Einführung in die Programmiersprache Java

unter Verwendung der Entwicklungsumgebung Eclipse

Dominik Köppl

Physikalische Fakultät
Universität Regensburg

Wintersemester 2011/12

Copyright

Skript zur Java-Programmierung

Diese Foliensammlung ist Bestandteil der Vorlesung “Einführung in die Programmiersprache Java” der physikalischen Fakultät der Universität Regensburg. Diese Sammlung erhebt in keinster Weise Anspruch auf Vollständigkeit und Fehlerfreiheit.
Korrekturvorschläge und Anregungen sind jederzeit erwünscht.

[1]

Die Unterlagen sind unter http://homepages.uni-regensburg.de/~kod15930/?s=java online verfügbar.

[1] Dieses Dokument wird unter der Creative Commons Lizenz BY-SA angeboten

Überblick I

Überblick II

Überblick III

Überblick IV

Was ist Java?

- Java ist eine indonesische Insel mit $132,187\ \mathrm{km}^2$ Fläche.
- Die in der USA vorherrschende umgangssprachliche Bezeichnung für eine “Tasse Kaffee”.
- Eine von SUN entwickelte Programmiersprache

Warum Java?

Warum haben sich die Mitarbeiter von SUN die Mühe gemacht, eine weitere Programmiersprache zu entwickelt?

- Es gab bis dahin noch keine Programmiersprache, mit der man plattformunabhängige graphische Programme schreiben konnte.
- Die Programmiersprachen C und C++ sind schwierig und fehleranfällig.
- Keine Programmiersprache hat die Entwicklung graphischer Benutzerelemente (**GUI**) in ihre Kernbibliothek so integriert wie Java.

Literatur

Zur Vertiefung der Materie kann ich folgende frei verfügbare Arbeiten/Seiten empfehlen:

- Script von Mario Jeckle - für Programmierer von C/C++ zu empfehlen
- Einführung in Java von Hubert Partl - für Anfänger
- Thinking in Java von Bruce Eckel - großes Nachschlagewerk

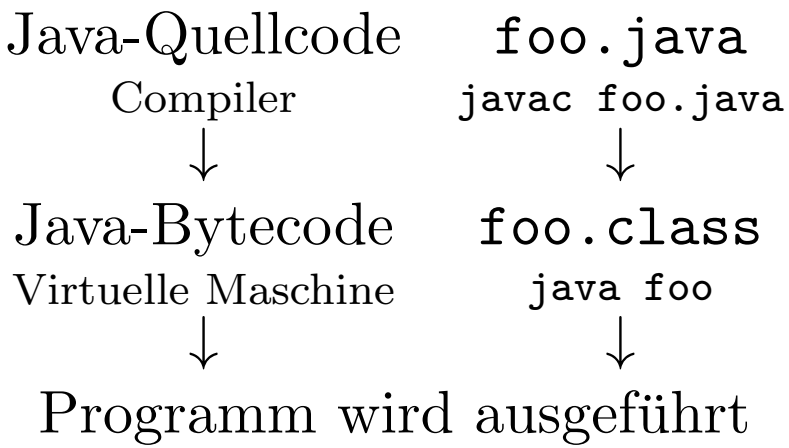
Programmierparadigmen

Durch welche Kernmerkmale ist Java charakterisiert?

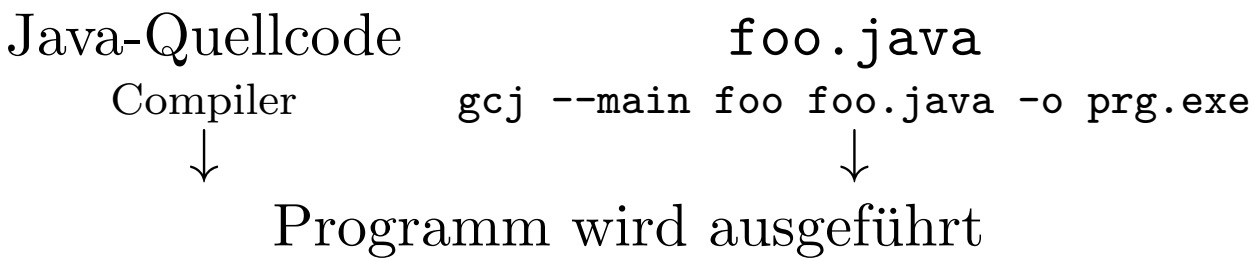
- Objektorientierung wie in C++
- Bytecoding
- vollständiger Unicode-Support

Bytecoding und Compilieren

Bytecoding ist die Verknüpfung von Compiler und Interpreter:



Java kann aber auch wie andere Programmiersprachen in Maschinencode übersetzt werden:



Compile once - run everywhere

Ist ein Java-Quellcode einmal in Bytecode kompiliert, so kann man den Java-Bytecode auf jeder Plattform laufen lassen, auf der eine Virtuelle Java Maschine (**Java VM**) installiert ist! Java kann man unter anderen auf Windows, Mac, Linux, BSD, Solaris, Android und vielen Smartphonesbetriebssystemen vorfinden. Man muss also für alle Betriebssyteme seinen Quellcode nur einmal kompilieren.

Kommentare und Schreibstil

In Java ist es zunächst einmal egal, wieviele Leerzeichen, Tabulaturen und Leerzeilen man schreibt. Man sollte sich aber einen sauberen Programmierstil angewöhnen, also den Code so gut wie möglich formatieren, sodass man sehr schnell des Geschriebene überblicken kann. Neben den freien Gestaltungsmöglichkeiten erlaubt es Java auch, Kommentare in den Quellcode zu schreiben.

- Einzeilige Kommentare können mitten in einer Zeile mit den Zeichen '//' initialisiert und gelten bis zum Zeilenumbruch.
- Ein mehrzeiliges Kommentar kann dadurch verwirklicht werden, dass man diesen mit '/*' beginnt und mit '*/' endet.

```
1 int foo = 4; // Hier beginnt das Kommentar
2 // Noch ein Einzeiler
3 double d; /* Ich bin
4 ein
5 Mehrzeiliges Kommentar */
```

Lauter Einsen und Nullen?

Leider ist es dem Rechner nicht gestattet, mit unserem Dezimalsystem intern Rechnungen durchzuführen: Er kennt nur den Status 0 (aus) und 1 (an), d.h. jeder Rechner muss Dezimalzahlen in das **Binärsystem** umwandeln können, und vice versa. Wenn wir uns zu einer Dezimalzahl ein Stellenwertsystem zeichnen (wie in der Grundschule), dann erkennen wir schnell, dass die Grundeinheit unseres Systems die "Einer", "Zehner" , "Hunderter" , etc. sind. Erstellt man ein Stellenwertsystem für Binärzahlen, bezeichnet man die "Einer" als **Bit** und fasst acht Bits als **Byte** auf.

Umrechnen: Vom Dezimalsystem ins Binärsystem

Die Methode ist schon seit der Grundschule bekannt: Division mit Rest! Man dividiert rekursiv die Dezimalzahl solange gegen die 2, bis kein Divionswert mehr vorhanden ist, und schreibt anschließend die Divionsreste in umgekehrter Reihenfolge auf.

Beispiel Nehmen wir die Zahl 42, so ergibt wegen der Rechnung

$$\begin{aligned} 42 &: 2 = 21 \quad \text{Rest } 0 \\ 21 &: 2 = 10 \quad \text{Rest } 1 \\ 10 &: 2 = 5 \quad \text{Rest } 0 \\ 5 &: 2 = 2 \quad \text{Rest } 1 \\ 2 &: 2 = 1 \quad \text{Rest } 0 \\ 1 &: 2 = 0 \quad \text{Rest } 1 \end{aligned}$$

die Binärzahl 101010.

Umrechnen: Vom Binärsystem ins Dezimalsystem

Hat man eine Binärzahl, so schreibt man diese vertikal hin und zeichnet daneben in absteigender Reihenfolge die Potenzen von 2. Man nimmt nun das Produkt von den beiden Spalten und summiert am Schluss über die Zeilen.

Beispiel Wir haben die Zahl 101010. Zunächst schreibt man die Tabelle

$$\begin{array}{ll} 1 & 2^5 \\ 0 & 2^4 \\ 1 & 2^3 \\ 0 & 2^2 \\ 1 & 2^1 \\ 0 & 2^0 \end{array}$$

und erhält anschließend die Summe $2^5 + 2^3 + 2^1 = 42$.

Variable

Kernstück jeder imperativen Programmiersprache ist der Schreib- und Lesemechanismus von Speichereinheiten. In Java kann man aus Sicherheitsgründen nicht den Hauptspeicher direkt adressieren (so wie in C mit Zeiger). Dafür kann man mit Variablen und Referenzen arbeiten.

Was sind Variablen? Variablen sind Bezeichner für bestimmte Speicherstellen, auf denen man lesend und (oft) schreibend zugreifen kann. Eine Variable ist gekennzeichnet durch:

- Modifizierer
- Namensbezeichner
- Typ
- Wert
- Sichtbarkeit und Lebensdauer

Variablen im Speicher

Da wir im Speicher nur Bytes ansprechen können, werden immer acht Bits zu einem Byte zusammengefasst, d.h. die kleinste Speichereinheit sind 8 Bits. Wir nehmen an, dass die Bytes nun der Reihenfolge nach gespeichert werden (was im Allgemeinen nicht der Fall ist) und betrachten eine hypothetische Programmiersprache, die Ganzzahlen in n-bit großen Speicherplätzen abspeichert. Der Datentyp `int_n` soll also eine Binärzahl mit n-Stellen beinhalten. Mit `int16_t x = 4;` weisen wir einer Variablen mit Namen x den Wert 4 zu, wobei x hier eine Binärzahl mit 16-Stellen sein soll. Nun sind 16-bits genau zwei Bytes, s.d. wir hier das erste Byte als das höherwertige Byte und das andere als das niederwertige Byte bezeichnen. Betrachten wir die einzelnen Bytes wieder als Dezimalzahlen, so sehen wir, dass das höherwertige Byte 0 ist, während das niederwertige Byte 4 ist.

Der binäre Tachometer

Probleme bereitet bei dieser Anschauung das Vorzeichenbit: In Java werden Ganzzahlen im sogenannten *zweier Komplement* abgespeichert. Man stelle sich einen binären Tachometer mit n Ziffern vor - er läuft also von $\underbrace{000\ldots0}_{n-mal}$ bis $\underbrace{111\ldots1}_{n-mal}$. Nun interpretiert man die erste Hälfte des Wertebereichs (also die kleineren Zahlen) als positive Zahlen und zweite Hälfte als negative Zahlen.

Speicher	Tachometer	`int`n`_t`	`int32_t`
`1111111...1`	$2^n - 1$	-1	-1
$\vdots$	$\vdots$	$\vdots$	$\vdots$
`1000000...0`	2^{n-1}	-2^{n-1}	-2^{+31}
`0111111...1`	$2^{n-1} - 1$	$+2^{n-1} - 1$	$+2^{+31} - 1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
`0000000...0`	0	0	0

Zum Tachometer...

Das ganze nutzt natürlich den Schönheitsfehler des Tachometers aus, dass $\underbrace{111...1}_{n-mal} + 1 = \underbrace{000...0}_{n-mal}$, d.h. $-1 + 1 = 0$ ist. Leider ist in dieser Interpretation $1000000...0 - 1 = 0111111...1$, d.h. $-2^{+31} - 1 = +2^{+31} - 1$. Dieser unvermeidbare Fehler wird *Overflow* genannt.

Das zweier Komplement

Um nun eine natürliche Zahl $n \in \mathbf{N}$ zu negieren, muss man folgende Schritte durchführen:

1. Stelle die Zahl als Binärzahl dar $\mapsto$ Binärmuster
2. Drehe alle Bits um
3. Addiere 1

Beispiel zum zweier Komplement

Beispiel Betrachten wir den fiktiven Datentyp *nibble*[2].

1. In der Darstellung eines Nibbles hat die Zahl 3 das Bitmuster 0011
2. Drehen wir alle Bits um, so erhalten wir das Bitmuster 1100
3. Addieren wir 1 zu diesem Bitmuster, erhalten wir das Ergebnis 1101

Tatsächlich handelt es sich bei der Binärzahl 1101 um die Dezimalzahl -3, da

$$\begin{array}{rr} & 0011 \\ + & 1101 \\ \hline 1 & 0000 \end{array}$$

[2] Ein Nibble ist ein halbes Byte, hat also vier Bits

Buchstaben im Rechner?

Auch Buchstaben werden am Rechner durch Zahlen interpretiert.

- Seit 1963 gab es einen Standard, der bis heute festlegt, was die Zahlen von 0 bis 127 für Zeichen repräsentieren: Der **ASCII**[3]. Wer sich den ASCII-Standard genauer anschaut, wird erkennen, dass darin nur Sonderzeichen und das lateinische Alphabet festgehalten sind. Dort ist also kein Platz für deutsche Umlaute![4]
- Seit 1991 gibt es nun ein Consortium, welches es sich zur Aufgabe gemacht hat, für sämtliche noch lebende Alphabete eine Übersetzungstabelle in Zahlen zu schaffen - die **Unicode**-Tabelle.

[3] American Standard Code for Information Interchange

[4] Das ist auch Grund, warum man in C nur mit den reinen lateinischen Alphabet schreiben darf.

Die ASCII-Tabelle

0	NUL	19	DC3	38	&	57	9	76	L	95	_	114	r
1	SOH	20	DC4	39	’	58	:	77	M	96	‘	115	s
2	STX	21	NAK	40	(	59	;	78	N	97	a	116	t
3	ETX	22	SYN	41	)	60	<	79	O	98	b	117	u
4	EOT	23	ETB	42	*	61	=	80	P	99	c	118	v
5	ENQ	24	CAN	43	+	62	>	81	Q	100	d	119	w
6	ACK	25	EM	44	,	63	?	82	R	101	e	120	x
7	BEL	26	SUB	45	-	64	@	83	S	102	f	121	y
8	BS	27	ESC	46	.	65	A	84	T	103	g	122	z
9	HT	28	FS	47	/	66	B	85	U	104	h	123	{
10	LF	29	GS	48	0	67	C	86	V	105	i	124	|
11	VT	30	RS	49	1	68	D	87	W	106	j	125	}
12	FF	31	US	50	2	69	E	88	X	107	k	126	~
13	CR	32		51	3	70	F	89	Y	108	l	127	DEL
14	SO	33	!	52	4	71	G	90	Z	109	m		
15	SI	34	"	53	5	72	H	91	[	110	n		
16	DLE	35	#	54	6	73	I	92	\	111	o		
17	DC1	36	$	55	7	74	J	93	]	112	p		
18	DC2	37	%	56	8	75	K	94	^	113	q		

Unicode

Die Tabelle ist nun schon so groß, dass man eine 32-bit Zahl braucht, um alle Zeichen darstellen zu können. Wenn man das mit der 7-bit Zahl vergleicht, die für den ASCII-Standard genügte, dann wird man leicht feststellen, dass dadurch jeder Text viermal so viel Speicher verbraucht! Deswegen gibt es Codierungsverfahren, die sich unterschiedlich gut/schlecht auswirken, wenn man reinen lateinschen Text mit exotischen Zeichen mischt:

`utf-8`	8-bit großes Zeichen falls im ASCII
`utf-16`	16-bit großes Zeichen falls im ASCII
`utf-32`	immer 32-bit groß

Während man im `utf-32` sofort feststellen kann, um wieviel Zeichen es sich handelt, muss man im `utf-8` und `utf-16` alle Zeichen durchwandern. Auf den Linux-Dateisystemen hat sich `utf-8` eingebürgert, in Java wird `utf-16` verwendet.

Bezeichner

Unter einem **Bezeichner** versteht man die vom Programmierer selbst vergebene Namen. Man kann Klassen, Variablen, Konstanten und Dateien Namen geben. Diese Namensvergabe muss aber bestimmten Regeln gehorchen:

- Nicht erlaubt sind: Minuszeichen, Leerzeichen, Schrägstriche
- Erlaubt ist als erstes Zeichen keine Zahl
- Ansonsten sind alle Buchstaben des Unicode-Standard erlaubt!

Bemerkung Es sind also sowohl die deutschen Umlaute als auch exotische Zeichen wie japanische Kanji zugelassen.
`muzukashii.java`

Elementare Datentypen

Name	Bytes	Wertebereich
Ganzzahlen		
`byte`	1	[`Byte.MIN_VALUE` ; `Byte.MAX_VALUE`]
`short`	2	[`Short.MIN_VALUE` ; `Short.MAX_VALUE`]
`int`	4	[`Integer.MIN_VALUE`; `Integer.MAX_VALUE`]
`long`	8	[`Long.MIN_VALUE` ; `Long.MAX_VALUE`]
Gleitpunktzahlen		positiver Wertebereich
`float`	4	[`Float.MIN_VALUE`; `Float.MAX_VALUE`]
`double`	8	[`Double.MIN_VALUE`; `Double.MAX_VALUE`]
besondere Datentypen		
`boolean`	1	`true`, `false`
`char`	2	[0; +65535]

Bemerkung Beim Datentyp `boolean` wird `true` implizit in 1 und `false` in 0 konvertiert.

Variablensyntax

Variablen werden nach dem folgenden Schema erstellt: `Modifizierer Bezeichner = Wert;` oder `Modifizierer Bezeichner;`

Beispiel

```
1 int x = 4;
2 double y = 3.4;
3 double z;
4 z = y; // weise z den Wert von y zu
```

variablensyntax.java

Post- und Suffixe

Zahlensystem	Präfix	Beispiel
Oktalzahl	0	`int x = 016;` [5]
Hexadezimalzahl	0x	`int x = 0x16;`
Dezimalzahl		`int x = 16;`

Datentyp	Suffix	Beispiel
`float`	f	`float x = 0.1f;`
`long int`	l	`long x = 14576l;`
`double`	.	`double x = 12.;`
`int`		`int x = 12;`

postsuffix.java

[5] Wer ein handliches Werkzeug zum Umrechnen der verschiedenen Zahlensysteme braucht, wird hier fündig: `http://www.devwork.org/?s=calc`

Block und Scope

- Ein Block ist eine im Programmcode eingebaute Einheit, die sich durch geschweifte Klammern abtrennt.
- Blöcke können verschachtelt werden. Man spricht dann auch von *Ebenen*.
- Einheiten, die innerhalb des Blockes definiert werden, heißen *lokal* bezüglich dieses Blockes.
- Die Einheiten, die außerhalb des Blockes definiert sind, sind *nicht-lokal* bezüglich dieses Blockes.
- Der Gültigkeitsbereich eines lokalen Bezeichners wird als *Scope* bezeichnet.
- Die Sichtbarkeit eines Bezeichners hängt von seinem Scope und dem *Shadowing* ab.

Block und Scope - Beispiel

Beispiel

```
1 int x;
2 { // Blockanfang, Scopeanfang von y
3   double y;
4 } // Blockende, Scopeende von y
5 // hier gibt es schon kein y mehr
```

scope.java

Shadowing

Man kann in verschiedene Scopes gleichnamige Variablen definieren. Eine lokale Variable verdeckt dann die gleichnamige Variable solange, bis ihr Scope wieder geschlossen wird.

Beispiel

```
1 int x = 3;
2 { // Scopeanfang
3     int x = 4; // hier ist x = 4
4 } // Scopeende
5 // hier ist x wieder 3
```

Merksatz Gleichnamige Variablen sollte man nur in Ausnahmefällen verwenden! `shadowing.java`

Der , Operator

Will man mehrere Variablen gleichen Types deklarieren, kann man sie durch Komma auflisten.

Beispiel

```
1  int x = 3, y, z = 4; // hier ist y noch
       uninitialisiert
```

Achtung Vor der Verwendung von uninitialisierten Variablen bitte eine Wertzuweisung durchführen!

Zurück zu den Buchstaben...

In Java werden Buchstaben durch den `char`-Datentyp repräsentiert, der eine 16-bit Binärzahl darstellt. Also kodiert Java in `utf-16`. Man kann einzelne Zeichen durch Apostrophe darstellen. Es gibt neben den Buchstaben auch Sonderzeichen.

`\t`	Tabulator
`\n`	neue Zeile
`\“`	Anführungszeichen
`\‘`	Apostrophe
`\\`	Backslash ‘\’
`\uXXXX`	Unicode Zeichen mit Dezimalkodierung XXXX
`\u0041`	Unicodezeichen ‘A’

Beispiel

```
1  char c = 'a';
2  char d = '\u0041';
```

Strings

Während ‘a’, ‘x’, ... nur einzelne Buchstaben repräsentieren, möchte man gerne konfortabel ganze Zeichenketten abspeichern. Dazu gibt es ein weiterer Datentyp in Java: Der `String`. Strings schreibt man in Anführungszeichen: `String str = ‘‘Hallo Welt!’’;`

Beispiel

```
1 String a = "Hallo ";
2 String b = "Welt\n";
3 String c = a + b; // c == "Hallo Welt\n"
```

strings.java

Bildschirmausgabe

Um eine Variable, Zahl oder Zeichenkette auf dem Bildschirm auszugeben, gibt es folgende Funktionen:

- `System.out.print` gibt die Variable aus.
- `System.out.println` gibt zusätzlich eine neue Zeile mit aus.

```
1 String a = "Hallo Welt\n";
2 double b = 3.14;
3 System.out.println(a);
4 System.out.println(b);
5 System.out.println(15);
6 System.out.println("15");
7 System.out.println("Hans ist " + b + " Jahre
    alt"); // man kann mit + auch Zahlen in
    Strings casten!
```

printf in Java

Um die Ausgabe zu erleichtern, gibt es zur C-Funktion `printf` ein Analogon in Java: Die Funktionen `String String.format` und `void System.out.printf`. Während `String.format` den erstellten String zurückgibt, gibt `System.out.printf` den String sofort am Bildschirm aus. Von den Argumenten unterscheiden sich beide Funktionen nicht: Das erste Argument ist immer ein Formatierungsstring. Die weiteren Argumente sind werden dann nach den Angaben im Formatierungsstring in den endgültigen String eingesetzt.

```
1 String a = String.format("Ich wohne in der %s-
      Strasse %d. Und %s?", "Liebhold", 6, "Anna"
      );
```

Verschiedene Java-Programme

Ungewöhnlich mag es am Anfang erscheinen, dass es verschiedene Programmuntergruppen gibt. Man kann aus Java-Quellcode

- ein Command-Line Interface (**CLI**)
- eine graphische Benutzeranwendung
- ein Applet
- ein Servlet

entwickeln. Wir betrachten im Kurs nur das CLI und das Applet.

ⓘ**Achtung** Hier bitte nicht Java und JavaScript verwechseln!

Schablone eines CLI

Folgender Inhalt ist in der klassengleichnamigen Datei `foo.java` abzupeichern.

```
1 class foo
2 {
3   public static void main(String[] args)
4   {
5     // hier kommt dann der Programmcode hinein
6   }
7 }
```

Schablone eines Applets

Folgender Inhalt ist in der klassengleichnamigen Datei `foo.java` abzupeichern.

```
1  import javax.swing.JApplet;
2  public class foo extends JApplet {
3    public foo() {} // Konstruktor
4  // hier kommt Code zum Zeichnen rein
5    public void paint(java.awt.Graphics g) {...}
6  // der Code zum Programmstart
7    public void init() {...}
8  // hier kommt Code zum Fortsetzen/Starten rein
9    public void start() {...}
10 // Code beim Stoppen
11   public void stop() {...}
12 // Code, der beim Beenden abgeklappert wird
13   public void destroy() { ... }
14 }
```

applet.java

Funktionskopf

Um häufig wiederkehrende Abläufe einzukapseln, kann man in Java ein Funktionenkonstrukt verwenden. Funktionen können **nur** innerhalb einer Klasse definiert werden.

✓**Syntax** `public static` *Rückgabewert* *Funktionsname(Parameterliste)* { *Code* }

♫**Beispiel**

```
1  public static double func(double x, int n)
2  {
3     return x*n;
4  }
```

Rückgabewert

Eine Funktion kann mit dem Befehl `return Wert;` einen Wert zurückgeben.

- Wird der Befehl `return` ausgeführt, wird der Rest des Codes innerhalb der Funktion ignoriert!
- Soll die Funktion keinen Wert zurückgeben, verwendet man den Rückgabewert `void` - dann kann man über `return;` die Funktion terminieren.
- Funktionen ohne Rückgabewert nennt man *Prozeduren*.
- Funktionen mit Rückgabewert `boolean` nennt man *Prädikate*.

`funktionen.java`

Funktionsaufruf

Eine Funktion kann man im Codeteil wie folgt aufrufen:

Syntax *Funktionsname(Parameterliste);*

Beispiel `double x = func(1.0, 2);`

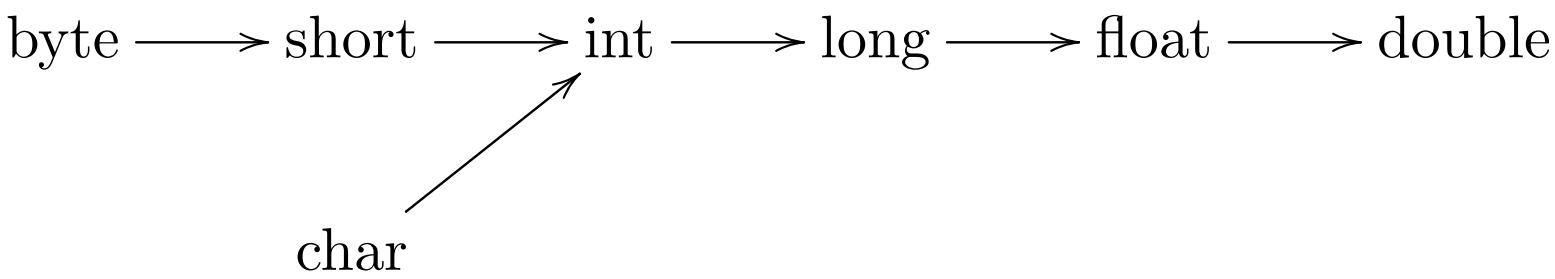
Funktionsüberladung

Man kann sogar mehrere Funktionen mit gleichen definieren, wenn sie eine unterschiedliche Signatur besitzen.

Was ist eine Signatur? Unter Signatur verstehen wir den Teil des Funktionskopfes, der den Funktionsnamen und die Parameterliste beinhaltet, d.h. alle Informationen außer den Rückgabewert. `signatur.java`

Konvertierungen

Unter einer Konvertierung versteht man eine Abbildung $f : A \to B$, die einen Wert aus A in einen Wert aus B umwandelt.
Konvertierungen sind also globale Funktionen. In Java gibt es implizite Konvertierungen, die automatische aufgerufen werden.
Diese sind im folgenden Diagramm dargestellt:



Achtung Die Rückrichtung erfolgt jedoch nicht automatisch!

explizite Konvertierungen

Explizite Konvertierungen sind mächtiger als die impliziten Konvertierungen in der Hinsicht, dass sie auch die Rückrichtung erlauben. Dazu gibt es den sogenannten Castoperator $(X) : Y \to X$ für jeden elementaren Datentyp X.

Beispiel Die Zuweisung `int i = (int) 1.0;` ist gültig, obwohl 1.0 eine Gleitpunktzahl ist.

Beispiel

```
1 char eins = 65; // ok, 65 passt in den
    Wertebereich von char
2 float f = 4.3; // falsch: 4.3 ist defacto
    double
3 f = 2; // ok: implizit int -> float
4 double d = 2/5; // -> d == 0 ??warum??
```

casten.java

Informationsverlust

Bei der Typumwandlung sollte man immer in Betracht ziehen, dass der neue Typ nicht mehr die gesamte Information des alten Typs wiederspiegeln kann.

- Von Gleitpunktzahl auf Ganzzahl werden alle Nachkommastellen abgeschnitten
- Während alle Ganzzahlen einen exakten Wert speichern, stellen Gleitpunktzahlen nur eine Näherung dar. Es gibt keine Gleitpunktzahl, die genau gleich π ist[6]!
- Eine `long`-Zahl hat mehr signifikante Stellen als eine `float`-Zahl

⟶`roundoff_error.java`

[6] Selbst die Konstante `Math.PI` ist nur eine Nährerung auf der Genauigkeit von `double`

Binäre Rechenoperatoren

Es gelten für elementare Datentypen die binären Rechenoperatoren für $a, b \in X$, X elementarer Datentyp:

Operator	Beispiel	Name
+	$a + b$	Addition
-	$a - b$	Subtraktion
/	a/b	Division
*	$a * b$	Multiplikation
%	$a\%b$	Modulo

Dazu gibt es jeweils einen entsprechenden Zuweisungsoperator, der die Operation $x = x \cdot y$ zu $x \cdot = y$ abkürzt:

Operator	Beispiel	Name
+=	$a+ = b$	Addition
-=	$a- = b$	Subtraktion
/=	$a/ = b$	Division
*=	$a* = b$	Multiplikation
%=	$a\% = b$	Modulo

⟶`modulo.java`

Post- und Präfixoperatoren

In Java gibt es Operatoren zum einfachen In- und Dekrementieren von Variablen: Die Post- und Präfixoperationen.

Operator	Beispiel	Name
++	$++a$ oder $a++$	Inkrementieren
--	$--a$ oder $a--$	Dekrementierung

Bemerkung Die Verwendung von Post- oder Präfixoperation unterscheidet sich nur um den Rückgabewert!

`postprefix.java`

Präfixoperationen

Bei den Präfixoperation handelt es sich um Funktionen, die die Veränderung bereits zurückgeben. D.h. man kann statt $++x$ auch einfach `x = x + 1;` bzw. statt $--x$ einfach `x = x - 1;` schreiben, da für eine weitere Variable y der Ausdruck $y = ++x$ äquivalent zu `y = x = x + 1;` ist.

Postfixoperationen

Bei den Postfixoperationen wird als Rückgabewert der Wert vor der Operation zurückgegeben, d.h. es wird der Wert zwischengespeichert. Also ist eine Zuweisung $y = x++$ äquivalent zu `tmp = x; x = x + 1; y = tmp`.

Merksatz Die Postfixoperationen sind ein Spezialfall und sind wegen der Zwischenspeicherung des Wertes etwas langsamer!

Unäre Operatoren

Neben den Castoperatoren gibt es noch

+	+a	Konvertierung `byte` $\cup$ `short` $\cup$ `char` $\rightarrow$ `int`
-	-a	Vorzeichenwechsel bei Ganzzahlen
!	!a	Negiert einen Wert vom Typ `boolean`

Relationale binäre Operatoren

Die folgenden Operatoren sind von der Form $X \times X \rightarrow$ `boolean`, liefern also einen Wahrheitswert zurück:

$<$	$a < b$	a kleiner b?
$<=$	$a <= b$	a kleiner gleich b?
$>$	$a > b$	a größer b?
$>=$	$a >= b$	a größer gleich b?
$==$	$a == b$	a gleich b?
$!=$	$a! = b$	a ungleich b?

Boolesche Algebra

Der Datentyp `boolean` wird mit den Operatoren `||`, `&&`, `!`, `^` zur Booleschen Algebra:

&&	true	false
true	true	false
false	false	false

||	true	false
true	true	true
false	true	false

^	true	false
true	false	true
false	true	false

$!\cdot : \begin{cases} \texttt{true} & \mapsto \texttt{false} \\ \texttt{false} & \mapsto \texttt{true} \end{cases}$ Da die Boole'sche Algebra insbesondere eine Algebra ist, können wir damit Bedingungen verknüpfen.

Eager-Evaluation

- Normalerweise abarbeitet Java alle Schritte der Reihenfolge nach so, wie sie beschrieben sind
- Würde man einer Funktion einen unendlich langen Vektor übergeben - aber die Funktion eigentlich nur die ersten zwei Elemente benötigen, dann wird sich Java damit krampfhaft bemühen, zunächst den unendlich langen Vektor zu erzeugen - und das dauert unendlich lange.
- Diese Strategie nennt man *Eager-Evaluation*.
- Der Vorteil dieser Strategie liegt in ihrer Geschwindigkeit - falls der Programmierer richtig programmiert hat.

Lazy-Evaluation

Zunächst führen wir die Definition ein, dass eine Funktion $f : X \to$ `boolean` für $X :=$ beliebige Menge von Datentypen *Prädikat* genannt wird.
Damit lässt sich am einfachsten das Prinzip der *Lazy-Evaluation* erklären:

- Nur im Fall der booleschen Algebraoperatoren wendet Java das Prinzip der *Lazy-Evalution* an:
- Hat man z.B. `Ausdruck1 || Ausdruck2` stehen, so wird - falls denn `Ausdruck1 == true` ist - `Ausdruck2` gar nicht mehr ausgewertet.
- Sind nun `Ausdruck1` und `Ausdruck2` Prädikate, so wird der Programmcode in `Ausdruck2` nicht ausgeführt!

Merksatz Die Boolesche Algebra ist wegen der *Lazy-Evaluation* nicht kommutativ, wenn man Prädikate verwendet!

`lazyEvaluation.java`

Mathematische Funktionen

Die Klasse `Math` hat ein ganzes Arsenal an brauchbaren Funktionen. Hier ein paar Anmerkungen:

- Trigonometrische Funktionen wie `Math.sin`, `Math.cos`, `Math.tan` arbeiten in Bogenmaß, nicht mit Grad!
- Die Funktion `Math.random` liefert eine gleichverteilte Doublezahl in $[0, 1)$. Um nun ein $n \in \{0, ..N\}$ zu bekommen, muss man `int n = (int) (Math.random() * (N+1));` schreiben.

⟶`fixpunkt_cosinus.java`

Bitoperationen

Java kann auch auf die einzelnen Bits von Ganzzahlen zugreifen. Dafür stehen die Operatoren `<<`, `>>`, `>>>` `|`, `&`, `~` zur Verfügung. Seien $a, b \in X$, X Ganzzahltyp. Werde a, b durch die Bitmuster $(a_0, ..., a_n), (b_0, ..., b_n)$ mit $a_i, b_i \in \{0, 1\}$ repräsentiert. Dann wirken die symmetrischen binären Operatoren auf jedes Bitpaar (a_i, b_i) wie folgt:

&	1	0
1	1	0
0	0	0

|	1	0
1	1	1
0	1	0

^	1	0
1	0	1
0	1	0

$$\tilde{}: \begin{cases} 1 & \mapsto 0 \\ 0 & \mapsto 1 \end{cases}$$

Shiftoperatoren

Die Shiftoperatoren `<<`, `>>`, `>>>` verschieben das Bitmuster nach links bzw. rechts.

ⓘ **Achtung** In Java ist bei Ganzzahlen an der ersten Stelle immer das Vorzeichenbit, das angibt, ob die Zahl positiv oder negativ ist.

- `z << n` verschiebt das Bitmuster von z um n Stellen nach links.
  - Hier kann es zu einem Vorzeichenwechsel kommen!
  - `z << n` entspricht $z \mapsto z \cdot 2^n$.
- `z >> n` verschiebt das Bitmuster von z um n Stellen nach rechts, lässt aber das Vorzeichenbit am Platz.
  - Die Zahl behält ihr Vorzeichen oder wird 0.
  - `z >> n` entspricht $z \mapsto z \cdot 2^{-n}$.
- `z >>> n` verschiebt das Bitmuster von z um n Stellen nach rechts und betrachtet das Vorzeichenbit als normales Bit.
  - Es kann zum Vorzeichenwechsel kommen.

Swappen

Hat man zwei Variablen $x, y \in X$, X elementarer Datentyp, so kann man beide Variablen wie folgt vertauschen:

- Mit einer Hilfsvariablen $t \in X$:
  - $t = x;$
  - $x = y;$
  - $y = t;$
- Mit den Bitoperationen bei Ganzzahlen:
  - $x = x \hat{} y;$
  - $y = x \hat{} y;$
  - $x = x \hat{} y;$

⊖`swappen.java`

Kontrollstrukturen

Kontrollstrukturen sind Möglichkeiten, den Codefluss zu steuern. Man hat die Möglichkeit

- bestimmte Codeblöcke nur unter bestimmten Bedingungen auszuführen (Bedingung)
- bestimmte Codeblöcke n-mal ausführen zu lassen (Schleife)

Bemerkung Ist der Codeblock einer Kontrollstruktur eine einzige Anweisung, kann man die umgebenden geschweiften Klammern fallen lassen.

Die if-Verzweigung

Will man bestimmte Programmabschnitte nur unter gewissen Voraussetzungen ausführen, bedient man sich dazu meist einer if-Verzweigung.

Syntax `if(`*Boolean-Wert*`) {` *Code, der ausgeführt wird, falls der Boolean-Wert wahr ist* `}`

Beispiel

```
1  boolean b = true;
2  if(b)
3  {
4     System.out.println("b ist wahr!");
5  }
```

Ansonsten...

In Java gibt es das zusätzliche Schlüsselwort `else`, welches man wie folgt verwenden kann:

Syntax `if(`*Boolean-Wert*`) { `*Code*` } else { `*wird ausgeführt, falls der Boolean-Wert falsch ist*` }`

Beispiel Die folgenden beiden Codestücke sind äquivalent:

```
1  boolean b = true;
2  if(b)
3  {
4    // b wahr
5  }
6  if(!b)
7  {
8    // b falsch
9  }
```

```
1  boolean b = true;
2  if(b)
3  {
4    // b wahr
5  }
6  else
7  {
8    // b falsch
9  }
```

Verschachtelung von if-Verzweigungen

Man kann nach einem `if` oder `else` wieder mit einer `if`-Verzweigung starten.

```
1  int x = 2;
2  if(x > 5) System.out.println("x groesser 5");
3  else if(x < 5) System.out.println("x kleiner 5
       ");
4  else System.out.println("x gleich 5");
```

mit Klammern:

```
1  int x = 2;
2  if(x > 5) { System.out.println("x groesser 5")
       ; }
3  else { if(x < 5) { System.out.println("x
       kleiner 5"); }
4  else { System.out.println("x gleich 5"); }}
```

ifelse.java

Das Dangling-else Problem

Bei obigen Code wird schnell klar, dass die Klammerung wichtig ist! Schreibt man nämlich

```
1  if(foo)
2    if(bar)
3      doSth();
4  else
5    doSth();
```

ist für ungeübte Augen nicht klar, zu welchen `if` das `else` gehört - es gehört zum `if(bar)` !

Merksatz Bei einer `if`-Verzweigung stets klammern!

Die Switch-Anweisung

Will man auf verschiedene Zustände des **gleichen** Wertes reagieren, bietet sich eine `switch`-Verzeigung an. Dabei übergibt man mit `switch(a)` der `switch`-Anweisung eine Variable/Konstante *a* eines elementaren Datentyps.

Beispiel

```
1  int a = 4;
2  switch(a)
3  {
4     case 0:
5        System.out.println("a ist 0");
6     case 1:
7     case 2:
8     case 3:
9        System.out.println("a ist < 4");
10       break;
11    default:
12       System.out.println("a ist >= 4");
13 }
```

In jeder `case A:` Anweisung wird überprüft, ob `A == a` ist. Falls dem so ist, wird der Code innerhalb des `case A:` ausgeführt.

Achtung Steht am Schluss kein `break;`, so wird automatisch der Code im nächsten `case`-Block abgearbeitet. Das `break` verlässt dann sofort die `switch`-Verzweigung.

Die `default`-Anweisung wird immer ausgeführt, und sollte deswegen am Schluss stehen.

Der tertiäre Operator ?:

In Java gibt es einen zusätzlichen Operator, der eine einfache `if-else`-Verzweigung abkürzt: Der ?:-Operator.

✓**Syntax** `a ? b : c;`

Dieser Ausdruck ist äquivalent zu `if(a) b; else c;`

♫**Beispiel**

```
1 int a = -2;
2 int betrag = (a > 0) ? a : -a;
```

äquivalent dazu wäre:

```
1 int a = -2;
2 int betrag;
3 if(a > 0) betrag = a;
4 else betrag = -a;
```

Will man repetitive Aufgaben ausführen, so kann man mit dem bereits gelernten zunächst auf folgende Form kommen:

```
1 public static long fac(long n)
2 {
3   if(n <= 0) return 1;
4   long r = fac(n-1);
5   return r;
6 }
```

Diesen Mechanismus nennt man *Rekursivierung*.

⊖`fixpunkt_cosinus.java` Je komplexer eine solche Schleife wird, desto schlechter kann man sie mit Rekursivierung verwirklichen. Daher gibt es bessere Möglichkeiten:

Die while-Schleife

Die `while`-Schleife ist eine Schleife, die solange ausgeführt wird, bis das boolesche Prädikat, das dem while übergeben wird, nicht mehr erfüllt ist.

✓**Syntax** `while(`*Boolescher Ausdruck*`)` { *Code* }

Die do-while-Schleife

Während die `while` Schleife immer **vor** dem Schleifencode den booleschen Ausdruck checkt, macht das die `do-while` Schleife immer am **nach** dem Ausführen des Schleifencodes.

🔔 **Bemerkung** Die `do-while`-Schleife wird alle mindestens einmal durchlaufen!

✓ **Syntax** `do { ` *Code* ` } while(`*Boolescher Ausdruck*`);`

Die for-Schleife

Die `for`-Schleife ist eine spezielle `while`-Schleife, die für iterative Zwecke gedacht ist.

✓**Syntax** `for(A ; B; C) { Code }`

wobei

- A := Initialisierungsliste. Hier kann man Variablen einen Anfangswert vor Schleifenbeginn zuweisen
- B := Abbruchskriterium, das **vor** jedem Schleifendurchlauf überprüft wird
- C := Code, der **nach** jedem Schleifenende ausgeführt wird

→`kleineEinMalEins.java`

Vergleich: for und while

Man kann jede for-Schleife in eine while-Schleife umwandeln:

```
1  for(int i = 0; i < 10; ++i)
2  {
3    System.out.println("Quadratzahl von " + i +
       " ist " + i*i);
4  }
```

kann geschrieben werden als

```
1  int i = 0;
2  while(i < 10)
3  {
4    System.out.println("Quadratzahl von " + i +
       " ist " + i*i);
5    ++i;
6  }
```

Schlüsselwörter continue/break

Innerhalb einer Schleife kann man mit den Schlüsselwörter `break` und `continue` das Verhalten der Schleife steuern:

- Wird ein `break;` aufgerufen, wird die aktuelle Schleife sofort beendet.
- Ein `continue;` beendet den aktuellen Schleifendurchlauf und startet mit dem Nächsten, falls das Abbruchkriterium erfüllt bleibt.

Beispiel Gibt nur die Quadratzahlen bis ausschließlich 5 aus:

```
1 for(int i = 0; i < 10; ++i)
2 {
3   if(i == 5) break;
4   System.out.println("Quadratzahl von " + i +
      " ist " + i*i);
5 }
```

Sprungmarken

Eine Neuheit in Java sind die Sprungmarken, die ausschließlich in Schleifen eingesetzt werden können.

- Eine Sprungmarke ist ein Bezeichner + ':', der vor einer Verzweigung von Schleifen stehen muss.
- Auf die Marke kann innerhalb dieser Schleifenverzweigung mit `break` *Bezeichner*; gesprungen werden.
- Wird zu einer Sprungmarke gesprungen, so wird der Code **nach** der Schleife fortgesetzt, vor die die Sprungmarke steht.

```
1  ausbruch:
2  for(int i = 0; i < 10; ++i) {
3    for(int j = 0; j < 10; ++j) {
4      if(j == 5) break ausbruch;
5      System.out.println(i + "*" + j + "=" + (i*
       j));
6    }
7  }
```

Die assert-Anweisung

Die `assert`-Anweisung eignet sich dazu, vom Programmierer verursachte falsche Werte zu identifizieren. Die assert-Anweisung überprüft einen booleschen Wert auf ihre Richtigkeit und beendet bei einer Falschaussage das Programm.

✓ **Syntax** `assert` **boolescher Ausdruck : Fehlermeldung;**

! **Achtung** Assert-Anweisungen dienen *nur* zum Debuggen von Programmen! Die assert-Anweisungen können aktiviert/deaktiviert werden!

- Mit `java -ea ...` wird Java-Code mit den Assertions ausgeführt
- Mit `java -da ...` werden alle Assertions ignoriert

→ `assert.java`

Eindimensionale Arrays

Arrays sind Tupel mit fester Dimension von gleichen Datentyp. Ein eindimensionalen Array kann man erzeugen durch

Syntax `Typ[] Bezeichner= new Typ[Länge ];`

Man sagt auch, dass der neue Bezeichner ein Arrayzeiger mit dem Wert eines Arrays ist.

Beispiel `int[] foo = new int[50];`

Bemerkung Man kann auch `int foo[] = new int[50];` schreiben.

Weiß man die einzelnen Werte schon, so kann man sich z.B. mit `double[] foo = { 1.9, 2.3, 4.6 }` die Größenangabe ersparen.

Aus dem Array kann man nun mit `foo[i]` auf den i.ten Eintrag zugreifen.

Achtung Java fängt immer an, bei 0 zu zählen, d.h. wenn man ein Array mit 50 Elementen bestellt, kann man nur Einträge von 0 bis 49 ansprechen!

Mehrdimensionale Arrays

Mehrdimensionale Arrays können auch sehr leicht erzeugt werden. Dazu macht man statt einer eckigen Klammerung einfach n-mal so viele:

Beispiel `double[][] matrix = new double[10][10];`
erzeugt eine 10x10 Matrix.
Für mehr Fine-Tuning muss man aber zu einer Schleife greifen. Will man z.B. nur eine obere Dreiecksmatrix erzeugen, so kann man das damit erreichen:

```
1  double[][] matrix = new double[10][];
2  for(int i = 0; i < 10; ++i)
3     matrix[i] = new double[i+1];
```

Zugriff auf das Element $(2, 2)$ erfolgt dann durch `matrix[1][1]`.
`array_mehrdim.java`

Arraylänge

Jede Variable vom Typ Array speichert ihre Länge zusätzlich mit ab. Diese kann man mit *Bezeichner*.`length` abrufen. So braucht man die Länge selbst nicht mit abspeichern und kann eine Funktion schreiben, die den Inhalt eines Arrays zeilenweise ausgibt:

```
1  public static void func(float[] arr)
2  {
3    for(int i = 0; i < arr.length; ++i)
4      System.out.println(arr[i]);
5  }
```

⊖array_mehrdimlength.java

Die for-each-Schleife

Will man einen Code auf jedes Element eines Arrays ausführen, so kann man eine vereinfachte for-Schleife verwenden: Die `for-each`-Schleife.
Obiges Beispiel kann man damit schreiben zu

```
1 public static void func(float[] arr) {
2   for(float f : arr)
3     System.out.println(f); // der aktuelle
      Wert wird in f abgespeichert!
4 }
```

array_mehrdimforeach.java

Programmargumente

Einem Programm kann man bei Programmstart beliebig viele Argumente mitgegeben werden, die durch Leerzeichen getrennt werden. Dieses wird dann in dem Stringarray gespeichert, welches die `main`-Funktion als Parameter verlangt.

```
1 public static void main(String[] args){
2   for(String s : args)
3     System.out.println(s); // gibt alle
    Argumente zeilenweise aus
4 }
```

programmargumente.java

Call by Value

Funktionsaufrufe von elementaren Datentypen wie `int`, `boolean`, etc. erfolgen stets nach dem Prinzip *Call by Value*: Call by Value besagt, dass eine Funktion nicht die Variablen erhält, sondern nur Kopien deren Werte. Anders gesagt, kann damit eine Funktion nur die Werte ihrer Parameter verändern - nicht die Werte der Argumente.

```
1 void swap(int a, int b) {
2   int tmp = a;
3   a = b;
4   b = tmp;
5 }
6 ...
7 int x = 0, y = 1;
8 swap(x,y); // x bleibt 0 und y bleibt 1 !
```

Call by Sharing

Hat man es hingegen mit Arrayzeigern zu tun, so behandeln die Funktionen zwar immernoch die Arrayzeiger 'by value', aber nicht mehr deren Inhalt, auf das das Array zeigt!

```
1  void swap(int[] arr, int a, int b) {
2    int tmp = arr[a];
3    arr[a] = arr[b];
4    arr[b] = tmp;
5  }
6  ...
7  int[] array = new int[2];
8  array[0] = 10; array[1] = 20;
9  swap(array, 0, 1); // -> array[0] == 20 &&
       array[1] == 10;
```

⟶callbyvalue.java

Wertsemantik

Bisher waren Zuweisungen und Vergleiche so definiert, wie es dem gewöhnlichen Menschenverstand entspricht:

`int a = 5;`	$a \mapsto 5$
`int b = 4;`	$b \mapsto 4$
`b = 5;`	$a \mapsto 5$ $b \mapsto 5$

Dieses Verhalten nennt man Wertsemantik. Dementsprechend liefert `a == b` den Wert `true` zurück.

Referenzsemantik

Anders verhält sich dies bei Zeigern: Nicht der Inhalt wird zugewiesen, sondern die Zeiger selber:

```
int[] a = new int[3];
int[] b = new int[2];
```
$a \mapsto \{a_0, a_1, a_2\}$
$b \mapsto \{b_0, b_1\}$

```
b = a;
```
$\left.\begin{matrix} a \\ b \end{matrix}\right\} \mapsto \{a_0, a_1, a_2\}$

Zum Schluss ist wieder `a == b`, aber b zeigt nun wirklich auf die gleiche Speicheradresse wie a. Ein `b[0] = 1;` würde eine Veränderung von `a[0]` bedeuten!

(!) **Achtung** Haben a und b getrennte Speicherplätze, so ist immer `a != b`, auch wenn die Inhalte beider Zeiger paarweise gleich ist - also $a[i] = b[i] \; \forall i \in \{0, ...\text{a}.length\}$.

⊖ `referenzsemantik.java`

Das final Schlüsselwort

Das verhalten von Variablen kann mit dem final-Schlüsselwort eingeschränkt werden:

- Bei elementaren Datentypen verwandelt ein `final` die Variable zu einer unveränderlichen Konstanten.
- Ein Zeiger kann nur noch auf seinen Inhalt zeigen. Man kann also den Zeiger keinen neuen Inhalt zuweisen!

```
1  final int i = 0;
2  ++i; // falsch!
3  final int[] zeiger = new int[2];
4  zeiger[0] = 4; // ok!
5  zeiger = new int[3]; // falsch! Hier wuerde
       ein neues Array dem zeiger zugewiesen
       werden
```

Der `null`-Zeiger

Um Zeiger, die auf nichts zeigen sollen, zu klassifizieren, gibt es den `null`-Zeiger:

- Jeder Zeiger kann dem `null`-Zeiger zugewiesen werden.
- Greift man auf den Inhalt eines solchen Zeigers zu, so wirft Java die `NullPointerException` und bricht das Programm damit ab.

⟶`nullPointer.java`

Dynamische Speicherallozііerung

- Objekte, die mit dem `new` Schlüsselwort erzeugt werden, nennt man *dynamisch erzeugte Objekte*.
- Bei dynamisch erzeugten Objekten wird zur Compilezeit ihre aktuelle Größe nicht ausgewertet - das geschieht dann zur Laufzeit.
- Man kann also die Arraylängen mit einer Variablen parametrisieren.
- Dynamisch erzeugte Objekte werden nicht **automatisch** gelöscht, wenn der Scope verlassen wird, da hier mit der Referenzsemantik gearbeitet wird.
- Zum Löschen gibt es einen *Garbage Collector*, der regelmäßig nach Speicher ausschau hält, auf den kein Zeiger mehr verweist.
- Will man seinen Speicher bewusst innerhalb des Scopes noch freigeben, kann man den Zeiger zurücksetzen, indem man ihm dem `null`-Zeiger gleichsetzt.

Strukturen

Je größer ein Java-Programm wird, desto komplizierter können die Funktionsaufrufe werden: Man hat viele Variablen, die ständig als festes Bündel Funktionen übergeben werden müssen. Um sich diese Schreibarbeit zu sparen (und damit auch die Fehleranfälligkeit) gibt es in Java den Typ `class`, mit dem man mehrere Datentypen zu einem neuen Datentyp zusammenfassen kann. Variablen innerhalb einer `class` werden als *Membervariable* oder *Attribute* bezeichnet.

```
1  class Haus
2  {
3     int hausnummer;
4     String strassenname;
5  }
```

Verwendung von Strukturen

Nun braucht man einer Funktion nicht mehr Hausnummer **und** Straßennummer mitgeben, sondern lediglich ein Objekt vom Typ `Haus`:

```
1  void tuwas(Haus haus) {
2    System.out.printf("Haus steht in der Strasse
       %s %d.\n", haus.strassenname, haus.
     hausnummer);
3  }
4  ...
5  Haus haus = new Haus();
6  haus.hausnummer = 3;
7  haus.strassenname = "Albertus-Magnus-Strasse";
8  tuwas(haus);
```

⟶strukturen.java

Methoden

In einer Klasse gibt es neben Membervariablen auch *Methoden*. Das sind einfach gesagt Funktionen, denen man nicht mehr die Klasse übergeben muss.

♡**Merksatz** Methoden sind Funktionen, die nicht das Schlüsselwort `static` besitzen, aber dafür einen versteckten Zeiger auf das aktuelle Objekt mitbekommen!

✓**Syntax**

```
1 class Hallo
2 {
3     membervariablen ...
4     methoden ...
5 }
```

Alle Elemente einer Klasse werden als *Member* bezeichnet.

Beispiel einer Klasse

Beispiel

```
1  class Person {
2  // Zugriff auf die folgenden Elemente nur von
       Methoden der Klasse:
3     private int alter;
4     private String name;
5  // Zugriff von ueberall auf die folgenden
       Elmente:
6     public void frageAlter() { ... }
7     public void sageAlter() { ... }
8     public void frageName() { ... }
9     public void sageName() { System.out.println(
       name); }
10 }
```

klasse_methoden.java

grobe Eigenschaften

- **Membervariablen/Attribute/Datenfelder:**
  - Klassen können beliebige Attribute enthalten (auch Klassen) - eine Instanz von sich selbst auch!
- **Methoden:**
  - Methoden haben vollen uneingeschränkten Zugriff auf alle Member
  - Methoden stehen genauso wie Funktionen innerhalb der Klassendefinition und brauchen nicht das Schlüsselwort `static`.

Gültigkeitsbereich von Attribute- und Methodennamen sind auf die Klasse beschränkt. Will man ein `public` Attribut einer Klasse Person ansprechen, geht das nur wenn man eine Instanz erzeugt hat (z.B. durch `Person i = new Person();`), dann kann man mit `i.alter;` auf das Alter von i zugreifen. Ausnahmen sind Attribute/Methoden, die das Schlüsselwort `static` verwenden!

Zugriffsrechte

Erwähnt man keine Zugriffsrechte, so kann man automatisch auf alle Elemente des aktuellen Packages, also insbesondere der aktuellen Quellcodedatei, zugreifen. In Java kann man dieses Rechte für spezifische Methoden und Memeber erweitern oder einschränken. Man spricht dann von *Kapselung*. Deswegen gibt es die folgenden Zugriffsspezifizierer, die ab ihrer Deklaration für alle untenstehende Klassenelemente gelten, bis ein anderer Zugriffsspezifizierer verwendet wird. Die Klasse beginnt automatisch mit dem Spezifizierer `private`.

public versus private

PRIVATE	PUBLIC
Member ist nur den Klassenmethoden zugänglich.	Member ist frei zugänglich
Kein Zugriff von außerhalb der Klasse möglich.	Direkter Zugriff von außen möglich. Man kann also mit `public` Methoden z.B. auf `private` Member indirekt zugreifen.
Kapselung ist guter Programmierstil, da man Zugriffe auf die Attribute dadurch überprüfen und Fehleingaben korrigieren kann.	Sind die Daten gekapselt, so kann man nur mit Methoden indirekt darauf zugreifen. Eine solche Methode nennt man auch Schnittstellenmethode oder Interfacemethode

Ein weiterer Zugriffsspezifizierer ist `protected`, der später bei der Vererbung eine Rolle spielt. Dieser verhält sich ähnlich wie `private`.

Objekte - Instanzen von Klassen

Deklariert man eine Klasse, so wird noch kein Arbeitsspeicher dafür während der Programmausführung belegt. Erst wenn man eine Instanz einer Klasse erzeugt geschieht dies. Solche Instanzen nennt man *Objekte*. Solche Objekte kann man nur mit dem Schlüsselwort `new` wie Arrays anlegen:

```
1 Person ich = new Person();
```

Merksatz Jede Membervariable wird für jedes Objekt eigens neu angelegt, die Methoden teilen sich aber alle Instanzen einer Klasse.

Der . Operator

Zugriff auf Member erfolgt durch den . Operator:

Beispiel

```
1 Person du = new Person();
2 du.frageAlter(); // hier ist du Zeiger auf ein
      Objekt der Klasse Person
```

Der this-Zeiger

Wir wissen bereits, dass sich alle Objekte einer Klasse die Methoden teilen müssen. Da eine Methode aber die Möglichkeit haben soll, auf die Membervariablen eines Objektes zugreifen, stellt sich die Frage:

Woher weiß die Methode, mit welchem Objekt sie gerade arbeiten?

Hier verwendet der Compiler wieder einen Trick: Er gibt jeder Methode eine versteckte Zeiger auf das Objekt mit: Den *this*-Zeiger. Die `this`-Zeiger ist ein finaler Zeiger auf das Objekt selbst!

Man kann also sagen, dass man Methoden äquivalent umschreiben kann zu: (*Stimmt nicht ganz: Wir vernachlässigen hier die Verkapselung!*)

```
1 void sageAlter(final Person this) {
2   System.out.println(this.alter);
3 }
```

this_zeiger.java

Finale Attribute

Man kann auch einzelne Attribute als final deklarieren.

- Ist das Attribut ein elementarer Datentyp, so wird er konstant
- Ist es ein Zeiger, so wird der Zeiger konstant
- Dies gilt dann auch für alle Instanzen dieser Klasse

?Wie belegt man überhaupt konstante Attribute? Da man diese nun nicht mehr mit Methoden/Funktionen beschreiben kann, gibt es nur noch die Möglichkeiten:

- In der Klassendeklaration die Attribute zu initialsieren
- Eine spezielle Methode aufzurufen: Den *Konstruktor*

Konstruktoren

Ⓢ**Lösung** Es gibt besondere Methoden, die beim Erstellen einer Instanz aufgerufen werden, die sogenannten *Konstruktoren*. Diese erkennt man daran, dass

- sie immer den Namen der Klasse tragen
- sie keinen Rückgabewert haben (nicht einmal `void`).

Beispiel

Wenn man also unsere Klasse Person um einen Konstruktor erweitert, können wir schreiben:

```
1  public class Person {
2    int alter; String name;
3    public Person(int _alter, String _name) {
4      alter = _alter;
5      name = new String(_name);
6    }
7    public static void main(String[] args) {
8      Person er = new Person(19, "Fabian");
9    }
10 }
```

konstruktoren.java

Der Default-Konstruktor

Bemerkung Generell hat jede Klasse einen Konstruktur. Falls man diesen nicht manuell schreibt, versucht der Compiler, einen *Default-Konstruktur* zu schreiben - also einen Konstruktor, der keine Parameter benötigt. Um einen Default-Konstruktor aufzurufen, gibt es folgende Möglichkeiten:

```
1 public class Person
2 {
3   public Person() { }
4   public static void main(String[] args) {
5     Person ich; // hier wird der Default-
      Konstruktor nicht aufgerufen, weil nur der
      Zeiger angelegt wird!
6     Person du = new Person(); // ok!
7     Person er(); //Fehler: Der Compiler meint,
      man deklariert hier eine Funktion
8   }
```

Default-Zuweisung

Weiß man, dass bestimmte Member immer mit den gleichen Wert zu initialsieren sind, so kann man die Wertzuweisung auch direkt in der Memberdefinition unterbringen.

```
1 public class Auto {
2   int jahrgang = 1996;
3   String marke = "Eigenbau";
4   ...
5 }
```

Achtung Die Reihenfolge der Initialisierungen ist immer die Reihenfolge, in der die Memberelemente einer Klasse deklariert worden sind.

Copy-Konstruktor

Will man bei der Initialisierung eines Objektes das Objekt einem anderen Objekt des selben Types zuordnen, so wird der Copy-Konstruktur dafür aufgerufen. Er wird **nicht** vom Compiler automatisch generiert. Sobald man einen Konstrukter geschrieben hat, schreibt der Compiler den Default-Konstruktor nicht mehr!

- `Person ich = new Person(); // ich hoffe, der Default-Konstruktor ist schon geschrieben`
- `Person du = ich; // falsch: Hier referenziert dann du auf ich, d.h. wird haben nur ein Objekt, aber zwei Referenzen`
- `Person du = new Person(ich); // richtig! Hier wird ein neues Objekt erzeugt`

Bemerkungen zum Copy-Konstruktor

Beispiele für einen Prototyp des Copy-Konstruktor sind `Person(final Person p);` oder wahlweise `Person(Person p);` .

Bemerkung Der Copy-Konstruktor muss immer als erstes Argument eine Referenzvariable auf die eigene Klasse verlangen.

Achtung Der Copy-Konstruktor wird wie jeder andere Konstruktur nur bei der Objektinitialisierung aufgerufen. Man kann keinem schon initialisierten Objekt ein anderes Objekt zuweisen!

`copykonstruktor.java`

Konvertierungskonstruktoren

Konstruktoren werden als *Konvertierungskonstruktoren* bezeichnet, wenn sie mehrere Argumente erwarten, aber kein Argument vom Typ der eigenen Klasse.

```
1 class Person {
2   int alter;
3   Person(int a) {
4     alter = a;
5   }
6 }
```

- `Person ich = new Person(25); //`ruft den Konvertierungskonstruktor Person(int) auf
- `ich = 22; //` geht so nicht!

Statische Member

Wenn man eine Membervariable haben will, die jede Instanz einer Klasse teilen soll, kann man diese mit `static` deklarieren. Dabei gilt:

- Statische Member extistieren nur einmal pro Klasse.
- Eine solche Membervariable lebt bis zum Programmende.
- Sie existieren auch, wenn überhaupt keine Instanz des Objektes erschaffen wurde.
- Sie sollten mit einem Wert direkt initialisiert werden - und nicht im Konstruktor
- Sie verhalten sich wie globale statische Variablen.

⊖`static_member.java`

Statische Methoden

Will man auch eine Methode haben, die direkt aufrufbar sein soll - also ohne eine Instanz zu erzeugen, dann muss man diese auch mit `static` deklarieren. Zu beachten ist:

- Statische Methoden werden wie normale Methoden von allen Objekten einer Klasse mitbenutzt.
- Sie haben keinen `this`-Zeiger
- Sie können nur auf die statischen Memberelemente einer Klasse zugreifen, da ja die normalen Memberelemente für jede Instanz verschieden sein können.

☎**Hinweis** Wenn man die Kapselung vernachlässigt, so kann man jede statische Membervariable als globale statische Variable und jede statische Methode als Funktion äquivalent umschreiben.

publike Klassen

Bekommt eine Klasse das Schlüsselwort `public`, so wird diese für andere Java-Klassen zur Verfügung gestellt, die nicht im aktuellen Package definiert sein müssen.

ⓘ **Achtung** Man muss aber jeder publike Klasse in eine separate Quellcodedatei abspeichern, die genau den Namen der Klasse besitzt!

Wenn man diese Klasse zu einer Applet-Klasse machen will, so **muss** man sie publik machen!

⊖`public_classes`

Packages I

Daneben kann es auch oft vorkommen, dass sich bei sehr vielen Klassen manche Klassennamen überschneiden. Um dem entgegenzukommen gibt es in Java Pakete:

- Quellcodedateien können zunächst in ein eigens genanntes `package` gelegt werden. Dazu schreibt man in die erste Zeile `package` *Packagebezeichner*;.
- Die Quellcodedateien müssen im Projektverzeichnis in dem Unterverzeichnis *Packagebezeichner* stehen.
- Nachher kann man auf eine jede publike Klasse des Packages *Packagebezeichner* zugreifen mit *Packagebezeichner.Klassenbezeichner*.
- Man kann den Packagebezeichner durch Punkte unterteilen - jeder Punkt entspricht dann ein weiteres verschachteltes Unterverzeichnis. Dann muss aber auch jeder Punkt mit einer publiken Klasse mit angegeben werden!

Packages II

- Will man den Packagebezeichner bei bestimmten Klassen weglassen, so kann man diese importieren mit `import` *Packagebezeichner*`.`*Klassenbezeichner*.
- Ganze Pakete lassen sich mit `import` *Packagebezeichner*`.*` importieren.
- Das Paket `java.lang` ist automatisch importiert, s.d. man nicht `java.lang.String` schreiben muss.

Statisches Importieren

Man kann zusätzlich statische Funktionen und Methoden so importieren, dass man sie ohne Klassenbezug aufrufen kann.

```
1 import static java.lang.Math.cos;
2 import static java.lang.Math.PI;
3 //oder gleich alles: import static java.lang.
      Math.*;
4 ...
5 double r = cos(PI * 1.5);
```

Enums I

Enums sind eine Spezialklasse, in der alle Attribute *konstant* sein müssen. In Enums werden dazu verwendet, bestimmte Konstante Zustände festzuhalten.

```
1 enum Richtung { LINKS, RECHTS, OBEN, UNTEN; }
2 ...
3 Richtung r = Richtung.LINKS;
4 if(r != Richtung.RECHTS) System.out.println("
    Du gehst nicht nach rechts!");
```

Enums II

Man kann auch Wertzuweisungen machen:

```
1  enum Ampel
2  {
3    ROT(0), GRUEN(1), GELB(2);
4    public int zahl;
5    private Ampel(int i) { zahl = i; }
6  }
7  ...
8  Ampel a = ...;
9  if(a.zahl == Ampel.ROT.zahl) System.out.
       println("Halt!");
```

Bemerkung Jedes Enum hat eine statische Methode `values()`, welches ein Array aus allen bekannten Zuständen zurückliefert.

`enums.java`

Klassenhierarchie

Wenn man eine Spezialisierung einer Klasse schreiben will, möchte man nicht die ganze alte Klasse abschreiben. Stattdessen sollte man nur die Spezialisierung schreiben und auf die alte Klasse hinweisen können. Zum Glück kann dies in Java durch die *Vererbung* erreicht werden: Man leitet von einer Basisklasse eine Kindklasse ab. Dann erbt die Kindklasse alle Elemente der Basisklasse und man kann neue Elemente hinzufügen oder alte verändern. Abgeleitete Klassen können wieder Basisklassen werden, was zu einer Klassenhierarchie führt.

```
1 class Ableitung extends Basisklasse
2 {
3 }
```


vererbung.java

Schlüsselwort `protected`

`protected` ist neben `private` und `public` ein weiterer Zugriffsspezifizier für Methoden und Attribute:

- Innerhalb des einen Packages ist `protected` wie `public`
- Kindklassen können auf geerbte `protected`-Elemente zugreifen, als wären sie `public`-Elemente
- Ansonsten ist ein Zugriff untersagt

Private Vererbung I

Hat die Basisklasse `private` Elemente, so kann die Kindklasse nicht darauf zugreifen!

Beispiel

```
1  class UniMitglied {
2    protected char Geschlecht;
3    private int ganzgeheim;
4  }
5  class Student extends UniMitglied {
6    public void setzeGeschlecht(char g) { //
       Geschlechtsumwandlung ;-)
7        Geschlecht = g;
8    }
9    public int gibGanzGeheim() {
```

Private Vererbung II

```
10          return(ganzgeheim); // fehler:
        ganzgeheim ist in der Basisklasse private
        und damit unzugreifbar
11      }
12  }
13  ...
14  Student ich = new Student();
15  ich.Geschlecht = 'w'; //fehler, falls wir hier
        ein anderes Package haben
16  ich.setzeGeschlecht('w'); // ok
```

⊖vererbung_protected.java

Shadowing I

Man kann bei der Vererbung die gleichen Namen von vererbten Member/Methoden für eigene Member/Methoden verwenden. Nun kann man aber auf alle vererbten Methoden und Member mit einem konstanten Zeiger zugreifen: Der `super`-Zeiger. (!)**Achtung** Anders als in C++ können Methoden nicht geshadowed werden.

Beispiel

```
1  class B {
2    public int a, b;
3    public void f();
4    public void g();
5  }
6
7  class A extends B {
8    public int a, b;
9    public void f(int c) {
```

Shadowing II

```
10        super.a = c; // weist dem a von B den Wert
          von c zu
11      }
12
13    }
14    A alvin = new A();
15    alvin.a = 1; // a aus A = 1
16    alvin.f(3); //ok, f(int) aus A
17    alvin.g(); //ok, g() aus B wird aufgerufen
18    alvin.f(); //ok, hier wird die Methode von A
          aufgerufen
```

vererbung_shadowing.java

Konvertierungen

Objekte vom Typ der abgeleiteten Klasse können **implizit** in den Typ der Basisklasse konvertiert werden - **Die Rückrichtung gilt nicht!**

```
1  class B {...}
2  class A extends B {}
3  class Main {
4    static void fb(B b) {}
5    static void fa(A a) {}
6  ...
7  B berta = new B();
8  A albert = new A();
9  fb(albert); // ok, implizite Konvertierung von
       A -> B
10 fa(berta); // fehler, keine Konvertierung von
       B -> A moeglich
```

updowncasting.java

Konstruktoren I

Man kann die Member der Basisklasse nur über einen Konstruktor der Basisklasse initialisieren. Deswegen muss dieser innerhalb des Konstruktors der abgeleiteten Klasse aufgerufen werden.

Beispiel

```
1  class B {
2    public int a,b;
3    public B() {}
4    public B(int a, int b) {
5      this.a = a; this.b = b;
6      }
7  }
8
9  class A extends public B {
10   public int c;
11   public A(int _c, int _a, int _b) { //ok
```

Konstruktoren II

```
12        super(_a, _b);
13        c = _c;
14      }
15      public A(int _c) { //ok, ruft
          Standardkonstruktor von B auf
16        c = _c;
17      }
18      public A(int _a, int _b) { // hier wird der
          Defaultkonstruktor von B aufgerufen
19        a = _a;
20        b = _b;
21      }
22  }
```

vererbung_konstruktorreihenfolge.java

Reihenfolge der Konstruktoraufrufe

Die Reihenfolge der Konstruktoraufrufe ist demnach:

1. Konstruktor der Memberelemente der Basisklasse
2. Konstruktor der Basisklasse
3. Konstruktor der nicht-vererbten Memberelemente der abgeleiteten Klasse
4. Konstruktor der abgeleiteten Klasse

⟶`vererbung_konstruktorreihenfolge_mitHat.cpp`

erneuter Konstruktoraufruf

Man kann innerhalb eines Konstruktores einen anderen Konstruktor aufrufen.

- Mit `super(...)` können wir den Konstruktur der Elternklasse aufrufen.
- Mit `this(...)` können wir einen weiteren Konstruktor unserer eigenen Klasse aufrufen.
- Sowohl `this()` also auch `super()` müssen in der ersten Zeile stehen.
- Man kann **entweder** `super()` oder `this()` benutzen.

⟶`this_konstruktor.java`

finale Klassen

- Es gibt Klassen, für die eine Ableitung nicht in Frage kommt.
- Mit dem Schlüsselwort `final` kann man eine Ableitung verbieten.

```
1 final class Unableitbar
2 {
3     ...
4 }
```

Zugriffsspezifizierer für Attribute/Methoden

Elemente einer Klasse wie Attribute oder Methoden können verschiedene Zugriffsspezifizierer aufweisen. Hier nochmals eine Zusammenfassung:

- Auf `public` Element kann man von überall zugreifen, wenn man auf die Klasse zugreifen kann ($\mapsto$ publike Klassen)
- `protected` Elemente sind
  - innerhalb der Klasse,
  - innerhalb abgeleiteter Klassen und
  - innerhalb des gleichen Paketes ansprechbar.
- `private` Elemente können nur innerhalb der Klasse direkt benutzt werden.
- Hat ein Element *kein Schlüsselwort*, so kann man es nur innerhalb des aktuellen Paketes ansprechen.

Zugriffsspezifizierer für Klassen

Klassen und Enums können nur zwei verschiedene Zugriffszustände haben:

- `public` Klassen sind aus allen Packages auffindbar.
- Klassen ohne `Zugriffsspezifizierer` sind nur innerhalb ihres Packages gültig. Außerhalb des Packages kann man die Klasse nicht mehr sehen.

Was sind Exceptions?

Exceptions sind Ausnahmeklassen, die man bei einer Ausnahme werfen kann.

- Ausnahmen sind in der Regel Fehler, die auftreten können.
- Auf den ersten Blick kann man mit `boolean`-Rückgabewerte und `if, else`-Abfragen auskommen.
- Oft kommt man mit dieser Methode nicht weit:
  - Ruft man fehlerträchtige Prädikate auf, muss man ständig den Rückgabewert prüfen.
  - Konstruktoren haben keinen Rückgabewert
  - Es ist besser, den Fehler an einen Bereich zu übergeben, wo man auf diesen reagieren kann

Realisierung

Programmiertechnisch kann man dies in drei Teile gliedern:

- `try`-Block, in dem Fehler geworfen werden können/dürfen.
- `catch`-Block, in dem ein abgefangener Fehler ausgewertet werden kann.
- `throw`-Anweisung, die eine Exception auswirft.

Eine Exception ist eine Klasse, die von der Klasse `Throwable` ableitet.

Beispiel einer Realisierung

Syntax

```
1  try {
2     ...
3     // Code, der Fehler produzieren koennte
4     Typ variable = new Typ();
5     throw variable; // wirft eine Exception!
6     ...
7  }
8  catch(Typ variable) {
9     ... // Code, der auf den Fehler reagiert.
10 }
```

exceptions.java

Ablauf einer Exception

- `try`-Block, in dem Fehler geworfen werden können/dürfen.
- Das Programm *läuft* in einen `try`-Block.
- In einer Funktionsebene oder direkt im `try`-Block wird eine `throw`-Anweisung ausgeführt.
- Wird ein `catch`-Block vom gleichen Typ wie der des `throws` gefunden, so wird diesem das `throw`-Argument übergeben.

ⓘ **Achtung** Wird kein zuständiger `catch`-Block gefunden, oder der `throw` außerhalb eines `try`-Blockes ausgeführt, wird das Programm abgebrochen.

Hinauswerfen mit `throws`

- Es gibt Methoden/Funktionen, in der eine Auswertung einer Exception keinen Sinn macht.
- Stattdessen sollte die Exception in die nächst höhere Ebene weitergegeben werden.
- Dies kann man dadurch erreichen, dass man zum Funktionsprototypen eine Liste `throws Exception0, Exception1, ...` aller möglichen Exceptions mitgibt, die ausgeworfen werden können.
- Nun braucht man für einen `throw`-Wurf innerhalb der Methode keinen `catch-try`-Block.

`throws.java`

mehrere Catch-Blöcke

- Es können mehrere `catch`-Blöcke definiert werden
- `catch`-Blöcke können parameterlos sein[7], z.B. `catch(int) { }` Dieser fängt dann alle Throws vom Typ `int` ab.

```
1  try {
2    int i;
3    if(i > 5) throw new XMLParseException();
4    else throw new CloneNotSupportedException();
5  }
6  catch(XMLParseException i) { ... }
7  catch(CloneNotSupportedException d) { ... }
```

[7] Das übergebene Argument wird dann einfach weggeworfen

finally - der Allesfänger

Man kann nach den `catch`-Blöcken einen Block definieren, der immer ausgeführt wird - egal ob ein Fehler aufgetreten ist oder nicht.

- Der `finally` Block ist ein spezieller, der immer nach dem `try-catch`-Blöcken ausgeführt wird.
- Man kann den `finally` Block nur überspringen, wenn man `System.exit` aufruft, oder die Virtuelle Maschine zum Absturz bringt.
- Ein einfaches `return` innerhalb der `try-catch`-Blöcke überspringt zwar den aktuellen Block, nicht aber den `finally`-Block!
- Der `finally`-Block ist der letzte Block einer `try-catch`-Blockreihe. Danach kann kein `catch` mehr folgen.

Syntax

```
1 try {
2   // irgendwelche Anweisungen
3 }
4 finally {
5 ...
6 }
```

finally.java

Polymorphie

Hat man von einer Basisklasse mehrere Klassen abgeleitet, so möchte man diese vielleicht in ein gemeinsames Array vom Typ der Basisklasse abspeichern[8]. Nun hat man aber in der abgeleiteten Klasse bestimmte Methoden anders wie in der Basisklasse definiert, und möchte, dass nicht die Methode der Basisklasse, sondern die Methode der abgeleiteten Klasse aufgerufen wird, wenn man eine Instanz dieser Klasse aus den Array nimmt.

[8] geht wegen der impliziten Konvertierung von abgeleiteten Klasse zur Basisklasse

Beispiel zur Polymorphie

- Wir schreiben eine Basisklasse Haustier, und leiten von ihr verschiedene Tiere wie Hund, Katze, Vogel, ... ab
- Alle Haustiere geben Laute von sich, aber jede Art hat einen anderen Laut
- Wenn wir nun Instanzen von verschiedenen Tieren in einem Haustier-Array abspeichern, könnten wir vielleicht die Laute verlieren?

Code zum Beispiel I

```
1  class Haustier {
2    public void gibLaut() {}
3  };
4  class Hund extends Haustier {
5    public void gibLaut() {
6      System.out.println("wuff!");
7    }
8  };
9  class Katze extends Haustier {
10   public void gibLaut() {
11     System.out.println("miau");
12   }
13 };
14 Haustier[] tiere = new Haustier[5]; // Array
       aus Zeigern auf Haustiere
```

Code zum Beispiel II

```
15  tiere[0] = new Hund(); // ok, implizite
        Konvertierung: abgeleitete Klasse ->
        Basisklasse
16  tiere[1] = new Katze(); // dito
17  for(int i = 0; i < 5; ++i) { //haben die Tiere
        uns was zu sagen?
18    tiere[i].gibLaut(); //-> hier wird der
        entsprechende gibLaut() aufgerufen - die
        Methode Haustier::gibLaut ist nur eine
        Dummy-Methode!
19  }
```

Es ist also nicht ausschlaggebend, welchen Typ der Zeiger `tiere[i]` hat, sondern auf welche Art von Objekt er zeigt!

Bemerkungen

- Diese Eigenschaft nennt man *Polymorphie*
- Die Polymorphie kann nur bei Zeigern ihre Wirkung zeigen.
- Erst, wenn man in einer Klassenhierarchie eine Methode *überschreibt*, greift die Polymorphie.
- Welche der überschriebenen Methode nun aufgerufen wird, wird erst zur Laufzeit entschieden[9].
- Compiler-intern wird eine sog. *Virtual method table* angelegt, mit Hilfe dieser das Programm dann die richtige Methode *findet*.

⊖`polymorphie.java`

⊙**Achtung** Bitte nicht das Überschreiben einer Methode mit der Überladung von Methoden verwechseln!

[9] Dies nennt man *Dynamic Binding*

finale Methoden

Will man die Polymorphie für eine Methode abschalten, kann man im Prototypen das Schlüsselwort `final` anfügen. Zwar kann man die Methode in abgeleiteten Klassen weiterhin überladen - ein Shadowing ist aber schon nicht mehr erlaubt!

```
1  class Haustier {
2    public final void gibLaut() {}
3  }
4  class Hund extends Haustier {
5  public void gibLaut() { // FEHLER
6    System.out.println("wuff!");
7  }}
8  class Katze extends Haustier {
9  public void gibLaut(String str) { //
       Ueberladen: OK
10 System.out.println("miau: " + str); // Eine
       sprechende Katze!
11 }}
```

Abstrakte/rein-virtuelle Methoden

Braucht man in der Basisklasse eine polymorphe Methode, die man aber nicht implementieren möchte/kann - vielleicht weil sie noch keinen Sinn macht - dann kann man sie als abstrakt definieren. Eine solche Methode nennt man *abstrakt* oder *rein virtuell*. Dies ist z.B. in der Klasse Haustier angebracht:

```
1 abstract class Haustier
2 {
3   public abstract void gibLaut();
4   //nun ist gibLaut rein virtuell
5   ...
6 }
```

Abstrakte/rein virtuelle Methoden

- Eine Klasse mit mindestens einer rein virtuellen Methode heißt *abstrakte Klasse*
- Diese Klasse muss mit `abstract` gekennzeichnet werden!
- Man kann keine Instanz einer abstrakten Klasse erzeugen.
- Wird in der abgeleiteten Klasse nicht jede vererbte rein virtuelle Methode implementiert, so wird diese automatisch zur abstrakten Klasse, und muss auch entsprechend gekennzeichnet werden!

Wer bin ich nun überhaupt?

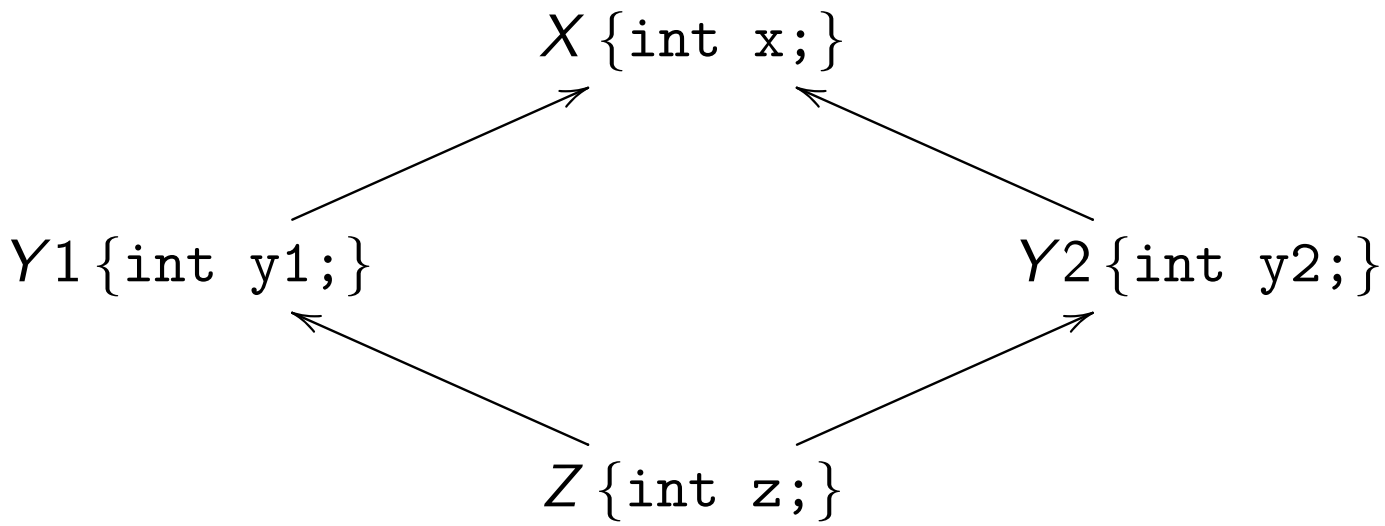
Sollten wir einmal einen Zeiger bekommen, möchten wir auch wissen, von welchem Typ er überhaupt ist! Damit wir Einsicht in das Virtual Method Table bekommen, gibt es noch den Operator *instanceof*

```
1 void streicheln(Haustier h) {
2   if(h instanceof Katze || h instanceof Hund)
3     System.out.println("*streichel*");
4   else
5     System.out.println("Kann dieses Tier nicht
      streicheln");
6 }
```

⟶instanceof.java

Mehrfachvererbung

Betrachten wir folgende Klassenhierarchie:



mit `class Z extends Y1, Y2;` . Dann enthält `Z` die eigens definierte Membervariable z und die geerbten Member $\{y1, x\}$ von `Y1` und $\{y2, x\}$ von `Y2`. Das ist problematisch, da für das Objekt Z instanz; nun der Aufruf `instanz.x;` mehrdeutig ist! **Deswegen ist diese Mehrfachvererbung in Java nicht möglich!**

Interfaces

Um dennoch Mehrfachableitung zu ermöglichen gibt es in Java spezielle Klassen: Die *Interfaces*. Ein Interface kennzeichnet sich aus, dass:

- keine Variablen erlaubt sind - nur Konstanten
- nur abstrakte Methoden erlaubt sind
- es also auch keinen Konstruktor/Destruktor hat

Implementierung von Interfaces

Mehrere Interfaces können mit dem `implements` (statt `extends`) Schlüsselwort implementiert (statt abgeleitet werden):

Syntax `class` *Klasse* `implements` *Interface0, Interface1* { ... }

Bemerkung Implementiert man nicht alle abstrakten Methoden der Interfaces, so wird die Klasse abstrakt!

`interfaces.java`

Ableitung eines Interfaces

Man kann nicht nur Interfaces mehrfachimplementieren, sondern auch mehrfachableiten:

✓**Syntax** `interface` *MeinInterface* `extends` *Interface0, Interface1* { ... }

Klassen, die dann `MeinInterface` implementieren wollen, müssen alle Methoden von `MeinInterface, Interface0` und `Interface1` definieren, um nicht-abstrakt zu sein.

Zugriffsspezifizierer für Interfaces

Ein Interface hat dieselben Zugriffsspezifizierer wie eine Klasse:

- `public` Interfaces sind aus allen Packages auffindbar.
- Interfaces ohne `Zugriffsspezifizierer` sind nur innerhalb ihres Packages gültig.

Methoden und Attribute eines Interfaces haben aber weder `protected` noch `private` Spezifizierer:

- Auf `public`e Element kann man von überall zugreifen, wenn man auf das Interface zugreifen kann.
- Hat ein Element *kein Schlüsselwort*, so kann man es nur innerhalb des aktuellen Paketes ansprechen.

Geschachtelte Klassen

Man kann neben Attribute und Methoden auch Klassen in eine Klasse schreiben:

```
1 class Aussen {
2   class Innen {
3   }
4 }
```

Hier nennt man die Klasse `Innen` eine geschachtelte Klasse von `Aussen`. Es gibt nun zwei verschiedene Subtypen von geschachtelten Klassen:

- Geschachtelte statische Klassen
- Innere Klassen

und zwei verschiedene Subtypen von lokal definierten Klassen:

- Lokale Klassen
- Anonyme Klassen

Zugriffsspezifizierer für geschachtelte Klassen

Wird eine Klasse geschachtelt, so kann es die gleichen Zugriffsspezifizierer wie Attribute/Methoden haben:

- Auf `publice` geschachtelte Klassen kann man genauso zugreifen, wie auf ihre äußere Klasse.
- geschachtelte `protected` Klassen können nur
  - von der äußeren Klasse
  - und von deren abgeleiteten Klassen angesprochen werden.
- `private` geschachtelte Klassen können nur von ihrer äußeren Klasse benutzt werden.
- Hat eine geschachtelte Klasse *kein Schlüsselwort*, so kann man sie nur innerhalb des aktuellen Paketes ansprechen.

Geschachtelte statische Klassen

- Verhält sich genauso wie eine “normale” Klasse
- Man kann aber obige Zugriffsspezifizierer verwenden $\mapsto$ bessere Kapselung
- Die Geschachtelte statische Klassen können nur statische Elemente der äußeren Klasse aufrufen, die aber auch `private` sein können.

```
1  class Aussen {
2    static class Innen {
3    }
4  }
```

⟶`nested_static.java`

Innere Klasse

- Eine Instanz einer inneren Klasse hat **immer** einen Bezug zu einer Instanz einer äußeren Klasse $\mapsto$ Die innere Klasse kann nicht für sich existieren.
- Innerhalb der inneren Klasse kann man mit `Innen.this` den `this`-Zeiger der inneren Instanz, mit `Aussen.this` den `this`-Zeiger der äußeren Instanz ansprechen.
- Um außerhalb der äußeren Klasse eine Instanz einer inneren Klasse anzulegen, muss man den `new`-Operator der äußeren Instanz aufrufen.

```
1  class Aussen {
2    class Innen {
3    }
4  }
5  Aussen aussen = new Aussen();
6  Innen innen = aussen.new Innen();
```

nested_inner.java

lokale Klassen

Legt man innerhalb einer Methode eine Klasse an, so spricht man von einer lokalen Klasse

- Lokale Klassen einer (statischen) Methode können auf alle (statischen) Elemente der äußeren Klasse zugreifen.
- Lokale Klassen können auch auf alle bereits lokal definierten Konstanten zugreifen.
- Lokale Klassen haben **keinen** Zugriffsspezifizierer

```
1 class Aussen {
2   void foo() {
3     class Innen {
4     }
5   }
6 }
```

nested_local.java

anonyme Klassen

Anonyme Klassen sind spezielle lokale Klassen.

- Sie haben keinen Bezeichner.
- Sie erzeugen bei ihrer Definition genau ein Objekt.
- Sie sind darauf beschränkt, entweder eine Schnittstelle zu implementieren oder eine Klasse zu erweitern.
- Sie haben keine `static` Member

```
1 void foo() {
2   KlasseOderInterface i = new
      KlasseOderInterface() {
3   // Definition der anonymen Klasse
4   };
5 }
```

⟶nested_anonym.java

Definition von "Member"

Wir wollen nun die Definition eines *Member* verallgemeinern: Ein Member einer Klasse ist:

- Ein Attribut (Membervariable)
- Eine Methode
- Eine geschachtelte Klasse

Wrapper-Klassen

Jeder elementare Datentyp besitzt ein Äquivalent als Klasse:

- Diese Klassen heißen *Wrapper-Klassen*
- Für jeden elementaren Datentyp gibt es eine Wrapperklasse.
- Jede dieser Klasse erweitert die abstrakte Klasse `Number`
- Die Klasse `Number` stellt Konvertierungen in alle elementaren Datentypen zur Verfügung.
- Für einen elmentaren Datentyp `TT` kann man `TT TTValue();` aufrufen, um von einer `Number` einen elementaren Datentyp zurückzubekommen.

Tabelle der Wrapper-Klassen

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

Autoboxing

Die Wrapperklassen haben die zusätzliche Eigenschaft, dass sie arithmetische Operationen wie Addition, etc. ausführen können. Dies bezeichnet man als *Auto-boxing*.

```
1 Integer i = new Integer(2);
2 System.out.println(++i); // das geht nicht mit
      einer "normalen" Klasse!
3 int i = Integer.intValue();
```

Zudem bietet jede Wrapperklasse das Auslesen von Zahlen aus einem String:

```
1 double d = Double.parseDouble("1.34");
```

autoboxing.java

Wählbare Genauigkeit

Vom `Number`-Interface haben sich zwei Klassen abgeleitet, die sich gut für numerische Rechnungen eignen:

- `BigInteger` speichert Ganzzahlen mit nahezu unbegrenzten Wertebereich.
- `BigDecimal` speichert Gleitkommazahlen mit nahezu unbegrenzter Genauigkeit.
- Beide Klassen
  - haben einen Konstruktor, der eine Zahl aus einem String liest.
  - Mit der statischen Methode `valueOf` kann man aus einer Zahl ein Objekt erstellen.
  - stellen neben den arithmetischen Operatoren (`add`, `divide`, `subtract`, `multiply`) auch viele Operatoren der `Math`-Klasse (`abs`, `pow`) bereit.
- `BigInteger` kann in eine `BigDecimal` umgeformt werden.
- `BigDecimal` stellt mit `movePointLeft` bzw. `movePointRight` zusätzlich Methoden bereit, um die Kommastelle zu verschieben.
- Intern arbeiten beide Klassen mit Arrays, nur dass `BigDecimal` zusätzlich die Zehnerpotenz in einer `int`-Variable abspeichert.

⟶`bignumbers.java`

Strikte Dezimalzahlen

Java verwendet beim Rechnen mit Gleitkommazahlen eine bessere Methode als der Standard.

- Der Standard IEEE-754 beschreibt das Verhalten von `float` und `double`.
- Dieses Verhalten ist bei den meisten handelsüblichen Rechnern veraltet. Ihre Prozessoren haben das Rechnen mit Gleitpunktzahlen besser optimiert.
- Java verwendet deswegen standardmäßig **nicht** den IEEE-Standard beim Rechnern.
- Deswegen sind die Rechenergebnisse meist genauer, sind aber plattformabhängig.
- Ist eine Verwendung des Standards erwünscht, kann man dies mit dem `strictfp` Schlüsselwort vor Methoden oder Klassen erreichen.
- Mit `strictfp` bekommt man dan auf allen Systemen das selbe Ergebnis.
- Alle Funktionen der `Math` Klasse sind `strictfp`, da sie bis jetzt intern mit den Methoden von `StrictMath` identisch sind.

Generics

Unter einem *Generic* versteht man die Generalisierung einer Javaklasse auf einen oder mehrere Klassentypen T, V, Z.

✓**Syntax** `class` *Klasse*`<T,V,Z>` `...`

- Jeder Buchstabe T, V, Z ist hierbei ein Platzhalter für einen Klassennamen.
- Die Platzhalter werden durch Komma voneinender getrennt.
- Man kann innerhalb der Klasse mit T, V und Z so arbeiten, als wären sie Klassenbezeichner.

Will man später ein Objekt von dieser generischen Klasse erzeugen, muss man alle Typen spezifizieren:

♫**Beispiel**

Klasse`<Integer,Double,String> objekt = new`
Klasse`<Integer,Double,String>();`

Man spricht auch oft von *Templates* - die Spitzenklammern werden als *Templateklammern* bezeichnet.

Eigenschaften des T I

Vom T kann man bei dieser Definition nur erwarten, dass es irgendeine Klasse ist, d.h. dass es von `Object` abgeleitet ist.
(!)**Achtung** Wir können nicht einmal ein Objekt vom Typ T erzeugen, weil wir nicht wissen, ob es überhaupt einen Konstruktor hat!
Man kann aber das T einfach einschränken:
(✓)**Syntax** `class` *Klasse*`<T extends` *Basisklasse*`>` `...`

- Als *Einschränkung* kann sowohl ein Interface als auch eine (abstrakte) Klasse verwendet werden.
- Nun kann man davon ausgehen, dass T zumindest die Eigenschaften der *Einschränkung* hat, und kann darauf zugreifen.

Eigenschaften des *T* II

Will man, dass *T* zusätzlich noch verschiedene Interfaces implementiert, so kann man diese nach der *Einschränkung*sklasse mit einem & anhängen:

✓ **Syntax** `class` *Klasse*`<T extends` *Einschränkung* `&` *Interface0* `&` *Interface1* `> ...`

→ `generics_class.java`

Zusätzlich gibt es noch den *Raw-Type*

- Man lässt bei der Instanz die Templateklammern weg
- Dann werden alle Templateklassen auf `Object` gesetzt

Generics-Joker

Für eine Klasse `class Foo<T>` würde

```
1 Foo<String> f = new Foo<String>(); // Ok
2 Foo<Object> g = f; // fehler, da Object != String!
```

einen Fehler verursachen:

- Auch wenn `Object` eine Superklasse von `String` ist, geht die Konvertierung nicht!
- Denn dann könnte g mit einfachen `Objecten` arbeiten und diese müssten für f nach `String` gecastet werden.
- Ein Ausweg bietet das Jokerzeichen ?
- Es ersetzt einfach den T-Templateparameter.

```
1 Foo<?> f = new Foo<?>(); // Ok
2 Foo<Object> g = f; // Ok, da Joker
```

Generics-Joker II

Falsch ist aber trotzdem:

```
1 Foo<?> f = new Foo<String>(); // Ok
2 Foo<Object> g = f;
3 // fehler, da Object != String!
```

Generische Methoden

- Methoden können generisch sein, obwohl es die Klasse nicht ist.
- Dazu schreibt man zwischen den Schlüsselwortspezifizierer und dem Rückgabewert die Templateparameter.

```
1  static <T> void foo(T a) {
2     ...
3  }
4  <T extends String> int g(T[] strings) {
5     ...
6  }
```

⟶generics_methoden.java

Generics-Joker einschränken

Innerhalb einer generischen Klasse/Methode für den Typ T kann man den Joker einschränken:

- `<? extends T>` ergibt ein Template, bei dem alle Typen erlaubt sind, die T erweitert.
- `<? super T>` ergibt ein Template, bei dem alle Typen erlaubt sind, von denen T ableitet.

Beispiel

- Man hat ein `interface Comparable<T>`, die für einen Datentyp T Vergleiche macht.
- eine `class Foo implements Comparable<Object>`, die sich mit allen Objekten vergleichen kann.
- Wir wollen eine Funktion schreiben, die vergleichbare Objekte braucht.

Generics-Joker einschränken II

```
1 static <T extends Comparable<T> >
2 void foo(T a) { ... }
```

würde für `Foo` nicht gehen, da Foo sich ja mit *jeder* Klasse vergleichen kann.

```
1 static <T extends Comparable<? super T> >
2 void foo(T a) { ... }
```

führt zum Ziel, da hier ein T verlangt wird, das

- `Comparable<? super T>` implementiert,
- d.h. eine Vergleichsmöglichkeit mit Objekten der gleichen Klasse oder Superklassen bietet.

Die Klasse Object

Jedes Objekt leitet automatisch von der Klasse `Object` ab, d.h. man erbt bereits alle Methoden von `Object` ohne jedes weitere Zutun! Die wichtigsten Methoden sind:

- `protected Object clone() throws CloneNotSupportedException` Erstellt eine Kopie vom Objekt
- `boolean equals(Object obj)` Überprüft, ob beide Objekte gleich sind
- `protected void finalize()` Destruktor des Objektes
- `String toString()` Wandelt das Objekt in einen String um

Achtung Leider ist es nicht so, dass die vordefinierte Methoden bereits unsere Klasse richtig behandeln - wir müssen diese oft überschreiben!

Der Zuweisungsoperator

Die Methode `clone()` wird in Java dazu verwendet, um Objekte einander zuzuweisen, sprich lax ausgedrückt "a:=b".

ⓘ **Achtung** Nicht mit der Zuweisung von Zeigern verwechseln!

- Die Methode `clone()` ist standardmäßig auf `protected` gesetzt. Wird diese aufgerufen, so wirft sie im Normalfall *immer* die Exception `CloneNotSupportedException`
- Um die Methode dennoch verwenden zu können, muss man das Dummy-Interface[10] `Cloneable` implementieren

[10] Dummy, weil man nichts implementieren muss

Shallow-Copy

Die geerbte `clone()`-Methode sollte man aber nur mit Vorsicht genießen!

- Sie kopiert sehr oberflächlich.
- Es wird nicht der Inhalt der Zeiger, sondern die Zeiger selbst kopiert!

Merksatz Bitte nicht verwenden, wenn man irgendwelche Zeiger als Attribut hat!

`clone_shallow.java`

Deep-Copy

Hat man beispielsweise ein Array in seiner Klasse, kommt man um eine Redefinierung dieser Methode nicht herum:

```
1  class Vektor implements Cloneable
2  {
3    int[] arr = new arr[50];
4    public Object clone() {
5      Vektor o = new Vektor();
6      for(int i = 0; i < 50; ++i)
7        o.arr[i] = this.arr[i];
8      return o;
9    }
10 }
```

clone_deep.java

Der Stringcastoperator

Der Stringcastoperator dient dazu, ein Objekt implizit zu einem String umzuwandeln.

- Die implizite Konvertierung geschieht durch `String+Object` oder auch innerhalb der Funktion `System.out.println(...);`
- Standardmäßig wird beim Umwandeln eines Objektes in einen String im String der Klassenname und die Speicheradresse abgespeichert.
- Will man informationsreichere Daten ausgeben lassen, darf man die Methode `public String toString();` überschreiben

⊖`toString.java`

Der Destruktor

Sobald man keine Referenz mehr auf ein Objekt hat, wird das Objekt automatisch vom Garbage Collector gelöscht. Will man seine Aufräumarbeiten vorher noch erledigen, z.B. das Schließen einer Datei, so bietet es sich an, einen Destruktor zu definieren. Ein Destruktor ist eine von `Objekt` geerbte Methode mit dem Prototypen.

```
1 protected void finalize() throws Throwable;
```

finalize.java

Wo ist mein Destruktor? Klasse B

Betrachten wir folgendes Beispiel:

```
1  class B {
2    public B() {
3      System.out.println("Basiskonstruktor");
4    }
5    protected void finalize() throws Throwable {
6      System.out.println("Basisdestruktor");
7    }
8  }
```

Wo ist mein Destruktor? Klasse A

```
1  class A extends B {
2    public A() {
3      System.out.println("Kindkonstruktor");
4    }
5    protected void finalize() throws Throwable {
6      System.out.println("Kinddestruktor");
7      // super.finalize(); sollte hier stehen!
8    }
9  }
10 ...
11 new A();
12 ...
```

Wo ist mein Destruktor? Die Lösung!

Damit auch der Destruktor von B aufgerufen wird, muss dieser im Destruktor von A aufgerufen werden!

```
1 protected void finalize() throws Throwable {
2   System.out.println("Kinddestruktor");
3   super.finalize(); // sollte hier stehen!
4 }
```

⊖destruktor_virtuell.java

Der Gleichheitsoperator

Hier geht man genauso vor:

- Man findet als Gleichheitsoperator die Methode `public boolean equals(Object obj)` vor.
- Diese Methode liefert nur `true` zurück, wenn beide Zeiger auf das gleiche Objekt zeigen!
- Will man also einen funktioniernden Vergleich, darf man wieder die Methode überschreiben.

⓵**Achtung** Überschreibt man diese Methode, sollte man - falls die Klasse gehasht werden soll - auch die Methode `int hashCode();` überschreiben.

⊖`equals.java`

Der Vergleichsoperator

Hat man eine Klasse, in der Objekte einer Ordnungsstruktur gehorchen, kann man das Interface `Comparable<T>` implementieren. Dann kann Java Arrays von diesen Objekten sortieren! Dazu muss man die Methode `public int compareTo(T o) throws ClassCastException` implementieren:

- Man prüft innerhalb der Methode zunächst, ob man sich mit dem `T o` vergleichen kann (bspw. durch den `instanceof`-Operator).
- Kann man sich mit den anderen Objekt nicht vergleichen, wirft man eine `ClassCastException`
- Falls man kleiner ist als das andere Objekt, gibt man -1 zurück.
- Bei Gleichheit gibt man 0 zurück.
- Ansonsten ist man größer und gibt 1 zurück.
  ⟶`compare.java`

Bemerkung Falls `compareTo` 0 zurückgibt, dann sollte auch `equals` ein `true` zurückgeben!

Sortieren

Hat man ein Array von `Comparable<T>`s, so kann man sich dieses sortieren lassen:

- Die Funktion `Arrays.sort(Object[] a)` sortiert das komplette Array.
- Die Funktion `Arrays.sort(Object[] a, int von, int bis)` sortiert nur einen Teilbereich des Arrays.

Einmal sortiert, kann man auch sehr leicht suchen:

- Sei X eine Klasse, die mit `compareTo` unserer Klasse kompatibel ist.
- Sei s nun ein Suchschlüssel von Typ X.
- Dann kann man mit der Funktion `int Arrays.binarySearch(array, s)` unser Array nach einem Objekt durchsuchen, das zu dem Schlüssel passt.
- Rückgabewert ist die position des gesuchten Objektes.

Im Gegensatz zur naiven linearen Suche ist die Suchzeit der binären Suche (binarySearch) logarithmisch! ⟶`comparesort.java`

Variadische Funktion

Variadische Funktionen sind Funktionen, die eine beliebige Anzahl an Argumente erlauben. Die variablen Parameter werden durch das Schreiben `Klasse... arrayvariable` in der Parameterliste identifiziert.

```
1  public static void printAll(int a, Object... objects) {
2    System.out.println(a);
3    for(Object o : objects)
4      System.out.print(o + " ");
5  }
6  printAll(2, "Hallo", 3, 'z'); printAll(4);
```

- Die variablen Parameter werden also in das `Object`-Array `objects` verpackt, s.d. wir in der Funktion statt mit Parametern mit einem Array arbeiten.
- Elementaren Datentypen werden via Autoboxing in Objekte konvertiert.

⊖`variadic.java`

Puffer?

?Was ist ein Puffer?

- Unter einem Puffer versteht man einen Zwischenspeicher.
- Wird bei Einlese/Auslese gefüllt.
- Erst wenn er voll ist, wird wirklich gelesen/gespeichert.
- Mit Puffer arbeitet man schneller als mit Direktzugriff.
- Dafür muss man den Puffer manchmal entleeren, damit man ein Ergebnis sofort sehen kann. $\Rightarrow$ Dies nennt man *flushen*

Man unterscheidet im übrigen zwischen

- Binärformat - Daten werden in Binärdarstellung gespeichert (platzsparend)
- Textformat - Daten werden lesbar gespeichert

Streams

Java verwendet sog. *Stream*klassen für die Eingabe/Ausgabe. Dazu gibt es die abstrakten Klassen

- `InputStream` - zum Einlesen von Daten
  - `int read()` liest ein einzelnes Byte als `int` Wert ein. -1 bedeutet Fehler.
  - `skip(long n)` überfliegt n-Bytes
  - `int available()` prüft, wieviele Bytes eingelesen werden können.
  - `close()` schließt den Stream
- `OutputStream` - zum Ausgeben von Daten
  - `write(int b)` schreibt ein Byte (in einer `int` gespeichert) in den Stream.
  - `flush()` flusht den Puffer.
  - `close()` schließt den Stream


`streams.java`

Einlesen im Textformat

- Gegeben sei ein `InputStream in;`
- Zeilenweise kann man bequem mit `BufferedReader` arbeiten:
  - Mit `BufferedReader r = new BufferedReader(new InputStreamReader(in));` wird ein neuer `BufferReader` angelegt.
  - Nun kann man mit dessen Methode `String readLine()` zeilenweise den Stream durchwandern

⟶`bufferedreader.java`

Einlesen mit Scanner

`Scanner` ist eine sehr elaborierte Klasse zum Auslesen von Daten im Textformat.

- Kann angewandt werden auf ein `File`, `InputStream`, `String`-Objekt
- Die Eingabe wird analysiert und in sogenannte *Tokens* mit einem *Delimiter* aufgetrennt
  - Der Delimiter ist ein Trennzeichen, und kann mit `useDelimiter(Pattern pattern)` gesetzt werden
  - `Pattern` ist eine Regex-Klasse (d.h. man kann insb. einen normalen String als Trennzeichen setzen)
  - Standardmäßig ist der Delimiter ein Leerzeichen, also so wie wir es bei der Standardeingabe erwarten
- Für einen Datentypen `TT` kann man
  - zunächst mit `boolean hasNextTT()` prüfen, ob der Scanner diesen einlesen kann
  - und dann mit `T nextTT()` das nächste Token einlesen lassen
- `String nextLine()` gibt die nächste Zeile zurück, `String next()` das nächste Token.
- `void close()` beendet den `Scanner`

Regex

Regex[11] sind komplexe Suchmuster. Eine kleine Einführung:

- `a+` das Zeichen "a" muss mindestens einmal vorkommen
- `b*` das Zeichen "b" darf beliebig oft vorkommen
- `.` ein beliebiges Zeichen
- `[ab]` entweder "a" oder "b"
- `[1-6]` sucht nach einer Zahl von 1 bis 6
- `dit|dot` sucht nach der Zeichenkette "dit" oder "dot"
- Eine erfolgreiche Suche wird als *Match* bezeichnet.

Die statische Methode `Pattern.compile(String regex)` liefert ein `Pattern`-Objekt für den `Scanner`.

⟶`scanner.java`

[11] auch als Reguläre Ausdrücke bekannt

Ausgabe im Textformat

Ergonomische Ausgabe bietet die Klasse `PrintStream`, die von `OutputStream` erbt.

- Für einen gegebenen `OutputStream out;` kann man einen `PrintStream p = new PrintStream(out);` anlegen.
- Die Methoden `print` und `println` sind für alle elementaren Datentypen überladen.
- `println` macht nach der Ausgabe eine neue Zeile, `print` nicht.

Daten im Binärformat

- Gegeben sei ein `InputStream in;` bzw. `OutputStream out;`
- Zum Einlesen eignet sich `DataInputStream`
  - Mit `DataInputStream r = new DataInputStream(in);` wird ein neuer `DataInputStream` angelegt.
  - Kann mit der Methode `FF readFF();` für `FF` := "elementarer Datentyp" jeden elementaren Datentyp aus dem Stream lesen.
- zum Ausgeben `DataOutputStream`
  - Mit `DataOutputStream r = new DataOutputStream(out);` wird ein neuer `DataOutputStream` angelegt.
  - Kann analog zu oben mit der Methode `void writeFF(FF element);` jeden elementaren Datentyp `FF` in den Stream schreiben.

Standardstreams

Die `System` stellt bereits einige Streams zur Verwendung bereit:

- `InputStream System.in` liest Eingaben von der Tastastur ein.
- `PrintStream System.out` ist die Standardausgabe am Bildschirm.
- `PrintStream System.err` ist zur Fehlerausgabe am Bildschirm vorhanden.

Dateien

Mit der Klasse `File` kann man Informationen über Dateien erlangen.

- Eine `File` wird über den Dateinamen angelegt.
- Zuerst kann man mit `boolean exists()` überprüfen, ob die Datei überhaupt existiert.
- `boolean isFile()` und `boolean isDirectory()` geben an, ob es sich um eine Datei oder einem Verzeichnis handelt.
- Eine Datei kann man mit `boolean createNewFile()` neu anlegen, ein Verzeichnis mit `boolean mkdirs()`.
- Dann sagt `boolean canRead()` bzw. `boolean canWrite()` ob die Datei lesbar bzw. beschreibbar ist.
- `boolean delete()` löscht die Datei

`dateien.java`

Filestreams

Mit `FileInputStream` und `FileOutputStream` können Dateien eingelesen und beschrieben werden.

- Den Konstruktoren übergibt man entweder den Dateinamen oder ein `File`-Objekt.
- Wichtig ist immer das `void close()` am Ende aufrufen, damit die Datei wieder freigegeben wird.
- Dazu kann man auch den Destruktor eines Filestreams aufrufen lassen.

⟶`filestreams.java`

I/O von Objekten

Bis jetzt haben wir nur von elementaren Datentypen gesprochen.

- Mit `ObjectOutputStream` und `ObjectInputStream` können Objekte aus- bzw. eingelesen werden.
  - `ObjectOutputStream` leitet von `OutputStream` ab.
  - `ObjectInputStream` leitet von `InputStream` ab.
- Dazu muss das Objekt aber wenigstens das Interface `Serializable` implementiert haben.
  - Das Interface `Serializable` ist ein Dummy-Interface.
  - Zusätzlich sollte man ein `private static final long serialVersionUID` Attribut haben - eine Art magische Zahl, mit der man die Klasse idenifizieren kann.
- Alles andere geht automatisch über die Methoden `writeObject` bzw. `readObject`!

⟶`serialize.java`

`transiente` Attribute

Manchmal will man aber ein Attribut nicht abspeichern/lesen, weil es "flüchtig" ist.

- Dazu gibt es das Schlüsselwort *transient*
- Dieses wird wieder vor der Attributdeklaration geschrieben.
- Dadurch erkennen `ObjectOutputStream` und `ObjectInputStream`, dass sie dieses Attribut ignorieren können.

`transient.java`

Volle Kontrolle mit `Externalizable`

Das Interface `Externalizable` erweitert `Serializable` und bietet die volle Kontrolle über Ein- und Ausgabe von Objekten.

- `void writeExternal(ObjectOutput out) throws IOException` wird für die Ausgabe aufgerufen. Der `ObjectOutput` kann mit
  - `writeObject(Object obj)` ein Objekt ausgeben.
  - `writeTT(TT v)` einen elementaren Datentypen `TT` ausgeben.
- `void readExternal(ObjectInput in)` wird für den Einlesevorgang aufgerufen.
  - Der `ObjectInput` kann mit
    - `Object readObject()` ein Objekt einlesen.
    - `TT readTT()` einen elementaren Datentypen `TT` einlesen.
  - Findet man nicht die passenden Werte vor, kann man eine `ClassNotFoundException` werfen.
- Schließlich braucht man einen Defaultkonstruktor, da der `ObjectInput` zunächst einmal das Objekt anlegen muss!

`external.java`

Das Interface Runnable

Das Interface `Runnable` deklariert eine Methode `void run();`

- Zunächst implementiert man das Interface `Runnable` in eine Klasse `Klasse`.
- In die Methode `void run();` schreibt man den Code, der in einem neuen Thread ausgeführt werden soll.
- Auf eine Instanz unserer Klasse kann man nun einen Thread anwenden:

```
1 Klasse klasse;
2 Thread t = new Thread(klasse, "hallo");
3 t.start();
```

konkurrent.java

Die Klasse `Thread`

- Threads erstellt man mit dem Konstruktor `Thread(Runnable target, String name);`
- Der Name dient dabei zum Kennzeichnen. Man kann mit der Methode `String getName()` diesen wieder abrufen.
- Die Methode `sleep` lässt den Thread für eine Zeitspanne anhalten.
- Mit `start` wird der Thread erst zum laufen gebracht.
- Ein Thread wird beendet, wenn er die `run`-Methode abgearbeitet hat.
- Auf den Thread kann man mit der `join` Methode auf seine Beendigung warten.
- Indirekt abwürgen kann man einen Thread mit der Methode `interrupt();`:
  - Mit `isInterrupted();` kann man testen, ob er abgewürgt werden soll.
  - Bei `sleep` wirft dann der Thread eine `InterruptedException`.

Synchronisation

Wird mit mehreren Threads gearbeitet, kann es passieren, dass diese zum selben Zeitpunkt auf die gleichen Variablen zugreifen, was meist zu Unfällen führt.

- Problematische Methoden können mit `synchronized` deklariert werden.
- Jede Klasse hat intern ein virtuelles Schloss, das hier eine Rolle spielt.
- Will ein Thread eine `synchronized` Methode betreten, so muss er solange warten, bis das Schloss aufgesperrt wird.
- Betritt ein Thread eine `synchronized` Methode, so wird das virtuelle Schloss versperrt.
- Verlässt der Thread diese Methode, so wird das Schloss wieder aufgesperrt.

Zwischen den Aufruf synchronisierter Methoden kann man ein `synchonized(objekt) { ... }` Block aufmachen, der im Scope das Schloss des Objektes `objekt` gesperrt hält. `synchronized.java`

Warten und Aufwachen

Jedes Objekt hat die beiden Methoden `void notify();` und `void wait();`.

- Wird `wait` innerhalb einer `synchronized` Methode aufgerufen, so wird das Schloss aufgesperrt, und der aktuelle Thread unterbrochen.
- Ein anderer Thread kann nur in einer anderen `synchronized` Methode mit `notify` den unterbrochenen Thread wieder aufwecken.
- Der aufgewachte Thread muss aber noch warten, bis das Schloss wieder offen ist, bevor er es wieder zusperren und seine Arbeit fortsetzen kann.

⟶`wakeup.java`

Schlüsselwort `volatile`

Greifen mehrere Threads auf ein Attribut unsynchron zu, sollte man diese unter `volatile` setzen.

- Mit `volatile` werden Optimierungen bzgl. dieser Variablen deaktiviert
- Erkennt der Compiler, dass man scheinbar die Variable nirgends manipuliert, so wird diese als Konstant angesehen.
  $\Rightarrow$ Schnellere Laufzeit
- Selbst bei einer Anweisung wie ++a kann es zu Konflikten zwischen den Threads kommen.
- Elementare Operationen, die gegen diese Art von Fehler geschützt sind, nennt man *atomar*.

⊖`volatile.java`

Das Entwurfsmuster Observer

Interessant bei asynchroner Programmierung ist die Klasse `Observable` und das Interface `Observer`.
Der `Observer` bekommt Signal zugesandt.

- Dazu muss er die Methode `void update(Observable o, Object arg)` implementieren.
- $\Rightarrow$ Ein `Observer` kann von mehreren `Observable` Signale erhalten.

Eine `Observable` kann ein Signal versenden.

- Dazu muss der `Observer` mit `addObserver(Observer o)` bei der `Observable` registriert werden.
- Das Signal kann nur gesendet werden, wenn wir `setChanged()` aufgerufen haben.
- `notifyObservers(Object arg)` sendet an alle registrierten `Observer` das Objekt `arg`.

beobachter.java

Pipes

Kommunikation zwischen Threads funktionieren über Pipes, die in Java als Streams implementiert sind.

- Man braucht sowohl `PipedOutputStream` und `PipedInputStream`
- Beide Instanzen müssen über den Konstruktor verbunden werden.
- Der Umgang ist dann wie bei normalen Streams möglich.
- Bitte in einem Thread nur eines der Enden der Pipe verwenden $\Rightarrow$ *Dead-Locks*

⟶`pipes.java`

Datenstrukturen

Das gewöhnliche Array bietet nicht viel Flexibilität: Es lässt sich nicht dynamisch vergrößern. In der Informatik gibt es verschiedene Strukturen, um Daten gut abzuspeichern:

- Arrays
- verkette Listen
- Binärbäume
- Hashtabellen

Daten können entweder

- als Set ungeordnet oder
- assoziativ als (Schlüssel,Wert)-Paar

abgespeichert werden.

Interface `Collection`

In Java gibt es das Interface `Collection<T>`, von der alle Datenstrukturen abgeleitet sind, die

- die Menge als Set unsortiert oder
- durch Verbund oder Nummerierung abspeichert

Eine `Collection<T>` hat folgene wichtige Methoden:

- Mit `boolean add(T o);` kann ein Objekt eingefügt werden.
- Über `boolean contains(T o);` wird geprüft, ob ein Objekt bereits in der Collection enthalten ist.
- `int size()` gibt die Anzahl der Elemente zurück.
- `boolean remove(T o);` entfernt das Objekt `o` aus der Collection, falls es vorhanden ist.

Interface List<T>

`List<T>` erweitert das Interface `Collection<T>` um die Durchnummerierung ihrer Einträge:

- Jedes Element einer `Liste` hat einen Index, s.d. man mit `T get(int index)` dieses Objekt abrufen kann.
- Das Pendant `int indexOf(T o)` gibt den Index für das gesuchte Objekt zurück
- Erneutes Setzen geht mit `T set(int index, T element)`

Die Klasse `Collections` stellt Algorithmen für Listen als statische Methoden bereit:

- `Collections.sort(List list)` sortiert eine Liste.
- `Collections.shuffle(List list)` mischt eine Liste durch (zufällige Permutation).
- `Collections.reverse(List list)` dreht die Reihenfolge um

Listen

Die wichtigsten `List<T>`-Datenstrukturen:

- `ArrayList<T>`
  - unsynchrone Klasse
  - verwendet intern ein Array
- `Vector<T>`
  - synchronisierte Klasse
  - verwendet intern ein Array
- `LinkedList<T>`
  - unsynchrone Klasse
  - verwendet intern Verkettungen

Interface Enumeration<T>

Eine `Enumeration<T>` ist ein abstraktes Interface zum Traversieren von Listen.

- Sobald man mit `boolean hasMoreElements()` geprüft hat, ob es noch ein Element in der Liste gibt,
- kann man mit `E nextElement()` dieses abfragen.

Bemerkung Die Klasse `Vector<T>` kann mit der Methode `Enumeration<E> elements()` zu einer `Enumeration` transformiert werden.

`aufzaehlung.java`

Interface Set

Eine `Set` ist eine Collection, die ihre Daten nicht indizieren. Es gibt folgende Implementierungen:

- `TreeSet<T>` speichert in einem Baum `Comparables` ab.
- `HashSet<T>` verwendet die Methoden `hashCode();` und `equals()` von `T`, um diese in einer Hashtabelle abzuspeichern

⊖`sets.java`

Der `int hashCode()`

Hashtabellen verwenden den `hashCode` zum Sortieren von Elementen der Klasse V.

- Eine Funktion $f : V \to \mathbb{R}$ heißt genau dann injektiv, wenn für alle $x, y \in V$ mit $f(x) = f(y)$ bereits folgt, dass $x = y$.
- Die Menge $\{(x, y) \in V : f(x) = f(y), x \neq y\}$ nennen wir *Gütemenge*.
- Wir bezeichnen den `int hashcode()` als *gute Hashfunktion*, falls sie eine sehr kleine Gütemenge hat.
- $\Rightarrow$ Hashfunktionen sollen möglichst injektiv sein!
- Beispiel: `Integer` $\xrightarrow{\sim}$ `int`
- In der Praxis verwendet man den `hashCode()` von bereits vorhandenen Klassen.

Datenstrukturen mit beschränken Zugriff

Es gibt auch Datenstrukturen, die dem Benutzer nur eine beschränkte Kontrolle ermöglichen:

- Vom `Stack<T>` kann nur man nur das zuletzt abgelegte Element abrufen (**LIFO**-Verfahren).
- Aus der `Queue<T>` kann man nur das zuerst abgelegte Element ansprechen (**FIFO**-Verfahren).
- Von der `PriorityQueue<T>` lässt sich nur das größte `Comparable` abrufen (**HIFO**-Verfahren).

Alle drei Datenstrukturen sind für die flüchtige Abspeicherung von Daten gedacht, da man beim Durchwandern die Objekte entfernen muss.

- Mit `T peek()` kann man sich zunächst das Objekt anschauen,
- Während man beim `Stack<T>` mit `T pop()` bzw. bei den beiden Schlangen mit `T poll()` das Objekt entfernt.

Iteratoren

Das Interface `Iterable<T>` kann mit `Iterator<T> iterator()` einen Iterator zurückgeben. Dieses Objekt stellt Methoden bereit, um eine Datenstruktur zu durchwandern:

- Da `Collection<T>` das Interface `Iterable<T>` ableitet, kann man bei allen Collections mit Iteratoren arbeiten.
- Mit `boolean hasNext()` kann man zunächst überprüfen, ob der Iterator noch ein Objekt in der Collection findet.
- Falls dem so ist, liefert `T next()` das nächste Objekt zurück.
- Will man dieses nun aus dem `Iterable<T>` entfernen, so kann man dies mit `void remove()` tun.

Listen bieten mit der Methode `ListIterator<T> listIterator()` einen erweiterten Iterator an, der auch rückwärts laufen und Objekte ersetzen kann.

⟶`iteratoren.java`

Map<K,V>

Map<K,V> ist eine neben Collection<T> weitere Schnittstelle für assoziative Datenstrukturen. Die folgenden wichtigen Methoden ermöglichen einen Umgang mit Map<K,V>:

- `V put(K key, V value)` speichert ein Objekt `value` mit seinem Schlüssel `key` in das Map ab. Gibt es bereits ein `V` mit dem Schlüssel `key` im Map, so wird dieses überschrieben.
- `V get(K key)` sucht nach dem `V` mit dem Schlüssel `key`.
- `boolean containsKey(K key)` bzw. `boolean containsValue(V value)` überprüft, ob bereits ein Schlüssel bzw. Wert in im Map vorhanden ist.
- `int size()` gibt die Anzahl der (Schlüssel,Wert)-Paare zurück.

Maps

Die wichtigsten `Map<K,V>`-Datenstrukturen:

- `TreeMap<K,V>`
  - Die Schlüssel (`K`-Objekte) müssen `Comparable<K>` implementieren.
  - Verwendet eine Baumstruktur zum Speichern der Schlüssel.
- `HashMap<K,V>`
  - Die Schlüssel müssen die Methoden
    - `hashCode();`
    - `equals()`

    implementieren.
  - Verwendet eine Hashtabelle zum Speichern der Schlüssel.

⟶`maps.java`

Welche `Collection` ist für mich die Richtige?

1. Wie will ich indizieren?
   - Über eine `Integer`? $\Rightarrow$ `List`
   - Über einen `Comparablen` Schlüssel? $\Rightarrow$ `TreeMap`
   - Ist der Schlüssel gut hashbar? $\Rightarrow$ `HashMap`
   - Ohne Schlüssel? $\Rightarrow$ `Set`
2. Sind Speicherverbrauch und Schnelligkeit ein Thema?
   - Wenn ja, brauchen wir eine Laufzeitanalyse.

Laufzeitanalyse

Eine Laufzeitanalyse ist eine Abschätzung der Komplexität einer Operation der Problemgröße $n \in \mathbb{N}$. Dazu verwenden wir das Landau-Symbol $\mathcal{O}$:

- Seien $f, g : \mathbb{N} \to \mathbb{R}_{>0}$ Maße für Komplexität.
- Wir sagen $f \in \mathcal{O}(g)$, falls $\limsup_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| < \infty$

Daraus leiten sich diese Beispiele ab:

- $\mathcal{O}(a) = \mathcal{O}(1)$ für ein festes $a \in \mathbb{R}$
- $\mathcal{O}\left(an^p + bn^{p-1} + cn^{p-2} + \ldots\right) = \mathcal{O}\left(n^p\right)$
- $\mathcal{O}\left(e^n + n^{1000}\right) = \mathcal{O}\left(e^n\right)$

Die Landau-Symbole geben nur eine relative Abschätzung bezüglich der Problemgröße.

Laufzeit der Datestrukturen

Die Problemgröße n ist die Anzahl der zu speichernden Elemente.

	`Vector`	`LinkedList`	`TreeMap`	`HashMap`
Indizierung	$\mathcal{O}(1)$	$\mathcal{O}(n)$	-	-
Einfügen/ Löschen am Anfang	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1) - \mathcal{O}(n)$
Einfügen/ Löschen mittig	$\mathcal{O}(n)$	$\mathcal{O}(1) +$ Suche	$\mathcal{O}(\log n)$	$\mathcal{O}(1) - \mathcal{O}(n)$
Einfügen/ Löschen am Ende	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1) - \mathcal{O}(n)$
Suche	$\mathcal{O}(\log n) - \mathcal{O}(n)$	$\mathcal{O}(\log n) - \mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1) - \mathcal{O}(n)$
Speicher-verschwen-dung	$\mathcal{O}(0)$	$\mathcal{O}(0)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$

Bemerkungen zur Laufzeit der Datestrukturen

Man beachte:

- `TreeSet`, `HashSet` sind nich auf der Liste - Eigenschaften sind genauso wie `TreeMap` bzw. `HashMap`, nur ohne Suche.
- `ArrayList` verhält sich genauso wie `Vector`.
- Für `Listen` gilt: Die lineare Suche braucht $\mathcal{O}(n)$, die Binärsuche $\mathcal{O}(\log n)$.
- `HashMap` ist von der Güte der `hashCode()`-Implementierung und dem freien Speicher abhängig.

Überklasse `Collections`

Die Klasse `Collections` ist eine Ansammlung statische Methoden für unterschiedliche `Collection<T>`-Datenstrukturen. Während

- `max(Collection<T> c)` sucht das Maximum (für `T implements Comparable`)
- `min(Collection<T> c)` sucht das Maximum (für `T implements Comparable`)

Es gibt auch statische Methoden, die eine unsynchrone Datenstruktur in eine Synchrone verwandeln:

- `Collection<T> synchronizedCollection(Collection<T> c)`
- `Map<K,V> synchronizedMap(Map<K,V> m)`
- `List<T> synchronizedList(List<T> list)`
- `Set<T> synchronizedSet(Set<T> s)`

Generische und Standarddatenstrukturen

Alle besprochene Datenstrukturen sind sowohl als Generic

- `Collection<T>`
- `Map<K,V>`

als auch als `Object`-Datenstrukturen (Raw-Type)

- `Collection`
- `Map`

definiert.

- Mit `Collection<T>` kann man den Datentyp einschränken, den man abspeichern will.
- Die Standardcollection `Collection` nimmt alle `Objecte` auf.

Achtung Man sollte immer mit Generics arbeiten, da die `Object`-Datenstrukturen viele Nachteile mit sich bringen!

Die foreach-Schleife

Bisher:

```
1  String[] array = new String[10];
2  for(String s : array)
3    System.out.println(s);
```

Für jede `Collection` kann man diese Schleife auch verwenden!

```
1  List<String> liste = new LinkedList();
2  for(String s : liste)
3    System.out.println(s);
```

Geschachtelte Iterationen

Wir haben Spielkarten `Karte(int rang, Farbe farbe);`, die wir in einer Liste generieren wollen.

```
1  List<Farbe> farbe = ...;
2  List<int> rang = ...;
3  List<Karte> karten = new ArrayList();
4  for(Iterator i = farbe.iterator(); i.hasNext(); )
5    for(Iterator j = rang.iterator(); j.hasNext(); )
6      karten.add(new Karte(i.next(), j.next() ));
```

Dieses Beispiel funktioniert *nicht*! Denn *i* wird in der inneren Schleife mit *j* erhöht, und nicht außerhalb! Besser:

```
1  for(Farbe i : farbe)
2    for(int j : rang)
3      karten.add(i,j);
```

Bitset

Eine spezielle Datenstruktur für `booleans` ist das `BitSet`.

- Das `BitSet` wird automatisch bei Bedarf erweitert.
- Mit `boolean get(int index)` und `void set(int index)` kann man das Boolean an der Stelle `index` ansprechen.
- Für `BitSet a, b;` gibt es bereits vorgefertigte Bit-Operationen:
  - $a|b \rightarrow$ `void or(BitSet set)`
  - $a\&b \rightarrow$ `void and(BitSet set)`
  - $\tilde{a} \rightarrow$ `void flip(int bitIndex)`
  - $a\hat{}b \rightarrow$ `void xor(BitSet set)`

Properties

Die Klasse `Properties` eignet sich zum Speichern von (Bezeichner, Wert)-Paaren. Ein (Bezeichner,Wert)-Paar nennt man *Eigenschaft*.

- `setProperty(String key, String value)` setzt wie bei einer `Map` eine Eigenschaft fest.
- `String getProperty(String key)` sucht nach dem Wert von `key` und gibt bei Misserfolg `null` zurück.
- `void storeToXML(OutputStream os, String comment);` speichert die `Properties` als XML ab, mit optionalen Kommentar.
- `void loadFromXML(InputStream in)` lädt `Properties` aus einem XML-Stream.

`eigenschaften.java`

Array-Algorithmen

Die Klasse `Arrays` stellt Algorithmen für Arrays durch statische Methoden bereit.

- Mit void sort(T[] a) kann man ein Array sortieren, für $T \in \{\text{Elementarer Datentyp}, \texttt{Comparable}\}$
- Anschließend kann das sortierte Array mit `int binarySearch(T[] a, T key)` durchsucht werden
  - Anders als die (normale) lineare Suche arbeitet `binarySearch` mit logarithmischer Laufzeit
  - Rückgabewert ist
    $$\begin{cases} >= 0 & \text{Key wurde gefunden} \\ < 0 & \text{negative Index, wo man den Key erwartet hätte} \end{cases}$$
- Will man zwei Arrays vom gleichen Typ vergleichen, kann man `boolean equals(T[] a1, T[] a2)` benutzen

Annotations

Unter einer Annotation versteht man eine Bemerkung, die an den Compiler gerichtet ist.

- Annotationen sind Bezeichner, die mit einem `@` beginnen.
- Es gibt vorgefertigte Annotationen und
- man kann Annotationen mit Kommentarfunktion selber schreiben
- Die meisten Annotationen dienen zur Einschränkung

Eigene Annotationen definieren

Eigene Annotationen werden durch eine Art `Interface` definiert:

```
1 public @interface MeineAnnotation {
2   int meineEigenschaft();
3   double meineAndereEigenschaft();
4   String nocheineAndere() default "
      Standardwert";
5 }
```

Die Annotationsdefinition unterscheidet sich aber von Interfaces durch:

- Es gibt nur Methoden
- keine Zugriffsspezifizierer
- man kann den Methoden einen Default-Rückgabewert mitgeben
- das Annotationsinterface kann wie ein Interface `public` oder ohne Spezifizierer sein

Annotationen verwenden

Man kann nun für `MeineAnnotation` verschiedene Elemente annotieren, indem man eine Annotation schreibt:

```
1  @MeineAnnotation(
2      meineEigenschaft = 1,
3      meineAndereEigenschaft = 2.5,
4      nocheineAndere = "Hallo, ich bin's"
5  )
6  public static void irgendwas(int nochwas) {
7      ...
8  }
```

annotationen.java

Abgekürzte Schreibweise

Für eine Dummy-Annotation `public @interface DummyAnnotation {}` kann man die Klammern auch weglassen:

```
1 @DummyAnnotation
2 public static void irgendwas(int nochwas) {
3     ...
4 }
```

Für eine Annotation mit einem Element kann man auch

```
1 public @interface EineAnnotation {
2   String str();
3 }
4 @EineAnnotation("Hallo")
5 public static void irgendwas(int nochwas) {
6     ...
7 }
```

schreiben.

Java-Annotations

Java kennt bereits ein paar Annotationen, die manchmal sogar hilfreich sind:

- `@Deprecated`
  - für Methode/Klasse, die man nicht mehr benutzen sollte.
  - bei Verwendung wird eine Warnung angezeigt.
- `@Override`
  - Wird zum Überschreiben von Methoden verwendet ($\rightarrow$ Polymorphie)
  - Wurde nicht richtig überschrieben (z.B. nur überladen), gibt es einen Compile-Fehler
- `@SuppressWarnings(String)`
  - Unterdrückt Warnungen vom Compiler
  - Mit "deprecation" werden `@Deprecated`-Warnungen ignoriert.
  - Mit "unchecked" werden Warnungen unterdrückt, die bei den alten `Object`-Datenstrukturen auftauchen.

Klasse System

Die Klasse `System` bietet eine Schnittstelle zu den wichtigsten Elementen der umgebenden "Welt". Alle Elemente sind `static`.

- `InputStream in` für Standardeingabe durch die Tastatur
- `PrintStream out` für Standardausgabe am Bildschirm
- `PrintStream err` für Fehlerausgabe am Bildschirm
- `long currentTimeMillis()` gibt die Anzahl der Millisekunden zurück, die seit dem 1. Januar 1970 GMT vergangen sind
- `long nanoTime()` gibt eine Zeit in Nanosekunden zurück - nur zur Differenzzeitmessung gedacht.
- `void exit(int status)` beendet das Programm sofort mit dem Status `status`
- `void gc()` aktiviert den GarbageCollector
- `Map<String,String> getenv()` gibt die Umgebungsvariablen zurück
- `Properties getProperties()` gibt Einstellungen der Windows Registry zurück

system.java

Klasse `String`

- Man kann Strings verketten: `String` + `String` $\rightarrow$ `String`, $a + b \mapsto$ "ab"
- Strings kann man über die statische `format`-Methode erzeugen.
- Mit `char charAt(int index)` kann man einen Buchstaben aus dem String lesen.
- Die Methode `int length()` gibt die Länge des Strings zurück.

(!) **Achtung** Bei einer Stringverkettung wird immer ein neuer `String` erzeugt $\Rightarrow$ Zeitaufwand!

Klasse StringBuilder

Ⓢ **Lösung** Deswegen gibt es die `StringBuilder` Klasse:

- `StringBuilder` verwaltet intern einen wachsenden String.
- Mit den `append`-Methoden kann man dem String wachsen lassen, indem man dieser Methode
  - einen elementaren Datentypen
  - einen String

  übergibt.
- Die `toString`-Methode spuckt dann den "gebauten" String zurück.

⊖ `stringbuilder.java`

Datum

Das Datum wird in Java durch die abstrakte Klasse `Calendar` gemanaged.

- Eine Implementierung ist eine Datumsangabe mit dem Gregorianische Kalender durch `GregorianCalendar`
- Der Defaultkonstruktur setzt den Kalender auf das aktuelle Datum.
- Mit der `set`-Methode kann das Datum verändert werden
- Mit `int get(int field)` kann ein Datenfeld aus dem Kalender entnommen werden.
- Ein `field` ist hierbei eine statisch definierte `int`-Konstante wie `WEEK_OF_YEAR`
- Mit `void add(int field, int amount)` kann man den Kalender durchwandern.

⟶`datum.java`

JavaDoc

Man kann aus Java-Quellcode eine Dokumentationsbeschreibung generieren. Dazu muss man folgende Kommentierungsgesetze wissen:

- Ein Dokumentationskommentar beginnt mit /** (statt mit /*)
- Man kann ziemlich alles in Java kommentieren: Klassen, Methoden, Variablen, Packages, ...
- Man schreibt immer *vor* den Element das Kommentar
- Der Beschreibungstext darf HTML-Elemente enthalten.

JavaDoc-Schlüsselwörter I

- `@author ich halt` setzt den Autor auf `ich halt`
- `@version 1.1` gibt die Versionsnummer an
- `@since 1.2` sagt, dass es das Element seit Version 1.2 gibt
- `@param Parameter Beschreibung` beschreibt den Parameter `Parameter` einer Mehtode/Funktion
- `@return Beschreibung` beschreibt den Returnwert
- `@exception Ausnahme Beschreibung` gibt an, wann die Exception `Ausnahme` geworfen wird.
- `@throws Ausnahme Beschreibung` ist ein Synonym zu `@exception`
- `@deprecated Begründung` gibt an, warum das Element `deprecated` ist
- `@serial Beschreibung` zeigt an, dass das Element serialisierbar ist

JavaDoc-Schlüsselwörter II

- `@see Verweis` verweist auf einen *Verweis*
- Man kann mit `@link Verweis` einen Verweis innerhalb der Beschreibung verwenden

Verweise in JavaDoc werden nach dem Schema

- `Package.Klasse` für Klassen/Interface/Enums
- `Package.Klasse#Methode(Parameterliste)` für Methoden
- `Package.Klasse#Attribut` für Attribute/geschachtelte Klassen

gemacht. Man kann auch Sachen einfach weglassen:

- `#Bar` für das Attribut `Bar` der einen Klasse
- `Foo` für die Klasse `Foo` im gleichen Package

Zusammenfassung der Schlüsselwörter I

`abstract`

- Methoden ohne Inhalt sind abstrakt
- Klassen mit abstrakten Methoden sind abstrakt

`assert`	dient zum Debuggen - wirft Exception, wenn Bedingung fehlschlägt
`boolean`	Boole'scher Datentyp
`break`	bricht die umgebende Schleife ab
`byte`	8-bit große Ganzzahl
`case`	ist ein Block einer `switch`-Anweisung
`catch`	Fängt eine `Throwable` ab
`char`	Buchstabendatentyp
`class`	Klasse

Zusammenfassung der Schlüsselwörter II

`const`	wird nicht verwendet
`continue`	Wandert sofort zum nächsten Schleifendurchlauf
`default`	- In einer `switch` Anweisung der Fall, der immer erfüllt ist - bei Annotationen der Standardwert
`do`	Anfang einer `do-while` Schleife
`double`	Gleitpunktzahldatentyp
`else`	zu `if` komplementäre bedingte Verzweigung
`enum`	ist eine spezielle Klasse für Aufzählungen
`extends`	Klasse erweitert Klasse, Interface erweitert Interface
`final`	konstante Attribute/Variablen, nicht ableitbare Methoden/Klassen

Zusammenfassung der Schlüsselwörter III

`finally`	optionaller Endblock eines `catch-try`-Blocks, der immer ausgeführt wird
`float`	Gleitpunktzahldatentyp
`for`	die `for`-Schleife
`goto`	wird nicht verwendet
`if`	bedingte Verzweigung
`implements`	Klasse erbt Interface
`import`	zum Inkludieren von Klassen aus anderen Packages
`instanceof`	Operator zur Klassen-Identifizierung eines Objektes
`interface`	spezielle Klasse für Mehrfachableitung und Abstraktion
`int`	Ganzzahltyp
`long`	Ganzzahltyp
`native`	Zum Aufrufen von Bibliotheken anderer Programmiersprachen - im Kurs nicht behandelt

Zusammenfassung der Schlüsselwörter IV

`new`	erstellt ein neues Objekt/Array
`package`	bezeichnet das Package, zu dem der Quellcode gehören soll
`private`	Zugriffsspezifizierer für Member
`protected`	Zugriffsspezifizierer für Member
`public`	Zugriffsspezifizierer für Klassen und Member
`return`	beendet die Methode und gibt ggf. einen Wert zurück
`short`	Ganzzahltyp
`static`	für Funktionen und Klassenvariablen
`strictfp`	Gleitpunktzahlarithmetik nach IEEE-Standard
`super`	

- Verweispräfix auf geerbte Variablen
- Aufruf eines Konstruktors der Basisklasse

Zusammenfassung der Schlüsselwörter V

`switch`	die `switch` Anweisung
`synchronized`	macht Klasse/Methode/Block thread-safe für das aktuelle Objekt
`this`	- Zeiger auf das aktuell behandelte Objekt - Aufruf eines weiteren Konstruktors
`throws`	Gibt an, dass eine Methode best. `Throwables` werfen kann
`throw`	Wirft ein `Throwable`
`transient`	ignoriert das Attribut beim Auslesen/Einlesen des Objektes
`try`	Einleitender `try-catch`-Block, in der `Throwables` geworfen werden dürfen
`void`	kein Rückgabewert

Zusammenfassung der Schlüsselwörter VI

`volatile`	deaktiviert Variablenoptimierung, die Threadarbeiten unsicher machen könnten
`while`	Eine `while`-Schleife

festgelegte Literale

Keine Schlüsselwörter sind:

`@Deprecated`, ...	Annotationen
`null`	Jeder Zeiger kann zum `null`-Zeiger werden. Bei Zugriff wird `NullPointerException` geworfen
`true`	Bool'scher Wert für Wahrheit
`false`	Bool'scher Wert für Falschaussage

Swing/AWT

Stellt die graphische Bibliotheken *AWT* und *Swing* zum sofortigen Verwenden bereit.

- Swing besteht aus den Bereichen
  - Widgets[12]
    - Komponenten
    - Container
- AWT[13] besteht aus
  - Layout
  - Listener
  - rudimentäre Widgets

$\Rightarrow$ Swing setzt auf AWT auf.

[12] window gadget - graphisches Element

[13] Abstact Tool Kit

Widgets

Viele vorgefertigte Widgets stehen in Swing zum Einsatz bereit:

`JLabel`	Konstanter Text/Grafikelement
`JTextField`	Texteingabefeld
`JSpinner`	Zahleneingabefeld
`JButten`	Knopf zum Drücken
`JCheckBox`	In dieses Feld kann man ein Häckchen setzen
`JList`	Bietet eine Liste an, aus der man einen oder mehrere Einträge auswählen kann.
`JComboBox`	Wie `JList`, nur im kompakten Design
`JPanel`	Containerwidget, in der weitere Widgets untergebracht werden können

Widget-Basisklassen

`java.awt.Component` ist die Wurzelklasse aller Widgets

- Definiert viele typische Eigenschaften eines Widgets wie Position, Größe
- Definiert Listener-Methoden zum Abfangen von Events
- Über `setSize(int width, int height)` kann man Weite und Höhe angeben
- Mit `setLocation(int x, int y)` gibt die Position relativ zum übergeordneten Widget an

`java.awt.Container` sind spezielle `Components`, die

- Widgets beherbergen können
- Diese Widgets müssen `Components` sein
- Mit der `add(Component comp);` Methode werden die zum `Container` hinzugefügt.
- Container können ein sog. Layout haben (später mehr)

Für Swing-Widgets gibt es die Klasse `JComponent`, die von `Container` ableitet.

Top-Level-Widgets

Unter einem Top-Level-Widget versteht man ein Widget, das nicht von einem anderen umgeben ist. Es gibt vier Arten von Top-Level-Widgets:

- `JFrame` - ein Fenster für Java-Applikationen
- `JWindow` wie ein `JFrame`, nur ohne Dekoration
- `JDialog` für Dialogfenster
- `JApplet` für Applets

Generell leitet man eine der drei Klassen ab und spezialisiert sie für seine Zwecke.

ⓘ **Achtung** Diese Widgets leiten *nicht* von `JComponent` ab.

Die Content Pane

- Jedes Top-Level-Widget hat eine sog. *Content Pane*.
- Dabei handelt es sich um eine Schicht, in der Widgets graphisch abgelegt werden können.
- Eine Contant Pane ist vom Typ `Container`
- Über der Methode `Container getContentPane()` erhält man das Content Pane
- Dem Content Pane kann man mit `setLayout` ein neues Layout mitgeben. Am Anfang übergeben wir immer `null`!

Man verwendet also *nicht* das Top-Level-Widget als Container, um Widgets darin unterzubringen!

Die Anwendung starten

Um eine Anwendung zu starten, muss ein `JFrame` erstellt werden.

- Das passiert am besten in der `main`-Routine
- Nun kann man aber nicht sofort eine `JFrame` erstellen.
- Man muss `SwingUtilities.invokeLater(Runnable r);` aufrufen.
- Das Erstellen passiert also in einer `Runnable`-Klasse
  - In der `run()`-Methode wird das Frame erzeugt

Am Besten lässt sich dies mit anonymen Klassen realisieren!

⊖`first.java`

Kleine Optimierungen

Zunächst ist nicht klar, was passieren soll, wenn das Frame geschlossen wird.

- Mit `frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);` wird die Anwendung dann beendet.
- Über die statische Methode `JFrame.setDefaultLookAndFeelDecorated(boolean b);` kann man den Style ändern
- Hat man alle Elemente eingefügt, kann man mit `frame.pack();` das Fenster auf die notwendige Größe anpassen

⊖`frame.java`

Listener

Der `Listener` ist eine Klasse, die Events von `Componenten` abfängt (lauscht).

- Jeder `Listener` ist durch ein Interface realisiert, das das Interface `EventListener` erweitert.
- Um selbst einen `Listener` zu schreiben, muss man nur ein Interface implementieren.
- Dann trägt man ihn im jeweiligen `Component` mit der `add*Listener(*Listener l);`-Methode ein.
- Es gibt spezielle `Listener`, die nicht für jede `Componente` geeignet sind, wie z.B. der `ActionListener`

Der `ActionListener`

Ein `ActionListener` fängt Aktionen wie das Drücken eines Knopfes ab.

- Implementiert werden muss die Methode `void actionPerformed(ActionEvent e)`
  - Über `Object e.getSource()` kann abgefragt werden, welches Objekt den Event gesandt hat
  - Der Listener kann also für mehrere Objekte gleichzeitig verwendet werden!
- Dieser Listener kann nur mit Widget verwendet werden, die die Methode `void addActionListener(ActionListener l)` haben

⟶`actionlistener.java`

JPanel

Widget, welches Widgets beinhalten kann.

- Ist ein `JComponent`
- `JComponenten` sind Swing-Widgets und `Container`
- Verwendet den `LayoutManager` `FlowLayout`
- Mit `setLayout(null)` können wir die Widgets des `JPanels` selbst ausrichten

ItemSelectable

`Interface` für Widgets, die selektiert werden können.

- Ein `ItemSelectable` kann entweder den Status *selektiert* oder *deselektiert* haben.
- Dieser Zustand kann vom Benutzer verändert werden.
- Bei einer Veränderung wird ein `ItemEvent` gesendet.
- Das Interface deklariert die Methoed `void addItemListener(ItemListener l)` zum Abfangen dieser Events

ItemListener

- Implementiert wird die Methode `public void itemStateChanged(ItemEvent e)` Der `ItemEvent` hält Informationen bereit:
- Mit `ItemSelectable getItemSelectable()` kann man den Auslöser des Events ermitteln.
- `int getStateChange()` liefert die Werte `SELECTED` oder `DESELECTED`.

JToggleButton

Ein Button, der gedrückt bleibt.

- Hat den Status selektiert (gedrückt) und deselektiert (losgelassen)
- Sendet beim Drücken ein `ItemEvent`.

⊖`toggleButton.java`

JCheckBox

Optionsfeld, in dem man ein Häkchen machen kann.

- Konstruktor kann Beschreibungstext übergeben werden.
- setSelected(boolean s) setzt/entfernt das Häkchen (Selektiert/Deselektiert das Element).
- isSelected() prüft, ob das Häkchen gesetzt wurde.
- Wird das Häkchen vom Benutzer verändert, wird der `ItemListener` aufgerufen.

⟶`checkBox.java`

JRadioButton

Optionsfeld, in dessen Gruppe nur ein `JRadioButton` selektiert sein kann.

- Verhält sich fast genau wie eine `JCheckBox`
- Für die Gruppierung gibt es die Klasse `ButtonGroup`:
  - Man erstellt eine `ButtonGroup` und
  - fügt die RadioButtons einzeln mit der `add()`-Methode zur `ButtonGroup`
  - kann auch `JToggleButtons` verwalten
- In jeder Gruppe kann immer nur ein Element selektiert sein.
- Ohne Gruppe ist das RadioButton mit der CheckBox identisch!

⊖`radioButton.java`

JComboBox

Kompakte Liste von Elementen zur Selektion.

- Man braucht zunächst einen `Vector` oder ein Array aus Objekten, die man in die Box eintragen will.
- Jedes Objekt muss die `String toString()`-Methode implementiert haben.
- Dem Konstruktor von `JComboBox` kann man dann diese Liste übergeben.
- `setSelectedItem(Object)` bzw. `setSelectedIndex(int)` legt fest, welches Objekt aus der Liste angezeigt werden soll.
- `Object getSelectedItem()` bzw. `int getSelectedIndex` gibt das selektierte Objekt bzw. dessen Index zurück.
- Die `JComboBox` ist auch eine Datenstruktur, in der man nachträglich Objekte eintragen/entfernen kann.

⟶`comboBox.java`

Layouts

- Bisher: Alle Widgets mit `setPosition` und `setLocation` in fixen Koordinaten ausgerichtet.
- Leider muss dabei auch das `JFrame` eine fixe Größe haben.
- Soll flexibler sein!

$\Rightarrow$ Es gibt das `LayoutManager`-Interface. Alle Implementationen kümmern sich um die automatische und dynamische Positionierung, Ausrichtung und Vergrösserung der Child-Widgets.

- Jeder Container besitzt bereits ein Layout
- Ein neues Layout können wir dem Container mit `setLayout(LayoutManager mgr)` übergeben
- `doLayout()` reorganisiert die Widgets

doLayout()

Die doLayout()-Methode eines LayoutManagers wird aufgerufen,

- wenn der Container zum ersten Mal angezeigt wird
- wenn beim Container revalidate() aufgerufen wird

- und falls der Container einem Top-Level-Widget als Content Pane gehört

- wenn das Widget pack() aufruft
- wenn das Widget seine Größe ändert

Kommunikation

Wie kommuniziert der `LayoutManager` mit den `Componenten`? Er frägt ab:

- `Dimension getMinimumSize()` gibt die minimale Größe zurück
- `Dimension getMaximumSize()` gibt die maximale Größe zurück
- `Dimension getPreferredSize()` gibt die optimale Größe zurück

Er verwendet:

- `setLocation(int x, int y)` zum Setzen der Position oder
- `setBounds(int x, int y, int width, int height)`

Schema F

Allgemeiner Ablauf im Konstruktur

1. Container erzeugen
2. optional passendes Layout erzeugen und dem Container übergeben
3. Componenten/Container dem Container hinzufügen
4. Container dem übergeordneten Container hinzufügen

Dieser Mechanismus ist rekursiv zu verstehen!

FlowLayout

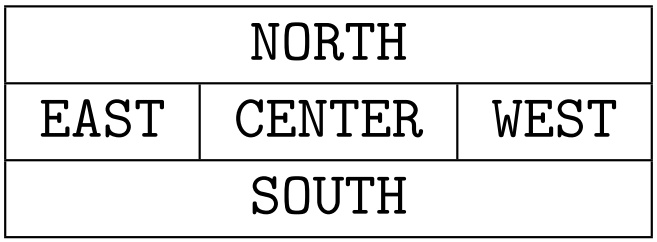
Legt die Widgets von links nach rechts, von oben nach unten an (ähnelt dem westlichen Schreiben). Der Konstruktor: `FlowLayout(int alignment, int horizontalSpace, int verticalSpace);`

- `alignment` ist die Ausrichtung:
  - `LEADING` für linksbündig
  - `CENTER` für zentriert (standard)
  - `TRAILING` für rechtsbündig
- `horizontalSpace` und `verticalSpace` geben den Abstand in Pixel zwischen den Widgets an

⟶`flowLayout.java`

BorderLayout

Das BorderLayout teilt die Widgets in 5 Bereiche auf:




```
1 Panel p = new Panel();
2 p.setLayout(new BorderLayout());
3 p.add(new Button("Mittig"), BorderLayout.
    CENTER);
```

borderLayout.java

BoxLayout

Richtet die Elemente *entweder* horizontal oder vertikal aus.

- `BoxLayout(Container target, int axis)` wobei `axis`
  - `X_AXIS` von links nach rechts (horizontal)
  - `Y_AXIS` von oben nach unten (vertikal)

  Die `Box`-Klasse kann Leerraum für uns schaffen:
- `Component Box.createVerticalGlue()` erstellt dynamischen vertikalen Leerraum
- `Component Box.createHorizontalGlue()` erstellt dynamischen vertikalen Leerraum
- Die geschachtelte Klasse `Box.Filler` fungiert Leerraumklasse:
  - Konstruktor `Box.Filler(Dimension min, Dimension pref, Dimension max)`
  - `min` ist die minimale, `pref` die gewünschte und `max` die maximale Größe.

boxLayout.java

JSeparator

Schafft eine sichtbare Trennlinie zwischen den Widgets.

- Konstruktoraufruf: `JSeparator(int orientation)`
- Orientation ist entweder
  - `SwingConstants.HORIZONTAL` oder
  - `SwingConstants.VERTICAL`

GridLayout

Eignet sich für eine tabellarische Ausrichtung.

- `GridLayout(int rows, int columns);` erstellt ein `GridLayout` mit `rows` Zeilen und `columns` Spalten.
- Die Elemente werden über `add` dann der Reihenfolge von links nach rechts, von oben nach unten der Tabelle hinzugefügt.
- Jede Zelle hat die gleiche Größe - nämlich die Größe des größten Widgets im GridLayout.

$\Rightarrow$ Gut geeignet, wenn alle Widgets die gleiche Größe haben.

⟶`gridLayout.java`

GridBagLayout

Sehr flexibles Layout für Tabellen.

- Jede Zeile kann ihre eigene Weite, jede Spalte ihre eigene Höhe haben.
- Widgets können sich über mehrere Spalten/Zeilen ausdehnen
- Steuerung erlangt man mit der `GridBagConstraints`-Struktur
- Dazu gibt es vom `Container` eine spezielle `add(Component comp, Object constraints)` Methode
- Jedes Widget im `GridBagLayout` hat also eine eigene `GridBagConstraints`

⊖`gridBagLayout.java`

GridBagConstraints

Man kann auf alle Attribute lesend und schreibend zugreifen:

- `gridx` und `gridy` setzen die Position
- `gridwidth` wieviele Spalten und `gridheight` wieviele Zeilen ein Widget umspannen soll
- `ipadx` und `ipady` geben den Abstand zu den anderen Zellen an
- `anchor` bestimmt die Ausrichtung des Widgets in seiner Zelle.

`NORTHWEST`	`NORTH`	`NORTHEAST`
`WEST`	`CENTER`	`EAST`
`SOUTHWEST`	`SOUTH`	`SOUTHEAST`

- `fill` lässt das Widget wachsend, wenn sich die Größe der Zelle ändert
  - `NONE` kein Wachstum
  - `HORIZONTAL` nur horizontales Wachsen
  - `VERTICAL` nur vertikales Wachsen
  - `BOTH` wachsen in alle Dimensionen

Jetzt mit Rekursion

Um Layouts zu verschachtelt, braucht man einen Zwischencontainer - am Besten geeignet ist `JPanel`:

a) Wir haben bereits einen Container mit passenden Layout (z.B. die Content Pane)

b) Wir erstellen uns ein `JPanel`

c) Mit `setLayout` legen wir unser Layout fest.

d) Wir fügen Widgets in das `JPanel` mit der `add`-Methode

e) Wir fügen das `JPanel` in unseren Container mit der `add`-Methode.

CardLayout

Zeigt immer ein `JPanel` aus einer Liste von `Componenten` an.

- Die Liste ist in der Reihenfolge geordnet, wie die `add()`-Befehle aufgerufen wurden.
- Es *muss* die add-Methode `add(Component c, String s);` aufgerufen werden!
- `first()` zeigt das erste Widget an
- `last()` zeigt das letzte Widget an
- `next()` geht die Liste um ein Widget weiter
- `previous()` zeigt das Widget zuvor an

⟶`cardLayout.java`

Zeichnen in AWT/Swing

Gezeichnet wird mit einem `Graphics`-Objekt.

- Damit kann man
  - in Bilder wie `Image`
  - direkt in AWT- und Swing-Widgets

  zeichnen.
- In Widgets kann man nicht einfach so zeichnen. $\Rightarrow$ Dafür gibt es den *Paint Event*.

Farbmodelle

Farben werden durch die Klasse `Color` bereitgestellt. Wir können eine Farbe mit verschiedenen Farbmodellen beschreiben:

- RGB(a) : rot, grün, blau, (α)
- HSV : hue, saturation, value[14]
- CMYK: cyan, magenta, yellow, black

Standardmäßig beschränkt man sich auf RGB/RGBa, also auf die Konstruktoren

- Color(float r, float g, float b, float a)
- Color(float r, float g, float b)

Die Klasse hält eine Anzahl an bereits vordefinierten statischen Farben bereit.

[14] deutsch: Farbton, Sättigung, Dunkelstufe

Die Klasse Canvas

Man kann bei jeder Componente das Zeichnen verändern.

- Speziell zum Zeichnen gibt es jedoch die Klasse Canvas
- Die Klasse ist eine minimale Erweiterung von Component

Zum Zeichnen gibt es die Methoden

- `public void paint(Graphics g)`
- `public void update(Graphics g)`
- `void repaint()`

paintAWT.java

Die `update`-Methode

Sobald die Fläche eines Widgets neu gezeichnet werden muss, ruft die JVMr die Methode `public void update(Graphics g)` auf. Das ist dann der Fall wenn

- das Widget zum ersten Mal angezeigt wird
- die Zeichenfläche des Widgets verdeckt wurde
- die Größe des Widget verändert wurde
- das umgebene Fenster minimiert bzw. maximiert wurde

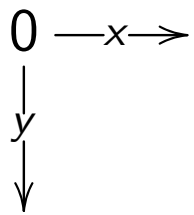
Generell ruft `update` die `paint`-Methode auf, s.d. wir diese Methode selten überschreiben werden.

Die `paint`-Methode I

Das übergebene Objekt vom Typ `Graphics` kann nun auf die Widget-Zeichenfläche zeichnen.

?Wie sieht nun die Zeichenfläche aus?

- Mit `Dimension super.getSize()` kann man die Größe der Zeichenfläche abrufen.
- Die y-Achse ist zum gewöhnlichen Koordinatensystem gespiegelt!
- Der Ursprung ist die Ecke links oben
- Der Punkt rechts unten ist `(super.getSize().width-1, super.getSize().height-1)`



Die `repaint`-Methode

- Die `paint`-Methode *nie direkt* aufrufen!
- Dazu gibt es die Methode `void repaint`:
  - `void repaint()` zum sofortigen Neuzeichnen
  - `void repaint(long tm)` zeichnet das Widget innerhalb `tm` Millisekunden neu
  - `void repaint(int x, int y, int width, int height)` zeichnet das Widget nur in diesen Koordinaten neu.
  - `void repaint(long tm, int x, int y, int width, int height)` Kombination aus obigen

Zeichnen mit JComponenten

Funktioniert fast genauso:

- Anstatt der `paint`-Methode überlädt man `paintComponent(Graphics g)`
- Innerhalb `paintComponent(Graphics g)` muss `super.paintComponent(g);` aufgerufen werden!
- Die `paint`-Methode zeichnet hier nämlich nicht nur das Widget, sondern auch sämtliche Widgetkinder und den Rand.
- Neben `repaint` gibt es dann noch `void paintImmediately(int x, int y, int w, int h)` zum sofortigen Neuzeichnen der angegebenen Fläche

Dazu gibt es noch die Transparenz, s.d. man `JComponenten` überlappen kann:

- `setOpaque(boolean)` setzt die Transparenz
- Ist unser Widget auf *opaque*, dann wird der Hintergrund gelöscht, ansonsten bleibt er unverändert.

Die Klasse `Toolkit`

Die Instanz `Toolkit.getDefaultToolkit` stellt uns nützliche Methoden bereit:

- Laden von Bildern
  - `Image createImage(String filename)` aus einer Bilddatei
  - `Image createImage(URL url)` aus dem Internet
- `sync();` zum sofortigen Neuzeichnen von gepufferten Widgets (später)

⊖`paintSwing.java`

Die Klasse `Image`

Eine `Image` ist (meist) eine bereits vorgefertigte Bilddatei.

- Der `Toolkit` kann uns Bilder aus Dateien erzeugen.
- Zusätzlich gibt es die Klasse `ImageIO` zum Ein- und Auslesen durch Streams.
- Mit der Methode `Graphics getGraphics()` können in das Bild zeichnen.

`images.java`

In Bilder zeichnen

Die Klasse `BufferedImage` leitet von `Image` ab und ist zum Zeichnen gedacht.

- Konstruktor: `BufferedImage(int width, int height, int farbmodell)`
- Das `farbmodell` ist meist `TYPE_INT_RGB` oder `TYPE_INT_ARGB` (mit Alpha)
- Direktes Zeichnen geht mit
  - `int getRGB(int x, int y)` liefert den RGBa-Wert des Pixels (x, y).
  - `void setRGB(int x, int y, int rgb)` setzt den RGBa-Wert.
  - Jede `Color` kann man mit `int getRGB()` in einen RGBa-Wert umwandeln.
- Zeichnen geht mit `Graphics2D createGraphics()`
- `Graphics2D` ist eine Subklasse von `Graphics` und stellt viele neue Methoden zur Verfügung.

Double Buffering

Wird das Bild oft verändert, können Artifakte am Bildschirm erscheinen:

- Das Widget wird gerade von der JVM gezeichnet, während der Paint-Event abgearbeitet wird.
- Um das zu vermeiden, gibt es das *Double Buffering*:
- Hier wird zuerst in ein Bild gezeichnet, und dann wird das fertige Bild mit dem Bildschirminhalt ausgetauscht.

Manuell

- Wir erstellen uns ein Bild, in das wir zeichnen.
- Im Paint-Event zeichnen wir lediglich das Bild in unser Widget.

Für `JComponent` lediglich

- `void setDoubleBuffered(boolean aFlag)` aktiviert/deaktiviert Double-Buffering
- Über `boolean isDoubleBuffered()` prüfen wir nach, ob Double-Buffering aktiviert ist.

Double Buffering II

Für Top-Level-Widgets und `Canvas` gibt es mehr Fine-Tuning:

- `createBufferStrategy(2);` setzt den Bildschirmpuffer auf 2 (doppelt).
- Wir erstellen uns zum Zeichnen ein `BufferStrategy bf = this.getBufferStrategy();`
- Mit `Graphics g = bf.getDrawGraphics();` zeichnen wir.
- Zum Abschluss müssen wir `g.dispose();` aufrufen.
- Anschließend bringen wir mit `bf.show();` das gezeichnete Bild auf den Bildschirm.
- Zum sofortigen Neuzeichnen rufen wir `Toolkit.getDefaultToolkit().sync();` auf.

doubleBuffering.java

Der Component Event

Wird ein Widget vergrößert, verkleinert, versteckt oder neu angezeigt, wird ein `ComponentEvent` Event generiert. Der `ComponentListener` lauscht auf:

- `componentShown(ComponentEvent e)` - Widget wird angezeigt
- `componentHidden` - Widget wird unsichtbar gemacht
- `componentMoved` - Widget wird bewegt
- `componentResized` - Widget wird vergrößert/verkleinert

Die Adapterklassen

Das dumme an Interface ist die Tatsache, dass man *alle* Methoden implementieren muss, damit wir eine nicht-abstrakte Klasse vor uns haben.

- Es gibt viele `Listener`-Interfaces mit sehr vielen Methoden.
- Oft wollen wir aber nur eine bestimmte Methode implementieren.
- Deswegen gibt es sogenannte *Adapterklassen*, die alle `Listener`-Methoden über Dummy-Methoden implementiert.
- Wir leiten also von einer Adapterklasse ab, und überschreiben nur die Methoden, die wir brauchen.
- Das Ganze funktioniert dann wieder über Polymorphie!

⟶`resizeEvent.java`

Das Lauschen mit der Maus

Auf die Maus kann man über drei verschiedene Listener zugreifen:

- `MouseListener` für das Klicken und Verlassen/Betreten eines Widgets mit der Maus
- `MouseMotionListener` für das Bewegen der Maus innerhalb eines Widgets
- `MouseWheelEvent` für das Bewegen des Mausrades
- Für alle drei Listener gibt es die Adapterklasse `MouseAdapter`.

Jede `Component` besitzt die zugehörige `add???Listener`-Methode.

Der Mäuselauscher

Das Interface `MouseListener` möchte folgende Methoden implementiert haben:

- `mouseClicked`
  - Eine Maustaste wurde angeklickt, ohne dass die Maus währenddessen bewegt wurde.
- `mousePressed`
  - Eine Maustaste wird gedrückt (gehalten)
- `mouseReleased`
  - Die Maustaste wurde losgelassen.
  - Zuvor wurde mousePressed aufgerufen
  - Wird die Maus im Widget gedrückt und außerhalb losgelassen, bekommt immernoch unser Widget dieses Event!
- `mouseEntered`
  - Maus betritt das Widget
- `mouseExited`
  - Maus verlässt das Widget

Bewegungssensor

Will man der Maus nachspionieren, kann man sich das Interface `MouseMotionListener` implementieren und damit auf Mäusejagd gehen:

- `public void mouseDragged(MouseEvent e)` wird dann aufgerufen, wenn man mit der Maus nach dem Objekt gegriffen hat, und dieses herumziehen will (Wie beim Drag&Drop während dem Festhalten der Taste)
- `public void mouseMoved(MouseEvent e)` wird aufgerufen, wenn sich die Maus bewegt hat.
- `addMouseMotionListener(MouseMotionListener l)` fügt zu unserem Widget einen `MouseMotionListener` hinzu.

⊖`scribble_mousevent.java`

Der MouseEvent

Der MouseEvent enthält alle Informationen bezüglich des aktuellen Status der Maus.

- Die Methode `int getID()` liefert den Ereignistyp des Events zurück. Der Typ kann mit Konstanten verglichen werden, die innerhalb der Klasse MouseEvent definiert sind.
  - Die wichtigsten Typen sind `MOUSE_PRESSED`, `MOUSE_RELEASED`, `MOUSE_CLICKED`.
  - Falls es einer der obigen Typen sind, so kann man mit `public int getButton()` nachfragen, um welche Maustaste es sich handelt. Auch hier braucht man wieder Konstanten: `BUTTON1`, `BUTTON2`, `BUTTON3`, `NOBUTTON`.
- `int getX()` und `int getY()` liefet die Koordinaten der Maus.

Tastatureingaben

Der `KeyListener` wird aktiv, wenn eine Taste

- gedrückt wurde ⇒ `keyPressed`
- kurz angetippt wurde ⇒ `keyTyped`
  - Nur für Buchstaben, Ziffern oder Sonderzeichen (Unicode-Zeichen).
  - Wird nicht bei Pfeiltasten oder Funktionstasten (z.B. F1) aktiv.
  - Liefert einen Unicode Character
- losgelassen wurde ⇒ `keyReleased`

wurde.

ⓘ **Achtung** Nur Widgets, bei denen die Methode boolean isFocusable() den Wert `true` zurückliefert, können Tastatureingaben empfangen!

Das Tastaturereignis

Die Taste wird im `KeyEvent` gespeichert:

- `char getKeyChar()` gibt das Zeichen zurück
- `int getKeyCode()` gibt eine Zahl zurück, die man mit den vordefinierten Konstanten aus der Klasse vergleichen kann.
- `getKeyCode() == KeyEvent.VK_ESCAPE` überprüft, ob die Escape-Taste gedrückt wurde.

Es gibt wieder eine Dummyklasse `KeyAdapter`, die `KeyListener` über Dummy-Methoden implementiert.

⟶`scribble_keyevent.java`

Superklasse `InputEvent`

Maus- und Tastaturereignisse leiten von `InputEvent` ab.

- Die Klasse behandelt Sondertasten wie Ctrl, Alt und Maustasten.
- `int getModifiersEx()` gibt zurück, welche Taste gedrückt wurde.
- Die Tasten sind als Integerkonstanten innerhalb der Klasse auffindbar.

⊖`scribble_keyfilter.java`

Dialoge

Dialoge sind Fenster, die für eine kurze Abfrage/Benachrichtung aufgemacht werden und nach Eingabe sofort geschlossen werden.

- Ein modaler Dialog blockiert alle Fenster, s.d. man auf diese nicht zugreifen kann
- Ein nicht-modaler Dialog verhält sich wie ein normales Fenster.

Kurzdialoge

Die Klasse `JOptionPane` stellt Dialoge durch statische Methoden bereit:

- `showMessageDialog` zeigt eine Nachricht an.
- `showConfirmDialog` Ja/Nein Dialog
- `showOptionDialog` Dialog mit Auswahlmöglichkeiten
- `showInputDialog` Dialog zur Eingabe eines Strings

Kurzmitteilung

`showMessageDialog(Component parentComponent, Object message, String title, int messageType)`

- `parentComponent`: Fenster, das auf die Beendigung des Dialogs warten soll
- `message`: String, der im Dialog angezeigt werden soll
- `title`: String in der Titelleiste
- `messageType` Symbol des Dialogs. Kann sein:
  - `ERROR_MESSAGE`
  - `INFORMATION_MESSAGE`
  - `WARNING_MESSAGE`
  - `QUESTION_MESSAGE`
  - `PLAIN_MESSAGE`

Auswahldialoge

`showOptionDialog(Component parentComponent, Object message, String title, int optionType, int messageType, Icon icon, Object[] options, Object initialValue)`
bzw. `showInputDialog(Component parentComponent, Object message, String title, int messageType, Icon icon, Object[] selectionValues, Object initialSelectionValue)`

- `icon`: Symbol des Dialogs, kann `null` sein
- `selectionValues`: ein Array aus Strings, die man auswählen kann
- `initialSelectionValue`: vorausgewählter String

Vorgefertigte Dialoge

- `JFileChooser` öffnet Dateien zum Lesen/Schreiben.
  - Zum Einschränken auf bestimmte Dateiendungen gibt es `setFileFilter(FileFilter filter)`
  - Wir rufen auf:
    - `int showOpenDialog(Component parent)` zum Lesen
    - `int showSaveDialog(Component parent)` zum Schreiben
  - Wir prüfen den Rückgabewert auf `APPROVE_OPTION`
  - Dann können wir mit `File getSelectedFile()` auf die Datei zugreifen
- `JColorChooser` lässt eine Farbe auswählen
  - Einfach die statische Methode `Color showDialog(Component component, String title, Color initialColor)` aufrufen

Eigene Dialoge

Für eigene Dialoge leiten wir einfach die Klasse `JDialog` ab. Die Besonderheiten sind:

- `JDialog` ist ein Top-Level-Widget
- Es gibt Konstruktoren für modale und nicht-modale Dialoge
- Wir verwenden bei eigene Dialoge selten statische Methoden, sondern erstellen unseren Dialog direkt!
- Die Methode `setVisible(boolean b)` bringt dann unseren Dialog auf dem Bildschirm.

`dialoge.java`

Menüs

Top-Level-Widgets haben neben der *Content Pane* eine *Menu Bar*.

- Diese ist nicht automatisch verhanden, sondern muss erstellt werden!
- Man fügt sie mit `setJMenuBar(JMenuBar)` dem Top-Level-Widget hinzu.
- Die wichtigsten Klassen für Menüs:
  - `JMenuBar` ist die komplette Menüleiste
  - `JMenu` ein Menü innerhalb der Menüleiste
- `JMenu` kann folgende Elemente beinhalten:
  - `JMenuItem` normaler Menüeintrag
  - `JRadioButtonMenuItem` ein RadioButton
  - `JCheckBoxMenuItem` eine CheckBox
  - `JMenu` ein Untermenü
  - `JSeparator` eine Trennlinie

Die `AbstractAction`-Klasse

Hat man verschiedene Widgets mit gleichen Label und gleicher Funktion, wird die Klasse `AbstractAction` bedeutsam:

- Eine `AbstractAction` ist ein erweiterter `ActionListener`
- $\Rightarrow$ `void actionPerformed(ActionEvent e)` ist zu implementieren.
- Zusätzlich kann man Label, Schnelltaste, Beschreibung und Icon festlegen:
  - `void putValue(String key, Object value)` setzt eine dieser Eigenschaften.
  - Der key ist eines der Konstanten aus der Klasse `Action`
- Viele Widgets haben einen Konstrukter `Konstruktor(Action a)`.
- Übergibt man diesem unser `AbstractAction`-Objekt, so wird dieses automatisch als `ActionListener` registiert.

`scribble_actions.java`

JScrollPane

Bietet Scrollbalken für ein Widget an, falls der Inhalt zu groß ist.

- Konstruktor: `JScrollPane(Component v, int verticalPolicy, int horizontalPolicy)`
- `verticalPolicy` bzw. `horizontalPolicy` geben das Verhalten des vertikalen bzw. horizontalen Scrollbalkens an.
- Ihre Werte sind respektive (`FOO = VERTICAL` bzw. `FOO = HORIZONTAL`)
  - `FOO_SCROLLBAR_ALWAYS`
  - `FOO_SCROLLBAR_AS_NEEDED`
  - `FOO_SCROLLBAR_NEVER`

⟶`jscrollpane.java`

Multiple Document Interface

Fenster können sogar als Unterfenster in einen Desktop eines Top-Level-Widgets verfrachtet werden. Dazu gibt es drei Klassen:

- `JDesktopPane` ist ein Widget, stellt den Desktop graphisch dar, und beherbergt `JInternalFrames`
- `JInternalFrame` ist ein Unterfenster, das in ein `JDesktopPane` untergebracht werden kann.
- `DesktopManager` kann `JInternalFrames` minimieren, maximieren, etc.
- Zunächst erstellen wir einen `JDesktopPane` und fügen es als Widget in unser Top-Level-Widget
- Mit der Methode `add` legen wir `JInternalFrames` in das `JDesktopPane`
- `DesktopManager JDesktopPane.getDesktopManager()` liefert uns den Manager, mit dem wir die Unterfenster manipulieren können.

`scribble_mdi.java`

Threads in der Grafikprogrammierung

Ohne Konkurrenz friert ein Fenster bei einer langen Rechnung prompt ein.

- Man unterscheidet drei Typen von Threads in der GUI-Programmierung:
  - Den Anfangsthread, der die Anwendung in der main-Routine erstellt.
  - Den Event-Thread, der die Ereignisschleife durcharbeitet und Events an die Listener verteilt.
  - Arbeiter-Threads, die im Hintergrund laufen und zeitfressende Arbeiten erledigen
- Ziel ist es, die schweren Arbeiten des Event-Threads an die Arbeiter-Threads abzugeben, damit die Anwendung flüssig läuft!

Aufteilen der Arbeit

Der Anfangsthread

- Muss `SwingUtilities.invokeLater` aufrufen, um die Anwendung zu erstellen.
- Er kann nicht direkt die Anwendung starten, denn er ist *nicht* der Event-Thread.
- $\Rightarrow$ Die Funktion `SwingUtilities.invokeLater` erweckt den Event-Thread zum Leben.

Was der Event-Thread nicht abgeben kann/darf:

- Zeichnen mit dem `Graphics`-Objekt
- Alle AWT-Swing Methoden, die nicht thread-safe sind
- Selbst ein Aufruf der Methode `JComponent.setFont()` ist nicht thread-safe!

Die Klasse `SwingWorker<T,V>`

`SwingWorker<T,V>` erweitert `Runnable` um nützliche Eigenschaften.

- Anstatt `void run()` implementiert man die Methode `T doInBackground()`
  - Hier können nur schwerlastige Probleme bearbeitet werden.
  - $\Rightarrow$ Der Rückgabewert kann ausgewertet werden!
- Starten können wir den `SwingWorker` mit der Methode `execute()`
- Über `T get()` bekommen wir das Ergebnis - und müssen ggf. warten
- `T get(long timeout, TimeUnit unit)` wartet hingegen nur eine bestimmte Zeit lang

Über den Fortschritt informieren

- Über `setProgress(int progress)` können wir prozentual den Fortschritt bekanntgeben, $progress \in [0 - 100]$
- Damit wird ein `PropertyChangeEvent` an die `PropertyChangeListener` versandt.
- Über `addPropertyChangeListener` registieren wir einen solchen Listener.
- Zur Anzeige eignen sich `JProgressBar` als Widget sowie `ProgressMonitor` als Dialog.

Nachrichten übermitteln

Der `SwingWorker` kann sogar dem Event-Thread Befehle übergeben:

- Zunächst rufen wir im `doBackground()` die Methode `publish(V... chunks)` auf.
- Dann werden die `chunks` temporär gespeichert.
- Der Event-Thread ruft damit dann die Methode `process(List<V> chunks)` auf.
- Wir überschreiben die `process`-Methode und können darin den Event-Thread nun Befehle erteilen!

⟶`swingworker.java`

Applets

Applets sind Java-Programme, die sich in einem Browser starten lassen.

- Dreh- und Angelpunkt ist die Widgetklasse `JApplet`.
- `JApplet` ist ein Top-Level-Widget.
- In der Applet-Programmierung haben wir *nur ein* Top-Level-Widget, das von `JApplet` ableitet.

Besonderheiten:

- Gestartet wird das Applet "automatisch" - nicht über `SwingUtilities.invokeLater`
- Für vollständige Funktionalität muss man verschiedene Methoden überladen.

Ansonsten ist das `JApplet` wie eine `Componente` hanzuhaben.

Zu Überschreiben

Automatisch aufgerufen wird

- `init()` sobald das Applet geladen wurde und zum Startet bereitsteht
- `start()` wenn das Applet gestartet werden soll
- `stop()` wenn das Applet gestoppt werden soll
- `destroy()` sobald das Applet beendet und sämtliche Ressources freigegeben werden sollen

Diese Methoden sind als Dummy-Methoden implementiert.

- Will man Animationen machen, muss man bereits diese Methoden überschreiben.

Besondere Methoden von `JApplet`

- `void play(URL url)` - Abspielen einer Audiodatei
- `Image getImage(URL url)` - Laden einer Bilddatei
- `void resize(int width, int height)` - Verändert die Größe des Applets
- `String getParameter(String name)` - Programmparameter abrufen, die mit dem HTML-Tag `<param name=''Parametername'' value=''Parameterwert''/>` gesetzt worden sind.

Starten von Applets

Applets lassen sich nicht wie Programme direkt starten.

- Applets können in der Eclipse-Entwicklungsumgebung gestartet werden.
- Das Konsolenprogramm `appletviewer` kann Applets ausführen.
- Browser können Applets anzeigen, wenn wir diese über HTML einbinden.

Einbinden in HTML

In HTML gibt es das <applet> Tag:

```
1 <applet
2     code="solitair/SolitairApplet.class"
3     width="100"
4     height="100"
5     alt="Solitaire Spiel"
6     archive="solitair.jar"
7     codebase="http://www.devwork.org/java/">
8 </applet>
```

Lädt das Applet `SolitairApplet.class` aus dem Package `solitair`, welches im jar-Archiv `solitair.jar` im Verzeichnis `http://www.devwork.org/java/` zufinden ist. Das Applet soll die Größe 100×100 haben. Falls es nicht angezeigt werden kann, soll der Alternativtext "`Solitaire Spiel`" erscheinen.

⊖`http://www.devwork.org/java/`