hochschule mannheim

Fakultät für Informatik

Programmierung 2 (PR2)

01 - Grundlagen prozeduraler Programmierung

Prof. Dr. Frank Dopatka

Hochschule Mannheim University of Applied Sciences

hochschule mannheim

Anweisungen, (primitive) Datentypen und Operatoren

Welche Arten von Variablen gibt es?

In Java werden **3 Arten von Variablen** unterscheiden:

- Lokale Variablen in Methoden oder generell in Blöcken dienen der temporären Speicherung von Daten, während diese Methode ausgeführt wird.

Was ist eine (lokale) Variable?

- Java nutzt Variablen zum Ablegen von Daten.
- Eine Variable ist ein reservierter Speicherbereich und belegt abhängig vom Inhalt eine feste Anzahl von Bytes.
- Alle Variablen haben einen Datentyp, der zur Übersetzungszeit bekannt ist.
- Java ist eine statisch typisierte Programmiersprache (im Gegensatz zu JavaScript oder PHP):
  - Jede Variable bekommt bei ihrer Erstellung einen Datentyp zugewiesen, den sie während ihrer gesamten Lebenszeit behält.

hochschule mannheim

Was ist ein Datentyp?

- Ein Datentyp ist eine Kategorie von Variablen.
- Er bestimmt auch die zulässigen Operationen, die man mit diesem Typ durchführen kann.
  Ein Beispiel:
  - Wahrheitswerte lassen sich nicht addieren, Ganzzahlen schon.
  - Dagegen lassen sich Fließkommazahlen addieren, aber nicht XOR-verknüpfen.

- Beispiele für einfache, sog. primitive Datentypen sind
  - Ganzzahlen,
  - Fließkommazahlen,
  - Wahrheitswerte und
  - Zeichen.

hochschule mannheim

Primitive Datentypen

Java hat eine Reihe von eingebauten Datentypen, die nicht durch Klassen repräsentiert werden. Sie werden primitive Datentypen genannt:

- Logisch – boolean
  Zwei mögliche Werte: true und false
- Ganzzahl – byte, short, int, long
  8, 16, 32, 64 Bit
  vorzeichenbehaftete Ganzzahlen
- Fließkomma – float, double
  32 bzw. 64 Bit Fließkommazahlen
- Zeichen – char
  16 Bit Unicode-Zeichen

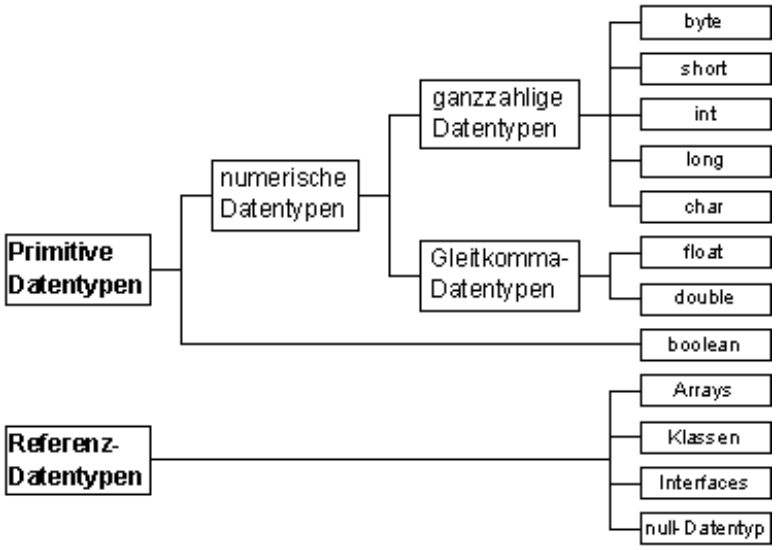
hochschule mannheim

Deklaration von lokalen Variablen

- Mit lokalen Variablen lassen sich Daten speichern, die innerhalb der Methode bzw. eines Blocks gelesen und geschrieben werden können.
- Um lokale Variablen zu nutzen, müssen sie ebenso wie alle anderen Variablen deklariert werden (Java ist streng typisiert)!
- Hinter dem Typnamen folgt der Name der Variablen.
- Die Deklaration selbst ist eine Anweisung und wird wie jede Andere Anweisung in Java mit einem Semikolon abgeschlossen.
- In Java werden Variablen üblicherweise klein geschrieben.

hochschule mannheim

Primitive Datentypen vs. Referenztypen



Primitive Datentypen: Wertebereiche

Schlüsselwort	Länge [Byte]	Belegung (Wertebereich)
boolean	1	true / false
char	2	16bit Unicode-Zeichen (0x0000 ... 0xFFFF)
byte	1	–128 bis 127
short	2	–32.768 bis 32.767
int	4	–2.147.483.648 bis 2.147.483.647
long	8	–9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
float	4	1,40239846E–45 bis 3,40282347E+38
double	8	4,94065645841246544E–324 bis 1,79769131486231570E+308

Primitive Datentypen: Wertebereiche

```
byte b = 126;
b++;
b++;
b++;
b++;
System.out.println(b);
```

```
<terminated>
-126
```

hochschule mannheim

Zahlendarstellung am „signed byte“: dezimal, binär, hexadezimal

dezimal	binär								hexadezimal	
	~~2^7~~	2^6	2^5	2^4	2^3	2^2	2^1	2^0	oberes nibble	unteres nibble
	+/-	64	32	16	8	4	2	1		
1	0	0	0	0	0	0	0	1	0	1
2	0	0	0	0	0	0	1	0	0	2
3	0	0	0	0	0	0	1	1	0	3
4	0	0	0	0	0	1	0	0	0	4
5	0	0	0	0	0	1	0	1	0	5
6	0	0	0	0	0	1	1	0	0	6
7	0	0	0	0	0	1	1	1	0	7
8	0	0	0	0	1	0	0	0	0	8
9	0	0	0	0	1	0	0	1	0	9
10	0	0	0	0	1	0	1	0	0	A
11	0	0	0	0	1	0	1	1	0	B
12	0	0	0	0	1	1	0	0	0	C

hochschule mannheim

Zahlendarstellung am „signed byte“: dezimal, binär, hexadezimal

dezimal	binär								hexadezimal	
	~~2^7~~	2^6	2^5	2^4	2^3	2^2	2^1	2^0	oberes nibble	unteres nibble
	+/-	64	32	16	8	4	2	1		
13	0	0	0	0	1	1	0	1	0	D
14	0	0	0	0	1	1	1	0	0	E
15	0	0	0	0	1	1	1	1	0	F
16	0	0	0	1	0	0	0	0	1	0
17	0	0	0	1	0	0	0	1	1	1
18	0	0	0	1	0	0	1	0	1	2
...										
126	0	1	1	1	1	1	1	0	7	E
127	0	1	1	1	1	1	1	1	7	F
-128	1	0	0	0	0	0	0	0	8	0
-127	1	0	0	0	0	0	0	1	8	1
...										
-2	1	1	1	1	1	1	1	0	F	E
-1	1	1	1	1	1	1	1	1	F	F
0	0	0	0	0	0	0	0	0	0	0

Die Antwort liegt in der Binärdarstellung!

dezimal	binär

`126`	`0111 1110`
	`1x64 + 1x32 + 1x16 + 1x8 + 1x4 + 1x2 + 0x1`
	`= 64 + 32 + 16 + 8 + 4 + 2 = 126`

`127`	`0111 1111`
`-128`	`1000 0000` Das 1. Bit im Byte ist das Vorzeichenbit; Ist es gesetzt, so ist die Zahl negativ und die restlichen Bits werden „anders herum“ addiert: `-128 + 0 = -128`
`-127`	`1000 0001` `-128 + 1x1 = -128 + 1 = -127`
`-126`	`1000 0010` `-128 + 1x2 = -128 + 2 = -126`

Deklarationen & Zuweisungen

```
// Deklaration
char c;
String str;
int x, y;
// Deklaration mit Initialisierung
float z = 1.712f;
double w = 3.1415;
boolean wahrheit = true;
String str1 = "Hi";
// Zuweisung
c = 'Z';
str = "Hallo Welt";
x = 60;
y = 400;
```

Wie arbeitet man mit primitiven Datentypen?

- Arithmetische Operatoren
  - Nur auf Zahlen anwendbar; Addition auch bei Zeichenketten;
  - Inkrement/Dekrement auch bei Zeichen (char)

`a + b`	Addition
`a - b`	Subtraktion
`a * b`	Multiplikation
a / b	Division
`a % b`	Modulo: gibt ganzzahligen Rest der Division zurück, z.B. 7%3=1, 8%3=2, 9%3=0, 10%3=1
`syso(a++)`	gibt a aus und inkrementiert a danach
`syso(a--)`	gibt a aus und dekrementiert a danach
`syso(++a)`	inkrementiert a und gibt a danach aus
`syso(--a)`	dekrementiert a und gibt a danach aus

Wie arbeitet man mit primitiven Datentypen?

- Bit-Operatoren
  - Nur bei ganzen Zahlen sinnvoll

`a & b` binäres UND: „beide müssen 1 sein“

0	0	0
0	1	0
1	0	0
1	1	1

z.B. 1101 (13) & 0110 (6) = 0100 (4)

Wie arbeitet man mit primitiven Datentypen?

- Bit-Operatoren
  - Nur bei ganzen Zahlen sinnvoll

`a | b` binäres ODER: „mindestens einer muss 1 sein“

0	0	0
0	1	1
1	0	1
1	1	1

z.B. 1101 (13) | 0110 (6) = 1111 (15)

Wie arbeitet man mit primitiven Datentypen?

- Bit-Operatoren
  - Nur bei ganzen Zahlen sinnvoll

`a ^ b` binäres XOR: „genau einer muss 1 sein“

0	0	0
0	1	1
1	0	1
1	1	0

z.B. 1101 (13) ^ 0110 (6) = 1011 (11)

`~a` binäres NOT

0	1
1	0

z.B. ~1101 (13) = 0010 (2)

Wie arbeitet man mit primitiven Datentypen?

- Vergleichsoperatoren
  - Ergebnis ist immer ein Wahrheitswert true/false
  - a und b muß miteinander vergleichbar sein
  - Anwendung in Verzweigungen oder Schleifen als Bedingung, welcher Weg im Programm weiter gegangen werden soll!

`a > b`	Ist a größer als b?
`a >= b`	Ist a größer als b oder gleich?
`a == b`	Ist a gleich b?
`a <= b`	Ist a kleiner als b oder gleich?
`a < b`	Ist a kleiner als b?
`a != b`	Ist a ungleich b?

Wie arbeitet man mit primitiven Datentypen?

- Logische Operatoren
  - Ergebnis ist immer ein Wahrheitswert true/false;
  - a und b sind hier beides Bedingungen, die selbst einen Wahrheitswert liefern
  - Anwendung in Verzweigungen oder Schleifen als Bedingung, welcher Weg im Programm weiter gegangen werden soll!

`a & b` Ist a und b true, so ergibt sich true, ansonsten false.
Sowohl a, als auch b werden immer ausgewertet.
Sinnvoll u.a. bei

```
if (a & (b++ < 10)) ...
```

`a && b` Ist a und b true, so ergibt sich true, ansonsten false.
Ist a false, wird b nicht mehr ausgewertet.
Sinnvoll u.a. bei

```
if ((a!=null) && (a.length()>2)) ...
```

Wie arbeitet man mit primitiven Datentypen?

- Logische Operatoren
  - Ergebnis ist immer ein Wahrheitswert true/false;
  - a und b sind hier beides Bedingungen, die selbst einen Wahrheitswert liefern
  - Anwendung in Verzweigungen oder Schleifen als Bedingung, welcher Weg im Programm weiter gegangen werden soll!

`a | b`	Ist a oder b true oder beide, so ergibt sich true, ansonsten false. Sowohl a, als auch b werden immer ausgewertet.
`a || b`	Ist a oder b true oder beide, so ergibt sich true, ansonsten false. Ist a true, wird b nicht mehr ausgewertet.
`!a`	Ist a false, so ergibt sich true und umgekehrt.
	`(!(x<0))` ist identisch zu `(x>=0)`

Wie arbeitet man mit primitiven Datentypen?

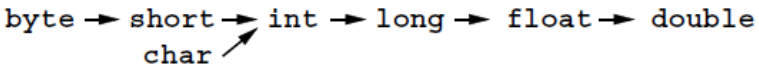
- Zuweisungsoperatoren

`a = b`	setzt a auf denselben Wert wie b
`a += b`	erhöht a um b; a=a+b
`a -= b`	zieht b von a ab; a=a-b
`a *= b`	multipliziert b mit a; a=a*b
`a /= b`	dividiert b durch a; a=a/b

```
a += (-5) <=> a + (-5) <=> a - 5 <=> a -= 5
```

Konvertierung von Datentypen: automatischer Type-Cast

- Immer nur von „kleineren“ zu „größeren“ Typen:



- In Ausdrücken (Operand1 X Operand2):

Mind. ein Op.	andere Operanden	konv. zu
double	double, float, long, int, short, byte, char	double
float	float, long, int, short, byte, char	float
long	long, int, short, byte, char	long
int, short, byte, char	int, short, byte, char	int

Konvertierung von Datentypen: manueller Type-Cast

- Der Typ eines Ausdrucks kann in Grenzen angepaßt werden:
  - Implizite, automatische Typkonversion
  - Explizite, manuelle Typkonversion
- Explizite Typkonvertierung:
- Dabei kann Information verloren gehen
- Keine Konversion von & nach boolean
- Typkonvertierung erfolgt durch Voranstellen von (<Typ>), z.B.:

```
d = (double) 3; // == 3.0
c = (char) 65;  // == 'A'
i = (int) x+y;  // == ((int)x) + y
i = (int) (x+y);
```

Konvertierung von Datentypen: Beispiele

```
int i = 1;
double d;

d = i / 2;
System.out.println(d);

d = (double) i / 2;
System.out.println(d);

d = i / 2.0;
System.out.println(d);
```


Konvertierung von Datentypen: Beispiele

```
double x,y=1.1;
int i,j=5;
char c='a';
boolean b=true;

x = 3;       // OK: x==3.0
x = 3/4;     // x==0.0
x = x+y*j;   // OK, Konvertierung nach double
i = j+x;     // FEHLER: (j+x) hat den Typ double
i = (int)x; // OK, i ist dann der ganzzahlige Teil von x
i = (int)b; // FEHLER: Keine Konvertierung von boolean
b = i;       // FEHLER: Keine Konvertierung nach boolean
b = (i!=0); // OK, der Vergleich liefert einen boolean
c = c+1;     // FEHLER: c+1 hat den Typ int
c += 1;      // OK, c=='b'
```

3 Arten von Operatoren

- Unäre Operatoren haben nur einen Operanden
  Syntax: op Operand
  Beispiel: Negation einer Zahl (a = ≠b)
- Binäre Operatoren verknüpfen zwei Operanden
  Syntax: Operand1 op Operand2
  Beispiel: Multiplikation zweier Zahlen (a = b * c)
- Ternäre Operatoren sind selten; sie verknüpfen drei Operanden
  Syntax: Operand1 op Operand2 op Operand3
  Beispiel: Bedingter Ausdruck (x = a > b ? a : b)

Prioritäten setzen

Bezeichnung	Operatorsymbol	Priorität
Klammern	() []	13
Negation Inkrement Dekrement	- ! ~ ++ --	12
Arithmetische Operatoren	* / %	11
	+ -	10
Shift Operatoren	<< >> >>>	9
Vergleichsoperatoren	> >= < <=	8
	== !=	7
Bitweise Operatoren	&	6
	^	5
	|	4
Logische Operatoren	&&	3
	||	2
Zuweisungsoperator	= += -= *= /= %= >>= <<= &= ^= |=	1

hochschule mannheim
Prioritäten setzen

- In welcher Reihenfolge werden die Operatoren bei Ausdrücken der Form
  `<Operand> <Operator> <Operand> <Operator> <Operand>...`
  ausgewertet?
- Die Reihenfolge richtet sich
  - nach der Priorität der Operatoren.
  - wobei Operatoren höherer Priorität vor jenen niedrigerer Priorität ausgewertet werden.
  - Bei Operatoren gleicher Priorität nach der Assoziativität: von links nach rechts oder von rechts nach links
- Gute Praxis: Im Zweifelsfall Klammern verwenden!

hochschule mannheim
Vorsicht bei den Datentypen!

- Bei Überlauf der Fließkomma-Wertebereiche und bei Division durch 0: [-] Infinity
- Bei 0.0 / 0.0 und ähnlichem: Nan (Not a Number)
- Reelle Zahlen können im Computer nicht exakt dargestellt werden; dadurch entstehen Rundungsfehler!
- In der Fließkomma-Arithmetik gelten Assoziativitäts- und Kommutativitäts-gesetz nicht mehr!

Lösung:
Festkomma-Zahlen verwenden mit der Klasse BigDecimal!

```
double a,b;

a = 5.0e-12; // a =        0.000 000 000 005
b = 1.0e+5;  // b = 100 000.000 000 000 000

System.out.println(a+a+b); // 100000.00000000001
System.out.println(b+a+a); // 100000.00000000000
```

hochschule mannheim
Das „+“ in Zeichenketten!

- Zusammenfügen / Konkatenieren:

```
String s = "Zahl 1" + "2";    // Zahl12
s += " ist gleich 3";         // Zahl12 ist gleich 3
s = "elf ist " + 1 + 1;       // elf ist 11
s = 1 + 1 + " ist zwei";      // 2 ist zwei
s = 0.5 + 0.5 + " ist " + 1; // 1.0 ist 1
```

- Der Operator + ist überladen:
  - int + int: Addition
  - String + String: Konkatenation
  - String + int und int + String:
    Umwandlung der Zahl in einen String und anschließende Konkatenation
- Dies gilt analog für die anderen primitiven Datentypen

hochschule mannheim
Konstanten mit final

- Der Modifier **`final`** zeigt an, daß eine initialisierte Variable nicht veränderbar ist.
- Es handelt sich dann um eine benannte Konstante.
- Konstanten sind explizite Datenwerte im Programm.
- Konstanten können bereits vom Compiler interpretiert werden.

Gültigkeit von Variablen

- Eine in einer Methode deklarierte Variable, also eine lokale Variable, ist ab der Deklaration bis zum Ende des umgebenden Blocks gültig:

```
public static void methodenName(){
  int a=3, b=1;
  if (a>0){
    int b; // FEHLER
    int c;
  }
  {
    char c;
  }
  double a; // FEHLER
}
```


Wie kommt man von einer Idee zu einem Algorithmus?

Was ist ein Algorithmus?

- Unter einem Algorithmus (auch: Lösungsverfahren) versteht man eine genau definierte Handlungsvorschrift zur Lösung eines Problems oder einer bestimmten Art von Problemen in endlich vielen Schritten.
- Algorithmen können in der UML als Aktivitätsdiagramme grafisch dargestellt werden.
- Algorithmen sind eines der zentralen Themen der Informatik.
  - Sie sind Gegenstand einiger Spezialgebiete der Theoretischen Informatik, der Komplexitätstheorie und der Berechenbarkeitstheorie.
  - In Form von Programmen und elektronischen Schaltkreisen steuern sie Computer und andere Maschinen.

hochschule mannheim

Was ist ein Algorithmus?

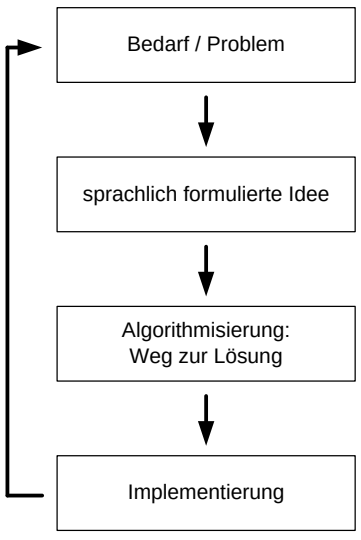
- Er beschreibt einen zu beschreitenden - vorher definierten - Weg, welcher zur Erreichung eines Ziels zu gehen ist.
- Es ist eine Folge von mechanisch ausführbaren Anweisungen.
- Es ist eine schrittweise Lösung eines Problems mittels einer endlichen Anzahl von Regeln.
- Algorithmen sind Verfahren, mit denen sich Probleme verschiedenster Art lösen lassen.
- Der Begriff „Algorithmus" ist scheinbar abstrakt genug, um ein oft genanntes Beispiel - den Vergleich mit einem Kochrezept - zu erlauben:

„Schrittweise,
genau und in endlichen Schritten durchführbar"

Sind Kochrezepte Algorithmen?

- Fasst man den Algorithmus-Begriff allgemein, so lassen sich im Alltag viele Formen davon finden:
  - Das Abfertigen eines Kassiervorgangs im Supermarkt,
  - das Spielen einer Melodie beim Musizieren oder eben auch
  - beim Kochen nach einem Rezept.
- Nach der Definition ist ein Kochrezept kein Algorithmus, da es ist beispielsweise nicht effektiv durchführbar ist:
  - Beschreibungen wie „eine Prise Salz" oder "man schmecke ab" sind nicht von einem Computer ausführbar, da sie zu ungenau sind.
  - Des Weiteren variiert das Ergebnis geschmacklich meist, selbst wenn man nach einem Rezept kocht; vom Problem der genauen Messbarkeit einmal abgesehen.

Vereinfachte Vorgehensweise zur Problemlösung



Abläufe darstellen mit der UML

Was ist UML?

- Die Unified Modeling Language, kurz UML (deutsch: „Vereinheitlichte Modellierungssprache“), ist eine standardisierte Sprache für die Modellierung von Software und Systemen.
- UML definiert Bezeichner für die meisten für die Modellierung wichtigen Begriffe und legt mögliche Beziehungen zwischen diesen Begriffen fest.
- UML definiert graphische Notationen für diese Begriffe und für Modelle von statischen (Daten-)Strukturen & dynamischen Abläufen.
- UML ist heute eine der dominierenden Sprachen für die Modellierung von betrieblichen Anwendungs- bzw. Softwaresystemen.

hochschule mannheim

Wer arbeitet mit UML?

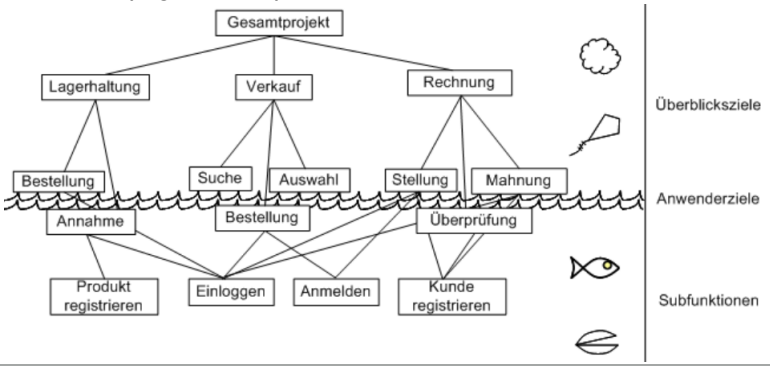
- Der erste Kontakt zu UML besteht häufig darin, dass Diagramme in UML im Rahmen von Softwareprojekten zu erstellen, zu verstehen oder zu beurteilen sind:
  - Auftraggeber & Fachvertreter prüfen und bestätigen zum Beispiel Anforderungen an ein System, die Wirtschaftsanalytiker festgehalten haben.
  - Softwareentwickler realisieren Arbeitsabläufe, die Wirtschaftsanalytiker in Zusammenarbeit mit Fachvertretern in Aktivitätsdiagrammen beschrieben haben.
  - Systemingenieure installieren & betreiben Softwaresysteme basierend auf einem Installationsplan, der als Verteilungsdiagramm vorliegt.

hochschule mannheim

Aktivitätsdiagramme grob & fein

- Aktivitätsdiagramme lassen sich
  - sehr grob zusammen mit dem Kunden in der Perspektive der Geschäftsprozesse, aber auch
  - sehr fein zusammen mit dem Entwickler in der Perspektive der Programm-Abläufe (Algorithmen)

  erstellen.



hochschule mannheim

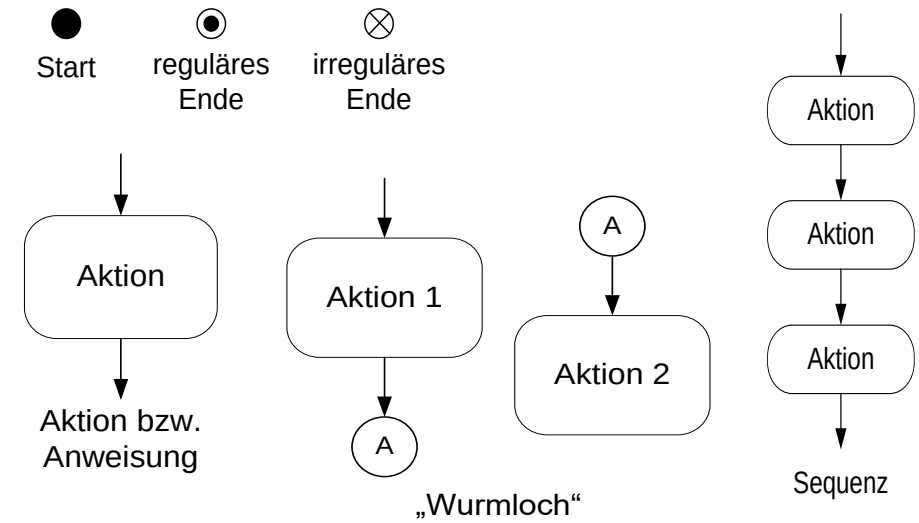
Welche UML-Version verwenden wir?

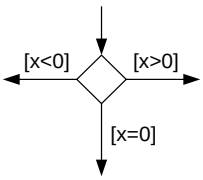
- Wir verwenden strikt die UML-Notationsübersicht der OOSE in der Version 2.5.
- Die OOSE arbeitet in internationalen Gremien und an der Entwicklung von Zertifizierungsprogrammen mit.
- Beispiele sind die
  - Co-Entwicklung der Zertifizierungsprogramme OMG-Certified Expert in BPM 2 (OCEB2),
  - OMG-Certified UML Professional (OCUP2) und
  - OMG-Certified Systems Modeling Professional (OCSMP) sowie das Mitwirken an den Spezifikationen zur
    - UML,
    - BPMN und
    - SysML.

https://www.oose.de/wp-content/uploads/2012/05/UML-Notations%C3%BCbersicht-2.5.pdf

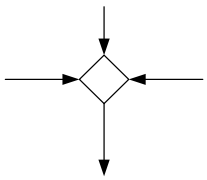
hochschule mannheim

Syntax eines Aktivitätsdiagramms UML-AD



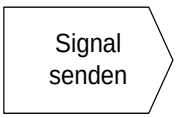


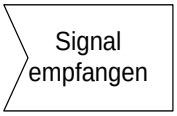
Verzweigung

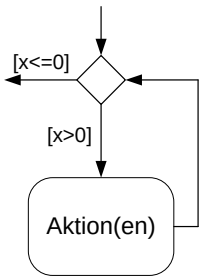


Zusammenführung

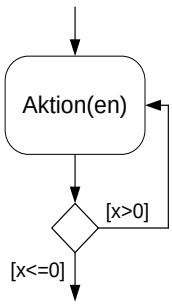
Zeitereignis empfangen







Verzweigung und Zusammenführung (kopfgesteuerte Schleife)



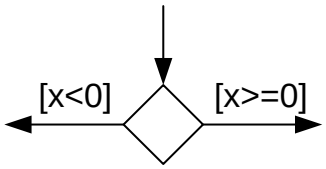
Verzweigung und Zusammenführung (fußgesteuerte Schleife)

Verzweigungen

Sie haben die Wahl...

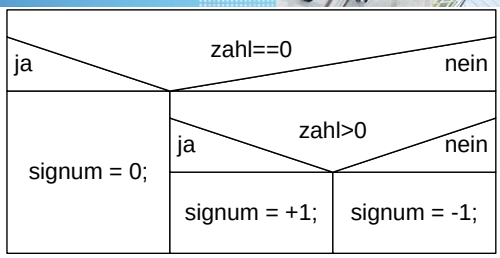
Was ist eine Verzweigung?

- Eine Verzweigung ist...
  - eine Entscheidung für einen Weg.
  - wenn-dann-sonst.
  - entweder hier lang, oder da lang.
  - immer mit einer Bedingung verbunden:
    Ist die Bedingung erfüllt, dann gehts hier lang, ansonsten da lang.

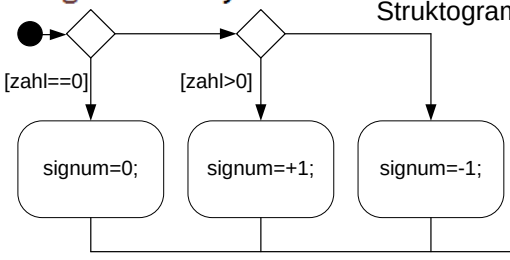


Was ist eine Verzweigung?

```
if(zahl==0){
  signum=0;
}
else{
  if(zahl>0)
    signum = +1;
  else
    signum = -1;
}
```



Nassi-Shneiderman-Diagramm
Struktogramm, DIN 66261



UML-
Aktivitätsdiagramm

hochschule mannheim
Bildung von Blöcken

- Von der Syntax her greift die **`if`**-Kontrollstruktur immer für das nächste Statement oder den nächsten Block.
- Schreiben Sie besser immer einen Block, auch wenn nur ein einziges Statement unter der **`if`**-Bedingung stehen soll.
- Tipp:
  - Mit Strg+Shift+F rückt Eclipse automatisch immer richtig ein.
  - Wenn Sie auf dem Papier coden, hilft Ihnen das leider nicht.

```
if (lottogewinn) {
  freuen();
  jobKuendigen();
}

if (lottogewinn)
  freuen();
  jobKuendigen(); // Wird immer ausgeführt!!!
```


hochschule mannheim
Zuweisung vs. boolsche Abfrage

```
boolean b = false;
if (b = true) {
  System.out.println("Oops");
}
```

Operatoren in Verzweigungen

```
String s = null;
if ((s != null) & (s.length() > 0)) {
  System.out.println(s.charAt(0));
}
System.out.println(s);
```


Das böse Semicolon

- An dieser Stelle ist ein Hinweis angebracht:
  Ein Java-Anfänger schreibt gerne hinter die schließende Klammer der `if`-Anweisung ein Semikolon.
- Das führt jedoch zu einer ganz anderen Ausführungsfolge.
  Ein Beispiel:

```
int alter=29;
if(alter<0);
  System.out.println("Aha, noch im Mutterleib!");
if(alter>150);
  System.out.println("Wow, uralt!");
```

- Streng genommen ist dies ein Fehler bei der Einrückung des Codes; Ecllipse würde dies mit Strg+Shift+F korrigieren.
- Bei Abgabe einer Prüfungsleistung würden Sie immer einen Abzug wegen „irreführenden Codes“ erhalten.

if-else Verzweigung

- Neben der einseitigen Alternative existiert in Java und auch in vielen anderen Sprachen die zweiseitige Alternative.
- Das optionale Schlüsselwort **`else`** mit angehängter Anweisung veranlasst die Ausführung einer Alternative, wenn die Bedingung **`false`** ist.

```
if(x<y){
  System.out.println("x ist kleiner als y.");
}
else{
  System.out.println("x ist größer oder gleich y.");
}
```


Problem des hängenden else

- Bei Verzweigungen mit else gibt es ein bekanntes Problem, das Hängendes-Else-Problem genannt wird.
- Zu welcher Anweisung gehört das folgende **`else`**?

```
if(x==y)
  if(a!=b)
    System.out.println("Ausgabe1");
else
  System.out.println("Ausgabe2");
```

- Die Einrückung suggeriert, dass das else die Alternative zur ersten **`if`**-Anweisung ist.
- Dies ist aber nicht richtig. Die Syntax von Java ist so definiert, dass das else zum letzten **`if`** gehört.

Lösung des hängenden else

```
if(x==y){
  if(a!=b){
    System.out.println("Ausgabe1");
  }
}
else{
  System.out.println("Ausgabe2");
}
```

- So kann eine Verwechslung gar nicht erst aufkommen.
- Wenn das else immer zum innersten **`if`** gehört und das nicht erwünscht ist, können wir mit geschweiften Klammern arbeiten.

if-elseif-else Verzweigung

- **`if`**-Anweisungen zur Programmführung kommen häufig in Programmen vor.
- Noch häufiger ist es, eine Variable auf einen bestimmten Wert zu prüfen.
- Dazu werden **`if`**- und **`if/else`**-Anweisungen gerne geschachtelt (kaskadiert).
- Wenn eine Variable einem Wert entspricht, dann wird eine Anweisung ausgeführt, sonst wird die Variable mit einem anderen Wert getestet und so weiter.
- Die eingerückten Verzweigungen nennen sich auch angehäufte **`if`**-Anweisungen oder **`if`**-Kaskade, da jede **`else-Anweisung`** ihrerseits weitere **`if`**-Anweisungen enthält, bis alle Abfragen gemacht sind...

if-elseif-else Verzweigung

```
if (monat==4)
  tage=30;
else if (monat==6)
  tage=30;
else if (monat==9)
  tage=30;
else if (monat==11)
  tage=30;
else if (monat==2)
  if (schaltjahr)
    tage=29;
  else
    tage=28;
else
  tage=31;
```


if-elseif-else Verzweigung: Kürzer mit einer or-Verknüpfung

```
if((monat==4)||(monat==6)||(monat==9)||(monat==11))
  tage=30;
else if (monat==2){
  if (schaltjahr)
    tage=29;
  else
    tage=28;
}
else
  tage=31;
```

if-elseif-else Verzweigung: Auch Funktionsaufrufe sind möglich

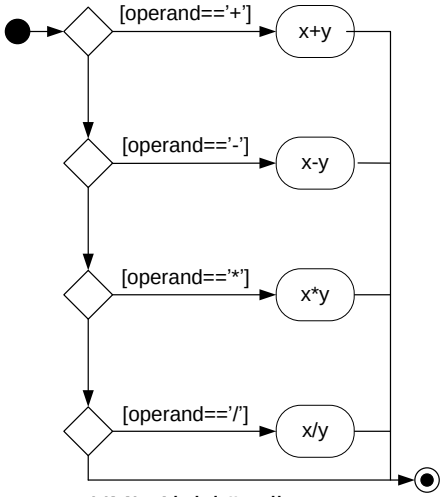
```
int count = getCount(); // Funktion im Programm
if (count < 0) {
  System.out.println("Fehler: Zahl ist negativ.");
} else if (count > getMaxCount()) { // Funktion im Programm
  System.out.println("Fehler: Zahl ist zu groß.");
} else {
  System.out.println("Es kommen " + count + " Leute");
}
```


switch Verzweigung

- Eine Kurzform für speziell gebaute, angehäufte **`if`**-Anweisungen bietet **`switch`**.
- Im übersichtlichen **`switch`**-Block gibt es eine Reihe von unterschiedlichen Sprungzielen, die mit **`case`** markiert sind.
- Der Compiler sucht eine bei **`case`** genannte Konstante, die mit dem in **`switch`** angegebenen Ausdruck übereinstimmt.
- Gibt es einen Treffer, so werden alle beim case folgenden Anweisungen ausgeführt, bis ein (optionales) **`break`** die Abarbeitung beendet.
- Ohne **`break`** geht die Ausführung im nächsten **`case`**-Block automatisch weiter!
- **`break`** ist eine Sprunganweisung (ähnlich zu goto in Basic), die den Regeln des strukturierten Programmierens widerspricht.
  Daher lehnen einige Programmierer die Verwendung der **`switch`**-Anweisung ab.

switch Verzweigung

```
switch (operand){
case '+':
  System.out.println(x+y);
  break;
case '-':
  System.out.println(x-y);
  break;
case '*':
  System.out.println(x*y);
  break;
case '/':
  System.out.println(x/y);
  break;
}
```



UML-Aktivitätsdiagramm

switch Verzweigung: default für alle Anderen

```
switch (mitarbeiterArt) {
  case 0: // Manager
    addEinzelnesBuero();
    addFirmenwagen();
    addSekretaerin();
    break;
  case 1: // Senior-Consultant
    addEinzelnesBuero();
    addFirmenwagen();
    break;
  default: // der normale Mensch
    addGrossraumbuero();
}
```

switch Verzweigung: Einschränkungen

- Ausdrücke sind auf den primitiven Datentyp **`int`** beschränkt.
- Elemente vom Datentyp **`byte`**, **`char`** und **`short`** sind somit auch erlaubt, da der Compiler den Typ automatisch auf **`int`** castet.
- Es können nicht die anderen primitiven Datentypen **`boolean`**, **`long`**, **`float`**, **`double`** benutzt werden.
- Die bei **`switch`** genannten Werte müssen konstant sein.
- Dynamische Ausdrücke, z.B. Rückgaben aus Methodenaufrufen, sind nicht möglich.
- Es sind keine Bereichsangaben möglich. Das wäre z.B. bei Altersangaben nützlich.
- Objekt-Typen sind nicht erlaubt, wohl aber feste Zeichenketten (Strings) ab Java 7.

switch Verzweigung: Seit Java7 auch mit Strings

```
switch (mitarbeiterArt) {
  case "Manager":
    addEinzelnesBuero();
    addFirmenwagen();
    addSekretaerin();
    break;
  case "Senior-Consultant":
    addEinzelnesBuero();
    addFirmenwagen();
    break;
  default:
    addGrossraumbuero();
}
```

Warnung vor dem vergessenen break

```
int x = 3;
switch (x) {
  case 1:
    System.out.print("1");
  case 2:
    System.out.print("2");
  case 3:
    System.out.print("3");
  case 4:
    System.out.print("4");
  default:
    System.out.print("Passt nicht");
}
```


break und fall-through

- Bisher wurde in die letzte Zeile stets eine **`break`**-Anweisung gesetzt.
- Ein häufiger Programmierfehler ist es, das **`break`** zu vergessen!
- Ohne ein **`break`** würden nach einer Übereinstimmung alle nachfolgenden Anweisungen ausgeführt.
- Sie laufen somit in einen neuen Abschnitt herein, bis ein **`break`** oder das Ende von **`switch`** erreicht ist.
- Da dies vergleichbar mit einem Spielzeug ist, bei dem Kugeln von oben nach unten durchfallen, nennt sich dieses auch Fall-Through.
- Ein beabsichtigter Fall-Through soll immer als Kommentar angegeben werden...

break und fall-through

```
char testVokal='i';

switch (testVokal){
  case 'a': // fall through
  case 'e':
  case 'i':
  case 'o':
  case 'u':
    System.out.println(testVokal+" ist Vokal");
    break;
  default:
    System.out.println(testVokal+" ist kein Vokal");
}
```


break und fall-through

```
if ((monat<1)||(monat>12))
  throw new RuntimeException("Was ist das denn?");
switch (monat){
  case 4: // fall through
  case 6:
  case 9:
  case 11:
    tage=30;
    break;
  case 2:
    if (schaltjahr)
      tage=29;
    else
      tage=28;
    break;
  default:
    tage=31;
}
```

Schleifen

Und noch eine Runde...
Oder auch nicht...

Was machen Schleifen?

- Schleifen dienen dazu, bestimmte Anweisungen immer wieder abzuarbeiten.
- Zu einer Schleife gehören die Schleifenbedingung, auch Schleifenkörper genannt, sowie der Rumpf der Schleife.
- Die Schleifenbedingung, stets ein boolescher Ausdruck, entscheidet darüber, unter welcher Bedingung die Wiederholung ausgeführt wird.
- In Abhängigkeit von der Schleifenbedingung kann der Rumpf mehrmals ausgeführt werden.
- Dazu wird bei jedem Schleifen-Durchgang die Schleifenbedingung geprüft.

while Schleife

- Solange der boolesche Ausdruck **`true`** liefert, wird die Anweisung ausgeführt und ggfs. gar nicht.
- Die Überprüfung findet vor Ausführung der Anweisung statt:
  - Die while-Schleife ist eine abweisende, kopf-gesteuerte Schleife.
- Bei mehr als einer Anweisung im Schleifen-Rumpf ist ein Block {} zu definieren!

while Schleife

```
int summe = 0;
int i = 1;

while (i <= 10){
  summe += i;
  i++;
}
```

summe=0	
i=1	
solange i kleiner oder gleich 10	
	summe um i erhöhen
	i um 1 erhöhen

Struktogramm

while Schleife: Endlos!

- Ist die Bedingung einer while Schleife immer wahr, dann handelt es sich um eine Endlosschleife.
- Die Konsequenz ist, dass die Schleife endlos wiederholt wird:

```
while (true){
  // immer wieder und immer wieder
}
```

- In Eclipse lassen sich Programme von außen beenden.
- Dazu bietet die Ansicht „Console“ eine rote Schaltfläche in Form eines Quadrats, die nach der Aktivierung im Fall eines laufenden Programms die VM mit den laufenden Programmen beendet.

hochschule mannheim
do-while Schleife

- Führt die Anweisung aus, solange der (boolesche) Ausdruck **`true`** liefert, mindestens jedoch ein Mal.
- Die Überprüfung findet nach Ausführung der Anweisung statt dieser Schleifentyp ist eine annehmende (fuß-gesteuerte) Schleife
- Bei mehr als einer Anweisung im Schleifen-Rumpf ist ein Block {} zu definieren!

hochschule mannheim
do-while Schleife

```
int pos = 1;
do {
  System.out.println(pos);
  pos++;
} while (pos<=10); // Beachte das Semikolon!
```

pos=1	
	pos ausgeben
	pos um 1 erhöhen
solange pos kleiner oder gleich 10	

hochschule mannheim
do-while Schleife

```
// Kindersimulator
int i = 0;
do {
  System.out.println("Sind wir schon da?");
  System.out.println("Ist es noch weit?");
  i++;
} while (i < 10);
System.out.println("Jetzt sind wir angekommen!");
```


hochschule mannheim
for Schleife

- Die **`for`** Schleife ist eine spezielle Variante einer **`while`** Schleife und wird typischerweise zum Zählen benutzt.
- Genauso wie while Schleifen sind **`for`** Schleifen kopf-gesteuert:
  Der Rumpf wird also erst dann ausgeführt, wenn die Bedingung wahr ist.
- Bei mehr als einer Anweisung im Schleifen-Rumpf ist ein Block {}
  zu definieren!

for Schleife

```
for (int i=1;i<=10;i++){ // i ist der Schleifenzähler
  System.out.println(i);
}
```

- Eine genauere Betrachtung der Schleife zeigt deren Bestandteile:
  1. Initialisierung der Schleife
     `int i=1` oder `i=1`, falls `i` zuvor deklariert wurde
  2. Schleifenbedingung
     `i<=10`; das Ergebnis immer boolean
  3. Schleifen-Inkrement
     `i++`; oder eine andere Modifikation von `i`

for Schleife

```
int summe=0;
for (int i=1;i<=10;i++){
  summe+=i;
}
```

summe=0	
i=1	
zähle i von 1 bis 10, Schrittweite 1	
	summe um i erhöhen

for Schleife: Endlos!

- Da alle drei Ausdrücke im Kopf der Schleife optional sind, können sie weggelassen werden, und es ergibt sich eine Endlosschleife.
- Diese Schreibweise ist somit semantisch äquivalent mit **`while(true)`**:

```
for ( ; ; ){
  // immer wieder und immer wieder
}
```


Geschachtelte Schleifen

- Schleifen können in beliebiger Kombination geschachtelt werden.
- Die übergeordnete Schleife nennt sich äußere Schleife, die untergeordnete innere Schleife.
- Im folgenden Beispiel wird die äußere Schleife die Zeilen zählen und die innere Schleife die Sternchen in eine Zeile ausgeben, also für die Spalte verantwortlich sein:

```
for (int i=1;i<=5;i++){
  for (int j=1;j<=i;j++){
    System.out.print("*");
  }
  System.out.println();
}
```

```
<terminated> Klasse02
*
**
***
****
*****
```

for-each Schleife

- Die erweiterte Form der `for` Schleife löst sich vom Schleifen-Index und erfragt jedes Element eines Arrays bzw. einer Collection.
- Das lässt sich als Durchlauf einer Menge vorstellen, denn der Doppelpunkt liest sich als »in«.
- Rechts vom Doppelpunkt steht immer ein Array bzw. eine Collection.
- Links wird eine lokale Variable deklariert, die später beim Ablauf jedes Element der Sammlung annehmen wird.

hochschule mannheim

for-each Schleife

```
int[] primzahlen = { 2, 3, 5, 7, 11, 13, 17, 19 };
int summe = 0;
// for each primzahl in der Menge primzahlen,
// tue folgendes...
for (int primzahl : primzahlen) {
  summe += primzahl;
}
```

for vs. for-each Schleife

```
int[] daten={23,4,3,6,2,456};

// konventionell
int sum = 0;
for (int i=0;i<daten.length;i++)
  sum+=daten[i];
System.out.println(sum);

// for-each n in daten erhöhe sum2 um n
int sum2 = 0;
for (int n:daten)
  sum2+=n;
System.out.println(sum2);
```


break & continue

- **`break`**:
  Springt aus Schleifen und switch-Statements heraus.
- **`continue`**:
  Springt an das Ende der Schleife und führt den Schleifen-Test erneut aus.

Wann welche Schleife?

- **`for`** Schleife:
  - Für Zählschleifen (z.B. „für alle i = 1 ...N“)
  - Wenn natürlicherweise eine Schleifen-Variable vorhanden ist oder benötigt wird.
- **`for-each`** Schleife
  - Zum Durchlaufen von Objekt-Mengen, deren Obergrenze nicht bekannt ist (Collections).
  - Wenn man keine Schleifen-Variable benötigt.
- **`while`** Schleife:
  - Wenn die (max.) Zahl der Wiederholungen nicht im Voraus bekannt ist.
  - Bei komplexen Wiederholungsbedingungen.
- **`do-while`** Schleife:
  - Wenn der Schleifenrumpf mindestens einmal ausgeführt werden soll (Benutzer-Eingaben).

Tracing

hochschule mannheim
Was ist Tracing?

- Mit Ablaufverfolgung (engl. tracing) bezeichnet man in der Programmierung...
  - die Analyse von Programmen und/oder
  - die Fehlersuche in Programmen.
- Dabei wird bei jedem Einsprung in eine Methode, sowie bei jedem Verlassen eine Meldung ausgegeben, sodass der Programmierer mitverfolgen kann, wann und von wo welche Methode aufgerufen wird. Die Meldungen können auch die Argumente an die Methode enthalten.
- Eine andere Möglichkeit besteht darin, den Code händisch Zeile für Zeile durchzugehen und den aktuellen Zustand aufzu-schreiben oder zu merken.
  - Ein Anfänger kann dies auf einem Blatt Papier durchführen.
  - Bei mehr Erfahrung kann dies auch „im Kopf des Programmierers“ erfolgen.

hochschule mannheim
Tracing einer while-Schleife

```
int i=1;
int summe=0;
while (i<=3){
  summe += i;
  i++;
}
System.out.println(summe);
```

i:	1

Tracing einer while-Schleife

```
int i=1;
int summe=0;
while (i<=3){
 summe += i;
 i++;
}
System.out.println(summe);
```

i:	1
summe:	0

Tracing einer while-Schleife

```
int i=1;
int summe=0;
while (i<=3){ // true
 summe += i;
 i++;
}
System.out.println(summe);
```

i:	1
summe:	0

hochschule mannheim
Tracing einer while-Schleife

```
int i=1;
int summe=0;
while (i<=3){
 summe += i;
 i++;
}
System.out.println(summe);
```

i:	1
summe:	1

hochschule mannheim
Tracing einer while-Schleife

```
int i=1;
int summe=0;
while (i<=3){
 summe += i;
 i++;
}
System.out.println(summe);
```

i:	2
summe:	1

Tracing einer while-Schleife

```
int i=1;
int summe=0;
while (i<=3){ // true
 summe += i;
 i++;
}
System.out.println(summe);
```

i:	2
summe:	1

Tracing einer while-Schleife

```
int i=1;
int summe=0;
while (i<=3){
 summe += i;
 i++;
}
System.out.println(summe);
```

i:	2
summe:	3

hochschule mannheim
Tracing einer while-Schleife

```
int i=1;
int summe=0;
while (i<=3){
 summe += i;
 i++;
}
System.out.println(summe);
```

i:	3
summe:	3

hochschule mannheim
Tracing einer while-Schleife

```
int i=1;
int summe=0;
while (i<=3){ // true
 summe += i;
 i++;
}
System.out.println(summe);
```

i:	3
summe:	3

Tracing einer while-Schleife

```
int i=1;
int summe=0;
while (i<=3){
 summe += i;
 i++;
}
System.out.println(summe);
```

i:	3
summe:	6

Tracing einer while-Schleife

```
int i=1;
int summe=0;
while (i<=3){
 summe += i;
 i++;
}
System.out.println(summe);
```

i:	4
summe:	6

Tracing einer while-Schleife

```
int i=1;
int summe=0;
while (i<=3){ // false
  summe += i;
  i++;
}
System.out.println(summe);
```

i:	4
summe:	6

Tracing einer while-Schleife

```
int i=1;
int summe=0;
while (i<=3){
  summe += i;
  i++;
}
System.out.println(summe);
```

i:	1
summe:	0

6

Arrays

Was sind Arrays?

- In einem Array kann man mehrere Variablen des gleichen Typs zusammenfassen.
- Beispiel: Koordinaten eines Punkts im Raum...
  - Mathematisch:

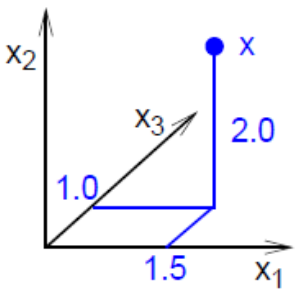
$$\vec{x} = (x_1, x_2, x_3) = (1.5, 2.0, 1.0)$$

  - In Java: Ein Array mit Elementen des Typs double...

```
double[] x=new double[3];
x[0]=1.5;
x[1]=2.0;
x[2]=1.0;
```

Alternativ:

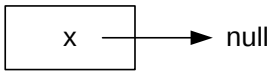
```
double[] x={1.5,2.0,1.0};
```



hochschule mannheim
Was sind Arrays?

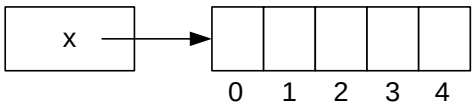
- Java-Arrays sind Referenz-Typen.
- Eine Array-Variable ist also eine Referenz:

```
double[] x;
```



- Das Array selbst muss dynamisch erzeugt werden:

```
x=new double[5];
```



hochschule mannheim
Kennen Sie immer Ihre Grenzen!

- Deklaration einer Referenzvariablen auf ein Array:
  `char[] charArray;`
  `String[] stringArray;`
- Arrays werden mit new angelegt und haben eine fixe Größe:
  `charArray = new String[10];`
- Arrays können nachträglich nicht mehr in der Größe verändert werden.
- Der Index eines Arrays beginnt bei 0.
- Arrays besitzen eine nur-lesen Eigenschaft **`length`**, um die Anzahl der Elemente auszulesen.
- Bei einem Indexüberlauf wird eine Ausnahme, eine **`IndexOutOfBoundsException`** geworfen.

Kennen Sie immer Ihre Grenzen!

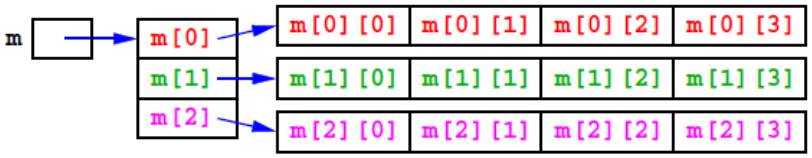
```
int[] array = new int[5];
for (int i = 1; i < 6; i++) {
  array[i] = i;
}
System.out.println("Fertig!");
```


Beispiele zum Umgang mit Arrays

```
double[] x={1.5,2.0,1.0};
double[] y=x; // Erzeugt neue Referenzvariable y,
              // die auf dasselbe Array zeigt wie x
y[2] = 4.0;   // Dies ändert auch den Wert von x[2]!

// Nur so wird z eine echte Kopie von x:
double[] z = new double[x.length];
for (int i=0;i<x.length;i++)
  z[i]=x[i];
```

Mehrdimensionale Arrays

- Die Elemente eines Arrays können auch Referenzen auf Arrays enthalten
  `int[][] m=new int[3][4]`:



- Deklaration der Referenzvariable:

```
int[][] m;     // 2-dimensionales Array / Matrix
int[][][] m3D; // 3-dimensionales Array / Matrix
```


Mehrdimensionale Arrays

- Erzeugung des Arrays:

```
m = new int[3][4];  // m = ⎛ 0 0 0 0 ⎞
                        ⎜ 0 0 0 0 ⎟
                        ⎝ 0 0 0 0 ⎠
```

```
m[2] = new int[2];
m[2][1] = 7;        // m = ⎛ 0 0 0 0 ⎞
                        ⎜ 0 0 0 0 ⎟
                        ⎝ 0 7     ⎠
```

- **`m[2]`** ist eine Referenz auf ein Array von **`int`**-Zahlen.
- **`m[2]`** hat den Typ **`int[]`**
- **`m.length`** ist 3
- Die Länge der Zeilen kann variieren:
  `m[0].length` ist 4, aber **`m[2].length`** ist 2

Mehrdimensionale Arrays

- **`int[][] m=new int[3][4]`**

  ist gleichbedeutend mit

```
// Legt ein Array für 3 Zeilenverweise an:
int[][] m=new int[3][];
// Initialisiert die Zeilenverweise:
for (int i=0; i<3; i++)
  m[i]=new int[4];
```

- aber **`m=new int[][4];`**
  führt zur Fehlermeldung, da
  - die Referenzen auf die Zeilen nirgends abgespeichert werden können und da
  - die Anzahl der zu erzeugenden Zeilen unbekannt ist.

Referenzen und null

- Soll die **`if`**-Verzweigung testen, ob eine Referenzvariable ref ein konkretes Array oder ein Objekt referenziert, so ist die Variable mit **`null`** zu vergleichen:

```
int[] x=null;
// Deklarieren eines Arrays?...
if(x!=null) {
  // ...
}
```

- Wieso ist

  `if ((x!=null) && (x.length>2))...`

  sinnvoll,

  `if ((x!=null) & (x.length>2))...`

  aber dumm?

Funktionen / Methoden

Welche Arten von Variablen gibt es?

In Java werden **3 Arten von Variablen** unterscheiden:

- Lokale Variablen in Methoden oder generell in Blöcken
  dienen der temporären Speicherung von Daten, während diese Methode ausgeführt wird.
- Formale Parameter in Methoden
  speichern die Werte der aktuellen Parameter, die beim Aufruf einer Methode übergeben werden.

Wieso wird Funktionalität in viele Methoden aufgeteilt?

- Der Grundsatz teile und herrsche findet in vielen Teilgebieten der Informatik Anwendung.
- Dabei wird ausgenutzt, dass bei vielen Problemen der Lösungsaufwand sinkt, wenn man das Problem in kleinere Teilprobleme zerlegt.
- Diese Teile werden wie eigenständige Probleme parallel oder sequenziell gelöst, bis sie auf triviale Lösungen zurückgeführt sind.
- In einem Quellcode äußert sich dies in Methoden, die andere Methoden zumeist auf niederer Ebene aufrufen. Stichwörter:
  - prozedurale Programmierung
  - strukturierte Programmierung

hochschule mannheim

Wieso wird Funktionalität in viele Methoden aufgeteilt?

- In der Objektorientierung bietet ein Objekt viele Dienste an.
- Dies entspricht der SOA-Denkweise (Service Oriented Architecture).
- Diese Dienste eines Objektes sind Funktions- oder Prozedur-Aufrufe.
- Sie werden in der Objektorientierung als
  - Methoden oder als
  - Operationen

  bezeichnet:

Deklaration:

```
int f(int u,int v){
  return (u*v);
}
```

Aufruf:

```
int y=f(a,b);
```

(2 und 3 sind die Werte von a und b; 6 ist der Rückgabewert)

- u und v sind dabei formale Parameter der Methode f.
- 2 und 3 sind die aktuellen Parameter der Methode während der Laufzeit.
- 6 ist der aktuelle Rückgabewert der Methode f, der in der Variablen y gespeichert wird.

hochschule mannheim

Ablauf eines Methodenaufrufs bei primitiven Datentypen

1. Die Ausdrücke in der Parameterliste werden ausgewertet.
2. Die Werte der aktuellen Parameter werden an die formalen Parameter der Methode **als Kopie zugewiesen** (call by value).
3. Der Methodenrumpf wird ausgeführt
   - bis zum Ende (nur bei void-Methoden erlaubt)
   - oder bis zur Anweisung **`return (<Ausdruck>).`**
   - Der Wert von **`<Ausdruck>`** bestimmt den Wert des Methodenaufrufs.
4. Die Ausführung der aufrufenden Methode wird nach dem Methodenaufruf fortgesetzt.

hochschule mannheim

Übergabe von Werten bei primitiven Datentypen

```
private static int f(int b, int a) {
  a=2*b+a*a;
  return a+1;
}

public static void main(String[] args) {
  int x=10;
  int a=12;

  a=f(a,++x)-f(a,a+3);

  System.out.println("a:"+a);
  System.out.println("x:"+x);
}
```

f:

main:
x=10

Übergabe von Werten bei primitiven Datentypen

```
private static int f(int b, int a) {
  a=2*b+a*a;
  return a+1;
}

public static void main(String[] args) {
  int x=10;
  int a=12;

  a=f(a,++x)-f(a,a+3);

  System.out.println("a:"+a);
  System.out.println("x:"+x);
}
```

f:

main:
x=10
a=12

Übergabe von Werten bei primitiven Datentypen

```
private static int f(int b, int a) {
  a=2*b+a*a;
  return a+1;
}

public static void main(String[] args) {
  int x=10;
  int a=12;

  a=f(a,++x)-f(a,a+3);

  System.out.println("a:"+a);
  System.out.println("x:"+x);
}
```

f:
b=12
a=11

main:
x=11
a=12

Übergabe von Werten bei primitiven Datentypen

```
private static int f(int b, int a) {
  a=2*b+a*a;
  return a+1;
}

public static void main(String[] args) {
  int x=10;
  int a=12;

  a=f(a,++x)-f(a,a+3);

  System.out.println("a:"+a);
  System.out.println("x:"+x);
}
```

f:
b=12
a=145

main:
x=11
a=12

Übergabe von Werten bei primitiven Datentypen

```
private static int f(int b, int a) {
  a=2*b+a*a;
  return a+1;
}

public static void main(String[] args) {
  int x=10;
  int a=12;

  a=f(a,++x)-f(a,a+3);

  System.out.println("a:"+a);
  System.out.println("x:"+x);
}
```

f:
b=12
a=145
return=146

main:
x=11
a=12

hochschule mannheim

Übergabe von Werten bei primitiven Datentypen

```
private static int f(int b, int a) {
  a=2*b+a*a;
  return a+1;
}

public static void main(String[] args) {
  int x=10;
  int a=12;

  a=f(a,++x)-f(a,a+3);

  System.out.println("a:"+a);
  System.out.println("x:"+x);
}
```

146

f:
b=12
a=15

main:
x=11
a=12

hochschule mannheim

Übergabe von Werten bei primitiven Datentypen

```
private static int f(int b, int a) {
  a=2*b+a*a;
  return a+1;
}

public static void main(String[] args) {
  int x=10;
  int a=12;

  a=f(a,++x)-f(a,a+3);

  System.out.println("a:"+a);
  System.out.println("x:"+x);
}
```

146

f:
b=12
a=249

main:
x=11
a=12

Übergabe von Werten bei primitiven Datentypen

```
private static int f(int b, int a) {
   a=2*b+a*a;
   return a+1;
}

public static void main(String[] args) {
   int x=10;
   int a=12;

   a=f(a,++x)-f(a,a+3);

   System.out.println("a:"+a);
   System.out.println("x:"+x);
}
```

146

f:
b=12
a=249
return=250

main:
x=11
a=12

Übergabe von Werten bei primitiven Datentypen

```
private static int f(int b, int a) {
   a=2*b+a*a;
   return a+1;
}

public static void main(String[] args) {
   int x=10;
   int a=12;

   a=f(a,++x)-f(a,a+3);

   System.out.println("a:"+a);
   System.out.println("x:"+x);
}
```

146 250

f:

main:
x=11
a=-104

Übergabe von Werten bei primitiven Datentypen

```
private static int f(int b, int a) {
  a=2*b+a*a;
  return a+1;
}

public static void main(String[] args) {
  int x=10;
  int a=12;

  a=f(a,++x)-f(a,a+3);

  System.out.println("a:"+a);
  System.out.println("x:"+x);
}
```

f:

main:
x=11
a=-104

Konsole:
a:-104

Übergabe von Werten bei primitiven Datentypen

```
private static int f(int b, int a) {
  a=2*b+a*a;
  return a+1;
}

public static void main(String[] args) {
  int x=10;
  int a=12;

  a=f(a,++x)-f(a,a+3);

  System.out.println("a:"+a);
  System.out.println("x:"+x);
}
```

f:

main:
x=11
a=-104

Konsole:
a:-104
x:11

hochschule mannheim

Übergabe von Werten bei primitiven Datentypen

```
private static void swap(int x, int y) {
   int tmp=x;
   x=y;
   y=tmp;
}

public static void main(String[] args) {
   int a,b;
   a=1;b=2;
   swap(a,b);
   System.out.println("a:"+a);
   System.out.println("b:"+b);
}
```

x=1

y=2

a=1

b=2

swap:
x=1
y=2

main:
a=1
b=2

hochschule mannheim

Übergabe von Werten bei primitiven Datentypen

```
private static void swap(int x, int y) {
   int tmp=x;
   x=y;
   y=tmp;
}

public static void main(String[] args) {
   int a,b;
   a=1;b=2;
   swap(a,b);
   System.out.println("a:"+a);
   System.out.println("b:"+b);
}
```

x=1

tmp=1

y=2

a=1

b=2

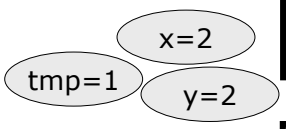
swap:
x=1
y=2
tmp=1

main:
a=1
b=2

hochschule mannheim

Übergabe von Werten bei primitiven Datentypen

```
private static void swap(int x, int y) {
   int tmp=x;
   x=y;
   y=tmp;
}

public static void main(String[] args) {
   int a,b;
   a=1;b=2;
   swap(a,b);
   System.out.println("a:"+a);
   System.out.println("b:"+b);
}
```



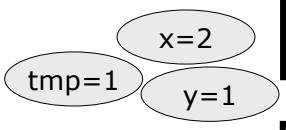
swap:
x=2
y=2
tmp=1

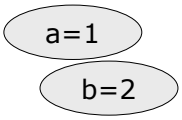
main:
a=1
b=2

hochschule mannheim

Übergabe von Werten bei primitiven Datentypen

```
private static void swap(int x, int y) {
   int tmp=x;
   x=y;
   y=tmp;
}

public static void main(String[] args) {
   int a,b;
   a=1;b=2;
   swap(a,b);
   System.out.println("a:"+a);
   System.out.println("b:"+b);
}
```





swap:
x=2
y=2
tmp=1

main:
a=1
b=2

Übergabe von Werten bei primitiven Datentypen

```
private static void swap(int x, int y) {
   int tmp=x;
   x=y;
   y=tmp;
}

public static void main(String[] args) {
   int a,b;
   a=1;b=2;
   swap(a,b);
   System.out.println("a:"+a);
   System.out.println("b:"+b);
}
```

a=1

b=2

swap:

main:
a=1
b=2

Konsole:
a:1

Übergabe von Werten bei primitiven Datentypen

```
private static void swap(int x, int y) {
   int tmp=x;
   x=y;
   y=tmp;
}

public static void main(String[] args) {
   int a,b;
   a=1;b=2;
   swap(a,b);
   System.out.println("a:"+a);
   System.out.println("b:"+b);
}
```

a=1

b=2

swap:

main:
a=1
b=2

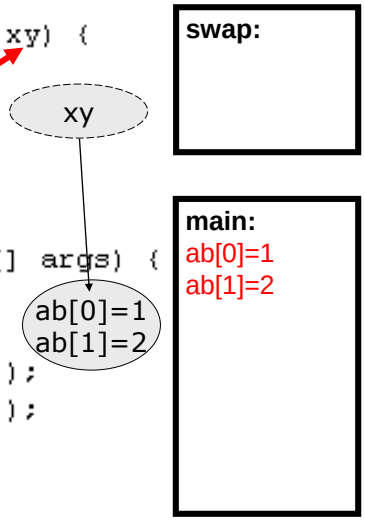
Konsole:
a:1
b:2

Übergabe von Referenzen bei Arrays & Obejekten

- Auch Arrays und Objekte können Parameter von Methoden sein.
- Übergeben werden dabei jedoch nicht die ganzen Arrays/Objekte, sondern nur die Werte der Speicherstellen, die auf die Arrays/Objekte verweisen, also die Referenzen.
- Die aufgerufene Methode kann dabei die übergebenen Ziele, also die Objekte/Arrays verändern!
  - Damit lassen sich auch Ein-/Ausgabe-Parameter realisieren
  - Ein unerwartetes Verändern übergebener Objekte sollte aber vermieden werden!
- Analog dazu können die Werte der Speicherstellen natürlich auch als Ergebnis einer Methode auftreten.

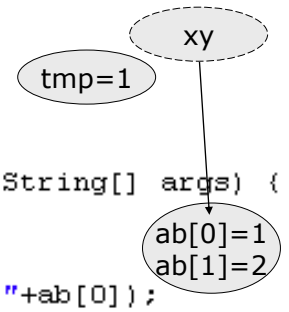
Übergabe von Referenzen bei Arrays & Obejekten

```
private static void swap(int[] xy) {
  int tmp=xy[0];
  xy[0]=xy[1];
  xy[1]=tmp;
}

public static void main(String[] args) {
  int[] ab={1,2};
  swap(ab);
  System.out.println("a:"+ab[0]);
  System.out.println("b:"+ab[1]);
}
```



Übergabe von Referenzen bei Arrays & Obejekten

```
private static void swap(int[] xy) {
  int tmp=xy[0];
  xy[0]=xy[1];
  xy[1]=tmp;
}

public static void main(String[] args) {
  int[] ab={1,2};
  swap(ab);
  System.out.println("a:"+ab[0]);
  System.out.println("b:"+ab[1]);
}
```

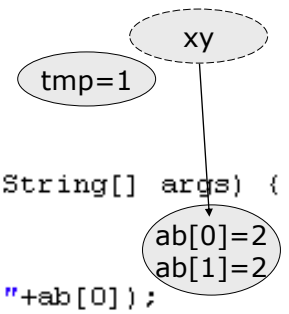


swap:
tmp=1

main:
ab[0]=1
ab[1]=2

Übergabe von Referenzen bei Arrays & Obejekten

```
private static void swap(int[] xy) {
  int tmp=xy[0];
  xy[0]=xy[1];
  xy[1]=tmp;
}

public static void main(String[] args) {
  int[] ab={1,2};
  swap(ab);
  System.out.println("a:"+ab[0]);
  System.out.println("b:"+ab[1]);
}
```



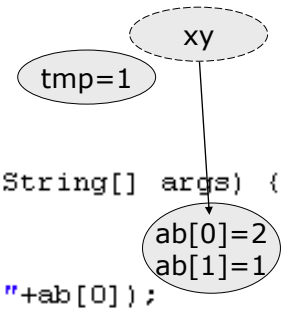
swap:
tmp=1

main:
ab[0]=2
ab[1]=2

hochschule mannheim

Übergabe von Referenzen bei Arrays & Obejekten

```
private static void swap(int[] xy) {
  int tmp=xy[0];
  xy[0]=xy[1];
  xy[1]=tmp;
}

public static void main(String[] args) {
  int[] ab={1,2};
  swap(ab);
  System.out.println("a:"+ab[0]);
  System.out.println("b:"+ab[1]);
}
```



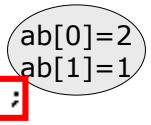
swap:
tmp=1

main:
ab[0]=2
ab[1]=1

hochschule mannheim

Übergabe von Referenzen bei Arrays & Obejekten

```
private static void swap(int[] xy) {
  int tmp=xy[0];
  xy[0]=xy[1];
  xy[1]=tmp;
}

public static void main(String[] args) {
  int[] ab={1,2};
  swap(ab);
  System.out.println("a:"+ab[0]);
  System.out.println("b:"+ab[1]);
}
```



swap:
tmp=1

main:
ab[0]=2
ab[1]=1

Konsole:
a:2

Übergabe von Referenzen bei Arrays & Obejekten

```
private static void swap(int[] xy) {
   int tmp=xy[0];
   xy[0]=xy[1];
   xy[1]=tmp;
}

public static void main(String[] args) {
   int[] ab={1,2};
   swap(ab);
   System.out.println("a:"+ab[0]);
   System.out.println("b:"+ab[1]);
}
```

ab[0]=2
ab[1]=1

swap:
tmp=1

main:
ab[0]=2
ab[1]=1

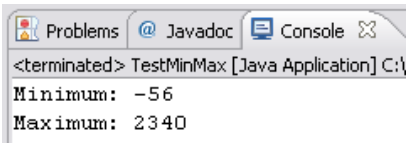
Konsole:
a:2
b:1

Arrays & Rückgabewerte

```
public static void main(String[] args) {
   int[] daten={999,4,7,227,-56,2340,-6,0,93};
   int[] ergebnis=minmax(daten);
   if (ergebnis==null){
      System.out.println("Kein Minimum und Maximum vorhanden!");
   }
   else{
      System.out.println("Minimum: "+ergebnis[0]);
      System.out.println("Maximum: "+ergebnis[1]);
   }
}
```

Arrays & Rückgabewerte

```
private static int[] minmax(int[] daten) {
  if (daten==null) return null;
  if (daten.length==0) return null;
  int min=daten[0];
  int max=daten[0];
  for (int i=1;i<daten.length;i++){
    if (daten[i]<min) min=daten[i];
    if (daten[i]>max) max=daten[i];
  }
  int[] erg=new int[2];
  erg[0]=min; erg[1]=max;
  return erg;
}
```



Eine Besonderheit bei Strings!

```
static void swap(String a, String b) {
  String temp = a;
  a = b;
  b = temp;
}

public static void main(String[] args) {
  String a = "Hallo";
  String b = "Welt";
  swap(a, b);
  System.out.println(a + " " + b);
}
```

Rekursion

Um Rekursion zu verstehen, müssen Sie zunächst Rekursion verstehen!

Was ist Rekursion?

- Als Rekursion (lat. recurrere „zurücklaufen“) bezeichnet man die Technik, eine Methode durch sich selbst zu definieren.
- Dies nennt man rekursive Definition.
- Jeder Aufruf der rekursiven Methode muss sich durch Entfalten der rekursiven Definition in endlich vielen Schritten auflösen lassen.
- Umgangssprachlich sagt man, sie darf nicht in eine Endlos-Rekursion geraten.
- Eine Endlos-Rekursion ist keine Endlos-Schleife!
- Die Rekursion ist eine von mehreren möglichen Problemlösungs-strategien, sie führt oft zu eleganten Darstellungen.
- Rekursion & Iteration sind im Wesentlichen gleichmächtige Sprachmittel.
- In ihrer Implementierung kann es Effizienz-Unterschiede geben.

Beispiel: Summierer

- Die Methode **`sum`** soll zu jeder Zahl n die Summe der ersten n Zahlen berechnen. Sie ist folgendermaßen definiert:

$$\mathrm{sum}(n) = \sum_{i=0}^{n} i$$

- Um eine gleichwertige rekursive Definition der Summenmethode zu erhalten, bestimmen wir zunächst den einfachen Fall, den Rekursionsanfang.
- Im Beispiel handelt es sich um den Methodenwert für 0:

$$\mathrm{sum}(0) = 0$$

Beispiel: Summierer

- Übrig bleibt der schwierige Fall, also hier der Methodenwert für n > 0.
- Den schwierigen Fall führen wir auf einen einfacheren Fall zurück, nämlich auf den Fall n − 1.
- Dieser einfachere Fall wird unser rekursiver Aufruf.
- Die entsprechende Vorschrift heißt Rekursions-Schritt.
- So lässt sich die Summe der ersten n Zahlen berechnen, indem man die Summe der ersten n − 1 Zahlen berechnet und dazu die Zahl n addiert:

$$\mathrm{sum}(n) = \mathrm{sum}(n-1) + n$$

Beispiel: Summierer

- Die beiden Gleichungen lassen sich zu einer rekursiven Definition der Summenmethode zusammenfassen:

$$\mathrm{sum}(n) = \begin{cases} 0 & \text{falls } n = 0 \quad \text{(Rekursionsanfang)} \\ \mathrm{sum}(n-1) + n & \text{sonst} \quad \text{(Rekursionsschritt)} \end{cases}$$

- Beispielhafter Aufruf:

$$\begin{aligned} \mathrm{sum}(3) &= \mathrm{sum}(2) + 3 && \text{(Rekursionsschritt)} \\ &= \mathrm{sum}(1) + 2 + 3 && \text{(Rekursionsschritt)} \\ &= \mathrm{sum}(0) + 1 + 2 + 3 && \text{(Rekursionsschritt)} \\ &= 0 + 1 + 2 + 3 && \text{(Rekursionsanfang)} \\ &= 6 \end{aligned}$$

- Aufruf-Kette: $\mathrm{sum}(3) \rightarrow \mathrm{sum}(2) \rightarrow \mathrm{sum}(1) \rightarrow \mathrm{sum}(0)$.

Beispiel: Fakultätsfunktion

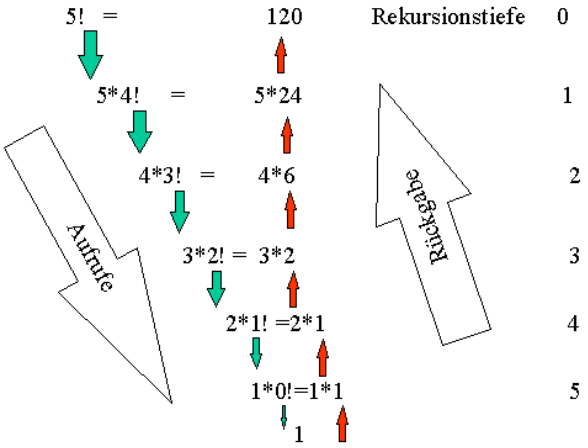
- Die Fakultät ist in der Mathematik eine Funktion, die einer natürlichen Zahl das Produkt aller natürlichen Zahlen kleiner oder gleich dieser Zahl zuordnet
- Sie wird durch ein dem Argument nachgestelltes Ausrufezeichen („!“) abgekürzt.
- Große Bedeutung hat die Fakultätsfunktion auf dem Gebiet der Kombinatorik.
- Beispiele: 3! = 3 x 2 x 1 = 6, 0! = 1
- Allgemein:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n = \prod_{k=1}^{n} k$$

Beispiel: Fakultätsfunktion

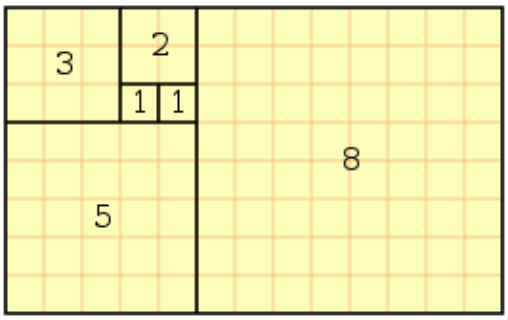
- Rekursive Definition:

$$n! = \begin{cases} n \cdot (n-1)! & n > 0 \\ 1 & n = 0 \quad \text{Rekursionsanfang} \end{cases}$$



Beispiel: Fibonacci-Folge

- Die Fibonacci-Folge ist eine unendliche Folge von Zahlen (den Fibonacci-Zahlen), bei der sich die jeweils folgende Zahl durch Addition der beiden vorherigen Zahlen ergibt:
  0, 1, 1, 2, 3, 5, 8, 13, …
- Benannt ist sie nach Leonardo Fibonacci, der damit 1202 das Wachstum einer Kaninchenpopulation beschrieb:



Beispiel: Fibonacci-Folge

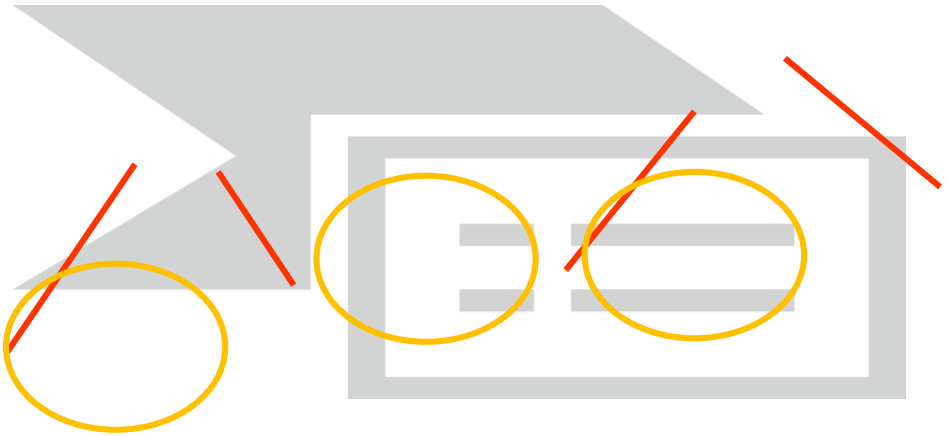
- Die Fibonacci-Funktion **`fib(n)`**, die jedem n die n-te Fibonacci-Zahl zuordnet, hat die einfachen Fälle fib(0)=0 und fib(1)=1.
- Sie genügt der Rekursionsgleichung für n>1:

$$\mathrm{fib}(n) = \mathrm{fib}(n-1) + \mathrm{fib}(n-2)$$

- So ergibt sich die rekursive Definition:

$$\mathrm{fib}(n) = \begin{cases} 0 & \text{falls } n = 0 \quad \text{(Rekursionsanfang)} \\ 1 & \text{falls } n = 1 \quad \text{(Rekursionsanfang)} \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{sonst} \quad \text{(Rekursionsschritt)} \end{cases}$$

Beispiel: Fibonacci-Folge



Zur Komplexität von Algorithmen

hochschule mannheim

Was ist die Aufgabe der Komplexitätstheorie?

- Die Komplexitätstheorie als Teilgebiet der Theoretischen Informatik befasst sich mit der Komplexität von algorithmisch behandelbaren Problemen auf verschiedenen formalen Rechnermodellen.
- Die Komplexitätstheorie unterscheidet sich von der Berechenbarkeits-theorie, die sich mit der Frage beschäftigt, welche Probleme prinzipiell algorithmisch gelöst werden können.
- Demgegenüber besteht das Forschungsziel der Komplexitätstheorie darin, die Menge aller lösbaren Probleme zu klassifizieren.
- Insbesondere versucht man, die Menge der effizient lösbaren Probleme von der Menge der inhärent schwierigen Probleme abzugrenzen.

hochschule mannheim

Effizient lösbare Probleme

- Grundrechenarten
- Potenzieren
- Wurzel ziehen
- Gleichungssysteme lösen
- Berechnung der kürzesten Route in einem Graph (Dijkstra-Algorithmus) zur Routenberechnung

Bitte machen Sie sich mit diesen Problemstellungen vertraut!

Inhärent schwierigen Probleme

- Stundenplanung mit Scheduling
- Graphenfärbung
- Problem des Handlungsreisenden als Routenberechnung über mehrere Stationen
- Das Rucksackproblem zur optimalen Kapazitätsausnutzung

Bitte machen Sie sich mit diesen Problemstellungen vertraut!

Was ist Komplexität?

- In der theoretischen Informatik beschreibt die Komplexität eine Abschätzung des Ressourcenaufwandes zur algorithmischen Behandlung eines Problems.
- Die Komplexität ist dann groß, wenn einerseits zu viele und andererseits in der Summe zu komplizierte Details zu behandeln sind.
- Die Komplexität von Algorithmen wird in deren Ressourcenverbrauch gemessen, meist die Anzahl der benötigten Rechenschritte (Zeitkomplexität) oder Speicherplatzbedarf (Platzkomplexität).
- Meist interessiert es, wie der Ressourcenbedarf wächst, wenn ein paar Daten mehr zu verarbeiten sind.

Komplexität und O-Notation

Definition & Sinn der O-Notation

- Die Landau-Notation wird verwendet, um das asymptotische Verhalten bei Annäherung an einen endlichen oder unendlichen Grenzwert zu beschreiben.
- Das große O wird verwendet, um eine maximale Größenordnung anzugeben.
- Grund: Oft ist es sehr aufwändig oder ganz unmöglich, eine genaue Funktion anzugeben, die allgemein zu jeder beliebigen Eingabe für ein Problem den zugehörigen Aufwand an Ressourcen angibt.
- Daher beschränkt man sich häufig darauf, eine obere Schranke für das asymptotische Verhalten anzugeben.

Die Landau-Symbole

- Landau-Symbole werden bei der Analyse von Algorithmen verwendet und geben ein Maß für die Anzahl der Elemantarschritte in Abhängigkeit von der Anzahl der Eingangsvariablen n an.
- Sie werden verwendet, um verschiedene Probleme danach zu vergleichen, wie „schwierig“ sie zu lösen sind.
- Man sagt „schwere Probleme“ wachsen exponentiell mit der Größe der Eingangsvariablen n.

Die Landau-Symbole

```
public static void main(String[] args) {
  int[] data={2,4,56,23,6,24,-4,346,3465,-2323};
  System.out.println(data[6]);
}
```

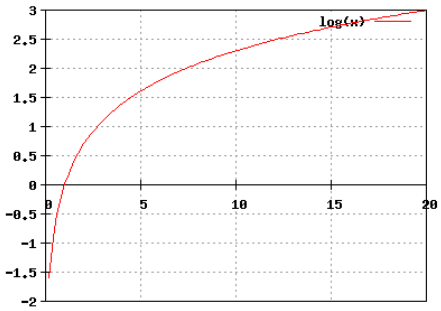
➔ Laufzeitkomplexität O(1)

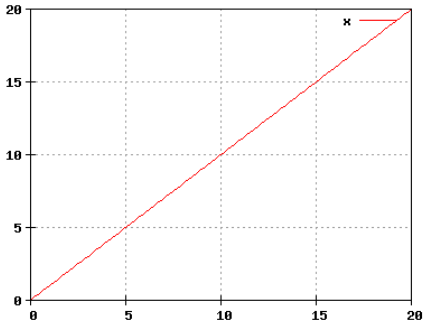
Die Landau-Symbole

```
int[] data={2,4,56,23,6,24,-4,346,3465,-2323};
boolean gefunden=false;
for(int i=0;i<data.length;i++){
  if (data[i]==3465){
    gefunden=true;
    break; // beendet die for-Schleife vorzeitig
  }
}
System.out.println(gefunden);
```

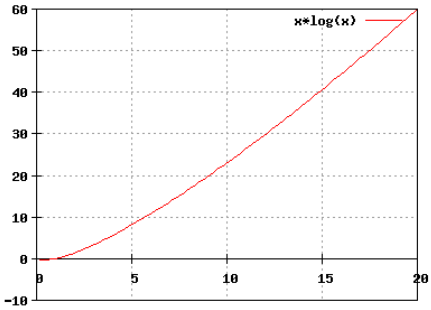
➔ Laufzeitkomplexität O(n) im worst-case,
Laufzeitkomplexität O(1) im best-case

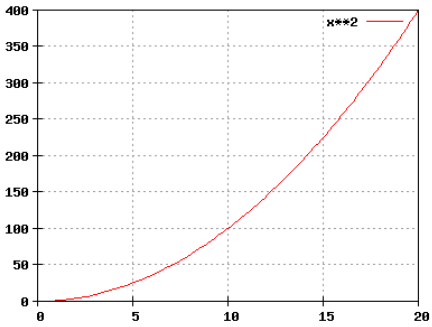
Funktionen der Komplexität



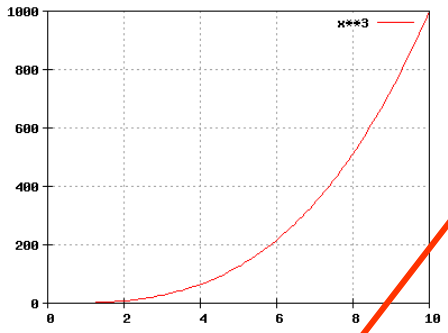


Funktionen der Komplexität



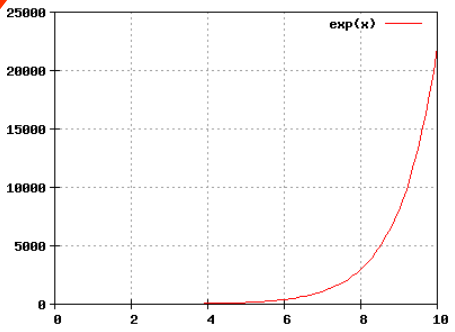


Funktionen der Komplexität



einfach

schwer



Funktionen der Komplexität

k = Konstante

Laufzeit	O-Notation	Beispiel
konstant	$O(1)$	Berechnung geschlossener Funktionen
logarithmisch	$O(\log n)$	Binäre Suche
linear	$O(n)$	Suche in unsortiertem Feld
n log n	$O(n \log n)$	Sortieren (intelligent)
quadratisch	$O(n^2)$	Sortieren „naiv“
kubisch	$O(n^3)$	Arbeit in Volumen der Kantenlänge n
polynominal	$O(n^k)$	
exponentiell	$O(k^n)$	Hanoi, Erfüllbarkeitsproblem der Aussagenlogik
faktoriell	$O(n!)$	Problem des Handlungsreisenden

Funktionen der Komplexität

n =	10	20	30	40	50	60
O(1)	1	1	1	1	1	1
O(log n)	3,3	4,3	4,9	5,3	5,6	5,9
O(n)	10	20	30	40	50	60
O(n log n)	33,2	86,4	147,2	212,9	282,2	354,4
O(n²)	100	400	900	1600	2500	3600
O(n³)	1000	8000	27000	64000	125000	216000
O(k^n) 2^n	1024	1048576	1073741824	1,09951E+12	1,1259E+15	1,15292E+18
O(k^n) 3^n	59049	3486784401	2,05891E+14	1,21577E+19	7,17898E+23	4,23912E+28
O(n!)	3628800	2,4329E+18	2,65253E+32	8,15915E+47	3,04141E+64	8,32099E+81

Annahme: 1 Schritt = 1 µs = 0,000001 s
→ 8,32099E+81 µs = 10^{66} Jahre