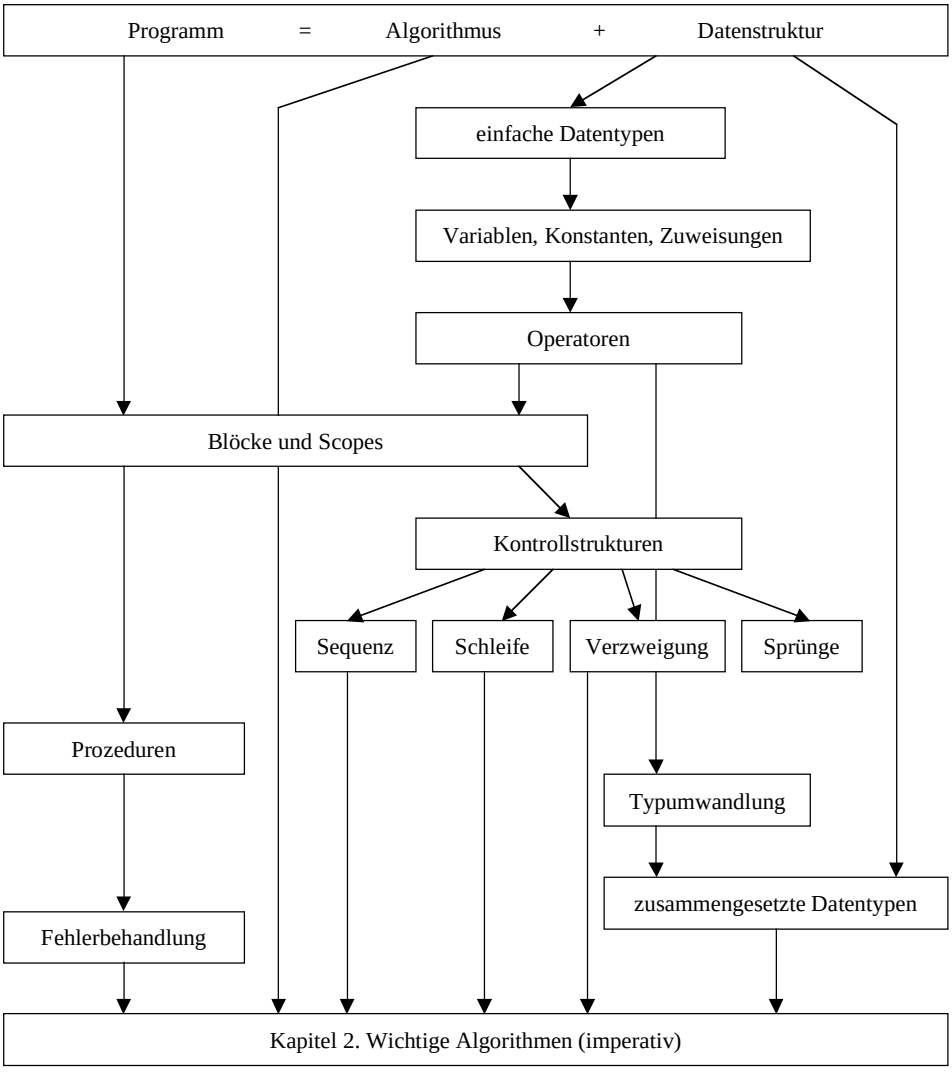
1 Sprachkonzepte I: Imperative Programmierung

Inhalt des Kapitels im Überblick:

- Einführung, imperativer Algorithmusbegriff
- Grundelemente von Algorithmen bzw. Programmen
- Einfache und zusammengesetzte Datentypen
- Variablen, Zuweisungen und Konstanten
- Ausdrücke und Operatoren
- Lebensdauer und Sichtbarkeit (Scopes)
- Kontrollstrukturen
- Prozeduren, Funktionen, Parameterübergabemechanismen
- Ausnahmen- und Fehlerbehandlung

„Fahrplan“ durch dieses Kapitel:



Gliederung des Kapitels

1.1 Einführung

Programmiersprachen:

- Ermöglichen formale Beschreibung von Problemlösungsverfahren, die auf einem Computer oder Computersystemen ausführbar sind.
- Bilden die Basis zur Entwicklung von Software und Betriebssytemen.

Programmentwicklung erfordert im Allgemeinen mindestens ein zweistufiges Verfahren:

- Entwurfsphase: Formulierung eines abstrakten Problemlösungsverfahrens in Form eines *Algorithmus*
- Codierungsphase: Transformation des Algorithmus in ein Programm; dabei Verwendung von Kontrollstrukturen und Datentypen

1.2 Imperatives Programm

- Darstellung als Folge von Anweisungen: A1; A2; A3; …
- Quellcode → Interpreter/Compiler → Programmausführung
- Programmausführung: Ausführung der Anweisungen; diese bewirken Effekte (Änderungen von Zuständen)

1.3 Klassischer Algorithmusbegriff

Der klassische Algorithmusbegriff

- abstrahiert von Rechnern und Programmiersprachen und
- ist imperativ.

(Imperativer) Algorithmus: Vorschrift zur Lösung einer Klasse gleichartiger Probleme, bestehend aus Einzelschritten.

Eigenschaften:

- Jeder Einzelschritt ist für die ausführende Instanz unmittelbar verständlich und ausführbar.
- Das Verfahren ist endlich beschreibbar.
- Praxisanforderungen:
    - Das Verfahren benötigt eine endliche Zeit; der Algorithmus terminiert.
    - Das Problem lässt sich mit endlichem Speicherplatzaufwand lösen.

Der nötige Aufwand an Betriebsmitteln, hier insbesondere Zeit (Laufzeit) und Raum (Speicherplatz), eines Algorithmus in *Abhängigkeit von der Problemgröße* wird abstrakt als *Komplexität des Algorithmus* bezeichnet.

Weitere Anforderungen an Algorithmen sind: *Sicherheit, Portabilität, Interoperabilität, Testbarkeit, leichte Änderbarkeit* und *Erweiterbarkeit* sowie *Wiederverwendbarkeit* und *Dokumentation*.

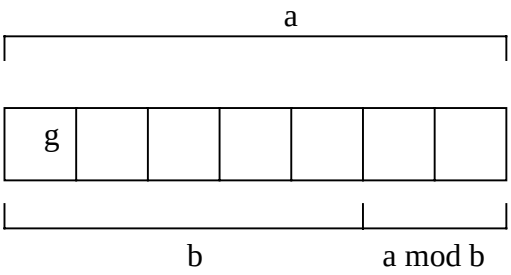
Man unterscheidet:

- Sequenzielle (sequencial) und nichtsequenzielle (concurrent, nebenläufige) Algorithmen,
- Deterministische und nicht-deterministische Algorithmen

Beispiel: Algorithmus ggt(a,b) zur Berechnung des größten gemeinsamen Teilers zweier ganzer Zahlen a und b.

```
static int getGGTOf(int a, int b) {
	// requires ((a > 0) && (b > 0)); ensures return > 0;
	int h;
	while (b != 0) {
		h = b;
		b = a % b; // % is the modulo operator
		a = h;
	}
	return a;
}
```

Wenn a < b ist, vertauscht der Algorithmus im ersten Durchlauf die beiden Zahlen. Angenommen, die Zahl a ist größer als b und g ist der ggT(a,b). Dann lässt sich nachvollziehen, dass g auch ggT(a mod b, b) ist.



Damit wird die Bildung des ggT(a,b) auf die Berechnung von ggt(a mod b, b) zurückgeführt. Durch Iteration erhält man immer kleinere Zahlenpaare und das Verfahren terminiert, wenn der Teilerrest, gleich 0 ist. Die andere Zahl a ist dann der ggt(a,b).

Beispiel: Algorithmus zur Berechnung der Fakultät n! einer natürlichen Zahl n.

```
static int getFactorialOf(int n) {
	int fact = 1;
	for (int i=1; i<=n; ++i) {
		fact = fact * i;
	}
	return fact;
}
```

1.4 Programmierparadigmen

Etwa 1970: strukturierte (prozedurale) Programmierung:

- Stepwise refinement
- Problem wird in kleine, leichter lösbare Probleme zerlegt und die Teillösungen werden zu einer Gesamtlösung zusammengesetzt (Top-Down-Ansatz)

- Für kleinere Probleme gut geeignet („Programmieren im Kleinen“)
- Weniger geeignet für den Entwurf und die Pflege größere Softwareprojekte

Neue Paradigmen: (Beispiele)

- objektorientierte Programmierung
- Logikprogrammierung

1.5 Grundelemente von Algorithmen bzw. Programmen

N. Wirth postuliert für die prozedurale Programmierung die Symbiose von Datenstruktur und Algorithmus:

Programm = Algorithmus + Datenstruktur

Grundelemente von Algorithmen sind:

- Variablen und Konstanten
- Ausdrücke
- Zuweisungen
- Kontrollstrukturen (Sequenzen, Verzweigungen, Schleifen/Iteratoren)
- Prozeduren und Funktionen

- (komplexe) Datentypen und (dynamische) Datenstrukturen

1.6 Einfache und zusammengesetzte Datentypen

Programme manipulieren Daten.

Ein Datum oder Wert (value) hat einen bestimmten Typ (type, mode).

Datentypen	
Einfache Datentypen • Wert ist unteilbar • Kann nur als Ganzes manipuliert werden • Beispiele: char, int, boolean	**Zusammengesetzte Datentypen** • Wert ist aus mehreren, einzeln manipulierbaren Teilwerten zusammengesetzt • Deren Typen können wieder einfach oder zusammengesetzt sein • Rekursives Bauprinzip • Erlaubt Einführung beliebig komplexer Typen • Beispiele: Mengen, Tupel

Beispiele:

1. Zusammengesetzter Datentyp termin: (Modula 2)

```
TYPE
       Termin =
              RECORD
                     datum: DATUM;
                     beschreibung: STRING;
              END;
```

23.04.01

2. Notation in Java: Zusammengesetzter Datentyp wird durch ein Objekt realisiert. (Vorgriff auf später)

```
import java.util.Date;

class Termin {
   public Date datum;
   public String beschreibung;
}

class TerminBeispiel {
   public static void main(String[] argv) {
       Termin termin = new Termin();
       termin.datum = new Date(); // Datum/Zeit zur Laufzeit
       termin.beschreibung = "Nichts tun!";
       System.out.println("termin.datum="+termin.datum);
       System.out.println("termin.beschreibung="+termin.beschreibung);
   }
}
```

Ausgabe des Programms:

```
termin.datum=Sun Apr 22 13:26:36 CEST 2001
termin.beschreibung=Nichts tun!
```

Einfache Datentypen

- heißen in Java auch *primitive Datentypen*.
- kennen Vergleichs*operatoren* und Zuweisungs*operatoren*

Primitve Datentypen in Java:

- ganze Zahlen: `byte, short, int, long`
- Gleitkommazahlen: `float, double`
- Logischer Datentyp: `boolean`
- Zeichen (Unicode, Zahlenwert ohne Vorzeichen): `char`

Anmerkung: Wie bekommt man in Java den Wertebereich heraus, den eine Variable eines bestimmten Typs umfassen kann?

Quellcode:

```
class PrintBounds {
 public static void main(String[] argv) {
  System.out.println("  byte=["+Byte.MIN_VALUE+".."+Byte.MAX_VALUE+"]");
  System.out.println(" short=["+Short.MIN_VALUE+".."+Short.MAX_VALUE+"]");
  System.out.println("   int=["+Integer.MIN_VALUE+".."+Integer.MAX_VALUE+"]");
  System.out.println("  long=["+Long.MIN_VALUE+".."+Long.MAX_VALUE+"]");
  System.out.println(" float=["+Float.MIN_VALUE+".."+Float.MAX_VALUE+"]");
  System.out.println("double=["+Double.MIN_VALUE+".."+Double.MAX_VALUE+"]");
 }
}
```

Ausgabe:

```
  byte=[-128..127]
 short=[-32768..32767]
   int=[-2147483648..2147483647]
  long=[-9223372036854775808..9223372036854775807]
 float=[1.4E-45..3.4028235E38]
double=[4.9E-324..1.7976931348623157E308]
```

1.7 Variablen, Zuweisungen und Konstanten

Wh: Ein Wert (oder Datum) hat einen bestimmten Typ.

Werte erhält man durch Auswertung eines *Ausdrucks*. (später dazu mehr)

Beispiel: `a = 12 * (3 + x);`

1.7.1 Variablen

Variablen (variables) bezeichnen Speicherplätze („Behälter“), in denen Werte eines Datentyps abgelegt werden.

Variablen

- besitzen einen Namen
- können während der Programmausführung verschiedene Werte annehmen
- Werte können überschrieben werden

Beispiel:

```
VAR x: INTEGER; // Modula 2
int x,a;        // Java
```

In typisierten Sprachen gilt:

- Variablen müssen vor der Verwendung definiert und typisiert werden.
- Vereinbarung (declaration) legt Namen (identifier) und Typ fest.
- Typangabe schränkt *statisch* die Werte ein, die eine Variable *dynamisch* annehmen kann.

Syntax (Java, vereinfacht):

```
Declaration: [final] Type VarDeclaration {, VarDeclaration};
Type: Identifier
VarDeclaration: Identifier [= Initializer]
Initializer: Expression
```

Programmierstil: Variablen sollten aussagekräftige Namen tragen und mit einem Kleinbuchstaben beginnen.

1.7.2 Zuweisung

Zuweisung (assignment) eines Wertes an eine Variable ist ein zusammengesetzter Ausdruck mit Effekt.

Syntax (Java, vereinfacht):

```
Assignment: Variable = Expression
```

Das `=` ist der Zuweisungsoperator.

Beispiel: (Java)

```
x = 4567;
a = x = 1;
```

Die Zuweisung `a = 12 * 3;` bewirkt den Effekt, dass die Variable `a` den Wert `36` zugewiesen bekommt.

(Anmerkung: Ein in anderen Sprachen üblicher Zuweisungsoperator ist `:=`)

Kontextbedingungen:

- Keine Namenskollisionen bei Typdeklaration
- Typkorrekte Benutzung bei Zuweisung
- Zuweisung vor erster Wertermittlung

Beispiele:

```
          { int i, j;
            char c = ' ', cc;
            final double PI = 3.1415; // Konstante, da final
            cc = c;
// Fehler     i = j;                  // weil j uninitialisiert
// Fehler     boolean j;              // Namenskollision
            double x = 2.7 * PI;
// Fehler     int n = n;              // n hat noch keinen Wert
            int n = (n = 2) * 3;
          }
```


1.7.3 Konstanten

Konstanten (constants) repräsentieren unveränderbare Werte, die einen bestimmten Typ besitzen.

Beispiel:

```
CONST K = 5;     // Modula
final int K = 5; // Java
```

Konstanten werden in Java mit dem Schlüsselwort `final` deklariert.

Programmierstil: Konstanten sollten aussagekräftige Namen tragen und in Großbuchstaben notiert sein.

Beispiele für in Java definierte **Bezeichner (Literale, literals)** für Konstanten:

	Typ	Beispiel
normale ganze Zahlen	`int`	`int i = 123;`
Anhängen von `L`	`long`	`long k = -1234L;`
Oktale Zahlen beginnen mit `0` (Null)		`int i = 011;`
Hexadezimale Zahlen beginnen mit `0x` (Null x)		`int i = 0xFF;`
Normale Zahlen mit Dezimalpunkt `.` oder `D` für Exponent	`double`	`double e = 2.7183;`
Anhängen von `F` oder `E` als Exponent	`float`	`float pi = 3.14F;`
Logische Konstante: `true` und `false`	`boolean`	`boolean f = true;`

1.8 Ausdrücke und Operatoren

Werte erhält man durch **Auswertung eines Ausdrucks** (expression).

Jeder Ausdruck hat einen *statischen* Typ. Die Auswertung (evaluation) eines Ausdrucks liefert i.d.R. einen Wert mit einem dynamischen Typ.

Einfache Ausdrücke (expressions)

- Wertbezeichner: `4567`
- *Variablen*namen: `x`
- *Kostanten*namen: `PI`

Zusammengesetzte Ausdrücke sind verbunden durch Operatoren.

Zusammengesetzte Ausdrücke bestehen aus

- Einfachen und zusammengesetzten Ausdrücken
- Prozedur/Funktionsaufrufen
- Klammern

Syntax (Java, vereinfacht):

```
Expression: Primary | Assignment | Expr {InfixOp Expr}
Primary: Literal | Identifier | ( Expression )
Expr: Primary [PostfixOp] | [PrefixOp] Primary | CastExpression
```

Operatoren in Java:

- Objekt-Operatoren: `new  .`
- Mathematische Operatoren: `+ - * / %`
- Mathematische Zuweisungen: `++ -- = *= /= %= += -=`
- Logische Operatoren: `< > <= >= == != ! && || instanceof`
- Bedingungsoperatoren: `? :`

- Bit-Operatoren: `<< >> >>> & | ~ ^`
- Bit-Zuweisungen: `<<= >>= >>>= &= |= ^=`
- String-Operatoren: `+ equals`

Regeln zur Auswertung von Ausdrücken:

1. *Operator precence.* Operatorvorrang („Bindungsstärke“) wird berücksichtigt, ebenso Klammerung mit ( ).
2. *Eager evaluation.* Erst werden die *Operanden* ausgewertet, dann werden die *Operationen* ausgeführt.
3. *Left-to-right-evaluation.* Bei dyadischen Operatoren (binary operators) wird erst der *linke* Operand ausgewertet, dann der *rechte*.

Beispiel: (LR-evaluation)

```
class Test {
  public static void main(String[] args) {
    int i = 2;
    int j = (i=3) * i;
    System.out.println(j);  // produziert die Ausgabe 9
  }
}
```

1.9 Blöcke und Scopes

Blöcke fassen Anweisungen (inkl. Deklarationen) zusammen.

Syntax (Java, vereinfacht):

```
Block: { {Declaration | Statement} }
Statement: Block | Assignment | Expression | ...
```

In Java fassen geschweifte Klammern einen Block ein.

Ausführung der eingeschlossenen Statements von links nach rechts und von oben nach unten.

Wichtige Eigenschaften bzgl. der deklarierten Namen:

- Blöcke können geschachtelt (nested) sein.
- Die in einem Block vereinbarten Namen (und dazugehörigen Variablen, Konstanten u.s.w.) heißen *lokal* zu diesem Block. Die in umschließenden Blöcken vereinbarten Namen heißen nichtlokal zu diesem Block.
- Der *Gültigkeisbereich* (scope) eines Namens erstreckt sich vom Ort seiner Vereinbarung bis zum Ende des zugehörigen Blocks.
- Der *Sichtbarkeitsbereich* einer Variablen (Konstante etc.)

Beispiele:

```
class ScopeTest {
  static int i = 3, n = 0;

  static void prozedur() {
    int k = i + 1;
    int n = 5;              // lokale Variable n
    i = n;                  // i = 5
  }
```

```
            public static void main(String[] args) {
              i *= i;                    // i = 9
              prozedur();
              n += 1;                    // n = 1
              i += 1;                    // i = 6
// Fehler       k = 0;                    // weil k uninitialisiert
            }
          }
```

1.10 Kontrollstrukturen

Datenstrukturen allein genügen nicht; um ein Programm zu realisieren, sind Ablauf- bzw. Kontrollstrukturen notwendig. In diesem Abschnitt werden Sequenzen, Verzweigungen, Schleifen/Iteratoren besprochen.

1.10.1 Sequenz

Unter Sequenz versteht man die Aneinanderreihung von Anweisungen, die in der Reihenfolge „von links nach rechts, von oben nach unten“ ausgeführt werden.

Sequenzen können in Blöcke (Abschnitt 1.9) zusammengefasst werden.

1.10.2 Verzweigung

Verzweigungen des Programmablaufs werden durch Alternative (if-then-else), Bedingungsoperator oder Fallauswahl (switch-case) realisiert.

1.10.2.1 Alternative (if-then-else)

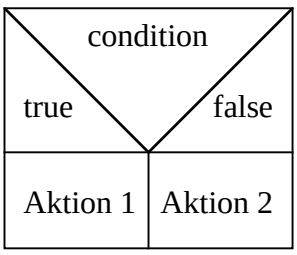
Syntax (Java, vereinfacht):

```
ConditionalStatement: if ( Expression ) Statement [else Statement]
```

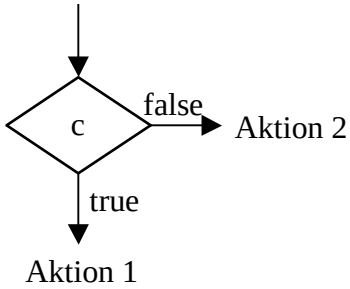
Die `Expression` muß vom Typ `boolean` sein.

Semantik: Wenn `Expression` wahr (`true`) ist, wird `Statement` ausgeführt. Andernfalls wird, sofern vorhanden, das `Statement` nach dem Schlüsselwort `else` abgearbeitet.

Die Alternative lässt sich graphisch in Form darstellen:



oder



Beispiele:

```
if (temperature > 0) {
       System.out.println("Keine Frostgefahr...");
       ...
}

if (temperature > 0); else System.out.println("Huh, ist das kalt hier!");

if (a)      x++;
else if (b) y++;
else        z++;
```

Programmierstil: Unschöne Notation:

```
if (a)
  if (b) x++;
else x--;
```

Das `else` gehört zum nächstmöglichen `if`, weshalb besser notiert wird:

```
if (a)
  if (b) x++;
  else x--;
```

30.4.2001

1.10.2.2 Bedingter Ausdruck

Syntax (Java, vereinfacht):

```
ConditionalOperator: Expression ? Expression : Expression
```

Die `Expression` muss vom Typ `boolean` sein. Die Semantik ist gleiche wie bei der Alternative mit dem Unterschied, dass als Alternativen keine Statements, sondern ein Ausdrücke stehen.

Beispiel: `z = y + (x >= 0 ? x : -x);`

Der Bedingungsoperator wird meinst geklammert verwendet, da er eine sehr geringe Bindungsstärke hat.

1.10.2.3 Fallauswahl (switch-case)

Syntax (Java, vereinfacht):

```
SwitchStatement: switch ( Expression ) { {SwitchLabel Statement} }
SwitchLabel: case ConstantExpression :    |
             default :
```

`ConstantExpression` ist ein konstanter Ausdruck. `ConstantExpression` und `Expression` müssen Typ-verträglich sein. `default` darf höchstens einmal auftreten.

Semantik: `Expression` wird ausgewertet und es erfolgt gemäß des erhaltenen Wertes ein Sprung an entsprechenden `case` (sofern vorhanden, sonst zu `default`, sonst hinter den gesamten Block.

Fallunterscheidung kann vorzeitig beendet werden mit `break` (siehe später).

Beispiel:

```
char letter;
...
switch (letter) {
       case 'e': {
              System.out.println("Hier was intelligentes tun!");
              ...
               break;
       }
       case 'x': {
              System.out.println("We will exit now!");
              System.exit(0);
               // break; unnoetig, da Ende durch System.exit(0);
       }
       default : System.out.println("Falsche Eingabe!");
}
```

1.10.3 Schleifen

Wenn Sequenzen mehrfach ausgeführt werden müssen, werden Schleifen eingesetzt. Eine Schleife arbeitet eine Sequenz solange zyklisch ab, solange eine **Schleifenbedingung** (loop control) gilt bzw. bis eine bestimmte Bedingung erfüllt ist.

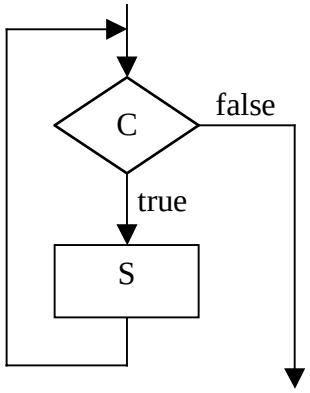
1.10.3.1 While und Do-While

Syntax (Java, vereinfacht):

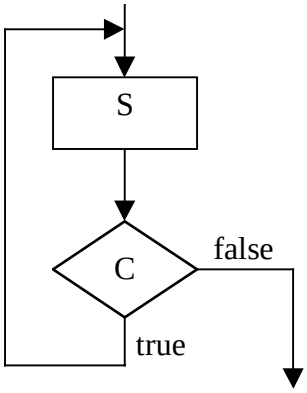
```
WhileStatement: while ( Expression ) Statement
DoStatement: do Statement while ( Expression ) ;
```

Semantik: Der Schleifenkörper S (loop body) wird abgearbeitet, *solange* der Bedingungsausdruck C wahr ist:

„pre-checked loop“:
while (C) S



„post-checked loop“:
do S while (C)



Beispiel für while, siehe ggt(a,b) in Abschnitt 1.3.

Beispiel für do-while: Das Programm `iskeywd` prüft, ob der Kommandozeilenparameter `<keyword>` ein Schlüsselwort von Java ist.

```
class iskeywd {
	public static void main(String[] argv) {
		// list of keywords defined in Java
		String[] keywords = {
			"abstract", "default", "if", "private", "this", "boolean",
			"do", "implements", "protected", "throw", "break",
			"double", "import", "public", "throws", "byte", "else",
			"instanceof", "return", "transient", "case", "extends",
			"int", "short", "try", "catch", "final", "interface",
			"static", "void", "char", "finally", "long", "strictfp",
			"volatile", "class", "float", "native", "super", "while",
			"const", "for", "new", "switch", "continue", "goto",
			"package", "synchronized"
		};
		// check for command line
		if (argv == null || argv.length != 1) {
			System.err.println("Usage: iskeywd <keyword>");
			System.exit(1);
		}
		// here we go
		String word = argv[0];
		boolean found = false;
		int i=0;
		do {
			if (keywords[i].equals(word))
				found = true;
			i++;
		} while ( i < keywords.length);
		// display result
		if (found)
			System.out.println(" ... yes, found!");
		else
			System.out.println(" ... no, not found!");
	}
}
```

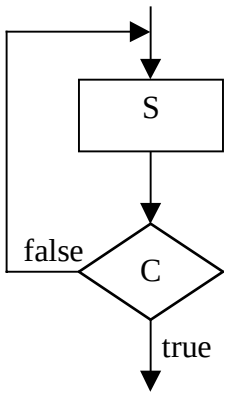
1.10.3.2 Repeat-until

Syntax (nicht in Java realisiert):

```
RepeatStatement: repeat Statement until ( Expression ) ;
```

Semantik: Bei repeat-until wird die Ausführung des Schleifenkörpers abgebrochen, *sobald* der Bedingungsausdruck wahr ist. *Beachte:* Repeat-until gehört nicht zum Sprachumfang von Java.

repeat S until (C)



1.10.3.3 Laufanweisung

Syntax (Java, vereinfacht):

```
ForStatement: for ( [ForInit] ; [Expression] ; [ForUpdate] ) Statement
ForInit: Declaration | Expression {, Expression}
ForUpdate: Expression {, Expression}
```

Semantik: Notation zur besseren Erklärbarkeit: `for (I;C;U) S`

`S` (Schleifenkörper) wird solange abgearbeitet, wie die Bedingung `C` erfüllt ist. `I` ist eine Initialisierung, `U` ein Ausdruck, der die Schleifenvariable erhöhen soll. Er wird nach `S` abgearbeitet.

Beispiele:

```
System.out.print("i =");
for(int i=0;i<10;i++,i++)
      System.out.print(" "+i);
System.out.println();
```

Ausgabe: `i = 0 2 4 6 8`

3.5.2001

```
System.out.print("c =");
for(char c='a';c<'e';c++)
      System.out.print(" "+c);
System.out.println();
```

Ausgabe: `c = a b c d`

Beachte: Die Termination von Schleifen muß sichergestellt sein!

Beispiel:

```
for(;;) {                          // forever (z.B. in einem Thread)
  if (runService == false) break;  // Abbruch über externe Variable
  ...                              // hier kommen die Aktionen hin
}
...                                // Hier gehts weiter nach break
```

1.10.3.4 Abbruch von Schleifen

Syntax (Java, vereinfacht):

```
BreakStatement: break [Label];
ContinueStatement: continue [Label];
Label: Identifier
```

Das Schlüsselwort `break` beendet die Schleife und setzt nach dem Schleifenkörper fort.

Das Schlüsselwort `continue` beendet den aktuellen Schleifendurchlauf vorzeitig und setzt mit `ForUpdate` (bei Laufanweisung, siehe Syntax) oder dem nächsten Schleifendurchlauf fort.

Beispiele:

```
for(int i=0;i<10;i++) {
       if (i == 5) continue;
       System.out.print(" "+i);
}
System.out.println();
```

Ausgabe: `0 1 2 3 4 6 7 8 9`

Achtung! Folgende Schleife terminiert nicht:

```
int i = 0;
while(i < 10) {
       if (i == 5) continue;     // Fehler! nicht terminierend!
       System.out.print(" "+i);
       i++;
}
System.out.println();
```

Richtig:

```
int i = 0;
while(i < 10) {
       if (i == 5) {
              i++;
              continue;
       }
       System.out.print(" "+i);
       i++;
}
System.out.println();
```

Beispiel mit Label:

Beachte: Ist eine Markierung angegeben, so bezieht sich der Abbruch auf den nächsten direkt umschließenden Block.

```
weiter: {
       System.out.println("Schleife beginnt");
       for(int i=0;i<10;i++) {
              if (i == 5) break weiter;
              System.out.print(" "+i);
       }
       System.out.println(" Schleife beendet. Text wird leider nie
ausgegeben!");
}
System.out.println(" Ende der break-mit-marke-Demo");
```

Ausgabe:

```
Schleife beginnt
 0 1 2 3 4 Ende der break-mit-marke-Demo
```

1.10.4 Sprünge

Die Anweisungen `break` und `continue` realisieren bereits Sprünge, allerdings mit eingeschränktem Sprungziel. In einigen Sprachen existiert eine Anweisung

```
GotoStatement: goto Label;          // nicht in Java !!!
```

mit dem an beliebige Ziele gesprungen werden kann.

Programmierstil: Die Folge der intensiven Anwendung von `goto` ist sog. „Spaghetti-Code“, weshalb diese Anweisung nach Möglichkeit vermieden werden sollte.

Im Java-Sprachumfang ist `goto` *nicht* enthalten.

1.11 Arbeiten mit einfachen Datentypen

In diesem Abschnitt soll es hauptsächlich um die Regeln bei der Typumwandlung und um Arrays, auf denen wir anschließend einfache Such- und Sortierverfahren realisieren.

1.11.1 Typumwandlung

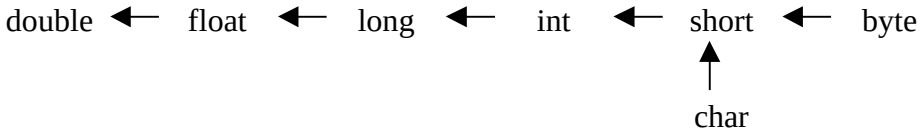
Typumwandlung (type conversion) kann implizit und explizit geschehen.

1.11.1.1 Implizite Typumwandlung

Wenn die Auswertung eines Ausdrucks einen Typ B liefert, aber ein Typ A erwartet wird, so wird versucht, eine (implizite) Typanpassung vorzunehmen.

Typumwandlung ist dann möglich, wenn eine Abbildung *f*: B → A existiert, mit der die Werte aus B nach A transformiert werden können.

Folgendes Diagramm ist eine Komposition von Abbildungen:



Implizite Typumwandlung ist nur entlang der Pfeile möglich (gerichteter Graph).

Beispiele:

```
            long l = -7;
            float x = l;
            x += 5;           // x = 2.0

// Fehler:  char zero = 0;    // nicht typkorrekt

            boolean  flag = true;
// Fehler:  int     iflag = flag;
            int     iflag = (flag?1:0);
```

1.11.1.2 Explizite Typumwandlung (explicit casting)

Die explizite Typumwandlung ermöglich die Typumwandlung auch entgegengesetzt der Pfeile im Diagramm von Abschnitt 1.11.1.1. Die Umwandlung findet erst zur Laufzeit statt.

Syntax (Java, vereinfacht):

```
CastExpression: ( Type ) Primary
```

Beispiel:

```
char space = ' ';
int  leer  = (int)space;    // leer = 32;
```

Gültige Varianten:

- angegebener Typ ist gleich dem des Primary
- angegebener Typ ist eine Erweiterung (widening, siehe Abschnitt 1.11.1.1)
- angebebener Typ ist eine Verengung (narrowing), **Beispiel:**

```
        double f = 3.1415;
        f = (int) f;      // f = 3.0, d.h fuehrt zu Informationsverlust
//          \_____/
//             3
```

1.11.2 Arbeiten mit Arrays

Arrays sind Datenstrukturen, in denen Werte in geordneter Form abgelegt werden können. Der Zugriff auf die Elemente eines Arrays erfolgt durch Indizes.

Deklaration:

Eindimensionales Array:

```
int a[];                // deklariert ein eindimensionales Feld aus Integer-
Werten
```

Mehrdimensionales Array:

```
String address[][];     // deklariert ein mehrdimensionales Feld aus Strings
```

Anlegen eines Feldes:

```
a[] = new int[MAX];     // MAX gibt die maximale Zahl an Elementen an

address[][] = new String[50][4];  // Semantik könnte sein:
                                  // address[...][0]: Vorname
                                  // address[...][1]: Name
                                  // address[...][2]: Strasse
                                  // address[...][3]: Ort
```

Zugriff auf Elemente: Hinter dem Namen des Arrays folgt der Index in eckigen Klammern:

```
int i,k;
...
a[i] = ...                  // ganz normale Wertzuweisung
address[k][0] = "Hannes";
```

7.5.2001

Syntax (Java, vereinfacht):

```
Type: ... | Identifier{[]}
ArrayAccess: Primary[Expression]
ArrayCreationExpression: new Type{Length}{[]}
Length: [Expression]
```

Beachte:

- Sei n die Länge eines Feldes, dann laufen die Indizes von 0 bis n-1.
- Wenn Index nicht aus [0, n-1] → „array index out of bounds“.
- Die Länge eines Feldes erhält man mit `name.length`
- Typ des Index muß mit `int` verträglich sein.
- Die Teilfelder mehrdimensionaler Arrays können verschiedene Längen haben:

  Beispiel [Partl, 24]:

```
double[][] tagesUmsatz = new double[12][];
int[] monatsLaenge =
      { 31,29,31,30,31,30,31,31,30,31,30,31 };  // Feld-Initialisierung
                                                // ueber Wertbezeichner
for (int monat=0; monat<tagesUmsatz.length; monat++) {
   tagesUmsatz[monat] = new double[ monatsLaenge[monat] ];
   for (int tag=0; tag<tagesUmsatz[monat].length; tag++) {
      tagesUmsatz[monat][tag] = 0.0;
   }
}
```

Beispiel: Sortieren eines Arrays von Integer-Zahlen

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

// Bubble-Sort auf einem Integer-Array:

```
static void bubbleSort() {
      for(int i=0;i<a.length;i++) {
            for(int j=0;j<a.length-1;j++)
                  if (a[j+1] < a[j]) {
                        // zwei benachbarte Elemente werden vertauscht,
                        // wenn das groessere vorne liegt
                        // swap(a[j+1],a[j])
                        int temp = a[j+1];
                        a[j+1]   = a[j];
                        a[j]     = temp;
                  }
      }
}
```

// Selection-Sort auf einem Integer-Array:

```
void selectionSort() {
      for(int i=0;i<a.length-1;i++) {
            // small auf den Index des ersten Vorkommens
            // des kleinsten verbleibenden Elements setzen
            int small = i;
            for(int j=i+1;j<a.length;j++)
                  if (a[j] < a[small]) small = j;
            // wenn man hier ankommt, ist small der Index des
            // ersten kleinsten elements in a[i..n]. Nun wird
            // a[small] mit a[i] vertauscht
            // swap(a[small],a[i])
            int temp = a[small];
            a[small] = a[i];
            a[i]     = temp;
      }
}
```

Bubble Sort und Selection Sort haben die Komplexität $O(n^2)$, d.h. quadratische Komplexität.

Funktionsweise von Bubble Sort: Durchgehen des Arrays und ggf. Vertauschen benachbarter Elemente.

Beispiel:

```
i       j       Array
-----------------------------------------
0       0       E       F       A       C       H
        1       E       F <--> A       C       H
        2       E       A       F <--> C       H
        3       E       A       C       F       H
1       0       E <--> A       C       F       H
        1       A       E <--> C       F       H
        2       A       C       E       F       H       // sortiert
        2       A       C       E       F       H       // ab hier keine Veränd. mehr
2       0
        1
        2
        3
3       0
        1
        2
        3
-----------------------------------------
```

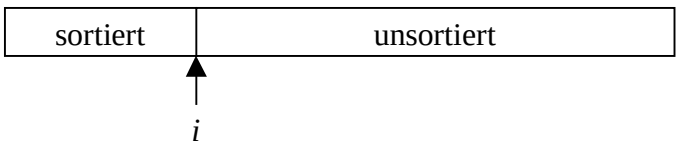
Optimierungsmöglichkeiten:

- vorzeitiger Abbruch, wenn in innerer Schleife kein Austausch mehr stattgefunden hat,
- Durchlaufrichtung abwechselnd ändern: Shaker-Sort. Nachdem eine „leichte Blase“ aufgestiegen ist, steigt eine „schwere Blase“ nach unten.

Idee von Selection Sort: Das kleinste Element wird mit dem untersten unsortierten Element vertauscht.

Beispiel:

```
i       j       Array
-----------------------------------------
0       1       E       F       A       C       H
        2       E       F       A       C       H
        3       E       F       A       C       H
        4       E       F       A       C       H
                ^               ^                       // vertausche E mit kleinstem
Element
1       2       A       F       E       C       H
        3       A       F       E       C       H
        4       A       F       E       C       H
                        ^               ^               // vertausche F mit kleinstem
Element
2       3       A       C       E       F       H       // sortiert
        4       A       C       E       F       H       // ab hier keine Veränderung
mehr
3       4       A       C       E       F       H
-----------------------------------------
```



Beachte: Die Zeitkomplexität (von Bubble Sort und Selection Sort) ist und bleibt $O(n^2)$.

1.12 Prozeduren, Funktionen

Es lohnt sich, zusammengehörende Programmfragmente in Prozeduren bzw. Funktionen zusammenzufassen.

- Erhöhung der Übersichtlichkeit/Lesbarkeit des Programms,
- Mehrfach aufrufbar (spart ggf. Programmspeicher),
- erlaubt Rekursion (siehe später).

Prozeduren (procedures) und Funktionen (functions) bezeichnen Unterprogramme, die eigenständige Programmblöcke mit einem eigenen Namen darstellen.

Begriffe:

- Unterprogramme werden auch als [sub]routine, subprogram, method, … bezeichnet.
- Liefert das Unterprogramm einen Wert zurück (return), so spricht man von einer *Funktion*, ansonsten von einer *Prozedur*.
- Argumente einer Prozedur/Funktion heißen *formale Parameter*.
- *Signatur* einer Prozedur/Funktion:

  Prozedurname + Folge der Typen der formalen Parameter

  Beispiel: ggT aus Abschnitt 1.3 hat die Signatur `ggt(int, int)`

10.5.2001

1.12.1 Aufbau

Syntax (Java, vereinfacht):

```
MethodDeclaration: MethodHeader MethodBody
MethodHeader: [Modifiers] ResultType MethodDeclarator
ResultType: Type | void
MethodDeclarator: Identifier ( [FormalArguments] )
FormalArguments: FormalArgument {, FormalArgument}
FormalArgument: Type Identifier
MethodBody: Block | ;
```

Beendigung der Unterprogrammausführung:

```
Statement: ... | ReturnStatement
ReturnStatement: return [Expression];
```

- Funktionen:
  - Rücksprung mit `return` und Wert der `Expression`
  - Typ von `Expression` muss verträglich mit Ergebnistyp sein
- Prozeduren:
  - entweder: statisches Prozedurende (und impliziter Rücksprung)bei Blockende
  - oder: Rücksprung mit `return;` (ohne Ausdruck).

Unterprogrammaufruf (procedure call, method invocation):

```
StatementExpression: ... | MethodInvocation
MethodInvocation: Identifier( [ActualArguments] )
ActualArguments: Expression {, Expression}
```

- *syntaktisch* gesehen entweder *Anweisung* (bei Prozeduren) oder *Ausdruck* (bei Funktionen)
- Beachte: In Java ist ein Unterprogrammaufruf stets eine `Expression` mit einem `ResultType`. Bei Prozeduren ist dies `void`.

Beispiele: (Unterprogrammaufruf)

```
1. y = x * ggT(a,b)

2. ...;
   ggT(a,b);     // nicht sinnvoll, da kein Effekt
   ...;

3. static void space() {
       System.out.println();
   }
   ...;
   space();      // bewirkt einen Zeilenvorschub
                 // (kein Effekt, aber "Seiteneffekt")
   ...;
```

Gültigkeitsbereich von Bezeichnern:

- Bezeichner, die in einem Unterprogramm (Block) definiert werden, gelten nur in diesem Block (lokale Variable)
- Gültigkeitsbereich der formalen Parameter: gesamtes Unterprogramm

1.12.2 Parameterübergabemechanismen

Welche Beziehung besteht zwischen einem *formalen* Parameter und dem zugehörigen *aktuellen* bzw. tatsächlichen Parameter?

Man unterscheidet:

- Wertparameter: Call-by-value
- Ergebnisparameter: Call-by-result
- Variablenparameter: Call-by-reference
- Namensparameter: Call-by-name

1.12.2.1 Wertparameter: Call-by-value

Formaler Parameter entspricht lokaler Variable und wird bei Prozeduraufruf mit Wert initialisiert.

Merke: In Java existieren nur Wertparameter.

Beispiel:

```
class PowerTest {
  static int getPowerOf(int a, int x) {
       // requires (x >= 0);
       int power = 1;
       while (x != 0) {
              power = power * a;
              x--;
       }
       return power;
  }
```

```
  public static void main(String[] argv) {
       int a = 3, x = 3;
       int y = getPowerOf(a,x);
       ...
  }
}
```

1.12.2.2 Ergebnisparameter: Call-by-result

Der Variable des formalen Parameters wird beim Rücksprung der Wert des aktuellen Parameters zugewiesen.

Merke: Call-by-result existiert nicht in Java, aber z.B. in Ada

Beispiel: Übergabe der *beiden* Lösungen von $x^2 + px + q = 0$: $x_{1,2} = -\frac{p}{2} \pm \sqrt{\frac{p^2}{4} - q}$

```
void solve (in float p,
            in float q,
            out float x1,
            out float x2)
{ float root = sqrt( p * p / 4 - q );
  x1 = -p / 2 + root;
  x2 = -p / 2 - root;
}
...;
solve(p,q,x1,x2);
...;
```

In Java umständlicher, da man nicht einfach zwei Rückgabewerte realisieren kann.

14.5.2001

1.12.2.3 Variablenparameter: Call-by-reference

Name des formalen Parameters ist ein *Alias* (Synonym) für den aktuellen Parameter, d.h. *keine Kopie*!

Merke: Call-by-reference existiert nicht Java (nicht auf primitiven Typen), aber z.B. in C, Modula, Pascal

Beispiel: Vertauschen zweier Integer-Zahlen: (siehe auch Sortieralgorithmen in Abschnitt 1.11.2)

```
void swap(var int a, var int b) {
    int temp = a;
    a = b;
    b = temp;
}
...;
swap(a[i],a[j]);
...;
```

Dies geht nicht mit call-by-value! Ausweg (z.B. in Java): Man übergibt eine *Referenz* auf Array und die Indizes als separate Parameter:

```
static void swap(int[] a, int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
...;
swap(a,i,j);
...;
```

Beispiel: Runden einer Gleitkommazahl:

```
void round (var double x) {
       x = (long) (x + 0.5);
}
...;
round(a);    // bewirkt Effekt, da call-by-reference
...;
```

Es ginge bei `round(a)` auch call-by-result: `void round (in out double x) { ... }`, aber nicht im folgenden Beispiel.

Beispiel: Ungewollter Effekt durch call-by-reference

```
int getPowerOf(var int a, var int x) {
       // requires (x >= 0);
       int power = 1;
       while (x != 0) {
              power = power * a;
              x--;
       }
       return power;
}
...;
int k = 3;
int y = getPowerOf(k,k);
...;
```

```
  k
 / \
a   x   power
-------------
3   3     1
2   2     3
1   1     6
    0           return 6
```

Problem: Beim Aufruf von `getPowerOf(k,k)` sind `a` *und* `x` Synonyme für `k`.

1.12.2.4 Namensparameter: Call-by-name

Name des formalen Parameters wird durch den Ausdruck (nicht Wert!) des aktuellen Parameters ersetzt.

Merke: Call-by-name existiert nicht in Java, aber z.B. in ALGOL 60.

```
void round (name double x) {
       x = ...;
}
```

Call-by-name ist Äquivalent zu lazy evaluation (Auswertung des Ausdrucks erfolgt erst so spät wie möglich).

Beispiel für call-by-name (konstruiert):

```
void P (<call-by-?> x) {
  i = i + 1;  // Zeile *
  x = x + 2;  // Zeile **
}

int[] f = new int[2];
int i;
...;
f[0] = 1;
f[1] = 2;
i = 0;
P( f[i] );
```

call-by-value:	x ← f[0]=1	➔ f[0]=1, f[1]=2
call-by-reference:	adrx ← adrf[0]	Wert an adrx wird um 2 erhöht wegen Zeile **
		➔ f[0]=3, f[1]=2
call-by-name:	x ← f[i]	i wird ersetzt durch i+1 wegen Zeile *
		daraus folgt x ← f[i+1]
		anschließend wird f[i+1] um 2 erhöht (mit i=0)
		➔ f[0]=1, f[1]=4

1.13 Zusammengesetzte Datentypen

Wh: (siehe Abschnitt 1.6 Einfache und zusammengesetzte Datentypen)

Man unterscheidet:

- Felder (Arrays) und
- Strukturen.

1.13.1 Felder

(siehe Abschnitt 1.11.2 Arbeiten mit Arrays), alle Elemente vom gleichen Typ

Eindimensionales Array:

a[0]	a[1]	a[2]	a[3]

Mehrdimensionales Array:

a[i][j] → i

↓ j

a[0][0]	a[1][0]	a[2][0]	a[3][0]
a[0][1]	a[1][1]	a[2][1]	a[3][1]

1.13.2 Strukturen

Eine Struktur (struct, record) ist ein Verbund von Datenelementen (Komponenten), die *unterschiedliche* Typen haben können. Der Zugriff auf die einzelnen Komponenten erfolgt durch *Selektion* in der Form `<name>.<komponente>`.

Beispiele:

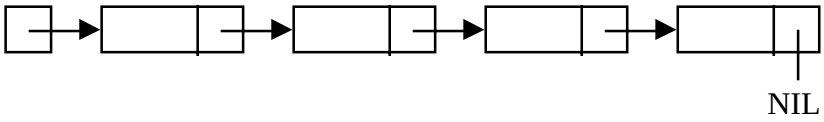
1. Struktur `termin` (Modula 2) aus Abschnitt 1.6
2. Komplexe Zahl in Modula 2:

```
TYPE Complex = RECORD
         re : REAL;
         im : REAL;
      END;

VAR c1,c2 : Complex;      // Anlegen einer Variable vom Typ Complex
...
c1.re := c2.im;           // Zugriff auf Komponenten
```

3. Zusammengesetzter Datentyp einer einfach verketteten Liste aus Integer-Werten in Modula 2:

```
TYPE
       eListElement = POINTER TO eListEntry;
       eListEntry = RECORD
                        content: INTEGER;
                        successor: eListElement;
                    END;
VAR
       anchor: eListElement;
```



Eine verkettete Liste ist eine lineare, dynamisch erweiterbare Anordnung von Datenelementen, die explizit durch Kanten verbunden sind (realisiert durch Zeiger bzw. Referenzen)

`eListElement` ist ein Zeigertyp auf die Struktur `eListEntry`. `anchor` und `successor` sind *Zeigervariable* (siehe folgenden Abschnitt) auf Elemente vom Typ `eListEntry`.

1.13.3 Zeiger und Referenzen

Eine Zeigervariable (pointer variable) enthält die Adresse eines Datenobjektes. Die Bezeichnung eines Datenobjektes durch eine Zeigervariable heißt Referenz.

17.5.2001

- Zeiger sind insbesondere im Zusammenhang mit dynamischen Datenstrukturen interessant
- Speicherplatz für dynamische Datenstruktur wird erst bei Bedarf zur Laufzeit angelegt: `new <type>`

Beispiel: Stapel (Keller, Stack) oder LIFO-(last in first out)-Behälter

// Stapel (Stack):

```
public class Stack {

    private Node anchor;

    public Stack() {
        anchor = null;
    }

    public void push(final Object element) {
        anchor = new Node(element, anchor); // neues Element vorne einketten
    }

    public Object pop() {
        if (anchor == null)
            // return new String("[Error: Stack is empty]");
            return null;
        Object value = anchor.content; // zwischenspeichern
        anchor = anchor.link;          // letztes eingetragenes Element
ausketten
        return value;
    }

    public static void main(String[] argv) {
        Stack s = new Stack();
        // Elemente ablegen
        s.push(new String("Erstes Element"));
        s.push(new Integer(2222222));
        s.push(new String("Drittes Element"));
        // Elemente holen
        System.out.println( s.pop() );  // gibt zuletzt abgelegtes Element aus
        System.out.println( s.pop() );
        System.out.println( s.pop() );
        System.out.println( s.pop() );
    }
}

class Node {

    Object content;
    Node link;

    Node(final Object value, final Node prevLink) {
        content = value;
        link = prevLink;
    }
}
```

Ausgabe:

```
Drittes Element
2222222
Erstes Element
null
```

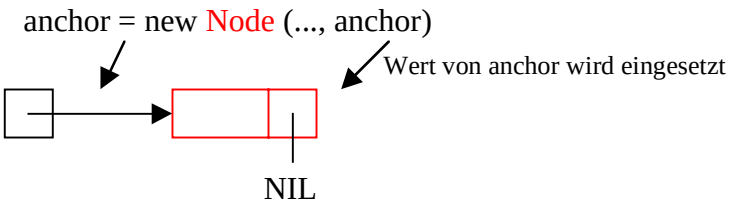
Wie funktionierts?

- nach Initialisierung:

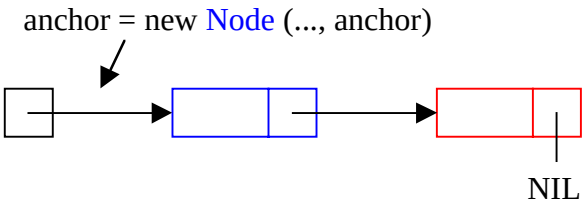
anchor:

NIL

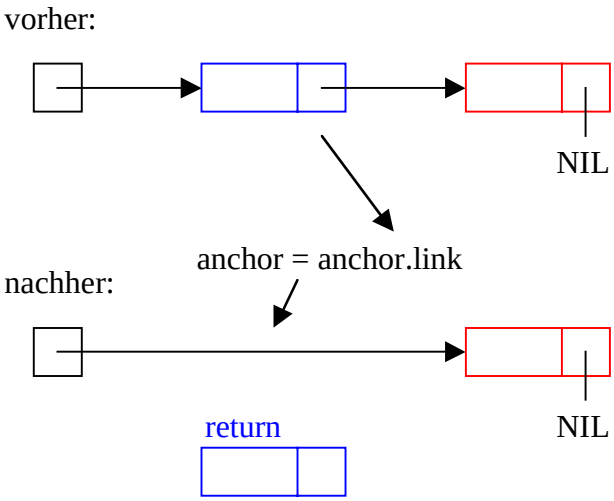
- Erstes Element hinzufügen: push(...)



- Nächstes Element hinzufügen: push(...)



- Element holen: pop()



Gegenüberstellung Felder — Liste:

- Felder:
  - Reihung wird innerhalb des Datenfeldes ausgedrückt (Indizes),
  - Größe wird mit der Feldinitialisierung festgelegt.
- Listen:
  - Reihung wird explizit in den einzelnen Listenknoten gespeichert,
  - Größe kann dynamisch wachsen bis zur Speichergrenze.

1.14 Ausnahmen- und Fehlerbehandlung

Mittels einer Ausnahmen- und Fehlerbehandlung kann auf kritische Situationen zur Laufzeit eines Programms reagiert werden. Eine Situation, die ohne Fehlerbehandlung zum Abbruch des Programms führen würde, kann mittels einer definierten Reaktion behandelt und das Programm fortgesetzt werden.

1.14.1 Laufzeitfehler

Ursachen für Laufzeitfehler (runtime exceptions) können z.B. sein:

- statische Fehler:
  - Syntaxfehler (interpretierte Sprachen, z.B. Basic, Perl),
  - semantische Fehler (z.B. statische Typfehler),
- dynamische Fehler:
  - dynamische Typfehler,
  - undefinierte Ausdrücke,
  - unerwartete Werte von Variablen und Ausdrücken,
  - Ausnahmen beim Zugriff auf Ressourcen.

Folge: Programm meldet Laufzeitfehler und beendet sich.

Beispiele:

`a/b`	falls `b == 0`	`ArithmeticException`
`a[i]`	falls `a == 0`	`NullPointerException`
`a[i]`	falls `i < 0` oder `i > a.length`	`IndexOutOfBoundsException`
Öffnen einer Datei, die nicht existiert		`FileNotFoundException`

1.14.2 Ausnahmebehandlung

Was, wenn der Speicherplatz für den Stack (siehe Beispiel aus Abschnitt 1.13.3) ausgeht? → Fehler abfangen und Ausnahmebehandlung einleiten.

Syntax (Java, vereinfacht):

```
TryBlock: try Block Catches |
          try Block [Catches] finally Block
Catches: CatchClause { CatchClause }
CatchClause: catch ( Exception Identifier ) Block }
Exception: Identifier
```

Beispiel:

```
int a[] = new int[2];
try {
       a[4] = 1;
} catch (ArrayIndexOutOfBoundsException e) {
       System.out.println("exception: " + e.getMessage());
       e.printStackTrace();
}
```

Ausgabe: (Apple Macintosh)

```
exception: null
java.lang.ArrayIndexOutOfBoundsException
       at alp2.main(Compiled Code)
       at
com.apple.mrj.JManager.JMStaticMethodDispatcher.run(JMAWTContextImpl.java)
       at java.lang.Thread.run(Thread.java)
```

Abfangen sinnvoll für:

- aussagekräftige Fehlermeldungen,
- dynamische Reparaturversuche,
- Vermeidung redundanter Abfragen.

Aufräumarbeiten mit `finally`:

Die `finally`-Klausel enthält letzte Anweisungen des `try`-Blocks, unabhängig von der Art seiner Beendigung.

```
try {
    // Anweisungen, die Fehler verursachen könnten
} catch (Exception1 e1) {
    // Behandlung von e1
} catch (Exception2 e2) {
    // Behandlung von e2
} finally {
    // Diese Anweisungen werden immer ausgeführt
}
```

Beispiel: Anstelle des mehrmaligen Schließens einer Datei im `try`-Block und in den `catch`-Klausel erfolgt das Schließen in der `finally`-Klausel:

Anstelle von:

```
try {
    // Oeffnen einer Datei
    // Lesen aus Datei
    // Schliessen der Datei <--
} catch (IOException ioe) {
    // Behandlung von ioe
    // Schliessen der Datei <--
}
```

mit `finally`-Klausel:

```
try {
    // Oeffnen einer Datei
    // Lesen aus Datei
} catch (IOException ioe) {
    // Behandlung von ioe
} finally {
    // Schliessen der Datei <--
}
```

Beispiel:

```
       int a[] = null;
       try {
         System.out.println("Start!");
          a[4] = 1;                          // Zeile *
       }
       catch (ArrayIndexOutOfBoundsException e) {
              System.out.println("Falscher Array-Index!");
       }
       finally {
              System.out.println("Der finally-Block wird abgearbeitet!");
       }
       System.out.println("Ende!");
```

Ausgabe:

```
Start!
Der finally-Block wird abgearbeitet!
Exception Occurred:
java.lang.NullPointerException
        at TryTest.main(TryTest.java:8)
        at
com.apple.mrj.JManager.JMStaticMethodDispatcher.run(JMAWTContextImpl.jav
a)
        at java.lang.Thread.run(Thread.java)
```

Die Zuweisung in `Zeile *` führt wegen des nicht angelegten Arrays zu einer `NullPointerException`. Es wird vor dem Beenden des Programms mit einem Laufzeitfehler noch der `finally-Block` abgearbeitet.

1.14.3 Auslösen von Ausnahmen in Unterprogrammen

Manchmal ist es sinnvoll, Ausnahmen selbst auszulösen, um sie entsprechend zu behandeln.

Syntax (Java, vereinfacht):

```
ThrowsClause: throws Exception {, Exception }

ThrowStatement: throw Expression;
```

Die `ThrowsClause` wird optional am Ende einer Methodendeklaration gesetzt. Die `Expression` bei `ThrowStatement` muss ein „Exception-Objekt“ darstellen.

Beispiel: Stack mit Abfangen von Fehlern. Anstelle von `OutOfMemoryError` und `NullPointerException` werden eigene Ausnahmen (`StackFullException`, `StackEmptyException`) ausgelöst, wenn kein Element mehr auf dem Stack abgelegt werden kann bzw. dieser leer ist.

// Stapel (Stack) mit Ausnahmebehandlung:

```
public class Stack {

    private Node anchor;

    public Stack() {
        anchor = null;
    }

    public void push(final Object element)
        throws StackFullException
    {
        try {
            anchor = new Node(element, anchor); // neues Elem. vorne einketten
        } catch (OutOfMemoryError e) {
            throw new StackFullException();
        }
    }

    public Object pop()
        throws StackEmptyException
    {
        try {
            Object value = anchor.content; // zwischenspeichern
            anchor = anchor.link;          // letztes eingetr. Elem. ausketten
            return value;
        } catch (NullPointerException e) {
            throw new StackEmptyException();
        }
    }
```

```
    public static void main(String[] argv) {
        Stack s = new Stack();
        // Elemente ablegen
        try {
            s.push(new String("Erstes Element"));
            s.push(new Integer(2222222));
            s.push(new String("Drittes Element"));
        } catch (StackFullException e) {
            System.err.println("Der Stack ist leider voll :-(");
        }
        // Elemente holen
        try {
            System.out.println( s.pop() );
            System.out.println( s.pop() );
            System.out.println( s.pop() );
            System.out.println( s.pop() );
        } catch (StackEmptyException e) {
            System.err.println("Der Stack ist leer!");
        }
    }
}

class Node {

    Object content;
    Node link;

    Node(final Object value, final Node prevLink) {
        content = value;
        link = prevLink;
    }
}

class StackFullException extends Exception {
    public String getMessage() { return "Stack full"; }
    public String toString()   { return "StackFullException"; }
}

class StackEmptyException extends Exception {
    public String getMessage() { return "Stack empty"; }
    public String toString()   { return "StackEmptyException"; }
}
```

1.14.4 Nachbildung von Ausnahmebehandlung

In Sprachen, die nicht über eine Ausnahmebehandlung verfügen, muss deren Funktion nachgebildet werden.

Idee: Statt `throw` verwendet man Funktion mit „Sonderwert“ als Rückgabe.

- Typische Werte (je nach `return`-Typ) für **Fehlerfall**: `-1, false, null`
- Andernfalls: sinnvoller Wert (z.B. beim Lesen aus einer Datei die Anzahl der gelesenen Bytes), `true`

Ein solches Vorgehen ist manchmal auch in Java sinnvoll, trotz Vorhandensein einer Ausnahmebehandlung.

Beispiel:

```
import java.io.File;
...
int processFile(String fileName) {
   File f = new File(fileName);
   if (f.exists() && f.canRead() && f.isFile()) {
          int nrOfBytesRead = 0;
          //
          // read content from file
          // in a loop
          //
          return nrOfBytesRead
   } else {
          return -1;
   }
}
```

Weitere Möglichkeit: (nicht in Java)

- Definition eine Sprungmarke, zu der im Fehlerfall gesprungen wird.

  Beispiel:

  ```
  void p(..., label exceptionLabel) {
          ...
          if (...) goto exceptionLabel;
          ...
  }
  ```
- In C siehe auch setjmp(3) und longjmp(3).