1234“ wird gemäß Folie 18 abgespeichert als ASCII–Folge (49,50,51,52).
- Um die ASCII–Folge (49,50,51,52) in eine Dezimalzahl umzuwandeln, muss ein Umrechnungsalgorithmus auf die ASCII–Folge angewandt werden:
  - ◇ Ziehe von jeder ASCII–Nummer die ASCII–Nummer von 0 (also 48) ab.
    $\longrightarrow (1, 2, 3, 4)$
  - ◇ Multipliziere die vorletzte Ziffer mit 10^1, die drittletzte Ziffer mit 10^2 usw.
    $\longrightarrow (1000, 200, 30, 4)$
  - ◇ Addiere die vier Zahlen.
- Zur Ausgabe einer Dezimalzahl als Sequenz von ASCII–Zeichen im Dezimalsystem wird ein dazu inverser Umrechnungsalgorithmus angewandt.

Sonderzeichen:

- ASCII kodiert auch alle gängigen Interpunktionszeichen.
- Auch beispielsweise typisch amerikanische Zeichen wie ’@’ (Nr. 64).
- Das Leerzeichen (auch *Space* oder *Blank* genannt), hat ASCII–Wert 20.
- Es gibt auch ASCII–Werte, die
  - ◇ keinem Zeichen entsprechen,
  - ◇ sondern eine *Funktion* haben.

  $\longrightarrow$ Siehe Folie 32 für eines der wichtigsten Beispiele.

ISO–Latin–1:

- Die meisten Computer und Programme können Zeichenkodierungen mit acht Bits verarbeiten.
- Es gibt verschiedene standardisierte Auswahlen von Zeichen für die zusätzlich verfügbaren 128 Code–Nummern.
- *Standard* in Mitteleuropa: eine standardisierte Auswahl namens *ISO–Latin–1*.
- Beispiele für weitere Zeichen in ISO–Latin–1:
  - ◇ „Scharfes s“: ’ß’ $\equiv$ 223.
  - ◇ Deutsche Umlaute (z.B. ’Ä’ $\equiv$ 196).
  - ◇ Vokale mit Akzenten aus romanischen Sprachen (z.B. ’Ã’ $\equiv$ 195).
  - ◇ ’$’ $\equiv$ 163.

Eingabe von ISO–Latin–1–Zeichen auf ASCII–Basis:

- Für jedes dieser Zeichen gibt es eine Umschreibung in reinen ASCII–Zeichen der Form „`&...;`“.
- *Beispiele*:
  - ◇ ’ß’ $\longrightarrow$ „`&szlig;`“ (s–z–Ligatur)
  - ◇ ’Ä’ $\longrightarrow$ „`&Auml;`“ (A–Umlaut)
  - ◇ ’ä’ $\longrightarrow$ „`&auml;`“ (a–Umlaut)
  - ◇ ’Ö’ $\longrightarrow$ „`&Ouml;`“ (O–Umlaut)
  - ◇ ’ö’ $\longrightarrow$ „`&ouml;`“ (o–Umlaut)
  - ◇ ’&’ $\longrightarrow$ „`&`“ (engl. Ampersand)
- *Konsequenz*:
  - ◇ Wenn eine solche Sequenz (z.B. „`&szlig;`“) wörtlich dastehen soll: Schreib einfach hin „`&szlig;`“.
  - ◇ Dank „`&`“ kann also jeder beliebige Text in ISO–Latin–1 mit reinen ASCII–Zeichen umschrieben werden.

Beispiel:

- HTML ist die „Sprache“, in der normale WWW–Seiten geschrieben sind,
- HTML–Seiten mit beliebigen ISO–Latin–1–Texten können in ASCII erstellt werden.
- Insbesondere kann der Inhalt von HTML–Seiten mit normaler Tastatur eingegeben werden.

  $\longrightarrow$ Auch wenn die Tastatur keine Umlaute usw. hat.
- Die WWW–Browser (Netscape, Explorer...) interpretieren solche Umschreibungen, indem sie die entsprechenden Zeichen auf dem Bildschirm darstellen.

  $\longrightarrow$ Wie auf Folie 20.

Unicode:

- *Weiterentwicklung*: Unicode–Standard mit 16 Bits, also $2^{16} = 65\,536$ möglichen Zeichen.
- Tatsächlich festgelegt in Unicode ist nur die Bedeutung von knapp $40\,000$ 16–Bit–Werten.
- *Inhalte*:
  - ◇ Sonderzeichen aus diversen Sprachen (einschl. chinesisches Alphabet),
  - ◇ diverse technische Piktogramme,
  - ◇ diverse einfache geometrische Formen,
  - ◇ ...
- Unicode ist der Standardzeichensatz in Java.

Achtung:

- Bei Computerprogrammen, die nur „reines“ 7–Bit–ASCII verarbeiten können, kann es zu textuellen Entstellungen bei der Verwendung von Umlauten u.ä. kommen.
- Zum Beispiel beim Verschicken von Email können Email–Verwaltungsprogramme auf dazwischenliegenden Internet–Knoten (*Routern*) dieses Manko immer noch haben.
- *Typisches Ergebnis*: Wenn auch nur ein einzelner solcher Router „auf dem Weg“ liegt,
  - ◇ wird das führende Bit einfach auf 0 gesetzt und
  - ◇ der nächste Router, der mit einem 8–Bit–Zeichensatz arbeitet, behält diese Setzung für diese Bits bei.

  $\longrightarrow$ Woher soll der Router auch wissen, ob das ursprünglich eine 0 oder 1 war?
- *Beispiel*: ’Ä’ $\equiv$ 196 wird zu $196 - 128 = 68 \equiv$ ’D’.

Operationen auf Zeichen (vgl. Folie 18):

- Test, ob ein ASCII–Wert x für einen Kleinbuchstaben steht:

  x steht für einen Kleinbuchstaben

  $\Longleftrightarrow$

  $x \geq 97$ und $x \leq 122$
- Umwandlung eines Kleinbuchstabens in einen Großbuchstaben:

  Falls x der ASCII–Wert eines Kleinbuchstaben ist, dann ist x+’A’−’a’$= x + 65 - 97 = x - 32$ der ASCII–Wert des entsprechenden Großbuchstabens.

Konsequenz (vgl. Folie 14):

Solche Textmanipulationen lassen sich arithmetisch formulieren und daher mit Computern automatisch durchführen.

Einfacheres Rechnen mit ASCII:

- Die ASCII–Werte sind nicht willkürlich zugeordnet, sondern so, dass bestimmte Operationen möglichst effizient sind.
- Insbesondere gilt das für Groß- und Kleinbuchstaben und Ziffern.
- Bisher schon ausgenutzt (Folie 21 und 28):

  Die Zuordnung unmittelbar aufeinanderfolgender ASCII–Werte jeweils für 'a'...'z', 'A'...'Z' bzw. '0'...'9'.

Weiteres Beispiel:

Die ASCII–Wert eines Großbuchstabens und seines zugehörigen Kleinbuchstabens unterscheiden sich nur im Bit Nr. 6:

'A'	65	1000001
'B'	66	1000010
...	...	...
'Z'	90	1011011
...	...	...
'a'	97	1100001
'b'	98	1100010
...	...	...
'z'	122	1111011

$\longrightarrow$ Klein- und Großbuchstaben können einfach durch „blindes“ Überschreiben des Bits Nr. 6 ineinander umgewandelt werden.

Beispiel 3: Texte

- Texte sind im Prinzip Sequenzen von Zeichen, die aufeinanderfolgend im Speicher eines Rechners abgelegt werden.
- Im Speicher eines Rechners ist es aber notwendig, das Ende eines Textes irgendwie zu markieren.
- *Idee*:
  - ◇ Ein ASCII–Wert wird reserviert, der keinem Zeichen entspricht und der auch keine sonstige Funktion hat.
  - ◇ Dieser Wert wird hinter das Ende jedes Textes gesetzt, um anzuzeigen, dass der Text hier zu Ende ist.
- Reservierter Wert: 0.
- *Erinnerung* an Folie 21: Das ist **nicht** der ASCII–Wert des Zeichens '0'.

Newline:

- Der ASCII–Wert Nr. 10 ist für „Newline“ reserviert.
- Das ist ein ASCII–Wert, der
  - ◇ nicht einem Zeichen entspricht,
  - ◇ sondern wie auf Folie 22 beschrieben eine *Funktion* hat.
- *Konkrete Funktion*: Damit werden Zeilenumbrüche in Files angezeigt.
- Editoren und andere Programme zum Anzeigen von Files
  - ◇ geben solche Zeichen nicht auf Bildschirm oder Drucker aus (in welcher Form auch???),
  - ◇ sondern verarbeiten jedes solche Zeichen, indem sie mit der Anzeige des restlichen Textes auf der nächsten Zeile fortfahren.
- *Achtung*: Microsoft–Produkte haben eine eigene, aber ähnliche Konvention für Zeilenumbruch!

Texte durchsuchen mit regulären Ausdrücken:

- Engl. Name für reguläre Ausdrücke: *Regular Expressions*.
- Mit dem UNIX–Kommando `grep` kann man in Dateien (*Files*) nach Zeichenketten (*Strings*) suchen. Das Kommando „grep“ steht für *global regular expression print*,

  Beispiel: Alle Zeilen mit Zeichenkette „class“ in einem File namens „Trial.java“ erhält man mit dem Kommando

  ```
  grep ¨class¨ Trial.java
  ```
- Man kann auch „unscharfe“ Anfragen stellen, zum Beispiel
  - ◇ `grep ¨cl.ss¨ Trial.java`
  - ◇ `grep ¨clas*¨ Trial.java`
  - ◇ `grep ¨c[j-m]ass¨ Trial.java`
  - ◇ `grep ¨c[:lower]ass¨ Trial.java`

Erläuterungen zum Beispiel:

- „`cl.ss`“:

  Alle fünfbuchstabigen Worte mit „`cl`“ vorne und „`ss`“ hinten.

  (Auch mit einem Großbuchstaben, einer Ziffer oder einem Sonderzeichen als drittem Zeichen.)
- „`clas*`“:

  „`clas`“ und „`class`“ und „`classs`“ und „`classss`“ und „`classsss`“ und ...
- „`c[j-m]ass`“:

  „`cjass`“ und „`ckass`“ und „`class`“ und „`cmass`“.
- „`c[:lower]ass`“:

  „`caass`“ und „`cbass`“ und „`ccass`“ und ... und „`cxass`“ und „`cyass`“ und `czass`“.

Allgemeine Regeln:

- Eine Zeichenfolge ohne die Zeichen „`.`“, „`*`“ und „`[`“ (und noch ein paar weitere, hier nicht erwähnte) steht einfach für sich selbst.
- Ein „`.`“ in einem Wort steht für jedes beliebige Zeichen.
- Ein „`*`“ in einem Wort, aber nicht am Anfang des Wortes, steht für beliebig viele Vorkommen des unmittelbar vorhergehenden Zeichens.

  (Am Wortanfang steht „`*`“ für sich selbst.)
- Die Zeichen innerhalb eines Paares von eckigen Klammern „`[...]`“ haben besondere Bedeutung (hier nicht näher ausgeführt, siehe die Beispiele auf der letzten Folie).
- Usw.

Wie logisch sind Regular Expressions:

- Jede Regel sollte natürlich in jeder Situation sinnvoll sein.

  Bsp.: Dadurch, dass „`*`“ am Wortanfang nur für sich selbst steht, ist der Bezug zum vorhergehenden Zeichen in der Hauptfunktion von „`*`“ in sich widerspruchsfrei.
- Jeder Ausdruck in Regular Expressions sollte eine einzige, eindeutige Bedeutung haben.

  Bsp.: Die beiden verschiedenen Bedeutungen von „`*`“ finden in Situationen Anwendung, die sich logisch ausschließen („am Wortanfang“ und „nach einem Zeichen“).
- Ein- und dieselbe Bedeutung sollte nicht unbedingt durch zwei oder mehr Ausdrücke ausdrückbar (*redundant*) sein.

  Bsp.: „`[:lower]`“ und „`[a-z]`“ bedeuten dasselbe, das heißt, das ganze Konstrukt „`[:lower]`“ ist redundant.

Ausdrucksmächtigkeit:

- Jede mögliche Bedeutung sollte möglichst ausdrückbar sein.
- *Beispiel*: Mit den bisher betrachteten Sprachmitteln lässt sich keine scharfe Anfrage stellen, deren Ergebnis genau alle Vorkommen des einzelnen Wortes „`cl*ss`“ umfasst (und nichts sonst!).
- Das Konzept der Regular Expressions umfasst weitere Sprachmittel, mit denen zum Beispiel gezielt nach „`cl*ss`“ gesucht werden kann.
- *Konkret*: Mit Text „`cl\*ss`“ wird gezielt nach „`cl*ss`“ gesucht.

Allgemeine Regel für „\“

- „\“ vor einem Sonderzeichen (wie bspw. „*“) nimmt dem Sonderzeichen wie oben die Sonderbedeutung.
- „\“ vor einem anderen Zeichen wird verschluckt:

  ⟶ „`cl\ass`“ steht für das einzelne Wort „`class`“.
- Dem Zeichen „\“ selbst nimmt man genauso mit einem vorangestellten „\“ die Sonderbedeutung.

  ⟶ „`cl\\ss`“ steht für das einzelne Wort „`cl\ss`“.
- *Insgesamt*: Mit ein paar weiteren solchen Sonderregeln und Sonderzeichen (hier nicht näher ausgeführt) wird insgesamt eine sehr große Ausdrucksmächtigkeit erreicht.

Beispiel: HTML (HyperText Markup Language)

- In dieser Sprache werden Seiten im WWW beschrieben
- Idee: Beschreibung der Struktur von Texten.
- Herausforderung: Darstellung vieler Dinge, die sich in ASCII nicht direkt darstellen lassen
- Lösung: HTML verwendet so genannte Tags, um die Struktur des Textes zu markieren. Ein Tag hat die Form:
  `<tag>Vom Tag betroffener Text</tag>`.
- Tags können ineinander geschachtelt werden, d.h. innerhalb des Anwendungsbereichs eines Tags können sich weitere Tags befinden.

Struktur eines HTML-Dokumentes

- HTML kennt ein oberstes Tag: `<html>`. Dadurch wird ein HTML-Dokument gekennzeichnet.
- Direkt innerhalb des `<html>`-Tags befinden sich i.d.R. zwei weitere Tags:

  `<head>` In diesem Tag befinden sich Informationen über das Dokument.
  `<body>` Dieses Tag umschliesst den eigentlichen Inhalt des Dokumentes.

In `<body>` sind z.B. die folgenden Tags bekannt:

`<h1>` Dies bezeichnet eine Überschrift der ersten Ordnung. Entsprechend gibt es `<h2>`, `<h3>`, etc.

`<p>` Der mit diesem tag umschlossene Text stellt einen Absatz dar.

⟶ Mehr dazu in der Übung.

Abschließendes Beispiel zu HTML:

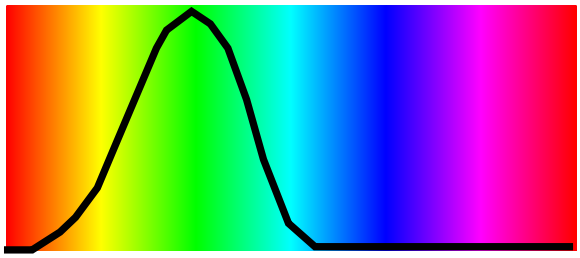
Beispiel 4: Farben

- *Erinnerung* aus dem Biologieunterricht:
  Das menschliche Auge hat
  - ◇ „Stäbchen“ zur Unterscheidung von Helligkeiten sowie
  - ◇ drei verschiedene Arten von „Zäpfchen“, die für rot, grün und blau empfindlich sind.
- *Genauer*:
  - ◇ Jede Art von Zäpfchen ist für einen großen Bereich des sichtbaren Frequenzspektrums empfindlich,
  - ◇ aber mit einem ausgeprägten Empfindlichkeitsmaximum im roten, grünen bzw. blauen Bereich.
  - ◇ Das Signal eines Zäpfchens an das Gehirn besteht aus einem einzelnen *Intensitätswert*, der sich aus der Stärke und Frequenzverteilung des Lichtstrahls ergibt.

Veranschaulichung:

- Am Beispiel der grünen Zäpfchen.
- Die Abbildung unten ist rein schematisch und ohne Anspruch auf Exaktheit im Detail!
- Schwarze Kurve: Sensibilität der grünen Zäpfchen für die einzelnen Frequenzen.

  ⟶ Der Intensitätswert, den jede Frequenz bei gleicher Lichtenergie im grünen Zäpfchen erzeugt.



(siehe auch Farbtafel hinten im Skript!)

Konsequenz:

- Die Farbtöne zweier Lichtstrahlen sind für das subjektive Farbempfinden ununterscheidbar, wenn die drei Intensitätswerte der Zäpfchen identisch sind.
- Eine (subjektiv) vollständige Farbpalette muss also nicht für jede mögliche Frequenzverteilung eine Farbe enthalten,
- sondern nur für jedes mögliche Tripel aus rotem, grünen und blauen Intensitätswert eine Farbe, die diese Intensitätswerte erzeugt.

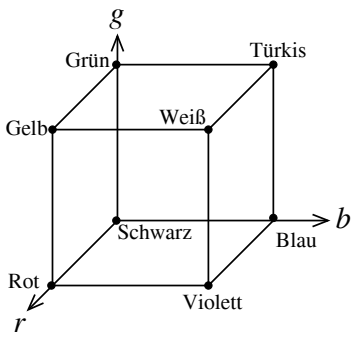
Fundamentales Schema: RGB

- Besteht aus drei Farbmischungen, die sich ähnlich wie auf der letzten Folie aus den einzelnen Frequenzen zusammensetzen,
- ebenfalls mit Spitzen im roten, grünen und blauen Bereich.

⟶ Jede subjektiv mögliche Farbempfindung kann daraus annähernd „gemixt“ werden.

RGB konkret:

- Eine Farbe wird definiert durch ein Zahlentripel (r, g, b).
- Jede der Zahlen r, g und b hat einen Wert im Bereich $[0 \ldots 1]$ (in endlich vielen Abstufungen).
- *Interpretation*:
  - ◇ Wert 0: Farbe leistet keinen Beitrag.
  - ◇ Wert 1: Farbe leistet maximalen Beitrag.
- *Veranschaulichung*: der sogenannte *RGB–Würfel*.



Rechnen mit Farben:

- *Einfaches Beispiel*:
  - ◇ Der Rotanteil r soll um 10% gesteigert werden.
  - ◇ Die Gesamthelligkeit $r + g + b$ soll aber gleich bleiben.

  $\longrightarrow$ Rechenregeln dieser Art stehen hinter den Möglichkeiten zur Farbmanipulation in Graphikprogrammen.
- *Ungefähre Realisierung*:
  - ◇ Der Wert r wird um 10% (aber maximal bis 1) erhöht.
  - ◇ Die Werte r und b werden anteilig soweit (aber höchstens bis 0) vermindert, dass $r + g + b$ wieder den ursprünglichen Wert hat.
- *Also*:

  Farbmanipulationen solcher Art lassen sich arithmetisch formulieren und daher von Computern durchführen (gemäß Folie 14).

Problem:

- Für die Mischung einer Farbe per Augenschein ist RGB nicht besonders intuitiv.
- Man versuche etwa, ein möglichst reines, „leuchtendes“ Braun aus rot, grün und blau zu mischen!

Idee:

- Lege der Farbpalette drei Freiheitsgrade zugrunde, die intuitiver verwendbar sind als rot, grün und blau.
- *Heuristischer Ansatz*: Wie würde man denn Farben typischerweise mit Worten beschreiben?
- *Oft verwendete Kandidaten*:
  - ◇ Farbton (*Hue*): rot, gelb, grün, blau...
  - ◇ Sättigung (*Saturation*) : von pastell bis grell leuchtend
  - ◇ Helligkeit (*Brightness*)

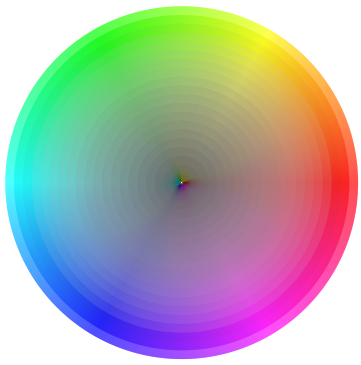
  $\longrightarrow$ *HSB–Schema*

Zusatzbeobachtung:

Die beiden Enden des Regenbogens erzeugen subjektiv praktisch identische Farbempfindungen (violett).

Umsetzung:

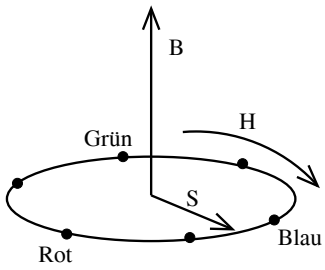
- Der Farbton als Parameter bildet einen Kreis.
- Sättigung und Helligkeit variieren die Farbmischung unabhängig davon (im Bild nur Sättigung).



(siehe auch Farbtafel hinten im Skript!)

Konkrete Umsetzung:

- Farbton, Sättigung und Helligkeit sind drei unabhängig variierende *Zylinderkoordinaten*.
- Farbton und Sättigung formen *Polarkoordinaten* in jedem Querschnitt des Zylinders:
  - ◇ Farbton = Winkel $\in [0^o \ldots 360^o]$,
  - ◇ Sättigung = Radius $\in [0 \ldots 1]$.
- Die Helligkeit $\in [0 \ldots 1]$ ist die Koordinate auf der darauf senkrechten Zylinderachse.



Diskussion von HSB:

- Das HSB–Schema ist damit ebenfalls auf einfache Zahlentripel zurückgeführt.

  $\longrightarrow$ Gemäß Folie 14 lassen sich Farben in HSB–Format also in Computern abspeichern und verarbeiten.
- Intuitive Farbmanipulationen wie die Erhöhung/Verminderung von Sättigung und Helligkeit sind mit HSB trivialerweise arithmetisierbar und daher durch Computer ausführbar.
- *Beispiel* (Folie 47): Leuchtendes Braun ist
  - ◇ Farbton gelb,
  - ◇ maximale Sättigung,
  - ◇ mittlere Helligkeit.
- RGB ist als Grundlage für Bildschirme u.ä. wesentlich besser geeignet als HSB.
- HSB kann aber durch ein einfaches mathematisches Schema in RGB (und umgekehrt) umgerechnet werden.

Exkurs: additive/subtraktive Farbmischung

- Wenn das Licht von mehreren Lichtquellen verschiedener Farbe zusammenkommt, ergibt sich *additive Farbmischung*, zum Beispiel:
  - ◇ Der Widerschein mehrerer verschiedenfarbiger Lampen auf einer weißen Leinwand.
  - ◇ Die Farbwahrnehmung bei einem Farbfernseher, zusammengesetzt aus den rot, grün und blau leuchtenden Pixeln.
- Wenn mehrere Farben zusammengemischt werden, spricht man von *subtraktiver Farbmischung*, zum Beispiel,
  - ◇ Farben mischen beim Malen.
  - ◇ Beim Farbdrucker wird die Farbe durch Aufspritzen von (meist vier) Grundfarben erzeugt. Das dabei verwendete Farbmodell nennt man CMYK. Dies ist wie RGB eine Abkürzung für die dabei benutzten Grundfarben: **C**yan, **M**agenta, **Y**ellow und Blac**k**.

Physikalischer Unterschied:

- Bei der Überlagerung von Lichtquellen werden deren Farbanteile addiert.

  $\longrightarrow$ rot + grün + blau = weiß
- Ein Objekt hingegen, das nur das Licht einer Lichtquelle reflektiert, bekommt dadurch seine Farbe, dass seine Oberfläche die entgegengesetzten Farbanteile (*Komplementärfarbe*) verschluckt.
- Beim Farben mischen u.ä. werden alle Farbanteile verschluckt, die in irgendeiner der Farben verschluckt werden.

  $\longrightarrow$ Nicht die subjektiv wahrnehmbaren Farbanteile, sondern die verschluckten (also subtrahierten) Farbanteile werden addiert.

  $\longrightarrow$ rot + grün + blau = schwarz

Zusammenhang mit RGB:

- Da auch das Auge die Farbwahrnehmung additiv aus Rot, Grün und Blau mischt, eignet sich RGB gut für additive Farbmischung.

  $\longrightarrow$ Standard bei Bildschirmen.

- Bei subtraktiver Farbmischung kann man aber zum Beispiel Weiß oder Gelb nicht aus Rot, Grün und Blau mischen.

  $\longrightarrow$ Eine Farbenmischung kann ja nur dunkler sein als die hineingemischten Farben.

- Für subtraktive Farbmischung eignen sich aber andere Kombinationen aus Grundfarben.

- *Zum Beispiel*: Die Kombination Cyanblau, Magentarot, Gelb und Schwarz.

  $\longrightarrow$ Oft benutzt bei Farbdruckern.

... Exkurs Ende

Beispiel 5: Bilder

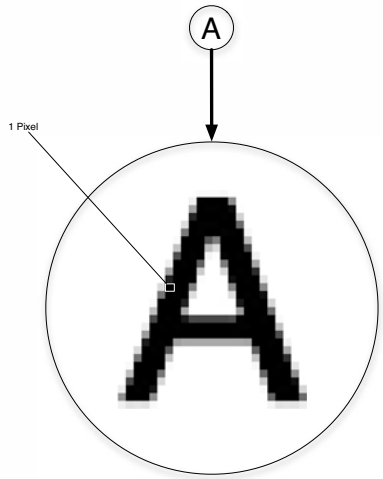
- Bilder sind im Prinzip Matrizen/Tabellen aus Farbwerten:
  - ◇ Im Standardfall RGB-Tripeln.
  - ◇ Für Graustufen-Bilder einzelne Helligkeitswerte.

- Jeder Farbwert bezeichnet einen Bildpunkt, ein sogenanntes Pixel (= **Pic**ture **El**ement).

- Dieser Bildpunkt bekommt durch den Farbwert seine Farbe. Seine Position in der Matrix bestimmt seinen Ort im Bild.

- Die Größe der Matrix ist die Größe des Bildes.

Graustufen-Beispiel:



Speicherung solcher Bilder:

- Tabellen bzw. Matrizen ,,passen nicht" direkt in die verschiedenen Speicher eines Rechners.

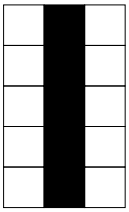
- Diese sind eher mit einer langen Folge von Werten zu vergleichen.

- Ansatz:
  - ◇ Die Farbwerte der Pixel werden einfach hintereinander gesetzt.
  - ◇ Am Anfang der Datei (,,im Header") wird vermerkt, wie lang eine Zeile des Bildes ist.
  - ◇ Daraus kann das eigentliche Bild wiederhergestellt werden.

$\longrightarrow$ Siehe nächste Folie.

Einfaches S/W-Beispiel:

Aus diesem veranschaulichten Speicherinhalt (jeder Kasten entspricht einem Pixel) kann man mit Hilfe der Angabe ,,eine Zeile besteht aus 3 Pixeln" das untere Bild bestimmen:



Praxis

In den Headern tatsächlicher Bilddateien stehen noch viel mehr Informationen als nur die Anzahl der Pixel pro Zeile:

- Der Name des verwendeten Dateiformates (Beispiel: JPEG, GIF).
- Kommentare zum Inhalt des Bildes.
- Datum der Aufnahme und andere Hinweise zur Entstehung (vor allem bei Digitalfotos).
- ...

Unterschied Pixel- und Vektorgrafik

- Alle bisher zu Bildern gemachten Aussagen beziehen sich auf sog. Pixel-Grafiken.
- Diese eignen sich sehr gut für Fotos, jedoch kaum für Zeichnungen, da sich Pixel-Grafiken z.B. nur sehr schlecht vergrößern lassen.
- Es gibt aber noch einen weiteren Typ von Grafiken am Rechner: Vektorgrafiken.
- Diesen werden Sie in der Übung kennenlernen, da Vektorgrafik im Prinzip Grafikprogrammierung ist und in den Übungen die Programmierung grafischer Anwendungen als Beispiel verwendet wird.

Abschnitt 1.3: Computerprogramme als Binärdaten

Entscheidender historischer Schritt auf dem Weg zum heutigen Allzweckcomputer:

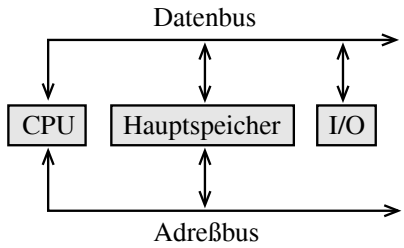
- Die Folge von Instruktionen, die ein Computer als „Programm" ausführen soll, lassen sich wie andere Daten in Binärform kodieren.
- Bei geeigneter Binärkodierung von Programmen lässt sich die Ausführung eines Programms auf die logische Manipulation dieser Binärdaten zurückführen.
- Programme lassen sich daher nicht nur ausführen, sondern auch wie beliebige andere Daten behandeln.

  $\longrightarrow$ Zum Beispiel durch andere Programme manipulieren oder über's Netz verschicken.

Von–Neumann–Modell:

- Keine Beschreibung realer Computer, sondern ein abstraktes *Muster* für die prinzipielle Organisation von Computern.
- *Grundidee*: Statt zur Lösung jedes neuen Problems eine neue Maschine zu bauen,
  - ◇ gibt es eine einzige, völlig problemunabhängige Maschine,
  - ◇ und jedes neue Lösungsverfahren wird nur noch als ein neues Programm realisiert
  - ◇ und zusammen mit seinen Daten im Speicher der Maschine unterschiedslos abgelegt.
- *Prinzipielle Aufteilung in Komponenten*:
  - ◇ Speicher,
  - ◇ Zentraleinheit (*Central Processing Unit*, *CPU*),
  - ◇ Schnittstelle zur Ein-/Ausgabe von Daten (I/O=Input/Output)
  - ◇ sowie Verbindungen dazwischen (*Busse*).

Veranschaulichung Von–Neumann–Modell:



Hauptspeicher:

- Der *Hauptspeicher* ist in eine feste Zahl 2^n von *Speicherzellen* aufgeteilt (auch *Maschinenworte* genannt).
- Die Speicherzellen haben fortlaufende *Adressen*: Binärzahlen der Länge n von $\boxed{000\ldots0} = 0$ bis $\boxed{111\ldots1} = 2^n - 1$ (vgl. Folie 16).
- Jede Speicherzelle besteht aus der gleichen Anzahl m von *Bits* (typisch heutzutage $m = 16$ oder $m = 32$).
- 1 Bit = eine elementare Speichereinheit zum Speichern eines einzelnen $0/1$–Wertes.
- Alle m Bits einer Speicherzelle können immer nur als Ganzes ausgelesen bzw. überschrieben werden.
- Neben dem Hauptspeicher gibt es einzelne weitere Speicherzellen für spezielle Aufgaben (*Register*), die nicht unbedingt genau m Bits haben.

Busse:

- Datenleitungen, die die einzelnen Elemente miteinander verbinden.
- *Breite*: Anzahl Bits, die gleichzeitig übertragen werden können.

  $\longrightarrow$ Ein Bus zum Auslesen/Überschreiben von ganzen Maschinenworten hat also Breite m.
- *Auslesen*: Kopieren ohne zu verändern.

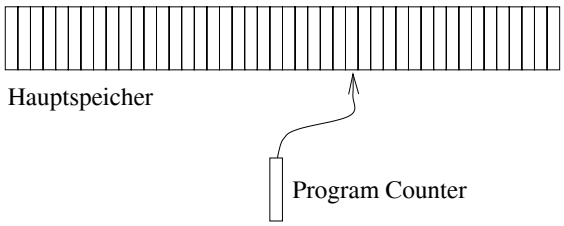
CPU:

- Besteht aus *Steuerwerk* und *Rechenwerk*.
- *Steuerwerk*: Steuert die Ausführung von Programmen.
- *Rechenwerk*: Erledigt die logischen und arithmetischen Datenmanipulationen.

  $\longrightarrow$ Auch *Arithmetic–Logical Unit* (*ALU*) genannt.

Ausführung von Programmen:

- *Vereinfachende Annahme*: Jede Instruktion in einem Programm belegt genau ein Maschinenwort.
- Kern des Steuerwerks ist der *Programmzähler* (*Program Counter*, *PC*).
  - ◇ Eine weitere Speicherzelle, aber mit n Bits (statt m).
  - ◇ *Inhalt*: Adresse der jeweils als nächstes auszuführenden Instruktion.
- Eine Art Uhr gibt einen „Takt“ vor, nach dem sich alle Operationen richten.



Inhalt einer einzelnen Instruktion:

- Im Bitmuster einer Instruktion ist kodiert,
  - ◇ wieviel an Daten die Instruktion braucht und wo sie zu finden sind,
  - ◇ wie die logischen Verschaltungen in der CPU konfiguriert werden müssen, damit genau die beabsichtigte arithmetisch–logische Operation auf den Daten ausgeführt wird, sowie
  - ◇ an welchen Hauptspeicheradressen bzw. in welchen Registern Ergebnisse abzulegen sind.
- *Also*: Der Durchlauf einer Instruktion durch die logischen Verschaltungen der CPU stößt die Öffnung der richtigen Kanäle an:
  - ◇ Von den Datenspeichern in die CPU.
  - ◇ Durch die datenmanipulierenden logischen Verschaltungen der CPU hindurch.
  - ◇ Von der CPU zurück in die Datenspeicher.

Ausführung einer einzelnen Instruktion in 5 Taktzyklen (idealisiert):

- Der Inhalt der Adresse, die im Program Counter steht, wird ausgelesen und in das Steuerwerk verbracht.
- Dieses Bitmuster durchläuft die logischen Verschaltungen des Steuerwerks und öffnet damit die richtigen Kanäle.
- Die Daten werden aus denjenigen Speicherstellen, deren Ausgangskanäle zur CPU dadurch geöffnet wurden, in die CPU ausgelesen.
- Die Daten durchlaufen diejenigen datenmanipulierenden logischen Verschaltungen der CPU, zu denen die Kanäle geöffnet wurden.
- Das Ergebnis wird an der Speicherstelle abgelegt, deren Eingangskanal zur CPU geöffnet wurde.

$\longrightarrow$ „Befehls–Pipeline“

Maschinencodes:

- Gibt es wohl soviele wie es Baureihen von Computern gibt.
- Unterscheiden sich auf den ersten Blick drastisch voneinander.
- Es gibt aber entscheidende gemeinsame Merkmale.

Wichtige gemeinsame Merkmale:

- Ein paar Bits sind in jeder Instruktion reserviert zur Identifizierung der Art der Instruktion (z.B. Addition).
- Ein ausgefeilter *Adressierungsmechanismus* bietet verschiedene Möglichkeiten zur Angabe von Hauptspeicher- und Registeradressen.
- Spezielle Instruktionen zum bedingten oder unbedingten Überschreiben des Program Counters („Sprungbefehle“) machen die Abarbeitung von Programmen erst wirklich flexibel.

Sprungbefehle:

- Die Instruktionen eines Programms folgen im Speicher typischerweise ohne Lücke aufeinander.

  $\longrightarrow$ Eine „normale“ Instruktion wird damit beendet, dass der Program Counter auf die Adresse des nächsten Maschinenworts nach der zuletzt abgearbeiteten Instruktion gesetzt wird.

  $\longrightarrow$ Bedeutet schlicht und einfach: Der Program Counter wird um 1 hochgezählt.

- *Sprungbefehl*: Eine Instruktion, deren Effekt darin besteht, den Program Counter auf einen anderen Wert zu setzen.
- Daten eines Sprungbefehls: der neue Wert.
- Ein Sprungbefehl sorgt also dafür, dass die Ausführung des Programms an eine andere Stelle im Macschinencode „springt“ und dort fortfährt.

Bedingter Sprungbefehl:

- Wertet zunächst einen logischen Ausdruck aus.
- Wenn der logische Ausdruck als „wahr“ zu verstehen ist, wird gesprungen.
- Ansonsten wird der Program Counter wie bei einer „normalen“ Instruktion hochgesetzt.
- *Zusammengefasst*: Ein bedingter Sprungbefehl ist ein Sprungbefehl,
  - ◇ der nur dann wirklich ausgeführt wird, falls eine gewisse logische Bedingung erfüllt ist,
  - ◇ und der keinerlei Effekt hat, wenn diese logische Bedingung *nicht* erfüllt ist.

Java–Beispiel:

```
summe = 0;
for ( i=1; i<n; i++ )
   summe += i;
```

Realisierung (informell formuliert):

- Instruktion 1: Überschreibe den Inhalt der Speicherzelle namens „summe“ mit dem Wert 0.
- Instruktion 2: Überschreibe den Inhalt der Speicherzelle „i“ mit dem Wert 1.
- Instruktion 3: Lese die Werte in „n“ und „i“ aus, subtrahiere den zweiten vom ersten und schreibe das Ergebnis in Register X.
- Instruktion 4: Falls der Inhalt von X kleiner oder gleich 0 ist, springe nach Instruktion 8.
- Instruktion 5: Addiere die Werte von „summe“ und „i“ und überschreibe den Inhalt von „summe“ mit dem Ergebnis.
- Instruktion 6: Erhöhe den Wert von „i“ um 1.
- Instruktion 7: Springe nach Instruktion 3.
- Instruktion 8: ...

Erläuterung:

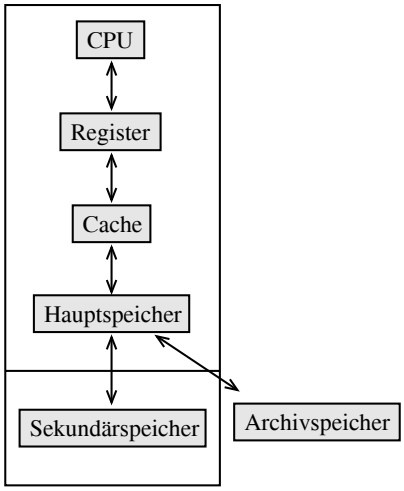
- So wie auf der letzten Folie muss man sich ungefähr die Struktur von Maschinencode vorstellen.
- Vor allem die Sprungbefehle sind ziemlich genau so verwendet worden, wie es in einer Übersetzung des kleinen Java–Beispiels auf der letzten Folie in Maschinencode tatsächlich aussehen würde.
- Natürlich besteht realer Maschinencode
  - ◇ nicht aus informellen deutschen Sätze wie auf der letzten Folie,
  - ◇ sondern aus Bitmustern von gewisser Länge,
  - ◇ die aber im Prinzip genau das bewirken, was die deutschen Sätze aussagen.
- Realer Maschinencode ist aber um einiges komplexer.
- Insbesondere würde in realem Maschinencode so ziemlich jede der acht Instruktionen im Beispiel auf der letzten Folie in Wirklichkeit aus mehreren Instruktionen besteht.

Abschließender Vorgriff:

- Sämtliche Sprachkonstrukte zur Steuerung des Programmverlaufs in Java oder anderen Programmiersprachen werden bei der Übersetzung ähnlich wie auf Folie 71 in Sprungbefehle aufgelöst.
- Welche Sprachkonstrukte sind gemeint: „`if`“–Verzweigung, „`while`“–Schleife, „`for`“–Schleife, Methodenaufruf, Beendigung einer Methode.
- Das Schema von Folie 71 bietet auch eine geeignete Möglichkeit, die Bedeutung solcher Sprachkonstrukte höherer Programmiersprachen wie Java präzise und unzweideutig zu definieren.
- Davon wird später in der Vorlesung noch ausgiebig Gebrauch gemacht werden.

  $\longrightarrow$ Ab Abschnitt 3.1.

Modellerweiterung: Speicherhierarchie



Erläuterung:

- Wie das Bild zeigt, gibt es mehrere Arten von Speichern in einem echten Rechner.
- Busse gibt es immer nur zwischen unmittelbar übereinanderliegenden Speichern (plus CPU):
  - ◇ CPU $\longleftrightarrow$ Register,
  - ◇ Register $\longleftrightarrow$ Cache,
  - ◇ Cache $\longleftrightarrow$ Hauptspeicher,
  - ◇ Hauptspeicher $\longleftrightarrow$ Sekundärspeicher,
  - ◇ Hauptspeicher $\longleftrightarrow$ Archivspeicher.
- Will man zum Beispiel Daten aus dem Sekundärspeicher in die CPU laden, dann muss man sie
  - ◇ aus dem Sekundärspeicher in den Hauptspeicher,
  - ◇ aus dem Hauptspeicher in den Cache,
  - ◇ aus dem Cache in ein Register und
  - ◇ aus dem Register schließlich in die CPU laden.

Register:

- Enthalten die Daten und Programmanweisungen, die unmittelbar zur Verarbeitung durch die CPU anstehen.
- Sind teilweise für Spezialaufgaben eingerichtet (z.B. Program Counter) und teilweise für die Zwischenspeicherung beliebiger Inhalte durch die CPU.

Cache:

- Eine Reaktion der Hardwarebauer auf den „Von–Neumannschen Bottleneck“ und auf die Erfahrungen mit dem typischen Lokalitätsverhalten von realen Computerprogrammen.
- „Von–Neumannscher Bottleneck“:

  Ein Zugriff auf den Hauptspeicher benötigt bei weitem mehr Zeit als die eigentliche Ausführung einer Instruktion.
- Lokalitätsverhalten:

  Die Daten, auf denen ein Programm arbeitet, lassen sich typischerweise derart in eine lineare Reihenfolge bringen, dass zwei Maschinenworte, auf die unmittelbar nacheinander zugegriffen wird, in der Regel sehr nah beieinander liegen.

Hintergrund zum Lokalitätsverhalten:

- Für Zugriffe auf auszuführende Instruktionen gilt das Lokalitätsprinzip in der Regel, da
  - ◇ Sprungbefehle relativ selten sind und
  - ◇ die Mehrzahl der Ausführungen von Sprungbefehlen den Program Counter im allgemeinen auch nur um einen kleinen Wert verändern (etwa im Beispiel auf Folie 71).
- Für Zugriffe auf Daten gilt:

  Bei vielen, insbesondere laufzeitintensiven Anwendungen wird die Laufzeit durch lineare Durchläufe durch große Datenmengen dominiert.
- *Konkrete Beispiele*:
  - ◇ Durchlauf durch Tabellen in einer Datenbank,
  - ◇ numerische Berechnungen (z.B. auf Matrizen) in graphischer Datenverarbeitung, Simulation u.ä.

Zurück zum Thema Cache selbst:

- Ein im Vergleich zum Hauptspeicher relativ kleiner Zwischenspeicher zwischen CPU–Registern und Hauptspeicher.
- Dafür aber schnelle Transferzeit zwischen CPU–Registern und Cache.

  $\longrightarrow$ Eigentlicher Sinn und Zweck von Caches.
- Wenn die CPU den Inhalt einer Hauptspeicheradresse anfordert,
  - ◇ wird erst nachgeschaut, ob er schon im Cache zwischengespeichert ist,
  - ◇ und falls nicht, wird nicht nur dieses einzelne Maschinenwort in den Cache geladen,
  - ◇ sondern gleich eine ganze Sequenz von Maschinenworten auf einmal.
- *Konsequenz*: Wenn der (System-)Programmierer die Daten seines Programms „cache–bewusst“ organisiert hat, ist nach dem Lokalitätsprinzip die Wahrscheinlichkeit sehr groß, dass die Mehrzahl der in nächster Zeit angeforderten Daten dadurch mit in den Cache geladen werden.

Sekundärspeicher:

- Heutzutage in der Regel Festplatte.
- Im Gegensatz zum Hauptspeicher permanente Speicherung der Daten (auch wenn der Computer ausgeschaltet wird).
- Um Größenordnungen höhere Speicherkapazität als der Hauptspeicher.
- *Allerdings*: auch um Größenordnungen höhere Zugriffszeiten.
- *Konsequenz für Hardwarebauer*:
  Der Zugriff auf den Sekundärspeicher wird so organisiert, dass nicht einzelne Maschinenworte, sondern immer gleich größere Speicherbereiche (*Seiten*) eingelesen werden.
- *Konsequenz für* (*System-*)*Programmierer*:
  Auf einer Seite im Sekundärspeicher sollten möglichst immer Daten abgespeichert werden, die mit großer Wahrscheinlichkeit fast gleichzeitig benötigt werden (*Clustering*).

  $\longrightarrow$ Wird in Datenbanksystemen u.ä. systematisch gemacht.

Abschnitt 1.4: Betriebssysteme

- Der „Makler“ zwischen
  - ◇ Hardware einerseits und
  - ◇ Anwendungsprogrammen und Endbenutzern andererseits.
- Kontrolliert die Datenverwaltung, Benutzerverwaltung, die I/O–Schnittstellen sowie den Ablauf aller Prozesse.

Beachte:

- Die diversen Betriebssysteme sehen auf den ersten Blick sehr unterschiedlich aus.
- Die Unterschiede sind auf den zweiten Blick aber nicht mehr so groß.
- Im folgenden lehnen wir uns trotzdem an eine konkrete Familie von Betriebssystemen an: *UNIX*.

Aufgabengebiet 1: User und Groups

- Jeder Arbeitsbereich ist einem virtuellen *Benutzer* (*User*) zugeordnet.
- Jede humanbiologische Funktionseinheit der Spezies *Homo Sapiens* erhält Rechnerzugang über die Einrichtung eines Users und die Weitergabe des zugehörigen Passwortes an diese Funktionseinheit.
- In einer *Group* sind mehrere User zu einer als Ganzes mit einem Gruppennamen ansprechbaren Einheit zusammengefasst.
- Groups dürfen sich beliebig überlappen.
- Jeder User gehört zu genau einer der Gruppen, denen er angehört, besonders fest zu.
- Diese Gruppe nennt man die *Primärgruppe* des Users.
- Alle anderen Gruppen, denen der User angehört, nennt man seine *Sekundärgruppen*.

UNIX–Attribute von Usern:

- Benutzerkennung (*Login–Name*).
- Momentanes Passwort.
- *Home Directory*: der zugeordnete Arbeitsbereich (siehe Folie 93).
- Die *Primärgruppe*.
- Beliebig viele *Sekundärgruppen* (auch keine einzige möglich).

Aufgabengebiet 2: Files

- Alle Daten sind in UNIX in Form von *Files* (*Dateien*) abgelegt.
- Auch Programme, Mailboxes, WWW–Seiten, Bilder, Filme, Tonspuren etc. sind unter UNIX ganz normale Files.
- In gewissen Sinne sind also z.B. Mail–Reader nichts anderes als besonders komplexe Editoren für Files, deren Inhalt nach bestimmten Konventionen formatiert sind:

  Eine Sequenz von Zeilenblöcken (eben den Emails), die jeweils aus einem (nach strengen Regeln formatierten) *Header* (Kopf) und dem eigentlichen Inhalt der Email (*Body*) bestehen.

Beispiel: Ausschnitt aus einer Mail-Datei:

```
[...]
From mw@allgemeineinformatik.de
Date: Fri, 26 Sep 2003 14:08:52 +0200
Subject: Testmail fuer das Skript
From: Markus Weimer <mw@allgemeineinformatik.de>
To: Markus Weimer <mw@allgemeineinformatik.de>
Message-ID: <BB99F8F4.1FE7>
Content-type: text/plain; charset="ISO-8859-1"
Content-Transfer-Encoding: 8bit

Hallo,

dies ist eine Testmail fuer
das Skript in Allgemeine Informatik.

Viel Spass in der Vorlesung,

Markus Weimer

From [...]
```


UNIX–Attribute von Files (nicht vollständig):

- Besitzer (ein User).
- Besitzende Gruppe
  (die primäre oder eine sekundäre Group des besitzenden Users).
- Drei *Zeitstempel* (*Time Stamps*):
  - ◇ *Access Time*: Zeitpunkt des letzten lesenden oder schreibenden Zugriffs (der spätere von beiden).
  - ◇ *Modification Time*: Zeitpunkt des letzten schreibenden Zugriffs.
  - ◇ *Status Change Time*: Zeitpunkt der letzten Änderung der Attribute des Files.
- Zugriffsrechte...

Zugriffsrechte für Files:

- Einteilung 1: lesen, schreiben, ausführen.
- Einteilung 2: Besitzer (*Owner*), besitzende Gruppe, Rest der Welt.

$\longrightarrow$ Es gibt insgesamt $3 \times 3 = 9$ Freiheitsgrade, um einzelnen Benutzerkreisen Rechte zu geben bzw. zu nehmen.

Ausführrecht:

- Ist zwar für jedes File definiert,
- ist aber sinnlos für Files, die nicht Programme (o.ä.) sind.
- *Bedeutung*: Das File darf als Programm vom betreffenden Benutzerkreis ausgeführt werden.

Anzeigen der wichtigsten Attribute:

```
$ ls -l MyFile
-rwxr-xr-- 1 weihe student 12822 Sep 29 18:45 MyFile
```

Erläuterungen zu „ls -l“ allgemein:

- Das Kommando „`ls`“ ist in erster Linie dafür da, sich den Inhalt von Directories anzeigen zu lassen (vgl. Folie 91 ff.).
- Einige Optionen von „`ls`“ sind dafür da, um festzulegen, was alles an Informationen für die einzelnen Files ausgegeben werden soll.
- Durch Option „`-l`“ (=long) wird festgelegt, dass ein paar Standardinformationen ausgegeben werden, die in den meisten Fällen alle Information enthalten, die man sucht.

Erläuterungen zur Ausgabe auf der letzten Folie:

- Das erste Minus besagt: „normales File“ (vgl. Folie 98).
- Die restlichen neun Zeichen der ersten Spalte geben die Zugriffsrechte an.
- Die „12822“ ist die Größe von File „MyFile“ in Bytes.
- User und Group sind „weihe“ bzw. „student“.
- Der Zeitstempel „Sep 29 18:45“ ist die Modification Time.
- Die „1“ vor „weihe“ wird hier nicht erklärt.

  ⟶ Stichwort für Freaks: Hard Links.
- Weitere Informationen finden Sie nach Eingabe von `man ls` an einem UNIX-Rechner.

Anzeige der Zugriffsrechte „`rwxr-xr--`“:

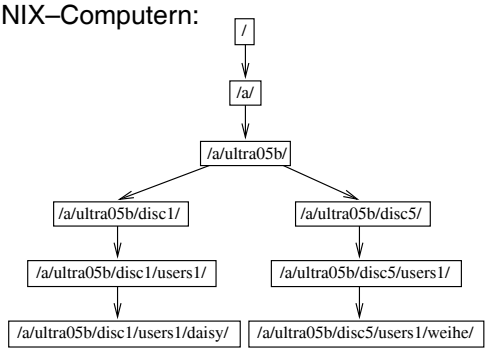
- Ein „r“=„read“, „w“=„write“ oder „x“=„execute“ besagt, dass das jeweilige Recht gegeben ist.
- Steht statt dessen ein „–“ an dieser Position, ist das jeweilige Recht statt dessen genommen.
- Zuerst kommen die drei Rechte des Owners, dann die drei Rechte der besitzenden Group, schließlich die drei Rechte für den Rest der Welt.
- In jedem der drei Tripel kommt immer zuerst Leserecht („r“), dann Schreibrecht („w“), schließlich Ausführrecht („x“).

⟶ Die Anzeige „`rwxr-xr--`“ besagt also, dass der User alle Rechte hat, die Group nur Lese- und Ausführrecht und der Rest der Welt nur Leserecht.

Aufgabengebiet 3: Directories

- Files sind hierarchisch in *Verzeichnissen* (*Directories*) organisiert.
- *Hierarchisch* heißt: Zum Inhalt einer Directory können
  - ◇ nicht nur Files gehören,
  - ◇ sondern auch wieder Directories (die unmittelbaren *Subdirectories* der Directory).
- *Pfad*: Jedes File und jede Directory ist das Ende eines *Pfades* von Directories, der mit der „*Root–Directory*“ beginnt.
- Der Name eines Files (oder einer Directory)
  - ◇ ist allgemein nicht aus sich heraus eindeutig,
  - ◇ aber zusammen mit dem Pfad ist er es.

Mini–Ausschnitt aus der Directory–Struktur auf unseren UNIX–Computern:



Erläuterung:

- „`/a/ultra05b/disc5/users1/weihe`“ ist die Home Directory des Users „`weihe`“.
- „`/a/ultra05b/disc1/users1`“ ist eine weitere Directory, in der Home Directories von Usern gesammelt werden.
- „`/a/ultra05b/disc1/users1/daisy`“ existiert allerdings nicht und steht nur beispielhaft für tatsächlich existierende Subdirectories von „`/a/ultra05b/disc1/users1`“.

Namensregeln:

- Der „/“ („*Slash*“) wird als Trennsymbol verwendet:
  - ◇ zwischen dem Namen einer Directory und dem Namen einer ihrer unmittelbaren Subdirectories,
  - ◇ zwischen dem Namen einer Directory und dem Namen eines Files in dieser Directory.
- Der Slash leitet auch den Namen eines Pfades ein.
- Die Root–Directory heißt einfach nur „/“.
- *Sondernamen*:
  - ◇ „`~weihe`“ steht für den absoluten Pfad der Home Directory von User „`weihe`“.
  - ◇ „~“ allein steht für den absoluten Pfad der eigenen Home Directory.

Genauer formuliert:

- Dies sind Namensregeln für *absolute Pfade*.
- Man kann Files und Directories auch durch *relative Pfade* bzgl. anderer Directories angeben.

⟶ Siehe nächste Folie.

Relative Pfade:

- Eine *Shell* (d.h. der Prozess in einem `xterm`–Fenster) hat zu jedem Zeitpunkt genau eine *Working Directory*.

  ⟶ Eigentlich jeder Prozess, siehe Folie 108 ff.
- Der Name eines Files oder einer Directory kann auch relativ zur Working Directory angegeben werden.
- Die Working Directory kann in der Shell immer durch „.“ angesprochen werden.
- Die Directory, die in der Hierarchie genau eine Stufe über der Working Directory steht, wird durch „..“ angesprochen.

Beispiel (vgl. Folie 92):
Die Working Directory sei zu Beginn „`~weihe/.nsmail`“.

- `$ nedit ./Inbox`
  ⟶ Das File namens „`Inbox`“ in der Working Directory wird gelesen.
- `$ nedit Inbox`
  ⟶ Dasselbe Ergebnis, d.h. „`./`“ darf weggelassen werden.
- `$ cd ../../../disc1/users1/daisy/`
  ⟶ Neue Working Directory „`~daisy`“.
- `nedit ../../../disc5/users1/weihe/nsmail/Inbox` ⟶
  Das File namens „`Inbox`“ in der unmittelbaren Subdirectory „`.nsmail`“ der Home Directory „`~weihe`“ von User `weihe` kann nun mit diesem Kommando gelesen werden.

UNIX–Attribute von Directories (unvollständig):

- Praktisch dieselben Attribute wie bei Files und mit identischer Bedeutung.
- *Ausnahme*: Zugriffsrechte haben bei Directories eine eigene, aber (halbwegs) analoge Bedeutung.
- *Genauer*:
  - ◇ *Leserecht*: Man darf sich den Inhalt der Directory (also die darin unmittelbar enthaltenen Files und Subdirectories) mit Kommandos wie „`ls`“ anzeigen lassen.
  - ◇ *Schreibrecht*: Man darf Files und Directories in die Directory hineinstellen oder daraus entfernen.
  - ◇ *Ausführrecht*: Man darf die Directory (oder eine Subdirectory) mit Kommandos wie „`cd`“ zur Working Directory machen.

Directories als Files:

```
$ cd ~weihe
$ ls -l
...
drwx------ 2 weihe algo 1024 Feb 1 1999 .elm
...
```

Erläuterung:

- Es ist kein Zufall, dass Directories dieselben Attribute wie Files haben und durch „`ls`“ gleich behandelt werden.
- *Erinnerung* an Folie 84: Mailboxes u.ä. haben wir schon als „normale“ Files erkannt.
- Directories sind in UNIX ebenfalls im Grunde nur Files, aber von anderer Art als „normale“.
- *Kenntlichmachung* bei „`ls -l`“: In der Zeile zu einer Directory steht ein „`d`“ anstelle eines „`-`“ als erstes Zeichen.

Exkurs: Abstrakte Files in UNIX

- Hinter der letzten Folie steht eine allgemeine UNIX–Philosophie: „Alles ist File“.
- *Das heißt*: Das UNIX–Konzept „File“ bündelt diverse Systemkonzepte, die
  - ◇ auf den ersten Blick überhaupt nichts miteinander zu tun zu haben scheinen,
  - ◇ auf den zweiten Blick aber so viel gemeinsam haben (z.B. Attribute), dass sie weitgehend gleich behandelt werden können.

⟶ Beispiel für das allgemeine Bestreben in der Informatik, Details soweit wie möglich „wegzuabstrahieren“ und die „reinen“ Konzepte herauszukristallieren.

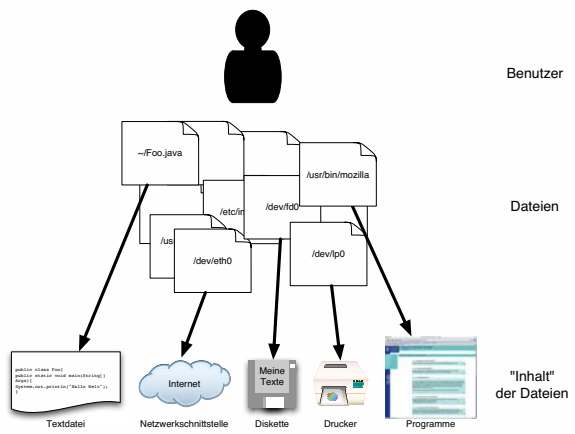
Frage: Was ist noch so alles „File“ unter UNIX?

⟶ Antwort auf der nächsten Folie.

Weitere Beispiele als Antwort:

- Ein *Character Special File* ist die Repräsentation eines zeichenorientierten Ein-/Ausgabegeräts, mit der es ein UNIX–Nutzer zu tun bekommt, wenn er systemnah mit dem Gerät arbeiten will, zum Beispiel:
  - ◇ Von Tastatur lesen heißt, aus einem bestimmten Character Special File lesen, das in einer bestimmten Directory steht.
  - ◇ Auf den Bildschirm (`xterm`–Fenster) schreiben heißt, auf ein bestimmtes Character Special File in einer bestimmten Directory schreiben.
- Ein *Block Special File* repräsentiert hingegen ein Ein-/Ausgabegerät, das immer gleich ganze Blöcke fester, spezifischer Größe von Zeichen einliest und/oder ausgibt (z.B. Diskettenlaufwerk).
- Ein *Socket* (Folie 122) ist ein File, in das ein Prozess hineinschreibt, und aus dem ein anderer Prozess herausliest.

Bildlich:



Aufgabengebiet 4: Prozesse

Beachte:

- Ein *Programm* ist eine Folge von Instruktionen in einer für den Computer ausführbaren Form.

  ⟶ Abgespeichert als „normales“ File (siehe Folie 84).

- Ein *Prozess* ist die Ausführung eines Programms.

  ⟶ Mehrere Prozesse können gleichzeitig, aber völlig unabhängig voneinander dasselbe Programm ausführen.

Prozessmanagement:

- Ein Benutzer kann (fast) beliebig viele Prozesse gleichzeitig laufen lassen.
- Es können auch mehrere Benutzer zugleich auf demselben Computer (aber normalerweise natürlich über verschiedene Terminals) eingeloggt sein und Prozesse laufen lassen.
- Zugleich laufen noch zentrale Dienste wie der Mail–Server (Folie 115).
- *Aber*: Die meisten „normalen“ Computer haben nur eine CPU (oder einige wenige).

Frage: Wie passt das zusammen?

Simple (eigentlich eher übersimple) Antwort:

- Das Betriebssystem teilt jedem Prozess reihum eine kurze Zeitspanne auf der CPU zu.
- Die Zeitspannen sind so kurz, dass der Benutzer am Terminal nicht mitbekommt, dass die Prozesse in Wirklichkeit nicht parallel, sondern *serialisiert* ablaufen.

  ⟶ Außer bei sehr hoher Auslastung, wenn es zu Engpässen kommt.

- Die CPU–Zeit wird dabei möglichst „gerecht“ unter den Prozessen verteilt.
- Wartet ein Prozess auf ein Ereignis (z.B. Benutzereingabe), wird er bis zum Eintreten des Ereignisses aus der Warteschlange genommen.
- Prozesse sind mit unterschiedlichen *Prioritäten* ausgestattet.

  ⟶ Prozesse mit höherer Priorität werden öfter und/oder länger bedient.

Priorität von Prozessen und Kommando „nice“:

- Alle Prozesse eines normalen Users werden normalerweise mit einer bestimmten Standardpriorität gestartet.

  ⟶ Ungefähr „mittlere“ Priorität.

- Unter UNIX ist das Kommando „`nice`“ dafür da, einen Prozess mit anderer als der Standardpriorität zu starten.

- Abgesehen von den Administratoren darf jeder User die Priorität der von ihm gestarteten Prozess aber immer nur verringern, nie erhöhen.

Frage: Warum zum Teufel soll man seinen eigenen Prozessen weniger Priorität geben als möglich und erlaubt wäre?

⟶ Antwort auf der nächsten Folie.

Warum soll man also auf Priorität verzichten:

- Manche umfangreicheren Rechenprozesse u.ä. brauchen Stunden, Tage oder Wochen.
- Auf ein bisschen Zeit mehr oder weniger kommt es dabei dann auch nicht an.
- Es wäre also nett (engl. „nice“), wenn der Prozess wenig oder keine Rechenzeit beansprucht, während irgendein User interaktiv mit dem Rechner arbeitet.
- Das geht sehr gut durch eine geringere Priorität:
  - ◇ Solange andere Prozesse mit höherer Priorität laufen, bekommt dieser Prozess kaum Rechenzeit zugeteilt.
  - ◇ Ansonsten kann er den Rechner sehr stark für sich beanspruchen.
- Wann kommt der Prozess dann vorwärts:
  - ◇ Wenn praktisch niemand arbeitet (z.B. frühes Morgengrauen).
  - ◇ Aber durchaus auch zwischendurch, zum Beispiel wenn „konkurrierende“ interaktive Prozesse auf Benutzereingaben warten und in dieser Zeit nichts tun.

Handhabung von „nice“ anhand eines Beispiels:

```
nice -n15 hallo -l1 bla
```

Erläuterungen:

- Ein Programm namens „`hallo`“ soll mit Option „`-l 1`“ und weiterem Argument „`bla`“ gestartet werden.

  ⟶ Zum Beispiel könnte „`bla`“ der Name eines Files sein.

- Durch vorangestelltes „`nice`“ wird dieser Prozess nicht direkt, sondern indirekt durch das Programm „`nice`“ aufgerufen.
- Durch Option „`-n15`“ zu „`nice`“ wird die Priorität von „`hallo -l1 bla`“ um 15 *vermindert*.
- Man kann die Werte $1 \ldots 19$ hinter „`-n`“ einsetzen.

Attribute von Prozessen (unvollständig):

- Besitzer (ein User)
- Eindeutige Zahlenkennung (*Prozess–ID*)
- Bisher verbrauchte Rechenzeit
- Priorität
- Momentane Working Directory (vgl. Folie 95 ff.)
- Status des Prozesses:
  - ◇ *Running*: Arbeitet gerade (und verbraucht dabei Rechenzeit).
  - ◇ *Runnable*: In der Warteschlange.
  - ◇ *Sleeping*: Wartet auf einen Event.
  - ◇ *Zombie*: Prozess sollte eigentlich schon beendet sein, ist dem Betriebssystem aber aus dem Ruder gelaufen und läuft (mehr oder weniger unkontrollierbar) immer weiter.

Prozesse starten (stark vereinfacht):

- Nach dem Hochfahren („Booten“) eines Computers startet zunächst einmal ein Hauptprozess.
- Jeder Prozess kann seinerseits (fast) beliebig neue Prozesse erzeugen („Kind–Prozesse“).

  ⟶ Alle Prozesse entstehen also direkt oder indirekt aus dem Hauptprozess.
- Ein Elter–Prozess „vererbt“ seinen Kind–Prozessen einige Attribute (z.B. Working Directory).
- Der Kind–Prozess kann diese Attribute auch durch eigene Setzungen überschreiben.

Shell:

- Der Prozess, der in einem `xterm`–Fenster läuft.
- Dient in erster Linie der Kreation neuer Prozesse.

  ⟶ Durch Aufruf des zugrundeliegenden Programms als Kommando an die Shell.
- Die Shell ist der Elter–Prozess jedes Prozesses, der durch Kommandoaufruf über diese Shell gestartet wurde.
- Ein so gestarteter Prozess kann dann ebenfalls auf dieses `xterm`–Fenster Ausgaben schreiben.
- *Wichtige Unterscheidung*: Prozesse können im Vordergrund oder im Hintergrund gestartet werden.

  ⟶ Details auf der nächsten Folie.

Start im Vorder- oder Hintergrund:

- *Äußerliches Merkmal*: Man startet einen Prozess in der Shell „im Hintergrund“, indem man an den Kommandoaufruf ein „`&`“ anhängt:
  - ◇ `nedit MyFile`
  - ◇ `nedit MyFile &`
- *Technischer Unterschied*:
  - ◇ Wird der Prozess im Vordergrund gestartet, wird das `xterm`–Fenster dem gestarteten Prozess übergeben.
  - ◇ Ansonsten bleibt das `xterm`–Fenster im Besitz des Shell–Prozesses.
- *Konsequenz*: Die Eingaben, die man in einem `xterm`–Fenster mit Tastatur und Maus macht, gehen
  - ◇ im ersten Fall an den gestarteten Prozess,
  - ◇ im zweiten Fall an die Shell.

  ⟶ Siehe später, Folie 135, für systematischere Erläuterungen.

Shell und Hintergrundmenüs:

- Hintergrundmenüs stehen zur Kreation neuer Prozesse heutzutage als Alternative zur Shell zur Verfügung (vgl. auch Folie 136).
- *Vorteil*: Bequemer und intuitiver in der Handhabung.
- *Nachteil*: Weniger flexibel.
  - ◇ Nur die paar Kommandos, die im Menü eingetragen sind, können so aufgerufen werden.
  - ◇ Die vielfältigen Möglichkeiten, diese Programme durch Angabe von Optionen zu konfigurieren, sind ebenfalls nur rudimentär oder gar nicht nutzbar.
  - ◇ Die vielfältigen Möglichkeiten der Shell selbst sind nicht nutzbar.
- Der Versuch, diese Nachteile in einem menü–orientierten Ansatz auszubügeln, führt wohl fast unvermeidlich(!?) zu komplexen, unübersichtlichen Kaskaden von Menüs.

⟶ Beispiel für den allgemeinen Zielkonflikt zwischen Flexibilität und Übersichtlichkeit.

Working Directory nach Prozess–Start:

- Wie auf Folie 109 gesagt, startet ein Prozess typischerweise mit der Working Directory seines Elter–Prozesses.
- Eine Shell startet normalerweise mit der Home Directory des Besitzers des Shell–Prozesses als Working Directory.
- Typische Regel bei Editoren:
  - ◇ Wenn der Editor mit einem Filenamen als Kommandozeilenargument aufgerufen wird, wird die Directory dieses Files die Working Directory des Editor–Prozesses. Beispiel: `nedit /a/b.txt`. Dann ist `/a/` das Working Directory des Editors.
  - ◇ Wenn der Editor ohne einen Filenamen aufgerufen wird, erbt er seine Working Directory vom Elter–Prozess (meist eine Shell).

UNIX–Kommando „cd“:

- Dieses Kommando dient dazu, die Working Directory einer Shell zu verändern.
- *Genauer*: Wenn „`cd`“ mit dem Namen einer Directory als Argument in einer Shell aufgerufen wird, wird dieses Argument die neue Working Directory dieser Shell.
- *Anmerkung*: Eigentlich ist „`cd`“
  - ◇ kein eigenständiges UNIX–Kommando.
  - ◇ sondern ein Bestandteil der Shell,
  - ◇ der aber so gestaltet ist, dass er für den User wie ein echtes UNIX–Kommando aussieht.

Beispiel:

```
[bellman:/a] weimer% ls
b.txt dir
[bellman:/a] weimer% cd dir
[bellman:/a/dir] weimer% ls
nocheintext.txt
```


Daemons:

- Ein *Daemon* (sprich: dih–men mit extrem kurzem „e“) ist ein Prozess, der ununterbrochen über längere Zeitspannen existiert und „im Hintergrund“ auf Arbeit wartet.
- Vor allem *Server*–Prozesse: Mail–Server, WWW–Server,...
- Beispiel Mail–Server (vereinfacht):
  - ◇ Mails von lokalen Usern und von außen werden nach strikten Formatierungsregeln zunächst in eindeutig definierten Adressbereichen abgelegt.
  - ◇ Wann immer der Mail–Server im Status *running* ist, fragt er den Inhalt dieser Bereiche ab.
  - ◇ Falls neue Emails da sind, werden sie vom Mail–Server bearbeitet und in die Mailboxes (vgl. Folie 84 ff.) der jeweiligen Adressaten kopiert.

UNIX–Interprozesskommunikation I: Signale

Ein Prozess kann einem anderen Prozess ein *Signal* schicken.

- Mit Kommando „`kill`“ schickt die Shell, in der das Kommando aufgerufen wurde, das spezifierte Signal an den Prozess, dessen Prozess–ID spezifiziert wurde.
- *Beispiele* für Signale: „`CONT`“, „`HUP`“, „`KILL`“, „`STOP`“, „`TERM`“.
- *Vorgehensweise*:
  - ◇ Mit dem UNIX–Kommando „`ps`“ lässt man sich Informationen zu den momentan auf dem Rechner laufenden Prozessen anzeigen.
  - ◇ Aus diesen Anzeigen sucht man die ID des Prozesses heraus, dem man das Signal schicken will.
  - ◇ Das Signal schickt man durch Aufruf von „`kill`“ mit dem Signal als Option (Minus vor dem Signalnamen) und der Prozess–ID als Argument.

Beispiel:

```
[bellman:/Skript] weimer% ps

PID TT STAT TIME COMMAND
469 std Ss 0:00.10 -csh (tcsh)
486 std S 0:01.48 xdvi folien_master.dvi
2311 p2 Ss+ 0:00.08 -csh (tcsh)

[bellman:/Skript] weimer% kill -HUP 486
```

Dies sendet das Signal ,,HUP" an den Prozess mit der Nummer ,,486", in diesem Fall läuft das Programm ,,xdvi" mit dieser Prozess-ID.

Default–Reaktion:

- Für jedes mögliche Signal ist eine *Default*-Reaktion festgelegt, zum Beispiel
  - ◇ Abbruch des Prozesses bei „*HUP*“ (*Hangup*), „*KILL*“ und „*TERM*“ (*Terminate*),
  - ◇ einschlafen bei „*STOP*“,
  - ◇ wieder aufwachen bei „*CONT*“ (*Continue*).
- Ein Programm kann auch so geschrieben sein, dass ein Prozess, der dieses Programm ausführt, auf einzelne Signale anders reagiert.

  ⟶ Wird hier nicht weiter ausgeführt.
- *Sinnvolles Beispiel*: Ein Editor reagiert auf „*HUP*“, indem er die letzten, noch nicht abgespeicherten Modifikationen des Fileinhalts in einer Sicherheitskopie abspeichert — und sich dann selbst beendet.

Signale mit Default–Reaktion Abbruch:

- Einige wenige dieser Signale (z.B. Signal `KILL`) lassen sich von Prozessen grundsätzlich nicht abfangen.

  ⟶ Hierarchie zwischen „weichen“ und „harten“ Signalen zum Abbruch.
- Auf den abfangbaren Signalen zum Abbruch gibt es ebenfalls eine (rudimentäre) Hierarchie durch eine Konvention, welche Signale wie „weich“ von Prozessen abgefangen werden sollten, z.B.:
  - ◇ `HUP` wird als „weich“ interpretiert und von vielen Prozessen zum „Aufräumen“ abgefangen, bevor sich der Prozess dann von selbst beendet.
  - ◇ `TERM` wird als eher „hartes“ Signal interpretiert und von vielen Prozessen gar nicht oder mit weniger weichen Reaktionen abgefangen.
- Beim Ausloggen sendet das Betriebssystem das Signal `HUP` zum Abbruch an alle Prozesse des Users.

UNIX–Interprozesskommunikation II: Pipes

- Durch „|“ beim Aufruf in der Shell wird ein Prozess gebildet, der aus den zwei angegebenen Prozessen gebildet wird.
- *Das heißt*:
  - ◇ Die Ausgaben des ersten Prozesses, die sonst auf den Bildschirm geschrieben worden wären, werden statt dessen in einem Zwischenpuffer abgespeichert.
  - ◇ Die Eingaben für den zweiten Prozess, die sonst von der Tastatur kämen, kommen nun aus diesem Zwischenpuffer.

Beispiele:

- `ls | wc -w`
  - ◇ „`wc -w`“ zählt die Wörter in einem Text.
  - ◇ Diese Pipe zählt also die Files in der Working Directory.
- `ls -l | grep hallo`
  - ◇ „`grep hallo`“ sucht aus einem Text alle Zeilen heraus, in denen die Zeichenkette „`hallo`“ vorkommt (vgl. Folie 33 ff.).
  - ◇ Diese Pipe gibt die „`ls-l`“–Zeilen aller Files in der Working Directory aus, deren Namen die Zeichenkette „`hallo`“ enthält.
- `ps | grep nedit`
  - ◇ Diese Pipe zeigt also Informationen zu allen Prozessen an, die mit dem Programm „`nedit`“ gestartet wurden.

UNIX–Interprozesskommunikation III: Sockets

- Ein „Kommunikationskanal“ zwischen zwei Prozessen, über den der eine Prozess Daten vom anderen bekommen kann.
- Kann im Gegensatz zu einer Pipe im allgemeinen nicht beim Aufruf der beiden Programme in der Shell eingerichtet werden,
- sondern die beiden zugrundeliegenden Programme müssen schon so programmiert sein, dass die Prozesse sich eigenständig auf die Einrichtung eines Sockets verständigen.
- Auch Sockets sind eine spezielle Art von Files mit den entsprechenden Attributen (vgl. Folie 100).
- Sockets können auch zwischen Prozessen auf verschiedenen Computern eingerichtet werden.
- *Beispiel*: Programme wie „`telnet`“, „`rlogin`“ und „`ssh`“ zum Einloggen auf anderen Computern basieren intern auf Socket–Paaren (ein Socket für jede Richtung der Kommunikation).

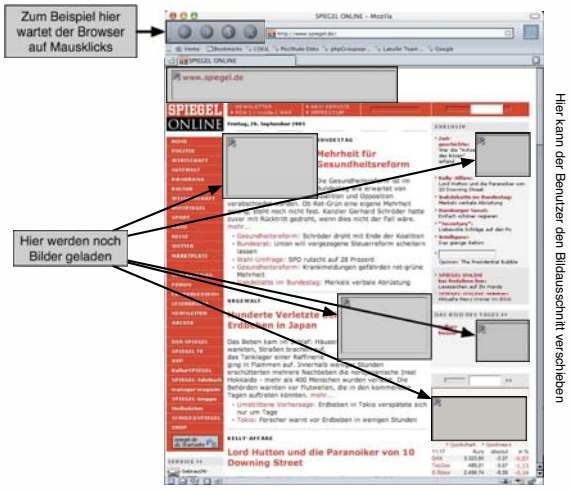
Parallele Prozesse:

Ein Prozess erzeugt ein oder mehrere Kind–Prozesse (Folie 109) für Teil- oder Sonderaufgaben.

Beispiel:

- Ein WWW–Browser kann offensichtlich mehrere Dinge gleichzeitig, zum Beispiel:
  - ◇ Auf Klick des Benutzers hin (im Balken rechts) den Ausschnitt aus einer längeren WWW–Seite hoch- und hinunterschieben („scrollen“).
  - ◇ Die noch nicht fertig geladenen Bilder weiter laden.
  - ◇ Auf Mausklicks auf den Buttons warten.
  - ◇ Usw.
- Jede einzelne dieser Aktivitäten ist ein eigener, selbstständiger Prozess.
- Alle diese Prozesse sind aus dem Hauptprozess des WWW–Browsers erzeugt worden.

Das Beispiel im Bild:



Variationen des Konzepts Parallelprozesse:

- *Verteilte Prozesse*: Die Kind–Prozesse werden nicht auf demselben, sondern über Netzwerke auf anderen Computern erzeugt.

  $\longrightarrow$ Siehe Lehrveranstaltungen zu verteilten Systemen für eine systematische Einführung.

- *Threads*:

  ◇ Parallelprozesse werden innerhalb eines Prozesses nur *simuliert*, ohne dass das Betriebssystem irgendetwas davon mitbekommt.

  ◇ Die Kommunikation zwischen solchen simulierten Parallelprozessen kann dann enger und effizienter gestaltet werden, als es vom Betriebssystem angeboten wird.

  ◇ In einigen Programmiersprachen (z.B. Java) ist diese Möglichkeit von vornherein in die Sprache eingebaut.

  $\longrightarrow$ Später in der Vorlesung mehr dazu (Abschnitt 4.6, Folie 584).

Beachte:

- Solange alle kollaborierenden Prozesse auf einer einzigen CPU laufen (vgl. Folie 104), könnte das ganze Programm absolut gleichwertig als eine Ein–Prozess–Lösung verwirklicht werden.
- Die Effizienz (v.a. Laufzeit) einer Ein–Prozess–Lösung dürfte in diesem Fall sogar signifikant besser sein, weil die Koordinierung mehrerer zusammenarbeitender Prozesse ebenfalls Laufzeit kostet.
- *Aber*:

  ◇ Parallelisierung dient nicht nur der Effizienzsteigerung,

  ◇ sondern auch einer möglichst leicht verständlichen *Strukturierung* des Programms.

  $\longrightarrow$ Programme, zu deren innerer Logik das asynchrone Auseinanderfallen in mehrere autonome Teile gehört, sollten auch parallelisiert implementiert werden.

Wieder Beispiel WWW–Browser:

- Es ist durchaus schwierig, Programme mit solchen Parallelprozessen zu entwickeln.
- Aber Parallelprozesse scheinen einfach das richtige „Denkmodell“ für solche Programme wie WWW–Browser zu sein.
- *Mit anderen Worten*:

  Ohne Parallelprozesse wäre ein solches Programm wahrscheinlich noch sehr viel schwieriger zu entwickeln.

Absicherung von Prozessen I:

Gegen fehlerhaftes Ablaufen anderer Prozesse:

- Das Betriebssystem teilt jedem Prozess eine eigene „Spielwiese“ im Speicher zu.
- Manche Programmiersprachen (z.B. C und C++) erlauben den direkten Zugriff auf Speicheradressen.

  $\longrightarrow$ Prozesse können also aus Versehen Adressen in fremden Spielwiesen ansprechen.

  $\longrightarrow$ Ein sehr häufiger Programmierfehler in Sprachen wie C und C++!

- Das Betriebssystem führt einen solchen Zugriff nicht aus, sondern schickt jedesmal statt dessen Signal „*SEGV*“ (*Segmentation Violation*) an den Prozess.
- Default–Reaktion: Beendigung („Absturz“) des fälschlicherweise zugreifenden Prozesses.

Absicherung von Prozessen II:

Gegen unbefugte Kontaktaufnahme anderer Prozesse:

- Signale werden durch das Betriebssystem in der Regel nur dann von einem Prozess an einen anderen wirklich weitergeleitet, wenn beide Prozesse demselben User (siehe Folie 108) gehören.
- Bei einer Pipe, die mit „|“ auf der Shell eingerichtet wurde, stellt sich die Frage der Befugnis gar nicht erst: Beide Prozesse gehören demselben User.
- Ein Socket zwischen zwei Prozessen
  - ◇ kann zwar zwischen zwei Prozessen verschiedener User aufgemacht werden

    (so dass sich die Frage der Befugnis durchaus stellt),
  - ◇ kommt aber nur zustande, wenn die beiden zugrundeliegenden Programme so implementiert sind, dass sie beide der Öffnung eines Sockets zwischen ihnen „zustimmen“.

Aufgabengebiet 5: Window–System

Zeichen- und graphikorientiert:

- Früher gab es nur zeichenorientierte Bildschirme und nur die Tastatur als interaktives Eingabegerät.
- Heutzutage sind Bildschirme graphikorientiert, und die Maus steht als zusätzliches Eingabegerät zur Verfügung.
- Neben der zeichenorientierten Ein-/Ausgabe über eine Shell (`xterm`–Fenster) kann ein Prozess auch eigene *Windows* (*Fenster*) zur Interaktion öffnen.

Window–System:

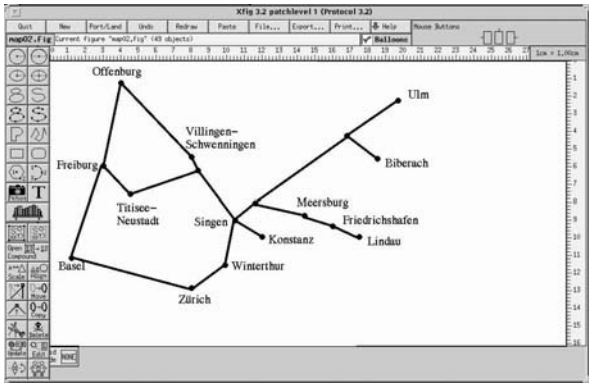
- Der Teil des Betriebssystems, der primär für diese neuen Möglichkeiten verantwortlich ist.
- Aus historischen Gründen ist das bei UNIX eine separate Einheit: das *X11 Window System*.

Window–orientiertes Display:

- Ähnlich wie bei Directories (Folie 91) sind die Windows, die momentan auf dem Bildschirm sind, hierarchisch organisiert.
- Der Hintergrund des Bildschirms ist das oberste Window in der Hierarchie (*Root–Window*).
- Auf der nächsten Hierarchiestufe kommen die *Top–Level–Windows*:
  - ◇ Die Windows, die eine separate graphische Einheit bilden, die als Ganzes verschoben, vergrößert/verkleinert, iconifiziert etc. werden kann.
  - ◇ *Visuelles Kennzeichen*: Das sind genau die Windows mit den charakteristischen Rahmenelementen zum Verschieben, Vergrößern/Verkleinern, Iconifizieren etc.

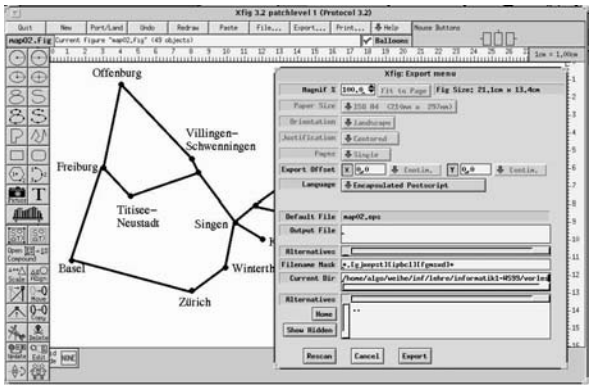
Die weiteren Windows:

- Sind in Top–Level–Windows als kleinere, abhängige Bestandteile integriert.
- *Beispiele*: Buttons, Menüs, Textfelder...
- Werden im dahinterstehenden Prozess als eigene Unterprozesse verwaltet.



Beachte:

- Ein zusätzliches Fenster wie hier das Export–Menü beim Zeichenprogramm „`xfig`“ ist zwar vom selben Programm aufgemacht worden (in einem weiteren Prozess),
- ist aber trotzdem ein separates Top–Level–Window (grauer Rahmen!).



Fenster auf Bildschirm anzeigen:

- Für den Inhalt eines Windows ist der Prozess verantwortlich, der das Window geöffnet hat.
- Das Betriebssystem
  - ◇ verwaltet eine Reihung der Windows und
  - ◇ berechnet für jedes Window, welche seiner Regionen sich mit Windows höherer Reihungsnummer überlappen.

  ⟶ Diese Regionen des Windows werden nicht angezeigt.

  ⟶ Die Präsentation der Windows erweckt wie beabsichtigt den visuellen Eindruck, dass die Windows „ausgeschnittene Papierrechtecke“ sind, die gemäß Reihung übereinandergelegt wurden.

Benutzereingaben zuordnen:

- Zu jedem Zeitpunkt ist ein Bildschirmpunkt ausgezeichnet: der *Mouse Pointer* (*Mauszeiger*).
- Kennzeichnung auf dem Bildschirm durch ein Icon (meist ein Pfeil).
- Verschieben des Mouse Pointers auf dem Bildschirm: durch Verschieben der Maus.
- Der Mouse Pointer definiert das *aktive Window*: das Window mit höchster Reihungsnummer unter allen Windows, die diesen Bildschirmpunkt enthalten.

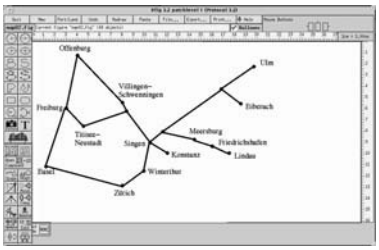
  ⟶ Also das Window, das um den Mouse Pointer herum auf dem Bildschirm gezeigt wird.
- Der Prozess, zu dem das momentan aktive Window gehört, erhält alle Benutzereingaben (Tastatureingaben und Mausklicks).
- Dieser Prozess legt auch fest, durch welches Icon der Mouse Pointer dargestellt wird.

Window Manager:

- Ein weiterer *Daemon* (siehe Folie 115), der solange läuft, wie das Window System insgesamt läuft.
- Es kann immer nur ein Window–Manager–Prozess pro Einheit Bildschirm+Tastatur+Maus laufen.
- Vom Window Manager verwaltete Windows:
  - ◇ das Hintergrund–Window,
  - ◇ Hintergrund–Menüs,
  - ◇ die einzelnen Windows, aus denen sich der Rahmen jedes Top–Level–Windows zusammensetzt.

    ⟶ Siehe nächste Folie.

Rahmen eines Top–Level-Windows:



Erläuterungen:

- Damit ist der schmale graue Rahmen einschließlich grauer Titelzeile gemeint.
- Dieser Rahmen setzt sich aus mehreren rechteckigen Windows zusammen:
  - ◇ die vier Seitenkanten des großen Windows,
  - ◇ die vier Ecken,
  - ◇ die Titelzeile,
  - ◇ die Icons in der Titelzeile.
- In jedem dieser kleinen Windows läuft ein eigener Unterprozess des Window Managers.

Operationen an diesem Rahmen:

- Benutzeraktionen mit der Maus zur Verschiebung, Größenveränderung, Iconifizierung etc. von Top–Level–Windows werden vom Window Manager verarbeitet.
- Bei der Größenveränderung eines Windows schickt der Window Manager das Signal `WINCH` (*Window Change*) an den in diesem Fenster laufenden Prozess.
- Der Prozess, dem dieses Fenster gehört, kann dann sofort den Fensterinhalt an die neue Größe des Fensters anpassen.

Beispiel:
Größenveränderung des `nedit`–Fensters

- Der `nedit`–Prozess wird durch Signal WINCH in Kenntnis davon gesetzt, dass „irgendetwas“ mit seinem Fenster passiert ist.
- Daraufhin fragt er die neuen Koordinaten des Fensters ab und berechnet damit einen neuen Fensterinhalt.
- Diesen neuen Inhalt lässt er dann ins Fenster zeichnen.
- Der Window Manager sorgt dann dafür, dass der neue Inhalt nur in den momentan sichtbaren Teilen des Fensters wirklich gezeichnet wird.

Virtuelle Window Manager

Einige neuere Window–Manager–Programme bieten auch die Möglichkeit eines *virtuellen Bildschirms*:

- Das Hintergrund–Window ist wesentlich größer als der Bildschirm selbst.
- Es wird immer nur ein Ausschnitt der gesamten Szenerie auf dem Bildschirm gezeigt.
- Durch Verschiebung der Maus, Drücken von Funktionstasten o.ä. wird der Ausschnitt, der auf dem Bildschirm gezeigt wird, über dem Hintergrund–Window verschoben.

Aufgabengebiet 6: Virtualisierte Ressourcen

Erinnerung:

- Folie 128: Jeder Prozess hat seine eigene Spielwiese.

  $\longrightarrow$ Gemeinsame Ressourcen aller Prozesse (insb. der Computerspeicher) dürfen dem Prozess nicht in unmittelbarer, sondern nur in irgendwie „virtueller“ Form zugänglich gemacht werden.
- Folie 140 führte ein weiteres Beispiel einer virtualisierten Ressource ein: den virtuellen Bildschirm.

Allgemein:

Wir reden von virtualisierten Ressourcen, wenn das Betriebssystem

- den Zugriff von Prozessen auf eine Ressource strikt kontrolliert und
- diese Ressource den Prozessen durch „Wegabstraktion“ technischer Details in einer einfacheren und/oder idealisierten Form präsentiert.

Beispiel: Speicherplatz

Problem: Die Adressierung einer Information im Speicher ist ziemlich kompliziert (vgl. Folie 74 ff.):

- Die Adresse auf der Festplatte wird durch Spurnummer, Offset in der Spur und ähnliche technische Details definiert.
- Zudem kommt noch der Fall hinzu, dass die Information schon irgendwo im Hauptspeicher (oder gar im Cache) ist und daher von dort anstatt von der Festplatte gelesen wird.
- Heutzutage werden die Daten häufig auch verteilt im Netzwerk gehalten.

Frage: Was heißt hier nun Virtualisierung?

$\longrightarrow$ Siehe nächste Folie.

Virtuelle Sicht auf den Speicher:

- Prozesse „sehen“ nur einen linearen, virtuellen Adressraum $[0...2^n - 1]$ wie im Von–Neumann–Modell (Folie 63), in dem jedem Maschinenwort eine einzelne, unveränderliche Zahl als Adresse zugeordnet ist.
- Bei jedem Speicherzugriff wird diese virtuelle Adresse „unsichtbar“ auf die zugrundeliegende reale Adressierung umgerechnet.
- Das alles wird durch das Betriebssystem selbst geleistet, ohne dass die Prozesse davon irgendwie betroffen wären.

Beispiel: Ausgabe von Ton

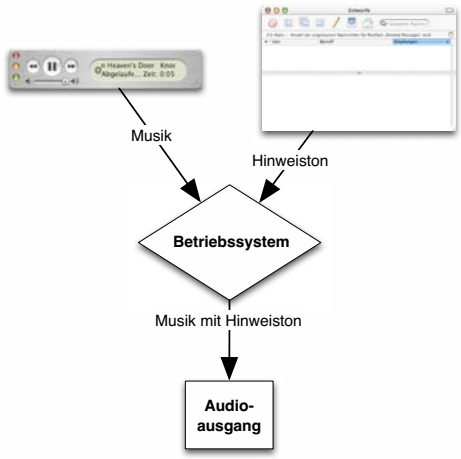
Problem:

- Bei normaler Nutzung eines Computers kommt es vor, dass mehr als ein Programm gleichzeitig Ton ausgeben will und soll.
- Beispielsweise soll das E-Mail Programm akustisch auf neue Mail hinweisen können, während man Musik hört.
- Gleichzeitig verfügen die meisten Computer nur über einen Audioausgang.

Lösung:

- Jedes Programm erhält seinen eigenen virtuellen Audioausgang mit Lautstärkeregler usw.
- Das Betriebssystem nimmt die Daten dieser virtuellen Audioausgänge und mischt sie anhand Ihrer individuellen Lautstärken.
- Das dabei entstandene Signal wird auf dem realen Audioausgang ausgegeben.

Beispiel als Bild:



Abschließende Bemerkung zu virtualisierten Ressourcen:

- Es gibt auch Programme, die die Dienste und Virtualisierungsmechanismen des Betriebssystems umgehen und den direkten Speicherzugriff aus Effizienzgründen selbst organisieren.
- *Wichtigstes Beispiel*: Datenbanksysteme.

  $\longrightarrow$ Siehe Veranstaltungen zu „Datenbankmanagementsystemen“.
- Solche Programme sind natürlich nicht einfach so aufrufbar, sondern müssen auf besondere Art (am Betriebssystem vorbei) gestartet werden.
- Insbesondere wird dabei die Möglichkeit genutzt, dass die Festplatte in *Partitionen* zerlegt werden kann, die potentiell durch eigenständige Systemprozesse verwaltet werden können.

  $\longrightarrow$ Wieder „Spielwiese“, aber nun auf einem tieferen Abstraktionsniveau, unmittelbar auf der Speicherhardware.

Abschnitt 1.5: Rechnernetze

- Aus Sicht eines Computers (bzw. seines Betriebssystems) ist ein Kommunikationsnetzwerk nichts anderes als eine andere Art von Ein-/Ausgabegerät.

  (Am ehesten vergleichbar einem Modem, also einer Ein-/Ausgabeeinheit für Telefonübertragungen).
- Allerdings ist die Kommunikation wesentlich komplexer.

  $\longrightarrow$ *Kommunikationsprotokolle* zur unmissverständlichen Verständigung zwischen Computern sind notwendig.
- Die Abgrenzung, welche beteiligten Dienste noch zum Betriebssystem gehören und welche Dienste eigenständig sind, ist willkürlich.

  $\longrightarrow$ Wir machen hier der Einfachheit halber keinen Unterschied, d.h. zählen alles mehr oder weniger zum Thema Betriebssysteme im weitesten Sinne.

Netzwerke und Kommunikationsprotokolle:

- Jeder Computer (*Netzwerk–Knoten*) in einem Netzwerk hat eine *Netzwerkadresse*, die im gesamten Netzwerk nur einmal vergeben ist.
- Eine Botschaft (z.B. Email), die von einem Computer zu einem anderen durch ein Netzwerk hindurch verschickt werden soll, wird üblicherweise in kleine *Pakete* (*Datagramme*) zerlegt, die einzeln verschickt werden.
- An jedes Paket wird vorne und/oder hinten weitere Zusatzinformation angehängt.

  $\longrightarrow$ Eine Art „Briefumschlag“ für die eigentlich zu übertragenden Daten.
- Minimal notwendige Zusatzinformation:
  - ◇ Netzwerkadresse des Zielknotens,
  - ◇ Identifikation des Prozesses, der das Datenpaket auf dem Zielknoten in Empfang nehmen und weiterbehandeln soll.
- Ein Regelwerk für die Formatierung von Datenpaketen nebst Zusatzinformationen bezeichnet man als *Kommunikationsprotokoll* (oder kurz *Protokoll*).

Datenübertragung:

- Auf jedem Computer in einem Netzwerk läuft ein Server (vgl. Folie 115), der nach diesem Protokoll Datenpakete von anderen Netzwerk–Knoten in Empfang nimmt bzw. an andere Netzwerk–Knoten weiterreicht.

  $\longrightarrow$ Im folgenden $<$*Protokoll*$>$*–Server* genannt (Ethernet–Server, IP–Server, TCP–Server...).
- Ein Prozess kann diesem Server ein Datenpaket zur Weiterleitung durch das Netzwerk schicken.

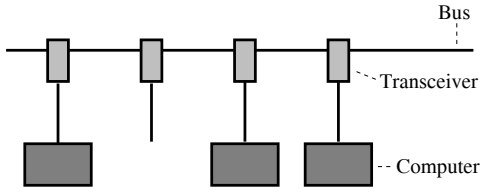
  $\longrightarrow$ Das Datenpaket wird vom Server gemäß Kommunikationsprotokoll in den „Briefumschlag“ gesteckt und losgeschickt.
- Wird ein Datenpaket von diesem Server aus dem Netzwerk in Empfang genommen, dann
  - ◇ wird der „Briefumschlag“ des Kommunikationsprotokolls entfernt und
  - ◇ der eigentliche Inhalt wird an den spezifizierten Prozess weitergereicht.

Intranet/Local Area Network (LAN):

LANs sind oft so realisiert, dass mehrere oder sogar alle darin untereinander vernetzten Computer am selben „Draht“ hängen.

Weit verbreitete Lösung als Beispiel: Ethernet

- Alle Computer hängen über *Transceiver* an einem einzigen Draht, dem *Bus*.



- Die Datenübertragung über den Bus läuft durch die Transceiver ungestört hindurch.

  $\longrightarrow$ Computer können in Transceiver ein- und ausgeklinkt werden (bzw. sogar ausfallen), ohne dass der Datenfluss zwischen anderen Computern dadurch irgendwie berührt wird.

Fortsetzung Ethernet:

- Auf jedem Computer hört der Ethernet–Server den gemeinsamen Draht nach Datenpaketen ab und verarbeitet die Pakete, die an seine eigene Netzwerkadresse (*Ethernet–Adresse*) adressiert sind.
- Wenn der Ethernet–Server ein Datenpaket verschicken soll, horcht er den Draht ab und wartet auf den Moment, wenn die momentan laufende Datenübertragung beendet ist.
- In diesem Moment beginnt er sofort selbst mit der Datenübertragung.

Problem: Was passiert eigentlich, wenn zwei Ethernet–Server

- gleichzeitig darauf warten, dass der Draht wieder frei wird und
- mehr oder weniger gleichzeitig mit der Datenübertragung beginnen?

$\longrightarrow$ Antwort auf der nächsten Folie.

Antwort:

- Falls zwei oder mehr Ethernet–Server mehr oder weniger im selben Moment zu senden beginnen, erkennt jeder von ihnen das sofort, weil er natürlich weiterhin den Draht abhört.
- Jeder der betroffenen Ethernet–Server
  - ◇ bricht daraufhin seinen Übertragungsversuch sofort ab und
  - ◇ wartet eine zufällig bestimmte Zeitspanne, bis er von vorne beginnt, d.h. wieder auf die Beendigung der momentanen Datenübertragung wartet, um daraufhin selbst loszusenden.

Sinn des Ganzen:

- Durch den (bewusst eingesetzten) Faktor Zufall kommt es nur in extrem ungünstigen Fällen zu einer erneuten Kollision derselben Datenpakete.
- Falls die Übertragung eines Datenpakets dennoch zu häufig wegen Kollisionen mit Übertragungsversuchen anderer Ethernet–Server abgebrochen werden musste,
  - ◇ bricht der Ethernet–Server den Übertragungsversuch ganz ab und
  - ◇ schickt eine Fehlermeldung an den sendenden Prozess (z.B. Mail–Server) zurück.
- Dieser Prozess wird dann typischerweise
  - ◇ das Paket sofort noch einmal versuchen zu verschicken oder
  - ◇ zu einem späteren Zeitpunkt noch einmal oder
  - ◇ dem sendenden Endbenutzer Bescheid geben oder
  - ◇ den Versuch der Verschickung ganz aufgeben.

Internet (*sehr* stark vereinfacht)

- Das Internet besteht konzeptionell aus Sendekanälen jeweils von einem einzelnen Knoten zu einem anderen einzelnen Knoten.
- Ein Datenpaket wird von Computer zu Computer weitergereicht.
- Jeder Internet–Knoten X hält dazu eine Tabelle (*Routing–Tabelle*) mit Einträgen der Art

  Intervall von Internet–Adressen

  $\longleftrightarrow$

  Nächster zuständiger Internet–Knoten

- Darin kommen nur Internet–Knoten vor, zu denen es einen direkten Sendekanal von X aus gibt.

Auswertung der Routing–Tabelle:

Wenn ein Internet–Knoten ein Datenpaket verschicken soll,

- entweder weil ein Prozess auf diesem Computer selbst das Datenpaket verschicken will
- oder weil das Datenpaket von einem anderen Internet–Knoten an diesen weitergereicht wurde,

dann

- wird die erste Zeile der Routing–Tabelle, bei dem die Internet–Adresse des Empfängers im Intervall enthalten ist, herausgesucht, und
- das Datenpaket wird an den in dieser Zeile eingetragenen Internet–Knoten auf dem direkten Sendekanal weitergeschickt.

Routing–Tabelle:

- *Vollständigkeit*: Damit jedes Datenpaket behandelt werden kann, müssen alle Internet–Adressen durch die Vereinigung dieser Intervalle überdeckt sein.
- *Redundanz*: In der Regel ist jede Internet–Adresse in mehr als einem Intervall in der Routing–Tabelle enthalten.

  $\longrightarrow$ Wenn die Verbindung über den ersten zuständigen Knoten nicht zustande kommt, kann es mit der zweiten Möglichkeit versucht werden, dann mit der dritten usw.

Organisation des Internets:

- Die Wege der Datenübertragung vom Sender zum Empfänger
  - ◇ werden nicht durch zentrale Instanzen festgelegt,
  - ◇ sondern durch das oben beschriebene lokale „Durchwursteln auf gut Glück“ von Knoten zu Knoten.
- *Konsequenz*: Totalausfälle sind recht unwahrscheinlich, da keine (verwundbaren) Zentralinstanzen nötig sind.
- *Allgemeine Erfahrung*:
  - ◇ Dieses „systematische Chaos“ funktioniert auch im Normalbetrieb erstaunlich gut,
  - ◇ d.h. vermeidet „Verkehrsstaus“ recht gut bzw. löst sie schnell wieder auf.

Angeblicher Hintergrund:

Das Internet ist ursprünglich auf das (militärisch motivierte) Ziel hin entworfen worden, dass möglichst viele Botschaften auch beim Zusammenbruch sehr weiter Netzwerkteile noch ihr Ziel erreichen („atomkriegsicher“).

$\longrightarrow$ Ist wohl eher ein Märchen,

Basisprotokoll des Internets: IP (Internet Protocol)

- Die einzelnen direkten Sendekanäle von Internet–Knoten zu Internet–Knoten sind typischerweise auf einem LAN–Protokoll (z.B. Ethernet) basiert.
- Die verschiedenen direkten Kanäle, zu denen ein Computer als Direktsender oder Direktempfänger gehört, sind in der Regel völlig separate LANs.

Problem:

- Das LAN–Protokoll kann nur die LAN–Adresse des allernächsten Zwischenknotens innerhalb des LANs als Zusatzinformation aufnehmen.
- Die Internet–Adresse des letztendlichen Ziels des Datenpakets muss auch irgendwie gespeichert werden.

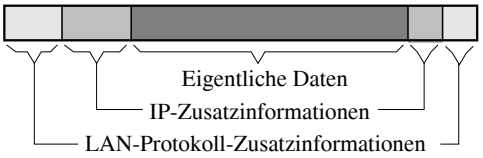
Lösung:

- In den „Briefumschlag“ des LAN–Protokolls wird ein zweiter „Briefumschlag“ gesteckt, der (unter anderem) die Internet–Zieladresse als Zusatzinformation enthält.
- Standard für das Internet: das sogenannte *IP* (*Internet Protocol*)

  $\longrightarrow$ Meist entgegen der Sprachlogik *IP–Protokoll* genannt.
- Die Zusatzinformation im IP–Protokoll enthält u.a. die Internet–Adressen von Sender und Empfänger.
- Datenpakete, die über das Internet geschickt werden sollen, werden zunächst einmal vom IP–Server behandelt.
- Zum Verschicken reicht der IP–Server ein Datenpaket an den Server des LAN–Protokolls weiter mit der Identifikation des IP–Servers als empfangender Prozess auf dem Empfänger im LAN.

Bemerkung: Internet–Adressen werden nach dem IP–Protokoll meist *IP–Adressen* genannt.

Schematische Veranschaulichung:

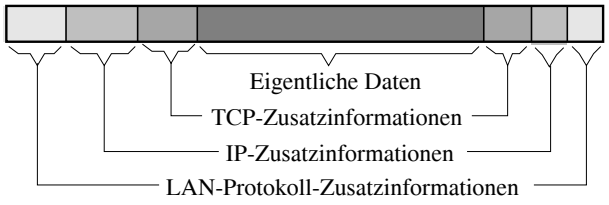


Problem mit IP:

- Ein Datenpaket wird einfach vom IP–Server (via LAN–Protokoll–Server) losgeschickt — und dann vergessen!
- Ob alle Datenpakete einer Botschaft planmäßig angekommen sind oder ob irgendwelche Datenpakete irgendwo hängengeblieben sind, lässt sich nicht mehr nachvollziehen.
- Die Datenpakete können auch auf unterschiedlichen Wegen durch das Internet gelaufen sein und in der falschen Reihenfolge ankommen.
- Datenpakete können durch Übertragungsfehler auch verfälscht ankommen.

Lösungansatz unter TCP:

- Das Datenpaket wird zusätzlich in einen dritten Briefumschlag im Innern der ersten beiden Briefumschläge gesteckt.
- Standard für die meisten Internetdienste: *TCP* (*Transmission Control Protocol*).



Zusatzinformationen des TCP–Protokolls u.a.:

- Eine laufende Nummer des Datenpakets innerhalb der Gesamtbotschaft, die bei der Zerlegung der Gesamtbotschaft in einzelne Datenpakete vergeben wird.
- *Port–Nummern* des Ausgangs- und Zielprozesses.
- *Checksumme*.

Port–Nummern:

- *Erinnerung* an Folie 149: Der Prozess auf dem Emfänger, der die Botschaft eigentlich empfangen soll, muss natürlich spezifiziert sein.
- Dafür ist im Internet das TCP–Protokoll zuständig.
- *Genauer*: Jeder solche Prozess muss eine numerische Kennung haben, die *Port–Nummer*.
- Überall verbreitete Standarddienste wie Email, Telnet, Ftp, Http (=WWW–Server) haben standardisierte Port-Nummern.
- *Konkretes Beispiel*: Http hat Port–Nummer 80.

Exkurs: Uniform Resource Locations (URLs):

- Über das Internet lässt sich jede Informationsressource jedweder Art durch Adressen in einem allgemeinen Format ansprechen.
- Stichwort in der Literatur: *Uniform Resource Location* (*URL*).
- *Beispiel*: Die Übersichtsseite zu Neuigkeiten am Fachbereich Informatik der TU Darmstadt hat die URL

`http://www.informatik.tu-darmstadt.de:80/5_0Neuigkeiten/index.html`

Fortsetzung URLs:

- *Allgemeine Grobstruktur von URLs*:
  - ◇ Zuerst das Protokoll („`http`“),
  - ◇ nach „`://`“ dann der Rechnername („`www.informatik.tu-darmstadt.de`“),
  - ◇ optional dann eine Portnummer gemäß Folie 164, mit Doppelpunkt abgetrennt („`:80`“),
  - ◇ nach einem “`/`“ schließlich die Adresse der Ressource auf diesem Rechner („`5_0Neuigkeiten/index.html`“).
- Speziell „`http`“ ist das gängige Standardprotokoll für WWW-Ressourcen (z.B. HTML–Seiten).

Beispiele für URLs:

`http://www.allgemeineinformatik.de/index.php` Dies bezeichnet die Homepage dieser Lehrveranstaltung. Der Port wurde an dieser Stelle weggelassen, das Hypertext Transfer Protocol (HTTP) ist standardmässig auf Port 80 aktiv.

`ftp://ftp.tu-darmstadt.de/pub/liesmich.txt` Dies verweist auf die Datei `liesmich.txt` auf dem FTP-Server der TU. FTP steht f¨ur File Transfer Protocol. Der Port kann auch hier weggelassen werden, da der TU-Server unter dem Standard-Port f¨ur FTP, 21, zu erreichen ist.

`ipp://printers.tu-darmstadt.de:631/printers/drucker1` Diese URL beschreibt einen Drucker, der ¨uber das Internet Printing Protocol (ipp) angesprochen wird. An dieser Stelle wurde der an sich optionale Port angegeben.

... Exkurs Ende

Checksumme:

- Die Berechnung von Checksummen ist eine Klasse von mathematischen Verfahren zur Aufdeckung von Übertragungsfehlern.
- Solche Verfahren basieren auf der Interpretation des Datenpakets als eine Sequenz von Binärzahlen fester Länge (vgl. Folie 15).
- Mit überwältigender Wahrscheinlichkeit hat ein durch Übertragungsfehler verfälschtes Datenpaket eine andere Checksumme.

  $\longrightarrow$ Wenn der Empfänger die Checksumme des empfangenen Datenpakets berechnet und mit der mitgelieferten Checksumme vergleicht, werden Übertragungsfehler mit sehr großer Sicherheit durch Nichtübereinstimmung der beiden Checksummen entdeckt.
- Zur Erläuterung betrachten wir beispielhaft das konkrete TCP–Verfahren.

Checksummenberechnung in TCP:

- Das Datenpaket wird als eine Sequenz

  $$b_1, b_2, \ldots, b_n$$

  von n 16–Bit– Binärzahlen interpretiert.

  $\longrightarrow$ Ggf. das Datenpaket mit bis zu 15 Nullen aufgefüllt, wenn die Anzahl Bits im Datenpaket kein Vielfaches von 16 ist.

- *Notation*: Für $i = 1, 2, \ldots, 16$ sei $b_j[i]$ die i–te Stelle von b_j.

- Die Checksumme ist ebenfalls eine 16–Bit–Binärzahl.

- *Genauer*: Die i–te Stelle der Checksumme ist

  =0 falls $b_1[i] + b_2[i] + \cdots + b_n[i]$ eine gerade Zahl ist,

  =1 falls $b_1[i] + b_2[i] + \cdots + b_n[i]$ eine ungerade Zahl ist.

(Un-)Sicherheit dieses Verfahrens:

- Offensichtlich kann es immer noch fehlerhaft übertragene Bits geben, die der Empfänger durch Vergleich von empfangener und selbst berechneter Checksumme nicht findet.
- Da sich je zwei fehlerhafte Bits an einer der 16 Stellen bei der Checksumme gegenseitig wegheben, passiert das genau dann, wenn an jeder der sechzehn Stellen eine gerade Anzahl von Fehlern im Datenpaket (einschließlich Checksumme) auftreten.

  $\longrightarrow$ Insbesondere mindestens zwei Fehler.
- Dass genau zwei Fehler auftreten — und das auch noch an derselben der 16 Stellen — ist schon extrem unwahrscheinlich:

  Wenn zwei fehlerhafte Bits so speziell verteilt auftreten, ist die Wahrscheinlichkeit sehr hoch, dass noch weitere Fehler im Datenpaket sind.

Fortsetzung (Un-)Sicherheit:

- Bei mehr als zwei fehlerhaften Bits ist die Wahrscheinlichkeit aber astronomisch gering, dass sie so ungewöhnlich verteilt sind, dass der Vergleich der Checksummen keinen Fehler findet.
- Daher kann man die verbleibende Restunsicherheit von Checksummen in der Praxis unbeschadet ignorieren.

Allerdings:

- Diese Überlegungen setzen voraus, dass fehlerhafte Bits durch zufälliges „Hintergrundrauschen“ entstehen.
- Zum Beispiel bei periodisch wirkenden Störquellen gelten diese Überlegungen nicht mehr so ganz.

  $\longrightarrow$ Obwohl es auch in diesem Fall schon mit dem Teufel zugehen müsste...
- Und bei gezielter, absichtlicher Störung der Übertragung kann man sowieso nichts mehr aussagen.

Botschaften verschicken mit TCP/IP:

- Vor dem Versenden der Datenpakete „sprechen“ sich der sendende und der empfangende TCP–Server mit Hilfe dreier Datenpakete ab:
  - ◇ Der Sender schickt ein TCP–Datenpaket zur Eröffnung einer Verbindung (notfalls wiederholt).
  - ◇ Falls der Empfänger bereit ist, schickt er eine positive Antwort zurück.
  - ◇ Der Sender schickt eine Empfangsbestätigung an den Zielknoten.
- Sobald der Sender alle Empfangsbestätigungen hat, leitet er den Verbindungsabbau analog zum -aufbau durch den Austausch von drei Datenpaketen ein.

Stichworte in der Literatur:

- Protokolle, die wie TCP (im Gegensatz zu Ethernet und IP) vorsehen, dass der Empfänger auch Nachrichten an den Sender schickt, heißen *Handshake Protokolle*.
- Der obige Austausch dreier Datenpakete ist das Kennzeichen von *Three–Way Handshake Protokollen*.

Zuverlässigkeit der Datenübertragung:

- Für jedes korrekt empfangene Datenpaket (d.h. auch die Checksumme muss stimmen) schickt der Empfänger eine Empfangsbestätigung mit der laufenden Nummer des Datenpakets an den Sender zurück.
- Falls der Sender diese Bestätigung nach einer gewissen Zeit immer noch nicht erhalten hat, schickt er das Datenpaket einfach noch einmal.
- Das tut er eine gewisse Anzahl von Malen, gibt dann schließlich auf und sendet statt dessen eine Fehlermeldung an den Prozess, von dem die zu versendenden Daten stammen.
- Wenn die Übertragung geklappt hat, leitet der empfangende TCP–Server die Datenpakete in der Reihenfolge ihrer laufenden Nummern (Folie 164) an den eigentlichen Adressaten weiter.

Fortsetzung Zuverlässigkeit:

- Wenn die Datenpakete nicht in dieser Reihenfolge beim empfangenden TCP–Server eintreffen, werden verfrühte Datenpakete von diesem TCP–Server zurückgehalten, bis alle Datenpakete mit kleinerer laufender Nummer eingetroffen sind.
- Bleiben die Daten nach einer gewissen Wartezeit unvollständig, ist die Übertragung gescheitert, und die bisher gesendeten Datenpakete werden vom Empfänger wieder „vergessen“.

Anmerkung:

- Solche Checksummen werden sehr häufig angewandt.
- *Beispiel*: Die letzte Ziffer der ISBN–Nummer eines Buches ist eine solche Checksumme.

Domain Name System (DNS):

- Numerische IP–Adressen sind natürlich alles andere als benutzerfreundlich.
- Zur Verbesserung der Benutzerfreundlichkeit ist das *Domain Name System* (*DNS*) entwickelt worden.
- Spezielle Internet–Knoten (*DNS–Server*) speichern Tabellen, mit denen DNS–Adressen in IP–Adressen übersetzt werden können.

  $\longrightarrow$ Die anderen Internet–Knoten müssen nur die Information halten, welchen DNS–Server sie für welche Adresse fragen können.
- Das DNS ist hierarchisch in *Top–Level–Domains*, *Subdomains*, *Subsubdomains* etc. gegliedert.

Konkretes Beispiel

`ultra10.rbg.informatik.tu-darmstadt.de`

Erläuterungen:

- DNS–Top–Level–Domain „`de`“ für Deutschland,
- DNS–Subdomain „`tu-darmstadt`“,
- DNS–Subsubdomain „`informatik`“ für den Fachbereich Informatik,
- DNS–Subsubsubdomain „`rbg`“ für die Rechnerbetriebsgruppe am Fachbereich Informatik,
- Computer mit DNS–Namen „`ultra10`“ in dieser Subsubsubdomain.

Beispiele für Top–Level–Domains:

- *Länder*:
  - ◇ „de“: Bundesrepublik Deutschland
  - ◇ „at“: Österreich (Austria)
  - ◇ „au“: Australien
  - ◇ „ch“: Schweiz (Confoederatio Helvetica)
  - ◇ „fr“: Frankreich
  - ◇ „uk“: Großbritannien (United Kingdom)
- *Sparten* (in erster Linie in den USA):
  - ◇ „com“: Firmen (commercial)
  - ◇ „edu“: Schulen und Hochschulen (education)
  - ◇ „gov“: US–Behörden (government)
  - ◇ „net“: Zentrale Netzadressen
  - ◇ „mil“: US–Militär
  - ◇ „org“: Nichtstaatliche Organisationen (z.B. wissenschaftliche)

Abschließende Bemerkung:

- TCP/IP/Ethernet ist ein Beispiel für eine generelle Strategie zur Entwicklung von Dienstprogrammen:
  - ◇ Die Gesamtaufgabe wird gedanklich in verschiedene Abstraktionsebenen (*Layer*) zerlegt.
  - ◇ Jede Abstraktionsebene wird durch einen möglichst autonom agierenden Teildienst erledigt.
- *Vorteile*:
  - ◇ Die Implementation einer Anzahl separater Teildienste ist meist wesentlich einfacher als die Implementation eines monolithischen Dienstes.
  - ◇ Für die einzelnen Teildienste kann es mehrere Alternativen geben, die ohne Effekt auf die anderen Teildienste ausgetauscht werden können.
- *Nachteil*: Erhöhte Laufzeit durch erhöhten Kommunikationsaufwand.

  $\longrightarrow$ Kann die Antwortzeiten spürbar erhöhen.

Abschnitt 2: Grundlegendes zur Programmierung mit Java

Warum eigentlich Java?

- Java ist nicht gerade als Lernsprache entwickelt worden.
- Man kann auch nicht gerade behaupten, dass Java als erste Programmiersprache für Neulinge gut geeignet ist.

$\longrightarrow$ **Warum also Java???**

Warum Java:

- Java ist heute und in weiterer Zukunft die wichtigste Programmiersprache wegen ihrer Möglichkeiten in Internet–Programmierung, Datenbankanbindung usw.
- Das weitere Lehrangebot des Fachbereichs Informatik, das man im Anschluss an die *Allgemeine Informatik I/II* belegen kann, baut weitgehend auf Java auf.
- Java bietet eigentlich alles an Konzepten und Schwierigkeiten, was man auch in anderen gängigen Programmiersprachen findet.

  $\longrightarrow$ Wer Java systematisch gelernt hat, kommt mit anderen Programmiersprachen besser klar.

Zur Vorgeschichte von Java:

- Java lehnt sich in den Äußerlichkeiten stark an die weit verbreiteten Programmiersprachen C und C++ an.
- *Hauptgrund*: C/C++–Programmierern soll der Umstieg erleichtert werden.

  $\longrightarrow$ Höhere Akzeptanz für Java in der Wirtschaft.
- *Unerwünschter Nebeneffekt*: Java erbt von C/C++ viele „Altlasten“.
- *Beispiel*:
  - ◇ Zuweisung mit „=“:

    ```
    i = 1;
    ```
  - ◇ Test auf Gleichheit mit „==“:

    ```
    if ( i == 1 )
    ```


Eigentlich intuitiver
(und auch eher üblich in Programmiersprachen):

- Zuweisung mit „:=“:

  ```
  i := 1;
  ```
- Test auf Gleichheit mit „=“:

  ```
  if ( i = 1 )
  ```

Hintergrund:

- C stammt aus dem Jahr 1970.
- Die Länge von Programmen war 1970 noch ein ernsthaftes Speicherplatzproblem.
- Die Zuweisung kommt in Programmen im allgemeinen öfter vor als der Test auf Gleichheit.
- Entscheidung 1970: lieber bei der Zuweisung ein Zeichen einsparen!

  $\longrightarrow$ Diese „prähistorische“ Entscheidung hat sich über C++ (entwickelt in den 80ern) bis nach Java (entwickelt in den frühen 90ern) vererbt.

Allgemeine Java–Philosophie:

- Hinter jeder Programmiersprache steht so etwas wie eine „Philosophie“.
- Zum Beispiel hinter C und C++ steckt die Philosophie:

  Der Programmierer soll durch die Programmiersprache maximale Freiheit bekommen.

  $\longrightarrow$ Alles, was nicht ausdrücklich verboten ist, ist erlaubt, egal, ob es Sinn macht oder nicht.
- Damit einhergehend wird dem Programmierer natürlich auch maximale Selbstverantwortung aufgebürdet.
- In der Philosophie lehnt sich Java **nicht** an C und C++ an:

  Beim Entwurf von Java ist man davon ausgegangen, dass diese Selbstverantwortung für die meisten Programmierer zu hoch ist.

Also Grundregel zum Verständnis von Java:

> Alles, was aller Erfahrung nach keinen Sinn macht, ist von vornherein verboten.

⟶ Es ist müßig darüber zu streiten, welche Philosophie nun „die Bessere“ ist...

Letzte Vorbemerkung:

- Die folgende Darstellung hält sich in Details und Beispielen strikt an Java.
- Trotzdem sind die wesentlichen Ideen, Konzepte und Prinzipien allgemeiner und zumindest für die gängigen Programmiersprachen im Großen und Ganzen ebenfalls gültig.

Abschnitt 2.1: Compiler und Interpreter

Arbeitsgang:

- Der Programmierer erstellt ein oder mehrere *Source Files* (*Quelltext*).
- Diese Source Files werden dann
  - ◇ *kompiliert* durch einen *Compiler* und/oder
  - ◇ *interpretiert* durch einen *Interpreter*.

Prinzipieller Unterschied:

- *Kompilieren*: Das Source File wird in ein Maschinenprogramm übersetzt, das danach jederzeit als ein normaler Prozess des Betriebssystems aufgerufen werden kann.
- *Interpretieren*: Das Source File wird ohne vorherigen Arbeitsgang vom Interpreter Schritt für Schritt ausgeführt.

Diskussion:

- *Kompilieren*: Das kompilierte Maschinenprogramm ist in der Regel *wesentlich* schneller in der Ausführung als die Abarbeitung des Source Files durch einen Interpreter.

  ⟶ Vorteilhaft für den Einsatz des Programms in der Praxis.
- *Interpretieren*: Die (bei großen Programmen mitunter recht hohe) Zeit für's Kompilieren wird eingespart.

  ⟶ Vorteilhaft bei der Entwicklung des Programms (bei der ja laufend das Programm ein wenig verändert und dann neu kompiliert wird).

Zusammenhang mit Programmiersprachen:

- Im Prinzip könnte jede Sprache sowohl kompiliert als auch interpretiert werden.
- Aber verschiedene Sprachen sind für das eine oder das andere besser geeignet.
- Die meisten gängigen Programmiersprachen sind auch von vornherein schon daraufhin entworfen worden, dass sie für eins von beiden besonders gut geeignet sind.

Simples Beispiel für's Interpretieren:

- Erinnerung an Folie 25 und 39: HTML ist die „Sprache des WWW".
- Wenn eine HTML–Seite in einen WWW–Browser geladen wird, startet der Browser ein weiteres Programm, nämlich einen HTML–Interpreter.
- Dieser Interpreter liest den HTML–Text und stellt die WWW–Seite gemäß der darin enthaltenen Formatierungsbefehle dar.

Analogie zum Compilieren: Konvertieren

- Annahme: Ein Programm A schreibt Bilder im Format RGB. Ein zweites Programm B liest aber nur Daten im Format BGR.
- Diese Daten sind also im falschen Format, um sie von Programm A nach Programm B zu übertragen.
- Ein Programm, das die Ausgabe von Programm A in eine Eingabe von Programm B verwandelt, konvertiert die Daten.
- Genauso wird das Programm, dass Sie schreiben, von einem Konverter (dann Compiler) in eine Form übersetzt, die der Prozessor Ihres Rechners versteht.

Java–Konzept:

- Für Java ist die Frage Kompilieren vs. Interpretieren strikt geregelt.
- Und zwar salomonisch: sowohl als auch.
- *Genauer*:
  - ◇ Ein Java Source File wird nicht in Maschinencode, sondern in den sogenannten *Java Byte Code* kompiliert.

    ⟶ Compiler–Programm „`javac`".
  - ◇ Die Abarbeitung des Programms besteht dann darin, dass der Java Byte Code interpretiert wird.

    ⟶ Interpreter–Programm „`java`".
- Es gibt auf den gängigen Computersystemen auch Compiler zur Übersetzung von Java Byte Code direkt in Maschinencode.

  ⟶ Im allgemeinen bessere Laufzeit.

Java Byte Code:

- Eine idealisierte und standardisierte Variation von Maschinencode.
- Basiert auf einem idealisierten Rechnermodell, das stark an das abstrakte Von–Neumann–Modell (Folie 61 ff.) angelehnt ist: die sogenannte *Java Virtual Machine*.
- Durch diese Abstraktion von konkreten Hardwareplattformen und Betriebssystemen kann im Prinzip jedes kompilierte Java–Byte–Code–Programm auf jedem System laufen.

  ⟶ Vorausgesetzt, die Systemanbieter sind kooperationsbereit!
- Neben den typischen Elementen von Maschinen· instruktionen (siehe Folie 66) enthält Java Byte Code noch weitere Informationen, die es z.B. erlauben zu prüfen, ob ein Java–Programm nur auf die Daten und Ressourcen zugreift, die ihm gestattet sind.

Laufzeitsystem:

- Ein Java-Programm interpretieren heißt, die einzelnen Anweisungen im Java-Programm abzuarbeiten. Das bedeutet aber mehr, als man dem Java-Quelltext direkt ansehen kann.
- Neben der Abarbeitung der Instruktionen im Source File werden noch einige unterstützende Dienste (Beispiele siehe nächste Folie) nebenher miterledigt.
- Die Gesamtheit aller solchen Dienste, die so im Hintergrund vom Interpreter erledigt werden und dafür sorgen, dass der selbst geschriebene Java-Quelltext überhaupt erst sinnvoll und sicher ausgeführt werden kann, nennt man das *Laufzeitsystem*.
- Diesbezüglicher Unterschied zum Compiler:
  Der Compiler „mixt“ die Anweisungen aus dem Source File bei dessen Übersetzung mit den vordefinierten Anweisungen des Laufzeitsystems.

Fortsetzung Laufzeitsystem:

- Programmiersprachen unterscheiden sich u.a. auch darin, was das Laufzeitsystem so alles leistet.
- In Java leistet das Laufzeitsystem wesentlich mehr als in den meisten anderen Programmiersprachen.
- *Erstes Beispiel dafür:* Die Prüfung des Zugriffs auf Daten und Ressourcen gemäß vorletzter Folie.
- *Weiteres Beispiel:* Umgang mit mathematischen Ausnahmesituationen wie z.B. Division durch 0. Wird für einen solchen Fall keine besondere Vorkehrung im Programm getroffen, ist es Aufgabe des Laufzeitsystems, das Programm mit einer Fehlermeldung zu beenden.

Abschnitt 2.2: Abstraktionsebenen

- Lexikalische Ebene (Wortebene)
- Syntaktische Ebene (Satzebene)
- Semantische Ebene (Bedeutungsebene)
- Logische Ebene
- Spezifikatorische Ebene

Beachte:

- Die Grenze zwischen den Ebenen ist fließend.
- Die Grenzziehung in dieser Vorlesung ist daher bis zu einem gewissen Grad auch willkürlich.
- Was allerdings nicht gleichbedeutend ist mit „ohne Überlegung“!

Allgemeine Vorbemerkung:

In Programmiersprachen sind die lexikalischen, syntaktischen und semantischen Regeln, was korrekt ist und was nicht, sehr viel rigider und pedantischer als etwa in natürlichen Sprachen.

Gründe:

- Es ist wesentlich schwieriger und aufwendiger, einen Spracherkenner und Compiler oder Interpreter für eine weniger pedantische Sprache zu entwickeln.

  $\longrightarrow$ Für natürliche Sprachen ist dieses Problem bis heute ungelöst!
- Die Möglichkeit mehrdeutiger Texte wäre
  - ◇ ab einem gewissen Komplexitätsgrad der Sprache wohl kaum vermeidbar,
  - ◇ bei normaler Kommunikation in natürlichen Sprachen ein häufiges Ärgernis,
  - ◇ bei Programmiersprachen eine Katastrophe.

Grundregel:

- Lexikalische und syntaktische Fehler resultieren in Fehlermeldungen beim Kompilieren (also beim Aufruf von „`javac`“).

  $\longrightarrow$ Kein Source File mit lexikalischen oder syntaktischen Fehlern kann in Java Byte Code übersetzt werden.

- Fehler der anderen Arten resultieren in Fehlverhalten des Programms zur Laufzeit (also beim Laufenlassen mit „`java`“):
  - ◇ Programmabsturz,
  - ◇ korrekte Programmbeendigung mit falschen Endergebnissen,
  - ◇ falsche oder unerwünschte Effekte des laufenden Programms auf Fileinhalte, Bildschirmausgaben etc.

Ebene 1: Lexikalisch

- Auf unterstem Abstraktionsniveau ist ein Programm zusammengesetzt aus
  - ◇ lexikalischen Einheiten (auch *Lexeme* oder *Token genannt*)
  - ◇ sowie trennenden Elementen (*Separatoren*).
- *Lexeme*:
  - ◇ Schlüsselwörter,
  - ◇ Bezeichner (*Identifier*),
  - ◇ Literale,
  - ◇ Operatoren,
  - ◇ Klammerungszeichen,
  - ◇ Semikolon.
- *Separatoren*:
  - ◇ Leerzeichen (auch *Blank* oder *Space* genannt),
  - ◇ Tabulatorzeichen,
  - ◇ Zeilenumbruch (vgl. Folie 32).

Schlüsselwörter:

- Reservierte Zeichenketten mit besonderer Bedeutung.
- *Beispiele* in Java: „`if`“, „`for`“, „`class`“, „`return`“.
- In Java bestehen alle Schlüsselwörter aus Kleinbuchstaben.
- *Auflistung* aller Java–Schlüsselwörter:

  Findet sich in so ziemlich jedem Buch oder Vorlesungsskript zu Java (nur nicht hier :-).

Identifier (Bezeichner):

- *Bedeutung*: Jede Bezeichnung, die vom Programmierer des Source Files selbst ausgedacht ist.

  $\longrightarrow$ Namen von Variablen, Konstanten, Klassen, Methoden, Paketen (später in der Vorlesung).

- Im Prinzip ist jede beliebige Zeichenkette erlaubt, die aus Groß- und Kleinbuchstaben, Ziffern sowie dem Unterstrich „_“ (engl. *Underscore*) besteht.
- *Ausnahmen* zur Vermeidung von Zweideutigkeiten:
  - ◇ Schlüsselwörter dürfen natürlich nicht als Identifier gewählt werden.
  - ◇ Ein Identifier darf nicht mit einer Ziffer beginnen, um Verwechslungen mit Zahlen zu vermeiden.

Beachte:

In Java wird im Gegensatz zu einigen anderen Programmiersprachen zwischen Groß- und Kleinschreibung bei Identifiern unterschieden:

- „`hallo`“, „`Hallo`“ und „`HALLO`“ sind drei verschiedene Identifier.
- „`IF`“, „`If`“ und „`iF`“ sind zulässige Identifier, obwohl „`if`“ ein Schlüsselwort ist.

Beispiele:

`int i = 7`
Hier ist ,,i” der Identifier.

`int iF = 7; int If; int IF`
Alle diese Identifier sind zulässig . . .

`if(iF==If) return IF;`
. . . aber nicht unbedingt sinnvoll!

Exkurs: Wahl von Identifiern

- In Java ist gemäß letzter Folie bei der Wahl von Identifiern sehr viel erlaubt.
- Aber damit ein Programm lesbar und verständlich bleibt, sollten die Identifier so gewählt sein, dass sie so etwas wie „sprechende Kommentare“ bilden.
- Das ist vor allem wichtig,
  - ◇ wenn das Programm deutlich größer ist als unsere Micky–Maus–Beispiele in Vorlesung und Übungen oder
  - ◇ wenn andere Leute das Programm auch verstehen sollen.

  ⟶ Also praktisch immer.

Beachte:

- Der Identifier selbst ist der einzige Kommentar zur Funktion dieses Identifiers, der bei jedem Auftreten des Identifiers garantiert dabei ist.

  ⟶ Man muss nicht erst umständlich irgendwo anders nach einem Kommentar zu diesem Identifier suchen.
- „Richtige“ Kommentare hinter „`//`“ oder in „`/* ... */`“ sind erfahrungsgemäß in der Praxis eher rar und oft falsch oder veraltet.
- Das natürliche Bedürfnis des Programmierers, möglichst wenig zu tippen und daher extrem kurze Identifier zu verwenden, muss dahinter zurücktreten.

Positive Beispiele:

- Methoden „`drawString`“, „`drawLine`“, „`drawOval`“, „`fillOval`“ von „`Graphics`“.
- „`zeichneHausVomNikolaus`“ für eine Methode, die das Haus vom Nikolaus zeichnet.

Stylistische Regeln:

- Im Gegensatz zu anderen Programmiersprachen hat sich in Java ein Regelwerk für die „Stylistik“ von Identifiern allgemein durchgesetzt.
- Viele vorgefertigte Java–Bausteine und unzählige Java–Anwendungsprogramme vertrauen darauf, dass diese Konventionen streng eingehalten werden.
- *Grundregeln*:
  - ◇ Der Underscore ist eher verpönt.
  - ◇ Wortanfänge innerhalb eines Identifiers werden mit Großbuchstaben gekennzeichnet.
  - ◇ Namen von Klassen beginnen mit Grobuchstaben.

    ⟶ „`IchBinEineKlasse`“
  - ◇ Namen von Variablen und Methoden beginnen mit Kleinbuchstaben.

    ⟶ „`ichBinEineVariable`“
    „`ichBinEineMethode`“

... Exkurs Ende

Operatoren:

- *Beispiele*:

```
+  -  *  /  %  ++  --  .  ?  :
```

- Allgemein bestehen Operatoren in Java aus einem oder zwei Sonderzeichen.
- *Auflistung* aller Java–Operatoren:

  Findet sich wieder in so ziemlich jedem Buch oder Vorlesungsskript zu Java (nur nicht hier :-).

Klammerungszeichen:

Runde, geschweifte und eckige Klammern, d.h.

```
(  )    {  }    [  ]
```


Literale:

- Literale sind explizit hingeschriebene Werte.
- *Beispiele*:
  - ◇ `69534`: ganzzahliges Literal,
  - ◇ `3.14159`: reellwertiges Literal,
  - ◇ `'a'`: Zeichenliteral,
  - ◇ `''Hello World''`: Stringliteral,
  - ◇ `true` und `false`: die beiden Literale zum Datentyp `boolean`.

Separatoren:

- In Java dürfen (wie in eigentlich allen modernen Programmiersprachen) beliebig viele Separatoren in beliebiger Mischung zwischen Lexeme gesetzt werden.
- Andererseits besteht nur in bestimmten Konstellationen überhaupt der Zwang, mindestens einen Separator zwischen zwei unmittelbar aufeinanderfolgende Lexeme zu platzieren.

  Beispiel: zwischen Typname und Variablenname bei der Variablendeklaration.

```
int i;       // Erlaubt
int     i;   // Erlaubt
inti;       // Verboten
```

- In den meisten Programmiersprachen (einschl. Java) ist es logischerweise strikt verboten, Separatoren *innerhalb* von Lexemen zu platzieren.
- Die Freiheit der Platzierung zwischen Lexemen kann man in Java zur übersichtlichen, gut lesbaren Formatierung des Source Files nutzen.

Beispiel:

- Zweimal derselbe Quelltext bis auf Platzierung von „unnützen“ zusätzlichen Separatoren.
- Welche Version ist übersichtlicher und besser lesbar?

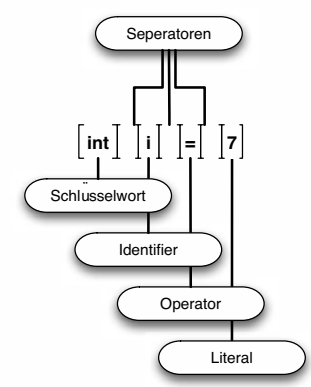
Version 1:

```
double summe        = 0;
double quadratsumme = 0;

for ( int i=0; i <= n; i++ )
 {
   summe        += i;
   quadratsumme += i * i;
 }
```

Version 2:

```
double summe=0;double quadratsumme=0;for(int
i=0;i<=n;i++){summe+=i;quadratsumme+=i*i;}
```

Abschließendes Beispiel zur lexikalischen Ebene:



Abschließende Bemerkung zur lexikalischen Ebene:

- In einigen älteren Programmiersprachen (bzw. älteren Versionen heutiger Programmiersprachen) müssen wesentlich strengere lexikalische Regeln befolgt werden.
- *Beispiel*: Alte Versionen von Fortran.
- *Konkretes Beispiel* daraus: Zeilenformatierung.
  - ◇ Die Zeilen dürfen nicht zu lang werden (z.B. nicht mehr als 80 Zeichen).
  - ◇ Einzelne Bestandteile einer Programmzeile müssen an vorgegebenen Abständen zum Zeilenbeginn platziert werden.

  $\longrightarrow$ Altlast aus dem Zeitalter der Lochkarten.

Ebene 2: Syntaktisch

- Hier geht es nicht um die Korrektheit einzelner Lexeme,
- sondern um die korrekte Gruppierung von Lexemen in Java Source Files.
- *Erinnerung* an Folie 197: Syntaktische Fehler werden wie lexikalische Fehler schon beim Kompilieren abgefangen.

Beispiel I: Korrekte Klammerungen

- Öffnende und schließende Klammern dürfen nur in Paaren im Source File auftreten.
- Die öffnende Klammer kommt dabei vor der schließenden.
- Die Textabschnitte innerhalb zweier Klammerpaare dürfen
  - ◇ nacheinander ohne Überlappung platziert sein:

    `(...)...(...)...{... }`
  - ◇ ineinander enthalten („geschachtelt“) sein:

    `(...[...{...}...(...)...]...)`
  - ◇ aber sich nicht anderweitig überlappen:

    `(...[...)...]` $\longleftarrow$ Verboten!

Beispiel II: Platzierung von Schlüsselwörtern

- Ein Schlüsselwort ist Teil eines Programmkonstrukts.
- Es darf nur innerhalb dieses Programmkonstrukts auftreten,
- und auch dort nur in genau festgelegter Form an genau festgelegter Stelle.

Konkretes Beispiel:

Das Schlüsselwort „`while`“ steht am Anfang des Konstrukts „`while`–Schleife“:

```
boolean theWorldIsNotEnough = true;
while ( theWorldIsNotEnough )
   if ( canGetSatisfactionFrom(thisWorld) )
      theWorldIsNotEnough = false;
```

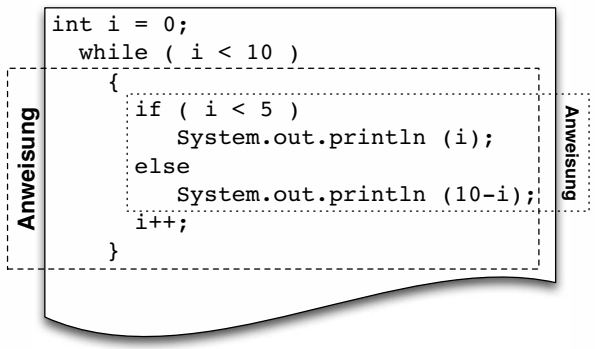

Syntax der while–Schleife:

`while ( `*Bedingung*` ) `*Block*

Erläuterungen:

- Die Schriftart für die einzelnen Bausteine hat hier eine Bedeutung:
  - ◇ `Schreibmaschinenstil`:
    Der Baustein muss wörtlich so dastehen.
    ⟶ `while ( )`
  - ◇ *Kursivstil*: Der Baustein ist ein „Platzhalter“ für den eigentlich dort erwarteten Quelltext.
- Platzhalter „*Bedingung*“: Ein Ausdruck mit einem binären logischen Wert true/false.
- Platzhalter „*Block*“:
  - ◇ entweder eine einzelne Anweisung
  - ◇ oder eine Sequenz von Anweisungen in geschweiften Klammern „$\{\ldots\}$“.

Beispiel mit „$\{\ldots\}$“:



Anweisungen allgemein:

- Elementare Anweisungen wie Zuweisungen mit „=“ oder Inkrement/Dekrement („$++$“/„$--$“).
- Methodenaufrufe wie

  `System.out.println(...).`
- `if`-/`while`-/`for`–Konstrukte etc.

⟶ Genauer in Abschnitt 3.1 (Folie 236 ff.).

Schachtelung:

- `if`-/`while`-/`for`–Konstrukte u.ä. können beliebig ineinander geschachtelt werden.
- Insbesondere können auch zwei oder mehr Konstrukte desselben Typs ineinander geschachtelt werden.

⟶ Siehe nächste Folie für ein Beispiel.

„Beispiel“ zur Illustration:

```
int i = 0;
while ( i<10 )
 {
   if ( i==5 )
    {
      int j = 0;
      while ( j<10 )
       {
         if ( j==i )
          {
            int k = 0;
            while ( k != i-j )
             {
               if ( k != j )
                  System.out.println (j);
               k++;
             }
          }
         j++;
       }
    }
   i++;
 }
```


Ebene 3: Semantisch

- Bisher haben wir zwei Ebenen betrachtet, die mit der *Form* von Texten zu tun hatten.

  ⟶ Wortebene und Satzebene.

- Der Begriff „Semantik einer Sprache“ bezieht sich im Gegensatz dazu auf die *Bedeutung* von Texten, die lexikalisch und syntaktisch korrekt formuliert sind.

Beispiel I: Bindungsstärke von Operatoren

- *Erinnerung* an Folie 205: Es gibt verschiedene Operatoren in Java (z.B. arithmetische).
- Die verschiedenen Operatoren haben verschiedene *Bindungsstärken*.

  ⟶ Damit wird die Auswertungsreihenfolge in einem Ausdruck mit mehreren Operatoren festgelegt.

- *Konkretes Beispiel*:

  ```
  int i = 3 - 10 / 2 * 4 + 1;
  ```

  ◇ Wie in der Mathematik gilt in Java Punkt- vor Strichrechnung.

  ◇ Ansonsten werden zumindest die hier verwendeten vier arithmetischen Operatoren strikt von links nach rechts ausgewertet.

  ⟶ Der Wert von „`i`“ in der obigen Quelltextzeile ist

$$\left(3 - \left(\frac{10}{2} \cdot 4\right)\right) + 1 \;=\; -16$$

Beispiel II: Operatoren und Datentypen

Kurzer Vorgriff auf Abschnitt 3.2, Aspekt 2 (Folie 287):

- Eine Variable vom Datentyp `int` speichert ganze Zahlen.
- Eine Variable vom Datentyp `float` oder `double` speichert reelle Zahlen (durch Näherungswerte).

Speziell Divisionsoperator:

```
int    i = 17;
int    j = 3;
int    k = i / j; // k == 5

float  x = 17;
float  y = 3;
float  z = x / y; // z == 5,66...
```

Unterschied in der Semantik:

- Ganzzahlige Division mit Rest, wenn beide Operanden ganzzahlig sind.

  $\longrightarrow$ Der Wert von „k“ auf der letzten Folie ist 5.
- Reelle Division, wenn mindestens einer der beiden Operanden reell ist.

  $\longrightarrow$ Bis auf numerische Ungenauigkeiten ist der Wert von „z“ auf der letzten Folie gleich $5.666\ldots$.

Achtung Falle: `float z = 17 / 3;`

- Der Datentyp für das Ergebnis der Division ist zwar reell.
- Die beiden Operanden sind aber ganzzahlig, also wird eine Ganzzahldivison durchgeführt, deren Ergebnis verlustfrei in einen `float` umgewandelt werden kann.

$\longrightarrow$ Gemäß ganzzahliger Division mit Rest ist der Wert von „z“ in diesem Fall gleich 5.0, nicht $5.666\ldots$.

Modulo–Operator:

- Der Operator „%“ berechnet den Rest bei einer Division mit Rest:

  ◇ `10 % 3` $\longrightarrow 1$
  ◇ `11 % 3` $\longrightarrow 2$
  ◇ `12 % 3` $\longrightarrow 0$
- Da diese Operation in der Mathematik „modulo“ genannt wird, heißt dieser Operator der *Modulo–Operator*.
- *Frage*: Wozu kann man das gebrauchen?
- *Antwort*: auf der nächsten Folie, an einem Beispiel...

Wozu also Modulo:

```
int wochentag = 0;  // Sonntag

...

if ( itsMidnight() )
   wochentag = ( wochentag + 1 ) % 7;
```

Erläuterungen:

- Um den aktuellen Wochentag zu speichern, könnte man zum Beispiel eine „`int`“–Variable nehmen, die die Werte $0 \ldots 6$ annimmt.
- Dabei würde dann zum Beispiel 0 für Sonntag, 1 für Montag usw. bis 6 für Samstag stehen.
- Wenn der Wert von Tag zu Tag umgeschaltet wird, soll er von 0 bis 6 wachsen und dann wieder auf 0 gehen.
- Das ist aber genau das Verhalten „`% 7`“.

Beispiel III: Einklammerungen

- *Erinnerung* an Folie 212:

  ◇ Klammerpaare dürfen nacheinander oder ineinander geschachtelt auftreten:

  `(...)...(...)...{... }`

  `(...[...{...}...(...)...]...)`

  ◇ Aber sie dürfen sich nicht anderweitig überlappen:

  `(...[...)...]`
- *Generelle Semantik*:

  ◇ Der Text innerhalb eines Klammerpaars bildet eine separate Einheit.

  ◇ Bei ineinander geschachtelten Klammerpaaren bildet der Text im inneren Klammerpaar eine Untereinheit der Texteinheit im äußeren Klammerpaar.

Beispiel:

```
int  i = 3 -   10 /   2 *   4 + 1       ;
int  j = 3 - ( 10 / ( 2 * ( 4 + 1 ) ) ) ;
```

Erläuterungen:

- Durch die Klammerung werden Einheiten, Untereinheiten und Unteruntereinheiten des mathematischen Ausdrucks gebildet.
- Der Inhalt jedes Klammerpaars wird als Einheit berechnet und mit dem Rest weiterverarbeitet.
- Bei ineinandergeschachtelten Klammerpaaren folgt daraus Auswertung von innen nach außen.

  $\longrightarrow$ Der Wert von „j“ im obigen Quelltextfragment ist

$$3 - \frac{10}{2 \cdot (4+1)} = 3 - \frac{10}{2 \cdot 5} = 2\,.$$

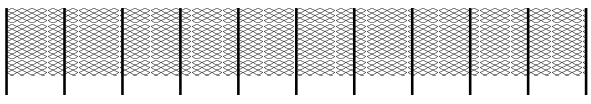
- Gemäß Folie 219 hat „i“ hingegen den Wert -16.

Ebene 4: Logisch

- Wir gehen in diesem Punkt von lexikalisch/syntaktisch korrekten Quelltexten aus, die in Programme übersetzbar sind und auch keine semantischen Fehler enthalten.
- *Frage*: Was heißt es dann eigentlich noch, dass ein Programm korrekt bzw. nicht korrekt ist?
- *Salopp formulierte Antwort*: Ein Programm ist dann korrekt, wenn es
  - ◇ alles das tut, was es soll,
  - ◇ und das mit allen erwünschten Effekten,
  - ◇ und darüber hinaus keine Effekte hat, die es definitiv nicht haben soll.
- Die Formulierung des erwünschten und unerwünschten Verhaltens eines Programms ist seine *Spezifikation*.
- Nur relativ zu einer Spezifikation macht es überhaupt Sinn, von Korrektheit/Nichtkorrektheit eines Programms zu sprechen.

Häufiges Beispiel für einen logischen Fehler:

- Das sogenannte „Fencepost Syndrom“ (auch „off–by–one error“).
- What's in a name: Wie viele Zaunpfähle braucht man, wenn
  - ◇ der Zaun 20 Meter lang werden soll und
  - ◇ alle 2 Meter ein Pfahl steht?
- *Antwort*:
  - ◇ Die unwillkürliche Antwort ist „10“.
  - ◇ Die korrekte Antwort zumindest bei normalen Zäunen ist aber eher „11“:



Typisches Programmierbeispiel dazu:

```
int  linkestesPixel = 187;
int rechtestesPixel = 379;
...

int fensterWeite = rechtestesPixel
                 -  linkestesPixel;
```

Erläuterungen:

- Die Weite eines Fensters ist die Anzahl der nebeneinanderliegenden Pixel von ganz links bis ganz rechts.
- Im obigen Beispiel geht das Fenster von Bildschirmpixel Nr. 187 ganz links bis Bildschirmpixel Nr. 379 ganz rechts.
- Unwillkürlich ist man geneigt, die Fensterweite als Differenz der beiden Pixelwerte zu berechnen.
- *Fencepost Syndrom*: Tatsächlich ist die Weite gleich

```
rechtestesPixel - linkestesPixel + 1
```

Reales, subtiles Beispiel für Arrayfehler:

- In einem nicht ganz unwichtigen Programm ist für den (englischen) Namen des Wochentags ein Array von acht Zeichen reserviert worden.
- *Konsequenz*:
  - ◇ Das Programm lief eigentlich immer problemlos und korrekt...
  - ◇ ...außer mittwochs!

M	O	N	D	A	Y			
T	U	E	S	D	A	Y		
W	E	D	N	E	S	D	A	Y
0	1	2	3	4	5	6	7	

⟶ Beispiel dafür, dass ein logischer Fehler häufig nicht so ohne weiteres reproduzierbar ist, um ihn einzukreisen und aufzuspüren.

Beachte:

- Programmierfehler auf der logischen Ebene werden gemäß Folie 197 nicht beim Kompilieren des Quelltextes mit Fehlermeldung abgefangen.
- Statt dessen führen solche Fehler zu *undefiniertem* (d.h. unvorhersehbarem) Verhalten des Programms bei seinem Ablauf als Prozess.
- *Zum Beispiel*:
  - ◇ Programmabsturz, d.h. vom Programm nicht beabsichtigte Termination durch das Eingreifen des Laufzeitsystems (Folie 193; siehe auch Folie 128).
  - ◇ keine *Termination* (=Beendigung),
  - ◇ kontrollierte Termination mit inkorrekten Ergebnissen,
  - ◇ kontrollierte Termination mit korrekten Ergebnissen (durchaus möglich!),
  - ◇ Rechnerstillstand (bei gewissen Betriebssystemen...),
  - ◇ ...

Keine Termination:

```
int i = 0;
while ( i < 10 )
   System.out.println (i);
   i++;
```

Erläuterung:

- Eigentlich sollte das obige Java–Programmfragment die Zahlen $0 \ldots 9$ ausgeben.
- Durch eine Unachtsamkeit ist aber vergessen worden, Klammern um die beiden Anweisungen zu setzen.
  - ⟶ Passiert sehr leicht, wenn man in eine Schleife mit einer einzigen Anweisung nachträglich noch weitere Anweisungen einfügt.
- Der Wert von „`i`“ bleibt daher immer gleich 0.
  - ⟶ Die Bedingung „`i<10`“ bleibt immer wahr.
  - ⟶ Nie abbrechende „Endlosschleife“.

Ebene 5: Spezifikatorisch

- Auf der logischen Ebene haben wir es mit Diskrepanzen zwischen Spezifikation und Programmverhalten zu tun.
- Auf der spezifikatorischen Ebene geht es um Diskrepanzen zwischen den eigentlichen Intentionen des Programms und seiner Spezifikation.

(Nicht mehr ganz) aktuelles Beispiel:

- Die Spezifikation verlangt, dass für ein Datum der Tag, der Monat und das Jahr abzuspeichern sind.
- Aber der Programmierer hat entschieden, für jedes Jahr nur die letzten beiden Ziffern abzuspeichern.
- Bis zum 31.12.1999 stimmten Spezifikation und Programm hundertprozentig überein...

Idealbild einer Spezifikation:

- Ein übersichtlich und systematisch strukturiertes Schriftstück.
- Unmissverständliche, unzweideutige, nicht interpretierbare Formulierungen.
- Verständlich und zugleich exakt.
- Deckt jeden möglichen Fall, der prinzipiell auftreten kann, auch ab.

Problem:

In der Realität wäre wohl

- die Erstellung einer solchen „idealen“ Spezifikation viel zu aufwendig und
- die resultierende Spezifikation zu umfangreich und komplex.

$\longrightarrow$ Liest kein Mensch!

Simples Beispiel:

- Eine saloppe Spezifikation für das UNIX–Programm „`cal`“ würde besagen:

  Das Programm „`cal`“ berechnet den Kalender für das angegebene Jahr und den angegebenen Monat.
- Eine exakte Spezifikation müsste noch konkret enthalten:
  - ◇ Welcher Kalendar eigentlich (gregorianisch, julianisch, ...).
  - ◇ Der Bereich von Jahren, in dem das Programm korrekte Kalender berechnen können soll.
  - ◇ Die Berücksichtigung von Angleichungen wie im September 1752 (siehe Man Page zu „`cal`“).
  - ◇ Die Regeln für die Berechnung von Schaltjahren.
  - ◇ Die Behandlung fehlerhafter Benutzereingaben.

$\longrightarrow$ Eine ganze Menge schriftlicher Stoff allein für eine simple Komponente zur Kalenderberechnung.

Abschnitt 3: Basiskonzepte von Java

- Die in Abschnitt 3 vorgestellten Konzepte von Java finden sich so oder so ähnlich in eigentlich jeder gängigen Programmiersprache.

  $\longrightarrow$ Abschnitt 3.2, Aspekt 3, Klassen (Folie 300...) nur in objektorientierten Programmiersprachen.
- Die Unterschiede in den Details (vor allem in der Syntax) sind allerdings sehr groß.
- Wir werden uns in diesem Abschnitt voll auf Java konzentrieren.
- Aber (hoffentlich) in dem Bewusstsein, dass Java letztendlich nur ein Beispiel ist.
- Wenn man diese Konzepte einmal an einem Beispiel wie Java verstanden hat, wird man sie in jeder anderen Programmiersprache leicht wiederfinden und durchschauen.

Abschnitt 3.1: Programmfluss

Erinnerung an Folie 191 ff.:

- Java–Quelltext wird durch einen Compiler wie „`javac`“ in eine idealisierte Form von Maschineninstruktionen überführt (*Java Byte Code*).
- Durch einen Interpreter wie „`java`“ wird das Programm in seiner Form als Java Byte Code ausgeführt.
- Im Prinzip läuft das wie die Ausführung von „echtem“ Maschinencode durch die Hardware ab.
- Andere Java–Compiler übersetzen Java–Quelltext auch wahlweise entweder in Maschinencode statt in Java Byte Code, der dann von der Hardware als „Interpreter“ auszuführen ist (wie es bei vielen anderen Programmiersprachen die Regel ist, z.B. Ada, C, C++ Cobol, Fortran, Pascal).

Erinnerung an Folie 65 ff.:

- Eine interne Uhr zerlegt die Zeit in einzelne Taktzyklen, und in jedem Taktzyklus wird eine Maschineninstruktion abgearbeitet (etwas idealisiert formuliert!).
- Im Prinzip werden die Maschineninstruktionen eines ausführbaren Programms Schritt für Schritt nach dieser Uhr in der Reihenfolge ihrer Speicheradressen abgearbeitet.
- Durch Sprunginstruktionen (und ausschließlich dadurch!) kann von dieser Abfolge beliebig abgewichen werden.
- Zum Beispiel ergibt sich eine Schleife, wenn mit einer bedingten Sprunganweisung zu einer vorher schon einmal abgearbeiteten Instruktion zurückgesprungen wird.

  $\longrightarrow$ Sprungbedingung $\equiv$ Abbruchbedingung
- Siehe Folie 71 für ein erstes Beispiel.

Nun Java:

- Die Abarbeitung von Java Byte Code durch einen Interpreter läuft im Prinzip genauso ab wie die Abarbeitung von Maschinencode im Von–Neumann–Modell (Folie 61 ff.).
- Jede elementare Anweisung ($\rightarrow$ Folie 215) eines Java–Quelltextes wird im allgemeinen in ein oder mehrere Instruktionen in Java Byte Code (bzw. Maschinencode) übersetzt.
- Konstrukte wie Verzweigungen und Schleifen werden in jeweils spezifische Konstellationen von bedingten und unbedingten Sprüngen übersetzt (vgl. Folie 71 sowie später Folie 244 ff.).

Begriff Programmfluss:

- Zu jedem Zeitpunkt der Abarbeitung eines Programms wird eine einzelne Instruktion abgearbeitet.
- Die Abfolge der abgearbeiteten Instruktionen ist der dynamische *Programmfluss* durch den statischen *Programmtext* hindurch.

Aspekt 1: Ausdrücke

Ein Ausdruck konstituiert sich durch

- einen *Rückgabetyp*,
- einen *Rückgabewert* und
- optionale Seiteneffekte.

Beispiel I: `double x = (3+6) / 2;`

Ausdruck

Erläuterungen:

- Der Rückgabewert des Ausdrucks, der mit der Klammer umrissen ist, ist 4 (vgl. Folie 220 ff.).
- Der Rückgabetyp dieses Ausdrucks ist „`int`“, nicht „`double`“.
- Bei der Zuweisung an die „`double`“–Variable wird dieser „`int`“–Wert in einen „`double`“–Wert konvertiert.

Beispiel II:

```
int fakultaet ( int n )
 {
   int rueckgabeWert = 1;
   for ( int i=2; i<=n; i++ )
      rueckgabeWert *= i;
   return rueckgabeWert;
 }
```

Erläuterungen:

- Operator „`*=`“ weist der Variablen auf der linken Seite das Produkt aus ihrem eigenen momentanen Wert und dem Wert des Ausdrucks auf der rechten Seite zu.
- Analog sind „`+=`“, „`-=`“ und „`/=`“ definiert.

Beispiel III:

```
double meineMethode ( int n )
 {
   System.out.println ( 2*n );
   if ( n > 3 )
      return 4*n;
   else
      return 5*n;
 }

...

double x = meineMethode (6) + 7;
```

⟶ Erläuterungen auf der nächsten Folie.

Erläuterungen:

- Eine Methode, die nicht „`void`“ ist, kann in Form eines Ausdrucks aufgerufen werden.
- Der Rückgabetyp einer solchen Methode ist der Typ, der am Anfang des Methodenkopfes steht.
- Der Rückgabewert ist der Wert des Ausdrucks hinter dem „`return`“, mit dem die Abarbeitung beendet wird.
- Methodenaufrufe können auch *Seiteneffekte* haben:
  - ◇ Alle Effekte einer Methode auf die „Umwelt“ außerhalb der Methode.
  - ◇ *Ausnahme*: Die Rückgabe eines Wertes wird nicht als Seiteneffekt, sondern sozusagen als der „Haupteffekt“ der Methode angesehen.
- *Im Beispiel*: Das Schreiben eines Wertes auf den Bildschirm ist ein typischer Seiteneffekt.

Exkurs: Ausgabe auf dem Bildschirm

`System.out.print(Argument)` Schreibt das Argument auf den Bildschirm. Das Argument darf dabei von folgenden Typen sein:

- Alle Eingebauten Typen (`int, float, double, char, boolean`)
- `String`

`System.out.println(Argument)` Tut das selbe, jedoch beendet es die Ausgabe mit einem Zeilenwechsel. Dadurch wird die nächste Ausgabe in eine neue Zeile geschrieben.

Beispiele:

```
// Gibt Hallo Welt aus
System.out.print(``Hallo Welt'');
// Gibt die Zahl 1.765 aus
System.out.print(1.765);
```

. . . Exkurs Ende

Aspekt 2: Verzweigungen und Schleifen

Allgemeiner Zusammenhang zwischen Programmstruktur und Programmfluss:

- In Programmteilen, in denen keine Verzweigungen, Schleifen, Methodenaufrufe, „`return`“*s* etc. vorkommen, folgt der Programmfluss strikt der sequentiellen Programmstruktur.
- *Das heißt*: Die Anweisungen werden sequentiell in der Reihenfolge abgearbeitet, in der sie im Quelltext auftreten.
- Verzweigungen und Schleifen sind in Java die einfachsten, grundlegenden Möglichkeiten, den Programmfluss von dieser sequentiellen Struktur des Quelltextes zu lösen.
- Wir werden sehen, wie sich diese Konstrukte auf bedingte und unbedingte Sprungbefehle zurückführen lassen (vgl. Folie 69 ff.).

Verzweigung:

```
if ( x == 1 )
   y = y + 1;   // If-Zweig
else
   y = y - 1;   // Else-Zweig
```

Semantik

1. Stelle fest, ob die Speicherzelle x momentan den Wert 1 enthält.
2. Falls ja, springe zur Instruktion Nr. 5 (bedingter Sprung).
3. Führe die Maschineninstruktionen für den „`else`“–Zweig aus (d.h. vermindere den Wert in der Speicherzelle y um 1).
4. Springe zur Instruktion Nr. 6 (unbedingter Sprung).
5. Führe die Maschineninstruktionen für den „`if`“–Zweig aus (d.h. erhöhe den Wert in der Speicherzelle y um 1).
6. ...

Verzweigung mit nur einem Zweig:

```
if ( x == 1 )
   y = y + 1;   // If-Zweig
```

Semantik

1. Stelle fest, ob die Speicherzelle x momentan den Wert 1 enthält.
2. Falls nein, springe zur Instruktion Nr. 4.
3. Führe die Maschineninstruktionen für den „`if`“–Zweig aus (d.h. erhöhe den Wert in der Speicherzelle y um 1).
4. ...

While–Schleife:

```
while ( x > 0 )
   x = x - 1;    // Schleifenrumpf
```

Semantik

1. Stelle fest, ob die Speicherzelle x momentan einen Wert größer als 0 enthält.
2. Falls nein, springe zur Instruktion Nr. 5.
3. Führe die Maschineninstruktionen für den Schleifenrumpf aus. (d.h. vermindere den Wert in der Speicherzelle x um 1).
4. Springe zur Instruktion Nr. 1.
5. ...

Do–While–Schleife:

```
do
   x = x - 1;    // Schleifenrumpf
while ( x > 0 );
```

Semantik

1. Führe die Maschineninstruktionen für den Schleifenrumpf aus.
2. Stelle fest, ob die Speicherzelle x momentan einen Wert größer als 0 enthält.
3. Falls ja, springe zur Instruktion Nr. 1.
4. ...

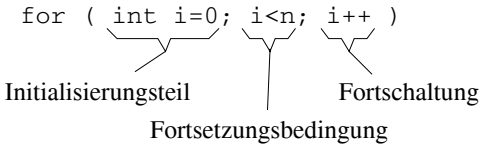
$\longrightarrow$ **Semantischer Unterschied** zur While–Schleife: Der Schleifenrumpf wird mindestens einmal durchlaufen.

For–Schleife:

Eine For–Schleife ist äquivalent zu einer While-Schleife mit

- Initialisierungsteil vor der eigentlichen Schleife,
- gleicher Fortsetzungsbedingung und
- Fortschaltung am Ende des Rumpfes.

Beispiel:



Zur Syntax:

Initialisierungsteil, Fortsetzungsbedingung und Fortschaltung müssen immer durch Semikolons voneinander getrennt werden.

Umsetzung For- in While–Schleife:

- Beispiel einer For–Schleife:

```
for ( int i=0; i<n; i++ )
   A[i] = i;
```

- Umsetzung in einen äquivalenten Java–Quelltext mit While–Schleife:

```
int i = 0;       // Initialisierung
while ( i<n )
 {
   A[i] = i;
   i++;          // Fortschaltung
 }
```


Semantik

1. Setze den Inhalt der Speicherzelle „`i`“ auf den Wert 0.
2. Stelle fest, ob der momentane Wert von „`i`“ kleiner als „`n`“ ist.
3. Falls nein, springe zur Instruktion Nr. 7.
4. Setze den Inhalt von „`A[i]`“ auf den Wert von „`i`“.
5. Erhöhe den Wert von „`i`“ um 1.
6. Springe zur Instruktion Nr. 2.
7. ...

Steuerung des Programmflusses im Schleifenrumpf:

- Bei jeder der drei Schleifenarten (While, Do–While, For) kann die Ausführung der Schleife im Schleifenrumpf durch spezielle Anweisungen noch feiner gesteuert werden.
- *Konkret*:
  - ◇ „`break`“: Bricht die ganze Schleife ab.
  - ◇ „`continue`“: Bricht die aktuelle Ausführung des Schleifenrumpfs ab und lenkt den Programmfluss zur Auswertung der Fortsetzungsbedingung zurück.

Beispiel:

```
while ( x > 0 )
 {
   y = y + 1;        // (a)
   z = z + 2;        // (b)
   if ( y > 100 )
      break;
   if ( z == 200 )
      continue;
   x = x - 1;        // (c)
 }
```


Semantik des Beispiels analog zu Folie 71:

1. Stelle fest, ob die Speicherzelle x momentan einen Wert größer als 0 enthält.
2. Falls nein, springe zur Instruktion Nr. 10.
3. Führe die Maschineninstruktionen für die Anweisungen (a) und (b) aus.
4. Stelle fest, ob die Speicherzelle y momentan einen Wert größer als 100 enthält.
5. Falls ja, springe zur Instruktion Nr. 10 („`break`“).
6. Stelle fest, ob die Speicherzelle z momentan den Wert 200 enthält.
7. Falls ja, springe zur Instruktion Nr. 1 („`continue`“).
8. Führe die Maschineninstruktionen für die Anweisung (c) aus.
9. Springe zur Instruktion Nr. 1.
10. ...

Blöcke mit geschweiften Klammern

Beispiel:

```
if ( x > 0 )
 {
   if ( x > 1 )
      System.out.println ( x-1 );
 }
else if ( x < 0 )
 {
   if ( x < -1 )
      System.out.println ( 1-x );
 }
else
   System.out.println ( "0" );
```

⟶ Schreibausgabe nur im Fall

`x>1` oder `x<(-1)` oder `x==0`.

Erläuterungen:

- Geschweifte Klammern fassen mehrere Anweisungen zu einer zusammen.
- Eine einzelne Anweisung kann (ohne Effekt) durchaus ebenfalls in geschweifte Klammern gesetzt werden.
- Will man in einem „`if`“–Zweig, einem zugehörigen „`else`“–Zweig oder einem Schleifenrumpf nicht nur eine, sondern mehrere Anweisungen ausführen lassen, muss man sie in geschweiften Klammern zusammenfassen.
- Nicht zuletzt für die Zuordnung eines „`else`“–Zweigs zum richtigen „`if`“–Konstrukt wie auf der vorherigen Folie sind solche Klammerungen von Bedeutung.

Beispiel zum direkten Vergleich:

```
if ( x > 0 )
   if ( x > 1 )
      System.out.println ( x );
   else
      System.out.println ( x );
```

⟶ Der „else“–Zweig gehört zum zweiten „if“.

⟶ Im Fall „x<=0“ wird nichts geschrieben.

```
if ( x > 0 )
 {
   if ( x > 1 )
      System.out.println ( x );
 }
else
   System.out.println ( x );
```

⟶ Der „else“–Zweig gehört zum ersten „if“.

⟶ Im Fall, dass x>0 und x<=1 ist, wird nichts geschrieben.

Aspekt 3: Methodenaufrufe

- Wird eine Methode aufgerufen, so wird der Programmfluss in diese Methode hineingelenkt.
- Mit „return“ wird der Programmfluss wieder an die aufrufende Stelle zurückgelenkt.
- Bei einer „void“–Methode braucht ganz am Ende des Quelltextes der Methode kein „return“ stehen, und der Programmfluss wird trotzdem zurückgelenkt.

Beispiel I:

```
public void meineMethode ( int n )
 {
   if ( n < 0 )
      return;
   System.out.println (n);
 }
```

Erläuterungen:

- Falls „n“ negativ ist, wird der Programmfluss durch das „return“ sofort an die aufrufende Stelle zurückgelenkt.
- *Ansonsten*: Sobald der Programmfluss am Ende des Quelltextes der Methode ankommt, wird er auch ohne „return“ wieder zurückgelenkt.
- Nichtsdestotrotz dürfte man ein (allerdings überflüssiges) „return“ auch als letzte Anweisung am Ende der Methode einfügen.

Beispiel II:

```
public int meineMethode ( int n )
 {
   if ( n < 0 )
      return -n;
   System.out.println (n);
   return n;
 }
```

Erläuterungen:

- Ist eine Methode nicht „void“, so muss mit „return“ zugleich ein Wert des Datentyps, der vor dem Methodennamen anstelle von „void“ steht, zurückgegeben werden.
- Insbesondere darf hier das „return“ (im Gegensatz zur „void“–Methode auf der letzten Folie) auch am Ende des Quelltextes der Methode nicht fehlen.

Beispiel III:

```
public int meineMethode ( int n )
 {
   System.out.println (n);
   return n;
 }

...

int m = 3;
meineMethode ( m );
```

Erläuterungen:

- Auch eine Methode, die nicht „`void`“ ist, kann wie eine „`void`“–Methode aufgerufen werden.
- Der Rückgabewert geht dann verloren.

Frage: Wozu soll so etwas gut sein?

⟶ Antwort auf der nächsten Folie.

Antwort am Beispiel:

```
boolean fakultaet ( int n )
 {
   if ( n <= 0 )
      return false;
   int fak = 1;
   for ( int i=2; i<=n; i++ )
      fak *= i;
   System.out.println ( fak );
   return true;
 }

...

fakultaet(10);
```

Erläuterungen:

- Vergleiche Folie 240.
- Der Rückgabewert gibt an, ob „`n`“ positiv ist und somit die Fakultät von „`n`“ berechnet werden kann.
- Wenn man aber wie im Beispiel *weiß*, dass „`n`“ positiv ist, braucht man den Rückgabewert nicht.

Übersetzung in Java Byte Code (idealisiert):

- Der Code zu einer Methode wie „`meineMethode`“ oder „`fakultaet`“ bildet einen separaten Codeblock mit fester Anfangsadresse.
- An allen Stellen im Source File, an denen der Compiler einen Aufruf der Methode „`meineMethode`“ findet, setzt er im Code eine unbedingte Sprunganweisung zu dieser Anfangsadresse ein.
- Vor dieser Sprunganweisung setzt der Compiler noch Code ein, mit dem die Parameter der Methode an die Stelle kopiert werden, wo sie von der Methode erwartet werden.

  ⟶ Konkret „`int n`“ in den Beispielen I–IV.
- Bei einer Methode mit Rückgabewert muss am Ende noch der Rückgabewert an die Stelle kopiert werden, an der er im aufrufenden Code erwartet wird.

Problem:

- Eine Methode wie „`meineMethode`“ kann ja durchaus an mehreren Stellen im Code aufgerufen werden.
- Woher „weiß“ die Methode eigentlich, wohin der Programmfluss mit „`return`“ jeweils zurückspringen soll?

Antwort:

- Zusätzlich zu den Parametern bekommt eine Methode eine *Rücksprungadresse* als weitere Information.
- Vor dem Sprung zur Anfangsadresse der Methode setzt der Compiler daher noch zusätzlichen Code ein, mit dem die Rücksprungadresse an der Stelle abgelegt wird, wo sie von der Methode erwartet wird.
- Jedes „`return`“ wird vom Compiler in Instruktionen übersetzt, die diese Adresse lesen und einen unbedingten Sprung dorthin ausführen.

Der Run–Time–Stack

Problem:

- Eine Methode kann intern wieder eine andere Methode aufrufen, die intern wieder eine andere Methode aufruft, die intern wieder eine andere Methode aufruft usw.
- Im allgemeinen „steckt“ der Prozess also in mehreren Methoden gleichzeitig.
- Nur die ganz zuletzt aufgerufene Methode ist allerdings aktiv in Abarbeitung.
- Die anderen Methoden „warten“ darauf, dass sie durch Rücksprung reaktiviert werden.
- Die Daten, die jede dieser Methoden mit der jeweils aufrufenden Codestelle austauscht (also Parameter, Rücksprungadresse und Rückgabewert) müssen nach einem einheitlichen Schema organisiert sein, so dass jede Methode „weiß“, welches genau ihre Daten sind.

Simples Beispiel:

```
int f1 ()
 {
   return 1;
 }

int f2 ()
 {
   return 2;
 }

int f3 ()
 {
   return f2()+f1();
 }
```

Erläuterungen:

- Wenn „`f3`“ aufgerufen wird, gibt es eine kurze Zeitspanne, in der der Prozess in „`f1`“, „`f2`“ und „`f3`“ zugleich „steckt“.
- In dieser Zeitspanne ist aber nur „`f1`“ bzw. „`f2`“ aktiv in Bearbeitung.

Fortsetzung Erläuterungen:

- Zusätzlich steckt der Prozess in dieser Zeitspanne auch in der Methode, die ihrerseits „`f3`“ aufgerufen hat, in der Methode, die die letztere aufgerufen hat usw.
- Anfang dieser Aufrufhierarchie: Eine Methode, die ihrerseits von außerhalb des Programms aufgerufen wird, zum Beispiel:
  - ◇ Methode „`main`“ wird vom Interpreter „`java`“ (vgl. Folie 191 ff.) direkt aufgerufen.
  - ◇ Methode „`paint`“ von Klasse „`Applet`“ und ihren Erweiterungen wird von „`mozilla`“ und „`appletviewer`“ direkt aufgerufen.
- Wenn man innerhalb seiner eigenen Methoden noch vordefinierte Standardmethoden wie z.B. „`System.out.println`“ aufruft, können darin unsichtbar noch etliche weitere Methodenaufrufe stattfinden.

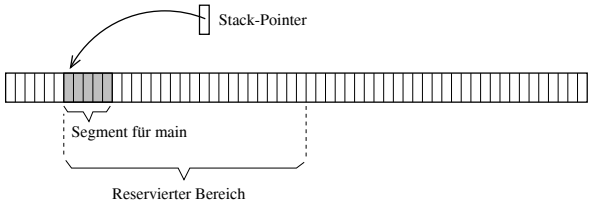
Nun konkret zum Run–Time–Stack:

- Zu jedem Java–Prozess gibt es einen reservierten Speicherbereich, in dem die Parameter, Rücksprungadressen und Rückgabewerte jeder momentan aufgerufenen Methode nach strikten Regeln abgelegt sind.
- Die Daten zu einem einzelnen Methodenaufruf werden in einem zusammenhängenden Segment abgelegt.
- Die Segmente werden in der zeitlichen Reihenfolge der zugehörigen Methodenaufrufe im Speicher abgelegt.
- Die Anfangsadresse des jeweils zuletzt angelegten Segments wird in einem eigens dafür vorgesehenen Register (vgl. Folie 76 ff.) namens *Stack–Pointer* gespeichert.
- Der Compiler fügt vor dem Methodenaufruf noch Code ein, mit dem der Stack–Pointer um die Größe des Segments hochgesetzt wird, und bei „`return`“ entsprechend Code zur Zurücksetzung des Stack–Pointers.

Abarbeitung des Beispiels von Folie 266:

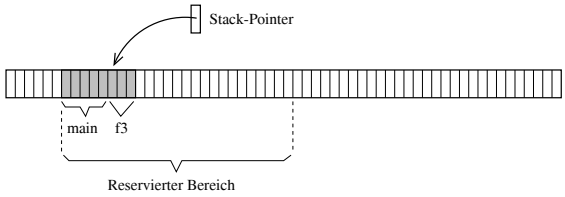
Annahme: „`f3`“ ist direkt von „`main`“ aus aufgerufen worden.

Nach Programmstart:

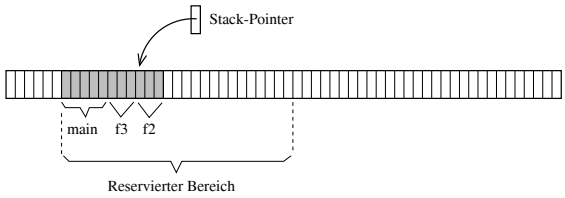


Fortsetzung Abarbeitung des Beispiels:

Nach Aufruf von „f3“ in „main“:

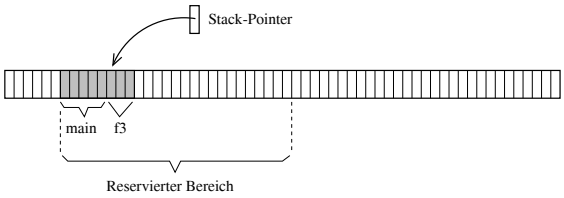


Nach Aufruf von „f2“ in „f3“:

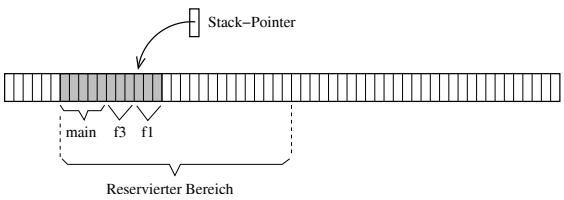


Noch Abarbeitung des Beispiels:

Nach Beendigung von „f2“ in „f3“:

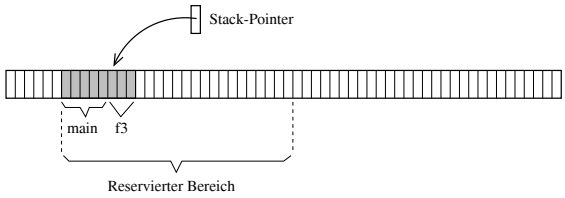


Nach Start von „f1“ in „f3“:

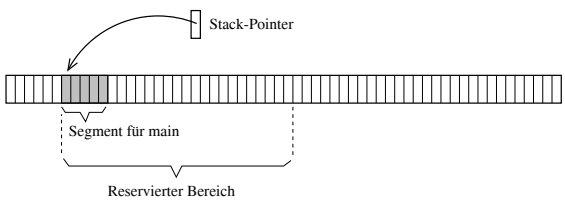


Letzte Folie zur Abarbeitung des Beispiels:

Nach Beendigung von „f1“ in „f3“:



Nach Beendigung von „f3“ in „main“:



What's in a name:

- *Run Time* = Laufzeit (des Programms)
- *Stack* = Stapel

⟶ Der Run–Time–Stack ist ein „Stapel", auf den während der Abarbeitung des Programms durch Methodenaufrufe laufend Segmente draufgepackt und durch „`return`"s wieder weggenommen werden.

Zusammensetzung des Segments für einen Methodenaufruf:

- Ein Stück Speicher für jeden Parameter der Methode.
- Ein Stück Speicher für die Rücksprungadresse.

Beispiel:

```
int f ( char c, double d )
  {
    return c+1;
  }
...

int i = f ( 'a', 3.14 );
```

Inhalt des Segments in diesem Beispiel:

- Ein Speicherstück der Größe eines „`char`" für „`c`".
- Ein Speicherstück der Größe eines „`double`" für „`d`".
- Ein Speicherstück für die Rücksprungadresse.

Umsetzung eines Methodenaufrufs nach Folie 71:

1. Erhöhe den momentanen Wert des Stack–Pointers um die Größe des momentan obenaufliegenden Segments (⟶ Anfang des neuen Segments).
2. Schreibe den ASCII–Wert des Zeichen–Literals „`a`" in das für Parameter „`c`" reservierte Speicherstück im neuen Segment.
3. Schreibe den Wert 3.14 in das für Parameter „`d`" reservierte Speicherstück im neuen Segment.
4. Schreibe die Adresse der nächsten Instruktion, die auf den Methodenaufruf „`f ( 'a', 3.14 );`" folgt, in den für die Rücksprungadresse reservierten Speicherplatz im neuen Segment.

```
int f ( char c, double d )
  {
    return c+1;
  }
...

int i = f ( 'a', 3.14 );
```


Umsetzung des „return"s gemäß Folie 71:

1. Schreibe den Wert von „`c+1`" in ein dafür momentan freigehaltenes Register (vgl. Folie 76).
2. Vermindere den momentanen Wert des Stack–Pointers um die Größe des obenaufliegenden Segments.

   ⟶ Segment ist „aus dem Spiel".

   (Seine Inhalte werden nie wieder angeschaut und bei späterem Bedarf durch Einrichtung neuer Segmente bei Methodenaufrufen überschrieben.)
3. Springe zu der Adresse, die in diesem „aus dem Spiel geratenen" Segment an der für die Rücksprungadresse reservierten Stelle steht.

Instruktion an dieser Rücksprungadresse:

1. Kopiere den Wert aus dem oben genannten Register in die Speicherzelle „`i`".

```
int i = f ( 'a', 3.14 );
```

Rekursion:

```
void meineRekursiveMethode ( int n )
 {
   if ( n < 0 )
     return;
   System.out.print ( n + " " );
   meineRekursiveMethode ( n - 1 );
   System.out.print ( n + " " );
   return;
 }
```

Terminologie:

Eine Methode, die sich selbst aufruft, heißt *rekursiv*.

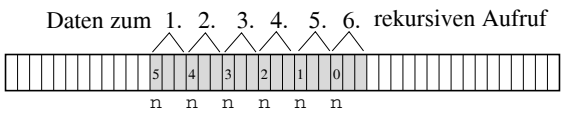
Ergebnis des Aufrufs „`meineRekursiveMethode(5)`“:

```
5 4 3 2 1 0 0 1 2 3 4 5
```

Frage: Wie lässt sich dieses Ergebnis erklären?

Antwort:

- Es gibt zwar im Quelltext von „`meineRekursiveMethode`“ nur eine einzige Variable namens „`n`“.
- Aber durch die Rekursion werden nacheinander mehrere Segmente auf den Run–Time–Stack gebracht, die alle zu Aufrufen von „`meineRekursiveMethode`“ gehören.
- In jedem dieser Segmente findet sich ein Speicherstück für „`n`“.
- Wie bei nichtrekursiven wechselseitigen Methodenaufrufen (Folie 269 ff.) gehört das jeweils als letztes eingerichtete Segment zum momentan aktiv in Bearbeitung stehenden Methodenaufruf.



Sinnvolleres Beispiel für Rekursion:

```
int fakultaet ( int n )
 {
   if ( n == 1 )
      return 1;
   return n * fakultaet (n-1);
 }
```

Erläuterungen:

- Vgl. Folie 240.
- Funktionen wie die Fakultät werden typischerweise rekursiv definiert:
  - $1! = 1$;
  - $n! = n \cdot (n-1)!$ für $n > 1$.
- Die rekursive Methode ist die unmittelbare Umsetzung dieser Definition.
- Achtung: Der Aufrufparameter muß $n \geq 1$ genügen!

Weiteres Beispiel für Rekursion:

```
public int fib(int n)
 {
   if (n<2) return 1;
   else return fib(n-1)+fib(n-2);
 }
```

Erläuterungen:

- Die sogenannten Fibonacci-Zahlen werden rekursiv folgendermaßen definiert:
  - $fib(0) = 1$;
  - $fib(1) = 1$;
  - $fib(n) = fib(n-1) + fib(n-2)$.
- Die rekursive Methode ist die unmittelbare Umsetzung dieser Definition.

What's in a name:

- „Rekursion“ und „rekursiv“ stammen vom lateinischen „recurrere“ = „zurücklaufen“.
- In diesem Kontext könnte man vielleicht sagen: „zu sich selbst zurückkommen“.
- Genau das passiert ja auch bei Rekursion: Die Methode kommt auf sich selbst zurück.

Abschnitt 3.2: Objekte

Aspekt 1: Variable, Konstante, Literale

Beachte: Der Begriff *Variable* bedeutet in Mathematik und Programmierung etwas fundamental verschiedenes!

- *Mathematik*: Eine Variable ist ein Platzhalter für die Elemente einer Menge.

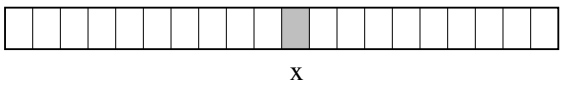
  Beispiel: Für alle reellen Zahlen x (kurz: für $x \in \mathbb{R}$) gilt $x^2 \geq 0$.
- *Programmierung*: Eine Variable ist ein zusätzlicher, symbolischer Name für eine feste Speicheradresse.

Konkretes Beispiel zum Unterschied:

- Die Java–Anweisung

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

  liest den Inhalt der Speicherzelle mit Namen „x“ in das Rechenwerk (Folie 64 ff.), addiert 1 drauf und schreibt das Ergebnis wieder in dieselbe Speicherzelle zurück.



- Mathematisch gesehen wäre $x = x + 1$ dagegen absoluter Nonsens.

Variable vs. Konstante:

- Eine Variable ist wie auf der letzten Folie beschrieben ein symbolischer Name für eine Speicheradresse.

  Beispiel: Eine Zeichenvariable namens „`var`“, die ein Zeichen speichern soll, kann man einrichten und sofort mit dem Zeichen „`a`“ initialisieren durch

  ```
  char var = 'a';
  ```
- Eine Konstante ist im Prinzip dasselbe, nur:
  - ◇ Eine Konstante *muss* sofort initialisiert werden.
  - ◇ Danach darf der Wert der Konstante nicht mehr geändert werden.
- Zur Einrichtung einer Konstante statt Variable schreibt man das Schlüsselwort „`final`“ vorweg:

  ```
  final char var = 'a';
  ```

Frage:

- Mit der Einrichtung von „`var`“ als einer Konstanten anstelle einer Variablen verbaut man sich ja Möglichkeiten.
- Welchen Sinn soll das haben?

Antwort:

- Oft ist ein Objekt von seiner inneren Logik her wirklich konstant.
- *Beispiele*:

```
final float pi            = 3.14159;
final char  dollarzeichen = '$';
```

- Mit „`final`“ kann man verhindern, dass der Wert irgendwo im Source File aus Versehen überschrieben wird.

  ⟶ Fehlermeldung beim Kompilieren.

Konstante vs. Literal:

- *Erinnerung* an Folie 206: Zeichenketten der folgenden Formen in einem Source File:
  - ◇ `69534`
  - ◇ `3.14159`
  - ◇ `'a'`
  - ◇ `''Hello World''`

  sind *keine* Konstanten, sondern *Literale*.
- *Also*:
  - ◇ Eine *Konstante* ist ein Objekt im Hauptspeicher, deren Wert nicht geändert werden kann.
  - ◇ Ein *Literal* ist ein explizit ins Source File hineingeschriebener (und damit natürlich ebenfalls unveränderlicher) Wert.
- *Beispiel*: `final char` $\underbrace{\texttt{var}}_{\text{Konstante}}$ `=` $\underbrace{\texttt{'a'}}_{\text{Literal}}$ `;`

Aspekt 2: Datentypen in Java

- Es gibt zwei Arten von Datentypen in Java:
  - ◇ „eingebaute Typen“: grundlegende Datentypen.
  - ◇ „Bausteintypen“: meist *Klassen* genannt.
- *Ausnahme*: der „Zwitter“ `Array` (später).

Zuerst eingebaute Typen:

- Ganze Zahlen (Folie 15): `int` und andere.
- Zeichen (Folie 17 ff.): `char`.
- Wahrheitswerte `true`/`false`: `boolean`.
- Reelle Zahlen: `float` und `double`.

Altlast aus C:

- Der Datentyp `float` (=floating–point numbers) war eigentlich gedacht als *der* Datentyp für reelle Zahlen schlechthin.
- Der Datentyp `double` (=double precision)
  - ◇ enthält in der Regel mehr Bits als `float` zum Abspeichern reeller Zahlen (in Java generell: 64 Bits für `double`, 32 Bits für `float`),
  - ◇ und war nur für Berechnungen mit besonders kniffligen numerischen Fehlerproblemen vorgesehen.
- Heute ist Speicherplatz kein Problem mehr.

  ⟶ Der Datentyp mit dem weitaus weniger intuitiven Namen „`double`“ hat sich inzwischen als *der* Standardtyp für reelle Zahlen etabliert.

Charakteristik der Datentypen

Eingebaute Typen:

- In diesen Datentypen finden alle konkreten, tatsächlichen Datenmanipulationen statt.

Bausteintypen:

- Wiederverwendbare Bausteine zur Entwicklung größerer Programme mehr durch „Zusammenstecken“ als durch Neuprogrammierung von Grund auf.
- Programmierung mehr auf einer abstrakten, problemorientierten als technischen Ebene durch Einkapselung der technischen Details in Bausteine.

⟶ Siehe Folie 300 ff.

Datenmanipulationen auf eingebauten Typen

Die wesentlichen Operationen I:

- Test auf Gleichheit („==“) oder Ungleichheit („!=“) auf allen Datentypen:

```
int     i = 1;
boolean b = true;
char    c = 'A';

if ( i == 2 )
if ( i != 2 )
if ( b == false )
if ( b != false )
if ( c == 'z' )
if ( c != 'z' )
```


Die wesentlichen Operationen II:

- Größenvergleich auf numerischen Datentypen:

```
int     i = 1;
double  d = 3.14;
char    c = 'a';

if ( i < 2 )
if ( d < 2 )
if ( c < 'k' )
if ( i > 2 )
if ( d > 2 )
if ( c > 'k' )
if ( i <= 2 )     (kleiner oder gleich)
if ( d <= 2 )
if ( c <= 'k' )
if ( i >= 2 )     (größer oder gleich)
if ( d >= 2 )
if ( c >= 'k' )
```

- Arithmetik auf Zahlentypen (vgl. Folie 220 ff.):

```
int i = 1;
int j = 2;
int k = i + 2 * j - 3;
```


Die wesentlichen Operationen III:

- Logische Verknüpfungen auf Datentyp „`boolean`“ (vgl. Folie 2):

```
boolean b1 = ( i < 3 );
boolean b2 = ( i > 1 );

if ( b1 && b2 )     (und)
if ( b1 || b2 )     (inklusiv–oder)
if ( ! b1 )         (negiert)
```

- Zeichenmanipulationen:

```
char c = 'A';
char d;
if ( Character.isLowerCase(c) )
  d = Character.toUpperCase(c);
else
  d = c;
```

⟶ Erläuterungen zu Zeichenmanipulationen auf den nächsten Folien.

Erläuterungen zu Zeichenmanipulationen:

- Das Beispiel zu Zeichenmanipulationen auf der vorherigen Folie tut folgendes:

  Falls momentan ein Kleinbuchstabe in `c` gespeichert ist, soll in `d` der entsprechende Großbuchstabe gespeichert werden, sonst das Zeichen von `c` selbst.

- What's in a name:
  - ◇ Kleinbuchstabe = engl. *lower–case letter*,
  - ◇ Großbuchstabe = engl. *upper–case letter* (auch *capital letter*).
- Die technischen Details sind hinter den einfachen, intuitiven Schnittstellen „`Character.isLowerCase`“ bzw. „`Character.toUpperCase`“ versteckt.

  ⟶ Mehr zu „`Character`“ auf Folie 330.

Fortsetzung Erläuterungen:

- Erinnerung an Folie 17 ff.: Die interne technische Realisierung könnte zum Beispiel so aussehen:
  - ◇ „`Character.isLowerCase`“: Größenvergleich der Unicode–Nummern.

    ⟶ Kann man auch direkt in Java schreiben:

    ```
    if ( c >= 'a' && c <= 'z' )
    ```
  - ◇ „`Character.toUpperCase`“: Ziehe die Unicode–Nummer von `'a'` ab und addiere die Unicode–Nummer von `'A'`.
  - ◇ Kann man fast direkt in Java schreiben:

    ```
    d = (char)(c-'a'+'A')
    ```

    ⟶ Mehr zu dieser merkwürdigen Verkomplizierung mit „`(char)`“ auf den nächsten Folien unter dem Stichwort *Konversionen*.

Konversionen zwischen eingebauten Typen

- Objekte unterschiedlicher numerischer eingebauter Typen können in Zuweisungen, Vergleichen und arithmetischen Operationen direkt miteinander kombiniert werden.
- *Beispiele*:

```
int    i = 1;
double d = 3.14;

double x = i;
double y = i + d;
if ( i < d )
```

- Der eingebaute TypC „`char`“ für Zeichen ist hier mit den Unicode–Nummern als Zahlenwerten mit im Spiel:

```
char c = 'a';
int  i = c + 1;
```


Fortsetzung Konversionen:

- Wenn bei einer solchen Zuweisung der Typ auf der linken Seite nicht jeden Wert darstellen kann, den der Typ auf der rechten Seite darstellen kann, ist man zur eigenen Sicherheit gezwungen explizit hinzuschreiben, dass man die Konversion wirklich will:

```
int  i  = 'a';
char c1 = (char)i;
char c2 = (char)(i+1);
```

  ⟶ Sonst Fehlermeldung beim Kompilieren.

- *Generelle Syntax* (Altlast aus C, nicht aus C++):
  - ◇ Der Zieltyp muss in Klammern vor den zu konvertierenden Ausdruck in Klammern hingeschrieben werden.
  - ◇ Der zu konvertierende Ausdruck muss ebenfalls in Klammern gesetzt werden, falls er nicht einfach aus einem einzelnen Literal oder Identifier besteht.

Noch Fortsetzung Konversionen:

Sichere Konversionen, bei denen der Zieltyp nicht explizit hingeschrieben werden muss:

- `char` $\longrightarrow$ `int` $\longrightarrow$ `float` $\longrightarrow$ `double`,
- alle aus dieser unmittelbaren Beziehung sich ergebenden transitiven Konversionen
- sowie Konversionen mit weiteren, hier nicht behandelten numerischen eingebauten Typen

Anmerkung zu `int` $\longrightarrow$ `float`:

- Die Konversion `int` $\longrightarrow$ `float` ist nicht 100% sicher.
- Bei sehr großen `int`–Zahlen ändert sich der Wert ein wenig bei der Konversion nach `float`.
- Trotz dieser kleinen Unsicherheit wird diese Konversion als sicher erachtet.
- Man kann auch davon ausgehen: Wenn jemand schon einen `int`–Wert in `float` umrechnen lässt, wird ihm die Genauigkeit in den allerletzten Stellen wohl nicht so wichtig sein.

Datentyp von Literalen (vgl. Folie 286):

- Zeichenliterale sind vom Datentyp „`char`“.
- Ganzzahlige Literale sind vom Typ „`int`“.
- Reelle Literale sind vom Typ „`double`“.

$\longrightarrow$ Die folgende Zeile ergibt widersinnigerweise eine Fehlermeldung beim Kompilieren:

```
float f = 3.14;
```

$\longrightarrow$ Man muss daher schreiben:

```
float f = (float)3.14;
```


Zurück zum Ausgangspunkt auf Folie 294:

```
d = (char)(c-'a'+'A');
```

Erläuterungen:

- Arithmetische Operationen sind für „`char`“ eigentlich gar nicht definiert.
- Im obigen Ausdruck werden die drei „`char`“–Werte (eine Variable, zwei Literale) daher implizit zu „`int`“ konvertiert.
- Das Ergebnis solcher arithmetischen Operationen auf „`int`“–Werten ist generell wieder vom Typ „`int`“.
- Um das Ergebnis in einer „`char`“–Variablen abzuspeichern, muss es daher explizit nach „`char`“ konvertiert werden.
- Das ist schlussendlich die Erklärung, warum nicht einfach dastehen darf:

```
d = c - 'a' + 'A';
```


Aspekt 3: Klassen

2. Art von Java–Datentypen:

- Bisher hier etwas salopp „Bausteintypen“ genannt.
- Offizielle Bezeichnung: *Klassen*.

Willkürlich herausgegriffene Beispiele für schon vorgefertigte Bausteine in Java:

- „`String`“: Verwaltung von beliebig langen, aber unveränderlichen Zeichenketten.
- „`StringBuffer`“: Wie „`String`“, aber Zeichenkette im `StringBuffer`–Objekt ist auch veränderbar (z.B. mit „`append`“).
- „`Math`“: Bereitstellung diverser mathematischer Funktionen und Konstanten.
- „`Applet`”: „Urstamm“ aller in Java programmierter Internet–Spielereien (allgemein *Applets* genannt).

Fortsetzung Beispiele für vorgefertige Java–Bausteine:

- „`Window`“: „Urstamm“ aller durch Java–Programme geöffneten Bildschirmfenster.
- „`Frame`“: Eine Erweiterung von „`Window`“, die gegenüber „`Window`“ die zusätzliche Funktionalität eines Top–Level–Windows bereitstellt (vgl. Folie 131).
- „Urstämme“ für diverse Ingredienzen von Fenstern, z.B.
  - ◇ „`Choice`“: Auswahlmenü,
  - ◇ „`Dialog`“: Dialogbox,
  - ◇ „`Button`“: Knopf zum Draufklicken mit dem Mauszeiger,
  - ◇ „`Image`“: Graphik-/Bildelement,
  - ◇ „`TextArea`“: Fläche zur interaktiven Textbearbeitung,
  - ◇ „`Scrollbar`“: Balken zum Verschieben („Scrollen“) des Fensterinhalts.

Frage: Was heißt „Urstamm“?

Antwort am Beispiel der Klasse „`Window`”:

Jedes von einem Java–Programm geöffnete Fenster ist in diesem Programm ein Objekt vom Typ „`Window`“ oder von einem anderen Bausteintyp, der „`Window`“ erweitert (z.B. „`Frame`“).

Anschlussfrage: Was heißt „erweitern“?

Antwort am selben Beispiel: Bspw. Einfügen einer Leiste von Buttons oder Menüs. Solche Buttons/Menüs sind Objekte, die

- im Fensterobjekt intern gespeichert und verwaltet werden und
- vom „`Button`“/„`Choice`“–Typ sind oder von Bausteintypen, die „`Button`“ bzw. „`Choice`“ wiederum erweitern.

⟶ Man muss sich nur um die Besonderheiten der eigenen Windows kümmern, nicht um das, was alle Windows gemeinsam haben.

Beispiel aus den Übungen: Applet

- Die Klassen namens „`DemoApplet`“ u.ä. aus den Übungen haben die Klasse „`Applet`“ als Urstamm.

  ⟶ Erkennbar an der „`extends`“–Klausel:

```
public class DemoApplet extends Applet
                                ^^^^^^^^^^^^^^^
```

- Aber eigentlich wurde die Klasse „`Applet`“ dabei gar nicht im Sinne der vorherigen Folie erweitert.
- Statt dessen wurde ihr *Verhalten* erweitert:
  - ◇ Die Methode „`paint`“ ist zwar schon für die Klasse „`Applet`“ definiert,
  - ◇ tut aber eigentlich gar nichts weiter,
  - ◇ man kann sie aber in erweiternden Klassen wie „`DemoApplet`“ mit künstlerischem Inhalt füllen,
  - ◇ und genau das war Inhalt diverser Übungsaufgaben.

  ⟶ Mehr zu solchen „Erweiterungen durch Verhaltensänderung“ auf Folie 489 ff.

Nachbemerkung:

- Eigentlich sollte in den Übungen oft nicht die Methode „`paint`“ mit Inhalt gefüllt werden,
- sondern eine Methode namens „`realPaint`“,
- und der wesentliche Inhalt von „`paint`“ selbst war jeweils ein Aufruf von „`realPaint`“.

Erklärung:

- Die Methode „`paint`“ enthält in manchen Fällen erzwungenermaßen einige technische Details, die in der Vorlesung und den Übungen erst später behandelt werden.
- Die Methode „`realPaint`“ schafft eine klar umgrenzte „Spielwiese“ für Sie, die von solchen Details freigehalten worden ist.

Aspekt 4: Objekte und Referenzen

- Mit dem Namen eines Objekts von einem eingebauten Typ spricht man das Objekt unmittelbar an.
- Bei Bausteintypen ist der Name des Objekts nur als eine *Referenz* (d.h. Verweis) auf ein anonymes Objekt vom Bausteintyp zu verstehen.
- Ausnahmsweise keine Altlast von C/C++, sondern
  - ◇ im wesentlichen ein Kompromiss aus Flexibilität und Programmeffizienz,
  - ◇ der sich auch in einigen anderen moderneren Programmiersprachen findet,
  - ◇ und zwar aus derselben Motivation heraus.

⟶ Mehr dazu beim Stichwort Polymorphie (Folie 489 ff.).

1. Konsequenz:

Objekte von Bausteintypen müssen erst noch umständlich mit „`new`“ kreiert werden.

Beispiel:

- `int i = 1;`

  Objekt „`i`“ ist schon fertig konstruiert und mit Wert `1` initialisiert.
- `String str;`

  Nur die **Referenz** „`str`“ auf eine Zeichenkette ist eingerichtet worden, aber damit ist noch keine Zeichenkette eingerichtet worden.
- `String str ( "Hello" );`

  Fehler: „`str`“ ist keine Zeichenkette, nur eine Referenz auf eine Zeichenkette.
- `String str = new String ( "Hello" );`

  Korrekt: Referenz „`str`“ ist eingerichtet und mit einem `String`–Literal des Inhalts „`Hello`“ initialisiert worden.

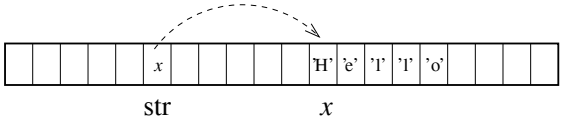
Zur Veranschaulichung: Schematische Darstellung der Umsetzung im Speicher

- `int i = 1;`

  Den Bezeichner „`i`“ kann man sich vorstellen als einen symbolischen Namen für die Speicheradresse, in der der Wert der Variable abgespeichert ist.

- `String str = new String ( "Hello" );`

  Der Bezeichner „`str`“ ist eher ein symbolischer Name für eine Speicheradresse, deren Inhalt die *Anfangsadresse* x der gespeicherten Zeichenkette ist.



Kleine Feinheit:

- Wie auf Folie 306 gesagt, ist folgendes falsch:

  ```
  String str ( "Hello" );
  ```

- Folgendes ist aber korrekt:

  ```
  String str = "Hello";
  ```

- *Hintergrund*: „`String`“ ist eine der wichtigsten Klassen überhaupt in Java.
- Aus diesem Grund hat man sich entschieden, einige der wichtigsten Operationen auf Strings
  - ◇ nicht nur mit der üblichen Syntax für solche Operationen zu realisieren,
  - ◇ sondern auch noch einmal mit einer wesentlich einfacheren und bequemeren Syntax.

Fortsetzung kleine Feinheit:

- Das obige

```
String str = "Hello";
```

ist ein solches Beispiel. Es stellt eine Abkürzung der folgenden Anweisung dar:

```
String str = new String("Hello");
```

- Weiteres Beispiel: Operator „+“ für Strings:

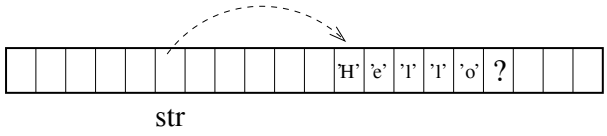
```
System.out.println ( a + " " + b );
```

⟶ Operator „+“ gibt es für Klassen ja eigentlich gar nicht. Angewendet auf zwei Strings hängt „+“ diese aneinander.

Exkurs: Interne Speicherung von Strings

Frage:

- Die Zeichen in `"Hello"` sind ja im Speicher einfach nur beliebige Bitmuster (siehe Folie 17 ff.).
- Auch die Bitmuster im Maschinenwort unmittelbar nach dem Ende der Zeichenkette könnten prinzipiell als Zeichen interpretierbar sein.
- Woher „weiß“ ein `String`–Objekt „`str`“ eigentlich, wo genau „seine“ Zeichenkette genau aufhört?

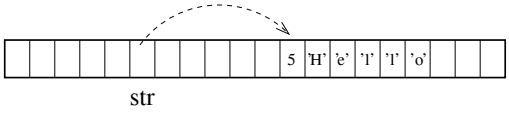


Antwort:

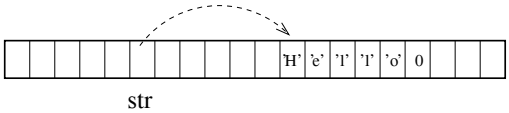
Mindestens zwei grundlegend verschiedene Möglichkeiten.

Nämlich:

- Vorweg wird die Anzahl der Zeichen gespeichert.



- An jede Zeichenkette wird als Begrenzungsanzeiger ein fest gewähltes „unmögliches“ Zeichen angehängt (z.B. Unicode–Wert 0, vgl. Folie 31).



Sichtbarkeit in Programmiersprachen:

- In C/C++ ist Begrenzung durch ASCII/Unicode–Wert 0 als Strategie festgelegt und muss beim Programmieren beachtet werden.
- In abstrakteren Sprachen wie Java sind das alles intern verwaltete technische Details hinter der Fassade von „`String`“, die der Programmierer gar nicht zu sehen bekommt.

Initialisierung:

```
int     i1 = 1;
int     i2;

String  str1 = new String ("Hello");
String  str2;
```

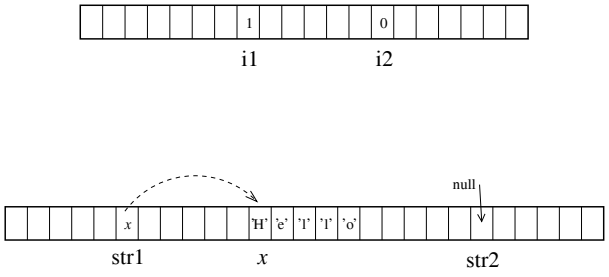
Frage:

- Es ist klar, dass das Objekt „`i1`“ den Wert 1 enthält und „`str1`“ auf ein (anonymes) Objekt mit Inhalt „Hello“ verweist.
- Was ist aber mit „`i2`“ bzw. „`str2`“.

Generelle Antwort:

- Zu jedem Datentyp gibt es einen „Nullwert“.
- Wenn ein Objekt des Datentyps bei seiner Einrichtung nicht explizit initialisiert wird, dann wird das Objekt mit dem zugehörigen Nullwert initialisiert.
- Nullwerte für eingebaute:
  - ◇ Numerische Typen: Wert 0.
  - ◇ Zeichentyp „`char`“: Das „Nichtzeichen“ mit Unicode–Wert 0.
  - ◇ Logiktyp „`boolean`“: Wert „`false`“.
- Für alle Klassen gibt es einen einzelnen symbolischen Referenzwert mit Namen „`null`“:
  - ◇ keine wirkliche Referenz auf irgendein Objekt,
  - ◇ sondern ein „unmöglicher“ Wert,
  - ◇ der nur anzeigt, dass die Variable momentan auf kein Objekt verweist.

Veranschaulichung am Beispiel von Folie 313:



Konsequenzen:

- Es gibt grundsätzlich kein uninitialisiertes Objekt in Java.

  $\longrightarrow$ Eine wichtige Quelle für undefiniertes Programmverhalten (vgl. Folie 230) ist eliminiert.

  $\longrightarrow$ Im Gegensatz zu anderen Programmiersprachen wie C und C++.
- *Allerdings*:
  - ◇ Wenn eine Klassenvariable den Wert „`null`“ hat
  - ◇ und man trotzdem auf das dahinterstehende Objekt,
  - ◇ bzw. auf das eben **nicht** dahinterstehende Objekt zugreift,
  - ◇ dann stürzt das Programm ab:

```
StringBuffer str;       // == null
str.append ( "bla" );   // Absturz!
```

Beachte:

- Dieser Absturz ist kein undefiniertes Programmverhalten gemäß Folie 230,
- sondern es ist „garantiert“, dass das Programm sofort abstürzt.

⟶ Damit ist immerhin garantiert, dass das Programm keinen weiteren Schaden anrichtet.

Vorgreifende Bemerkungen:

- Durch geeignete Java–Konstrukte kann man einen solchen Programmabsturz auch abfangen und behandeln.

  ⟶ Stichwort *Exceptions* (Folie 552 ff.).
- Die Initialisierung von Objekten einer selbstgebastelten Klasse kann man auch selbst programmieren.

  ⟶ Stichwort *Konstruktoren* (Folie 441 ff.).

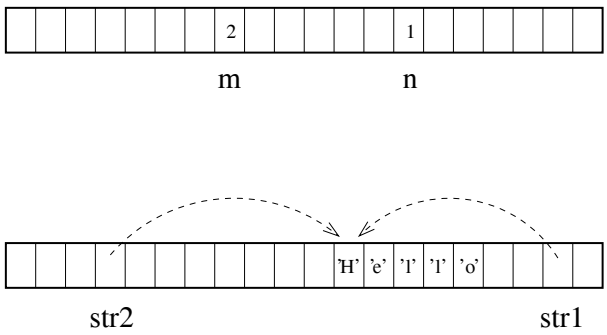
2. Konsequenz:

Zuweisung bei Bausteintypen bedeutet, dass nun zwei Referenzen auf dasselbe anonyme Objekt verweisen!

Beispiel:

```
int n = 1;
int m = n;
m++;
System.out.print (m);   // Ausgabe: "2"
System.out.print (n);   // Ausgabe: "1"

StringBuffer str1
   = new StringBuffer ( "Hello" );
StringBuffer str2 = str1;
str2.append ( ", World" );
System.out.print ( str2 );
    // Ausgabe: "Hello, World"
System.out.print ( str1 );
    // Ausgabe: "Hello, World"!!!
```


Veranschaulichung der 2. Konsequenz am Beispiel von der letzten Folie:



Beachte:

- Eine Methode einer Klasse wird zwar mit dem Namen einer Variablen dieser Klasse aufgerufen.
- Aber sie macht eigentlich gar nichts mit dieser Variable.
- Statt dessen macht sie etwas mit dem *Objekt*, auf das diese Variable verweist.
- Wenn zwei Variable einer Klasse auf dasselbe Objekt verweisen, ist es also logisch, dass der Effekt eines Methodenaufrufs (z.B. „`append`“) mit einer Variablen („`str2`“) zugleich über die andere Variable („`str1`“) sichtbar wird.

3. Konsequenz:

- Test mit „==“ auf Gleichheit bedeutet bei Klassentypen
  - ◇ nicht Test auf Wertgleichheit wie bei eingebauten Typen,
  - ◇ sondern Test auf Objektidentität!
- Jede vorgefertigte Klasse in der Java–Standardbibliothek besitzt für den Test auf Wertgleichheit eine Methode namens „`equals`“ mit
  - ◇ einem Argument, dem zu vergleichenden Objekt, und
  - ◇ Rückgabetyp „`boolean`“

    ⟶ Also „`true`“/„`false`“.

Beispiel zur 3. Konsequenz:

```
String  str1 = new String ( "Hello" );
String  str2 = str1;
String  str3 = new String ( "Hello" );
if ( str1 == str2 )
   System.out.println ( "Ja" );
else
   System.out.println ( "Nein" );
// Ausgabe: "Ja"
if ( str1.equals(str2) )
   System.out.println ( "Ja" );
else
   System.out.println ( "Nein" );
// Ausgabe: "Ja"
if ( str1 == str3 )
   System.out.println ( "Ja" );
else
   System.out.println ( "Nein" );
// Ausgabe: "Nein"!!!
if ( str1.equals(str3) )
   System.out.println ( "Ja" );
else
   System.out.println ( "Nein" );
// Ausgabe: "Ja"
```

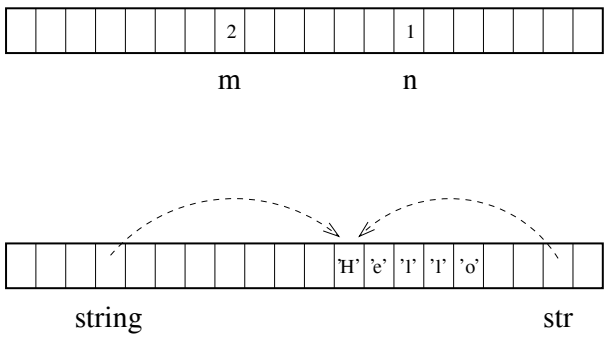

4. Konsequenz:

Argumente von Methoden haben unterschiedliche Bedeutung für eingebaute Typen und Bausteintypen.

Simples Beispiel:

```
void f ( int m, StringBuffer string )
 {
   m++;
   string.append ( ", World" );
 }
...

int n = 1;
StringBuffer str
   = new StringBuffer ( "Hello" );

f ( n, str );

System.out.print (n);
    // -> "1"
System.out.print (str);
    // -> "Hello, World"
```


Veranschaulichung der 4. Konsequenz am Beispiel der letzten Folie (vgl. Folie 319):



5. Konsequenz:

Rückgabewerte von Methoden haben unterschiedliche Bedeutung für eingebaute Typen und Bausteintypen.

Betrachte als Beispiel dazu folgende zwei Methoden:

```
int f1 ( int n )
 {
   return n;
 }

StringBuffer f2 ( StringBuffer string )
 {
   return string;
 }
```

⟶ Verwendung auf der nächsten Folie.

Verwendung der beiden Methoden:

```
int          m1  = 1;
StringBuffer str1 = new StringBuffer ( "Hello" );

int          m2  = f1 ( m1 );
StringBuffer str2 = f2 ( str1 );

m1++;
str1.append ( ", World" );

System.out.print ( m1 );    // -> "2"
System.out.print ( m2 );    // -> "1"

System.out.print ( str1 );  // -> "Hello, World"
System.out.print ( str2 );  // -> "Hello, World"
```


6. Konsequenz:

Wenn eine Methode für eine Variable des entsprechenden Klassentyps aufgerufen wird, dann hat das nur einen Effekt auf das dahinterstehende Objekt, nicht auf die Variable. Wenn der Wert der Variable „`null`“ ist, dann stuerzt das Programm ab (vgl. Abschnitt 4.5, Folie 552 ff.).

```
void f ( int m, StringBuffer sb )
 {
   m++;
   sb.append ( ", World" );
 }
```

Der Aufruf der Methode „`append`“ hat nur Auswirkungen auf das Objekt, auf das „`sb`“ verweist.

Bindeglieder zwischen eingebauten Typen und Klassen

Beispiel:

```
java.lang.Integer x
   = new java.lang.Integer (1);
int n = x.intValue ();

System.out.print (n);   // -> "1"
```

Erläuterung:

- Zu jedem eingebauten Typ gibt es eine spezifische Klasse („Wrappertyp“).
- Ein Wert des eingebauten Typs kann in ein Objekt des zugehörigen Wrappertyps „eingepackt“ (engl. „to wrap“) und „herumtransportiert“ werden.
- Der Wrappertyp bietet eine Methode `intValue`, mit dem der momentane Wert des eingepackten Wertes abgefragt werden kann.

Fortsetzung Beispiel:

```
String  str1 = new String ( "7654" ); // (a)
Integer x    = new Integer ( str1 );
String  str2 = x.toString();          // (b)
double  y    = x.doubleValue();       // (c)
```

Erläuterung:

Unter anderem bietet die Klasse `java.lang.Integer` auch Möglichkeiten, ganze Zahlen

(a) aus Zeichenketten, die Dezimalzahlen darstellen, zu konstruieren,

(b) in ebensolche Zeichenketten zu transformieren und

(c) in andere numerische Datentypen (z.B. `double`) zu konvertieren.

⟶ Einfaches Beispiel für die Einkapselung der technischen Aspekte in eine abstraktere Komponente gemäß Folie 289.

Übersicht über Wrapper (nicht 100% vollständig):

eingebauter Typ	Wrappertyp
`int`	`java.lang.Integer`
`char`	`java.lang.Character`
`float`	`java.lang.Float`
`double`	`java.lang.Double`
`boolean`	`java.lang.Boolean`

Bemerkung:

Auf Folie 293 ff. wurden weitere spezifische Funktionalitäten von „`Character`“ verwendet.

Aspekt 5: Zusammengesetzte Objekte

- Objekte von eingebauten Typen sind aus Sicht eines Java–Programmierers *atomar*.
- *Das heißt*: Eine etwaige weitere interne Struktur und Zerlegbarkeit eines solchen Objekts auf Maschinenebene wird im Java–Quelltext nicht sichtbar.
- *Ausnahme*: Es gibt Operatoren in Java zur logischen Verknüpfung von Zahlen „Bit–für–Bit“.

  ⟶ Betrachten wir in dieser Veranstaltung nicht (für Interessierte: Stichwort *Bitlogik*).
- Objekte von Klassen sind hingegen im allgemeinen aus mehreren Variablen von eingebauten Typen und/oder Klassen zusammengesetzt.

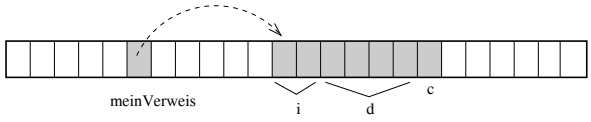
Einfaches Beispiel:

```
public class MeineKlasse
 {
   public int    i;
   public double d;
   public char   c;
 }
...

MeineKlasse meinVerweis = new MeineKlasse();
```

Erste, einführende Erläuterung:

- Mit obigem Code ist eine Klasse namens „`MeineKlasse`“ definiert und sogleich eine Variable „`meinVerweis`“ von „`MeineKlasse`“ mit dahinterstehendem Objekt angelegt worden.
- Jedes Objekt der Klasse „`MeineKlasse`“ ist aus einem „`int`“–Objekt, einem „`double`“–Objekt und einem „`char`“–Objekt zusammengesetzt.
- Die genauen Details der Syntax (insbesondere der Sinn von „`public`“) werden erst später in der Vorlesung behandelt (⟶ Folie 467 ff.).

Veranschaulichung:



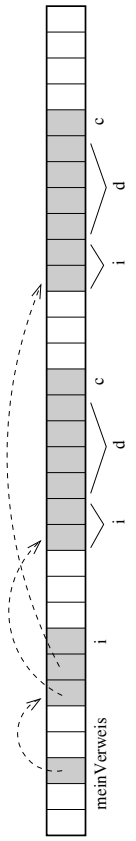
Erläuterungen:

- *Erinnerung* an Folie 305: Variablen von Klassen sind nur Verweise auf die eigentlichen (anonymen) Klassenobjekte.
- Die drei Objekte von eingebauten Typen, die im Objekt hinter „`meinVerweis`“ zusammengefaßt sind, werden mit „`meinVerweis.i`“, `meinVerweis.d`“ und „`meinVerweis.c`“ angesprochen.
- Zur Namensgebung für „`MeineKlasse`“ bzw. „`meinVerweis`“ vergleiche auch Folie 202 ff.

Klassen mit Klassenkomponenten:

```
public class ErsteKlasse
 {
   public int    i;
   public double d;
   public char   c;
 }

public class ZweiteKlasse
 {
   public ErsteKlasse verweis1;
   public ErsteKlasse verweis2;
   public int         i;
 }

...

ZweiteKlasse meinVerweis
   = new ZweiteKlasse();
meinVerweis.verweis1 = new  ErsteKlasse();
meinVerweis.verweis2 = new  ErsteKlasse();
```


Veranschaulichung:



⟶ Zum Beispiel greift `meinVerweis.verweis2.c` in dieser Schemazeichnung auf die rechteste graue Speicherzelle zu.

Aspekt 6: Arrays

- Eine spezielle Familie von Datentypen.
- Im Gegensatz zu den bisherigen eingebauten Typen sind Objekte von Arrays zusammengesetzt.
- *Hauptunterschied* zu zusammengesetzten Klassenobjekten:
  - ◇ Die in einem Klassenobjekt zusammengefassten Objekte können von unterschiedlichen Typen sein und werden mit symbolischen Namen (d.h. Identifiern) angesprochen.
  - ◇ Die in einem Arrayobjekt zusammengefassten Objekte müssen alle von demselben Typ sein (dem *Elementtyp* des Arrayobjekts) und werden mit ganzzahligen *Indizes* angesprochen.

Achtung:
In gewisser Weise sind die Arraytypen ein „Zwitter“ zwischen eingebauten Typen und Klassen.

Das heißt:

- Eigentlich ist ein Arrayobjekt ein eingebauter Typ wie in anderen Programmiersprachen auch.
- *Aber*: Der Name eines Arrays ist wie bei Bausteintypen nur eine Referenz.

Konsequenz: Dieselben sechs Konsequenzen (Folie 306 ff.) wie bei Klassen.

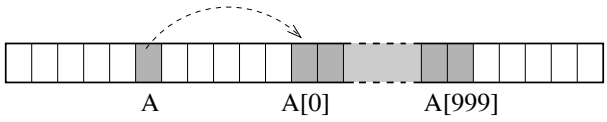
Beispiel:

```
int[] A = new int [1000];
  // Arrayobjekt erzeugt wie bei
  // Klassenobjekten

sort (A);
  // 'A' wird wie Klassenobjekt als Referenz
  // uebergeben und kann daher in 'sort'
  // veraendert (z.B. sortiert) werden.
```


Details zu Arrays:

- Zu jedem beliebigen eingebauten- und Klassentyp kann man Arrays mit diesem Typ als Elementtyp einrichten.
- Altlast aus C/C++: Der Indexbereich eines Arrays mit n Elementen ist immer das Intervall
$$[0, 1, 2, \ldots, n-1].$$
- Syntax zum Ansprechen eines einzelnen Elements eines Arrayobjekts: „`A[i]`“.

  $\longrightarrow$ Dieser Ausdruck spricht das Element mit Index „`i`“ (also das $(i+1)$–te!) im Arrayobjekt namens „`A`“ an.



Achtung:

- Es können grundsätzlich keine neuen Elemente in ein Arrayobjekt eingefügt oder Elemente aus einem Arrayobjekt entfernt werden.

  $\longrightarrow$ Der Indexbereich eines Arrayobjekts ist unveränderlich.
- Die Elemente des Arrays können also wie andere Variable auch **nur** gelesen und überschrieben werden:

```
int[]  A = new int [1000];
A[123] = 1;
System.out.print ( A[123] );  // -> "1"
```


Vorgreifende Bemerkung:

Es gibt in Java zusätzlich noch eine Klasse namens „`java.util.Vector`“, die

- dieselbe Funktionalität wie Arrays bietet,
- aber zusätzlich noch das Einfügen und Löschen und einige weitere bequeme Zusatzfunktionalität bietet.

Nachteile:

- Höhere Laufzeit für die einzelnen Komponentenzugriffe.
- Die Typprüfung des Java–Compilers muss bis zu einem gewissen Grad umgangen werden.

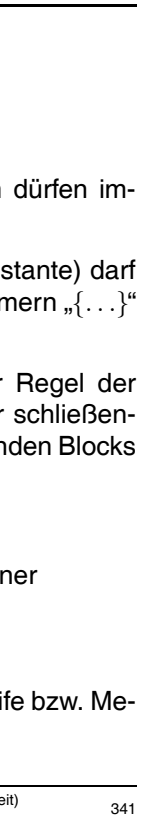
  $\longrightarrow$ Mehr dazu auf Folie 547.

Aspekt 7: Scope und Lebenszeit

- *Erinnerung* an Folie 212: Klammern dürfen immer nur strikt paarweise auftreten.
- *Grundregel*: Eine Variable (bzw. Konstante) darf nur innerhalb von geschweiften Klammern „{...}“ deklariert werden.
- Der *Scope* einer Variable ist in der Regel der Bereich von ihrer Deklaration bis zur schließenden Klammer des ersten umschließenden Blocks „{...}“.
- *Wichtigste Ausnahmen*: Der Scope
  - ◇ einer Variable deklariert im Kopf einer „`for`“–Schleife oder
  - ◇ einem Parameter einer Methode

  endet mit dem Ende der „`for`“–Schleife bzw. Methode.

Schematisches Beispiel:



Schematisches Beispiel mit „for“:

Schematisches Beispiel mit einer Methode:

Beispiel:

```
public void F ( int a )
 {
   int b = 1;
   if ( a == b )
    {
      for ( int i=0; i<10; i++ )
       {
         int c = 2;
       }            // <- Scope-Ende von 'c'
                    //               und 'i'
      int d = 3;
      {
        int e = 4;
      }             // <- Scope-Ende von 'e'
      int f = 5;
      {
        int g = 4;
      }             // <- Scope-Ende von 'g'
    }               // <- Scope-Ende von 'd'
                    //               und 'f'
 }                  // <- Scope-Ende von 'a'
                    //               und 'b'
```


Verhältnis von Scope und Lebenszeit:

- Eine Variable existiert, solange die Abarbeitung des Programms in ihrem Scope ist.
- Eine Komponente eines Array- oder Klassenobjektes existiert, solange das Gesamtobjekt existiert.

Achtung:

- Wenn der Scope einer Variablen verlassen und wieder betreten wird, wird die Variable
  - ◇ nicht nur wieder eingerichtet,
  - ◇ sondern auch neu initialisiert.

  $\longrightarrow$ Der alte Wert, den die Variable beim Verlassen des Scopes hatte, ist verloren.
- Das Objekt, auf das eine Variable eines Klassentyps verweist, kann durchaus länger als die Variable selbst leben.

Beispiel für 1. „Achtung“:

```
for ( int i=0; i<10; i++ )
 {
   int a = 1;
   meineKlasse meinVerweis = new meineKlasse();
   a++;
   meinVerweis.i++;
   System.out.println(meinVerweis.i);
 }
System.out.println(meinVerweis.i);
// Fehlermeldung des Compilers!
```

Erläuterung:

- *Erinnerung* an Folie 313 ff.: Die Variable „`meinVerweis.i`“ im Beispiel oben wird mit Wert `0` initialisiert.
- Diese implizite Initialisierung von „`meinVerweis.i`“ sowie die explizite Initialisierung von „`a`“ werden jedesmal aufs Neue ausgeführt, wenn die Schleife durchlaufen wird.
- Die Werte `a==2` und `meinVerweis.i==1` am Ende jedes Durchlaufs haben keinerlei Einfluss auf diese Setzungen am Anfang des jeweils nächsten Durchlaufs.

Noch ein kleines Beispiel dazu:

```
public void f ()
 {
   meineKlasse meinVerweis = new meineKlasse();
   // meinVerweis.i == 0
   System.out.print (meinVerweis.i);
   meinVerweis.i++;
   System.out.print (meinVerweis.i);
 }

...

f();     // "01"
f();     // "01", nicht "12"!
```


Beispiel für 2. „Achtung“:

```
public String englischerNikolaus ()
 {
   String str = new String ( "Santa Claus" );
   return str;
 }

...

String nicksName = englischerNikolaus();
System.out.println ( nicksName );
                 // ^^^^^^^^^^ "Santa Claus"
```

Erläuterungen:

- Die Zeichenkette „`Santa Claus`“ ist zwar über die Variable „`str`“ erzeugt worden,
- und die Variable „`str`“ beendet ihre Existenz mit dem Ende der Abarbeitung der Methode „`englischerNikolaus`“,
- aber die Zeichenkette existiert darüber hinaus.

Aspekt 8: Referenzen und Garbage Collection

Erinnerung:

- Von Folie 305: Variablen von Klassen
  - ◇ bezeichnen nicht die Objekte selbst,
  - ◇ sondern nur *Referenzen* auf die eigentlichen Objekte,
  - ◇ und die eigentlichen Objekte müssen mit „`new`“ erst noch explizit angelegt werden.
- Von Folie 193: Bei der Abarbeitung eines Programms arbeitet im Hintergrund immer ein Laufzeitsystem mit.

Weitere Aufgabe des Laufzeitsystems:

- Verwaltung eines „Pools“ von Speicherplatz.
- Jedes „`new`“ ist eine Anfrage an diese Poolverwaltung.

Abarbeitung von „new“:

- Ein Ausdruck „`new` *X* `...`“ liefert als Rückgabe einen Verweis auf ein Objekt der Klasse „*X*“ zurück.
- Falls die Poolverwaltung momentan ausreichend Speicherplatz zur Verfügung hat,
  - ◇ wird wie gewünscht ein neues Objekt erzeugt,
  - ◇ seine Adresse wird als Wert des „`new`“–Ausdrucks zurückgeliefert
  - ◇ und kann daher mit Operator „=“ einer Variablen der zugehörigen Klasse zugewiesen werden.
- Falls der Speicherplatz hingegen **nicht** ausreicht, tritt ein Fehlermechanismus in Aktion.
- Nach momentanem Stand der Vorlesung bedeutet das: unvermeidbarer Programmabsturz.
- *Später* in Vorlesung und Übungen: Vermeidung des Programmabsturzes durch *Exceptions* (Abschnitt 4.5, Folie 552 ff.).

Problem:

```
String str1 = new String ( "Hallo" );
String str2 = str1;
...
str1 = new String ( "Holla" );
str2 = str1;
```

Frage:

- Am Schluss gibt es keinen Verweis auf die zuerst erzeugte Zeichenkette „`Hallo`“ mehr.
- Beide Variable, die zuerst auf diese Zeichenkette verwiesen haben, sind ja später auf die Adresse einer anderen Zeichenkette umgesetzt worden.
- Der für „`Hallo`“ reservierte Speicherplatz ist nun vom Programm aus nicht einmal mehr erreichbar und daher völlig nutzlos.
- Er steht dem Laufzeitsystem aber nicht für die Bedienung weiterer Anfragen mit „`new`“ zur Verfügung.

Krasses Beispiel:

```
String str;
while ( true )
   str = new String ("Hallo");
```

Erläuterung:

- Mit „`while`“ wird bekanntlich eine Schleife eingeleitet, die solange durchlaufen wird, bis die logische Bedingung in Klammern falsch (`==false`) wird.
- Das Literal „`true`“ wird natürlich niemals falsch.
  ⟶ Endlosschleife.
- Es wird also endlos neuer Speicherplatz eingerichtet.

Frage:

Ist Programmabsturz damit nicht vorprogrammiert?

1. mögliche Gegenstrategie:

- Neben „`new`“–Anweisungen gibt es noch ein weiteres Konstrukt, um Speicherplatz wieder an die Poolverwaltung zurückzugeben.
- Zum Beispiel „`delete`“ in C++.

  Beispielhafter C++–Code:

```
char* str;          // Referenzvariable ("Pointer")
str = new char[100]; // Neuer Speicherplatz fuer 100 Zeichen
...
delete[] str;       // Wieder freigegeben
```

- Ähnliche Konstrukte gibt es in Pascal, C, Ada...
- Aber zum Beispiel nicht in Java!

Problem mit der 1. Gegenstrategie:

- In komplexeren Programmen
  - ◇ können ein „`new`“ und das zugehörige „`delete`“ potentiell sehr weit auseinanderliegen,
  - ◇ oft genug in verschiedenen Source Files,
  - ◇ die vielleicht sogar von verschiedenen Programmierern geschrieben worden sind.
- *Praktisch unvermeidliches Resultat*: Schwer zu findende Programmierfehler mit beliebig üblen Konsequenzen.
  - ◇ Das „`delete`“ wird schlicht vergessen.
    ⟶ Wenn oft genug Speicherplatz angefordert, aber nicht zurückgegeben wird, geht irgendwann gar nichts mehr.
  - ◇ Ein Stück Speicherplatz, das mit „`delete`“ wieder freigegeben (und vielleicht schon weiterverwendet!) wurde, wird aus Versehen weiter benutzt oder ein weiteres Mal mit „`delete`“ freigegeben.
    ⟶ Programmabsturz wäre nicht das Schlimmste, was passieren könnte...

Strategie in Java und einigen anderen Programmiersprachen:

- Das Laufzeitsystem startet hin und wieder im Hintergrund einen zusätzlichen Prozess, der
  - ◇ alle momentan reservierten Speicherbereiche absucht, ob sie vom Programm über Referenzen überhaupt noch erreichbar sind und
  - ◇ jedes als nicht mehr erreichbar klassifizierte Stück Speicherplatz an die Poolverwaltung zurückgibt.
- Stichwort in der Literatur: *Garbage Collection*.
- *Ergebnis*: Das Problem von Folie 353 ist fast gelöst.
- Warum nur fast: Man kann natürlich immer noch zuviel Speicherplatz anlegen, ohne dass ein einziges Byte davon unerreichbar wird.
  ⟶ Siehe Folie 372.

Frage:

- Wenn ein mit „`new`“ erzeugtes Objekt über den Scope der auf ihn verweisenden Variablen hinaus existiert,
- wie lange existiert es dann eigentlich?

Erinnerung an Folie 348 ff.:

- Das Objekt existiert mindestens noch solange, wie es eine „Kette“ von Verweisen gibt, über die man das Objekt vom Programm aus ansprechen kann.
- Wenn die letzte solche Kette „abreißt“, existiert das Objekt zunächst einmal weiter.
- Erst wenn der *Garbage Collector* das nächste Mal aktiv wird, vernichtet er das Objekt.
- Das passiert zu einem Zeitpunkt den der Java–Programmierer weder vorhersehen noch beeinflussen kann.

Abschnitt 3.3: Rekursive Datentypen

```
class Element
 {
   int      info;
   Element  naechster;
 }

...

Element element   = new Element();

element.info = 1;
element.naechster = new Element();
element.naechster.info = 2;
```

Erläuterung:

- Objekte einer Klasse können auch Verweise auf Objekte derselben Klasse als Variable enthalten.
- Eine solche Klasse nennt man *rekursiv*.
  ⟶ Hat nicht direkt mit Folie 281 ff. zu tun!

Wofür braucht man rekursive Datentypen:

- Man kann Mengen von unvorhergesehener Größe damit modellieren.

  ⟶ Auch Klasse „`java.util.Vector`“ (Folie 340) benutzt intern rekursive Datentypen.

  ⟶ Allerdings etwas komplizierter als hier betrachtet.

- Man kann damit generell Strukturen bauen aus Objekten, die irgendwie „in Beziehung“ zueinander stehen.

Zunächst einmal Mengen von „int“:

```
public class Menge
{
  private Element erstesElement;

  public int     groesse     ( )              { ... }
  public boolean istEnthalten ( int info )    { ... }
  public boolean fuegeEin    ( int info )     { ... }
  public boolean entferne    ( int info )     { ... }
}
```

⟶ Was die einzelnen Methoden **ungefähr** tun sollen, sollte eigentlich selbsterklärend sein.

⟶ Feinheiten auf der nächsten Folie.

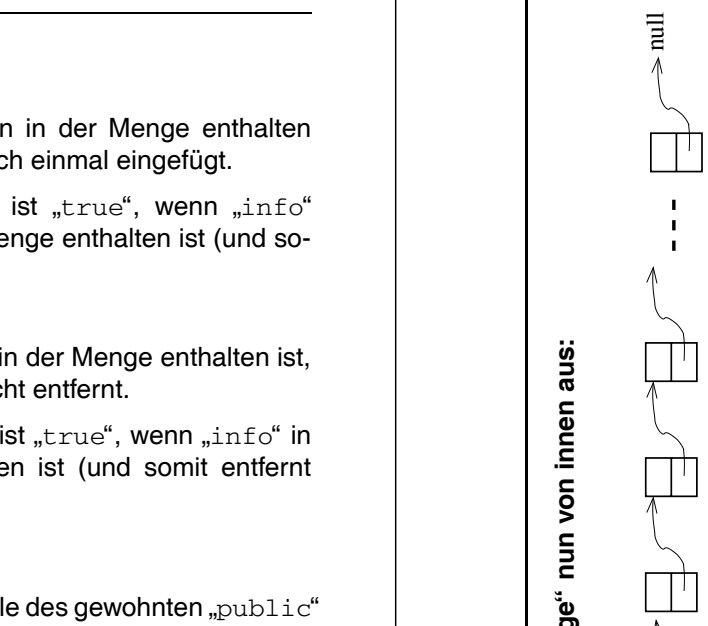
Feinheiten:

- Bei „`fuegeEin`“:
  - ◇ Wenn „`info`“ schon in der Menge enthalten ist, wird es nicht noch einmal eingefügt.
  - ◇ Der Rückgabewert ist „`true`“, wenn „`info`“ noch nicht in der Menge enthalten ist (und somit eingefügt wird).
- Bei „`entferne`“:
  - ◇ Wenn „`info`“ nicht in der Menge enthalten ist, wird es natürlich nicht entfernt.
  - ◇ Der Rückgabewert ist „`true`“, wenn „`info`“ in der Menge enthalten ist (und somit entfernt wird).

Vorgriff:

- Das „`private`“ anstelle des gewohnten „`public`“ sorgt dafür, dass „`erstesElement`“ nur von den Methoden von „`Menge`“ angesprochen werden darf.
- Mehr dazu auf Folie 467 ff.

Wie sieht „Menge“ nun von innen aus:

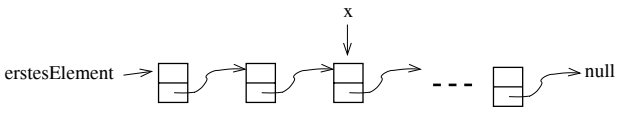


Erläuterung:

- Die einzelnen Elemente der Menge bilden also eine Art Kette, die durch Verweis „`naechster`“ zusammengehalten wird.
- Das Ende dieser Kette wird sinnvollerweise durch „`naechster==null`“ angezeigt.
  - ⟶ Siehe Folie 314 ff.
- Eine Kette dieser Art heißt in der Informatik *Liste*.

Methode „groesse“:

```
public int groesse ()
 {
   Element x         = erstesElement;
   int     rueckgabe = 0;

   while ( x != null )
    {
      x = x.naechster;
      rueckgabe++;
    }
   return rueckgabe;
 }
```

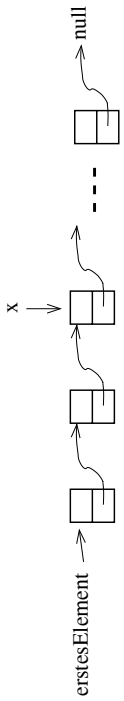


Erläuterung:

- Durch „`x = x.naechster`“ springt „`x`“ von einem Element zum nächsten in der Liste.
- Bei jedem Sprung wird „`rueckgabe`“ um eins erhöht.

⟶ Die Elemente der Liste werden gezählt.

Situation nach zwei Durchläufen durch die Schleife:

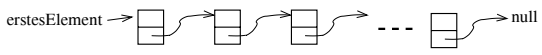


Methoden „istEnthalten“ und „fuegeEin“:

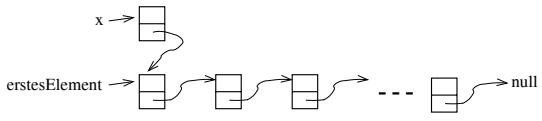
```
public boolean istEnthalten ( int info )
 {
   Element x = erstesElement;
   while ( x != null )
    {
      if ( x.info == info )
         return true;
      x = x.naechster;
    }
   return false;
 }

public boolean fuegeEin ( int info )
 {
   if ( istEnthalten(info) )
      return false;
   Element x = new Element();
   x.info = info;
   x.naechster = erstesElement;
   erstesElement = x;
   return true;
 }
```


Illustration zur Methode „fuegeEin“:

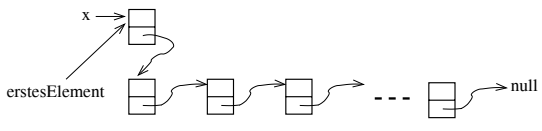
- Ausgangsliste:



- Nach „`x.naechster = erstesElement`“:



- Nach „`erstesElement = x`“:



Methode „entferne“:

- Die Idee ist, das Element mit der gesuchten „`info`“ einfach aus der Liste auszukoppeln.
- Hinterher gibt es dann keinen Verweis mehr auf dieses Element.
- *Erinnerung* an Folie 348 ff.: Ein solches Element wird früher oder später vom Garbage Collector weggeräumt.
- *Methodisches Problem*:
  - ◇ Um ein Element aus der Liste zu entkoppeln, muss man Komponente „`naechster` seines Vorgängers ändern.
  - ◇ Beim Durchlauf durch die Liste, um das Element zu finden, muss man also immer um ein Element zurückbleiben.
- Falls das zu löschende Element das allererste ist, geht die Entkopplung ganz einfach:

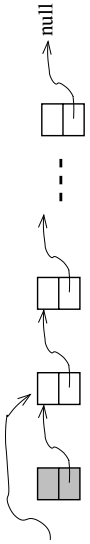
```
erstesElement = erstesElement.naechster;
```


Methode „entferne“ in Java:

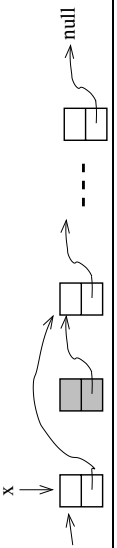
```
public boolean entferne ( int info )
 {
   if ( erstesElement == null )
      return false;

   if ( erstesElement.info == info )
    {
      erstesElement = erstesElement.naechster;
      return true;
    }

   Element x = erstesElement;
   while ( x.naechster != null )
    {
      if ( x.naechster.info == info )
       {
         x.naechster = x.naechster.naechster;
         return true;
       }
      x = x.naechster;
    }
   return false;
 }
```


Veranschaulichung von „entferne“:

- Nach `erstesElement = erstesElement.naechster` im Fall `erstesElement.info == info`:



- Nach `x.naechster = x.naechster.naechster` im Fall, dass `x.naechster.info == info` beispielsweise schon im ersten Durchlauf der `while`–Schleife erreicht wird.



Weitere denkbare Methoden

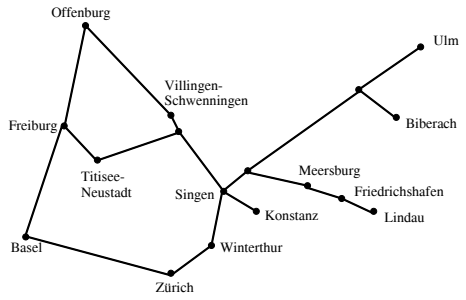
Es sind noch weitere Methoden denkbar, die auf Listen operieren:

- Vermischen zweier Listen: `merge`
- Eine Liste an eine andere hängen: `append`
- Eine Liste sortieren: `sort`
- Eine Liste ausgeben: `print`

Vgl. Übungen zur Allgemeinen Informatik II

Strukturen bauen

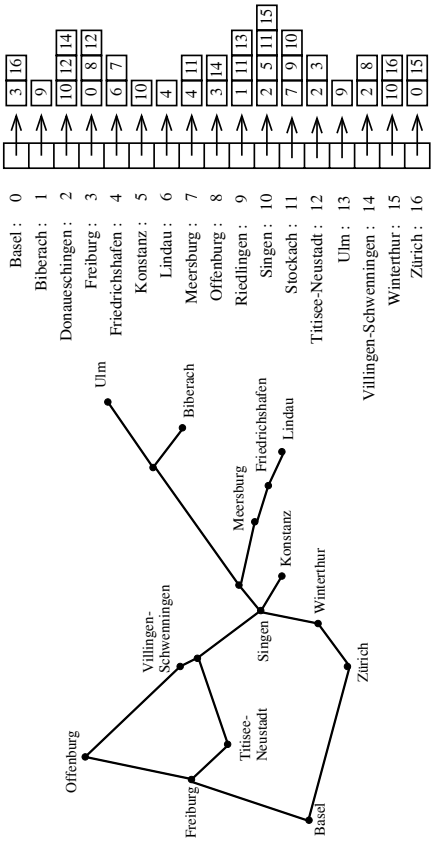
- Auf Folie 357 haben wir gesagt, dass wir mit rekursiven Datenstrukturen auch Strukturen bauen können.
- Wir haben auch schon ein Beispiel angedeutet: Straßennetze.
- *Frage*: Wie geht das?
- *Idee*:
  - ◇ Ein Array von Knotenpunkten.
  - ◇ Für jeden Knotenpunkt eine Liste von Nachbarknotenpunkten.



Realisierung in Java:

```
public class KnotenInfo
 {
   public String name;
   public Menge  nachbarn;
 }

...

KnotenInfo[] autobahnnetz
  = new KnotenInfo[17];
autobahnnetz[0] = new KnotenInfo();
autobahnnetz[0].name = "Basel";

autobahnnetz[1] = new KnotenInfo();
autobahnnetz[1].name = "Biberach";
...

autobahnnetz[0].nachbarn.fuegeEin(3);
autobahnnetz[0].nachbarn.fuegeEin(16);
...
```


Ergebnis im Bild:



Garbage Collector in die Knie zwingen:

- Auf Folie 354 haben wir gesagt, dass man beliebig viel Speicherplatz erzeugen kann, ohne dass auch nur ein Stück davon unerreichbar wird.
- Bisher nicht klar, wie das möglich sein soll.
- Das geht mit Listen jetzt ganz einfach:

```
public class Element
{
   Element naechstes;
 }

...

Element x;        // == null

while ( true )
 {
   // Neues Element vorne in die
   // Liste einfuegen
   Element y = new Element();
   y.naechstes = x;
   x = y;
 }
```

Abschnitt 4: Objektorientierte Programmierung

- In Abschnitt 3 ging es um fundamentale Basiskonzepte von Java, wie es sie in jeder anderen gängigen Programmiersprachen so oder so ähnlich auch gibt.
- In Abschnitt 4 nun geht es um „fortgeschrittene“ Konzepte, die es nur in sogenannten *objektorientierten* Programmiersprachen gibt.

  ⟶ Die „wahre“ Welt der Java–Programmierung.

Abschnitt 4.1: Mehr zu Methoden

- Aus anderen Programmiersprachen sind *Unterprogramme* (engl. *subroutines*), *Funktionen* und *Prozeduren* bekannt.
- Sie bezeichnen alle im Grunde dasselbe abstrakte Konzept:
  - ◇ Einzelne Teile des gesamten Quelltextes werden zu einer Einheit zusammengefasst.
  - ◇ Eine solche Einheit kann von anderen Stellen des Quelltextes aus *aufgerufen* werden.
  - ◇ Die ganze Kommunikation zwischen aufrufender Stelle und dieser Einheit läuft über eine kleine, fest umrissene Schnittstelle: Parameter und (ggf.) Rückgabewert.
- Das Äquivalent in Java sind die *Methoden*.

Methoden und Klassen:

- Im Gegensatz zu anderen Programmiersprachen gibt es in Java keine isolierten Unterprogramme.
- Statt dessen gehört jede Methode zu einer festen Klasse.
- Der vollständige Name einer Methode ergibt sich durch
  - ◇ Anhängen des Pfades der zugehörigen Klasse (Folie 376),
  - ◇ separiert wie üblich durch einen Punkt.
- *Beispiele*:
  - ◇ „`java.applet.Applet.paint`“,
  - ◇ „`java.lang.StringBuffer.append`“,
  - ◇ „`java.lang.Thread.sleep`“,

Exkurs: Klassenpfade

Die Menge aller Klassen ist grundsätzlich hierarchisch organisiert.

Beispiel:

- Es gibt eigentlich überhaupt keine Klasse „`Window`“,
- sondern eine Klasse „`java.awt.Window`“.
- Namensbestandteil
  - ◇ „`java`“: Die Klasse gehört zum Standardumfang von Java dazu.
  - ◇ „`awt`“: „Abstract Windowing Toolkit“, d.h. die Klasse gehört (keine Überraschung) zum Werkzeugkasten für Fensterbasteleien.

Weiteres Beispiel: „`java.lang.StringBuffer`“
Namensbestandteil „`lang`“ („language“): Die Klasse gehört zu den Kernbausteinen der Programmiersprache Java.

Frage:

- In Java Source Files sieht man häufig Klassennamen ohne Klassenpfad.
- Wie lässt sich das mit der Aussage der letzten Folie vereinbaren?

Antwort:

- Nach einer `import`–Zeile am Anfang des Java Source Files braucht man den dort angegebenen Klassenpfad nicht mehr mit anzugeben.
- *Beispiel*:

```
import java.awt.*
Window win;   ⟵ gleich „java.awt.Window win;“
```

- Speziell für „`java.*`“ und „`java.lang.*`“ betrachtet der Java–Compiler die `import`–Zeilen automatisch als gegeben.

Beachte:

- Zum Beispiel „`import java.*`“ importiert
  - ◇ nur die Klassen mit Klassenpfad „`java`“,
  - ◇ nicht beispielsweise die mit Klassenpfad „`java.awt`“.
- Um zusätzlich die Klassen mit Klassenpfad „`java.awt`“ zu importieren, muss man eben eine zusätzliche Zeile „`import java.awt.*`“ schreiben.

Aspekt 1: Signatur und Überladung

Die *Signatur* einer Methode setzt sich zusammen aus

- dem vollständigen Namen mit Klassenpfad gemäß vorheriger Folie,
- der Anzahl der Parameter (potentiell auch gar keine Parameter),
- den Typen der Parameter in ihrer Reihenfolge in der Parameterliste,
- dem Rückgabetyp (bzw. „`void`“),
- der (potentiell leeren) Liste der in der „`throws`“–Klausel angegebenen Exception–Typen
- sowie den *Modifiern* der Methode wie „`public`“ und „`static`“.

Achtung: „`throws`“–Klauseln sind bis jetzt noch gar nicht eingeführt und Modifier bisher noch nicht ernsthaft betrachtet worden.

⟶ Wird später nachgeholt.

Überladung:

Zwei (oder mehr) Methoden dürfen in Java denselben vollständigen Namen haben, das heißt, sie dürfen

- zur selben Klasse gehören und zugleich
- mit demselben Identifier als Namen der Methode bezeichnet sein,

wenn

- sie sich entweder in der Anzahl der Parameter unterscheiden
- oder (falls die Anzahl gleich ist) wenigstens die Liste der Typen der Parameter sich unterscheidet.

⟶ Eine derart duplizierte Methode heißt *überladen*.

Beispiel für korrekte Überladung:

```
public class MeineKlasse
 {
   public void f ()                    { ... }
   public void f ( int x )             { ... }
   public void f ( double x )          { ... }
   public void f ( int x, double y )   { ... }
   public void f ( double x, int y )   { ... }
 }
```

⟶ Alle diese Methoden dürfen in derselben Klasse „MeineKlasse“ mit demselben Identifier „f“ bezeichnet werden.

Nicht hinzugefügt werden zu obiger Klasse „MeineKlasse“ dürfen zum Beispiel folgende Methoden:

```
public        int  f ( double x )   { ... }
private       void f ( double x )   { ... }
public static void f ( double x )   { ... }
public        void f ( double x )   { ... }
```


Extrembeispiel für überladene Methoden:

- Es gibt insgesamt zehn Methoden mit dem vollständigen Namen „java.lang.StringBuffer.append“.
- *Konkrete Beispiele*:
  - ◇ Mit einem einzelnen „String“–Parameter: Fügt die Zeichenkette im Parameter an die momentan in „StringBuffer“ gehaltene Zeichenkette an.
    ⟶ Die auf Folie 318 und anderswo auf den bisherigen Folien verwendete Variante von „java.lang.StringBuffer.append“.
  - ◇ Mit einem einzelnen „char“–Parameter: Fügt dieses Zeichen hinten an.
  - ◇ Mit einem einzelnen „double“–Parameter: Fügt eine Zeichenkette hinten an, die den numerischen Wert dieses „double“–Parameters als Zeichenkette darstellt.
- Alle diese Beispiele haben die gleiche Anzahl Parameter: 1.
- Aber die Typen unterscheiden sich.

Frage:

- Warum müssen sich zwei Methoden mit dem gleichen vollständigen Namen unbedingt in der Parameterliste unterscheiden?
- Warum reicht es nicht, wenn sie sich im Rückgabetyp, der „throws“–Liste oder den Modifiern unterscheiden?
  ⟶ Erst in Abschnitt 4.5 (Folie 552 ff.) betrachten wir „throws“–Listen.

Antwort:

- Dann kann der Compiler nicht mehr für jeden Aufruf zweifelsfrei entscheiden, welche Methode nun eigentlich gemeint ist.
- Einen Unterschied in der „throws“–Klausel oder der Modifier–Liste allein könnte man einem Aufruf einer Methode überhaupt nicht ansehen.
- Speziell den Rückgabetyp kann der Compiler nicht erkennen, wenn der Rückgabewert einer Methode beim Aufruf wie auf Folie 261 unter den Tisch fällt.

Konkretes Beispiel zum Problem Rückgabetyp:

```
public class MeineKlasse
 {
   public int  f ( int n )  { ... }
   public char f ( int n )  { ... }
      // Verboten!
 }
...

MeineKlasse meinObjekt = new MeineKlasse();
meinObjekt.f(1);
```

Erläuterung:

- *Problem*: Welche der beiden Varianten der Methode „MeineKlasse.f“ ist denn nun mit „meinObjekt.f(1)“ oben gemeint?
- *Lösung in Java*: Durch das Verbot, Methoden allein durch Variation des Rückgabetyps zu überladen, ergibt die Deklaration der zweiten Methode „f“ eine Fehlermeldung vom Compiler.

Anmerkung zur Klarstellung:

- Selbstverständlich
  - ◇ dürfen sich überladene Methoden in Rückgabetyp, Modifiern und „`throws`“–Liste beliebig unterscheiden,
  - ◇ nur eben nicht darin *allein*,
  - ◇ sondern auf jeden Fall müssen sich auch die Parameterlisten voneinander unterscheiden.
- Selbstverständlich
  - ◇ dürfen Methoden aus *verschiedenen* Klassen identischen Namen und zugleich identische Parameterliste haben,
  - ◇ und in diesem Fall dürfen sie sich dann (müssen aber nicht) auch in Rückgabetyp, Modifiern und „`throws`“–Liste beliebig unterscheiden,

Signatur und Interpreter:

- Ein Java–Interpreter wie „`java`“ oder „`appletviewer`“
  - ◇ bekommt den Namen einer Java–Klasse als Argument beim Aufruf mit und
  - ◇ erwartet als Einstiegspunkt immer eine Methode dieser Klasse mit ganz bestimmter Signatur.
- *Konkret*:
  - ◇ „`java`“ erwartet eine Methode namens „`main`“ mit Rückgabetyp „`void`“, mit einem (einzigen) Parameter vom Typ „`String`“–Array und mit den Modifiern „`public`“ und „`static`“.
  - ◇ „`appletviewer`“ erwartet eine Methode namens „`paint`“ mit Rückgabetyp „`void`“, einem Parameter vom Typ „`Graphics`“ und dem Modifier „`public`“.
  - ◇ Beide erwarten dabei eine leere „`throws`“–Liste.

Fortsetzung Signatur und Interpreter:

- Diese erwartete Methode wird vom jeweiligen Interpreter als Programmstart aufgerufen.
- Wenn ihre Abarbeitung beendet ist, ist das Java–Programm zu Ende.
- Wenn die jeweils erwartete Methode nicht *mit genau der erwarteten Signatur* in der Klasse vorhanden ist,
  - ◇ bricht der Interpreter sofort ab und
  - ◇ gibt eine Fehlermeldung aus, dass „die–und–die Methode“ nicht gefunden wurde.
- Da man sich erfahrungsgemäß oft bei den Details der Signatur vertut, schauen viele Interpreter genauer hin und geben ggf. eine Fehlermeldung, die
  - ◇ nicht einfach besagt, dass die „die–und–die Methode“ nicht gefunden wurde,
  - ◇ sondern die besagt, dass die fragliche Methode „die–und–die Signatur“ hat.

Applets mit „appletviewer“ starten:

- *Erinnerung* an Folie 25: HTML ist die Sprache, in der WWW–Seiten (hauptsächlich) geschrieben sind.
- Im Gegensatz zur Behauptung auf Folie 386 ist das Argument von „`appletviewer`“’
  - ◇ eigentlich nicht der Name einer Java–Klasse,
  - ◇ sondern der Name eines HTML–Files.

Auflösung des Widerspruchs:

- Das Programm „`appletviewer`“
  - ◇ ist eigentlich gar kein Java–Interpreter,
  - ◇ sondern ruft einen Java–Interpreter auf,
  - ◇ der seinerseits den Namen einer Java–Klasse als Argument erhält
  - ◇ und in dieser Klasse eine Methode „`paint`“ mit der richtigen Signatur erwartet.

⟶ Weiter auf der nächsten Folie...

Fortsetzung Auflösung des Widerspruchs:

- In HTML gibt es sogenannte *Applet–Tags*.
- Einfaches Beispiel:

```
<APPLET CODE=XY.class>Hallo</APPLET>
```

- Wenn „`appletviewer`“ beim Lesen seines Arguments auf ein solches Applet–Tag stößt,
  - ruft er den Interpreter auf,
  - übergibt dem Interpreter den Namen der Java-Klasse hinter „`CODE=`“ als Argument und
  - richtet eine Zeichenfläche für das Applet ein.
- Der Java–Interpreter
  - richtet ein Objekt vom Typ „`Graphics`“ ein,
  - verknüpft es mit dieser Zeichenfläche und
  - ruft dann „`paint`“ mit diesem „`Graphics`“-Objekt als Argument auf.

Applets mit WWW–Browsern starten:

- Jeder WWW–Browser tut im Prinzip dasselbe wie „`appletviewer`“:
  - Er schaut jedes eingeladene HTML–File durch.
  - Im Gegensatz zu „`appletviewer`“ bearbeitet ein WWW–Browser alle HTML–Tags (zumindest alle gängigen).
- Speziell bei einem Applet–Tag macht er mehr oder weniger dasselbe wie „`appletviewer`“:
  - eigentlichen Java–Interpreter starten,
  - Java–Klasse hinter „`CODE=`“ dem Java–Interpreter als Argument übergeben,
  - Zeichenfläche einrichten und
  - „`Graphics`“–Objekt einrichten und „`paint`“ damit aufrufen.
- Das ist der Grund,
  - warum man Java–Applets so einfach ins WWW hängen kann und
  - warum es ein Applet mit „`paint`“ sein muss und nicht irgendeine andere Art von Java-Programm.

Kleiner Ausblick:

- Wir haben jetzt gesagt, dass ein Interpreter immer eine Einstiegsmethode hat.
- Genauer muss es heißen: *mindestens* eine.
- *Konkretes Beispiel*: WWW–Browser und das Programm „`appletviewer`“ erwarten noch weitere Methoden neben „`paint`“ (ebenfalls mit exakt vorgegebener Signatur) und rufen diese Methoden in bestimmten Situationen auf.
- Zum Beispiel:
  - „`void`“–Methode „`init`“ mit leerer Parameterliste.

    $\longrightarrow$ Wird beim Start des Applets aufgerufen (noch vor „`paint`“).
  - „`void`“–Methode „`repaint`“: Wann immer der Inhalt des Fensters neu zu zeichnen ist.
- *Frage*: Wieso mussten wir diese Methoden im Gegensatz zu „`paint`“ nicht implementieren?

  $\longrightarrow$ Antwort auf den nächsten zwei Folien.

Antwort:

- Diese Methoden sind in der Klasse „`Applet`“ implementiert:
  - „`Applet.init`“: Macht schlichtweg gar nichts.
  - „`Applet.repaint`“: Macht im Prinzip dasselbe wie „`Applet.paint`“.
- *Erinnerung* an Folie 303 ff.: Selbstgebastelte Applet–Klassen müssen durch „`extends Applets`“ die Klasse „`Applet`“ als Urstamm bekommen.
- *Vorgriff* auf Abschnitt 4.4 (Folie 489 ff.): Wenn
  - eine Methode nicht in einer Klasse A implementiert ist,
  - aber in ihrer Urstammklasse B,

  dann „erbt“ A diese Methode von B.

Fortsetzung Antwort:

- „`Applet.init`“ und „`Applet.repaint`“ (und ein paar weitere Methoden) sind also ohne unser Zutun schon in jeder unserer Applet–Klassen vorhanden.
- Nach der kurzen Erläuterung oben auf dieser Folie, was die beiden Methoden tun (bzw. nicht tun), ist auch klar, warum wir bei unserer Arbeit mit Applets bisher nichts von Ihnen bemerkt haben.

Erinnerung: Die Frage lautete exakt:

> Wieso mussten wir diese Methoden im Gegensatz zu „`paint`“ nicht implementieren?

Verfeinerte Antwort:

- Auch „`paint`“ ist in Klasse „`Applet`“ schon implementiert.
- *Inhalt*: „`Applet.paint`“ macht überhaupt nichts.
- Das in der obigen Frage eingeschobene „im Gegensatz zu ‚`paint`‘“ ist also eigentlich unbegründet:
  - ◇ Auch „`paint`“ hätte in keiner unserer Applet-Klassen implementiert werden müssen.
  - ◇ Statt dessen hätte auch „`paint`“ von „`Applet`“ geerbt werden können.
- *Frage*: Warum haben wir nicht auch „`paint`“ einfach von „`Applet`“ geerbt?
- *Klare Antwort*: Weil eine „`paint`“–Methode, die gar nichts macht, nicht besonders prickelnd ist.

Aspekt 2: Klassen- vs. Objektmethoden

Zunächst zur Syntax:

- Eine *Klassenmethode* erkennt man daran, dass der Modifier „`static`“ vor dem Rückgabetyp steht.
- Die Methoden, die wir bisher in den Übungen selbst gebastelt haben, hatten alle noch kein „`static`“.

  ⟶ Waren also alles Objektmethoden, keine Klassenmethoden.
- „`static`“ ist nicht gerade ein sehr intuitives Schlüsselwort für Klassenmethoden.

  ⟶ Tatsächlich wieder eine Altlast aus C/C++.

Was für eine Altlast:

- „`static`“ wird in C und C++ eigentlich für ganz andere Sachen verwendet.

  ⟶ Sachen, bei denen das Wort „`static`“ auch wirklich Sinn macht.
- C++ sollte von Anfang an möglichst kompatibel mit C sein.

  ⟶ Jedes korrekte C–Programm sollte nach Möglichkeit auch ein korrektes C++–Programm sein.
- Ein wichtiger Punkt dabei war, dass man in C++ möglichst nicht weitere Schlüsselwörter einführt, die es in C nicht gibt.

  ⟶ Könnten ja in vielen C–Programmen als Identifier verwendet worden sein.
- Als für Klassenmethoden u.ä. ein Schlüsselwort gesucht wurde, hat man also ein schon vorhandenes zweckentfremdet: „`static`“.

Beispiele für Klassenmethoden:

- „`java.lang.Character.isLowerCase`“ und „`java.lang.Character.toUpperCase`“ (Folie 292),
- „`java.lang.Thread.sleep`“,
- „`java.awt.Color.getHSBColor`“: Bekommt Werte für Farbton, Sättigung und Helligkeit als drei Parameter (Folie 47 ff.) und liefert ein Objekt vom Typ „`Color`“, also in RGB–Kodierung:

```
Color c = Color.getHSBColor (1,1,1);
```

⟶ Violett mit voller Helligkeit und Sättigung.

Beachte zur Sprechweise:

- „Klassenmethode“ und „Methode einer Klasse“ bedeuten nicht genau dasselbe.
- Wie gesagt, gehört ja **jede** Methode zu einer Klasse.
- In diesem Sinne ist also
  - ◇ nicht nur jede Klassenmethode eine „Methode einer Klasse“,
  - ◇ sondern auch jede Objektmethode ist genauso eine „Methode einer Klasse“,
- Wann immer von „Methoden einer Klasse“ die Rede ist, sind daher Objekt- wie Klassenmethoden gleichermaßen gemeint.

Beispiel zur Syntax mit „static“:

```
public class MeineKlasse
 {
   public void objektMethode ()
    {
      System.out.println ( "Hello 1" );
    }

   public static void klassenMethode ()
    {
      System.out.println ( "Hello 2" );
    }
 }
```

Erläuterung:

- „`objektMethode`“ ist eine Methode wie bisher bekannt.
- Wie auf Folie 395 gesagt, ist der syntaktische Unterschied bei Klassenmethoden zunächst einmal nur das „`static`“.

Was sind denn nun Klassenmethoden:

- Klassenmethoden sind im Grunde nichts anderes als Unterprogramme (Funktionen, Prozeduren), wie es sie in anderen Programmiersprachen auch gibt.
- Der wesentliche Unterschied ist, dass in Java eben Unterprogramme nur in Form von Methoden von Klassen möglich sind.

Beispiel: Quadratwurzelberechnung

- „`y := sqrt(x);`“ in Pascal,
- „`y = sqrt(x);`“ in C,
- „`y = java.lang.Math.sqrt(x);`“ in Java.
- sqrt = square root

Exkurs: java.lang.Math:

- Diese Klasse dient im wesentlichen als Zusammenfassung für diverse grundlegende mathematische Funktionen (alle realisiert als Klassenmethoden).
- *Beispiele*:
  - ◇ Sinus:
    ```
    y = java.lang.Math.sin(x);
    ```
  - ◇ Kosinus:
    ```
    y = java.lang.Math.cos(x);
    ```
  - ◇ Potenzbildung x^y:
    ```
    z = java.lang.Math.pow(x,y);
    ```
  - ◇ Absolutbetrag $|x|$:
    ```
    y = java.lang.Math.abs(x);
    ```
  - ◇ Maximum aus zwei Zahlenwerten:
    ```
    z = java.lang.Math.max(x,y);
    ```


Aufruf von Klassenmethode mit Variable:

```
char   c1 = 'a';
String str = new String ( "Hello" );

char   c2 = java.lang.String.toUpperCase (c1); // (1)
char   c3 = String.toUpperCase (c1);           // (2)
char   c4 = str.toUpperCase (c1);              // (3)
```

⟶ In den Zeilen (1)–(3) passiert immer das selbe

⟶ „`java.lang.String.toUpperCase`“: vgl: Folie 28

⟶ Erläuterungen auf der nächsten Folie.

Erläuterung der letzten Folie:

- Der Aufruf von „`toUpperCase`“ in der Initialisierung von „`c2`“ ist wie gehabt.
- *Erinnerung*: Wie in der Initialisierung von „`c3`“ noch einmal demonstriert, geht es gemäß Folie 377 auch ohne Import von „`java.lang.*`“.
- Die Variante in der Initialisierung von „`c4`“ ist neu: Man kann eine Klassenmethode auch mit dem Namen einer Variablen anstelle des Klassennamens aufrufen.
- Aber egal ob Klassenname oder Variablenname: Es macht in der Auswirkung absolut keinen wie auch immer gearteten Unterschied. Die Semantik ist in beiden Fällen dieselbe.

Fortsetzung: Erläuterung der Folie 402:

- Wozu ist dann Variablenname möglich:
  - ◇ Wenn man eine Methode mit einem Variablennamen aufruft, braucht man sich keine Gedanken darum zu machen, ob dies nun eine Klassen- oder Objektmethode ist.
  - ◇ Wenn eine Objektmethode (einfach durch Einfügen von „`static`“) nachträglich zu einer Klassenmethode gemacht wird, braucht kein Stück Java–Quelltext bei der Verwendung der Methode deswegen geändert zu werden.
- Beachte jedoch:
  Klassenmethoden dürfen nicht auf Datenkomponenten des Objekts hinter der Variablen, mit der sie aufgerufen wurden, zugreifen. Sie müssen daher auch nicht mit einem Objekt aufgerufen werden.

Nun ein selbstgebasteltes Beispiel dazu:

```
public class MeineKlasse
   {
     public static void f ()
        {
          System.out.println ( "Hello" );
        }
   }
...
MeineKlasse  meinObjekt = new MeineKlasse ();

MeineKlasse.f ();   // Wie bisher
meinObjekt.f ();    // Auch ok!
```


Objektmethoden:

- Können im Gegensatz zu Klassenmethoden nur mit dem Namen einer Variablen der Klasse, nicht mit dem Namen der Klasse selbst aufgerufen werden:

```
StringBuffer str
   = new StringBuffer ( "Hello" );

// Ok:
str.append ( ", World" );

// Verboten:
StringBuffer.append ( ", World" );

// Auch verboten:
java.lang.StringBuffer.append ( ", World" );
```

- *Terminologie*: Wir sagen, die Methode „`append`“ ist auf „`str`“ *angewandt* worden.

  ⟶ Genauer gesagt, auf das Objekt, auf das „`str`“ verweist (vgl. Folie 305 ff.).

- Oft wird in solchen Fällen schlampig verkürzt von dem „Objekt `str`“ gesprochen.

Wieso nicht mit Klassennamen:

```
StringBuffer str
   = new StringBuffer ( "Hello" );

str.append ( ", World" );
      //  Erlaubt
StringBuffer.append ( ", World" );
      //  Verboten!
```

- to append = hinten anhängen

Erläuterung:

- In der Zeile mit Kommentar „Erlaubt“ ist eine der „`append`“–Methoden von „`StringBuffer`“ auf das Objekt „`str`“ angewandt worden.
- Die Semantik der (insgesamt zehn) Methoden mit dem vollständigen Namen „`java.lang.StringBuffer.append`“ besagt ja gerade, dass an ein konkretes Objekt etwas angehängt werden soll.

- Es macht daher überhaupt keinen Sinn, „`append`“ wie in der Zeile mit Kommentar „Verboten!“ ohne ein Objekt aufzurufen, an das der Parameter angehängt werden soll.

Noch einmal allgemeiner formuliert:

- Eine Objektmethode darf auf das Objekt, auf das es angewandt wurde, sowie auf dessen Komponenten (vgl. Folie 332) lesend und verändernd zugreifen.
- Natürlich muss eine Objektmethode nicht auf das Objekt (und dessen Komponenten) zugreifen, mit dem es aufgerufen wurde.

  $\longrightarrow$ In diesem Fall gäbe es kein Problem mit einem Aufruf einer Objektmethode ohne Objekt.
- *Entscheidung* beim Design von Java:

  Eine Objektmethode darf dennoch generell nicht ohne Objekt aufgerufen werden.
- Warum auch nicht:

  Eine Methode, die nicht auf das Objekt und seine Komponenten zugreift, kann man ja einfach zu einer Klassenmethode machen.

  $\longrightarrow$ Dazu reicht ja aus, ein „`static`“ einzufügen.

Erläuterungen zur Syntax:

- Bei Licht besehen ist „`str`“ im Grunde nichts anderes als ein zweiter Parameter der Methode „`append`“, der aus unerfindlichen(?) Gründen
  - ◇ nicht in der Parameterliste auftaucht,
  - ◇ sondern vor den Methodennamen geschrieben wird,
  - ◇ mit einem Punkt davon getrennt.
- Die syntaktische Konvention

  ```
  str.append(", World");
  ```

  ist reiner „syntaktischer Zucker“ zur Unterstreichung, dass
  - ◇ „`append`“ eine Methode der Klasse von „`str`“ ist und
  - ◇ der Parameter „`str`“ herausragende Bedeutung für die Logik des Aufrufs von „`append`“ hat.

Fortsetzung Erläuterungen zur Syntax:

- Man hätte es zum Beispiel durchaus statt dessen so festlegen können, dass
  - ◇ der Aufruf

    ```
    append ( str, ", World" );
    ```

    lautet, wie man es aus anderen Programmiersprachen gewohnt ist,
  - ◇ mit der Regel, dass der erste Parameter derjenige ist, auf den die Methode angewandt wird.
- Man hat sich aber in Java (und in den anderen gängigen objektorientierten Programmiersprachen) für diese eher „krasse“ syntaktische Hervorhebung von „`str`“ als wichtigstem Parameter von „`append`“ entschieden.

Gegenseitiger Aufruf von Methoden:

```
public class MeineKlasse
 {
   int n;

   public static void meineKlassenMethode1 ()
    {
      System.out.println ( "Hallo" );
    }
   public static void meineKlassenMethode2 ()
    {
      meineKlassenMethode1();
    }
   public void meineObjektMethode1 ()
    {
      n = 1;   // Darf nur Objektmethode!
      meineKlassenMethode2();
    }
   public void meineObjektMethode2 ()
    {
      meineObjektMethode1();
    }
 }

...

MeineKlasse meinObjekt = new MeineKlasse();
meinObjekt.meineObjektMethode2();
```

Erläuterungen:

- Durch den Aufruf „`meinObjekt.meineObjekt-Methode2()`“ auf der vorherigen Folie werden implizit auch die anderen Methoden von „`MeineKlasse`“ auf „`meinObjekt`“ angewandt.
- Insbesondere wird „`meinObjekt.n`“ durch den Aufruf von

  „`MeineKlasse.meineObjektMethode1`“

  innerhalb von

  „`MeineKlasse.meineObjektMethode2`“

  auf 1 gesetzt.

Allgemeine Regel dahinter:

- Wenn eine Objektmethode in einer anderen Objektmethode ohne den vorangestellten (mit „.“ abgetrennten) Namen einer Variablen wie „`meinObjekt` aufgerufen wird, dann wird der erstere Aufruf auf dasselbe Objekt wie der letztere Aufruf angewandt.
- Das ginge auch gar nicht anders:

```
MeineKlasse meinObjekt1 = new MeineKlasse();
MeineKlasse meinObjekt2 = meinObjekt1;
MeineKlasse meinObjekt3 = new MeineKlasse();
```

  ⟶ Zum Beispiel die Situation in „`MeineKlasse.meineObjektMethode2`“:

  Soll es darin nun „`meinObjekt1.meine-ObjektMethode1`“ heißen oder was sonst?

Frage:

- Eine Objektmethode kann jederzeit eine Klassenmethode auf diese Art und Weise aufrufen.
- Auch umgekehrt?

Umgekehrt:

- Der Aufruf einer Objektmethode durch eine Klassenmethode ist strikt verboten.
- Das wäre auch semantischer Unsinn:
  - ◇ Eine Klassenmethode kann ja auch ohne Anwendung auf ein Objekt aufgerufen werden.
  - ◇ Eine Objektmethode hingegen darf grundsätzlich nur durch Anwendung auf ein Objekt aufgerufen werden.
- *Mit anderen Worten*:
  - ◇ Wenn eine Klassenmethode ohne Objekt aufgerufen wird
  - ◇ und in dieser Klassenmethode eine Objektmethode aufgerufen werden dürfte,
  - ◇ dann würde dieser Objektmethode das notwendige Objekt fehlen.

Abschnitt 4.2: Mehr zu Klassen

Aspekt 1: Was sind Klassen

- Eine Klasse ist ein Datentyp (vgl. Folie 287 sowie Folie 300 ff.).
- Ein Datentyp ist generell charakterisiert durch
  - ◇ die abstrakten Zustände, die ein Objekt dieses Typs annehmen kann, und
  - ◇ die Zugriffsmöglichkeiten, mit denen der Zustand eines Objekts gelesen und/oder verändert werden kann.

Beispiel I: Datentyp „int“

- Die abstrakten Zustände, die ein Objekt vom Datentyp „`int`“ annehmen kann, sind alle ganzen Zahlen zwischen der kleinsten und der größten mit „`int`“ (32 Bit) darstellbaren Zahl.
- Abstrakte vs. konkrete Zustände:

  Die *konkreten* Zustände sind die verschiedenen Bitmuster aus 32 Bit (vgl. Folie 14 ff.).
- Die Zugriffsmöglichkeiten zum Verändern des Zustands einer „`int`“–Variable sind Zuweisungen mit „`=`“, „`+=`“, „`*=`“ usw. sowie „`++`“ und „`--`“:

```
int i = 1;
```

- Zum Lesen des Zustands eines „`int`“–Objekts „`i`“ schreibt man den Namen des Objekts einfach hin:

```
int j = i + 1;  // Zustand von 'i' gelesen
```


Beispiel II: Klasse „StringBuffer“

- Die abstrakten Zustände sind alle möglichen Zeichenketten aus Unicode–Zeichen (Folie 26) einschließlich der „leeren“ Zeichenkette ohne Zeichen.
- Die konkreten Zustände im Innern eines „`StringBuffer`“–Objekts konstituieren sich im wesentlichen in
  - ◇ den konkreten Bitmustern der einzelnen Unicode–Zeichen und
  - ◇ der Art und Weise, wie Zeichenketten intern organisiert sind (vor allem die Fragestellung von Folie 310 ff.).
- Verändernde Zugriffsmöglichkeiten:
  - ◇ Zum Beispiel die zehn Methoden mit Namen „`append`“ (Folie 382).
  - ◇ Analog dazu gibt es z.B. auch neun verschiedene Methoden „`StringBuffer.insert`“, mit denen neuer Text irgendwo mitten in der Zeichenkette eingefügt werden kann.

Lesende Zugriffsmöglichkeiten bei „StringBuffer“:

- Insbesondere (aber nicht nur) mit Methode

  `StringBuffer.charAt`

  kann man lesend auf ein „`StringBuffer`“–Objekt zugreifen.
- *Verwendung*: Der Ausdruck „`str.charAt(i)`“
  - ◇ hat „`char`“ als Rückgabetyp,
  - ◇ und der Rückgabewert ist das Zeichen mit Index „`i`“ in der momentan im „`StringBuffer`“–Objekt „`str`“ gehaltenen Zeichenkette.
- *Index*: Wie bei Arrays, d.h. das erste Zeichen hat Index 0 usw.
- Eine völlig identische Methode „`charAt`“ ist übrigens auch für Klasse „`String`“ definiert.

```
String str = new String("Hello");
System.out.print(str.charAt(4)); // -> "o"
```


Kurzexkurs: Variable und Konstanten in Klassen

```
public class MeineKlasse
  {
    int       n1;
    final int n2 = 1;
  }
```

Erläuterungen:

- Der Unterschied zwischen Variablen und Konstanten (mit Schlüsselwort „`final`“) wurde auf Folie 284 ff. behandelt.
- Bei Komponenten von Klassen (vgl. Folie 332) gibt es exakt dieselbe Unterscheidung mit exakt denselben Konsequenzen:
  - ◇ Eine konstante Komponente muss sofort initialisiert werden.
  - ◇ Der Wert einer konstanten Komponente darf nach der Initialisierung nicht mehr geändert werden.

Aspekt 2: Klassen- vs. Objektvariable, Klassen- vs. Objektkonstanten

- Wie bei Methoden gibt es auch bei den Datenkomponenten einer Klasse (vgl. Folie 332) die Unterscheidung zwischen
  - ◇ Klassen- und Objektvariable bzw.
  - ◇ Klassen- und Objektkonstanten (also mit Schlüsselwort „`final`“).
- Syntaktische Unterscheidung:

  Analog zu Methoden durch Schlüsselwort „`static`“ bei Klassenvariablen bzw. -konstanten vor der Angabe des Datentyps.

Semantischer Unterschied:

- Eine Klassenvariable/-konstante ist ein einzelnes Objekt.
- Eine Objektvariable/-konstante gibt es einmal pro Objekt der Klasse.
- Die Bestandteile von Objekten auf Folie 332 ff. sind also genauer gesagt Objektvariable.
- Analog zu Klassenmethoden kann man auf Klassenvariable auch ohne konkretes Objekt der Klasse zugreifen.

Beispiel:

```
public class MeineKlasse
 {
   public        int n1;   //  Objektvariable
   public static int n2;   // Klassenvariable
 }

...

MeineKlasse  meinObjekt1 = new MeineKlasse();
MeineKlasse  meinObjekt2 = new MeineKlasse();

meinObjekt1.n1 = 1;
meinObjekt1.n2 = 2;
meinObjekt2.n1 = 3;
meinObjekt2.n2 = 4;

System.out.println ( meinObjekt2.n1 );   // -> 3
System.out.println ( meinObjekt2.n2 );   // -> 4
System.out.println ( meinObjekt1.n1 );   // -> 1
System.out.println ( meinObjekt1.n2 );   // -> 4(!)
System.out.println ( MeineKlasse.n2 );   // -> 4(!)
```


Erläuterungen:

- „`meinObjekt1.n1`“ und „`meinObjekt2.n1`“ sind zwei separate „`int`“–Objekte, die wie auf Folie 332 ff. Bestandteile der Objekte „`meinObjekt1`“ bzw. „`meinObjekt2`“ sind.
- „`MeineKlasse.n2`“ ist im Gegensatz dazu nur ein einzelnes, isoliertes „`int`“–Objekt, das ein einziges Mal irgendwo im Speicher völlig unabhängig von „`meinObjekt1`“ bzw. „`meinObjekt2`“ angelegt wird.

  ⟶ „`meinObjekt1.n2`“ und „`meinObjekt2.n2`“ bezeichnen dasselbe „`int`“–Objekt.
- Die letzte Zeile auf der vorherigen Folie zeigt, wie man ohne ein Objekt von „`MeineKlasse`“ auf dieses einzelne Objekt „`n2`“ zugreifen kann.

  ⟶ Vgl. Benutzung von Klassenmethoden ohne Objekt (Folie 395 ff.).

Veranschaulichung:

Weitere Erläuterung:

- Beim Übersetzen interpretiert der Compiler die beiden Ausdrücke

  „`meinObjekt1.n1`“ und „`meinObjekt1.n2`“

  völlig verschieden.

- In beiden Fällen konstruiert der Compiler Java Byte Code, der die Adresse des Objektes

  „`meinObjekt1.n1`“ bzw. „`meinObjekt1.n2`“

  berechnet.

- Bei „`meinObjekt1.n1`“ wird die Adresse berechnet, indem die Position von „`n1`“ in „`MeineKlasse`“ auf den Wert von „`meinObjekt1`“ draufaddiert wird.

- Der Compiler hat sich natürlich irgendwo intern die Adresse von „`MeineKlasse.n2`“ gemerkt.

- Bei „`meinObjekt1.n2`“ wird diese Adresse einfach eingesetzt.

Beispiel mit/ohne „final“:

```
public class Nonsens
  {
    public      int a = 1;        // Ok
    public final int b = 1;       // Ok
    public      int c;            // Ok
    public final int d;           // Fehler!

    public static     int e = 1;  // Ok
    public static final int f = 1;// Ok
    public static     int g;      // Ok
    public static final int h;    // Fehler!
  }
```


Klassenkonstanten in der Standardbibliothek:

- Die Kreiszahl $\pi = 3.14159$ ist selbstverständlich reellwertig und konstant (also „`final double`“):

  „`java.lang.Math.PI`“

- Die wichtigsten Farben sind als Konstanten vom Typ „`java.awt.Color`“ mit den entsprechenden RGB–Werten schon in der Klasse „`java.awt.Color`“ definiert (vgl. diverse Übungsaufgaben):

  ◇ „`java.awt.Color.red`“

  ◇ „`java.awt.Color.yellow`“

  ◇ „`java.awt.Color.green`“

  ◇ usw.

Implementation der Farbobjekte:

```
public class Color
  {
    static final Color red    = new Color ( 1, 0, 0 );
    static final Color yellow = new Color ( 1, 1, 0 );
    static final Color green  = new Color ( 0, 1, 0 );
      ...
  }
```

- „red“ ist logisch gesehen konstant ⟶ „final“.
- „red“ ist immer und überall gleich ⟶ „static“ ⟶ weniger Speicherplatz

Weiteres Beispiel: System.out.print(ln):

- „java.lang.System.out“: Eine Klassenvariable der Klasse „java.lang.System“.

  ⟶ Daher verwendbar in der Form „System.out“.
- Typ von „java.lang.System.out“: „java.io.PrintStream“.
- „java.io“: Sammlung von Standard–Klassen für Eingaben von Tastatur und Files und für Ausgaben auf „xterm“–Fenster und Files.
- What's in a name: I/O = Input/Output = Ein-/Ausgabe.
- „PrintStream“: Speziell zur Datenausgabe.

Fortsetzung: System.out.print(ln):

- Klasse „PrintStream“ bietet unter anderem Methoden namens „print“ und „println“ zur Ausgabe.
- Diese Methoden sind für alle eingebauten Typen und einige gängige Klassentypen wie „String“ als Parametertypen überladen (vgl. Folie 380 ff.).
- Erinnerung (vgl. Folie 377): „java.lang.*“ wird immer automatisch importiert, deswegen kann man für „java.lang.System.out.print()“ auch „System.out.print“ verwenden.

Zugriff in Objektmethoden durch „this“:

```
public class MeineKlasse
 {
   public int n = 2;

   public void meineMethode1 ()
    {

      int n = 7;
      System.out.print   ( n );
      System.out.print   ( " " );
      System.out.println ( this.n );
    }

   public void meineMethode2 ()
    {
      (this.n)++;
      System.out.println ( n );
      System.out.print   ( " " );
      System.out.println ( this.n );
    }
 }

...

MeineKlasse meinObjekt = new MeineKlasse();
meinObjekt.meineMethode1();       // -> "7 2"
meinObjekt.meineMethode2();       // -> "3 3"
meinObjekt.meineMethode2();       // -> "4 4"
meinObjekt.meineMethode1();       // -> "7 4"
```

Zugriff in Objektmethoden durch „this“ II:

```
public class MeineKlasse2
 {

   int n = 1;

   public void nocheineMethode( int n )
    {

        n = 27;
        System.out.print( n );
        System.out.print(" ");
        System.out.print( this.n );

    }

 }


...

MeineKlasse2 meinObjekt2 = new MeineKlasse2();
meinObjekt2.nocheineMethode( 12 );  // -> "27 1"
```


Erläuterungen:

- Das Schlüsselwort „`this`“ ist in einer Objektmethode einer Klasse ein Verweis auf das Objekt, mit dem diese Methode aufgerufen wurde.

  ⟶ Mit „`this`“ kann insbesondere auf eine Klassen- oder Objektvariable/-konstante zugegriffen werden („`this.n`“).

- Da dies ein extrem häufiger Fall ist, darf man „`this`“ dann auch weglassen (d.h. `this.n==n`).
- *Aber*:
  - ◇ Der Name einer Klassen- oder Objektvariable bzw. -konstante kann wie auf der vorherigen Folie (Methode „`meineMethode1`“) innerhalb der Methode für ein gänzlich anderes Objekt noch einmal vergeben werden.
  - ◇ Die Deklaration eines solchen Objekts „überdeckt“ die Klassen- bzw. Objektvariable/-konstante.

    ⟶ Letztere kann von da an *nur* noch mit Hilfe von „`this`“ angesprochen werden.

Wozu braucht man so etwas:

- Man braucht „`this`“ natürlich nicht wirklich.
- Aber manchmal erleichtert es die Programmierarbeit und verbessert die Lesbarkeit des Programms.
- *Konkret*: Manchmal ist es einfach unnatürlich, verschiedenen Variablen, die im Konflikt zueinander stehen, unterschiedliche Namen zu geben.
- *Beispiel*:
  - ◇ Eine Kreisklasse, in der Kreise durch Mittelpunkt und Radius gegeben sind.
  - ◇ Die entsprechenden Datenkomponenten sollten dann sinnvollerweise auch „`x`“, „`y`“ und „`radius`“ heißen.
  - ◇ In einer Methode „`setzeKreis`“ heißen die Parameter aber ebenfalls natürlicherweise „`x`“, „`y`“ und „`radius`“.

Beispiel „Kreis“ in Java:

```
public class Kreis
 {
   private double x;
   private double y;
   private double radius;

   public void setzeKreis ( double x,
                            double y,
                            double radius )
    {
      this.x      = x;
      this.y      = y;
      this.radius = radius;
    }
    public void setzeKreis ( Kreis k )
     {
       x      = k.x;
       y      = k.y;
       radius = k.radius;
     }

 }
```

Klassenmethoden contra Objektvariable I:

```
public class MeineKorrekteKlasse
 {
   public        int n1;
   public static int n2;

   public void meineObjektMethode ()
    {
      System.out.println ( n1 );
      System.out.println ( n2 );
    }

   public static
    void meineKorrekteKlassenMethode
             ( MeineKorrekteKlasse weiteresObjekt )
    {
      System.out.println ( n2 );
      System.out.println ( weiteresObjekt.n1 );
      System.out.println ( weiteresObjekt.n2 );
      weiteresObjekt.meineObjektMethode();
    }
 }
```


Klassenmethoden contra Objektvariable II:

```
public class MeineFehlerhafteKlasse
 {
   public        int n1;
   public static int n2;

   public static void
    meineFehlerhafteKlassenMethode1 ()
    {
      System.out.println ( n1 );
    }

   public static void
    meineFehlerhafteKlassenMethode2 ()
    {
      System.out.println ( this.n1 );
    }
 }
```


Erläuterungen:

- *Erinnerung* an Folie 395 ff.: Zum Aufruf einer Klassenmethode bedarf es keines Objekts der Klasse.
- Objektvariable und -konstante gehören aber per Definition zu konkreten Objekten.
- Daher wäre es semantischer Unsinn, wenn eine Klassenmethode
  - ◇ mit „`this`“ auf dieses nicht unbedingt existierende Objekt selbst oder
  - ◇ mit oder ohne „`this`“ auf die Objektvariablen und -konstanten dieses Objekts zugreifen oder
  - ◇ eine andere Objektmethode mit diesem Objekt aufrufen dürfte.

Fortsetzung Erläuterungen:

- Andererseits spricht nichts dagegen (und ist auch absolut korrekt), wenn eine Klassenmethode wie „`meineKorrekteKlassenMethode`“ in Klasse „`MeineKorrekteKlasse`“ auf die Objektvariablen, -konstanten und -methoden eines benannten Objekts derselben Klasse wie etwa „`weiteresObjekt`“ zugreift.
- Philosophie in Java: Alles, was Unsinn produzieren könnte, ist in Java von Anfang an verboten!

Aspekt 3: Konstruktor:

- Ein *Konstruktor* einer Klasse ist eine bestimmte Art von Methode.
- Syntaktische Eigenarten:
  - ◇ Der Name der Methode muss zugleich der Name der Klasse selbst sein.
  - ◇ Bei einem Konstruktor wird kein Rückgabetyp angegeben (auch nicht „`void`“!).
- Eine Methode, die genau gleich wie ihre Klasse heißt, ist durch diese Namensgleichheit automatisch ein Konstruktor, und es darf daher kein Rückgabetyp angegeben werden.
- Konstruktoren sind die einzigen Methoden in Java, bei denen kein Rückgabetyp angegeben wird (wieder aus C++ übernommen).
- Analog zu Folie 380 ff. darf eine Klasse mehrere Konstruktoren haben.
- Absolut identische Regel wie bei „normalen“ Methoden: Verschiedene Konstruktoren müssen verschiedene Parameterlisten haben.

Aufruf von Konstruktoren:

- Ein Konstruktor einer Klasse wird normalerweise in einer einzigen spezifischen Situation aufgerufen: bei der Erzeugung eines Objekts dieser Klasse mit „`new`“ (vgl. Folie 306).
- Der Klassenname hinter „`new`“ ist also genauer gesagt der (identische) Name des Konstruktors bei seinem Aufruf:

```
String str = new String ( "Hello" );
                    ^^^^^^
```

- Damit wird auch klar, was die Klammern hinter dem Klassennamen in einem „`new`“–Ausdruck sollen:
  - ◇ Das ist einfach die Parameterliste für den Aufruf des Konstruktors.
  - ◇ Ein leeres Klammerpaar bedeutet dann, dass ein Konstruktor mit leerer Parameterliste aufgerufen wird.

Vorgreifende Bemerkung:

- Bis jetzt ist überhaupt nicht einsichtig, dass bei einer Zeile

  ```
  X x = new X ( ... );
  ```

  unbedingt zweimal das „`X`“ hingeschrieben werden muss.
- Beim Thema Vererbung werden wir sehen, dass die beiden Vorkommen des Klassennamens „`X`“ (also Typ der Variablen und Name des Konstruktors) sich in ganz speziellen Situationen durchaus unterscheiden können:

  ```
  X x = new Y ( ... );
  ```

  ⟶ Siehe Abschnitt 4.4, Folien 489 ff.
- Eben weil die beiden Klassen nicht immer identisch sind, steht also durchaus ein Sinn dahinter, dass man den Namen der Klasse in den bisher auftretenden Situationen zweimal hinschreiben musste.

Beispiele für Konstruktoren in der Standardbibliothek:

- ```
  String       str1 = new String       ( "Hallo" );
  StringBuffer str2 = new StringBuffer ( "Hallo" );
  ```
- ```
  Color c = new Color ( 1, 1, 0 );
  ```
  ⟶ Farbobjekt „`c`“ hat RGB–Wert (1,1,0) (also reines Gelb).
- ```
  Integer i = new Integer ( 1 );
  ```
  ⟶ Vgl. Folie 328.
- ```
  String       str1 = new String       ();
  StringBuffer str2 = new StringBuffer ();
  ```
  ⟶ Jeweils Konstruktor mit leerer Parameterliste.
  (Semantik: Die Zeichenkette ist leer.)

Erzwungener Aufruf von Konstruktoren:

- Wenn eine Klasse einen oder mehrere Konstruktoren hat, dann *muss* einer davon bei der Erzeugung eines Objekts dieser Klasse mit „new“ aufgerufen werden.
- Ansonsten gibt es einen Fehler beim Kompilieren.
- Bei der Implementation einer Klasse kann man mit diesem Mechanismus also erzwingen, dass die Initialisierung eines Objekts niemals vergessen werden kann:
  - ◇ Man gibt der Klasse eben einen oder mehrere Konstruktoren, die ein mit „new“ neu eingerichtetes Objekt der Klasse adäquat initialisieren.
  - ◇ Solange man bei der Einrichtung eines Objekts der Klasse keinen dieser Konstruktoren benutzt, wird das Programm halt nicht übersetzt, sondern der Compiler liefert eine Fehlermeldung.

Beispiel:

```
public class MeineKlasse
 {
     int    i;
     double d;
     char   c;
   public MeineKlasse ( int i )
    {
      this.i = i;
      d = 3.14;
      c = 'a';
    }
   public MeineKlasse ( double d )
    {
      i = 1;
      this.d = d;
      c = 'a';
    }
   public MeineKlasse ( int i, double d )
    {
      this.i = i;
      this.d = d;
      c = 'a';
    }
   public MeineKlasse ( int i, double d, char c )
    {
      this.i = i;
      this.d = d;
      this.c = c;
    }
   ...
 }
```


Fortsetzung des Beispiels:

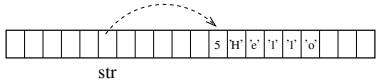
```
// 1. Konstruktor:
public MeineKlasse objekt1
  = new MeineKlasse (2);             // (2,3.14,'a')

// 2. Konstruktor:
public MeineKlasse objekt2
  = new MeineKlasse (2.71);          // (1,2.71,'a')

// 3. Konstruktor:
public MeineKlasse objekt3
  = new MeineKlasse (2, 2.71);       // (2,2.71,'a')

// 4. Konstruktor:
public MeineKlasse objekt4
  = new MeineKlasse (2, 2.71, 'b');  // (2,2.71,'b')

public MeineKlasse objekt5
  = new MeineKlasse (2.71, 2, 'b');
         // Fehler: Kein Konstruktor mit Parameter-
         // liste (double, int, char) definiert!

public MeineKlasse objekt6
  = new MeineKlasse ();
         // Fehler: Kein Konstruktor mit leerer
         // Parameterliste definiert!
```


Frage:

- Ist „adäquate Initialisierung“ wirklich so wichtig, dass man in Java (und anderen Programmiersprachen) so feste Regeln einführen muss?
- Was heißt eigentlich „adäquate Initialisierung“?

Antwort durch einfaches Beispiel:

- *Erinnerung*: Auf Folie 310 ff. wurde diskutiert, wie man die Klasse „java.lang.String“ intern realisieren könnte.
- Betrachten wir zum Beispiel die erste Variante:



- Bei der Initialisierung des String–Objekts mit „Hello“ muss unbedingt gewährleistet sein, dass die Anzahl der Zeichen als 5 initialisiert wird.

  ⟶ Sonst macht hinterher jeder Zugriff Blödsinn...

Technische Frage:

- Man kann ja Klassen schreiben, die überhaupt keinen Konstruktor haben.
- Trotzdem haben wir dann immer bei „`new`“ noch ein leeres Klammerpaar hinter den Klassennamen geschrieben.
- Also wie bei einem Konstruktor mit leerer Parameterliste.

Beispiel:

```
public class MeineKlasse
  {
    public int    i;
    public double d;
    public char   c;
  }

...

MeineKlasse meinObjekt = new MeineKlasse ();
```

⟶ Antwort auf der nächsten Folie.

Antwort:

- Falls eine Klasse überhaupt keine Konstruktoren hat, „denkt“ sich der Compiler einfach einen Konstruktor mit leerer Parameterliste hinzu. Dieser Konstruktor tut überhaupt nichts.
- *Sinn der Sache*: Vereinheitlichung der Schreibweise beim Aufruf von Konstruktoren.

  ⟶ Eben immer ein rundes Klammerpaar (mit oder ohne Inhalt) hinter dem Klassennamen.
- *Beachte*: Wenn eine Klasse Konstruktoren besitzt, dann „denkt“ sich der Compiler keinen Konstruktor hinzu.
- *Konsequenz*: Wenn man dann keinen Konstruktor mit leerer Parameterliste für die Klasse selbst implementiert hat, darf man *nicht* einfach ein leeres rundes Klammerpaar hinter den Klassennamen schreiben (vgl. Folie 447).
- *Letzte Frage dazu*: Wann immer man Konstruktoren selbst implementiert, warum dann nicht immer gleich auch einen Konstruktor mit leerer Parameterliste dazu, um sich das Leben beim Einrichten von Objekten einfacher zu machen?

Antwort durch Beispiel:

- In der Java–Standardbibliothek gibt es eine Klasse „`Date`“ (genauer: „`java.util.Date`“).
- Ein Objekt dieser Klasse speichert ein Datum ab (Tag, Monat, Jahr).
- „`Date`“ hat
  - ◇ einen Konstruktor mit drei „`int`“–Parametern für Tag, Monat und Jahr
  - ◇ sowie einen weiteren Konstruktor mit nur einem Parameter, der für technische Belange gut ist und hier nicht weiter interessiert.
- Ein Konstruktor mit leerer Parameterliste ist für „`Date`“ nicht implementiert.
- Wozu auch:
  - ◇ Welches Datum sollte ein derart initialisiertes „`Date`“–Objekt denn auch haben?
  - ◇ Es macht nur Sinn, ein „`Date`“–Objekt einzurichten, wenn man das Datum auch gleich festsetzt.

Abarbeitung des Konstruktors:

- *Erinnerung* an Folie 314 ff.:

  Wird eine Variable bei der Deklaration nicht sofort initialisiert, dann wird ihr Inhalt automatisch auf einen datentypspezifischen Null–Wert gesetzt.
- Das Gleiche gilt natürlich auch für die Datenkomponenten eines Objekts, das mit „`new`“ eingerichtet wurde.
- Erst nach dieser Initialisierung auf Null–Werte wird der Konstruktor aufgerufen.
- *Konsequenz*:

  Falls der Konstruktor nur einen Teil der Datenkomponenten explizit initialisiert, sind die anderen Datenkomponenten nicht uninitialisiert.

Konstruktor in Konstruktor aufrufen:

- Man kann einen Konstruktor einer Klasse auch in einem anderen Konstruktor aufrufen.
- *Syntax*: „`this`“ steht vor der Parameterliste.
- Dieser Aufruf eines Konstruktors in einem zweiten Konstruktor muss die allererste Anweisung im zweiten Konstruktor sein!

  ⟶ Sonst Fehlermeldung vom Compiler!
- Es gibt insgesamt nur drei Möglichkeiten überhaupt, wie man einen Konstruktor aufrufen kann:
  - ◇ Bei der Einrichtung eines Objektes mit „`new`“.
  - ◇ In der allerersten Zeile eines anderen Konstruktors derselben Klasse.

    ⟶ Was wir gerade betrachten.
  - ◇ In der allerersten Zeile einer abgeleiteten Klasse.

    ⟶ Betrachten wir erst auf Folie 500 ff.

Beispiel (vgl. Folie 446 ff.):

```
class Bla
 {
   private int i;
   private double d;
   private char c;

   public Bla ( int i, double d, char c )
    {
      this.i = i;
      this.d = d;
      this.c = c;
    }
   public Bla ( int i )
    {
      this ( i, 3.14, 'a' );
      System.out.println ( "Konstruktor 2" );
    }
   public Bla ( int i, char c )
    {
      this ( i, 3.14, c );
      System.out.println ( "Konstruktor 3" );
    }
 }
```


Nachbemerkung:

- Mit „`this`“ kann man
  - ◇ nicht nur Datenkomponenten und Konstruktoren ansprechen,
  - ◇ sondern generell auch alle anderen Methoden der Klasse.
- *Beispiel*: Mit „`this.f(1,3.14)`“ ruft man in einer Methode der Klasse „`X`“ die Methode

  „`X.f(int,double)`“

  mit dem Objekt „`this`“ auf.
- Ist aber eigentlich nicht wirklich in irgendeiner Situation nützlich.
- Dient auch eher zur Vereinheitlichung der Regeln: dass man nämlich alles von „`this`“ (jede Datenkomponente, jede Methode) über das Schlüsselwort „`this`“ ansprechen kann.

Abschnitt 4.3: Zugriffsrechte und Packages

- Bis jetzt haben wir (fast) immer unreflektiert das Schlüsselwort „`public`“ an den Anfang aller Deklarationen von Klassen, Methoden und Datenkomponenten gestellt.
- Auf den folgenden Folien wird (hoffentlich!) klar werden,
  - ◇ was es nun mit „`public`“ auf sich hat und
  - ◇ was es für Alternativen zu „`public`“ gibt.
- *Erinnerung* an Folie 416: Ein Datentyp ist charakterisiert durch
  - ◇ die möglichen abstrakten Zustände der Objekte dieses Datentyps und
  - ◇ die lesenden und verändernden Zugriffsmöglichkeiten.

Prinzip Datentyp und „public“:

- Die Idee hinter „`public`“ ist, dass bei einem neuen, selbstdefinierten Datentyp (sprich: Klasse)
  - ◇ die konkreten Zustände und ihre mitunter komplexen semantischen Konsistenzbedingungen versteckt werden und
  - ◇ der Benutzer der Klasse nur eine Art abstraktes Zustandsmodell nebst zugehörigen abstrakten Zugriffsmöglichkeiten zu sehen bekommt.
- *Einfaches Beispiel*: Die beiden Möglichkeiten zur internen Realisierung von Strings auf Folie 310 ff.
  - ◇ Konsistenzbedingung bei der 1. Möglichkeit: Die gespeicherte Anzahl der Zeichen muss stimmen.
  - ◇ Konsistenzbedingung bei der 2. Möglichkeit: Die erste Speicherstelle hinter dem letzten Zeichen muss das Null–Zeichen (Bitmuster 0) sein.

Packages:

- Mehrere Klassen können auch in einem *Package* zusammengefasst werden.
- Die Klassen in einem Package haben etwas mehr Zugriffsmöglichkeiten aufeinander.
- Ein Package kann neben Klassen auch weitere Packages enthalten.
  - ⟶ Ähnlich wie Directories, die ja neben Files auch wieder Directories enthalten können (Folie 91 ff.).
- Vergleiche Folie 401.

Klassen, Packages und Quelltext–Files:

- Man kann prinzipiell in einem Quelltext–File Klassen benutzen, die nicht im selben Quelltext–File implementiert sind.
- Sind zwei Quelltext–Files in derselben Directory (vgl. Folie 91 ff.), dann reicht der Name der Klasse ohne weitere Zusätze (vgl. Übungsaufgaben).

Beispiel:

- Betrachte eine Klasse „`MeineKlasse1`“ und eine Klasse „`MeineKlasse2`“.
- Wir nehmen an,
  - ◇ dass „`MeineKlasse1`“ im Quelltext von „`MeineKlasse2`“ vorkommt und
  - ◇ dass „`MeineKlasse1.java`“ und „`MeineKlasse1.java`“ in derselben Directory abgelegt sind.
- Mit dem Aufruf

```
javac MeineKlasse1.java MeineKlasse2.java
```

oder

```
javac MeineKlasse2.java MeineKlasse1.java
```

kann man dann beide Quelltexte zu einem gemeinsamen Java–Programm übersetzen (Zugriff auf beide „`.class`“–Dateien).

Packages in der Java–Standardbibliothek:

- Zur Benutzung von Klassen aus der Java–Standardbibliothek gibt man die genaue Spezifikation des Klassenpfades an bzw. fügt eine entsprechende „`import`“–Zeile ein (siehe Folie 376 ff.).
- Die Java–Standardbibliothek ist ein Beispiel für die auf Folie 456 ff. erwähnten Packages.
- Wie die Java–Standardbibliothek demonstriert, lässt sich sich ein Package wie „`java`“ hierarchisch in Subpackages wie „`java.awt`“, Subsubpackages wie „`java.awt.image`“ usw. aufgliedern.

Grundsätzlicher Mechanismus:

- Im Prinzip besteht jedes Quelltext–File aus einer Klasse.
- Der Name der Klasse muss identisch mit dem Namen des Quelltext–Files vor der Extension „`.java`“ sein.

  ⟶ Zum Beispiel „`public`“–Klasse „`MeineKlasse`“ in Quelltext–File „`MeineKlasse.java`“.

  ⟶ Zugehöriger Java Byte Code in File „`MeineKlasse.class`“.
- *Konsequenz*: Bei Klassen in der gleichen Directory weiß der Compiler daher ohne weitere Zusatzinformation, in welchem File er den Code der benutzten Klasse findet.
- Packages, Subpackages, Subsubpackages usw. werden in einer analogen Directorystruktur organisiert (vgl. Folie 91), deren Wurzel dem Compiler bekannt sein muss.

Beispiel: Java–Standardbibliothek

- Im Prinzip gibt es eine Wurzel–Directory für die Java–Standardbibliothek.
- Diese Wurzel–Directory ist einem Compiler wie „`javac`“ schon durch entsprechende Konfiguration bei der Installation bekannt gemacht worden, so dass man beim Aufruf von „`javac`“ nichts mehr tun muss.
- Der Klassenpfad jeder Klasse in der Standardbibliothek wird vom Compiler als ein relativer Directorypfad gemäß Folie 95 ff. interpretiert.
- *Beispiel*: „`java.awt.image.Raster`“
  - ◇ wird vom Compiler in File „`Raster.class`“ gesucht,
  - ◇ und zwar in der Directory, deren relativer Pfadname in Bezug auf die Wurzel–Directory „`java/awt/image`“ lautet.

  ⟶ „`java/awt/image/Raster.class`“

Technische Nachbemerkung:

- Aus Effizienzgründen ist die Java–Standardbibliothek technisch nicht ganz so organisiert wie hier beschrieben.

  ⟶ Daher die Einleitung „Im Prinzip“ am Anfang der letzten Folie.
- Aber für's Programmieren in Java spielt der Unterschied keine Rolle.

Packages:

- Ohne „`public`“ und „`private`“ dürfen nur die Methoden derjenigen Klassen, die im selben *Package* sind, auf eine Datenkomponente oder Methode zugreifen.

  $\longrightarrow$ *Frage*: Was ist ein Package?

- *1. Antwort*: Wenn man sich um Packages überhaupt nicht kümmert (wie wir in dieser Veranstaltung abgesehen von diesen paar Folien), dann ist einfach jede „`public`“–Klasse ein Package für sich.
- *2. Antwort*: Man kann auch die Klassen aus mehreren Quelltext–Files zu einem Package zusammenschnüren.

  $\longrightarrow$ Wie das genau zu bewerkstelligen ist, lassen wir hier aus Zeitgründen aus.

  $\longrightarrow$ Sieht auch bei jedem Java–Compiler/Interpreter etwas anders aus.

Organisation von Packages:

- Soll ein Package in Subpackages, Subsubpackages usw. zerlegt werden, dann müssen die „`.class`“–Files in einer Directory–Hierarchie analog zu Folie 463 organisiert sein.
- Die Wurzel–Directory dieser Directory–Hierarchie muss dem Compiler durch globale Konfiguration oder durch Option beim Aufruf bekannt gemacht werden.
- Danach funktionieren die hierarchischen Klassenpfade bzw. die entsprechenden „`import`“–Zeilen wie bei der Standardbibliothek (vgl. Folie 463).

Zugriff auf Datenkomponenten und Methoden:

- Eine Datenkomponente oder Methode
  - ◇ darf in den Methoden jeder beliebigen Klasse angesprochen werden, wenn sie durch „`public`“ gekennzeichnet ist,
  - ◇ nur in den Methoden derselben Klasse, wenn sie durch „`private`“ gekennzeichnet sind, und
  - ◇ in den Methoden aller Klassen desselben Packages, wenn sie weder durch „`public`“ noch durch „`private`“ oder irgendetwas anderes gekennzeichnet sind.
- Was bedeutet „irgendetwas anderes“:
  - ◇ Neben „`public`“ und „`private`“ gibt es noch „`protected`“.
  - ◇ Diese Möglichkeit wird erst später betrachtet (Abschnitt 4.4, Folie 489 ff.).

Beispiel zu „public“ und „private“:

```
public class MeineKlasse
 {
   public  int n1;
   private int n2;

   public  void f1 ()  { ... }
   private void f2 ()  { ... }
 }

...

MeineKlasse meinObjekt = new MeineKlasse();

meinObjekt.n1 = 1;    // Erlaubt
meinObjekt.n2 = 1;    // Verboten!
meinObjekt.f1();      // Erlaubt
meinObjekt.f2();      // Verboten!
```

Realistischeres Fallbeispiel:

```
public class Hoererschaften
 {
   private int[][] hoererInVeranstaltung;
   private int[][] gehoerteVeranstaltungen;

   public int[] hoererschaft ( int veranstaltungsNr )
    {
      return hoererInVeranstaltung [veranstaltungsNr];
    }
   public int[] veranstaltungen ( int matrikelNr )
    {
      return gehoerteVeranstaltungen [matrikelNr];
    }
   public void fuegeEin ( int veranstaltungsNr,
                          int matrikelNr )
    {
      ...
    }
   public void entferne ( int veranstaltungsNr,
                          int matrikelNr )
    {
      ...
    }

   ...
 }
```


Erläuterungen:

- Ein Objekt der Klasse „`Hoererschaften`“ soll die Hörerschaften der einzelnen Veranstaltungen bspw. in einem Fachbereich verwalten.
- Es gibt zwei lesende Methoden:
  - ◇ „`hoererschaft`“ zur Ausgabe der Hörerschaft einer einzelnen Veranstaltung sowie
  - ◇ „`veranstaltungen`“ zur Ausgabe der von einem einzelnen Hörer besuchten Veranstaltungen.
- Ebenso gibt es zwei verändernde Methoden:
  - ◇ „`fuegeEin`“ zum Einfügen eines neuen Hörers in eine Veranstaltung sowie
  - ◇ „`entferne`“ zum Löschen eines solchen Eintrags.
- Die Datenkomponenten sind „`private`“ deklariert.

  ⟶ Nur die Methoden der Klasse „`Hoererschaften`“ selbst dürfen darauf zugreifen.

Interne Details von „Hoererschaften“:

- Es gibt insgesamt

  `hoererInVeranstaltungen.length`

  viele Veranstaltungen und

  `gehoerteVeranstaltungen.length`

  viele Hörer.
- Die Veranstaltungen sind mit

  `0...hoererInVeranstaltung.length-1`

  durchnumeriert, die Hörer mit

  `0...gehoerteVeranstaltungen.length-1`.
- Für Veranstaltungsnummer `i` aus

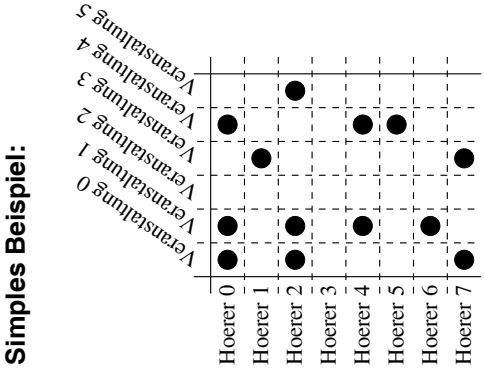
  `0...hoererInVeranstaltung.length-1`

  enthält `hoererInVeranstaltung[i]` die Hörer der Veranstaltung Nr. `i`, und für Hörer Nr. `j` aus

  `0...gehoerteVeranstaltungen.length-1`

  enthält `gehoerteVeranstaltungen[j]` die Veranstaltungen, die von Hörer Nr. `j` besucht werden.

Simples Beispiel:



„hoererInVeranstaltung“:

- hoererInVeranstaltung[0][0] == 0
- hoererInVeranstaltung[0][1] == 2
- hoererInVeranstaltung[0][2] == 7
- hoererInVeranstaltung[1][0] == 0
- hoererInVeranstaltung[1][1] == 2
- hoererInVeranstaltung[1][2] == 4
- hoererInVeranstaltung[1][3] == 6
- hoererInVeranstaltung[3][0] == 1
- hoererInVeranstaltung[3][1] == 7
- hoererInVeranstaltung[4][0] == 0
- hoererInVeranstaltung[4][1] == 4
- hoererInVeranstaltung[4][2] == 5
- hoererInVeranstaltung[5][0] == 2

Fortsetzung simples Beispiel:

„gehoerteVeranstaltungen“:

- gehoerteVeranstaltungen[0][0] == 0
- gehoerteVeranstaltungen[0][1] == 1
- gehoerteVeranstaltungen[0][2] == 4
- gehoerteVeranstaltungen[1][0] == 3
- gehoerteVeranstaltungen[2][0] == 0
- gehoerteVeranstaltungen[2][1] == 1
- gehoerteVeranstaltungen[2][2] == 5
- gehoerteVeranstaltungen[4][0] == 1
- gehoerteVeranstaltungen[4][1] == 4
- gehoerteVeranstaltungen[5][0] == 4
- gehoerteVeranstaltungen[6][0] == 1
- gehoerteVeranstaltungen[7][0] == 0
- gehoerteVeranstaltungen[7][1] == 3

Methode „fuegeEin“:

```
public void fuegeEin ( int veranstaltungsNr, int matrikelNr )
  {
    int[] alteHoerer = hoererInVeranstaltung[veranstaltungsNr];
    int[] neueHoerer = new int [ alteHoerer.length+1 ];
    for ( int i=0; i<alteHoerer.length; i++ )
      neueHoerer[i] = alteHoerer[i];
    neueHoerer[neueHoerer.length-1] = matrikelNr;
    hoererInVeranstaltung[veranstaltungsNr] = neueHoerer;

    int[] alteVeranstaltungen = gehoerteVeranstaltungen[matrikelNr];
    int[] neueVeranstaltungen = new int [ alteVeranstaltungen.length+1 ];
    for ( int i=0; i<alteVeranstaltungen.length; i++ )
      neueVeranstaltungen[i] = alteVeranstaltungen[i];
    neueVeranstaltungen[neueHoerer.length-1] = veranstaltungsNr;
    gehoerteVeranstaltungen[matrikelNr] = neueVeranstaltungen;
  }
```


Fortsetzung Erläuterungen:

- In der Klasse „`Hoererschaften`“ werden dieselben Daten also zweimal abgespeichert.
- So lassen sich die Methoden „`hoererschaft`“ und „`veranstaltungen`“ sehr einfach implementieren, und sie laufen auch noch schnell.

  ⟶ Siehe Folie 469 für eine solche Implementation.
- In typischen Anwendungen wird nach einer Aufbauphase kaum noch eingefügt oder entfernt, aber viel abgefragt.

  ⟶ In solchen Anwendungen ergibt eine Realisierung wie auf Folie 469 bestmögliche Effizienz.
- *Aber*: Das ist erkauft worden durch ein Konsistenzproblem:

  Die Daten in „`hoererInVeranstaltung`“ einerseits und in „`gehoerteVeranstaltungen`“ andererseits müssen penibel miteinander konsistent gehalten werden.

Fortsetzung Erläuterungen:

- Dadurch, dass „`hoererInVeranstaltung`“ und „`gehoerteVeranstaltungen`“ als „`private`“ deklariert wurden, können nur die Methoden von „`Hoererschaften`“ auf diese Daten zugreifen.
- *Konsequenz*:
  - ◇ Nur diese Methoden müssen die Konsistenzbedingung beachten.
  - ◇ Jedes andere Stück Quelltext kann nur indirekt mittels dieser Methoden auf die Daten zugreifen.

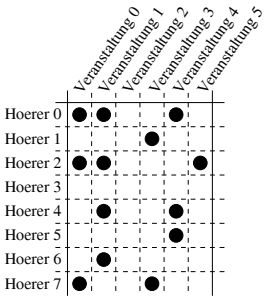
    ⟶ Es ist völlig unmöglich, dass ein solches Stück Quelltext die Konsistenzbedingungen verletzt.
  - ◇ Sofern die Methoden „`fuegeEin`“ und „`entferne`“ fehlerfrei programmiert sind, kann die Konsistenzbedingung also garantiert durch nichts und niemanden verletzt werden.

Warnung:

- So wie die Klasse „`Hoererschaften`“ hier realisiert wurde, ist sie alles andere als korrekt und sicher.
- *Konkretes Beispiel*:
  - ◇ *Erinnerung* an Folie 314 ff.: In der Realisierung auf Folie 469 werden die Datenkomponenten „`hoererInVeranstaltung`“ und „`gehoerteVeranstaltungen`“ auf „`null`“ gesetzt.
  - ◇ Methode „`fuegeEin`“ auf Folie 474 greift aber blindlings auf diese beiden Datenkomponenten zu, ohne diesen Fall geeignet zu behandeln.
  - ◇ *Ergebnis*: Das Programm würde abstürzen.
- Solche Mängel der Klasse „`Hoererschaften`“ nehmen wir hier bewusst in Kauf, um uns auf den eigentlichen Gedankengang zu konzentrieren.

Klasse „Hoererschaften“ als abstrakter Datentyp:

- Die Klasse „`Hoererschaften`“ bildet im Grunde erst durch „`private`“ einen abstrakten Datentyp gemäß Folie 416 ff.
- *Abstrakte Zustände*: Die verschiedenen möglichen Listen von Hörerschaften aller Veranstaltungen, also:



- *Konkrete Zustände*: Die verschiedenen Zustände der beiden „`private`“ deklarierten Datenkomponenten „`hoererInVeranstaltung`“ und „`gehoerteVeranstaltungen`“.

Allgemeine Philosophie dahinter: Einkapselung

- Wenn die Daten der Objekte einer Klasse Konsistenzbedingungen erfüllen müssen, dann sollten sie „`private`“ sein.

  ⟶ Nur die Methoden der zugehörigen Klasse müssen sich mit diesen Konsistenzbedingungen „herumschlagen“.

  ⟶ Unnötige Konsistenzfehler bei der Benutzung eines Objekts dieser Klasse sind von vornherein ausgeschlossen.
- Wenn die Daten der Objekte einer Klasse kompliziert zu verwalten sind, dann sollten sie „`private`“ sein (siehe Folie 310 ff. für ein reales Beispiel).

  ⟶ Nur die Methoden der zugehörigen Klasse müssen sich mit der Komplexität „herumschlagen“.

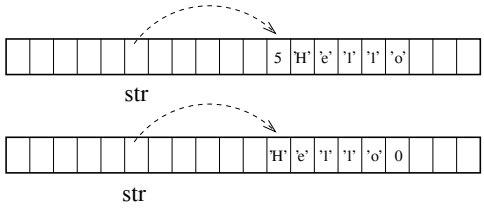
Fortsetzung
Allgemeine Philosophie Einkapselung:

- Wenn sich die konkrete Organisation der Daten eines Objekts vermutlich im Lauf der Zeit ändern wird, dann sollten sie „`private`“ sein.

  ⟶ Nur die Methoden der Klasse selbst müssen an eine veränderte Organisation angepasst werden.
- Beispiel: Arrays durch Listen ersetzen in Klasse „`Hoererschaften`“ (vgl. Folie 469).

Konkretes Beispiel zum letzten Punkt:

- *Erinnerung*: Auf Folie 310 ff. wurden zwei Möglichkeiten vorgestellt, wie man Objekte der Klasse „`String`“ intern verwalten könnte.



- Vermutlich wird eine dieser beiden Möglichkeiten tatsächlich verwendet.

⟶ Fortsetzung des Gedankenganges auf der nächsten Folie.

Fortsetzung Gedankengang:

- Irgendwann könnte man durchaus auf die Idee kommen, dass die andere, nicht verwendete Möglichkeit aus irgendwelchen Gründen besser ist.
- Dann kann man einfach die „`private`“–Datenkomponenten und den Code in den Methoden anpassen, neu kompilieren, fertig.
- Durch „`private`“ ist garantiert, dass kein anderes Stück Quelltext außer den Methoden von „`String`“ selbst mit den Datenkomponenten von „`String`“ direkt umgeht.
- *Konsequenz*: Sämtliche Java–Programme, die Strings benutzen, laufen hinterher weiterhin genauso wie vorher (sämtliche Java–Programme, die Strings *nicht* benutzen, natürlich auch).

Klassen und „public“:

- Die Aussage von Folie 462, dass jedes Quelltext–File genau eine Klasse enthält, war dort aus gutem Grund nur als Aussage „im Prinzip“ formuliert.
- *Genauere Formulierung*:
  - ◇ Es gibt in jedem Quelltext–File maximal eine Klasse, die „`public`“ deklariert ist.
  - ◇ Der Name dieser Klasse (sofern vorhanden) muss mit dem Namen des Files nach der Regel von Folie 462 übereinstimmen (das heißt, mit Endung „`.java`“ bzw. „`.class`“).
- Die anderen Klasse haben anstelle von „`public`“ gar nichts (auch kein „`private`“).
- Nur die „`public`“–Klasse ist in anderen Quelltext–Files verwendbar.
- Die anderen Klassen im Quelltext–File sind für andere Quelltext–Files schlicht nichtexistent.

  ⟶ Der Zugriff darauf von einem anderen Quelltext–File aus resultiert in einen Fehler beim Kompilieren.

Frage:

Wozu weitere Klassen in einer Quelldatei, auf die man in anderen Quelldateien nicht zugreifen kann?

Antwort:

- Oft verbessert es die Programmstruktur, wenn man Teile einer Klasse „`X`“ in eigenständige Klassen auslagert, ohne an den „`public`“–Deklarationen irgendetwas zu ändern.
- Die Existenz dieser Teile ist dann ein Detail, das nach Logik der Einkapselung eigentlich eingekapselt gehört.
- Das heißt, es ist wünschenswert, wenn keine Klasse außer „`X`“ diese weiteren Klassen zu sehen bekommt.
- Andererseits muss „`X`“ diese Klassen aber sehen, weil „`X`“ ja darauf aufbaut.
- *Lösung* in Java: Ins File „`X.java`“ als weitere Klassen (und dann eben *nicht* „`public`“) mit hineinpacken.

Beispiel:

- Wir betrachten noch einmal das Beispiel von Folie 469 ff.
- Im Grunde zerfällt die Klasse „`Hoererschaften`“ ja in zwei völlig voneinander unabhängige Teile:
  - ◇ „`hoererInVeranstaltung`“
  - ◇ „`gehoerteVeranstaltungen`“
- Es kann die Lesbarkeit des Quelltextes durchaus erhöhen, wenn diese beiden Teile
  - ◇ nicht einfach in einer Klasse zusammengepackt,
  - ◇ sondern in zwei separate Klassen ausgelagert werden.
- Da diese beiden Teile im Grunde identisch sind, kann man sie sogar beide durch eine einzige Klasse realisieren.

  ⟶ Doppelter Quelltext wird eingespart.

  ⟶ Quelltext, der eingespart ist, kann auch keine Programmierfehler enthalten.

Konkrete Realisierung:

```
class ArrayVonArrays
  {
    private int[][] daten;

    public int[] einzelnerArray ( int n )
      {
        return daten[n];
      }

    public void fuegeEin ( int i, int j )
      {
        // Fuegt neuen Wert 'j' hinten in Array Nr. 'i'
        // ein, so wie in der bisherigen Realisierung
        // in der Methode Hoererschaften.fuegeEin.
      }
  }
```


Fortsetzung konkrete Realisierung:

```
public class Hoererschaften
 {
   private ArrayVonArrays  daten1;
   private ArrayVonArrays  daten2;

   public int[] hoererschaft ( int veranstaltungsNr )
    {
      return daten1.einzelnerArray ( veranstaltungsNr );
    }

   public int[] veranstaltungen ( int matrikelNr )
    {
      return daten2.einzelnerArray ( matrikelNr );
    }

   public void fuegeEin ( int veranstaltungsNr,
                          int matrikelNr )
    {
      daten1.fuegeEin ( veranstaltungsNr, matrikelNr );
      daten2.fuegeEin ( veranstaltungsNr, matrikelNr );
    }
 }
```


Abschließende Bemerkungen zum Beispiel:

- Auch in dieser Fortsetzung des Beispiels von Folie 469 sind wieder viele wichtige Details ausgelassen worden, die die Klassen erst korrekt und sicher machen würden (vgl. Folie 477).
- Ob es Sinn macht oder nicht, Klassen so zu zerlegen,
  - ◇ hängt immer auch vom konkreten Einzelfall ab
  - ◇ und ist auch ein Stück weit Geschmackssache.
- Vielleicht
  - ◇ kommen die Vorteile in einem solchen Minibeispiel noch nicht so gut heraus,
  - ◇ in einem richtig großen Beispiel aber ganz sicher.
- In richtig großen Programmen ist eine gut gewählte Zerlegung in überschaubare Teile sogar das wichtigste überhaupt für Erfolg oder Scheitern des Projekts.

Abschnitt 4.4: Vererbung und Polymorphie

Aspekt 1: Vererbung

Erinnerung an Folie 302 und anderswo:

- Eine Klasse kann eine andere erweitern.
- Die erstere Klasse kann dann die Funktionalität der zweiten Klasse quasi „gratis“ mitbenutzen.
- Dass eine Klasse eine Erweiterung darstellt, wird mit Schlüsselwort „`extends`“ angezeigt:

```
public class MeinApplet extends Applet ...
```

- To extend = erweitern

Vererbung von Datenkomponenten:

```
public class BasisKlasse
 {
   public int n1;
 }

public class AbgeleiteteKlasse
 extends BasisKlasse
 {
   public int n2;
 }

...

BasisKlasse       x1 = new BasisKlasse();
AbgeleiteteKlasse x2 = new AbgeleiteteKlasse();

System.out.println ( x1.n1 );
System.out.println ( x2.n2 );
System.out.println ( x2.n1 );   //  <- !!
```


Erläuterungen:

- *Terminologie*:
  - ◇ Ein „Urstamm“ von Erweiterungen heißt meist *Basisklasse*.
  - ◇ Von einer „Erweiterung“ einer Basisklasse sagen wir, sie sei von der Basisklasse *abgeleitet*.
- Wenn eine Klasse wie „`AbgeleiteteKlasse`“ von einer Klasse wie „`BasisKlasse`“ abgeleitet wird, dann „erbt“ sie deren Datenkomponenten.
- Methoden werden im Prinzip genauso vererbt (siehe Folie 494 ff.).
- Eine abgeleitete Klasse darf natürlich ihrerseits als Basisklasse für weitere abgeleitete Klassen fungieren.

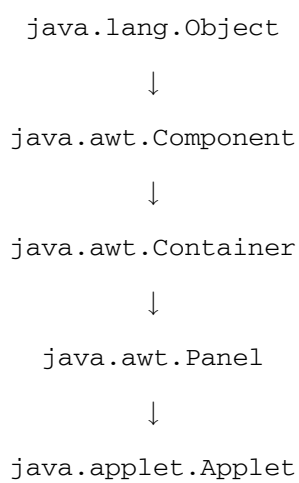
  $\longrightarrow$ Die Klassen in der Java–Standardbibliothek sind in vielen, durchaus langen, sich verzweigenden Ketten von gegenseitigen Vererbungsbeziehungen organisiert.

Ketten von Vererbungsbeziehungen:

- Damit ist gemeint, dass eine Klasse „`A`“ von einer Klasse „`B`“ abgeleitet ist, „`B`“ wiederum von einer Klasse „`C`“, „`C`“ von einer Klasse „`D`“ usw.
- Wir sagen, „`A`“ ist von „`B`“, „`B`“ von „`C`“ usw. *direkt* abgeleitet,
- Daraus ergeben sich auch *indirekte* Vererbungsbeziehungen:
  - ◇ „`A`“ ist von „`C`“,
  - ◇ „`A`“ ist von „`D`“,
  - ◇ „`A`“ ist von „`E`“,
  - ◇ „`B`“ ist von „`D`“,
  - ◇ „`B`“ ist von „`E`“,
  - ◇ „`C`“ ist von „`E`“

  indirekt abgeleitet.

Konkretes Beispiel aus der Java–Standbibliothek:



Vererbung von Methoden:

```
public class BasisKlasse
 {
   public void f1 ()
    {
      System.out.println ( "f1" );
    }
 }

class AbgeleiteteKlasse
 extends BasisKlasse
 {
   public void f2 ()
    {
      System.out.println ( "f2" );
    }
 }

BasisKlasse  x1
   = new BasisKlasse();
AbgeleiteteKlasse  x2
   = new AbgeleiteteKlasse();

x1.f1 ();
x2.f2 ();
x2.f1 ();   //  <- !!
```

- Analog zu Folie 490

Beispiel zu Sinn und Zweck dieser Vererbung:

- In der Klasse „`java.applet.Applet`“ ist alles implementiert, was man braucht, um ein Fenster zu öffnen und zu verwalten.
- Wenn man eine eigene Klasse von „`java.applet.Applet`“ ableitet, dann erbt die eigene Klasse alle diese Funktionalität von „`Applet`“ automatisch.
- Man braucht sich also um solche wirklich (wirklich!) lästigen technischen Details nicht zu kümmern.
- Statt dessen kann man sich voll und ganz auf die Logik der eigenen Applet–Klasse kümmern:
  - ◇ Bisher hat uns bei der Logik von Applets nur die Methode „`paint`“ interessiert.
  - ◇ Zum Beispiel auch „`init`“ und „`repaint`“ (siehe Folie 391) können aber ebenfalls interessant sein.

Regeln zur Vererbung von Methoden:

- Jede Methode der Basisklasse ist automatisch auch eine Methode der abgeleiteten Klasse:
  - ◇ exakt dieselbe Signatur (natürlich abgesehen vom Klassennamen),
  - ◇ exakt dieselbe Implementation.
- Einzige Ausnahme: Konstruktoren werden nicht in diesem Sinne vererbt.
  - ⟶ Sind meist zu eng von ihrer Logik her mit „ihrer“ Klasse verbunden.

Aspekt 2: Überschreibung von Methoden:

- Wenn eine Methode in der Basisklasse vorhanden ist, ist sie auf jeden Fall auch in der abgeleiteten Klasse vorhanden.
- Es gibt aber zwei Möglichkeiten, wie sie in der abgeleiteten Klasse vorhanden sein kann:
  - ◇ von der Basisklasse vererbt (wie bisher betrachtet);
  - ◇ in der abgeleiteten Klasse *überschrieben*.
- Was heißt Methode überschreiben:

  Eine Methode mit exakt identischer Signatur(!) wie in der Basisklasse wird in der abgeleiteten Klasse noch einmal definiert.

Achtung:

- Das Ausrufezeichen hinter „mit exakt identischer Signatur“ auf der letzten Folie hatte schon einen guten Grund.
- Es ist durchaus auch möglich, den Namen einer Methode, die in der Basisklasse schon definiert wurde, in der abgeleiteten Klasse für eine weitere Methode *mit anderer Signatur* zu verwenden.
- Das ist dann aber keine Vererbung, sondern nur ein Fall von Überladung (siehe Folie 380 ff.):
  - ◇ Eben eine neue Methode,
  - ◇ die halt „zufällig“ genauso heißt.
- *Einfache allgemeine Regel*:

  Wenn zwei Methoden gleichen Namens unterschiedliche Signatur haben, handelt es sich *immer* um Überladung.

  ⟶ Die beiden Parameterlisten müssen wie immer unterschiedlich sein.

Beispiel zu Überschreibung/-ladung:

```
public class BasisKlasse
 {
   public void f ( int i )
    {
      System.out.println ( "1" );
    }
 }

public class AbgeleiteteKlasse extends BasisKlasse
 {
   public void f ( int j )  // ueberschrieben
    {
      System.out.println ( "2" );
    }
   public void f ( double d ) // ueberladen
    {
      System.out.println ( "3" );
    }
 }

...

BasisKlasse       x = new BasisKlasse();
AbgeleiteteKlasse y = new AbgeleiteteKlasse();

x.f (1);     // -> "1"
y.f (1);     // -> "2"
y.f (3.14);  // -> "3"
x.f (3.14);  // Fehler: BasisKlasse.f(double)
             //         existiert nicht!
```


Schlüsselwort „super“:

- Betrachte wieder die allgemeine Situation, dass eine Methode „`f`“ in einer BasisKlasse „`X`“ definiert und in einer von „`X`“ abgeleiteten Klasse „`Y`“ überschrieben ist.
- In dieser Methode von „`Y`“ oder auch in anderen Methoden von „`Y`“ möchte man aber vielleicht auch gerne „`X.f`“ benutzen.
- Aber durch die Überschreibung ist in den Methoden von „`Y`“ zunächst einmal nur „`X.f`“ ansprechbar.

  ⟶ Ähnlich wie bei „`this`“ (Folie 432 ff.).
- Mit Schlüsselwort „`super`“ kann man die Methoden der Basisklasse in den Methoden der abgeleiteten Klasse aufrufen.

  ⟶ Egal ob in der abgeleiteten Klasse überschrieben oder nicht.
- Die Syntax ist wie bei „`this`“.
- Geschachtelte Aufrufe von Methoden indirekter Basisklassen über „`super.super`“, „`super.super.super`“ usw. sind nicht erlaubt.

Beispiel:

```
public class X
 {
   public void f ()  { ... }
 }

public class Y extends X
 {
   public void f ()
    {
      System.out.println ( "Vorher" );
      super.f(); // -> X.f()
      System.out.println ( "Hinterher" );
    }

   public void g ()
    {
      System.out.println ( "Vorher" );
      super.f();  // -> X.f()
      System.out.println ( "Hinterher" );
      f();  // -> Y.f()
    }
 }
```


Konstruktoren der Basisklasse:

- *Erinnerung* an Folie 496: Konstruktoren werden nicht vererbt.
- Trotzdem möchte man im Konstruktor der abgeleiteten Klasse oft auch den Konstruktor der Basisklasse aufrufen, um den Anteil der Basisklasse am Gesamtobjekt mit einem dafür schon vorgesehenen Konstruktor zu initialisieren.
- Mit Schlüsselwort „`super`“ kann man einen Konstruktor der Basisklasse in einem Konstruktor der abgeleiteten Klasse aufrufen.

  $\longrightarrow$ Auf Folie 453 schon angekündigt.

Regeln:

- Nur die allererste Zeile in einem Kontruktor darf der Aufruf eines anderen Konstruktors mit „`this`“ oder „`super`“ sein.

  $\longrightarrow$ Insbesondere darf es in jedem Konstruktor nur einen einzigen Aufruf eines anderen Konstruktors geben.
- Auch hier wird dafür gesorgt, dass immer ein Konstruktor aufgerufen werden muss:
  - ◇ Wenn die direkte Basisklasse keinen Konstruktor mit leerer Parameterliste hat (weder selbst geschrieben noch vom Compiler hinzugedacht), dann **muss** ein Konstruktor mit „`super`“ auf diese Art aufgerufen werden.
  - ◇ Wenn die direkte Basisklasse einer Klasse X einen Konstruktor mit leerer Parameterliste hat und die erste Zeile eines Konstruktors von X kein Konstruktoraufruf ist, wird dieser Konstruktor der Basisklasse implizit am Anfang aufgerufen

Beispiel:

```
public class Auto
 {
    public Auto() { ... }
 }

public class Kombi extends Auto
 {
    public Kombi()
     {
        super();  // OK!
        ...
     }
 }

public class Cabrio extends Auto
 {
    public Cabrio()
     {
        // irgendwelche Variablen werden
        // initialisiert
        ...
        super(); // -> Fehler!
     }
 }
```

Aspekt 3: Polymorphie:

```
public class BasisKlasse
 {
   public void f ()
    {
      System.out.println ( "Basisklasse" );
    }
 }

public class AbgeleiteteKlasse
 extends BasisKlasse
 {
   public void f ()
    {
      System.out.println ( "Abgeleitete Klasse" );
    }
 }
...
```


Fortsetzung: Polymorphie

```
BasisKlasse  x1
   = new BasisKlasse();

AbgeleiteteKlasse  x2
   = new AbgeleiteteKlasse();

BasisKlasse  x3
   = new AbgeleiteteKlasse();  // <- !!!

x1.f ();   //  -> "BasisKlasse.f()"
x2.f ();   //  -> "AbgeleiteteKlasse.f()"
x3.f ();   //  -> "AbgeleiteteKlasse.f()" !!!
```

⟶ Vergleiche Folie 499

Erläuterungen:

- Eine Variable vom Typ der Basisklasse kann also auch auf ein Objekt vom Typ einer davon abgeleiteten Klasse verweisen:

```
AbgeleiteteKlasse x = new AbgeleiteteKlasse();
BasisKlasse       y = x;
```

- Wie im Beispiel auf der letzten Folie lassen sich diese beiden Anweisungen selbstverständlich auch zu einer Anweisung zusammenfassen:

```
BasisKlasse x = new AbgeleiteteKlasse();
```


Fortsetzung Erläuterungen:

- Das ist die auf Folie 443 angesprochene spezifische Situation: In einer Zeile „`X x = new Y (...);`“
  - ◇ muss „`Y`“ nicht unbedingt identisch mit „`X`“ sein,
  - ◇ sondern kann auch von „`X`“ abgeleitet sein.
- Das überflüssig erscheinende, aber erzwungene doppelte Hinschreiben von „`X`“ in einer Zeile

  „`X x = new X (...);`“

  enthält also doch eine notwendige Information.

Formaler und aktualer Typ:

- Wegen solcher Möglichkeiten der Form

  „`X x = new Y (...);`“

  hat das Objekt hinter der Variablen „`x`“ also nicht unbedingt denselben Typ wie die Variable „`x`“ selbst.
- Bei einer Variablen wie „`x`“, die von einem Klassentyp ist, müssen wir daher sorgfältig zwischen zwei Typen unterscheiden:
  - ◇ dem *formalen* Typ: dem Typ der *Variablen* (also „`BasisKlasse`“), und
  - ◇ dem *aktualen* Typ: dem Typ des *Objekts* (also „`AbgeleiteteKlasse`“).
- Bei einer Zeile der Form

  „`X x = new X (...);`“

  sind aktualer Typ und formaler Typ von „`x`“ zunächst einmal identisch.
- What's in a name: *actual* = engl. *tatsächlich* (nicht etwa *aktuell*)!

Fortsetzung formaler und aktualer Typ:

- Der formale Typ einer Variablen bleibt immer gleich.
- Der aktuale Typ kann sich jederzeit ändern.

  ⟶ Einfach durch Zuweisung eines neuen Objekts an die Variable wird der Typ des neuen Objekts der neue aktuale Typ:

```
X x = new Y (...);      // Aktualer Typ von
                        // 'x' ist 'Y'

Z z = new Z (...);

x = z;                  // Aktualer Typ von
                        // 'x' ist nun 'Z'
```


Noch Fortsetzung formaler und aktualer Typ:

- Es gibt nur zwei Möglichkeiten für eine Variable von einem Klassentyp:
  - ◇ Der aktuale und der formale Typ sind identisch.
  - ◇ Der aktuale Typ ist direkt oder indirekt vom formalen Typ abgeleitet.

    ⟶ Vgl. Folie 492 für direkte vs. indirekte Vererbung.
- *Warum* diese Einschränkung:
  - ◇ Der formale Typ fungiert als „Fassade“, hinter dem Objekte von allen möglichen aktualen Typen stecken können.
  - ◇ Dazu ist es natürlich notwendig, dass der aktuale Typ alles kann, was der formale Typ „verspricht“.
  - ◇ Das ist am einfachsten und elegantesten eben durch diese Einschränkung gewährleistet.

Genauere Regeln:

- Welche Methoden mit einer Variablen eines Klassentyps aufgerufen werden dürfen, hängt von seinem *formalen* Typ ab.
- Falls eine Methode für Basisklasse und abgeleitete Klasse(n) implementiert ist (so wie „`f`“ auf Folie 505):

  Dann wird die Implementation des aktualen Typs aufgerufen.
- Falls eine Methode nur für die Basisklasse, aber nicht für die abgeleitete Klasse implementiert ist:

  Dann wird die Implementation der Basisklasse auf die abgeleitete Klasse vererbt.

  ⟶ Dann ist die Implementation der Basisklasse in jedem Fall auch die Implementation des aktualen Typs.

Konsequenz (am Beispiel illustriert):

- Wo immer eigentlich ein „`Applet`“–Objekt erwartet wird,
- zum Beispiel auch im Interpreter „`appletviewer`“,
- kann statt dessen auch ein Objekt einer davon abgeleiteten Klasse verwendet werden,
- und soweit „`Applet`“–Methoden wie „`paint`“ oder „`init`“ oder „`repaint`“ (vgl. Folie 391) in der abgeleiteten Klasse implementiert sind (mit identischer Signatur, vgl. Folie 497 ff.),
- werden diese Implementationen verwendet.
- „`appletviewer`“ und co. bekommen also Variablen vom formalen Typ „`Applet`“, dahinter kann aber auch ein Objekt vom aktualen Typ „`MeinApplet`“ stehen („`MeinApplet`“ wurde von „`Applet`“ abgeleitet).

Beispiel zur vorletzten Folie:

```
public class BasisKlasse
 {
   public void f1 ()
    {
      System.out.println ( "1" );
    }
 }

public class AbgeleiteteKlasse extends BasisKlasse
 {
   public void f1 () // Ueberschrieben
    {
      System.out.println ( "2" );
    }
   public void f2 () // Neu in abgeleiteter Klasse
    {
      System.out.println ( "3" );
    }
 }
...

BasisKlasse x = new AbgeleiteteKlasse();

x.f1();  // -> "2"
x.f2();  // Verboten: zwar definiert fuer den
         //   aktualen Typ ("AbgeleiteteKlasse"),
         //   aber nicht fuer den formalen Typ
         //   ("BasisKlasse").
```


Frage:

- Warum dürfen denn eigentlich nur die Methoden aufgerufen werden, die für den *formalen* Typ schon definiert sind.
- Der Compiler kann doch den aktualen Typ „erkennen“ und dementsprechend auch alle Methoden erlauben, die erst für den *aktualen* Typ definiert sind.

Einfache Antwort: Nein!

- Es stimmt leider nicht immer, dass der Compiler den aktualen Typ einer Variable erkennen kann.
- Der aktuale Typ einer Variablen kann von Informationen abhängen, die erst zur Laufzeit des Programms zur Verfügung stehen (und sich auch von Programmlauf zu Programmlauf ändern können).

⟶ Siehe nächste Folie für ein Beispiel.

Beispiel:

```
public class X { ... }

public class Y extends X { ... }

public class Z extends X { ... }
...

boolean b;
...  //  Bestimme die momentane Mondphase
     //  und setze b=true, falls zunehmend,
     //  sonst b=false.

X x;
if ( b )
   x = new Y (...);
else
   x = new Z (...);
```

⟶ Der aktuale Typ von „x“ kann von Programmlauf zu Programmlauf variieren und daher prinzipiell nicht vom Compiler erkannt werden.

Down–Cast:

```
public class BasisKlasse { ... }

public class AbgeleiteteKlasse1 extends BasisKlasse
 {
   public void f () { ... }
 }

public class AbgeleiteteKlasse2 extends BasisKlasse { ... }

...

BasisKlasse x = new AbgeleiteteKlasse1 ();

AbgeleiteteKlasse1 y = (AbgeleiteteKlasse1)x;  // Ok!
y.f();                                         // Ok! weil Formaler Typ
                                               // AbgeleiteteKlasse1

AbgeleiteteKlasse2 z = (AbgeleiteteKlasse2)x;  // !???
```


Erläuterungen:

- Der Quelltext lässt sich problemlos kompilieren.
- Aber beim Laufenlassen stürzt das Programm in der Zeile, in der „z“ eingerichtet wird, ab mit der Fehlermeldung:

  „`java.lang.ClassCastException`“

  ⟶ Wenn diese Zuweisung ohne Fehler durchgegangen wäre, dann wäre die Regel von Folie 511 auch nicht eingehalten worden.
- Die Zeile, in der „y“ eingerichtet wird, ist absolut korrekt:
  - ◇ Die Variable „x“ verweist hinterher auf dasselbe Objekt wie „x“.
  - ◇ Der aktuale und der formale Typ von „y“ sind dann gleich.

  ⟶ Die Regel von Folie 511 wird eingehalten.

Fortsetzung Erläuterungen:

- *Erinnerung* an Folie 295:
  - ◇ Unsichere Konversionen zwischen eingebauten Typen sind durchaus möglich.
  - ◇ Man muss den Zieltyp dann in runden Klammern vor den zu konvertierenden Ausdruck schreiben.
- Wie die vorletzte Folie zeigt, kann man auch von einer Basisklasse auf eine abgeleitete Klasse „herunter“ konvertieren (engl. *down–cast*).
- Dies ist durchaus eine unsichere Konversion:
  - ◇ Wenn der aktuale Typ des Objekts gleich dem Zieltyp oder vom Zieltyp abgeleitet ist, geht alles gut.
  - ◇ Wenn nicht: Programmabsturz.
- Da diese Art von Konversion generell unsicher ist, muss man eben wieder den Zieltyp in runden Klammern vor die zu konvertierende Variable „x“ schreiben.

Allgemeine Regel:

Ein Down–Cast

```
AbgeleiteteKlasse x = (AbgeleiteteKlasse)y;
```

ist genau dann ok, wenn der aktuale Typ von „y“

- entweder „`AbgeleiteteKlasse`“ selbst oder
- direkt oder indirekt von „`AbgeleiteteKlasse`“ abgeleitet ist.

⟶ Mit anderen Worten: wenn die Regel von Folie 511 eingehalten wird.

Schlüsselwort „instanceof“:

- Bevor man einen unsicheren Down–Cast macht — und vielleicht nicht weiß, ob das Programm dadurch abstürzt — kann man in Java abprüfen, ob der Down–Cast ok ist.
- Dafür gibt es in Java das Schlüsselwort „`instanceof`“.
- *Gebrauch*: Der logische Ausdruck

  „`x instanceof Y`“

  liefert „`true`“ genau dann, wenn der aktuale Typ von „`x`“ entweder gleich „`Y`“ ist oder direkt oder indirekt von „`Y`“ abgeleitet ist.
- Mit anderen Worten: genau dann, wenn der Down–Cast ok ist.

```
Y y;
if ( x instanceof Y )
   y = (Y)x;
else
   System.out.println ( "Falscher Typ!" );
```


Was ist nun Polymorphie?

- What's in an (ancient Greek) name:
  - ◇ „Poly“ $\approx$ viel
  - ◇ „Morphe“ $\approx$ Gestalt
- *Idee*:
  - ◇ Eine Variable der Basisklasse (z.B. „`Applet`“) fungiert als eine einheitliche Fassade.
  - ◇ Dahinter können sich Objekte von unterschiedlichen (abgeleiteten) Klassentypen verbergen (zum Beispiel „`MeinApplet`“).
  - ◇ Dieselbe Methode kann nun in den verschiedenen abgeleiteten Klassen unterschiedlich implementiert sein (z.B. „`MeinApplet.paint`“).

    $\longrightarrow$ Ein Methodenaufruf vermittels dieser Fassade kann je nach Typ des dahinterstehenden Objekts völlig unterschiedliche Effekte erzeugen.

Aspekt 4: Interne Realisierung

- Jedes Objekt einer Klasse enthält eine weitere, „unsichtbare“ Datenkomponente.
- In dieser Datenkomponente ist kodiert, von welchem Typ das Objekt ist.
- Zu jeder Klasse wird separat irgendwo im Speicher einmal eine Tabelle angelegt, in der zu jeder *Objekt*methode (siehe Folie 406 ff.) der Klasse die Startadresse gespeichert ist.
- Falls eine Objektmethode in der Klasse nicht selbst implementiert ist, sondern von der Basisklasse vererbt wird, dann wird die Startadresse der ererbten Methode dort gespeichert.

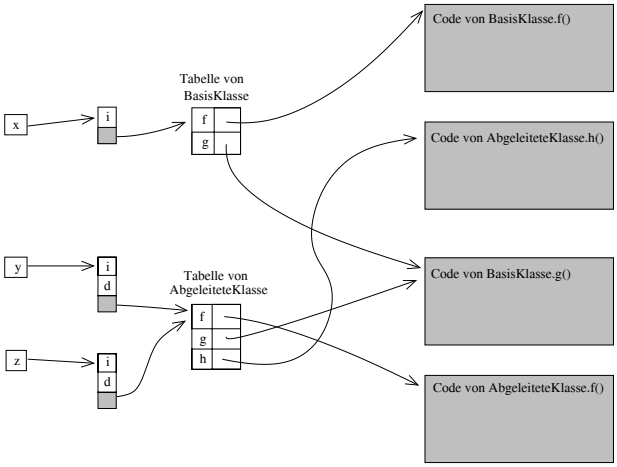
Beispiel zur Veranschaulichung:

```
public class BasisKlasse
  {
    public int i;

    public void f ()
      {
        System.out.println ( "1" );
      }

    public void g ()
      {
        System.out.println ( "2" );
      }
  }

public class AbgeleiteteKlasse extends BasisKlasse
  {
   public double d;

    public void f ()
      {
        System.out.println ( "3" );
      }

    public void h ()
      {
        System.out.println ( "4" );
      }
  }
```

Fortsetzung Beispiel:

```
BasisKlasse       x = new BasisKlasse();
BasisKlasse       y = new AbgeleiteteKlasse();
AbgeleiteteKlasse z = new AbgeleiteteKlasse();
```

Veranschaulichung:



Aufruf einer Methode:

Jeder Aufruf einer Objektmethode im Java–Quelltext wird vom Compiler in Code übersetzt, der

- zuerst die zusätzliche Datenkomponente mit der Typinformation liest,
- damit die Speicheradresse der zu diesem Klassentyp zugehörigen Tabelle von Methodenadressen heraussucht,
- in dieser Tabelle dann den Eintrag der aufgerufenen Methode nachschlägt und
- einen Methodenaufruf mit der dort gefundenen Startadresse ausführt.

⟶ Erst im letzten dieser vier Schritte werden die ganzen Aktionen von Abschnitt 3.1, Aspekt 3 (Folie 258 ff.) ausgeführt.

Frage:

- Auf den letzten beiden Folien war nur von *Objekt*methoden die Rede.
- Warum sind *Klassen*methoden ausgeklammert worden?

Antwort:

- *Technische* Antwort: Da eine Klassenmethode auch ohne Objekt aufgerufen werden kann, fehlt ihr diese Datenkomponente.
- *Inhaltliche* Antwort: Eine Klassenmethode ist ja „nur“ so etwas wie eine Funktion oder Prozedur in anderen Programmiersprachen (vgl. Folie 400).

  ⟶ Hat überhaupt nichts mit Vererbung zu tun.

Aspekt 5: Vererbung und Zugriffsrechte

- *Erinnerung* an Folie 467 ff.: Eigentlich können die Methoden einer Klasse alle Datenkomponenten und Methoden dieser Klasse benutzen.
- *Nun eine Ausnahme*: Die Methoden einer abgeleiteten Klasse dürfen „`private`“–deklarierte Datenkomponenten und Methoden, die sie von einer Basisklasse ererbt haben, nicht verwenden.
- Mit „`private`“ wird also genauer gesagt der Zugriff erlaubt
  - ◇ nur für die eine Klasse, in der die Datenkomponente bzw. Methode eingeführt wurde,
  - ◇ und nicht auch für alle davon abgeleiteten Klassen, in denen dieselbe Datenkomponente bzw. Methode (ererbt) ebenfalls vorkommt.

Regel zu „protected“:

Wird eine Datenkomponente oder Methode einer Klasse „X“

- nicht „public“ oder „private“,
- sondern „protected“

deklariert, dann kann sie verwendet werden

- nicht nur von den Methoden von „X“ selbst,
- sondern auch von den Methoden aller direkt und indirekt von „X“ abgeleiteten Klassen,
- aber nicht von den Methoden irgendeiner anderen Klasse.

Aspekt 6: Abstrakte Methoden und Klassen

- Manchmal macht es überhaupt keinen Sinn, eine Methode für eine Basisklasse tatsächlich zu implementieren.
- Für solche Fälle gibt es in Java die Möglichkeit,
  - ◇ eine Methode in einer Klasse einzuführen
  - ◇ ohne sie zu implementieren.
- *Syntaktische Unterschiede*:
  - ◇ Direkt vor dem Rückgabetyp das Schlüsselwort „abstract“.
  - ◇ Der Methodenrumpf „{...}“wird einfach durch ein Semikolon hinter der Parameterliste der Methode ersetzt.
- Eine solche Methode nennt man *abstrakt*.
- Wenn eine Klasse mindestens eine abstrakte Methode hat, nennt man die Klasse ebenfalls *abstrakt*.

  ⟶ „abstract“ muss auch vor „class“ geschrieben werden.

Beispiel:

```
abstract public class X
 {
   public void f ()
    {
      System.out.println ( "Hallo" );
    }

   abstract public void g ();
 }

public class Y extends X
 {
   public void g ()
    {
      System.out.println ( "Holla" );
    }
 }

abstract class Z extends X   // Immer noch abstrakt!
 {
   public void h ()
    {
      System.out.println ( "Holladrio" );
    }
 }
```


Fortsetzung abstrakte Methoden und Klassen:

- Von einer abstrakten Klasse „X“ kann man
  - ◇ durchaus Variable definieren,
  - ◇ aber keine Objekte anlegen!
- Es wäre auch fatal, wenn das ginge:
  - ◇ Dann könnte „X“ der aktuale Typ einer Variable vom Klassentyp werden.
  - ◇ Gemäß Folie 512 werden dann die Methodenimplementationen von „X“ aufgerufen.
  - ◇ Die existieren aber gar nicht alle!
- *Glücklicherweise*:
  - ◇ Wenn es keinen Sinn macht, alle Methoden einer Klasse zu implementieren,
  - ◇ dann macht es typischerweise auch keinen Sinn, ein Objekt dieser Klasse zu erzeugen.

Beispiel:

```
abstract public class X
 {
   X () { ... }

   abstract public void f ();

   public void g ()  { ... }
 }
...

X x = new X();
x.f();
```

⟶ Fehlermeldung des Compilers in der vorletzten Zeile: „X“ ist abstrakt!

Einigermaßen realistisches Fallbeispiel:

- In einem Zeichenprogramm sollen verschiedene geometrische Figuren als Objekte verwaltet werden: Kreise, Ellipsen, Dreiecke, Vierecke, ...
- Jede dieser Objektarten muss natürlich anders gespeichert werden:
  - ◇ Kreis: Mittelpunkt und Radius
  - ◇ Ellipse: Mittelpunkt, großer Radius, kleiner Radius und Winkel
  - ◇ Dreieck: die drei Punkte
  - ◇ Viereck: die vier Punkte
  - ◇ ...

  ⟶ Sollte jeweils eine eigene Klasse werden.
- Andererseits sollen zum Beispiel auch Arrays von geometrischen Objekten verwaltet werden.

  ⟶ Eine Basisklasse für alle diese Klassen.
- Zum Beispiel würde es sich anbieten, schematische Zeichnungen als Arrays von geometrischen Objekten abzuspeichern.

Fortsetzung Beispiel:

- Jede dieser Klassen soll eine Methode „zeichne“ bekommen, die das Objekt mit Hilfe eines „Graphics“–Objekts zeichnet.

  ⟶ Auch die Basisklasse, damit man „zeichne“ zum Beispiel in einer Schleife auf alle Elemente eines Arrays (=Bild) mit „gemischten“ geometrischen Objekten anwenden kann.
- Natürlich muss jede dieser Methodenimplementationen an die internen Daten der jeweiligen Klasse angepasst werden.
- Dann macht es überhaupt keinen Sinn, die Methode „zeichne“ für die Basisklasse zu implementieren.

Beispielhafter Code dazu:

```
public abstract class GeometrischeFigur         // !!!
 {
   public abstract void zeichne ( Graphics g ); // !!!
 }

public class Kreis extends GeometrischeFigur
 {
   private int xVonMittelpunkt;
   private int yVonMittelpunkt;
   private int radius;

   public Kreis ( int xVonMittelpunkt,
                  int yVonMittelpunkt,
                  int radius )
    {
      this.xVonMittelpunkt = xVonMittelpunkt;
      this.yVonMittelpunkt = yVonMittelpunkt;
      this.radius          = radius;
    }

   public void zeichne ( Graphics g )
    {
      g.drawOval ( xVonMittelpunkt-radius,
                   yVonMittelpunkt-radius,
                   2*radius, 2*radius );
    }
 }

// Klassen "Ellipse", "Dreieck", usw. analog
```

Fortsetzung beispielhafter Code:

```
GeometrischeFigur[] A = new GeometrischeFigur[3];

A[0] = new Kreis    ( 100, 200, 10 );
A[1] = new Dreieck ( ... );  // 3 Punkte anzugeben
A[2] = new Viereck ( ... );  // 4 Punkte anzugeben

for ( int i=0; i<A.length; i++ )
  A[i].zeichne(g);
```

Verboten ist zum Beispiel:

```
A[0] = new GeometrischeFigur();
```

⟶ „`GeometrischeFigur`“ ist halt abstrakt!

Aspekt 7: Mehrfachvererbung und Interfaces

- Bis jetzt haben wir
  - ◇ immer nur eine einzige Klasse hinter „`extends`“ geschrieben,
  - ◇ das heißt, immer nur von einer einzelnen Basisklasse direkt abgeleitet.
- In vielen objektorientierten Programmiersprachen kann man eine Klasse von mehreren Basisklassen zugleich direkt ableiten.
- Aus verschiedenen (guten!) Gründen ist in Java direkte Vererbung von mehreren Klassen zugleich nicht möglich.

  ⟶ Es darf also tatsächlich auch nur der Name einer einzigen Klasse hinter „`extends`“ stehen.
- Oft ist es aber wünschenswert, dass ein und dasselbe Objekt hinter verschiedenen „Fassaden“ verwendet werden kann.

Beispiel (leicht gekünstelt):

- Universitäre Kurse kann man grob zerlegen in Vorlesungen (z.B. „Mathematik I“, „Allgemeine Informatik II“), Seminare, Praktika u.a.
- Sinnvolle Attribute zu Vorlesungen und Seminaren wären etwa Dozent(en), wöchentliche Termine und Räume.
- Sinnvolle Attribute zu Praktika wären vielleicht Rechnerbetreuungszeiten, Betreuer und verwendete Software.
- Aus diesen Attributen kann man ablesen, dass unsere Veranstaltung in einer solchen Klassifikation ein „Zwitter“ wäre.

  ⟶ Möchte man vielleicht in einem Array von Vorlesungen *und* in einem Array von Praktika dabei haben.

Was wollen wir haben:

- Es bietet sich an, Vorlesungen, Praktika und „Kombinationen“ aus beidem jeweils zu einer Klasse zu machen.

  ⟶ Für sich genommen kein Problem.
- Eigentlich würden wir die Klasse für solche „Kombinationen“ gerne ableiten
  - ◇ *sowohl* von der Klasse für Vorlesungen,
  - ◇ *als auch* von der Klasse für Praktika.

  ⟶ Geht nicht in Java.
- In anderen Programmiersprachen (z.B. C++) gibt es solche Mehrfachvererbung.

  ⟶ Allerdings mit großen Problemen.
- Wegen dieser Probleme haben sich die Entwickler von Java gegen Mehrfachvererbung entschieden.
- Da diese Art von Mehrfachvererbung so häufig wünschenswert ist, haben sie sich aber einen Ersatz einfallen lassen: *Interfaces*.

Interfaces:

- Die Syntax von Klassen- und Interface–Deklarationen sind im Großen und Ganzen identisch.
- *Wesentliche Unterschiede*:
  - ◇ Das Schlüsselwort „`interface`“ ersetzt das Schlüsselwort „`class`“.
  - ◇ Methoden werden nicht implementiert, sondern nach dem Methodenkopf steht nur noch ein Semikolon.

    ⟶ Wie bei abstrakten Methoden.
  - ◇ Bei der Ableitung einer Klasse von einem Interface wird das Schlüsselwort „`extends`“ durch das Schlüsselwort „`implements`“ ersetzt.
  - ◇ Alle Methoden müssen „`public`“ sein
  - ◇ Klassenmethoden (Folie 395 ff.) sind nicht erlaubt.

Fortsetzung Interfaces:

- Man sagt dann auch nicht, die Klasse ist vom Interface abgeleitet.
- Sondern man sagt, die Klasse *implementiert* das Interface.
- Eine Klasse kann mehrere Interfaces zugleich implementieren.
- Aber sie darf weiterhin nur von maximal einer Klasse abgeleitet sein.
- Interfaces haben keine Datenkomponenten.

Beispiel:

```
public interface Vorlesung
 {
   public Name []  dozenten ();  // Die Klassen "Name",
   public Zeit []  termine  ();  // "Zeit", "Raum" und
   public Raum []  raeume   ();  // "Software" nehmen
 }                               // wir als gegeben an.

public interface Praktikum
 {
   public Name     [] betreuer           ();
   public Zeit     [] betreuungsZeiten   ();
   public Software [] verwendeteSoftware ();
 }
```

Normale implementierende Klassen:

```
public class NormaleVorlesung implements Vorlesung
 {
   public Name []  dozenten ()  { ... }
   public Zeit []  termine  ()  { ... }
   public Raum []  raeume   ()  { ... }
 }

public class NormalesPraktikum implements Praktikum
 {
   public Name     []  betreuer           ()  { ... }
   public Zeit     []  betreuungsZeiten   ()  { ... }
   public Software []  verwendeteSoftware ()  { ... }
 }
```


Mehrfach implementierende Klasse:

```
public class Kombination
 implements Vorlesung, Praktikum
 {
   public Name     []  dozenten           ()  { ... }
   public Zeit     []  termine            ()  { ... }
   public Raum     []  raeume             ()  { ... }
   public Name     []  betreuer           ()  { ... }
   public Zeit     []  betreuungsZeiten   ()  { ... }
   public Software []  verwendeteSoftware ()  { ... }
 }
```

Beispielhafter Umgang:

```
Vorlesung   x1 = new NormaleVorlesung  ();
Praktikum   x2 = new NormalesPraktikum ();
Kombination x3 = new Kombination       ();

Vorlesung   x4 = x3;
Praktikum   x5 = x3;

System.out.println ( x1.dozenten()[0] );
System.out.println ( x2.betreuer()[0] );
System.out.println ( x3.dozenten()[0] );
System.out.println ( x3.betreuer()[0] );
System.out.println ( x4.dozenten()[0] );
System.out.println ( x5.betreuer()[0] );
```

Hierarchie von Interfaces:

- Ein Interface kann auch ein anderes erweitern.
- Dann steht wieder „`extends`“ statt „`implements`“.

⟶ Beispiel auf der nächsten Folie.

Beispiel:

```
public interface MeinInterface1
  {
    public void f ();
    public void g ();
  }

public interface MeinInterface2
  extends MeinInterface1
  {
    public void h ();
  }

class MeineKlasse1 implements MeinInterface1
  {
    public void f ()  { ... }
    public void g ()  { ... }
  }

class MeineKlasse2 implements MeinInterface2
  {
    public void f ()  { ... }
    public void g ()  { ... }
    public void h ()  { ... }
  }
```


Aspekt 8: Klasse „Object“

- *Regel:* Wenn eine Klasse
  - ◇ nicht mit „`extends`“ explizit von einer anderen Klasse abgeleitet ist,
  - ◇ ist sie *im*plizit von der Klasse „`java.lang.Object`“ abgeleitet.
- *Einzige Ausnahme*: natürlich „`java.lang.Object`“ selbst.
- Die Klasse „`java.lang.Object`“ enthält eine ganze Reihe von Methoden, zum Beispiel:
  - ◇ „`boolean equals (Object obj)`“ ⟶ siehe Folie 321.
  - ◇ „`String toString ()`“ ⟶ vgl. Folie 329.

Was fängt man damit an:

- Jede Methode von „`java.lang.Object`“ ist gemäß einer allgemeinen Bedeutung implementiert, zum Beispiel
  - ◇ „`java.lang.Object.equals`“: identisch mit „==“.
  - ◇ „`String toString ()`“: der Name der Klasse und eine technische Zusatzinformation wird in einem einzelnen String zusammengefasst.
- Wenn man diese Methoden
  - ◇ für eine selbstgebastelte Klasse haben will,
  - ◇ und auch genau mit dieser Semantik,

  dann muss man gar nichts tun: alles schon fertig.
- Will man hingegen eine dieser Methoden
  - ◇ für eine selbstgebastelte Klasse haben,
  - ◇ aber nicht mit genau dieser Semantik,

  dann muss man sie eben selbst noch einmal mit der erwünschten Semantik implementieren.

Typische Semantik solcher Neuimplementationen:

```
public class MeineKlasse
 {
   private int    i;
   private double d;
   private char   c;

   public boolean equals ( Object obj )
    {
      MeineKlasse x = (MeineKlasse)obj;
      return i == x.i && d == x.d && c == x.c;
    }

   public String toString ()
    {
      return new String ( i + " " + d + " " + c );
    }
 }
```


Sinn von Klasse „Object“ am Beispiel:

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

Vector v = new Vector();

for ( int i=0; i<4; i++ )
 {
   Integer x = new Integer ( i * i );
   v.add (x);
 }

for ( int i=0; i<v.size(); i++ )
 {
   Integer x = (Integer)(v.elementAt(i));
   System.out.print ( x.intValue() );
 } // Gesamtausgabe der Schleife: "0149"

v.remove(2);

for ( int i=0; i<v.size(); i++ )
 {
   Integer x = (Integer)(v.elementAt(i));
   System.out.print ( x.intValue() );
 } // Gesamtausgabe der Schleife: "019"
```


Erläuterungen:

- Dies ist jetzt die Stelle, auf die Folie 340 vertröstet hat.
- Die Klasse „`java.util.Vector`“ realisiert im Prinzip ein Array, in dem man auch einfügen und löschen kann.
- Ist aber eine ganz normale Klasse.
- *Dilemma*:
  - ◇ In Methoden wie „`add`“ und „`elementAt`“ muss man den Elementen des Vektors ja irgendeinen Typ geben.
  - ◇ Aber eigentlich wollen wir ja wie bei Arrays Vektoren für alle möglichen Typen haben.
- *Lösung* in Java: „`java.lang.Object`“.
- Wie die letzte Folie gezeigt hat, ist „`Vector`“ ausserdem ein Beispiel dafür, dass die Wrapper–Klassen von Folie 328 ff. durchaus wichtig sind.

  ⟶ Werte von eingebauten Typen können nur in Objekten der jeweils zugehörigen Wrapper–Typen in einem „`Vector`“ gespeichert werden.

Abschnitt 4.5: Exceptions

```
import java.applet.*;
import java.awt.*;

public class TrialApplet extends Applet
 {
   public void paint ( Graphics graphics )
    {
      Thread.sleep(1000);
    }
 }
```

Fehlermeldung von „`javac`“:

```
TrialApplet.java:8:
  Exception java.lang.InterruptedException
  must be caught, or it must be declared
  in the throws clause of this method.
```

Hintergrund:

- In der Abarbeitung eines Methodenaufrufs kann die Methode immer potentiell auf Probleme stossen, mit denen sie selbst nicht umzugehen weiß.
- *Exceptions* geben einer Methode die Möglichkeit,
  - ◇ den Methodenaufruf in einem solchen Fall umgehend, aber dennoch kontrolliert zu beenden
  - ◇ und das Problem damit an die aufrufende Methode zu delegieren.
- *Idee dahinter*: Vielleicht weiß ja die aufrufende Methode besser mit dem Problem umzugehen.

  ⟶ Soll die sich eben damit herumschlagen.

Beachte:

„Die Methode weiß“ etc. sind Ausdruck eines vermenschlichenden Stils, der hier (im Gegensatz etwa zu: „Der Computer hat Schuld“) allein zur Vereinfachung der Formulierungen dienen soll.

Fallbeispiel für ein solches Problem:

- Betrachte ein Programm zur Auswertung beliebig komplexer mathematischer Ausdrücke mit den vier Grundrechenarten und verschiedenen Arten von Klammern.
- *Beispiel*: `3 + 2 * (6-4) / (7-5)`
- Eine interaktive Methode „`m1`“ in diesem Programm sei dafür zuständig, einen solchen Ausdruck von der Tastatur einzulesen und den Wert des Ausdrucks auf dem Bildschirm zu präsentieren.
- Die Methode „`m1`“ ruft dann zweckmäßigerweise eine andere, separate Methode „`m2`“ auf, die
  - ◇ diesen Ausdruck (z.B. als „`String`“–Objekt) als Parameter erhält,
  - ◇ den Wert des darin gespeicherten Ausdrucks berechnet und
  - ◇ das Ergebnis an „`m1`“ zwecks Präsentation auf dem Bildschirm zurückreicht.

Zwischenfrage:

Warum nicht „m1“ und „m2“ einfach zu einer einzigen Methode zusammenfassen?

Gute Gründe sprechen dagegen:

- Die Funktionalität von „`m2`“ ist auch in anderen Programmpaketen sinnvoll (z.B. Steuerabrechnungen).

  ⟶ Als separate Methode lässt sich diese Funktionalität leicht in andere Pakete übernehmen.
- „`m2`“ kann in „`m1`“ bei Bedarf problemlos durch eine andere Methode zur Berechnung des Wertes von mathematischen Ausdrücken ausgetauscht werden.
- Insgesamt wird die Programmstruktur durch strikte Trennung von grundverschiedenen Aspekten wie Benutzerinteraktion und mathematischer Berechnung nur besser.

  ⟶ Wichtig für das menschliche Verständnis des Quelltextes.

Wichtige Randbemerkung:

- Selbstverständlich würde man in einer richtigen Problemstellung aus der Praxis solche Methoden nicht einfach „`m1`“ oder „`m2`“ nennen.
- Jedem Softwareentwickler muss eigentlich in Fleisch und Blut übergehen, „sprechende“ Namen zu verwenden.

  ⟶ Der Name ist dann eine Kurzerklärung, die bei jedem Aufruf der Methode automatisch dasteht.

  ⟶ Besonders wichtig wegen der allgemein üblichen Lustlosigkeit beim Schreiben von Kommentaren.
- In unserem Fallbeispiel könnte die Methode „`m2`“ beispielsweise sinnvoll

  „`berechneWertVonMathAusdruck`“

  heißen (vgl. Folie 202 ff.).

Zurück zum Thema Exceptions:

- Der eingegebene mathematische Ausdruck kann auch fehlerhaft sein.
- *Beispiele*:
  - ◇ Öffnende und schließende Klammern passen im eingegebenen Ausdruck vielleicht nicht nach den Regeln von Folie 212 zusammen.
  - ◇ An irgendeiner Stelle wird in einem kleinen Teilausdruck des Gesamtausdrucks durch Null geteilt.
- Bei der Berechnung des Wertes des Ausdrucks in „m2“ fallen solche Fehler automatisch auf.
- Es ist also zweckmäßig, wenn die Suche nach solchen Fehlern nicht von „m1“, sondern quasi nebenher von „m2“ erledigt wird.
- *Nur*:
  - ◇ Falls „m2“ wirklich irgendwo mitten in den Berechnungen auf einen Fehler stößt,
  - ◇ was soll „m2“ dann eigentlich machen?

Konzeptionelle Antwort:

- Die Methode „m2“ ist ja eigentlich nur ein „blinder Rechenknecht“ und im Grunde für die Behandlung von Benutzerfehlern inkompetent.
- Kompetent dafür ist eher die benutzerorientierte Methode „m1“.
- Zum Beispiel könnte „m1“ dem Benutzer
  - ◇ eine informative Fehlermeldung (genaue Fehlerstelle, Art des Fehlers) geben,
  - ◇ ein Fenster zur unmittelbaren Nachkorrektur des Ausdrucks aufmachen oder sogar
  - ◇ konkrete Vorschläge zur Korrektur machen.
- Welche dieser Optionen in einem konkreten Anwendungskontext sinnvoll wäre, kann nur „m1“, nicht aber „m2“ wissen.

Also Konzeption:

- Die Methode „m2“ berechnet den Ausdruck und achtet dabei nebenher auf Fehler.
- Im Fehlerfalle stellt sie ihre weitere Arbeit ein und reicht eine Fehlerdiagnostik (genaue Fehlerstelle, Art des Fehlers o.ä.) zurück an „m1“.
- Die Methode „m1“ behandelt nun den Fehler bspw. auf eine der auf der letzten Folie angedeuteten Arten.
- Erhält „m1“ durch Korrekturen des Benutzers einen neuen, korrigierten Ausdruck, könnte „m2“ zum Beispiel damit erneut durch „m1“ aufgerufen werden.

  $\longrightarrow$ Bis der Benutzer es fertig bringt, den gewünschten Ausdruck korrekt hinzuschreiben (oder entnervt aufgibt).

Technische Umsetzung in Java:

- Bei der Programmierung von „m1“ und „m2“ ist in den Quelltexten beider Methoden „vereinbart“ worden, dass
  - ◇ „m2“ in gewissen Fällen eine *Exception* werfen darf,
  - ◇ die dann von „m1“ gefangen werden muss.
- Wenn „m2“ auf einen Fehler im mathematischen Ausdruck stößt, dann
  - ◇ wirft „m2“ eine Exception,
  - ◇ was zugleich bedeutet, dass die Abarbeitung von „m2“ sofort (und ohne Rückgabewert) beendet wird.
- Die aufrufende Methode „m1“ muss darauf mit einer (wie auch immer gearteten) Fehlerbehandlung reagieren.

Beispiel für das Werfen einer Exception in „m2“:

```
public double m2 ( String ausdruck )
 throws Exception
  {
    double wertDesAusdrucks;

    ...

    if ( divisor == 0 )
     {
       Exception exception
         = new Exception
                  ( "Fehler im Ausdruck!" );
       throw exception;
     }

    ...

    return wertDesAusdrucks;
  }
```


Erläuterungen zu „throws“:

- Dies ist der Punkt, auf den auf den Folien 379 ff. mehrmals vertröstet worden ist.
- Mit „`throws Exception`“ wird in „`m2`“ vereinbart, dass diese Methode potentiell eine Exception vom Typ namens „`Exception`“ wirft.
- Andere Typen von Exceptions betrachten wir erst später (Folie 571 ff.).

Erläuterungen zu „throw“ (nun ohne „s“!):

- Solange kein Fehler auftritt, wird in der Variablen „`wertDesAusdrucks`“ nach und nach der mathematische Wert von „`ausdruck`“ zusammengebastelt.
- Falls in diesem ganzen Arbeitsgang kein einziger Fehler aufgetreten ist, wird das Endergebnis dann mit „`return`“ ausgegeben.
- Falls doch ein Fehler entdeckt wird, wird die Abarbeitung „`m2`“ durch das „`throw`“–Konstrukt sofort beendet.
- Im Beispiel auf der vorletzten Folie wurde nur eine einzige Stelle exemplarisch gezeigt, an der ein möglicher Fehler abgetestet wird: ob der zweite Operand (der *Divisor*) einer Division gleich Null ist.
- Falls eine Division durch Null auftritt, wird durch „`throw`“ ein neu kreiertes Objekt des Typs „`Exception`“ als Exception geworfen.

Vergleich „throw“ und „return“:

- In gewisser Weise ist „`throw`“ daher so etwas wie eine Variation von „`return`“:
  - ◇ Der Methodenaufruf wird beendet,
  - ◇ und es wird ein Objekt an die aufrufende Methode zurückgereicht.
- Allerdings hat „`throw`“ für die aufrufende Methode „`m1`“ völlig andere syntaktische und semantische Konsequenzen als „`return`“.
- Dazu mehr auf den nächsten Folien.
- Klitzekleiner weiterer Unterschied:
  - ◇ Nicht jedes „`return`“ muss ein Objekt zurückliefern.

    ⟶ „`return`“ liefert genau dann ein Objekt zurück, wenn die Methode nicht „`void`“ ist.
  - ◇ Durch „`throw`“ muss hingegen grundsätzlich immer eine Exception geworfen werden.

Nun das Ganze aus Sicht von „m1“:

```
public void m1 ( ... )
 {
   String ausdruck;
   double wertDesAusdrucks;

   ...

   try
    {
      wertDesAusdrucks = m2 ( ausdruck );
      System.out.println ( wertDesAusdrucks );
    }
   catch ( Exception exc )
    {
      System.out.println ( exc.getMessage() );
    }

   ...
 }
```


Erläuterungen:

- Eine Exception, die von einer aufgerufenen Methode wie „m2“ geworfen wird, muss von einer sie aufrufenden Methode wie „m1“ umständlich mit einem solchen „try–catch“–Konstrukt behandelt („gefangen“) werden.
- Der Aufruf der potentiell werfenden Methode muss im „try“–Block stehen.
- Wenn durch „m1“ *keine* Exception geworfen wird, dann
  - ◇ wird der „try“–Block normal zu Ende abgearbeitet (also „wertDesAusdrucks“) ausgegeben,
  - ◇ der „catch“–Block wird übersprungen,
  - ◇ d.h. nach Beendigung des „try“–Blocks wird mit den nächsten Anweisung, die regulär nach dem „catch“–Block folgt, fortgefahren.

Im Fall des Wurfs einer Exception:

- Nicht nur der Aufruf von „m2“, auch der ganze „try“–Block wird sofort beendet.

  ⟶ Die Ausgabe des Werts von „wertDesAusdrucks“ findet nicht statt (würde eh' einen unsinnigen Wert ausgeben).
- Statt dessen wird die Abarbeitung des „catch“-Blocks begonnen.

  ⟶ Der „catch“–Block ist für die Anweisungen zur Fehlerbehandlung da.
- Der „catch“–Block besitzt einen Kopf, in dem der Programmierer von „m1“ für das geworfene Exception–Objekt (genauer: dem Verweis darauf; siehe Folie 305) einen Identifier als Namen vergibt.

  ⟶ Im Beispiel hat der Programmierer von „m1“ also den Namen „exc“ gewählt.
- Nach Abarbeitung des „catch“–Blocks geht es normal mit der nachfolgenden Anweisung weiter.

Semantik von Folie 565 gemäß Folie 71 (noch mehr abstrahiert als sonst):

1. Richte die „String“–Variable „ausdruck“ und die „double“–Variable „wertDesAusdrucks“ ein.
2. ...
3. Rufe „m2“ mit Argument „ausdruck“ auf.
4. Falls eine Exception geworfen wird, springe zur Instruktion Nr. 8.
5. Kopiere den Rückgabewert von „m2“ in die Variable „wertDesAusdrucks“.
6. Gib den Inhalt von „wertDesAusdrucks“ aus.
7. Springe zur Instruktion Nr. 11.
8. Kopiere den als Exception zurückgelieferten Verweis in die Variable „exc“.
9. Rufe Methode „getMessage“ für „exc“ auf.
10. Gib den Rückgabewert dieses Aufrufs aus.
11. ...

Frage:

- Das „`try`–`catch`“–Konstrukt wurde eigentlich ja nur angewandt, um die Exception von „`m2`“ abzufangen.
- Wieso steht dann die Schreibausgabe von „`wertDesAusdrucks`“ ebenfalls im „`try`“–Block?

Antwort:

- Diese Schreibausgabe macht nur Sinn, wenn keine Exception geworfen wurde.
- Sie sollte daher an einer Stelle stehen, die nach Wurf einer Exception nicht erreicht wird.

Bemerkung zur Syntax:

- Nach „`try`“ bzw. „`catch`“ müssen geschweifte Klammern kommen,
- auch wenn nur eine einzelne Anweisung folgt (im Gegensatz zu Folie 256).

Zur Klasse Exception:

- Genauer ist damit die Klasse „`java.lang.Exception`“ gemeint.
- Ein Objekt der Klasse „`Exception`“ besitzt eine „`String`“–Variable als Datenkomponente.
- Das „`Exception`“–Objekt dient damit praktisch als Bote, der diesen String als Botschaft von der aufgerufenen zur aufrufenden Methode trägt.
- In der Methode „`m2`“ (Folie 561) wird die Botschaft dem Boten schon bei seiner Erzeugung mitgegeben:

```
new Exception ( "Fehler im Ausdruck!" )
```

- Die Klasse „`Exception`“ hat eine Methode „`getMessage`“, die diese Botschaft zurückliefert.
- Mit dieser Methode ist dann in „`m1`“ (Folie 565) auf die Botschaft von „`exc`“ zugegriffen werden.

Frage:

- In den Quelltexten zu den Übungsaufgaben sind uns einige Male diese Konstrukte begegnet.
- Die Exceptions waren dort aber gar nicht vom Typ „`Exception`“!?
- Auch im einleitenden Beispiel auf Folie 552 war der Typ nicht „`Exception`“, sondern „`InterruptedException`“!?

Antwort:

- Alle diese Klassen sind abgeleitet von der Basisklasse „`Exception`“.
- Ein Objekt einer solchen abgeleiteten Klasse kann also wie üblich anstelle eines Objekts der Basisklasse (z.B. nach „`throw`“) verwendet werden.
- Jede dieser Klassen
  - ◇ ist für spezifische Anwendungsfälle konzipiert und
  - ◇ enthält in der Regel weitere Möglichkeiten, Botschaften in spezifischerer Form als nur durch eine einfache Zeichenkette zu kodieren.

Anwendung im Fallbeispiel von Folie 554 ff.

Beispielhafte Skizze:

- Definiere eine eigene Klasse „`FehlerInAusdruckException`“ als Erweiterung von „`Exception`“.
- Als Datenkomponenten werden zum Beispiel eingerichtet:
  - ◇ Eine Zahl zur Lokalisierung des genauen Fehlerpunktes als Index in der Zeichenkette.
  - ◇ Eine weitere Zahl als Fehlerdiagnostik (z.B. 0=fehlende schließende Klammer, 1=schließende Klammer zuviel, 2=...).

  $\longrightarrow$ Mit einer solchen von „`m2`“ mittels Exception empfangenen Botschaft kann „`m1`“ recht informative Fehlermeldungen an den Benutzer ausgeben.
- Falls nicht nur ein, sondern alle Fehler im Ausdruck zugleich zurückgeliefert werden sollen, bieten sich bspw. zwei Arrays von solchen Zahlen als Botschaft an.

Beispielhafte Realisierung:

```
public class FehlerInAusdruckException
 extends Exception
 {
   private int fehlerStelle;
   private int fehlerArt;

   public FehlerInAusdruckException
                    ( int fehlerStelle,
                      int fehlerArt )
    {
      this.fehlerStelle = fehlerStelle;
      this.fehlerArt = fehlerArt;
    }

   public String getMessage ()
    {
      String fehlerBeschreibung;
      if ( fehlerArt == 0 )
         fehlerBeschreibung
            = "Schliessende Klammer fehlt";
      else if ( fehlerArt == 1 )
         fehlerBeschreibung
            = "Schliessende Klammer zuviel";
      else ...

      return fehlerBeschreibung + " an Index "
           + fehlerStelle + "!";
    }
 }
```


Exceptions weiterreichen statt fangen:

- Im Beispiel auf Folie 565 wurde ein „`try`"–„`catch`"–Konstrukt für die von „`m1`" potentiell geworfenen Exceptions eingerichtet.
- Alternativ könnte „`m1`" aber auch
  - ◇ diese Exceptions nicht selbst abfangen und bearbeiten,
  - ◇ sondern durch ein eigenes „`throws`" im Methodenkopf an die nächsthöhere Methode weiterreichen.
- Diese letztere Methode muss sich dann damit herumschlagen (oder ebenfalls weiterreichen).

⟶ Daher das Entweder–Oder in der Fehlermeldung auf Folie 552:

```
Exception java.lang.InterruptedException
must be caught, or it must be declared
in the throws clause of this method.
```


Methode „m1" also nun ohne try/catch:

```
public void m1 ( ... )
 throws Exception  // <- !!!
 {
   String ausdruck;
   double wertDesAusdrucks;
   ...

   wertDesAusdrucks = m2 ( ausdruck );
   System.out.println ( wertDesAusdrucks );
   ...
 }
```

Erläuterung:

Wenn nun „`m2`" eine Exception wirft, dann

- wird nicht nur die Bearbeitung von „`m2`" sofort abgebrochen,
- sondern auch die von „`m1`",
- und die Exception wird an die Methode, die „`m1`" aufgerufen hat, weitergereicht.

Erinnerung an Folie 386:

- Interpreter wie „`java`" oder „`appletviewer`" erwarten, dass die Signatur der Startmethode exakt so ist wie erwartet.
- Insbesondere wird eigentlich immer eine leere „`throws`"–Liste erwartet.

Konsequenz:

- Exceptions können nicht beliebig gemäß Folie 574 mit „`throws`" in der Aufrufhierarchie weiter hochgereicht werden.
- Spätestens die Startmethode, die vom Interpreter aufgerufen wird, muss die Exception fangen.
- Reicht diese Methode Exceptions statt dessen mittels „`throws`" weiter, entspricht die Signatur nicht mehr der Erwartung des Interpreters.
- Der Interpreter steigt sofort mit einer entsprechenden Fehlermeldung aus.

Methoden mit mehreren Exception–Typen

```
class MyException1 extends Exception { ... }

class MyException2 extends Exception { ... }

...

void F ( int a, int b )
  throws MyException1, MyException2
  {
    if ( a < b )
       throw new MyException1 (...);
    else
       throw new MyException2 (...);
  }
```

Erläuterungen:

- Eine Methode kann auch Exceptions von verschiedenen Typen werfen.
- Diese verschiedenen Typen werden in einer einzigen „`throws`“–Klausel (durch Kommas voneinander getrennt) aufgelistet.

Aufruf einer solchen Methode:

```
try { F(2,3); }
catch ( MyException1 exc )
   { System.out.println ( "2 <  3" ); }
catch ( MyException2 exc )
   { System.out.println ( "2 >= 3" ); }
```

Erläuterungen:

- Für jeden potentiell geworfenen Exception–Typ muss eine entsprechende „`catch`“–Klausel eingefügt werden.
- *Beachte*:
  - ◇ Es kommt dabei allein darauf an, welche Exception–Typen in der throws–Klausel deklariert sind.
  - ◇ Selbst wenn (wie im Beispiel oben) klar ist, dass ein bestimmter Exception–Typ nie geworfen werden kann, muss die zugehörige „`catch`“–Klausel vorhanden sein.

Alternativer Aufruf:

```
try { F(2,3); }
catch ( Exception exc )
  {
    System.out.println
                  ( "2 < 3 oder 2 >= 3" );
  }
```

Erläuterungen:

- Es muss nicht unbedingt genau die Exception–Klasse gefangen werden, die geworfen wurde.
- Es kann auch eine beliebige Exception–Klasse gefangen werden, von der die geworfene Exception–Klasse direkt oder indirekt abgeleitet wurde.
- Wie das Beispiel oben zeigt, kann sich dadurch die Zahl der notwendigen „`catch`“–Klauseln durchaus verringern.
- Im Extremfall, wenn Typ „`Exception`“ selbst gefangen wird, reicht (wie im Beispiel oben) natürlich auch eine einzige „`catch`“–Klausel.

Nicht unbedingt zu fangende Exceptions

1. Versuch:

```
int i = 1;
int j = 0;
System.out.println ( i/j );
```

Ergebnis:

- Beim Kompilieren mit „`javac`“ gibt es wegen der Division durch Null keinen Fehler.
- Beim Lauf des Programms gibt es Absturz mit Fehlermeldung:

  ```
  java.lang.ArithmeticException: / by zero
  ```

- Die Divisionsoperation wirft also wie eine Methode ebenfalls eine Exception.
- Der Compiler hat den Code aber augenscheinlich akzeptiert und übersetzt, obwohl die Exception weder gefangen noch weitergereicht wurde.

2. Versuch:

```
int[] A = new int [100];
int   i = 100;
System.out.println ( A[i] );
```

Ergebnis:

- Praktisch identisch mit dem Ergebnis von der vorherigen Folie.
- Nur die Meldung beim Absturz lautet nun anders:

```
java.lang.ArrayIndexOutOfBoundsException: 100
```

Bemerkung:

- Die Zusatzinformation „`100`“ in der Fehlermeldung oben
- und die Zusatzinformation „`/ by zero`“ auf der vorherigen Folie

sind Beispiele dafür, dass in spezifischen Exception–Klassen weitere, spezifischere Zusatzinformationen enthalten sein können (Folie 571).

Erläuterungen:

- Exceptions von gewissen Typen müssen nicht abgefangen oder weitergereicht werden.

  ⟶ Technisch gesprochen „`java.lang.RuntimeException`“ (abgeleitet von „`Exception`“) und alle direkt oder indirekt davon abgeleiteten Klassen.

- *Grund*:

  ◇ Arithmetische Operationen, Arrayzugriffe usw. kommen in vielen Java–Quelltexten häufig vor.

  ◇ Müssten diese Exceptions alle abgefangen werden, würde man die eigentliche Programmlogik hinter den vielen „`try`“s und „`catch`“s kaum noch finden.

  ◇ Alle diese Exceptions in der Aufrufhierarchie weiter hochreichen wäre auch keine Lösung (wohin am Ende?).

Resümee:

- Ausnahmsweise wurde hier deshalb einmal beim Entwurf von Java die Entscheidung contra Ablaufsicherheit getroffen.
- Nur durch diese Sicherheitslücke sind Programmabstürze in Java überhaupt möglich.

Abschnitt 4.6: Threads

- *Erinnerung* an Folie 125: Threads sind eine Art von Parallelprozessen, die in Java schon eingebaut sind.
- Ein Java–Programm kann also potentiell aus mehreren Prozessen (*Threads*) bestehen.
- Wenn man sich um Threads nicht kümmert (wie wir bisher), besteht jedes Programm aus genau einem Thread.
- Mit Hilfe der Klasse `java.lang.Thread` kann man Threads einrichten, beenden, kontrollieren und manipulieren.
- *Bisherige Verwendung*: Mit

  „`Thread.sleep(521)`“

  kann man den Thread, in dem die Methode aufgerufen wird, für 521 Millisekunden schlafenlegen.

Asynchronizität:

- Zwei (oder mehr) Threads arbeiten ihren Code nebeneinander her ab.
- Dabei gibt es keine zeitliche Abstimmung zwischen den einzelnen Aktionen der beiden Threads untereinander.
- Ob eine bestimmte Aktion des einen Threads vor oder nach einer bestimmten Aktion des anderen Threads ausgeführt wird, ist also völlig dem Zufall überlassen.
- Wenn zwei Threads an dieselbe Stelle zeichnen, malen oder schreiben, können die entsprechenden Operationen also wild, unvorhersehbar und potentiell sogar unreproduzierbar durcheinandergemischt werden.

Was ist ein Thread in Java?

- *Durchgängige Idee* in Java: Jedes Konzept ist realisiert als eine „Urstammklasse“ (siehe Folie 302).
- *Beispiele*:
  - ◇ Alle Applets sind Objekte von Klassen mit Urstamm „`java.applet.Applet`“ (Folie 303).
  - ◇ Alle von Java–Programmen aufgemachte Windows sind Objekte von Klassen mit Urstamm „`java.awt.Window`“ (Folie 301).
  - ◇ Alle Exceptions sind Objekte von Klassen mit Urstamm „`java.lang.Exception`“ (vgl. Folie 570).
- *Threads*:
  - ◇ In dem einen Thread, aus dem der Prozess eines Java–Programms zuerst besteht, können beliebig weitere Threads erzeugt werden, in diesen wiederum weitere Threads usw.
  - ◇ Jeder Thread ist innerhalb des erzeugenden Threads repräsentiert durch ein Objekt einer Klasse mit Urstamm „`java.lang.Thread`“.

Genauer:

- Das notwendige Thread–Objekt (mit Urstamm „`java.lang.Thread`“) ist in gewisser Weise nur der reine Thread als solcher noch ohne Inhalt.
- Der Code, der eigentlich durch einen solchen neuen Thread auszuführen ist, wird in der Java–Philosophie als eigenständiges Konzept angesehen.
- Folgerichtig gibt es einen eigenen Urstamm dafür in Java: „`java.lang.Runnable`“.
- Ein Thread–Objekt wird mit einem Runnable–Objekt initialisiert.
- Das Thread–Objekt repräsentiert einen Thread, der den Code dieses Runnable–Objekts ausführt.

Einfaches Beispiel:

```
public class MeinRunningGag
  implements Runnable
  {
    public void run ()
      {
        try {
          while ( true )
            {
              Thread.sleep ( 5000 );
                System.out.println ( "Still alive..." );
            }
        }
        catch( Exception exp ){
          System.out.println(exp.getMessage());
        }
      }
  }
```

Erläuterungen:

- So wie bei einem Abkömmling von Klasse „`Applet`“ die Methode

  ```
  public void paint ( Graphics g)
  ```

  implementiert werden muss,
- so wird für „`Runnable`“ die Methode

  ```
  public void run ()
  ```

  verlangt.

⟶ Enthält den Code, den der Thread ausführen soll.

Neuen Thread ans Laufen bringen:

```
Runnable meinCode   = new MeinRunningGag();
Thread   meinThread = new Thread (meinCode);
```

Mit diesem neuen Thread–Objekt arbeiten:

```
try
 {
  meinThread.start ();       // Los geht's
  meinThread.suspend ();     // Anhalten
  meinThread.resume ();      // Weiter geht's
  meinThread.stop ();        // Ende
 }
catch ( InterruptedException exc )
 {
   System.out.println ( exc.getMessage() );
 }
```


Systematisch zusammengefasst:

Einen Thread kann man vom aufrufenden Programm aus

- erst einmal starten mit „`start`“,
- bis auf weiteres schlafenlegen mit „`suspend`“,
- nach einem „`suspend`“ wieder aufwecken mit „`resume`“ und
- endgültig stoppen mit „`stop`“.

Erläuterung zur Exception auf Folie 590:

- Ein Thread kann (unter gewissen Regeln) von einem anderen Thread „unterbrochen“ werden.
- *Mechanismus*: Der „unterbrochene“ Thread beendet sich selbst durch sofortigen Wurf einer Exception.
- Diese Exception ist vom Typ „`java.lang.InterruptedException`“.
- Sie muss in der aufrufenden Methode gefangen (oder von ihr weitergereicht) werden.

Was kann man noch alles mit Threads machen:

- *Synchronization*:

  Durch die Anweisung

  ```
  thread.join();
  ```

  bleibt der aufrufende Thread stehen, bis der in Objekt „`thread`“ aufgerufene Thread beendet ist.

  ⟶ Genauer: Bis die Methode „`run`“ des dahinterstehenden „`Runnable`“–Objekts beendet ist.

- *Kommunikation*:

  Zwischen zwei Threads lässt sich ein ein- oder zweiseitiger Informationsfluss etablieren.

Abschlussfrage: wozu eigentlich Threads in Java?

Reales Fallbeispiel als Antwort:

- Moderne WWW–Browser können etliche Dinge gleichzeitig erledigen.
- *Beispiel*:
  - ◇ Nach Laden der eigentlichen WWW–Seite werden noch die viel größeren Bilder langsam nachgeladen.
  - ◇ Zugleich kann man in der WWW–Seite schon hin und her „scrollen“.
  - ◇ Gleichzeitig lässt der WWW–Browser schon einmal ein Applet laufen, das in dieser WWW–Seite mit einem HTML–Applet–Tag eingebunden ist.
- Alle diese einzelnen Aktiväten sind ihrer Natur nach weitgehend unabhängig voneinander und asynchron.
- Sie lassen sich daher zweckmäßigerweise als Threads implementieren.

Abschnitt 4.7: Events

- Diverse Klassen sind dafür konzipiert, in *Laufzeitumgebungen* zu laufen.
- *Beispiel*: `java.lang.Applet` und Subklassen laufen typischerweise in WWW–Browsern oder anderen „Applet–Viewern“.
- Laufzeitumgebungen arbeiten mit *Ereignissen* (*Events*).
- Klassen können sich da „einklinken“.
- Das heißt: Ein Objekt einer solchen Klasse erhält Nachricht, wenn ein bestimmter Event aufgetreten ist, und kann dann darauf reagieren.
- Allerdings werden solche Aufgaben typischerweise an eigene, registrierte „Listener“–Objekte delegiert.
- Im Code der Klassen dieser Objekte sind die Reaktionen auf Events implementiert

Einfaches, fundamentales Beispiel:

- Das „Abhorchen“ einer Maus wird in Java in zwei verschiedene Aspekte zerlegt:
  - ◇ Abhorchen von Maus*operationen*,
  - ◇ Abhorchen von Maus*bewegungen*.
- Für beide Aspekte gibt es folgerichtig je einen Urstamm:
  - ◇ „`java.awt.event.MouseListener`“ für Mausoperationen,
  - ◇ „`java.awt.event.MouseMotionListener`“ für Mausbewegungen.
- Damit ein Applet (oder ein anderes Java–Programm mit Window) auf Mausoperationen bzw. Mausbewegungen reagieren kann, muss
  - ◇ jeweils eine Erweiterung von „`MouseListener`“ bzw. „`MouseMotionListener`“ mit den gewünschten Reaktionen implementiert werden und
  - ◇ ein Objekt dieser neuen Klasse bei diesem Applet registriert werden.

Eigener MouseListener:

```
class MeinMouseListener implements MouseListener
 {
   public void mouseClicked ( MouseEvent event )
    {
      System.out.println ( "Mouse clicked at (" );
      System.out.println ( event.getX() );
      System.out.println ( "," );
      System.out.println ( event.getY() );
      System.out.println ( ")" );
    }
   public void mousePressed ( MouseEvent event )
    {
      System.out.println ( "Mouse pressed" );
    }
   public void mouseReleased ( MouseEvent event )
    {
      System.out.println ( "Mouse released" );
    }
   public void mouseEntered ( MouseEvent event )
    {
      System.out.println ( "Mouse entered" );
    }
   public void mouseExited ( MouseEvent event )
    {
      System.out.println ( "Mouse exited" );
    }
 }
```


Erläuterungen:

- Klasse „`MeinMouseListener`“ ist nur beispielhaft und reagiert auch nicht besonders smart auf Mausoperationen.
- Im wesentlichen protokolliert es alle Mausoperationen nur durch Schreibausgaben mit.
- *Ausnahme*: In dem Fall, dass eine Taste der Maus kurz angeklickt wird, werden außerdem auch noch die Koordinaten des Mauszeigers (Folie 135) ausgegeben.
- Die Klasse „`java.awt.event.MouseEvent`“ bietet Methoden „`getX`“ und „`getY`“, um diese Koordinaten abzufragen.
- Welche Methoden implementiert werden müssen, wie sie heißen, welche Parameter und welchen Rückgabetyp sie haben, alles das ist schon im Urstamm „`java.awt.event.MouseListener`“ festgelegt.
- Offensichtlich ist „`MouseListener`“ ein Interface.

Zu implementierende Methoden im Überblick:

- „`mouseClicked`“: Eine Taste der Maus ist kurz angeklickt worden.
- „`mousePressed`“: Eine Taste der Maus ist nicht nur für einen kurzen „Klick“ heruntergedrückt worden, sondern wird heruntergedrückt festgehalten.
- „`mouseReleased`“: Eine solchermaßen heruntergedrückt festgehaltene Taste ist wieder losgelassen worden.
- „`mouseEntered`“: Der Mauszeiger kommt ins Window.
- „`mousedExited`“: Der Mauszeiger verlässt das Window.

Nun eigener MouseMotionListener:

```
class MeinMouseMotionListener
 implements MouseMotionListener
 {
  public void mouseMoved ( MouseEvent event )
   {
    System.out.println ( "Mouse moved" );
   }
  public void mouseDragged ( MouseEvent event )
   {
    System.out.println ( "Mouse dragged" );
   }
 }
```

Erläuterungen:

- „`mouseMoved`“: Der Mauszeiger ist durch Mausbewegung im Fenster verschoben worden.
- „`mouseDragged`“: Wie „`mouseMoved`“, nur dass eine Maustaste bei der Verschiebung heruntergedrückt gehalten wurde.

Die beiden „Listener“ registrieren:

```
public class MeinApplet extends Applet
  {
    public void init ()
      {
        addMouseListener       ( new MeinMouseListener() );
        addMouseMotionListener ( new MeinMouseMotionListener() );
      }
    public void paint ( Graphics g )
      {
        ...
      }
  }
```


Erläuterungen zum Inhalt von „init“:

- Vgl. Folie 391 ff. zu „`Applet.init`“.
- Die Klasse „`Applet`“ ist vorsorglich schon auf den Fall hin konzipiert, dass die üblichen Maus–Events abgehört werden sollen.
- Aber in „`Applet`“ ist sinnvollerweise noch nicht festgelegt, wie auf das Eintreten eines solchen Events reagiert werden soll.
- Statt dessen delegiert „`Applet`“ diese Festlegungen an Objekte von Klassen mit Urstamm „`MouseListener`“ bzw. „`MouseMotionListener`“.
- Im Code der Klassen dieser Objekte sind die Reaktionen auf diese Events festgelegt.

  ⟶ Konkret im Code der Klassen „`MeinMouseListener`“ bzw. „`MeinMouseMotionListener`“ in unserem Beispiel.

Fortsetzung Erläuterungen:

- Zur Registrierung dieser beiden „Abhorcher“ stellt die Klasse „`Applet`“ die Methoden „`addMouseListener`“ bzw. „`addMouseMotionListener`“ zur Verfügung.
- Wann immer eines der diskutierten Events im Window des Applets eintritt, wird die entsprechende Methode des registrierten Listener–Objektes aufgerufen.

Beispiele:

- Wenn der Mauszeiger in diesem Window ist und eine Maustaste angeklickt wird, wird Methode „`mouseClicked`“ des mit „`addMouseListener`“ registrierten Objekts aufgerufen.

  ⟶ Bei einem Objekt vom Typ „`MeinMouseListener`“ wird also die Notiz „Mouse clicked at“ nebst momentanen Koordinaten des Mauszeigers ausgegeben.
- Wenn der Mauszeiger in diesem Window bewegt wird, wird je nachdem, ob dabei eine Maustaste untengehalten bleibt, entweder „`mouseMoved`“ oder „`mouseDragged`“ des mit „`addMouseMotionListener`“ registrierten Objekts aufgerufen.

  ⟶ Bei einem Objekt vom Typ „`MeinMouseMotionListener`“ wird also die Notiz „Mouse moved“ bzw. „Mouse dragged“ ausgegeben.

Übersicht über Event–Handling bei „Applet“:

- „`MouseListener`“ und „`MouseMotionListener`“: auf den vorherigen Folien ausführlich behandelt.
- „`ComponentListener`“: Behandelt die Events
  - ◇ Window durch anderes Window in den Hintergrund geschoben und damit verdeckt,
  - ◇ Window wieder unverdeckt,
  - ◇ Window ist verschoben worden,
  - ◇ Windowgröße ist verändert worden.

  ⟶ Die Reaktion auf Signal WINCH (Folie 138) muss in einem Java–Programm also in einer Erweiterung des Urstamms „`Component-Listener`“ implementiert werden.
- „`FocusListener`“: Behandelt die Events, dass das Window aktiv bzw. inaktiv gemäß Folie 135 wird.
- „`KeyListener`“: Horcht die Tastatur (engl. *keyboard*) ab.

Konkretes Abschlussbeispiel „KeyListener“:

```
class MeinKeyListener implements KeyListener
 {
  public void keyTyped ( KeyEvent event )
   {
    System.out.println ( "Key typed: " );
    System.out.println ( event.getKeyChar() );
   }
  public void keyPressed ( KeyEvent event )
   {
    System.out.println ( "Key pressed: " );
    System.out.println ( event.getKeyChar() );
   }
  public void keyReleased ( KeyEvent event )
   {
    System.out.println ( "Key released" );
   }
 }
```


Erläuterungen:

- Wann immer ein Event auf der Tastatur passiert, wird die entsprechende Methode des registrierten „`KeyListener`“–Objekts aufgerufen.

  ⟶ Im konkreten Beispiel also ein Objekt vom Typ „`MeinKeyListener`“.
- Methode „`keyTyped`“ bzw. „`keyPressed`“ wird aufgerufen, wenn eine Taste getippt bzw. untengehalten wird.
- Methode „`keyReleased`“ wird aufgerufen, wenn eine untengehaltene Taste wieder freigegeben wird.
- Methode „`getKeyChar`“ von Klasse „`KeyEvent`“ liefert das vom Benutzer eingetippte Zeichen.

  ⟶ Durch „`MeinKeyListener`“ wird bei Antippen oder Untenhalten einer Taste also das damit ausgelöste Zeichen mit ausgegeben.

Wozu dieses ganze Event–Abhorchen?

- Die Implementationen der durch die jeweiligen Urstämme vorgeschriebenen Methoden in „`MeinMouseListener`“, „`MeinMouseMotion-Listener`“, „`MeinKeyListener`“ usw.
  - ◇ sind natürlich nur beispielhaft und instruktiv zu verstehen
  - ◇ und sind daher bewusst einfach gehalten.
- Aber es sollte klargeworden sein, dass alle interaktiven Java–Programme,
  - ◇ die eigene Windows aufmachen
  - ◇ und die auf Mausklicks und Tastatureingaben in diesen Windows reagieren,

  auf die eine oder andere Art auf dem hier vorgestellten Event–Handling basieren.
- Die Implementationen der hier betrachteten Methoden sind in einem solchen Programm dementsprechend umfangreich und komplex.

Klasse „KeyEvent“:

In „`java.awt.event.KeyEvent`“ (Folie 606 ff.) ist nebenbei auch für jede Taste noch eine Klassenkonstante (vom Typ „`int`“) mit einem symbolischen Namen („virtual key“) zur Identifizierung von Tastatureingaben definiert, z.B.

- „`VK_A`“ ... „`VK_Z`“,
- „`VK_a`“ ... „`VK_z`“,
- „`VK_0`“ ... „`VK_9`“,
- „`VK_F1`“ ... „`VK_F12`“ für die 12 Funktionstasten,
- „`VK_COMMA`“, „`VK_SEMICOLON`“ etc. für die entsprechenden Interpunktionszeichen sowie
- „`VK_CONTROL`“, „`VK_META`“, „`VK_SHIFT`“ etc. für die entsprechenden Steuerungstasten.