Objektorientierte Programmierung

Hannes Federrath / Christian Wolff

Textgrundlage: Vorlesungsskript Federrath, mit Ergänzungen

Literaturempfehlungen

- Dokumentationen und Spezifikationen (EN)
  - Java Development Kit (JDK) Documentation
    http://java.sun.com/j2se/1.4.2/docs/index.html
  - Java API Specification
    http://java.sun.com/j2se/1.4.2/docs/api/index.html
  - Java-Tutorial
    http://java.sun.com/docs/books/tutorial/
- Bücher (DE)
  - Hubert Partl: Java-Einführung
    http://www.boku.ac.at/javaeinf/
  - Christan Ullenboom: Java ist auch eine Insel
    http://www.galileocomputing.de/openbook/javainsel4/
  - Guido Krüger: Handbuch der JAVA-Programmierung
    http://www.javabuch.de
  - Java 2. Grundlagen und Einführung. RRZN Hannover.
    Erhältlich am Infostand im Rechenzentrum.

Javakurse in Informationswissenschaft / Medieninformatik

Grundwissen

Gunrdkurs Java
- **Einführung in die Sprache Java**
- **Einführung in das objektorientierte Programmieren**

Entwicklung medienverarbeitender Systeme (WS 06/07)
- **Medien-APIs**
- **Java TV**
- **Java Speech**
- **Bildverarbeitung**
- **XML**

Gestaltung grafischer Benutzeroberflächen (WS)
- **Einführung in Swing**
- **Einführung in JSP / Tomcat**

Java Business Computing
- Einführung in die J2EE
- Enterprise Java Beans
- Web Services

(Durchführung offen bzw. bei WInf)

Aufbauwissen

⌘ Programmiersprachen

- Sind formale Sprachen mit festegelegter Syntax und Semantik, die für Maschinen verständlich sind.
- Ermöglichen die formale Beschreibung von Problemlösungsverfahren, die auf einem Computer oder Computersystemen ausführbar sind.
- Bilden die Basis zur Entwicklung von Software und Betriebssystemen.

⌘ Syntax, Semantik, Pragmatik

- Eine Sprache besteht aus einem Alphabet von Zeichen.
- Die Wörter einer Sprache werden durch (formale) Regeln - der Grammatik - gebildet.
- Die Grammatik für Programmiersprachen wird Syntax genannt.
- Die inhaltliche Bedeutung wird durch die Semantik ausgedrückt wird.
- Die Pragmatik einer Sprache beschreibt, wie die Konstrukte einer Sprache sinnvoll eingesetzt werden.

Vorkenntnisse der Kursteilnehmer (inkl. Vorjahre 2004/2005/2006)

- Maschinennahe Sprachen
  - Assembler (1/-/-)
- Imperative Sprachen
  - Basic: (ca. 5/13/4)
  - Pascal (ca. 10/7/6)
  - C (fast alle/fast alle/11)
  - Ada, Modula II -
- Objektorientierte Sprachen
  - C++ (1-2/5/2)
  - Smalltalk (ca. 2-3/6/2)
  - C# (-/-/-)
- Keine PS in Schule: 12
- Deklarative / funktionale Sprachen
  - Lisp (-/-/1)
  - Prolog (-/2/1)
- Skriptsprachen
  - PHP (10/10/2)
  - Perl Python (1/3/1)
  - VBScript Javascript (wenige/6/3)
- Shell-Programierung (UNIX) 1
- Anwendungsspezifische Sprachen (z. B. CNC-Steuerung)
- PL/SQL 1
- Autorensysteme -
- Keine: (-/3/-)

Algorithmen und Programme

- Algorithmus
  - Der klassische Algorithmusbegriff abstrahiert von Rechnern und Programmiersprachen.
  - Ein Algorithmus ist eine Vorschrift zur Lösung einer Klasse gleichartiger Probleme, bestehend aus Einzelschritten.
  - Beispiele: Euklidischer Algorithmus, Such- und Sortieralgorithmen, etc.

- Programme
  - Um dem Rechner einen Algorithmus verständlich mitzuteilen, muss man diesen für ihn verständlich als Programm formulieren.
  - Ein Programm ist in einer Programmiersprache verfasst und ist eine Folge von Arbeitsanweisungen für den Rechner.

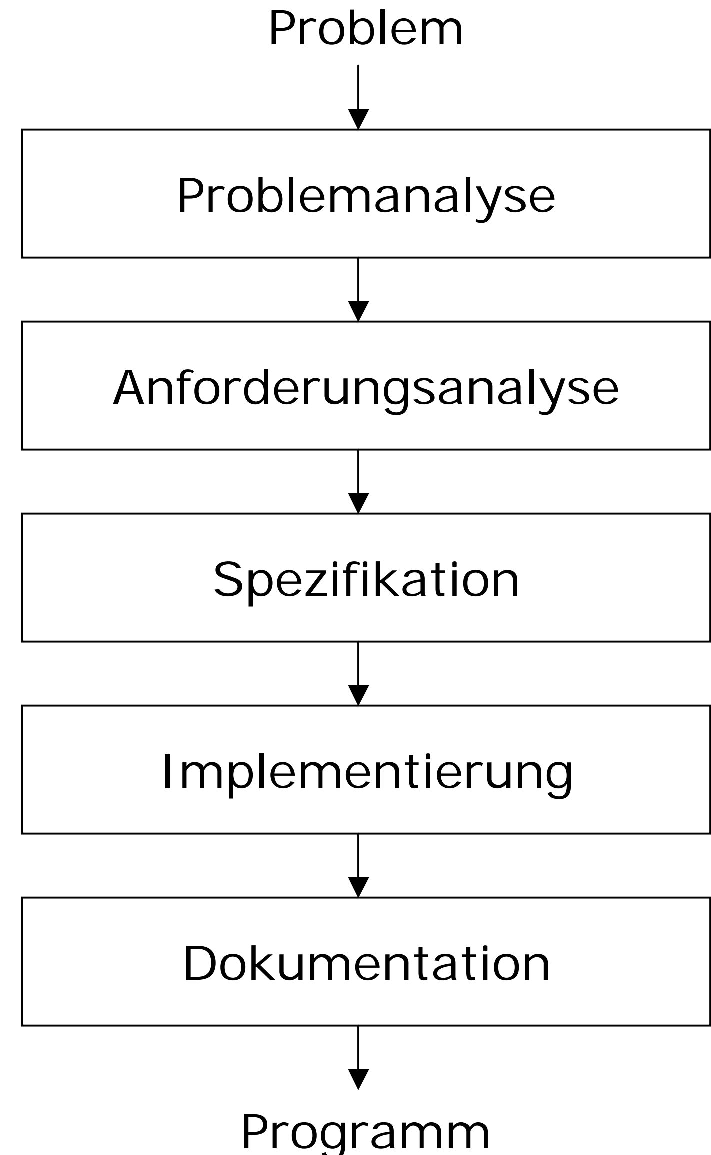
- Programmiersprachen:
  - Ermöglichen formale Beschreibung von Problemlösungsverfahren, die auf einem Computer oder Computersystemen ausführbar sind.
  - Bilden die Basis zur Entwicklung von Software und Betriebssystemen.
- Programmentwicklung erfordert im Allgemeinen mindestens ein zweistufiges Verfahren:
  - Entwurfsphase: Formulierung eines abstrakten Problemlösungsverfahrens in Form eines Algorithmus
  - Codierungsphase: Transformation des Algorithmus in ein Programm; dabei Verwendung von Kontrollstrukturen und Datentypen

Es gibt keinen Algorithmus zum Schreiben eines Programms bzw. Algorithmus.

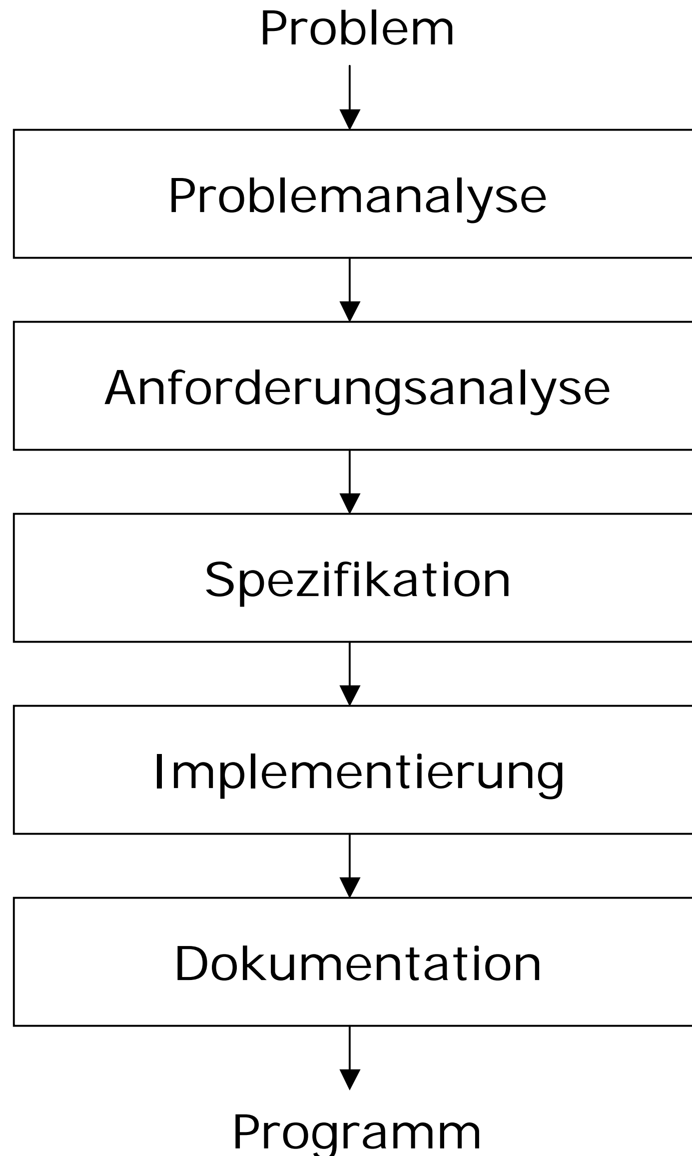
Phasen des SW-Entwicklungsprozesses

- Fragestellung:
  - Wie komme ich vom Problem zur Lösung?
- oder spezieller:
  - Welche Phasen muss ich durchlaufen, um vom Problem zum Programm zu kommen?

- Die Entwicklung einer Software sollte im Idealfall folgende Phasen durchlaufen:



Phasen des SW-Entwicklungsprozesses



Dokumentation beschreibt:

Welche generellen Leistungen soll das System erbringen?

Was soll das System im Einzelnen leisten?

Wie funktioniert das System?

Entwicklungswerkzeuge

- Texteditor
  - Ein Texteditor ist ein Programm zum Erstellen, Lesen und Ändern von Dateien, die Texte aller Art (beispielsweise Quellprogramme) enthalten
- Entwicklungsumgebung
  - Eine Entwicklungsumgebung stellt Werkzeuge zur Verfügung, die für die Erstellung von Computerprogrammen benötigt werden. Hierzu zählen: Editor, Werkzeuge zur Übersetzung und Werkzeuge zur Fehlerbehebung.
- Compiler
  - Ein Compiler ist ein Übersetzungsprogramm, das ein in einer höheren Programmiersprache abgefasstes Quellprogramm in eine andere Sprache, z.B. Maschinensprache, übersetzt.
- Weitere Entwicklungssoftware
  - Binder, Lader, Interpreter

- Präprozessor:
  - Vorbereitung des Quellcodes für Compiler
  - textuelle Ersetzungen von Codesegmenten
  - Makro-Ersetzung und Einkopieren von Deklarationsdateien bei C

- Beachte: Java arbeitet ohne Präprozessor

- Beispiel:

```
#include <stdio.h> // Einkopieren von stdio.h

#define ende(x) ((x=='\0')||(x=='\n')||(x=='\r')||(x=='\t')) // Makro
...
if (ende(zeichen)) wird ersetzt zu
if (((zeichen =='\0')||( zeichen =='\n')||( zeichen =='\r')||( zeichen =='\t')))
```

Programmübersetzung

- Präprozessor:
  - Vorbereitung des Quellcodes für Compiler
  - textuelle Ersetzungen von Codesegmenten
  - Makro-Ersetzung und Einkopieren von Deklarationsdateien bei C
- Beachte: Java arbeitet ohne Präprozessor
- Beispiel:

```
// plattformabhängige Übersetzung
#define PLATFORM_UNIX //oder alternativ PLATFORM_MAC oder PLATFORM_WINDOWS
...
#ifdef PLATFORM_UNIX
  ... // hier Zeilen, die nur in Unix gebraucht werden
#endif
```

Java: Bedingte Übersetzung

- static kann eingeschränkt zur bedingten Übersetzung (Direktiven) genutzt werden:

```
class StaticTest {
  final static boolean flag = ....... ;
  final static String s;
  static {
    if(flag) s="FLAG_IS_TRUE";
    else s="FLAG_IS_FALSE";
  }
}
```

- StaticTest.class

flag = true

```
...........................
.....flag...Z...ConstantVal
ue........s...Ljava/lang/St
ring;...<init>...()V...Code
...<clinit>........FLAG_IS_
TRUE........StaticTest...ja
va/lang/Object. ...........
...........................
..................*........
...........................
.......
```

flag = false

```
...........................
.....flag...Z...ConstantVal
ue........s...Ljava/lang/St
ring;...<init>...()V...Code
...<clinit>........FLAG_IS_
FALSE........StaticTest...j
ava/lang/Object. ..........
...........................
..................*.......
...........................
........
```

Java: Bedingte Übersetzung

- static kann eingeschränkt zur bedingten Übersetzung (Direktiven) genutzt werden:

```
class StaticTest2 {
  final static boolean DEBUG = ...... ;
  void prozedur() {
    if(DEBUG)
      System.out.println("IST NUR ZU SEHEN, WENN DEBUG TRUE IST.");
    System.out.println("IST IMMER ZU SEHEN");
  }
}
```

- StaticTest.class:

DEBUG = true

```
.....................................DEBUG...Z...ConstantValu
e........<init>...()V...Code...prozedur..............&IST NUR Z
U SEHEN, WENN DEBUG TRUE IST...........IST IMMER ZU SEHEN...Sta
ticTest2...java/lang/Object...java/lang/System...out...Ljava/io/
PrintStream;...java/io/PrintStream...println...(Ljava/lang/Strin
g;)V. ...................................................*.........
...........................................
```

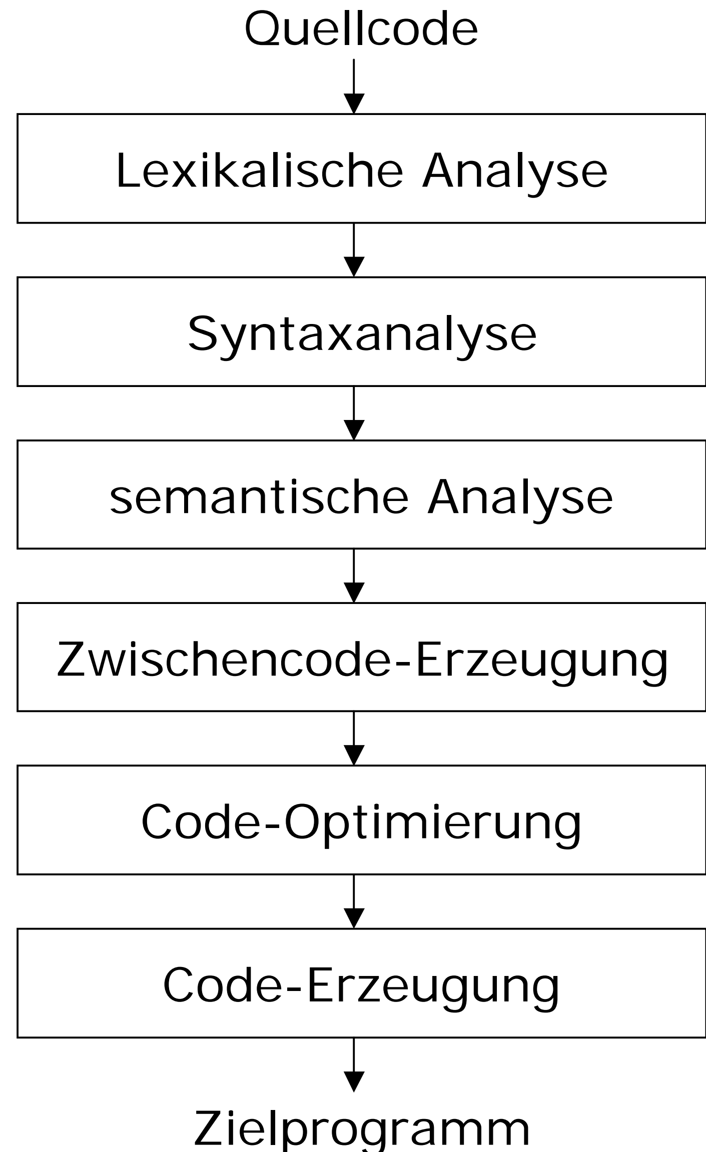
DEBUG = false

```
....................................DEBUG...Z...ConstantValue..
......<init>...()V...Code...prozedur...............IST IMMER ZU
 SEHEN..........StaticTest2...java/lang/Object...java/lang/Syst
em...out...Ljava/io/PrintStream;...java/io/PrintStream...println
...(Ljava/lang/String;)V. ....................................
..........*.........................................
```

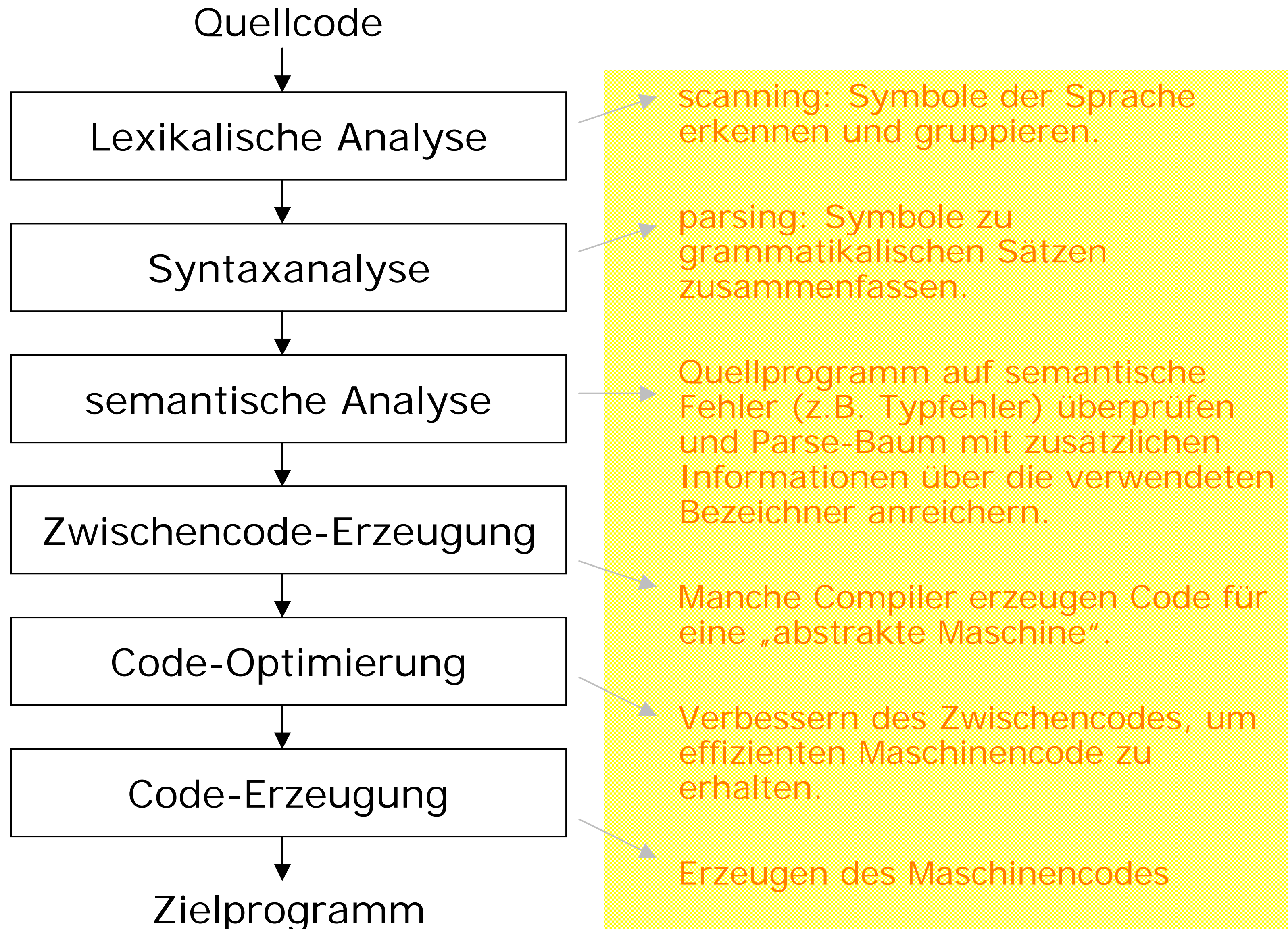
Programmübersetzung

- Compiler:
  - Überführen des in einer höheren Programmiersprache formulierten Programms (Algorithmus) in eine andere Sprache
    - z.B. Maschinensprache
  - Erzeugen des Codes der Zielsprache

- Phasen der Codeerzeugung:



Programmübersetzung: Compiler



- Binder und Lader:
  - Zusammenfassen verschiedener Maschinencode-Fragmente (mit relativen Adressen) zu einem ausführbaren Programm (z.B. Code aus Bibliotheken und eigener Code)
  - Umwandeln relativer in absolute Adressen und Laden des Programms an eine geeignete Stelle im Hauptspeicher

Interpreter

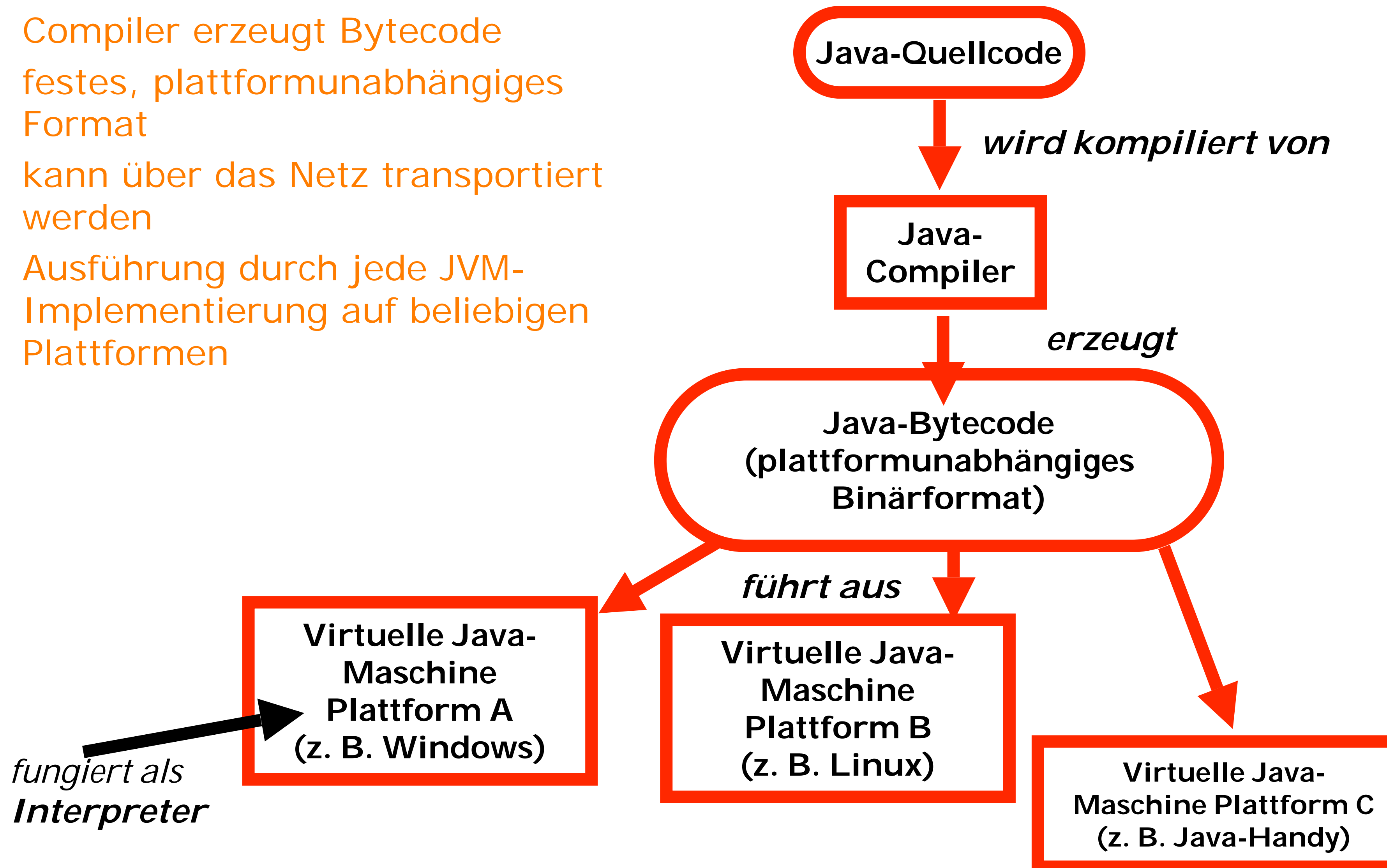
- Ein Interpreter analysiert wie ein Compiler den Quelltext, führt aber keine vollständige Übersetzung in Maschinensprache durch.
  - Programmtext wird entweder unmittelbar ausgeführt oder
  - in einen Zwischencode übersetzt, der den Programmtext interpretiert.

- Beispiele
  - Perl, Basic

- Nachteil
  - Da der Quellcode bei jedem Programmaufruf immer wieder neu übersetzt werden muss, laufen zu interpretierende Programme häufig langsamer als compilierte.

Schema der Bytecode-Generierung

- Compiler erzeugt Bytecode
- festes, plattformunabhängiges Format
- kann über das Netz transportiert werden
- Ausführung durch jede JVM-Implementierung auf beliebigen Plattformen



Klassifikation von Programmiersprachen

- **historisch** nach Generationen
  - 1. Generation – **Maschinensprache** – Befehle der Sprache entsprechen direkt dem von der Maschine ausführbaren Code, binäre Kodierung
  - 2. Generation – **Assemblersprachen**, ähnlich Maschinensprachen, aber Einführung verständlicherer Bezeichner für Befehlstypen
  - 3. Generation – **prozedurale Sprachen**, die kompiliert oder interpretiert werden, Programm als Anweisungsfolge
  - 4. Generation – **nicht-prozedurale** Sprachen
- **systematisch** nach Programmierparadigmen (s. u.)
- nach den den generellen Trend bestimmenden zentralen Konzepten: **Anweisung** → **Funktion** → **Modul** → **Objekt** → **Framework**, **Entwurfsmuster**, **Component Ware**

Programmierparadigmen

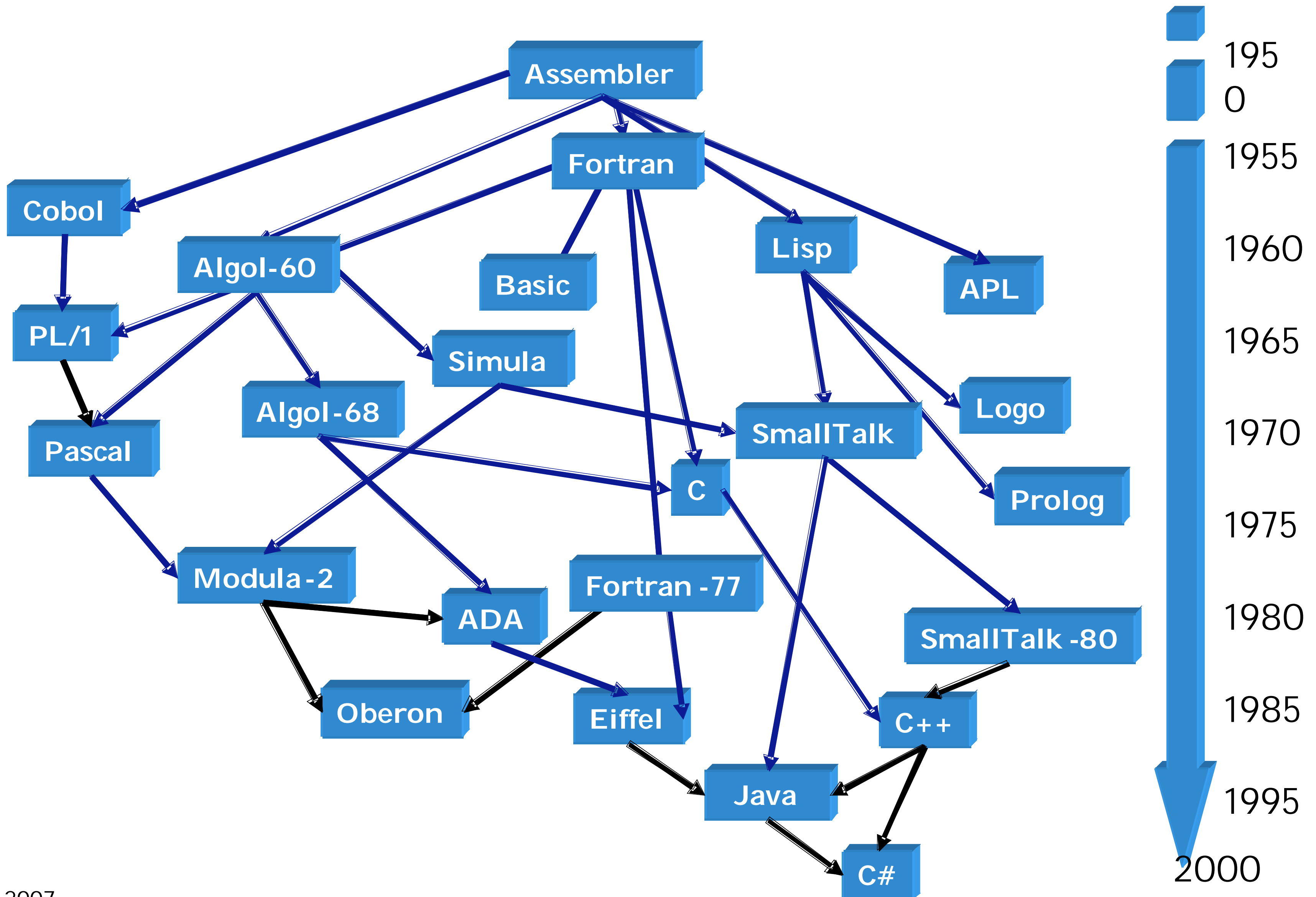
- Programmierparadigmen beschreiben grundsätzliche Konzepte oder Problemlösungsstrategien die einer Sprache zugrunde liegen.
- Etwa 1970: strukturierte (prozedurale) Programmierung (Pascal, C)
  - Stepwise refinement: Problem wird in kleine, leichter lösbare Probleme zerlegt und die Teillösungen werden zu einer Gesamtlösung zusammengesetzt (Top-Down-Ansatz)
  - Für kleinere Probleme gut geeignet („Programmieren im Kleinen“)
  - Weniger geeignet für größere Softwareprojekte
- Ab etwa 1980: Objektorientiertes Programmieren (Smalltalk, C++, Java, C#)
  - Entwicklung allgemein wiederverwendbarer und anpassbarer Softwarebibliotheken
  - Entwurf und die Pflege größere Softwareprojekte
- Andere Beispiele für Programmierparadigmen:
  - Funktionale Programmierung (Lisp, Haskell)
    - Lambda-Kalkül
  - Deklarative Programmierung (Prolog)
    - Meist zur Logikprogrammierung verwendet

Eigenschaften höherer Programmiersprachen

Sprache	Jahr	Typ	Anwendung	Vorläufer
Fortran	1956	prozedural	technisch-wissenschaftliche Anwendungen	
Cobol	1960	prozedural	Betriebswirtschaft	
Lisp	1960	funktional	künstliche Intelligenz	
Algol 60	1960	prozedural	universell	
Simula 67	1967	prozedural, objektorientiert	Simulation	Algol 60
PL/1	1968	prozedural	universell	Algol 60, Cobol
Pascal	1970	prozedural	Ausbildung, universell	Algol 60
Prolog	1970	Logik	künstliche Intelligenz	
C	1970	prozedural	Systemprogrammierung	Algol 60
Smalltalk	1980	objektorientiert	grafische Oberflächen	Simula 67
PEARL	1981	prozedural	Echtzeitprogrammierung	PL/1
Ada	1980	prozedural, objektorientiert	universell	Pascal
C++	1986	prozedural, objektorientiert	universell	C, Simula 67
Java	1992	objektorientiert	Internet, universell	C++, Smalltalk
C#	2000	objektorientiert	Internet, universell	C++, Java

Quelle: Taschenbuch der Informatik, 5. Auflage S. 273

Historische Entwicklung der Programmiersprachen



Objektorientierte Programmierung (OOP)

- Prinzip der OOP
  - Abbildung von Objekten der realen Welt
  - Alles ist ein Objekt
  - Objekte arbeiten zusammen
  - Ein laufendes Programm ist ein Universum miteinander interagierender Objekte
- Maxime der objektorientierten Programmierung
  - Entwicklung allgemein wiederverwendbarer und anpassbarer Softwarebibliotheken
- Objektorientierte Sprachen:
  - Smalltalk, C++, Java, C#

Objektorientierte Programmierung (OOP)

- Ziele der OOP
  - Verkürzung der Entwicklungszeit
  - Senkung der Fehlerrate
  - verbesserte Erweiterbarkeit und Anpassungsfähigkeit

- Hauptmerkmale
  - Datenkapselung
    - genau definierte Schnittstellen
    - Verbergen der Implementierungsdetails
  - Vererbung
    - einfache Modifikation und Erweiterung von bereits vorhandenen Komponenten
  - Polymorphie
    - gleiche Funktionalität für verschiedene Datentypen
    - Datentypabhängige Semantik von Operatoren und Funktionen

- Java
  - »Internet-Programmiersprache«
  - sehr umfangreiche Klassenbibliotheken
  - ab 1992 von Sun Microsystems vorgestellt
  - objektorientierte Programmiersprache
  - Vorläufer: C++, Smalltalk

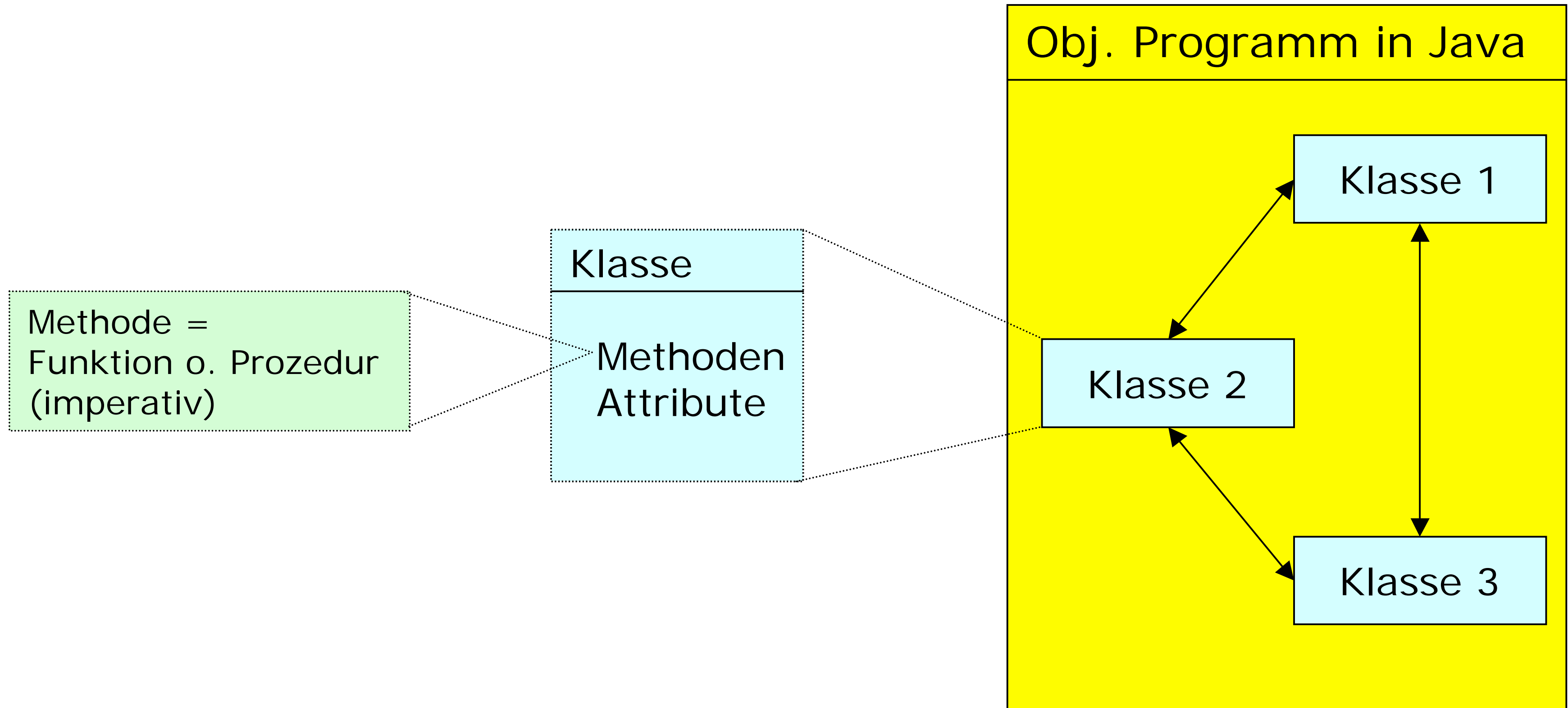
- einige Grundkonzepte
  - Einfachheit
  - Objektorientierung
  - Portabilität und Architekturneutralität
  - Interpretierte Sprache
  - Robustheit
  - Multithreading
  - Leistungsfähigkeit
  - Speicherverwaltung
  - Sicherheit
- Die wichtigsten Grundelemente objektorientierter Programmierung sind Klassen und die eigentlichen Objekte (Instanzen)

Klasse → Anlegen des Objektes → Objekt (Instanz)

Vorgeschichte von Java

- Ausgangspunkt von Java:
  Forschungsprojekt zur Entwicklung einer Programmiersprache für vernetzte und eingebettete Anwendungen (OAK)
- ca. 5 Jahre Entwicklungsdauer seit 1990
- entwickelt auf der Basis verfügbarer OO-Programmiersprachen (C++, SmallTalk, Eiffel) und mit Hinblick auf weitverbreitete Standards (C++)
- keine innovative Sprache, eher eine Mischung aus bewährten und / oder weit verbreiteten Konzepten, z. B. Syntax: wie C / C++ - sehr weit verbreitet, aber eigentlich schwer verständlich
- Java-Motto: "write once, run anywhere"

Klassen, Methoden, Attribute

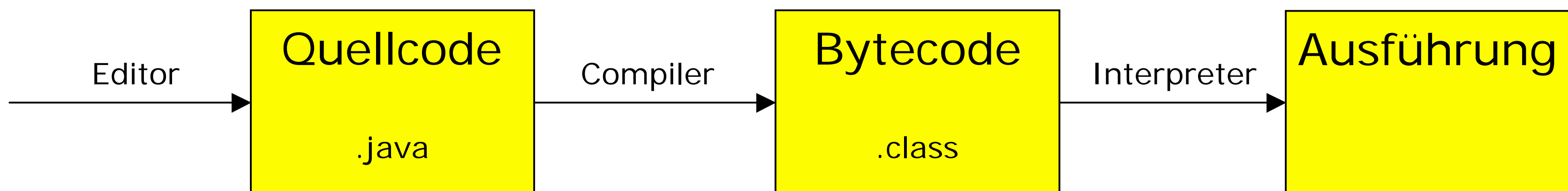


Schritte zum Erstellen eines Java-Programms

- Java kombiniert Übersetzungstechniken

- Java-Programme werden nach ihrer Erstellung *kompiliert*.
  - Dabei entsteht ein Zwischencode, der sog. Bytecode.
  - Dieser ist *nicht* unmittelbar ausführbar, dafür aber vollkommen unabhängig von der Hardware und dem Betriebssystem.

- Der Bytecode wird anschließend durch die sog. Virtuelle Maschine (JVM) *interpretiert*.



Zwei wesentliche Arten von Programmen in Java

- Applikation
  - eigenständiges Programm
  - durch einen Java-Interpreter ausgeführt

- Applet
  - üblicherweise kleineres Programm
  - zusammen mit einer HTML-Seitenbeschreibung auf einem Web-Server als Bytecode abgelegt
  - auf einem Web-Client von einem im Browser integrierten Java-Interpreter ausgeführt
  - Hauptanwendung:
    - Realisierung dynamischer Inhalte in Webseiten

- (weitere: Servlets, XLets, ...)

»Mein erstes Java-Programm«

⌘ Dateiname: SayHello.java

```
public class SayHello {
    public static void main(String[] args) {
      Mouth mouth = new Mouth();
      mouth.say("Hello, world");
    }
}
class Mouth {
    public Mouth() { } // Konstruktor (hier leer)
    public void say(String what) {
      System.out.println(what);
    }
}
```

Anmerkungen zu »Mein erstes Java-Programm«

- class
  - Definieren einer Klasse
- new
  - Anlegen eines Objektes (Instanzierung einer Klasse)
- public
  - bestimmt die *Sichtbarkeit* eines Objektes bzw. einer Methode
- main
  - sog. Klassenmethode (auch: statische Methode, static)
  - benötigt keine Instanzierung der Klasse, um ausführbar zu sein

- Zugriff auf Methoden (und Variablen) fremder Objekte:
  - <Objektname>.<Methodenname>
  - Beispiel: mouth.say(...)

- Dateiname muss gleich dem Klassennamen sein

Übersetzen und Ausführen

- Übersetzen (in Konsole):

```
> javac SayHello.java
```

  - Datei mit der Endung .class wird erzeugt
  - .class-Datei enthält plattformunabhängigen Bytecode

- Ausführen (in Konsole):

```
> java SayHello
Hello, world
> _
```

- Wo gibt es Java?
  - Ausführungsumgebung (auch: Laufzeitumgebung bzw. Java Runtime Environment, JRE) heute in fast allen Betriebssystemen enthalten
  - Entwicklungsumgebung (Java Development Kit, JDK) ebenfalls für die meisten Betriebssysteme vorhanden

Sprachelemente von Java

- Im folgenden wird beschrieben,
  - welche Wörter zum Schreiben von Java-Programmen existieren,
  - wie sie zusammengesetzt werden und

  Syntax

  - welche Bedeutung diese haben.

  Semantik

Syntax, Semantik und EBNF

- **Syntax** einer Programmiersprache
  - beschreibt die Menge der erlaubten Zeichenketten für Programme
  - wird heute unter Verwendung kontextfreier Grammatiken angegeben
  - Formale Beschreibungsmittel
    - Backus-Naur-Form (BNF)
    - erweiterte Backus-Naur-Form (EBNF)
    - Syntaxdiagramme

- **Semantik** einer Programmiersprache
  - definiert die Bedeutung der einzelnen Sprachkonstrukte
  - wird für die meisten Programmiersprachen heute textuell beschrieben

EBNF

- Erweiterte Backus-Naur-Form (EBNF)
  - formale Beschreibung der Syntax einer Programmiersprache

- Metazeichen:

<table>
<tr><th>Metazeichen in EBNF</th><th>Bedeutung</th></tr>
<tr><td><Sprachkonstrukt></td><td>Sprachkonstrukt der Programmiersprache</td></tr>
<tr><td><l> ::= <r></td><td>Ableitungsregel: Der linke Teil <l> wird durch den rechten Teil <r> definiert.</td></tr>
<tr><td>|</td><td>Alternative Definition (Oder)</td></tr>
<tr><td>{ }</td><td>Das in Mengenklammern eingeschlossene Sprachkonstrukt kann 0-mal oder mehrfach vorkommen.</td></tr>
<tr><td>[ ]</td><td>Das in Intervallklammern eingeschlossene Sprachkonstrukt kann 0-mal oder höchstens 1-mal vorkommen.</td></tr>
</table>

EBNF: Beispiel

- Einfacher Taschenrechner
  - erwartet zunächst Eingabe einer Zahl
  - dann die Eingabe eines Operators (z. B. + oder –)
  - danach wieder Eingabe einer Zahl
  - anschließend Eingabe eines weiteren Operators
  - danach wieder Zahl u.s.w.
  - Ausdrücke können geklammert sein
- Beispielsyntax:

```
<Expression> ::= <Term>                          |
                 <Term> "+" <Expression>    |
                 <Term> "-" <Expression>
<Term>       ::= <Atom>   |  "(" <Expression> ")"
<Atom>       ::= <Number>{<Number>}
<Number>     ::= "0" | "1" | ... | "9"
```

- Semantik:
  - + ist Addition, – ist Subtraktion, (...) wird vorrangig ausgewertet

- Syntaxbeschreibungen der Java-Sprachkonstrukte für diese Vorlesung erfolgen in leicht modifizierter Form:

s	Sprachkonstrukt s
Links: Rechts	Ableitungsregel; Rechts wird durch Links ersetzt
<u>foo</u>	Literal (wortwörtlich); wird nicht vom Syntaxanalysator ersetzt (entspricht "foo" in der EBNF); durch Unterstreichung gekennzeichnet
a | b	Alternative (wie EBNF); Die Sprachkonstrukte a oder b sind möglich
{x}	Das Sprachkonstrukt x kann 0-mal oder mehrfach vorkommen (wie EBNF).
[y]	Das Sprachkonstrukt y kann 0-mal oder höchstens 1-mal vorkommen (wie EBNF).

»Mein zweites Java-Programm«

```
class Mimic {                                          // Definition einer Klasse
    public static void main(String[] args) {           // Definition einer Methode
        Mouth mouth = new Mouth();                     // Erzeugen eines Objekts
        Eye leftEye = new Eye();                       // Erzeugen eines Objekts
        Eye rightEye = new Eye();                      // Erzeugen eines Objekts
        rightEye.twinkle();                            // Aufruf auf eine Methode
        mouth.say("This is an example");               // Aufruf auf eine Methode
    }
}
class Mouth {                                          // Definition einer Klasse
    void say(String what) {                            // Definition einer Methode
        System.out.println(what);                      // Aufruf auf eine Methode
    }
}
class Eye {                                            // Definition einer Klasse
    void twinkle() {                                   // Definition einer Methode
        ...
    }
}
```

Klassen: Syntax

⌘ Syntax (vereinfacht) für Java-Klassen

```
ClassDeclaration: [Modifiers] class Identifier
                        [ extends Type ]
                        [ implements Type {, Type } ]
                        ClassBody
ClassBody: { { VariableDeclaration }
             { ConstructorDeclarartion }
             { MethodDeclaration }
           }
VariableDeclaration: [Modifiers] Declaration
ConstructorDeclaration: Identifier ( [FormalArguments] ) Block
Modifiers: public | abstract | final
```

Klassen: Semantik

- Klassendefinition beschreibt die
  - Attribute (attributes, Instanzvariable) und
  - Methoden (methods, Prozeduren, Funktionen) eines Objektes
- Spezielle Methoden innerhalb der Klassendefinition:
  - Konstuktor (constructor)
    - legt die Anfangswerte der Attribute fest
    - führt ggf. Methoden zur Initialisierung aus
    - Klasse → Anlegen des Objektes (Aufruf des Konstruktors) → Objekt (Instanz)
  - Destruktor (destructor, finalizer)
    - entfernt dynamisch erzeugtes Objekt aus Hauptspeicher
    - führt ggf. vorher Aufräumarbeiten aus
- Beachte: Definition einer Klasse erzeugt noch keine Objekte
  - Deklaration von Variablen
  - dynamisches Erzeugen eines Objektes mit new

- Syntax (vereinfacht) zum Erzeugen:

```
StatementExpression: ... |
                     MethodInvocation |
                     CreationExpression
CreationExpression: new Identifier ( [ActualArguments] )
```

- Semantik
  - Mit **new** wird ein Objekt *dynamisch* erzeugt.
  - Beim Anlegen des Objektes wird eine Instanz der Klasse erzeugt.
    - Aufruf des Konstruktors
  - Ein Objekt ist eine Einheit von Daten und Funktionen, die auf den Daten operieren.
  - Jedes Objekt besitzt eine eigene Identität (identity) und einen eigenen Satz Daten.
  - Struktur von Daten und Funktionen gleichartiger Objekte ist in ihrer gemeinsamen Klasse (class) definiert Variablen eines Objektes: Attribute
  - Funktionen eines Objektes: Methoden
    - definieren das Verhalten des Objektes.

Zugriff auf Attribute und Methoden: Semantik

- Innerhalb des Objektes
  - mit ihrem einfachen Namen angesprochen

- Attribute und Methoden fremder Objekte
  - über ihren qualifizierten Namen (qualified name)
    - Objektname.Variablenname bzw.
    - Objektname.Methodenname()

- this
  - Attribute und Methoden innerhalb des Objektes können mit
    - this.Variablenname bzw.
    - this.Methodenname()

    angesprochen werden.
  - this ist eine Referenz auf sich selbst.

Zugriff auf Attribute und Methoden: Syntax

- Syntax (vereinfacht):

```
Primary: ... |
         FieldAccess
FieldAccess: Primary . Identifier
```

- Bemerkungen:
  - Primary muss einen Verweistyp haben, d.h. eine Referenz sein *und*
  - zugehörige Klasse muss ein Attribut bzw. eine Methode mit dem angegebenem Namen haben
  - wenn Auswertung des Primary null liefert (Null pointer exception), dann
    - Laufzeitfehler

Beispiel: Klasse Bruch

- Klassendefinition:

```
class Bruch {

    /** Es folgen die Attribute */
    int zaehler;
    int nenner;

    /** Konstruktor */
    Bruch (int z, int n) {
        zaehler = z;
        nenner = n;
        kuerze();
    }

    /** Hier eine Methodendefinition zum Kürzen */
    void kuerze () {
        ...
    }
```

Weiterführung des Beispiels: siehe
Referenz- und Wertsemantik

Beispiel: Klasse Bruch

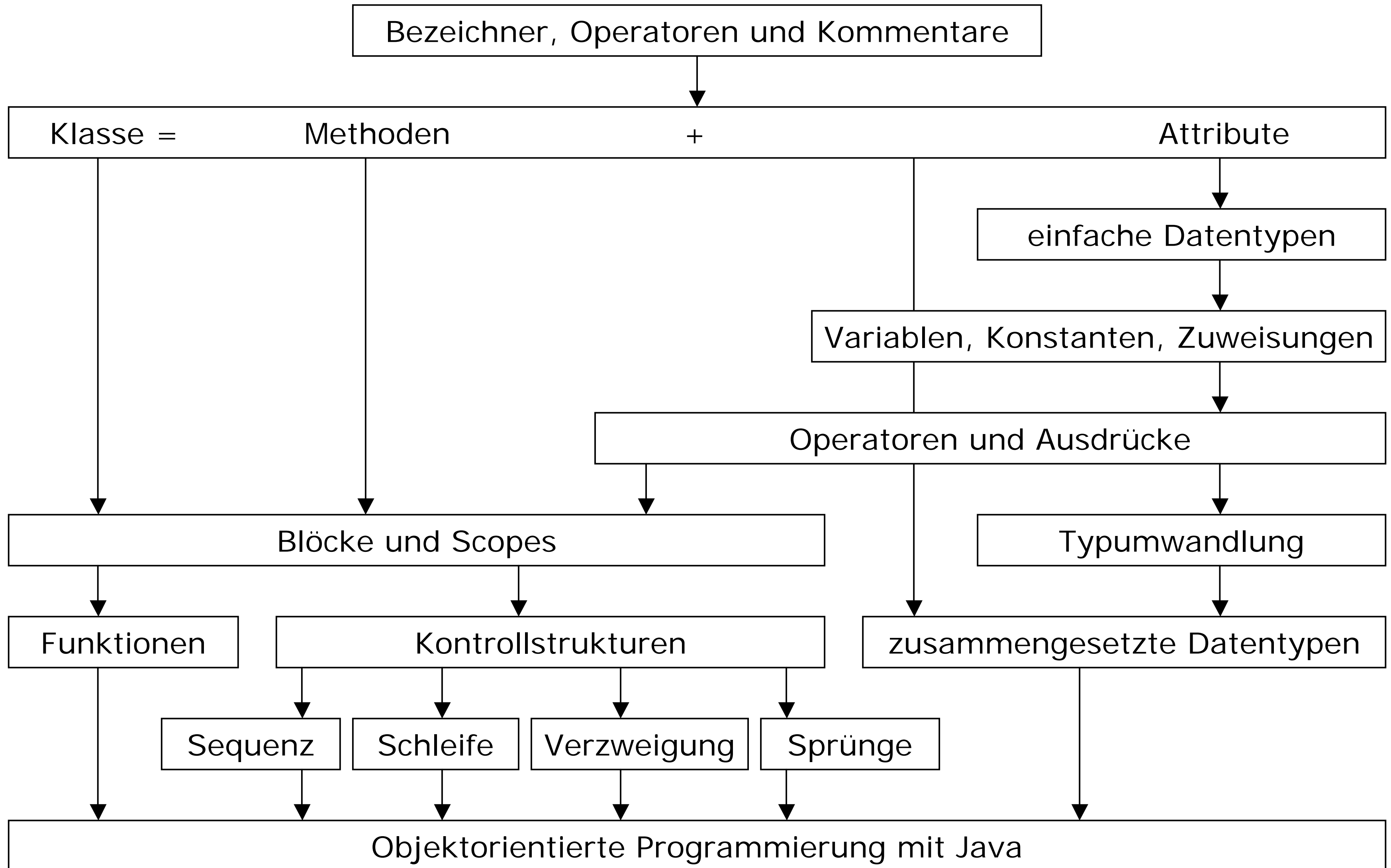
```
    /** Methode zum Hinzuaddieren */
    void add(Bruch r) {
        zaehler = zaehler*r.nenner + r.zaehler*nenner;
        nenner = nenner*r.nenner;
        kuerze();
    }

    /** Methode zum Wandeln in einen String */
    public String toString() {
        return "(" + zaehler + "/" + nenner + ")";
    }

}
```

Beispiel: Verwenden der Klasse Bruch

⌘ Anlegen eines Objektes:

```
class BruchApplication {

  public static void main (String[] argv) {
     Bruch b = new Bruch (2,6);    // Anlegen des Bruchs 2/6 mit new
     b.add(new Bruch(1,2));        // Hinzuaddieren von 1/2
     System.out.println("b=" + b); // ruft implizit b.toString() auf
                                   // ... rauskommen sollte 5/6
  }
}
```

»Fahrplan« durch die Sprachemelente



Grundlegende Sprachelemente von Java

- Zeichensatz Quellcode
  - 16-Bit-Zeichensatz: Unicode
- Bezeichner
  - Groß- und Kleinschreibung ist signifikant
  - Erstes Zeichen muss <buchstabe>, $ oder _ sein
  - Schlüsselwörter
    - sind reservierte Wörter
    - dürfen nicht für eigene Bezeichner verwendet werden
- Operatoren
  - Zusammengesetzte Ausdrücke sind durch Operatoren verbunden.
- Kommentare
  - vom Compiler ignoriert

Namenskonventionen

Schlüsselwörter

Kommentare

Namenskonventionen (Empfehlungen)

- Klassennamen
  - beginnen mit einem Großbuchstaben
  - bei mehreren Wörtern: jedes Wort mit Großbuchstaben beginnen
    - Window
    - MyOwnClass
    - NullPointerException

- Methoden-, Funktions- und Variablennamen
  - beginnen mit einem Kleinbuchstaben
  - bei mehreren Wörtern: jedes folgende Wort mit Großbuchstaben beginnen
    - send(), getBytes(), getByteFromKeyboard()
    - foo, windowSize

- Konstantennamen
  - alles Großbuchstaben
    - MAXINT, CRLF, DB_LIMIT

- Schlüsselwörter
  - stellen Grundwortschatz der Sprache dar
  - dürfen nicht für eigene Bezeichner verwendet werden
- Nicht alle Schlüsselwörter werden momentan tatsächlich benutzt.

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

- Reservierte Wörter

```
true     false     null
```

```
*     not used
**    added in 1.2
***   added in 1.4
****  added in 5.0
```

http://java.sun.com/docs/books/tutorial/java/nutsandbolts/_keywords.html

Kommentare

- Kommentare werden vom Compiler ignoriert.
  - Verbessern der Lesbarkeit des Quelltextes
  - Erhöhen Verständlichkeit des Programms
  - Übersetzung fehlerhafter oder unvollständiger Codefragmente in der Entwicklungs- und Testphase eines Programms verhindern
- Drei Schreibweisen:

  // Kommentar

    - Bis zum Ende der Zeile werden alle Zeichen vom Compiler ignoriert.

  /* Kommentar */

    - Alle Zeichen zwischen den »Klammern« werden vom Compiler ignoriert.

  /** javadoc-Kommentar */

    - Alle Zeichen zwischen den »Klammern« werden vom Compiler ignoriert.
    - wird verwendet zur Erstellung einer Online-Dokumentation einer Klasse mit javadoc

javadoc: Dokumentation von Klassen

- Generierung von HTML-Code zur Dokumentation einer Klasse

  javadoc <.java-Datei(en)>

- javadoc-Kommentare

  /** <text> */

- javadoc-Schlüsselwörter (Auswahl)
  - Klassen: dient der Beschreibung der
    - Funktionalität der Klasse

      @version <text>

      @author <text>
  - Methoden und Variablen: dient der Beschreibung der
    - Funktion der Methode und Variable sowie
    - Vorbedingungen, Nachbedingungen

      @param <name> <beschreibung>

      @return <beschreibung>

      @exception <voller Klassenname> <Beschreibung>

javadoc: Beispiel

```
/**
 * Ein ein einfaches Testprogramm f&uuml;r Java.
 * <p> Im Gegensatz zu anderen <tt>HelloWorld</tt>-Progies arbeitet dieses hier
 * <b>objektorientiert</b>! Es ist eines der ersten Beispiele der
 * <a href="http://www-sec.uni-regensburg.de/inf/">Informatik-Vorlesungen</a>.
 *
 * @author Hannes Federrath
 * @version 0.1b31415
 */
public class SayHello {
    /** Testroutine des Programms */
    public static void main(String[] args) {
          Mouth mouth = new Mouth();
          mouth.say("Hello, world");
    }
}
```

```
/** Beschreibung des Mundes, aus dem Botschaften
 *  wie <i>Hello World</i> herauskommen.
 */
public class Mouth {
    /** druckt den String <tt>what</tt> auf der Konsole aus.
     *  @param what auszugebender Text
     */
    public void say(String what) {
          System.out.println(what);
    }
}
```

Variablen

- Variablen (variables) bezeichnen Speicherplätze (»Behälter«), in denen Werte eines Datentyps abgelegt werden.
- Variablen
  - besitzen einen Namen
  - können während der Laufzeit verschiedene Werte annehmen
  - Werte können überschrieben werden
- In typisierten Sprachen
  - Variablen vor Verwendung deklarieren und typisieren
  - Vereinbarung (declaration) legt Namen (identifier) und Typ fest
  - Typangabe schränkt statisch die Werte ein, die eine Variable dynamisch annehmen kann.
- Syntax (vereinfacht):

  Declaration: [final] Type VarDeclaration {, VarDeclaration};

  Type: Identifier

  VarDeclaration: Identifier [= Initializer]

  Initializer: Expression

Zuweisung

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

  Assignment: Variable = Expression
- Zuweisungsoperator =
  - rechtsassoziativ, d.h. es wird erst der Ausdruck rechts vom = ausgewertet, bevor die Zuweisung erfolgt.
- Beispiel: (Java)
  - `x = 4567;`
  - `a = x = 1;`
  - `a = 12 * 3; // bewirkt den Effekt, dass a den Wert 36 zugewiesen bekommt`

Zuweisung: Beispiele

⌘ 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;
            }
```

Konstanten

- Konstanten (constants) repräsentieren unveränderbare Werte, die einen bestimmten Typ besitzen.
  - in Java mit dem Schlüsselwort final deklariert.
- Beispiel:
  - final int K = 5;

- vordefinierte Bezeichner 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`; Anhängen von `E` für Exponent	`double`	`double e = 2.7183;` `double x = 5.1E-2;`
Bereichsbegrenzung auf `float`: Anhängen von `F`	`float`	`float pi = 3.14F;` `float y = 5E-2F;`
Logische Konstante: `true` und `false`	`boolean`	`boolean f = true;`

Exkurs: Codierung von Zahlen

- Verschiedene Zahlensysteme
  - Dezimalsystem (10 Ziffern --> Basis: 10)
  - Dualsystem (Basis: 2)
  - Oktalsystem (Basis: 8)
  - Hexadezimalsystem (Basis: 16)

- Dualsystem in Computern zur internen Darstellung von Zahlen
  - leichte Realisierbarkeit in digitalen Schaltungen

- Zahlendarstellung
  - Positionssystem
  - Umrechnung unproblematisch

Beispiel:

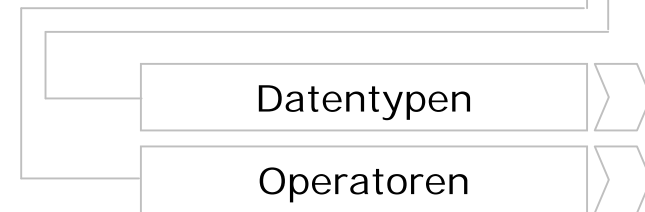
$12_{dez} = 1 \cdot 10^1 + 2 \cdot 10^0$

$12_{dez} = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 1100_{bin}$

- Spezielle Codierungen für
  - Kommazahlen
  - negativen Zahlen (Java: Zweierkomplement)

Ausdrücke

- Werte erhält man durch Auswertung eines Ausdrucks (expression).
  - Jeder Ausdruck hat einen statischen *Typ*.
  - Auswertung (evaluation) liefert i.d.R. Wert mit dynamischem Typ
- Zusammengesetzte Ausdrücke
  - verbunden durch *Operatoren*.
  - bestehen aus
    - Einfachen und zusammengesetzten Ausdrücken
    - Prozedur/Funktionsaufrufen
    - Klammern



- Syntax (vereinfacht):

  Expression: Primary | Assignment | Expr {InfixOp Expr}

  Primary: Literal | Identifier | (Expression)

  Expr: Primary [PostfixOp] | [PrefixOp] Primary | CastExpression

Datentypen in Java

⌘ Grunddatentypen

Datentyp	*Std-wert*	*Bescheibung*
boolean	false	boolescher Datentyp (wahr, falsch)
char	\u0000	16-bit-Unicode-Zeichen
byte	0	8-bit-Ganzzahl mit Vorzeichen
short	0	16-bit-Ganzzahl mit Vorzeichen
int	0	32-bit-Ganzzahl mit Vorzeichen
long	0	64-bit-Ganzzahl mit Vorzeichen
float	0.0	32-bit-Fließkommazahl
double	0.0	64-bit-Fließkommazahl

⌘ Boolescher Datentyp

- ⌦ true entspricht »wahr«
- ⌦ false entspricht »falsch«

16-bit-Unicode-Zeichen

- erste 256 Zeichen entsprechen Latin-1
- Zeichenkonstanten: 'z' (einfache Anführungszeichen)
- Escape-Sequenzen für Darstellung spezieller Zeichen

Esc-Seq	*Unicode*	*Bedeutung*
\b	\u0008	Backspace
\t	\u0009	horiz. Tabulator
\n	\u000A	Zeilenende (newline)
\f	\u000C	Seitenende (form feed)
\r	\u000D	Zeilenrücklauf (carriage return)
\\	\u005C	Backslash (\)
\"	\u0022	Anführungszeichen (double quote)
\'	\u0027	Apostroph (quote)
ooo		numerische Angabe des Zeichen als 3-stellige Oktalzahl

- Alle Zeichen dürfen in Unicode geschrieben werden
  - \u*hhhh* — *hhhh* ist 4-stellige Hexadezimalzahl

Exkurs: ASCII-basierte Zeichensätze

- ASCII – American Standard Code for Information Interchange
  - Genormter Zeichensatz für Schrift- und Steuerzeichen
  - 128 Zeichen
- Zahlreiche Zeichensätze erweitern den 7-Bit-ASCII-Code
  - 8-Bit Darstellung von Zeichen
  - z.B. Zeichensätze der ISO/IEC 8859-Reihe
    - Ergänzung um länder- oder sprachspezifische Zeichen
      - ISO-8859-1 (Latin-1) wird für die wichtigsten westeuropäischen Sprachen verwendet
- Die ersten 128 Zeichen stimmen bei allen ASCII-basierten Zeichensätzen überein

ASCII-Tabelle

	Rechtes Halbbyte															
Linkes Halbbyte	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0000																
0001																
0010		!	"	#	$	%	&	‘	(	)	*	+	,	_	.	/
0011	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0100	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0101	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0110	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0111	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Unicode

- International genormter Zeichensatz
- Ziel: einheitliche Codierung für jedes Textdokument aller Sprachen und Kulturen der Erde
- Enthält z.B. Zeichen für
  - Westliche und slawische Sprachen
  - Arabisch
  - Chinesisch
  - Mathematische und technische Symbole
  - usw.
- In der Version 4.1.0 gibt es 96.382 Zeichen
- Verschiedene Codierungen für Unicode-Zeichen
  - UTF-8, UTF-16 und UTF-32

Ganzzahlige Datentypen: byte, short, int, long

- Wertebereiche ermitteln

```
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+"]");
```

- Ausgabe

```
  byte=[-128..127]
 short=[-32768..32767]
   int=[-2147483648..2147483647]
  long=[-9223372036854775808..9223372036854775807]
```

int: entspricht $[-2^{31}..2^{31}-1]$
long: entspricht $[-2^{63}..2^{63}-1]$

- Darstellung als Dezimal-, Oktal- oder Hexadezimalwert
  - führendes Zeichen: `0` (Null) — Oktalzahl
  - führende Zeichen: `0x`, `0X` — Hexadezimalzahl
- Konstanten-Darstellung von long
  - Anhängen von `l` oder `L`
- Automatische Typanpassung, falls Wertebereich gültig ist
  - byte --> short --> int --> long
- Bei Division durch Null
  - `ArithmeticException`

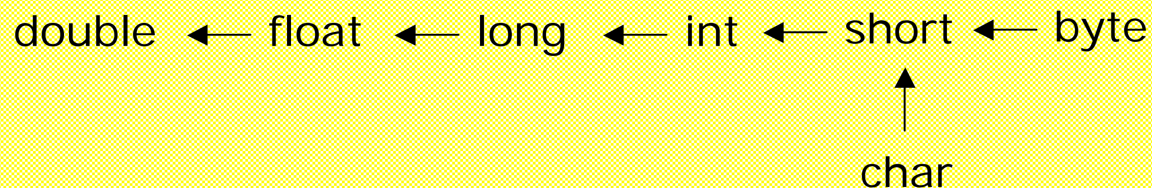
Fließkommatypen: float, double

- Wertebereiche

```
[Float.MIN_VALUE ..Float.MAX_VALUE ] --> [1.4E-45..3.4028235E38]
[Double.MIN_VALUE..Double.MAX_VALUE] --> [4.9E-324..1.7976931348623157E308]
```

- Darstellung in Exponentialschreibweise
  - mit/ohne Vorzeichen
  - mit/ohne Exponent `e` oder `E`
  - `18 18. 1.8e1 .18E2 0.018E3 +1800E-3`
- Konstanten-Darstellungen von float und double
  - float: Anhängen von `f` oder `F`
  - double: Anhängen von `d` oder `D`
- Fließkommaoperationen verursachen keine Ausnahmen
  - `System.out.println(5.0/0d); // Ausgabe: Infinity`

- Implizite Typumwandlung
  - Auswertung eines Ausdrucks liefert Typ B, aber Typ A wird erwartet
    - (implizite) Typanpassung wird versucht
  - Typumwandlung ist dann möglich, wenn eine Abbildung f: B → A existiert, mit der die Werte aus B nach A transformiert werden können.



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

Implizite Typumwandlung

⌘ 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);

              float g = 2 / 5;  // g = 0.0, d.h. erst Division,
                                // dann type conversion
              float h = 2 / 5F; // h = 0.4
              float j = 2F / 5; // j = 0.4
```

Explizite Typumwandlung (explicit casting)

- ermöglicht die Typumwandlung auch entgegengesetzt der Pfeile im Diagramm
- 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).
  - Angebebener Typ ist eine Verengung (narrowing).

Explizite Typumwandlung (explicit casting)

⌘ Beispiel: (narrowing)

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

      double g = 3.55;
      System.out.println( (int) g       ); // gibt 3 aus
                                           // (keine Rundung bei type cast)
      System.out.println( Math.round(g) ); // gibt 4 aus
                                           // (verwendet Funktion round
                                           // aus Bibliothek java.lang.Math)
```

Felder: Deklaration

- Eindimensionales Array:

```
int[] a; // eindimensionales Feld aus int-Werten
a = new int[DIM]; //Alternativ: int[] a = new int[DIM];
```

a[0]	a[1]	a[2]	a[3]	...	a[DIM-1]

- Mehrdimensionales Array:

```
int[][] b;      // zweidimensionales Feld
b = new int[DIMY][DIMX];
```

Höhere Dimensionen? weitere [...] anfügen

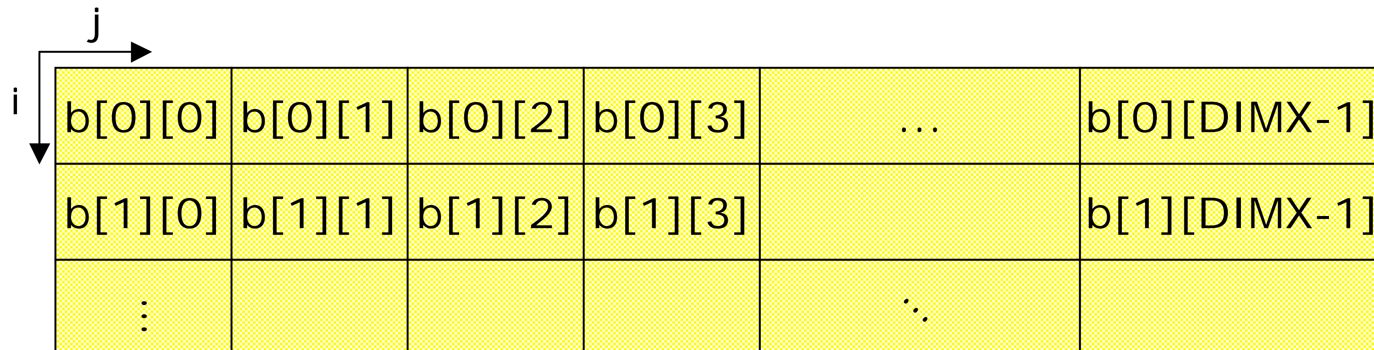
j →, i ↓

b[0][0]	b[0][1]	b[0][2]	b[0][3]	...	b[0][DIMX-1]
b[1][0]	b[1][1]	b[1][2]	b[1][3]		b[1][DIMX-1]
⋮				⋱	
				b[i][j]	

Felder: Zugriff auf Elemente

⌘ Hinter dem Namen des Arrays folgt der Index in eckigen Klammern:

```
int[] a = new int[DIM]; // eindimensionales Feld aus int-Werten
int[][] b = new int[DIMY][DIMX];  // zweidimensionales Feld
int i,k;
...
a[i] = ...                  // ganz normale Wertzuweisung
b[i][k] = ...
```



Syntax (vereinfacht):

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

Beachte

- Sei n die Länge eines Feldes a; Indizes laufen von 0 bis n-1.
- Wenn Index nicht aus [0, n-1]
  - Fehler: »array index out of bounds«.
- Die Länge des Feldes: a.length
- Typ des Index muss mit int verträglich sein.
- Alle Elemente sind vom gleichen Typ.
- Die Teilfelder mehrdimensionaler Arrays können verschiedene Längen haben.

- Teilfelder mehrdimensionaler Arrays können verschiedene Längen haben

- Beispiel (PartI):

```
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;
   }
}
```

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

- Operator precedence
  - Operatorvorrang (»Bindungsstärke«) wird berücksichtigt, ebenso Klammerung mit ( ).
- Eager evaluation
  - Erst werden die Operanden ausgewertet, dann die Operationen ausgeführt.
- Left-to-right-evaluation
  - Bei dyadischen Operatoren (binary operators) wird erst der linke Operand ausgewertet, dann der rechte (linksassoziativ).

- Ausnahme: Zuweisungsoperatoren
  - Bei Zuweisungsoperatoren wird erst der rechte Teil ausgewertet, bevor die Zuweisung erfolgt (rechtsassoziativ).
  - `=  +=  -=  *=  /=  %=  >>=  <<=  >>>=  &=  ^=  |=`

Regeln zur Auswertung von Ausdrücken

⌘ Beispiel für Left-to-right-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
  }
}
```

Mathematische Operatoren

- Addition, Subtraktion, Multiplikation, Division

  + - * /

- Vorzeichen-Operatoren

  + -

- Modulo-Operator

  %

Mathematische Zuweisungen

- Inkrementierung, Dekrementierung von x, nachdem x im Ausdruck verwendet wurde:

  `x++, x--`

- Inkrementierung, Dekrementierung von x, bevor x im Ausdruck verwendet wird:

  `++x, --x`

- Beispiel (RRZN-Javakurs, S.73):

```
int x=1, y;
y = ++x + 1; // x=x+1 (=2); y=x+1 (=3)
y = x++ + 1; // y=x+1 (=3); x=x+1 (=3)
x = x++ + 2; // x=x+2 (=5); x++ hat keinen Effekt
x = ++x + 2; // x=x+1 (=6); x=x+2 (=8)
x = x++;     // x=8; x++ hat keinen Effekt
```

Mathematische Zuweisungen

- Wertzuweisungsoperator
  - `=`
  - ist rechtsassoziativ!
  - a=b=c wird ausgewertet wie a=(b=c)

- Kurzformen
  - `x *= y      x = x * y`
  - `x /= y      x = x / y`
  - `x %= y      x = x % y`
  - `u.s.w.`

  - Beachte:
    - `a *= b + 1` bedeutet `a = a * (b+1)`

Logische Operatoren

- Vergleichsoperatoren
  - kleiner, größer, kleiner oder gleich, größer oder gleich

    `< > <= >=`
  - gleich, ungleich

    `== !=`
- Logische Operatoren
  - Negation, logisches UND, logisches ODER

    `! && ||`
- Instanceof-Operator
  - dient zum Feststellen, ob ein Objekt zu einer gesuchten Klasse gehört
  - Beispiel:

```
class Fahrzeug {
  public boolean equals(java.lang.Object obj) {
    if (obj instanceof Fahrzeug) return id==((Fahrzeug)obj).id;
    else return false;
  }
}
```

Bit-Operatoren und -Zuweisungen

- Bitweise-Verknüpfungen
  - binäres Komplement, bitweises UND, bw. ODER, bw. XOR

    ~ & | ^

- Bit-Schiebeoperatoren
  - x Linksschieben um y Stellen (Multiplikation mit 2^y)

    x << y
  - x Rechtsschieben um y Stellen (Vorzeichenbit füllt MSB)

    x >> y (arithmetic shift)
  - x Rechtsschieben um y Stellen (Auffüllen von Nullen)

    x >>> y (Division durch 2^y, logical shift)

- Bit-Zuweisungen
  - analog mathematischen Zuweisungen

    <<= >>= >>>= &= |= ^=

Bit-Operatoren und -Zuweisungen

- Beispiele
  - `int a=2:    // 00000000 00000010`
  - `a = a | 5; // 00000000 00000111 =  7dez`
  - `a = a & 3; // 00000000 00000011 =  3dez`
  - `a = a ^ 5; // 00000000 00000110 =  6dez`
  - `a = ~a;     // 11111111 11111001 = -7dez`

  - `int b=2;    // 00000000 00000010`
  - `b = b<<5;   // 00000000 01000000 = 64dez`
  - `b = b>>3;   // 00000000 00001000 =  8dez`
  - `b = b>>>2; // 00000000 00000010 =  2dez`
  - `b = -6;     // 11111111 11111010`
  - `b = b>>>2; // 00111111 11111110 = 1073741822dez`

Blöcke und Scopes

- Blöcke fassen Anweisungen (inkl. Deklarationen) zusammen.
  - durch geschweifte Klammern eingefasst.
  - Ausführung der eingeschlossenen Statements von links nach rechts und von oben nach unten.

- Syntax (vereinfacht):

  Block: { {Declaration | Statement} }

  Statement: Block | Assignment | Expression | ...

Blöcke und Scopes

- 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.
- **Gültigkeitsbereich** (scope) eines Namens erstreckt sich vom Ort seiner Vereinbarung bis zum Ende des zugehörigen Blocks
- **Sichtbarkeitsbereich** einer Variable (Konstante etc.) ist evtl. kleiner als der Gültigkeitsbereich ihres Namens, da sie durch Wiederverwendung des Namens in einem inneren Block verdeckt werden können.

Blöcke und Scopes

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

            void prozedur1() {
              i *= i;                          // i = 9
              prozedur2();
              n += 1;                          // n = 1
              i += 1;                          // i = 6
// Fehler     k = 0;                           // weil k nicht deklariert
            }

            void prozedur2() {
              int k = i + 1;
              int n = 5;                     // lokale Variable n
              i = n;                         // i = 5
            }

            public static void main(String[] args) {
              ScopeTest t=new ScopeTest();
              t.prozedur1();
            }

          }
```

Blöcke und Scopes

⌘ Syntaktisch korrekt, jedoch kaum noch lesbar: (und dazu noch unsinnig)

```
class Bezeichner {
   Bezeichner Bezeichner(Bezeichner Bezeichner) {
        Bezeichner:
        for(;;) { // Endlosschleife
                if(Bezeichner.Bezeichner(Bezeichner)==Bezeichner)
                        break Bezeichner;
        }
        return Bezeichner;
   }
}
```

Kontrollstrukturen

- 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 zusammengefasst werden.
- Verzweigung
  - Alternative (if-then-else)
  - Bedingungsoperator
  - Fallauswahl (switch-case)
- Schleifen
- Prozeduren und Funktionen

Alternative (if-then-else)

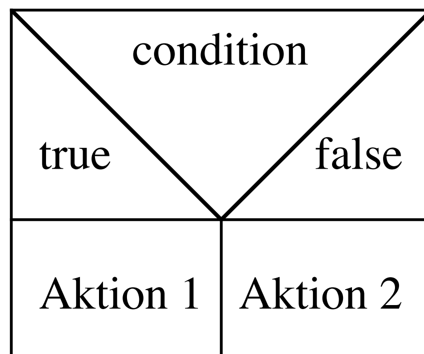
⌘ Syntax (vereinfacht):

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

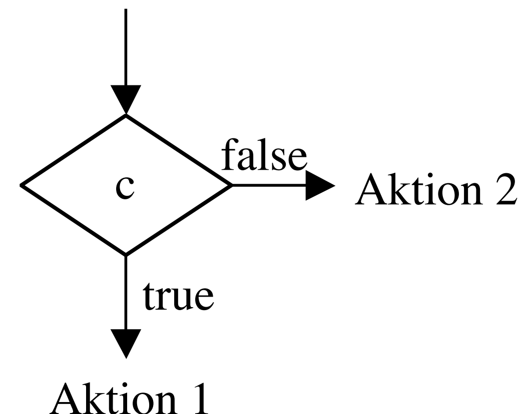
- Die Expression muss vom Typ boolean sein.

⌘ Semantik:

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



oder



Alternative (if-then-else)

⌘ 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--;
```

Bedingter Ausdruck

- Syntax (vereinfacht):

```
ConditionalOperator: Expression1 ? Expression2 : Expression3
```

  - Die Expression1 muss vom Typ boolean sein.
- Semantik
  - wie Alternative mit dem Unterschied, dass als Alternativen keine Statements, sondern Ausdrücke stehen
- Beispiele:

```
z = y + (x >= 0 ? x : -x); // Absolutwert von x wird auf y addiert

boolean b; // soll angeben, ob ein Geraet ein- oder ausgeschaltet ist
...
System.out.println( "Geraet ist " + (b?"ein":"aus") + "geschaltet" );
```

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

Fallauswahl (switch-case)

- Syntax (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.

Fallauswahl (switch-case)

⌘ 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!");
}
```

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.

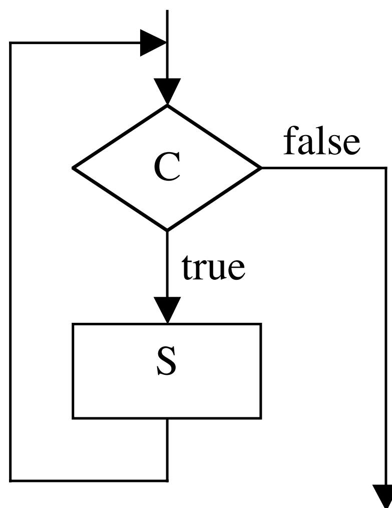
  - While
  - Do-While
  - Repeat-until (nicht in Java realisiert)
  - Laufanweisung

- Syntax (vereinfacht):

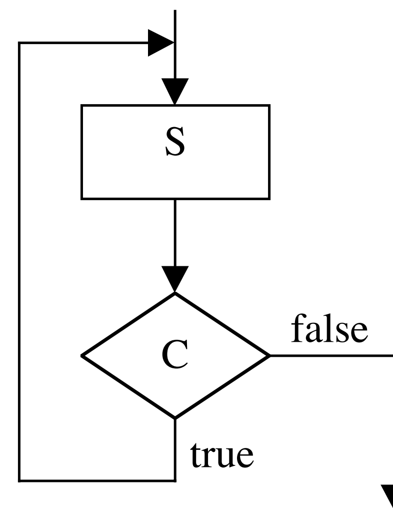
```
WhileStatement: while ( Expression ) Statement
DoStatement: do Statement while ( Expression ) ;
```

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

„pre-checked loop“:
while (C) S



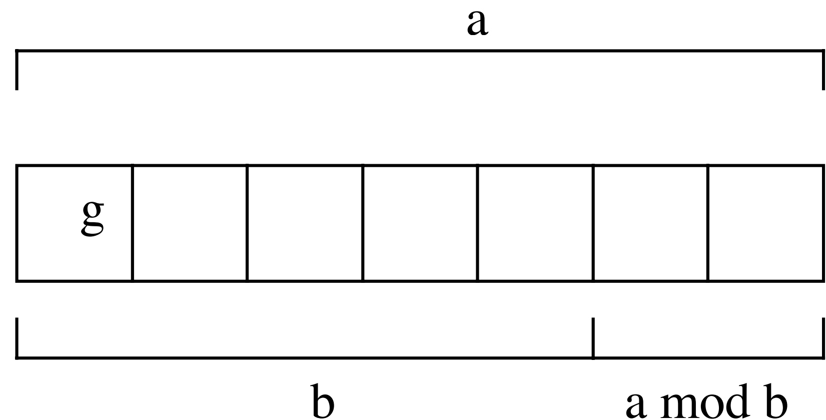
„post-checked loop“:
do S while (C)



Beispiel für while

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

```
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;
}
```

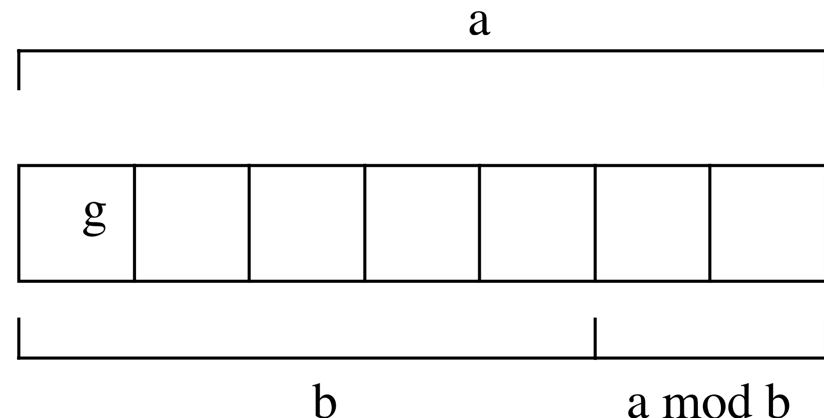


Algorithmus ggt(a,b)

- 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).

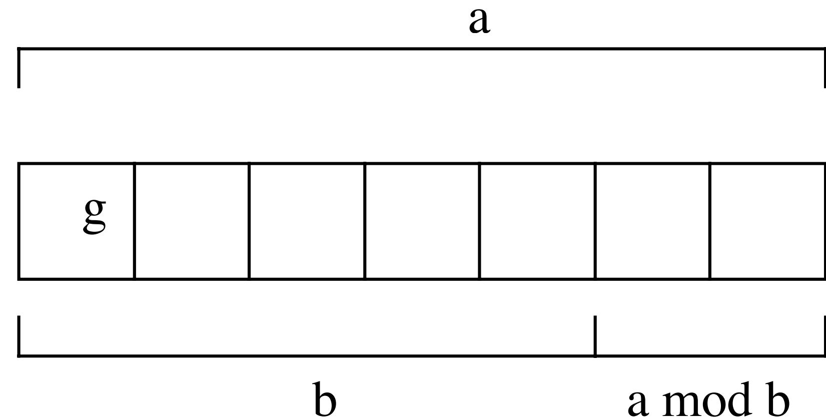
```
int getGGTOf(int a, int b) {
    int h;
    while (b != 0) {
        h = b;
        b = a % b;
        a = h;
    }
    return a;
}
```



Algorithmus ggt(a,b)

```
a       b       h
4       10          Beispiel: Aufruf von ggt(4,10)
                    (b!=0) ---> erster Durchlauf
                10          h=b
        4                   b=a%b (4:10=0R4)
10                          a=h
                    (b!=0) ---> zweiter Durchlauf
                4           h=b
        2                   b=a%b (10:4=2R2)
4                           a=h
                    (b!=0) ---> dritter Durchlauf
                2           h=b
        0                   b=a%b (4:2=2R0)
2                           a=h
                    (b==0) ---> Abbruch, Ergebnis steht in a
```

```
int getGGTOf(int a, int b) {
    int h;
    while (b != 0) {
        h = b;
        b = a % b;
        a = h;
    }
    return a;
}
```



Beispiel für do-while

⌘ 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);
      }
```

Beispiel für do-while

```
        // 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!");
    }
}
```

Laufanweisung

- Syntax (vereinfacht):

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

- Semantik:

for (I; C; U) S

  - Der Schleifenkörper S 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.

- Dauerschleife (Forever)
  - for(;;)

Beispiel für for

- Algorithmus zur Berechnung der Fakultät n! einer natürlichen Zahl n.

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

In einer Schleife von `i=1` bis `n` wird `i` mit dem derzeitigen Wert von `fact` multipliziert.

Abbruch von Schleifen

- Syntax (vereinfacht):

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

- Semantik
  - 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.

- ⌘ Beispiel:

```
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();
```

- 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

  - Wenn im obigen Beispiel continue anstelle von break geschrieben wird?
    → Fehler: "not a loop label: weiter".

Prozeduren und Funktionen

- Syntax (vereinfacht):

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

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

Prozeduren und Funktionen

- 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).

Prozeduren und Funktionen

- Unterprogrammaufruf (procedure call, method invocation):

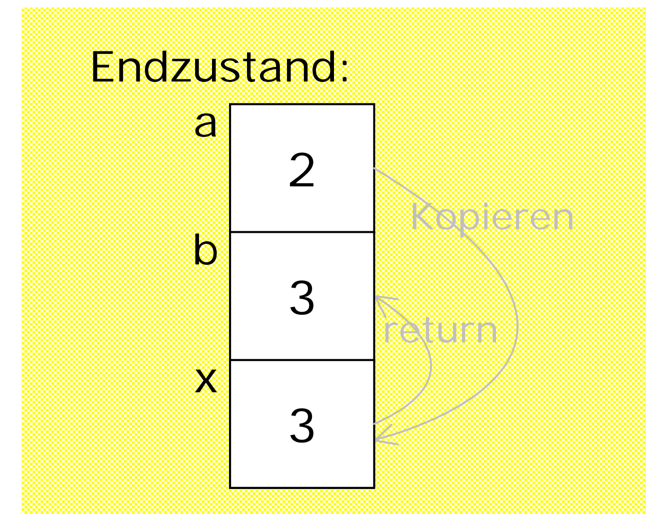
```
StatementExpression: ... | MethodInvocation
MethodInvocation: Identifier( [ActualArguments] )
ActualArguments: Expression {, Expression}
```

- Allgemein:
  - syntaktisch gesehen entweder Anweisung (bei Prozeduren) oder Ausdruck (bei Funktionen)
- Java:
  - stets Expression mit einem ResultType.
    - Bei Prozeduren: void

Prozeduren und Funktionen

- Parameterübergabe in Java:
  - by-value: Wert wird in lokale Variable kopiert

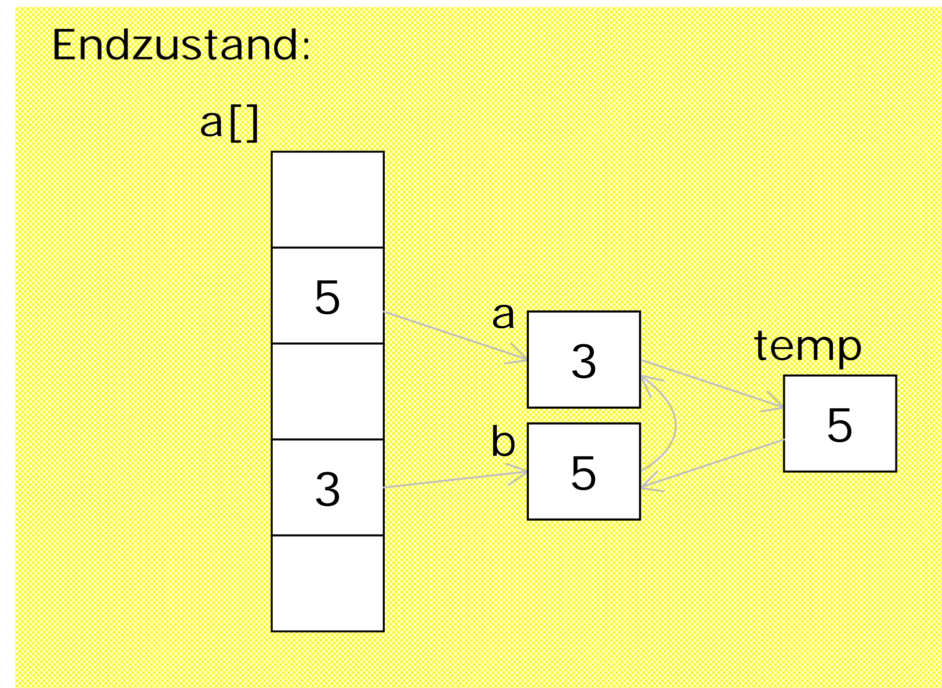
```
int funktion(int x) {
    // x ist eine lokale Variable
    x=x+1;
    return x;
}

int a=2;
int b=funktion(a);    // verändert den Wert von a nicht
System.out.println("a="+a);
System.out.println("b="+b);
```



- Parameterübergabe in Java:
  - by-value: Wert wird in lokale Variable kopiert

- Das wird nicht gehen:
  Vertauschen zweier Integer-Zahlen

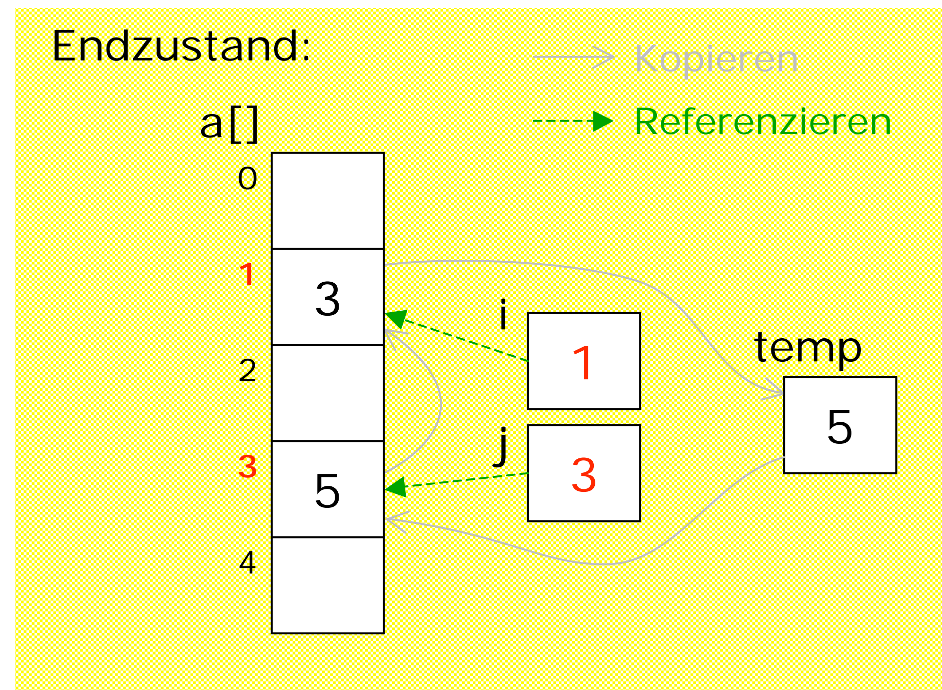
```
void swap(int a, int b) { // call-by-value
    int temp = a;
    a = b;
    b = temp;
}
...;
swap(a[i],a[j]);
...;
```



- Parameterübergabe in Java:
  - by-value: Wert wird in lokale Variable kopiert

- Geht: Übergabe der Referenz auf das Array und der Indizes
  Vertauschen zweier Integer-Zahlen

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



Sortieralgorithmen (Teil 1)

- Algorithmen
  - Bubble Sort
  - Selection Sort
  - Quick Sort
  - Merge Sort
  - Heap Sort

  (Quick Sort, Merge Sort, Heap Sort: Sommersemester)

- Aufgabe:
  - Sortieren eines Arrays von Integer-Zahlen `int[] a;`

Bubble-Sort auf einem Integer-Array

```
⌘ 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
       3      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 Elem.
1       2       A       F       E       C       H
        3       A       F       E       C       H
        4       A       F       E       C       H
                        ^               ^           // vertausche F mit kleinstem Elem .
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 $O(n^2)$.