Einführung in die Programmierung mit Java

1. Semester
WS 2019

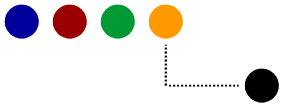
Prof. Dr. Herbert Neuendorf
DHBW Mosbach

herbert.neuendorf@mosbach.dhbw.de

DHBW Mosbach

© H.Neuendorf

Rahmenbedingungen



- Inhalte
- Literatur + Skript
- Tools
- Organisatorisches …

© H.Neuendorf

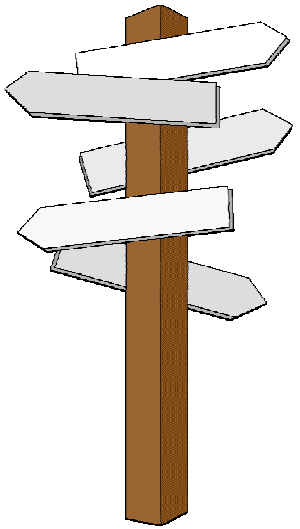
Inhalte

- **1.Teil : Grundlegende Konzepte der (Java-) Programmierung**

- **Grundlegende (prozedurale) Strukturen**
  - **Java** : Technische Grundlagen, Plattformunabhängigkeit, JVM, Bytecode ...
  - **Typkonzept** - primitive Datentypen, Casts, Zuweisungen, Operatoren ...
  - Grundlegender Aufbau von Java-Programmen
  - Kontrollstrukturen : Verzweigungen + Schleifen
  - Methoden, Rekursion
  - Blockkonzept, Namensraum, Sichtbarkeit

© H.Neuendorf

- **Grundlegende objektorientierte Strukturen (in Java)**
  - Sinn der Objektorientierung
  - Klasse, Objekt, Konstruktoren
  - Programmieren + Verwenden von Klassen - **Kapselung**
  - Speziellere Konzepte :
    - Klassenvariablen + Klassenmethoden (statische Elemente)
    - Referenzen - Zeigerkonzept
    - Referenz versus Kapselung
    - Immutable Objects
    - Primitive Datentypen + Referenzen : Call by Value + Call by "Reference"
    - Überladen von Methodennamen …



- **Arrays** – mit primitive Datentypen und Objekten

© H.Neuendorf

2.Teil : Vertiefung Objektorientierung

- **Konzepte der Vererbung**
  - **Implementierungs-Vererbung** von Attributen + Methoden
  - Objekterzeugung und Konstruktoren in Vererbungshierarchien
  - **Polymorphie und Typkompatibilität :**
  - Upcast + Downcast, statischer + dynamischer Typ, Späte Bindung
  - Abstrakte Klassen + Methoden / Finale Klassen + Methoden
  - **Interfaces - Schnittstellen-Vererbung**
  - Annotations

- **Paketkonzept** : (Keine Module)
  - Sichtbarkeit und Kapselung auf Paketebene
  - Aufbau JSE, Dokumentation, Archivierung, Tools …

Je nach Zeit evtl. erst im 2.Semester …

© H.Neuendorf

Ausblick

2. Semester

- **Vertiefung Objektorientierung – spezielle Techniken**
- Spezielle Interfaces, Klassen und Methoden
- Strukturierte Ausnahmebehandlung / Exceptions
- Reflection
- Generics
- Collection Framework
- Threading & Synchronisation
- Ereignisorientierte Programmierung

…

- **Algorithmen und Datenstrukturen …**

© H.Neuendorf

Literatur / Ressourcen

Deck, Neuendorf : Java-Grundkurs für Wirtschaftsinformatiker (Vieweg) ⊗

H. Mössenböck : Sprechen Sie Java? (dpunkt)

D.Ratz et al. : Grundkurs Programmieren in Java (Hanser)

2.Auflage

Vorlesungsfolien :

https://www.mosbach.dhbw.de/dhbw-mosbach/who-is-who/prof-dr-herbert-neuendorf

dort : *Downloads zu aktuellen Veranstaltungen*

Kein Skript ersetzt die Lektüre eines guten Buches !!

Ressourcen :

www.oracle.com/technetwork/java Java-Page von Oracle *Lizenzproblematik …*

dort : ***Software Downloads*** → ***Java SE***

https://docs.oracle.com/en/java/javase/index.html **JSE Doku**

www.eclipse.org → Download

Eclipse IDE for Java Developers

© H.Neuendorf

Voraussetzungen : *keine ... aber ...*

Erwartungen

- *Mitarbeit & Kritik(!)* ***während****(!) Vorlesung*
- *Fragen bei Unklarheiten !!*
- *Feedback, ob Tempo zu hoch / zu niedrig*
- ***Übungen*** *eigenständig durchführen*
- *Skript zur* ***Nachbereitung*** *verwenden !*
- *Anregungen zur* ***Verbesserung*** *der Veranstaltung !*

Ablauf :

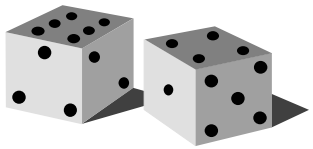
Vorlesung: Blöcke zu **4** h

Übungen: integriert

Software: **JDK 8** (optional höher)

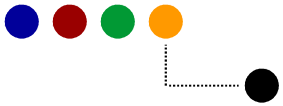
Eclipse IDE 2019 - ..

Klausur …



© H.Neuendorf

Umfeld



- **Informatik und ihre Gebiete**
- Programmiersprachen
- Motivation Java

Programming is not about writing code.

It is about building systems.

© H.Neuendorf

Was ist Informatik ?

Information + Mathematik + Automation
Ingenieurs- / Grundlagen-/ System-Wissenschaft

Wissenschaft von ...

- *der systematischen Verarbeitung und globalen Bereitstellung symbolischer Information, insbesondere zur automatisierten Wissensverarbeitung*
- *den Algorithmen & Datenstrukturen + deren Darstellung in digitalen* ***Systemen***

Speziell ***WI*** *: Analyse + Design von* ***Prozessen*** *und Darstellung mittels IKT*

Was kann Wie berechnet werden ?

Was kann Wie automatisiert werden ?

Was lässt sich Wie als ***System*** *darstellen ? …*

Architektur + Bau + Kontrolle großer ***verteilter Softwaresysteme***

Ziel : ***Beherrschung der System-Komplexität !***

Mittel : ***Modularisierung + Kapselung + Information Hiding!***

Europa :
Informatik

USA :
Computer Science

© H.Neuendorf

Teilgebiete

Gesellschaftlicher Anspruch

*"[…] allen Menschen einen selbstbestimmten Umgang mit ihren Daten zu ermöglichen und dies zu gewährleisten." (**GI** 2010)*

Informatik

Technische Informatik

Hardware

Prozessoren

Mikrocontroller

Embedded Devices

Rechnerarchitektur

Netzwerke

…

Praktische Informatik

Algorithmen

Datenstrukturen

Sprachen

Design Patterns

Betriebssysteme

Software Engineering

Ergonomie

DS / ML

…

Theoretische Informatik

Automatentheorie

Theorie der formalen Sprachen

Compilerbau

Theorie der Berechenbarkeit

Algorithmenanalyse

KI

…

Angewandte Informatik

Informationssysteme

Digitale Medien

Modellierung & Simulation

⇓

Wirtschaft

Ingenieurwissenschaften

Naturwissenschaften

Medizin

…

John von Neumann

(1907 - 1957)

"*Wörter, die Befehle kodieren, werden im Speicher genauso verarbeitet wie Zahlen.*"

Informatik = Querschnitts-Wissenschaft :

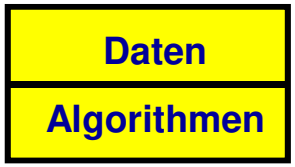
Sichtbar in **Integrierten** Informationssystemen :

ERP = Enterprise Resource Planning Systems

Die Komplexität eines modernen Unternehmens wird detailliert modelliert + in Software-Architekturen umgesetzt

© H.Neuendorf

Komplexität und Modularisierung



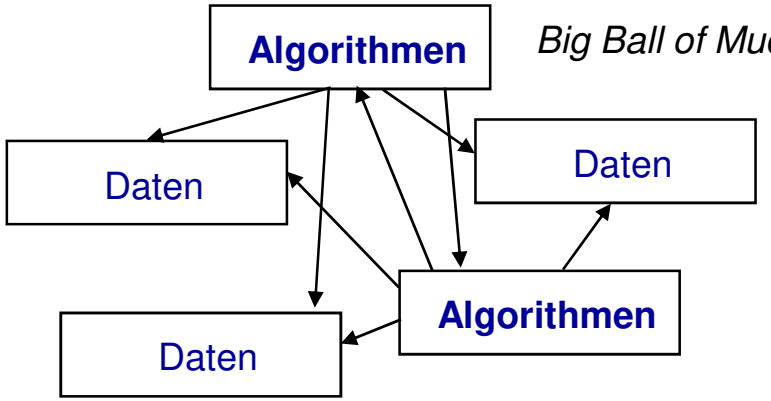
Programm = Daten + Anweisungen

Ziel : *Reduktion der Komplexität*

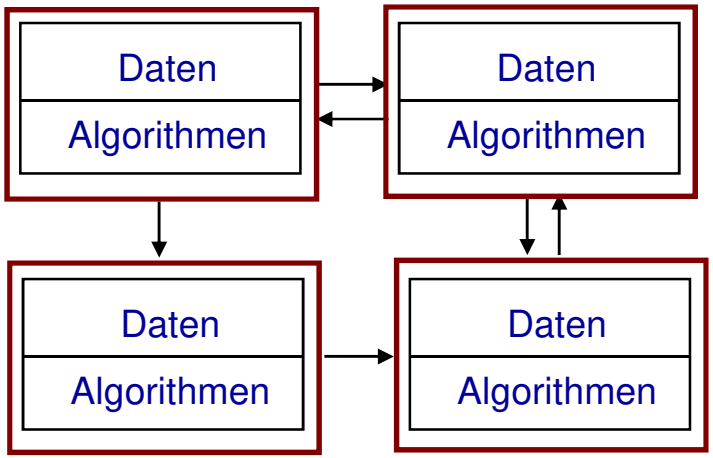
"*Einen neuen Code zu schreiben ist leichter, als einen alten zu verstehen.*"

John von Neumann

⇒ **Modularisierung + Kapselung ↔ keine "ungeschützen" Daten**



Big Ball of Mud



Reduktion unkontrollierter Zugriffe :

Kapselung von Daten + Algorithmen

→ Einführung von **eingeschränkter Sichtbarkeit !**

Schnittstellenprinzip

→ **Information Hiding / Geheimnisprinzip**

⇒ **Objektorientierung – Begriff der Klasse :**

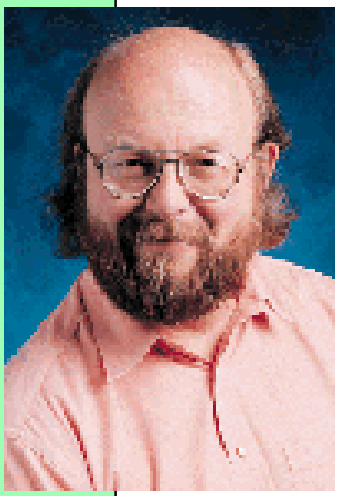
Klasse = Daten + Methoden

in gekapselter Form

© H.Neuendorf

Historie der Programmiersprachen

B. Stroustrup



James Gosling

Motive für Java : *von **C++** inspiriert - jedoch "bereinigt & vereinfacht" ...*

Vermeiden der Komplexität von **C++** → hybrider Sprachumfang, echte Pointer, Speicherverwaltung ...

Java → klar objektorientiert, überschaubarer Sprachumfang, streng typisiert, Garbage Collection ...

© H.Neuendorf

Java Historie + Motivation :

"OAK"- Projekt von SUN Microsystems → Sprache für Consumer Electronics ("IoT")

1990: Beginn der Entwicklung von Java durch Gosling & Joy

1993: **WWW**

1995: **JDK 1.0** von Sun + Browser-Unterstützung (*Applets!*)

1998: **JDK 1.2** - deutliche Erweiterung (**Java 2**) [aktuell JDK **13**]

Ständige Weiterentwicklungen :

JDK 1.0 ≈ 210 Klassen

JDK 1.1 ≈ 650 Klassen

JDK 1.2 ≈ 1250 Klassen ...

Java = Plattformunabhängigkeit

⇒ Konkurrent **Microsoft** (+ Apple)

⇒ **.net** : **C#** Common Language Runtime

Aktuell :

Funktionale Sprachen auf JVM (Scala, Clojure)

Modularisierung / Profiles …

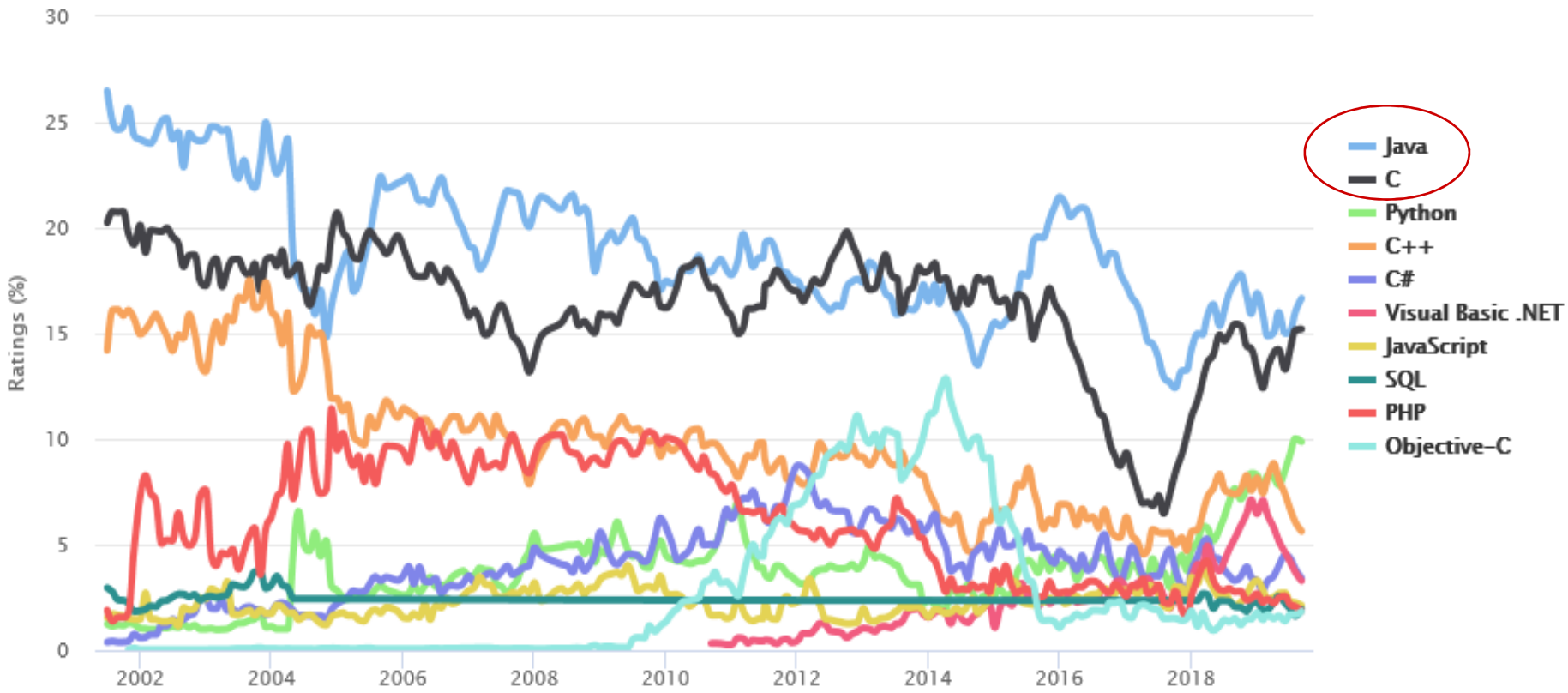
Java im Vergleich zu **C++ :** Weniger komplex, klarer, beherrschbarer - deckt **90%** aller Fälle ab

⇒ **" Langsamer für die Maschine, aber schneller für den Menschen " (= Entwickler)**

Prinzip der Orthogonalität - nur *ein* Konzept pro typischer Problemstellung

© H.Neuendorf

Relative Bedeutung von Java

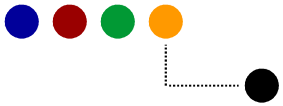


Quelle : **Tiobe Index for September 2019**

www.tiobe.com/tiobe-index/

© H.Neuendorf

Java-Infrastruktur



- **Java als Sprache und Plattform**
- Java-Editionen und -Umgebungen
- Konzept der Java Virtual Machine (**JVM**)
- Java-Bytecode
- Aufbau der JVM

… *Plattformunabhängigkeit* …

© H.Neuendorf

Java = Sprache + Plattform = "Systemsprache"

Programmiersprache :

"100%" **objektorientiert** – "alles" als Klassen

Klare Syntax ohne Altlasten

Vermeiden komplexer Sprachmittel von C++

Speicherverwaltung *nicht* programmiert, sondern durch Garbage Collector geleistet

Threading & Synchronisation

Plattform :

Fülle von **Klassenbibliotheken (Paketen) :**
Einheitliche Schnittstellen für alle Betriebssysteme

Umgebung, die Unterschiede der realen Betriebssysteme + Hardware "verbirgt" (kapselt) = **Virtuelle Maschine :**

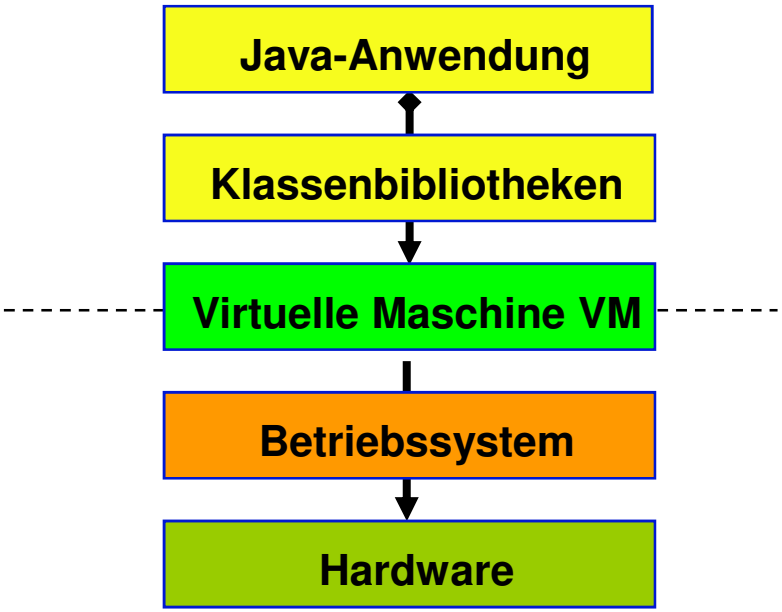
Für Entwickler stellt sich Java wie eigenständige Plattform dar ⇒ **Plattform-Unabhängigkeit !**

Editionen : *u.a. …*

Standard Edition für Fat Clients **Java SE**

Enterprise Edition für App-Server **Jakarta EE**

Micro Edition für Mobile Devices **Java ME**



© H.Neuendorf

Wo "lebt" Java ? → Java-Laufzeitumgebungen

1. Client JR für Applikationen

(2. Webbrowser mit Applets)

3. Webserver mit Servlet Container

4. Application Server mit EJB-Container

Nutzen Java-Klassenbibliotheken + **JVM**

Unterschiede bestehen in Anwendungs-Struktur und Umgebung, in der **JVM** arbeitet

Standalone-Applikationen :

Programme ohne spezielle Umgebungsanforderungen

Applets :

Liefen im Browser mit integrierter JVM

Servlets + JSPs :

Generieren auf Webserver dynamisch HTML

Enterprise Java Beans (EJB) :

Services, die ein Enterprise Application Server bereitstellt - transaktionale Dienste / Messaging …

www.oracle.com/technetwork/java

JDK : Java Development Kit

Laufzeitsystem + Entwicklungstools

JRE : Java Runtime Environment

= nur JVM-Laufzeitbestandteile

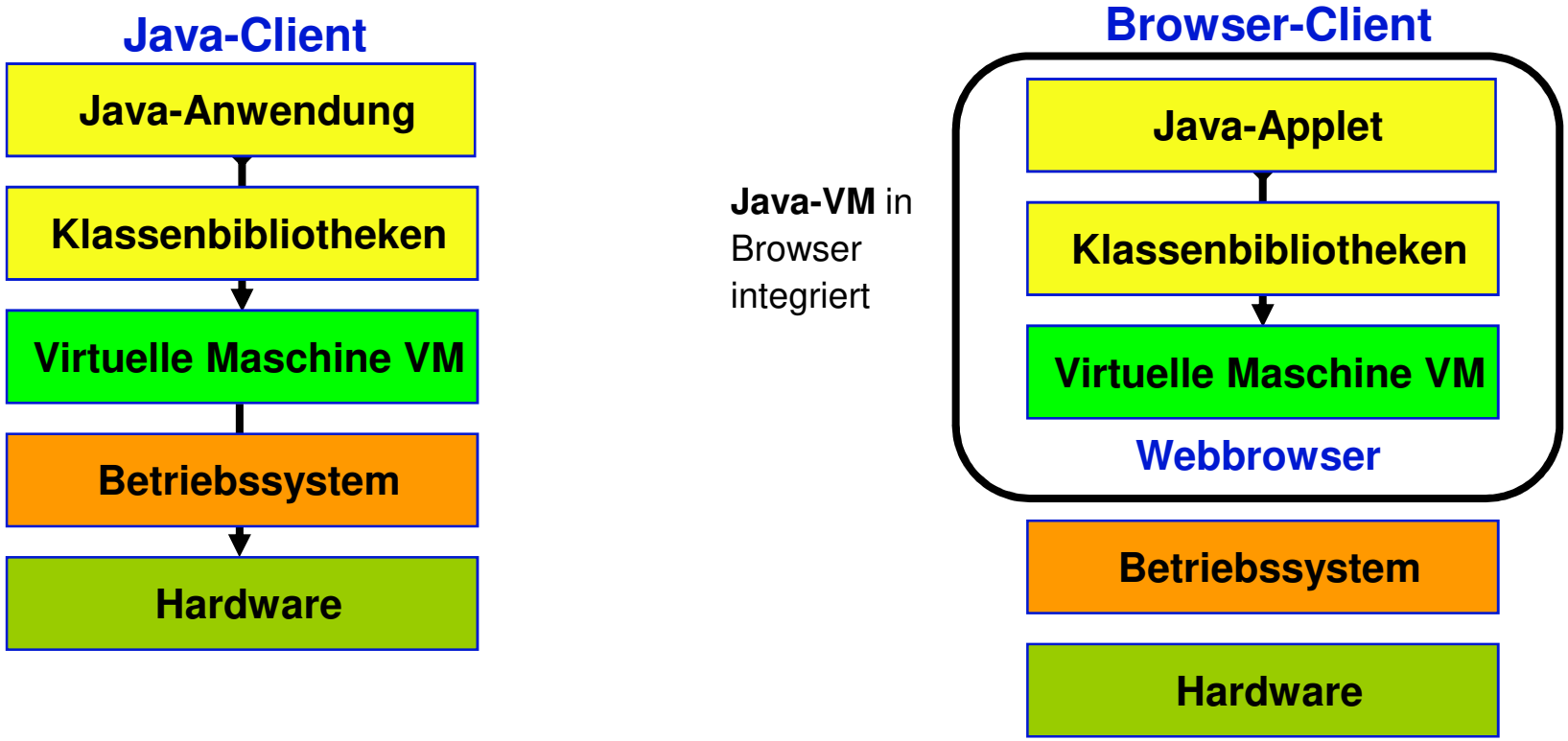
VM : Java läuft überall ... !!

Vom Smartphone bis zum Server ... ⇒

*In allen Ebenen **Verteilter Systeme***

© H.Neuendorf

Varianten der Java-Laufzeitumgebung

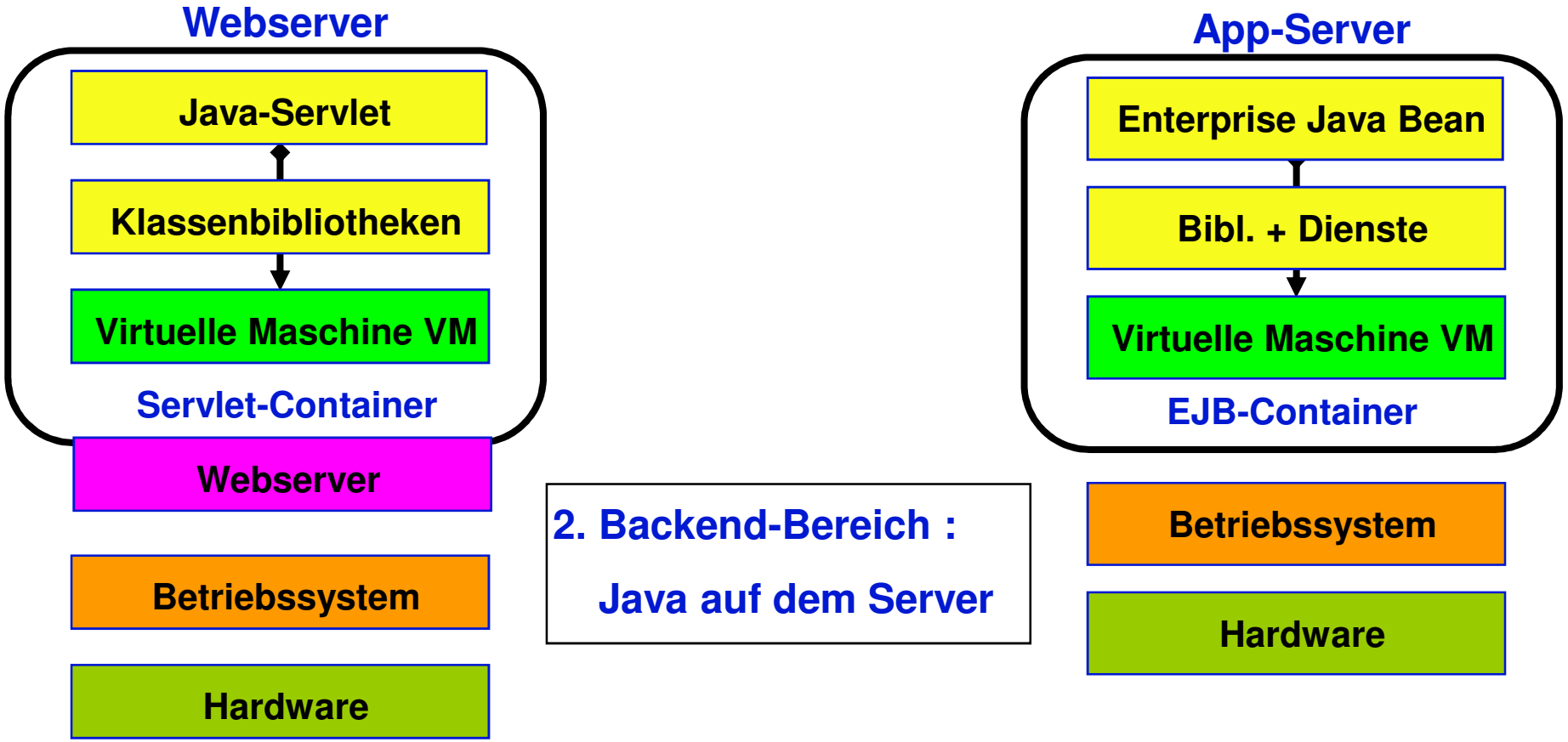


1. Frontend-Bereich: Fat Client, Browser - lokaler Rechner

= ursprünglicher Anwendungsbereich von Java

© H.Neuendorf

Varianten der Java-Laufzeitumgebung



2. Backend-Bereich :

Java auf dem Server

3. Middleware-Bereich :

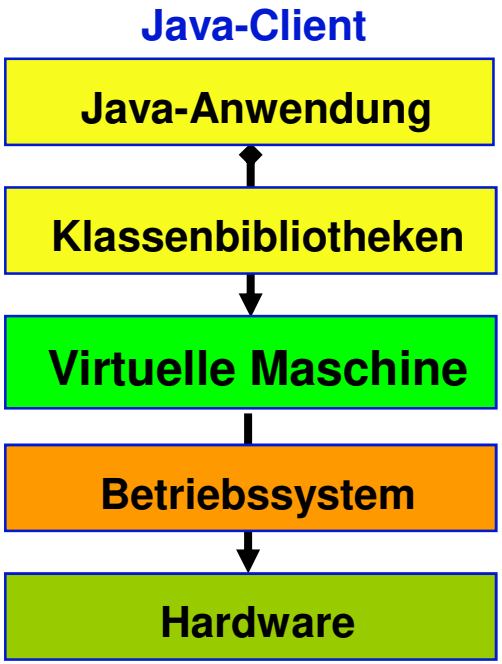
Kommunikation von Objekten in verteilten Systemen

Netzwerk / Client-Server / DB-Zugriff / … :

Sockets, RMI, JDBC, Message Queues + Broker, WebServices, MicroServices ...

© H.Neuendorf

Java-Laufzeitumgebung JVM



Konzeption von Java → Write once - run everywhere

Plattformunabhängigkeit !

Keine Annahmen über *Zielhardware* oder *Betriebssystem* !

VM : Zwischenschicht, die BS und Hardware kapselt

"Prozessor" - in Software realisiert

- als **Maschine** definiert, die Maschinenbefehle interpretiert
- setzt Anweisungen in plattform**spezifische** Anweisungen um

⇒ **VM muss für jede Plattform, auf der Java-Programme laufen sollen, speziell implementiert werden**

Konsequenz : **Java-Programm …**

→ **wird von Java-Compiler in Code für die VM kompiliert = Bytecode**

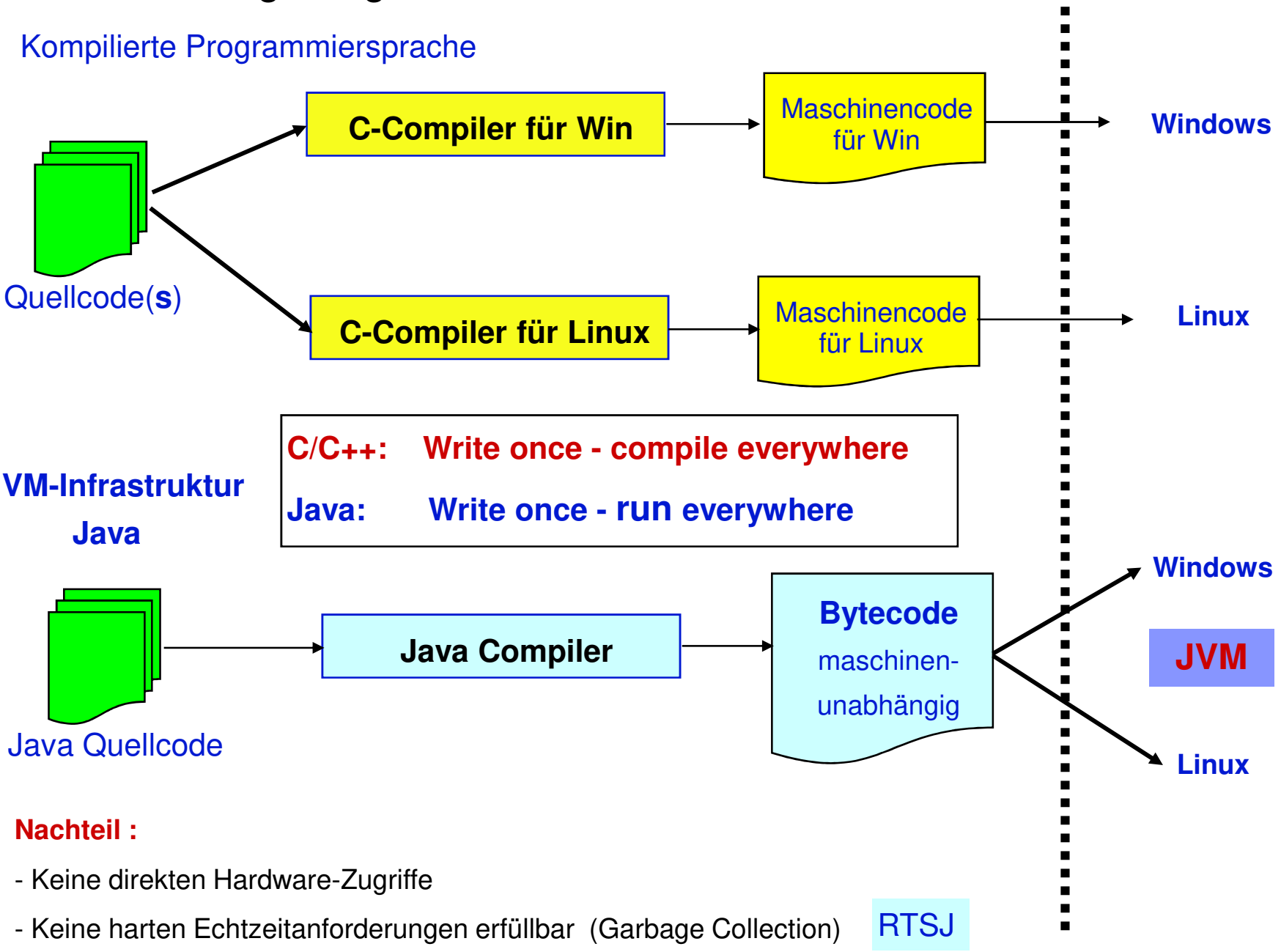
→ **kann von jeder Java-VM ausgeführt = interpretiert werden !**

→ **läuft auf jeder Plattform (BS, Hardware) *für die* VM zur Vfg steht !**

⇒ **Kein erneutes Anpassen, Kompilieren, Linken …**

© H.Neuendorf

Java-Laufzeitumgebung



Nachteil :

- Keine direkten Hardware-Zugriffe
- Keine harten Echtzeitanforderungen erfüllbar (Garbage Collection)

RTSJ

© H.Neuendorf

Interpretierte Sprachen - Bytecode kompilierte Sprachen

Deutliche Unterschiede bei Performanz !

Interpreter : Ausführung ohne Optimierung

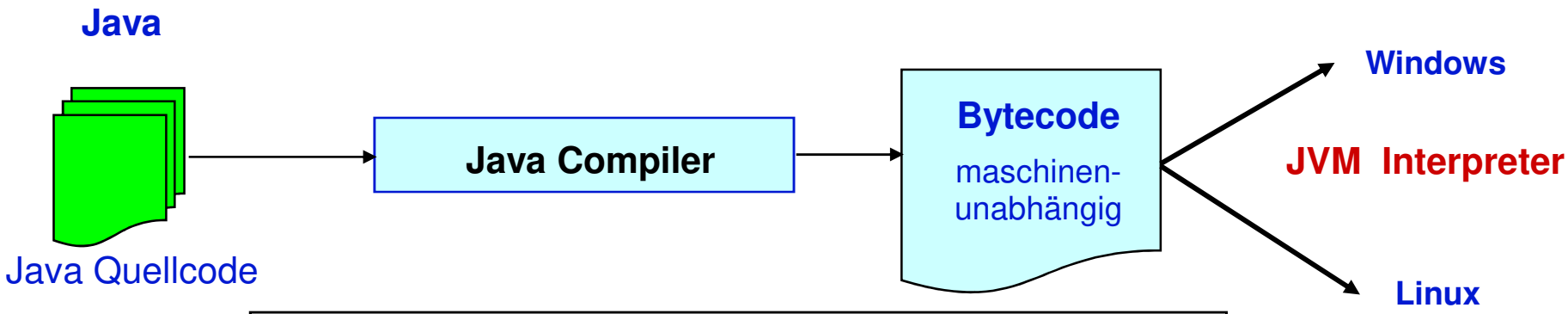
→ Durchläuft Programmtext, übersetzt + führt jede Zeile immer wieder einzeln aus ⇒ **Langsam !**

Bytecode-Architektur : Zweiteilung der Programmbearbeitung - **C**ompiler **+ I**nterpreter

→ 1. Schritt **C** : Parsen + Analysieren des Programmtextes (*+ Finden von Fehlern!*)
Übersetzen in optimierte VM-Instruktionen = Bytecode

→ 2. Schritt **I** : Ausführen der im Bytecode enthaltenen VM-Instruktionen

⇒ **"semi-kompilierte" Sprache : Viel schneller als reine Interpreter**
Langsamer als reine Compiler (C/C++)



VM = zusätzliche Schicht zwischen Bytecode und Ausführungsplattform ⇒ **Kostet Zeit !**

© H.Neuendorf

Bytecode = Maschinensprache der JVM

Opcodes (Maschinenbefehle) für JVM

Haben (8-bit) **Nummer + Namen**

zB: Opcode **96** = iadd = Addiere zwei Integer

Analyse : JDK-Tool **javap**

Optionen : javap –help ↵

Ausgabe Opcodes :

javap –c –private *nameClassFile* ↵

Bsp :

Java-Anweisung: int y = x + 5 ;

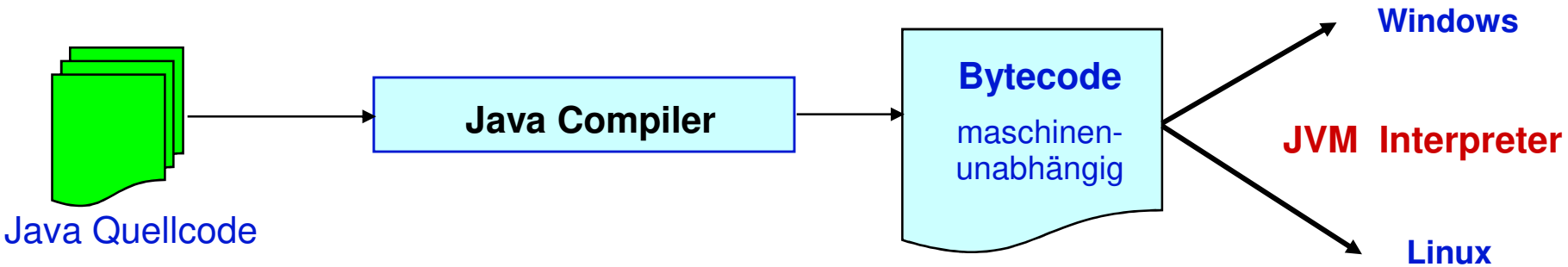
JVM-Befehlsfolge:

iload_1	lege Integerwert aus Variable 1 auf den Stack
iconst_5	lege konstanten Wert auf den Stack
iadd	addiere die beiden Integer-Zahlen
istore_2	speichere das Ergebnis in der Variablen 2

The Java® Virtual Machine Specification
Java SE 10 Edition
Tim Lindholm
Frank Yellin
Gilad Bracha
Alex Buckley
2018-02-20

Befehle in Bytecode übertragen :

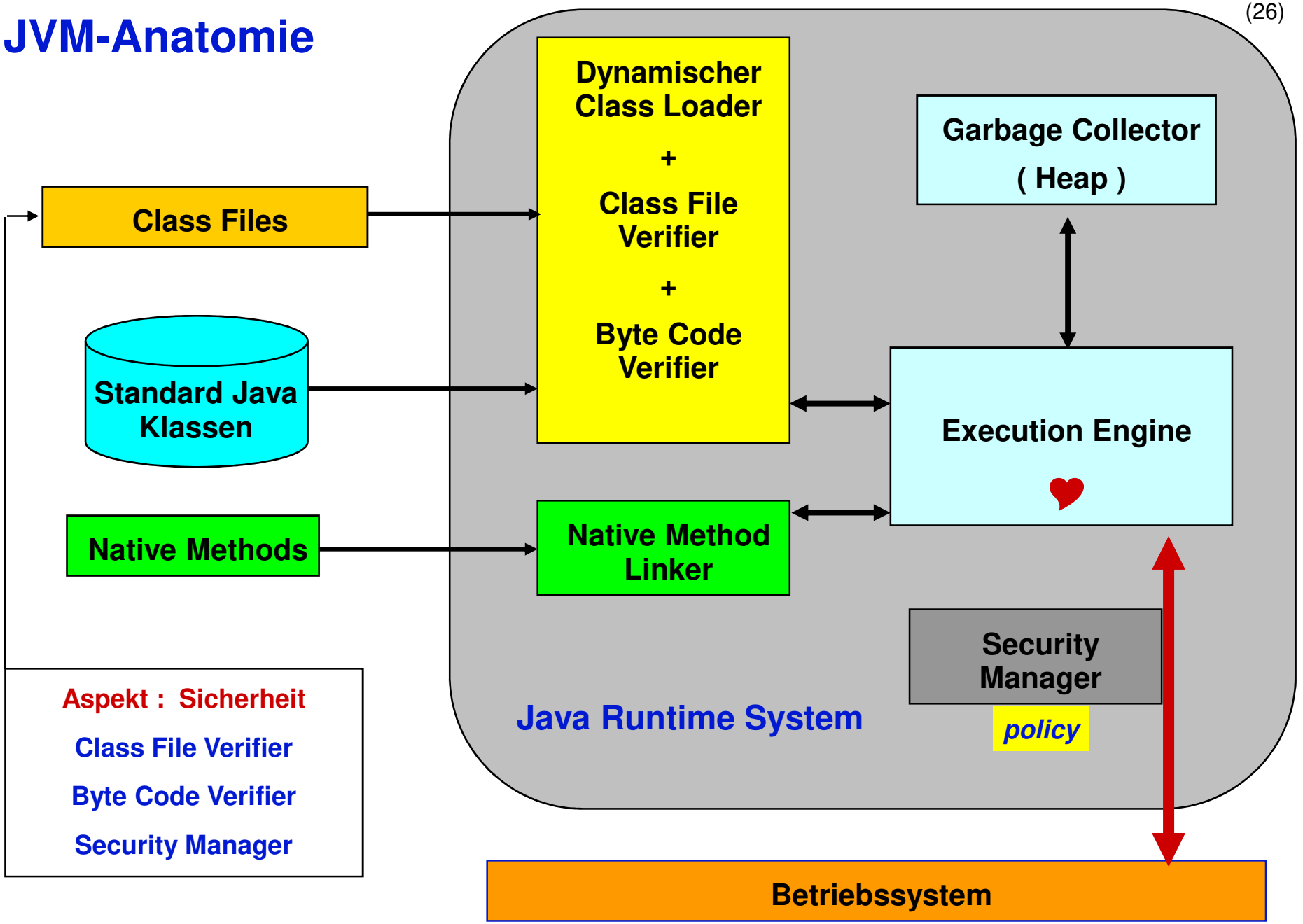
In Bytecode :	27	8	96	61
	iload_1	iconst_5	iadd	istore_2



JVM Spezifikation : > 200 Instruction Opcodes

© H.Neuendorf

JVM-Anatomie



© H.Neuendorf

Java-Virtual Machine

Execution Engine : - "Herz" der JVM = **virtueller Prozessor** der **virtuellen Maschine**

- Bewirkt Ausführung der Bytecode-Instruktionen

Class Loader : - Lokalisieren + Laden + Verifizieren + Initialisieren der benötigten Klassen **.class**-Files = Bytecode

- dynamisch: **erst wenn** Klasse benötigt, wird sie in JVM geladen

Garbage Collector : - Objekte werden im Hauptspeicher angelegt - *new*

- Speicher-Freigabe *nicht* durch Programm - *kein delete*
- sondern durch periodisch laufende Speicherverwaltung der JVM

Native Method Linker : - Aufruf von Routinen, die als **plattformspezifischer** Maschinencode vorliegen

- nicht Java-Bytecode, sondern Kompilat anderer Sprache (DLLs, libs)

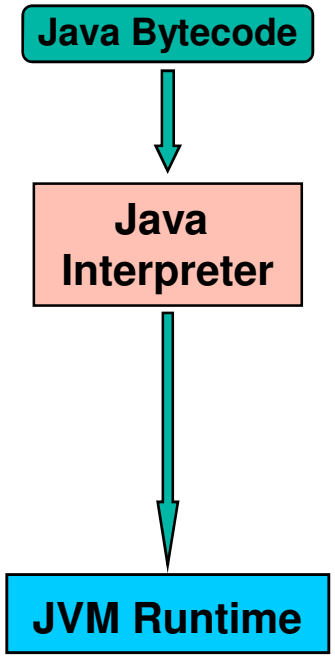
Security Manager : - Java-Laufzeitsystem definiert **Sicherheitsrichtlinien**

- bei Programmausführung vom Security Manager durchgesetzt :
- Überwachung von Java-Programmen = Durchsetzen von Restriktionen

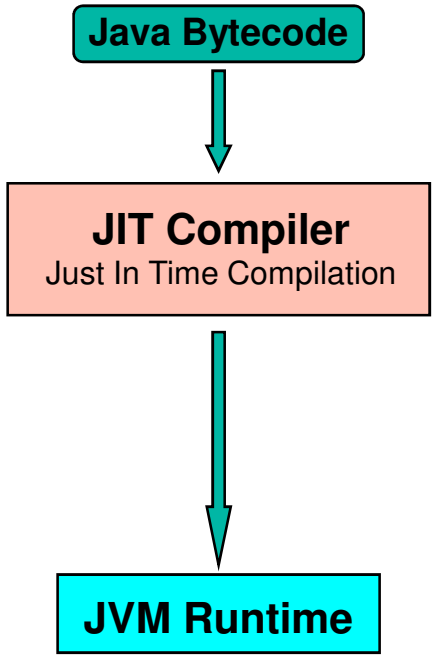
zB: Zugriff auf lokales Filesystem, Reflection, Netzzugriff …

© H.Neuendorf

Weiterentwicklung Java VM : **Java Hot Spot VM**



Schlechte Performance



Gute Performance

JDK 1.4 (2002)

Java Hot Spot VM in zwei Varianten :

Server VM und **Client VM**

mit unterschiedlich optimiertem Start- und Ausführungsverhalten

Seit **Java 10** (nur 64 bit)

Nur noch Server VM

Bytecode wird durch **JIT Compiler** beim JVM-Start *möglichst schnell* in Maschinencode übersetzt.

Oft auszuführende Programm**teile** (**hot spots**) werden durch **Hot Spot JIT Compiler** zur **Laufzeit** zudem in *möglichst schnellen* Maschinencode übersetzt (*optimierte* teilweise Neuübersetzung von Maschinencode)

Java Hot Spot VM

Java Programme können **WarmUp-Verhalten** zeigen:

Anfangs *langsamere*, nach einiger Zeit *schnellere* Ausführung

(bei Messungen beachten!)

© H.Neuendorf

Warum Java ?

“Java: A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language ... ” (Sun: The JavaLanguage – White Paper)

einfach: Keine Zeigerarithmetik, Mehrfachvererbung, komplizierte C++ Konzepte

objektorientiert: Klassenenkonzept, Vererbung, Interfaces …

Kleiner Sprachumfang + umfangreiche Klassenbibliotheken

verteilt: Unterstützt netzbasierte, verteilte Anwendungen

interpretativ: Bytecode-Ansatz verbindet Vorteile interpretativer + kompilierter Sprachen

robust: Fehlerquellen aus C++ vermieden, Garbage Collection

sicher: Bytecode Verifier, Security Manager …

architektur-unabhängig: Bytecode + APIs abstrahieren von Hardware + BS

portierbar: JVM interpretiert identischen Bytecode auf allen unterstützten Plattformen

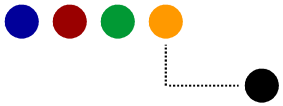
leistungsstark: Langsamer als C++ - aber meist ausreichend !

Thread-unterstützend: Sprachelemente zur Synchronisation paralleler Threads

dynamisch: Dynamisches Laden + Initialisieren nur benötigter Klassen zur Laufzeit

© H.Neuendorf

Java-Programmerstellung



- Klassische Compilersprachen
- Programmerstellung / -Ausführung unter Java

© H.Neuendorf

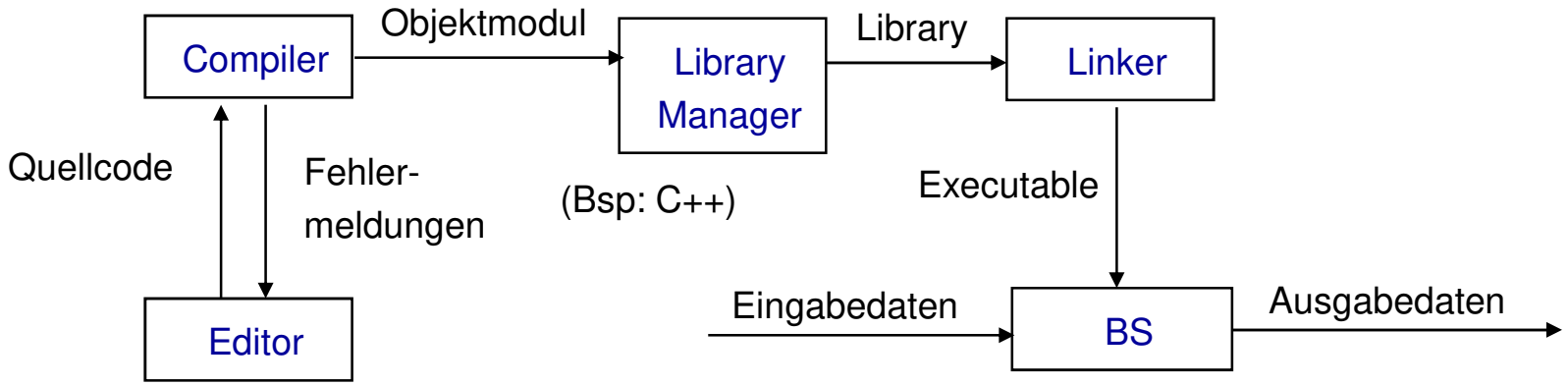
Programmerstellung – Kompilierte Sprachen

Programm erstellen : Code in Quellcodedatei

Programm kompilieren : Maschinensprache in Binärdatei

Compilation :

Vorteil → Optimierung + Check Programm vor(!) Ausführung auf Fehlerfreiheit



⇒ **Optimale Übersetzung zur Ausführung auf speziellem BS / Prozessor**

⇒ **Ausführung auf verschiedenen Plattformen (Win, Linux) erfordert Neu-Compilieren !**

Nicht ideal für Anforderungen heterogener Umgebungen (Web !)

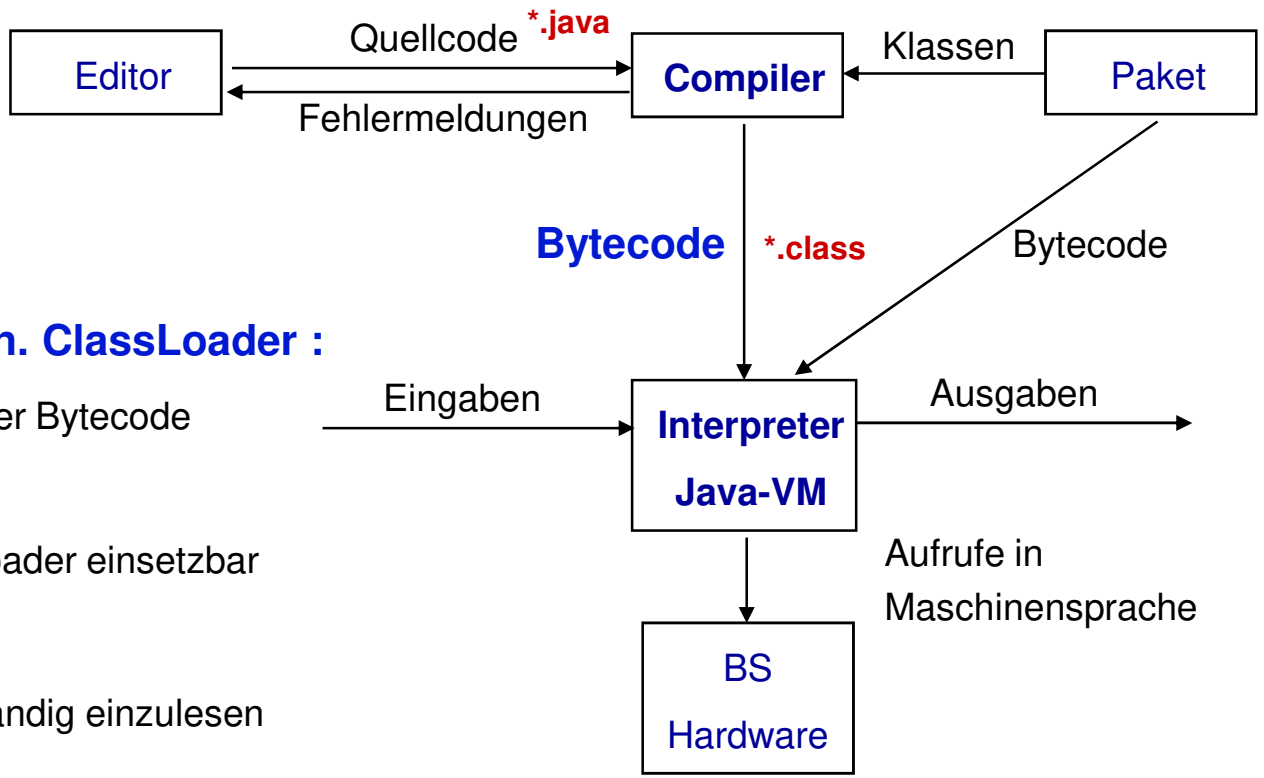
Gefordert ist eher "Write & compile once, run everywhere“ ... ***Java ist anders !***

© H.Neuendorf

Kompilieren + Interpretieren in Java

Java-**Übersetzungseinheit**
ist das **.java - File**

JDK-Compiler **javac**



Vorteil Compiler + JVM + Dyn. ClassLoader :

Plattformunabhängiger + effizienter Bytecode

Prüfungen schon bei Compilierung

HotDeployment + eigene ClassLoader einsetzbar

Nachteil :

Bytecode-Dateien zur LZ zeitaufwändig einzulesen

Compiler erzeugt plattformunabhg. Bytecode ⇒ durch JVM ausgeführt

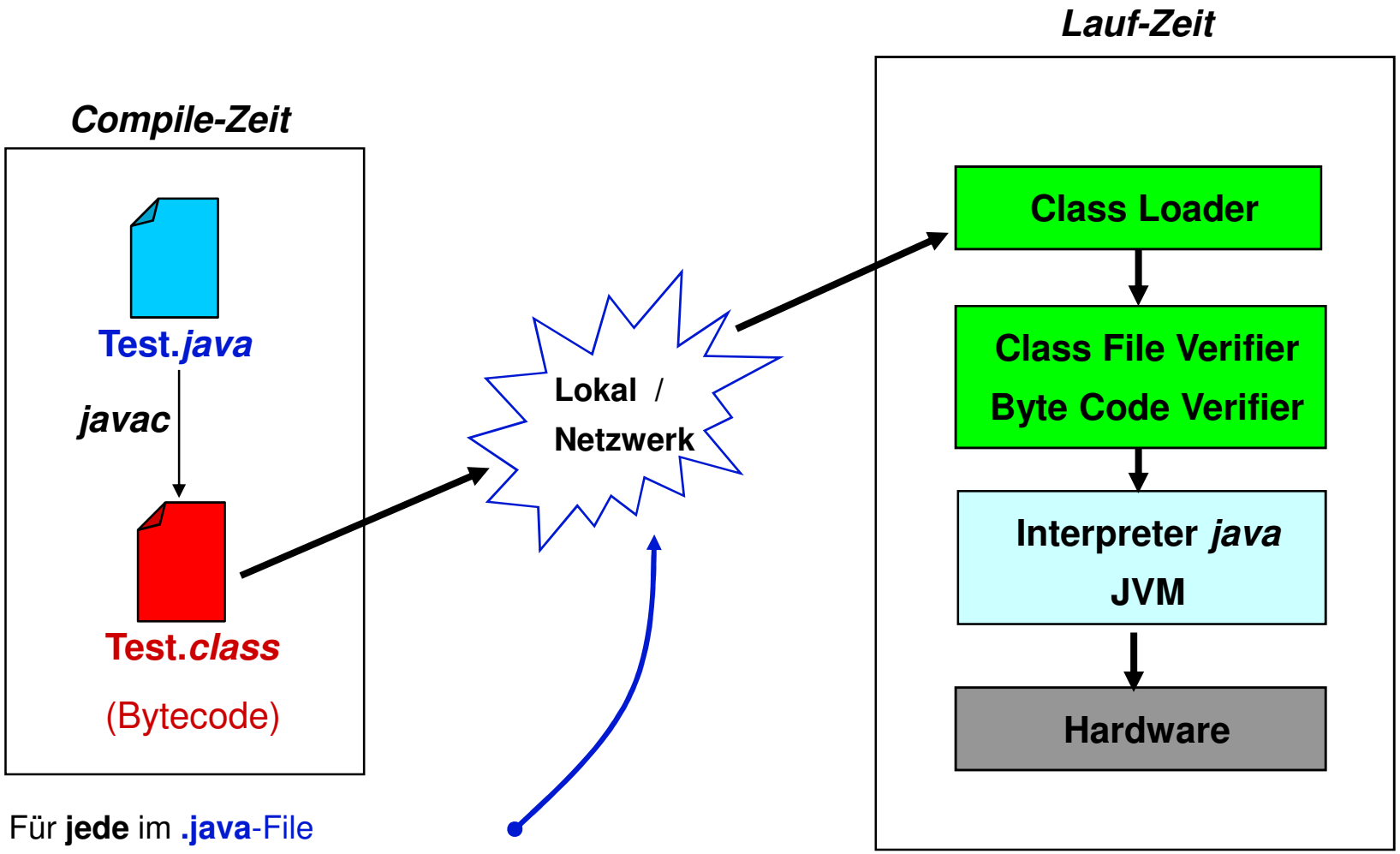
Java Virtual Machine = Java-Laufzeit-Umgebung :

Vollzieht kontrollierte Bytecode-Übersetzung & -Ausführung auf spezieller Plattform

Java-VMs für relevante Plattformen zur Verfügung - verstehen alle den selben Bytecode

© H.Neuendorf

Erstellen + Ausführen von **Java**-Programmen



Für **jede** im **.java**-File enthaltene **Klasse** wird **eigenes .class**-File vom Compiler generiert

Benutzte Klassen (class-Files) werden zur **Laufzeit** bei **Bedarf** geladen !

© H.Neuendorf

Erstellen + Ausführen von **Java** Programmen

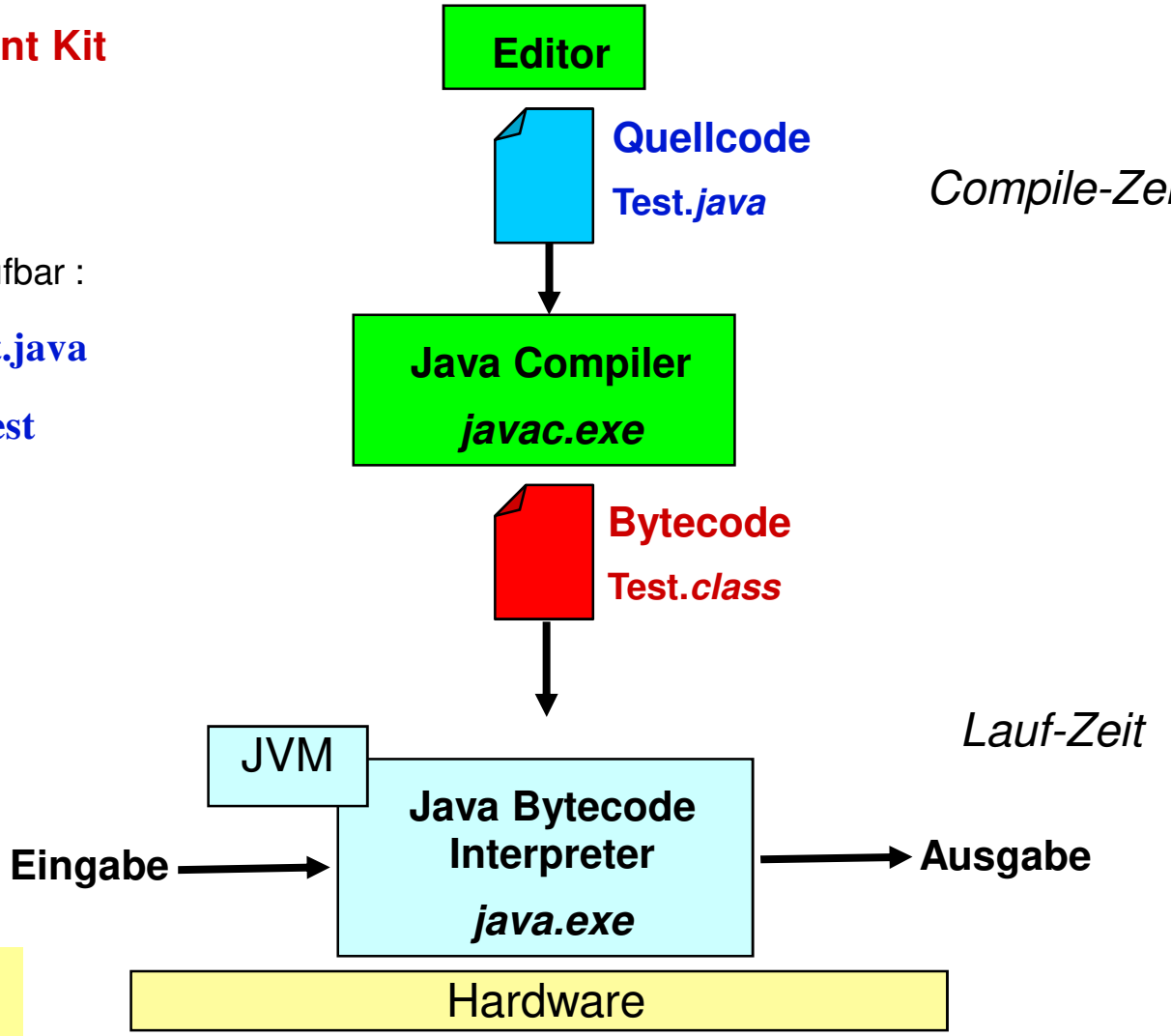
JDK: Java Development Kit

Compiler + JVM

Auf **Kommandozeile** aufrufbar :

C:\ Demos > javac Test.java

C:\ Demos > java Test



Weitegabe kompilierter **class**-Files *ohne* Quellcode genügt !

© H.Neuendorf

Java-Werkzeuge - Java Development Kit **JDK**

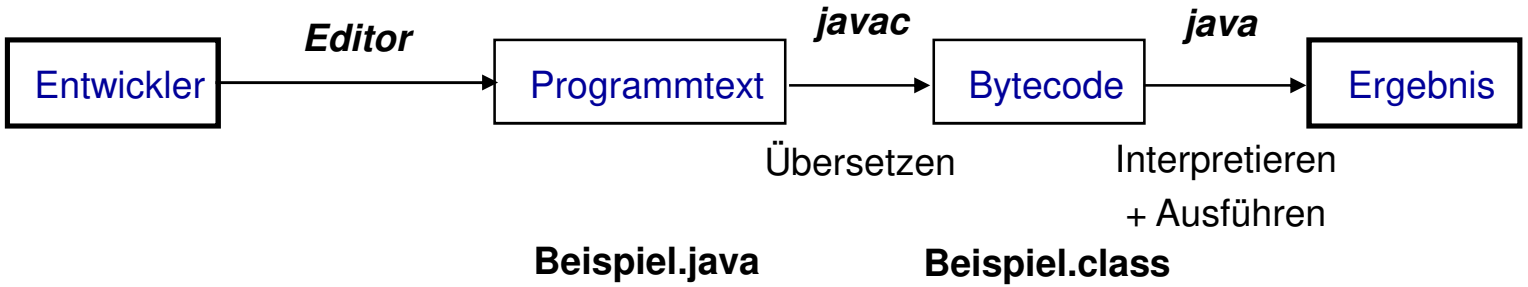
javac	Compiler
java	JVM-Launcher
javadoc	Dokumentations-Tool
jar	Archivierung
javap	Bytecode-Analyse
…	… (*vieles mehr*) …

Finden JDK-Werkzeuge + eigene Klassen mittels Umgebungs-Variablen :

SET **PATH**=C:\Java\ jdk1.8.0_202\bin

SET **CLASSPATH**=.;C:\MyFolder

Kein statischer Linker - da Java-LZ-System den Bytecode benötigter Klassen ***dynamisch*** bei Bedarf ***zur Laufzeit*** (nach)lädt durch **Class Loader**

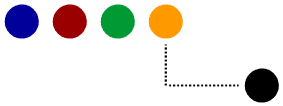


Integrierte Entwicklungsumgebung IDE → **Eclipse :**

Massive Unterstützung bei Erstellung, Verwaltung, Bearbeitung, Ausführung von Entwicklungsprojekten

© H.Neuendorf

Syntaktische Elemente der Sprache Java



- Grundlegende Elemente und Symbolsorten
- Syntax + Semantik
- Namen, Literale, Schlüsselwörter
- Zahlenliterale
- Zeichenliterale
- Bool'sche Literale

© H.Neuendorf

Grundkonzepte imperativer Programmiersprachen

Ebene oberhalb Maschinensprache $\Rightarrow$ *Mittel der Abstrakton und Strukturierung* :

Datentypen : (primitive + strukturierte)

Modellierung verschiedenartiger Größen + Sicherheit durch **Typprüfung**

Variablen + Konstanten :

Zugriff via Namen anstelle Speicheradressen

Ausdrücke :

Verknüpfung von Variablen und Konstanten mittels Operatoren zu Termen

Zuweisungen :

Typprüfung + Speicherung von Ausdruckswerten in Variablen

Ablaufsteuerung : (Verzweigungen & Schleifen)

Kontrolle des Programmablaufs – Steuerung des Kontrollflusses

Unterprogramm : (Prozedur, Funktion, Methode …)

Kapselung von Programmteilen zur Wiederverwendung und Strukturierung

↓

Klassen :

Zusammenfassung von Daten **+** Methoden in eigenständiger **gekapselter** Einheit

Zunehmende Komplexität ↓

© H.Neuendorf

Sorten zulässiger Quellcode-Symbole

Compiler akzeptiert + kennt :

- **Schlüsselwörter**

  class while for private …

- **Namen / Bezeichner (Identifier)**

  customerQueue MyStack …

- **Literale**

  17 012 0xFF true false …

- **Operatoren**

  + / < instanceof …

- **Begrenzer**

  ( ) { } [ ] ; , …

- **Whitespaces**

  <spaces> \t \n // /* … */ …

Sprachdefinition : (Oracle)

Java Language Specification

Fester Wortschatz :

Schlüsselwörter = reservierte Wörter

Neue Bezeichner :

Benutzerdefinierte **Namen** aus zulässigen Zeichen

Verwendung von Zeichen :

Klein-/ Großbuchstaben **NICHT** austauschbar!

Java ist Case Sensitive !!

© H.Neuendorf

Syntax und Semantik

Syntax "*Grammatik*"

Bildungsregeln zulässiger Ausdrücke aus Symbolen + Schlüsselwörtern

Syntaxregeln = **Metasprache** *Backus-Naur-Form* ***BNF**-Syntax*

Ziffer ::=	0|1|2 |3 |4 |5 |6 |7 |8 |9
Zahl::=	Ziffer { Ziffer }
Kleinbuchstabe::=	a|b|c|d|e|f| .....
Großbuchstabe::=	A|B|C|D|E|F| .....
Buchstabe::=	Kleinbuchstabe | Großbuchstabe
Bezeichner::=	Buchstabe { Buchstabe | Ziffer }

The Java® Language Specification
Java SE 10 Edition

James Gosling
Bill Joy
Guy Steele
Gilad Bracha
Alex Buckley
Daniel Smith

Syntaktische Ableitung

("bootstrapping")

Syntaxdiagramm für " Zahl "

engl.: railroad diagrams

Semantik

Bedeutung der einzelnen Sprachkonstrukte = **Wirkung** der Ausdrücke

" Was bedeuten / bewirken die Konstrukte der Sprache im jeweiligen Kontext? "

© H.Neuendorf

Java-Sprachelemente Namen, Schlüsselwörter, Literale

Namen : Bezeichnen Konstanten, Variablen, Methoden, Klassen (Typen) ...

Bestehen **nur** aus Buchstaben, Ziffern, '_' und '$' Nicht ' , # * ....

Erstes Zeichen **muss** Buchstabe, '_' oder '$' sein (keine Leerzeichen !)

Groß- und Kleinbuchstaben unterschieden - **Case Sensitive** ! (min ≠ Min)

Bsp : x x12 birth_date ***Ungültig :*** 1Wert Hallo Welt class A'1

Schlüsselwörter : **Reservierte** Namen zur Kennzeichnung von Sprachbestandteilen

Java-Schlüsselwörter ausschließlich in **Kleinbuchstaben**

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

Kommentare : Vom Compiler ignorierte Erläuterungen im Quellcode

// Zeilenrest wird Kommentar = Zeilenendkommentar

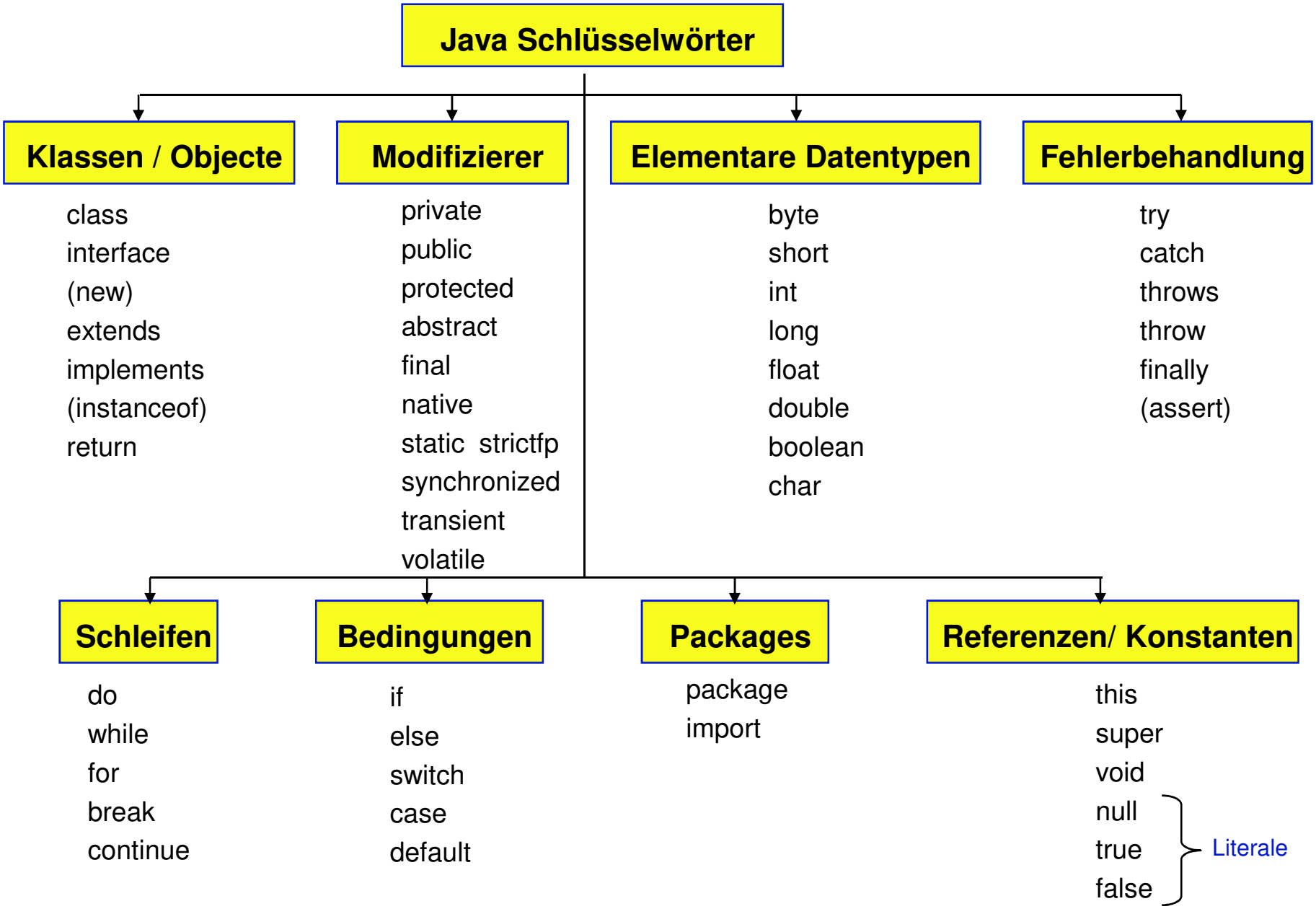
/* **Klammerkommentar**, der sich über beliebig viele Zeilen erstrecken darf; endet mit dem Zeichen: */

// *Klammerkommentare nicht **schachtelbar** !*

Jede Java-Anweisung endet mit einem Semikolon ;

© H.Neuendorf

Java-Schlüsselwörter - Kategorien



© H.Neuendorf

Zahlenliterale

Ganzzahlen :

Dezimalzahlen = Ziffern **0 ... 9** Bsp : 0, 127, 1024

Hexadezimalzahlen = Ziffern **0 ... 9, a, b, c, d, e, f** (für 0 …10,11,...,15)

Kennzeichnung durch Präfix **0x** **Bsp : 0x1A** = 16+10 = 26

Oktal = Ziffern **0 … 7**

Kennzeichnung durch Präfix **0** **Bsp : 012** = 8+2=10

Binär = Ziffern **0** und **1**

Kennzeichnung durch Präfix **0b** oder **0B** **Bsp : 0b101** = 5

Die Ganzzahltypen byte, short, int, long (und char) können ab Java7

als **binäre Literale** ausgedrückt werden

Suffix l oder **L** : 123L ausdrücklich vom Typ long

© H.Neuendorf

Zahlenliterale

Gleitkommazahlen :

Verschiedene Darstellungsweisen Kommateil mit **Punkt** !

Bsp : 3.2 3. 0.3**E**1 3**E**-1 3.2**f** 3.2**F** 3.2**d** 3.2**D**

Bedeutung Suffixe :

e oder **E** 10 hoch Bsp : 2.201**E2** = 220.1 1E1 = 10.0 // stets Fließkomma !

f oder **F** Typ float **d** oder **D** Typ double

Speziell : Underscores _ in numerischen Literalen ("Zahlen")

Java 7

Underscores zwischen(!) Ziffern(!) in numerischen Literalen zulässig

Somit können Zahlen im Code besser strukturiert und lesbarer geschrieben werden

Bsp :

```
double d = 456_789.123_123 ;
long l = 1234_4567_9012_3456L ;
byte b = 0b0010_0101 ;
```

Nicht zulässige Positionen :

- Am Beginn oder Ende der Zahl _12
- Beim Dezimalpunkt 3._12
- Vor den Suffixes f,F,d,D,l,L 3.12_f

© H.Neuendorf

Zeichenliterale Bool'sche Literale

Zeichen : *characters char*

Einzelne in **Hochkommata** eingeschlossene Buchstaben, Ziffern und Sonderzeichen

Bsp: **'x'** **'+'** **'3'** **'z'** **' '** **'\n'**

Zeichenketten : *String*

kein primitiver Typ !

Folge in **doppelter Hochkommata** eingeschlossener Zeichen

Dürfen Zeilengrenzen nicht überschreiten (Java 13 Mehrzeilige Textblocks / Multi-line: """ ... """)

Bsp: **"This is a simple string."** **"§3 a + 2% #$"** **"3"** **" "**

Anmerkung :

Zeichen und Zeichenketten entsprechen in Java unterschiedlichen Typen (s.u.)

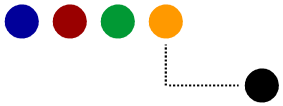
Bool'sche Literale / Wahrheitswerte :

(**kein** Zahlentyp in Java)

true *und* **false**

© H.Neuendorf

Java Typkonzept



- **Prinzip der strengen Typisierung**
- Deklaration + Initialisierung
- Ganzzahltypen
- Fließkommatypen
- Zeichentyp
- Bool'scher Typ
- Cast-Operationen
- Kleine Fallen …

© H.Neuendorf

Java-Typkonzept - Strenge Typisierung

Der Prozessor kennt nur Speicheradressen

Der Prozessor kennt nur bits - und deren Gruppierung zu Byte-Folgen

⇓

Wichtige Abstraktionen moderner Programmiersprachen :

Konzept des Variablen- / Konstantennamens erspart das Operieren mit Speicheradressen - die sich zudem während Programmausführung evtl. ändern.

Konzept des Datentyps macht Programmierung intuitiver & sicherer.

Unterschiedliche Datentypen entstehen …

… indem **Bit-Muster** (Folgen von Nullen und Einsen) **je nach Typ-Angabe** durch den Prozessor **unterschiedlich interpretiert** werden …

… ohne dass Mensch die technischen Details (direkt) beachten müsste.

© H.Neuendorf

Java-Typkonzept - Strenge Typisierung

Datentypen :

- Jede Variable / Konstante **muss** mit **Typangabe deklariert** werden = **Strenge Typisierung**
- Primitive **Standardtypen** + selbst deklarierte strukturierte Typen (= Klassen !)

  Primitiv → keine tiefere Struktur, können nur einen Wert aufnehmen

  Strukturiert → Klassen - können beliebig viele Werte aufnehmen

Seit Java 10 Deklaration mit Schlüsselwort `var` *mittels Type-Interference - hier nicht betrachtet.*

Typ legt fest :

- **Welche Werte** der Variablen zuweisbar sind !
- **Welche Operationen** durchführbar sind !

Kernkonzept moderner typisierter Programmiersprachen → **Compiler prüft** :

Variablen erhalten **zulässige Werte** + nur **zulässige Operationen** werden angewendet

⇓

Statische Typprüfung beim Übersetzen (**Compilezeit**) ***vor*** Programmausführung (**Laufzeit**)

© H.Neuendorf

Java-Typkonzept - Deklaration + Initialisierung

Deklaration :

Jede Variable ***muss*** **vor** Verwendung deklariert = mit ihrem **Typ** angemeldet werden.

Bei Programmausführung wird entsprechender **Speicher** belegt.

Seit Java 10 Deklaration mit Schlüsselwort `var` *mittels Type-Interference*

Aufzählungen möglich - Deklaration endet mit **Strichpunkt ;**

Bsp: **int x ; short w, y, z ;**

x = 12 ;

Andernfalls ***Compilerfehler :***

… may not have been initialized

Initialisierung :

Auf ***Methodenebene*** *keine Defaultwerte - anders als auf Klassenebene und bei Arrays …*

Bei Deklaration ***kann*** bereits **Initialwert** mitgegeben werden.

Spätestens ***vor*** erster Verwendung ***muss*** Variable expliziten Wert erhalten.

Bsp: **int x = 100 ; int y = 0, z = 5 ;**

Speziell Konstanten-Deklaration : final int MAX = 527 ; // Schlüsselwort final

Dürfen ihren **Wert nach einmaliger Initialsierung nicht mehr verändern**.

Einmalige Initialisierung auch nach Deklaration möglich – sogar auch erst zur LZ !

Vorteil : Lesbarkeit + Wartbarkeit durch Pflege konstanter Werte an einer Code-Stelle.

© H.Neuendorf

Java Standard-Datentypen : Ganze Zahlen

- **byte** 8-Bit -2^{7} .....$2^{7}-1$ (-128 ... 127) *(1Bit fürs **Vorzeichen**)*
- **short** 16-Bit -2^{15} .....$2^{15}-1$ (-32768 ... 32767)
- **int** 32-Bit -2^{31} .....$2^{31}-1$ (-2147483648 ... 2147483647)
- **long** 64-Bit -2^{63} .....$2^{63}-1$

8 Bits = 1 Byte

Speicherverbrauch + Wertebereich bedenken !

Gefahr : Bei Programmausführung **können Werte überlaufen** !

***Teilmengenbeziehung** gemäß Wertebereich :* **long ⊃ int ⊃ short ⊃ byte**

⇓

Bsp: short-Wert kann an long- oder int-Variable zugewiesen werden, nicht aber an byte-Variable

Bei Zuweisung an *größeren* Datentyp passiert **implizite automatische Typkonvertierung.**

Bsp: `int i = 17;` **`long lg = i ;`** // ok, unproblematisch, in **lg** steht Wert 17L

© H.Neuendorf

Java Standard-Datentypen : Casten

Explizites Casting mittels :

Castoperator : ***(type) x***

bewirkt / erzwingt eine explizite Typkonvertierung !

Bei Cast zum ***kleineren*** Datentyp werden überstehende Bits "**abgeschnitten**" !

⇒ eventuell (nicht-intuitive) **Wertänderung** !

Bsp: short s = 258 ; **byte b = (byte) s ;**

// in **b** steht **2** da "überstehende" Bits abgeschnitten wurden !

*Beim **Casten zwingt** man den Compiler etwas zu tun, was dieser freiwillig **nicht** tut – **also Vorsicht !***

Cast-Konzept auch bei Referenztypen anwendbar.

Dort von fundamentalerer Bedeutung …

© H.Neuendorf

Ganzzahldarstellung im Rechner

Exakte Darstellung **ganzer** Zahlen als Folge binärer Zeichen **0** oder **1**

Repräsentieren 2er-Potenz : (0 oder 1) * $\mathbf{2^n}$

Ganzzahl mit **8** Bits :

8.Bit	7.Bit	6.Bit	5.Bit	4.Bit	3.Bit	2.Bit	1.Bit
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Maximaler Wert - Alle Bits mit **1** belegt :

11111111 = $1*2^7 + 1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0$ =

= 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = **255**

Aber **1 Bit** zur Kennzeichnung des **Vorzeichens** erforderlich: 8.Bit = **+** oder **-**

Maximaler Wert somit :

$\pm (1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0)$ =

$\pm$ (64 + 32 + 16 + 8 + 4 + 2 + 1) = $\pm$ **127**

Darstellung im Zweierkomplement erlaubt um 1 größere negative Zahl

⇒ **Zahlenbereich** für 8 Bit-Zahlen : **-128 bis +127** (Typ **byte**)

Durch Hinzunahme weiterer Bits können entsprechend größere ganze Zahlen exakt dargestellt werden : ⇒ **short, int, long**

Physikalische Darstellung von **"1"** und **"0"** ist Sache der Hardware :

Spannungswert

Kondensatorladung

…

© H.Neuendorf

Standard-Datentypen : Gleitkommazahlen

- **float** **32-Bit** *größte Zahl :* $\approx 10^{+38}$
- **double** **64-Bit** *größte Zahl :* $\approx 10^{+308}$

$\Rightarrow$ **double** hat **mehr signifikante Dezimalstellen** + **größeren Bereich** als **float**

$\Rightarrow$ **größere** / **kleinere** / **genauere** Gleitkommazahlen darstellbar !

aber : **Genauigkeit begrenzt** $\Rightarrow$ **Rundungsfehler** !

z.B. bei Vergleichsoperationen !!

Gleitkommaberechnungen
sind aufwändiger als
Ganzzahlrechnungen - und :

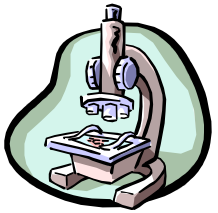
Prinzipiell UNGENAU !!

Teilmengenbeziehung : **double $\supset$ float " $\supset$ " long $\supset$ int $\supset$ short $\supset$ byte**

Bsp:
```
double d = 1.0 ;    float f = 2.0f ;    int i = 3 ;

f = i ;          // ok          i = f ;  // Fehler !

i = (int) f ;    // ok - Typumwandlung (type cast)

                 // Bei Cast :  Nachkommastellen abgeschnitten !
```

© H.Neuendorf

Gleitkommazahlen : float 32-Bit double 64-Bit



Intern Darstellung von Gleitkommazahlen durch Vorzeichen, Mantisse, Exponent :

$$z = (-1)^{v} * \text{Mantisse} * 2^{\text{Exponent}}$$

Darstellung mit verfügbaren Bits gemäß IEEE-Norm:

	v	Exponent	Mantisse
float	1 Bit	8 Bit	23 Bit
double	1 Bit	11 Bit	52 Bit

Aufgrund **exponentieller Zahlendarstellung** können mit **float** (**32 bit**) größere Zahlen als mit **long** (**64 bit**) (trotz weniger Bits) dargestellt werden – allerdings auf Kosten der Genauigkeit !

Zwischen zwei benachbarten darstellbaren **float**-Werten liegen ca. $2^{52-23} = 2^{29}$ = **537 Millionen double**-Werte !

Die meisten Zahlen im Zahlenbereich sind nicht exakt darstellbar – „Lücken" nehmen mit Größe zu !!

Konsequenz für Wertebereich + Darstellung :

float → Exponentialteil von 2^{-127} bis 2^{+127} (entspricht etwa 10^{-38} bis 10^{+38})

Mantissen erlauben Darstellung mit ca. **7** Dezimalstellen Genauigkeit

double → Exponentialteil von 2^{-1023} bis 2^{+1023} (entspricht etwa 10^{-308} bis 10^{+308})

Mantissen erlauben Darstellung mit ca. **16** Dezimalstellen Genauigkeit

In Wirklichkeit komplizierter :

JLSP

© H.Neuendorf

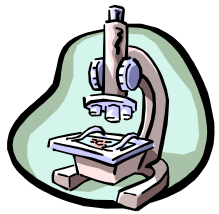
Gleitkommazahlen : Regeln + Besonderheiten

Notation: Durch Suffix **f** oder **d** als **float** oder **double** kennzeichenbar (Default double)

Bsp: 1.0 2.56**f** 4.45**e**-10 2.2**E**5**f** (**E** oder **e** bedeutet **10 hoch**)

Java-Default-Typ für Gleitkommazahlen ist **double !**

```
double d = 3.14 ;     // ok
float f = 3.14 ;      // Fehler !
float f = 3.14f ;     // ok!   float-Wert
```



Typregeln in Ausdrücken beim Typ-"Mischen" :

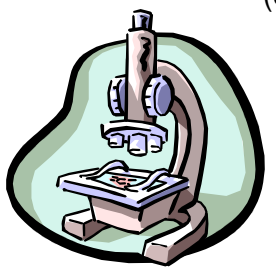
*Der kleinere Operandentyp wird vor Operations-Ausführung in den größeren **konvertiert**.*

*Der berechnete Ausdruck bekommt diesen Typ - **zumindest** **aber** den Typ **int** !*

```
double d = 1.0 ;      float f = 1.0f ;
int i = 2 ;           short s = 3 ;
… f + i …        //  float
… d * i …        //  double
… s + s …        //  int !!
```

© H.Neuendorf

Standard-Zahlentypen : *Kleine Fallen*



Ganzzahlen :

1. Java **schneidet** bei Ganzzahlrechnungen **ab**, um Ganzzahl zu erhalten

   $\Rightarrow$ Kommastellen ohne Rundung **abgeschnitten** !

   Bsp: `double x = 1 / 2 ;` // in x **0.0** ! `double y = - 3 / 2 ;` // in y **– 1.0** !

2. Ganzzahlenbereiche *quasi* zu **Ring** geschlossen :

   Bei Überschreiten positiven Bereichs $\rightarrow$ Wiedereintritt in negativen Bereich

   Bsp: `byte b = 127 ; b = (byte) (b + 1) ;` // in b nun **-128** !

   `byte b = 120; for( int i = 0; i<20; i++ ) { b++; }` // **Überlauf** – keine Exception !!

3. Bei **Rechnen mit byte- und short-Variablen** wandelt **Java mindestens zum Typ int !**

   Bsp: `byte a = 5 ; byte b = 10 ;`

   `byte c = a + b ;` // Compilerfehler ! Typ von (a + b) ist **int** !

   // $\Rightarrow$ Stattdessen **casten** :

   `byte c = (byte) (a+b) ;` // **ok**!

© H.Neuendorf

Fließkommazahlen :

1. **Default double !** ⇒ 1 + 1.0 ist double, 1 + 1.0**f** ist float

   Bsp: float **f** = 1 + 1.0 ; // Fehler !

2. **Rundung von Ganzzahlen** auch bei Zuweisung zu Fließkommatypen !

   ⇒ double d = 1 / 2 ; // in d: **0.0** ! **1** und **2** als **Ganzzahlen**

   double d = 1.0 / 2 ; // in d: 0.5 **1.0** als **double-Zahl**

   double d = (double) 1 / 2 ; // in d: 0.5 **1** zu double **gecastet**

   double d = 0.8 + 1 / 2 ; // in d: **0.8** ! **1/2** liefert 0

3. **Fließkomma-Berechnungen sind ungenau** !

   double d = 0.1 + 0.2 ; // in d steht 0.300000000000004

Somit Fließkommatypen im Grunde ungeeignet für ***monetäre*** Berechnungen !!

4. **Casting** : *explizite Typkonvertierung*

   Bei Cast zum *kleineren* Datentyp wird **abgeschnitten** ⇒ **Wertänderung** !

   Bsp: float f = 17.35f ; int i = **(int)** f ;

   // in i steht 17 - Verlust Nachkommastellen !

   Bsp: short s = 258 ; byte b = **(byte)** s ;

   // in b steht 2 - "überstehende" Bits abgeschnitten !

© H.Neuendorf

Standard-Datentypen : Zeichen - `char`

Zeichen = Typ **char** : **UC-Codierung mit 16 bit = 2 Byte Unicode**

Einzelnes Zeichen (≠ Zeichenkette) in einfachen Hochkommata **' '** **char c1 = 'a' ;**

Binäre Codierung :

1 Zeichen = 1 Byte ⇒ 256 Zeichen

1 Zeichen = **2** Byte ⇒ **65536** Zeichen ! ***Unicode*** ← **Java**

Erste Zeichen von UC entsprechen den ASCII-Zeichen

char gemäß Codierung **auch** "ganzzahlenartig" aber **unsigned** ≠ short-Bereich :

double ⊃ float ⊃ long ⊃ int "⊃ char ⊃" (short) ⊃ byte

⇒ Berechnungen :

```
int n = 'c' - 'a';   // n-Wert 2.  Zwei Plätze weiter in UC-Tabelle
int k = 'a';         // k-Wert 97
short s = ch1;   // Fehler – char sprengt positiven Bereich von short – mit int ok !
char ch2 = (char) k;   // Bei Zahlentyp-Variablen nur mit Cast (wegen Vorzeichen)
```

char-Variablen *können* Zahlenwerte **[0 ; 65535]** direkt zugewiesen werden – aber keine Werte < 0.

Zuweisung von Zahlentyp-Variablen nur mittels Cast - Bedeutung : Zeichen am Platz in UC-Tabelle

© H.Neuendorf

Standard-Datentypen : char

Vergleichsoperatoren sind anwendbar ==, !=, <, >, <=, >=

```
char ch = 't' ;   if ( 'a' <= ch && ch <= 'z' || 'A' <= ch && ch <= 'Z' ) { /* … */ }
```

Unicode-Zeichen auch durch hex-Nummerncode ausdrückbar : '**\u**nnnn'

nnnn = vierstelliger Hexadezimalcode des UC-Zeichens - z.B. char ch = '\u0040'; // für @

Nutzbar zur Setzung in Strings :

```
String s = "Hallo\u0040DH" ;
```

Spezielle **Steuerzeichen** durch *Escape-Sequenzen* ansprechbar - geschützt mit \

Escape-Sequenzen	
\n	new line
\r	return
\t	Tabulatorsprung
\u	UC-Ansprache
\\	Backslash …

UC-Bereiche	
\u0000 - \u007f	ASCII-Z
\u0080 - \u024f	Sonderzeichen
\u0370 - \u03ff	Griechisch
\u0400 - \u04ff	Kyrillisch
\u0600 - \u06ff	Arabisch …

© H.Neuendorf

Standard-Datentypen : `char` versus `String`

Zeichenkette = Typ **String** : (aus java.lang.String)

Kein primitiver Standard-Datentyp !

→ In **" "** : **String s1 = "Hallo" ; String z = "37.6" ;**

→ Mit Operator **+** Zeichenketten-Konkatenation :

```
s1 = s1 + " Welt !" ;
```

→ String-Konvertierung wenn durch **+** String mit **anderem** Datentyp verknüpft :

```
String s2  =  "Alter = " +  40.5 ;        //  Inhalt von s2 :   Alter = 40.5
// Vorsicht Reihenfolge :
String s3  =  10 + 10 + " Alter" ;        //  Inhalt von s3 :   20 Alter
```

*Dass Strings **keine** primitiven Typen sind, sondern **Objekte***

(Referenztypen) macht sich später noch bemerkbar …

© H.Neuendorf

Standard-Datentypen : `boolean`

Nur Werte **true** und **false**

Diese Werte sind **Literale** der Sprache Java – und **nicht** mit 0 oder ≠0 identisch !

(andere Verhältnisse als in C/C++ Welt)

- boolean ist **nicht** zu anderen Datentypen **cast-bar** !
- Andere Datentypen sind **nicht** zu boolean **cast-bar** !

Ergebnis der Anwendung von **Vergleichsoperatoren** (s.u.) :

```
int x = 10 ;   int y = 20 ;      boolean q =  x < y ;
```

Als Operanden und Ergebnis **logischer Operatoren** (s.u.) :

```
boolean a = true ;    boolean b = false ;      boolean c = a && b ;
```

© H.Neuendorf

Theoretische Darstellung von Typen durch Algebra

Datentypen werden durch **Wertemenge** und **Algebra** definiert / konstruiert :

Zulässige **Operationen** und deren **Ergebnistyp :**

Beispiel : **boolean**

Datentyp : boolean

Werte: true, false

Operationen :

and, or, xor : boolean x boolean $\rightarrow$ boolean

not : boolean $\rightarrow$ boolean

Gleichungen : für alle x,y $\in$ boolean gilt :

not true = false not false = true

true or x = true true and x = x

false or x = x false and x = false

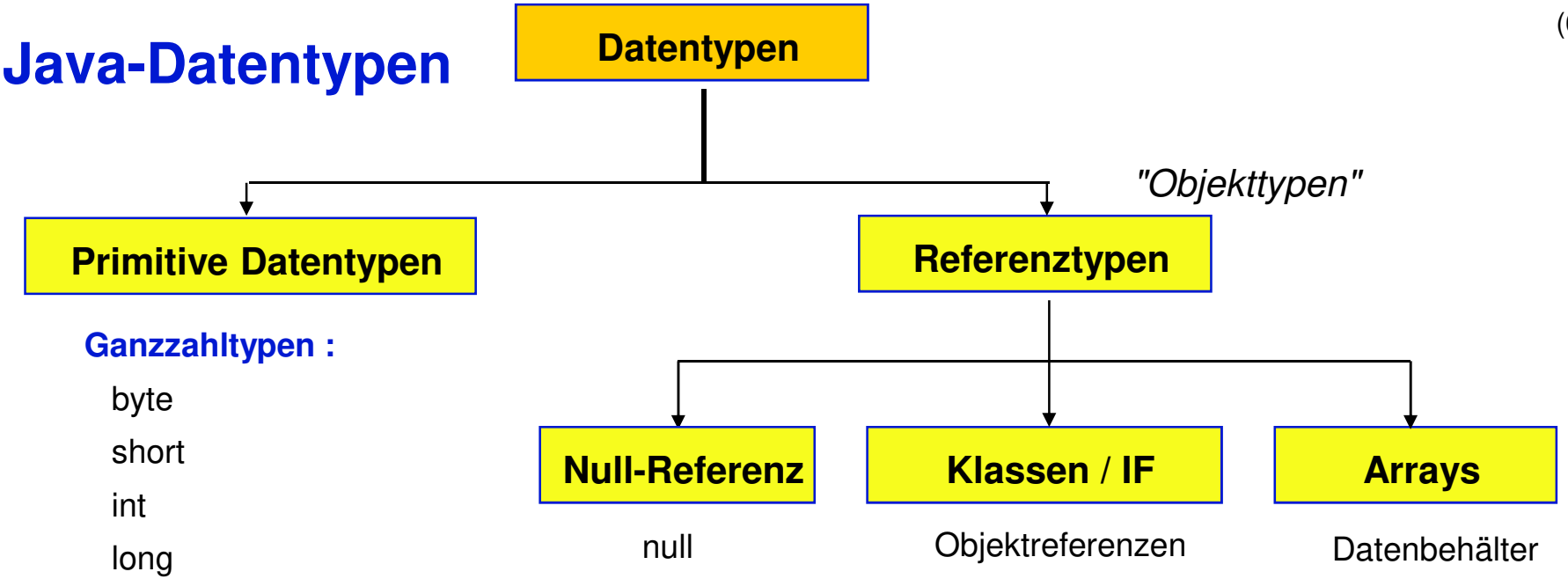
x xor y = (x and not y) or (not x and y) ...

Analog auch für alle anderen Typen definierbar.

Wird im Rahmen der Objektorientierung auf strukturierte Typen erweitert.

© H.Neuendorf

Java-Datentypen



Ganzzahltypen :

- byte
- short
- int
- long

Fließkommatypen :

- float
- double

boolsche Werte :

- boolean

Zeichen :

- char

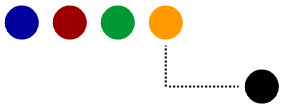
) **Defaults erst wirksam bei Initialisierung von Objekt-Attributen und Array-Inhalten …*

Primitive Datentypen

Typ	**Länge**	**Wertbereich**	**Initialisierung / Default** *)
byte	8 Bit	-128 … 127	(byte) 0
short	16 Bit	-32768 … 32767	(short) 0
int	32 Bit	-2147483648 … 2147483647	0
long	64 Bit	≈ ± 2^63	0L
float	32 Bit	≈ ± 3E+38	0.0f
double	64 Bit	≈ ± 1E+308	0.0d
boolean		**true / false**	false
char	16 Bit	Unicode !	'\u0000' = (char) 0

© H.Neuendorf

Java Zuweisungen und Operatoren



- Zuweisungsoperationen und strenge Typisierung
- Ausdrücke und Standardoperatoren :
  - Arithmetik / Inkrement + Dekrement-Operatoren
  - Vergleichsoperatoren
  - Logische Operatoren + Kurzschlussauswertung
  - Bitoperatoren
  - Zuweisungsoperatoren
- Operatorenpriorität

© H.Neuendorf

Java - Zuweisungen

Zuweisungen : **x = y + 1 ;** **x = x + 1 ;** // **nicht** mathematische **Identität** !

- **Rechter** Ausdruck berechnet und **links** stehender Variablen zugewiesen.
- **Nur** erlaubt, wenn **Typen** beider Seiten **zuweisungskompatibel** !
- Sonst Compiler Fehler gemäß **statischer Typprüfung** →

  Verhindert LZ-Abbrüche oder falsche Werte.

Regeln für gültige Zuweisungen

- beide Seiten **desselben** Typs *oder*
- Typ **linker** Seite schließt Typ **rechter** Seite gemäß **Typ-Teilmengenbeziehung** ein.

```
int i = 2, j = 5;    short s = 5;    byte b = 4 ;

i = j ;  // ok       i = s ;  //  ok        s = i ;  // Fehler!

i = 300 ;  // ok                           b = 300 ;  // Fehler!
```

© H.Neuendorf

Ausdrücke (Terme) - Definition

Ein Ausdruck ist eine Folge von Operatoren und Operanden , welche …

- einen Wert berechnen und/oder
- ein Objekt bezeichnen und/oder
- den Typ eines Objekts festlegen oder ändern und/oder
- Seiteneffekte erzeugen.

Im **einfachsten Fall** : eine **Konstante, Variable oder ein Methodenaufruf**.

Durch Operatoren werden Operanden zu **komplexeren Ausdrücken** verknüpft - z.B.

x + 12.9 Operanden : x und 12.9 Operator : +

In Java hat jeder Ausdruck einen resultierenden **Typ**.

Man kann nicht Äpfel & Birnen verrechnen …

Grundsätzlich:

Operanden und Operatoren müssen zueinander passen :

- **Nicht alle Operatoren sind auf alle Operanden anwendbar !**
- **Nicht alle Operanden lassen sich miteinander zu Ausdrücken kombinieren !**

© H.Neuendorf

Java Operatoren - Arithmetik

Liefern **Ergebnis** vom **entsprechenden Zahlentyp**

Arithmetische Ausdrücke : **z = - (3 * x + y) ;**

Berechnung eines **numerischen** Werts aus Variablen und Konstanten mittels Operatoren.

Unäre Operatoren :

Vorzeichenfestlegung durch **+** und **-** → **"binden"** am stärksten …

Bsp: 4 + - 3 // liefert 1 - 4 + 3 // liefert -1

Binäre Operatoren :

Übliche Vorrangregeln ⇒ Punkt vor Strich, durch Klammern änderbar !

Addition **+** Subtraktion **-** Multiplikation *****

Division / : Ergebnis bei Ganzzahlen eine Ganzzahl (ohne Rest) **4 / 3 → 1**

Ergebnis bei Gleitkommazahlen eine Gleitkommazahl

Modulo **%** : bei Ganzzahlen der ganzzahlige Divisionsrest **4 % 3 → 1** **-7 % 2 → -1**

bei Gleitkommazahlen: *(selten verwendet …)*

Gleitkomma-Divisionsrest von x/y

8.5 % 2.1 = 8.5 - 4 * 2.1 = 0.1 Divisionsrest

© H.Neuendorf

Operatoren - Arithmetik

1. Bei **Ganzzahltypen** ist **/0** und **%0** nicht zulässig - liefert LZ-**ArithmeticException :**

```
int a = 5 / 0 ;     // Exceptions !!
int b = 6 % 0 ;
```

2. **Fließkommatypen können dabei jedoch gegen Unendlich "überlaufen" !**

Operationen können mathematisch "uneigentliche" Werte **ohne** Exception liefern :

NaN (*Not a Number)* **Infinity** **-Infinity** // nicht direkt zuweisbar - nur mittels Hüllklassen

```
double d  =  0.0 / 0.0 ;      //  in d steht NaN !
float f   =  1.0f / 0f ;      //  in f steht Infinty !
double g  =  5. % 0 ;         //  in g steht NaN !
```

Weiterverarbeitung "pathologischer" Werte mittels **Hüllklassen** (s.u.)

© H.Neuendorf

Operatoren - Inkrement- und Dekrement-Operatoren

Erhöhung oder Erniedrigung des Werts von **x** um **1**

x++ ; **++x ;** **x- - ;** **- -x ;** // entspricht **x = x+1;** bzw **x = x-1;**

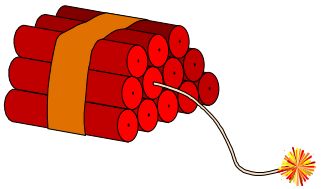
Nur auf **Variablen** anwendbar, **nicht** auf **Ausdrücke**: x = (x * y)++ ; // **Fehler!**

Vorsicht bei Verwendung **in** Ausdrücken !

Stellung Operator bestimmt Auswertreihenfolge ! :

```
int x = 1;
a)  int y = ++x * 3 ;      // liefert 6 für y
b)  int y = x++ * 3 ;      // liefert 3 für y
```

x trägt am Ende in beiden Fällen Wert 2



Präfixform **++x** : Erst **x** manipulieren, dann weiterrechen

Postfixform **x--** : Erst Zuweisung durchführen, dann **x** manipulieren

© H.Neuendorf

Operatoren - Vergleichsoperatoren

Liefern **Ergebnis** vom **Typ** **boolean**

==	gleich	x == 3
!=	ungleich	x != y
>	größer	x > y
<	kleiner	x < y
>=	größer oder gleich	x >= y
<=	kleiner oder gleich	x <= y

nicht : x = 3

Zuweisung - kein Vergleich !

Vergleiche mit **NaN -Werten** liefern "immer" false **NaN** is *unordered*

(0.0/0.0 == 0.0/0.0 liefert *false* jedoch 5 != 0.0/0.0 liefert *true*)

Vergleiche eigentlicher Werte gegen **Infinty** und **–Infinty** funktionieren :

```
double d1=1./0 , d2 = -1./0 ;        boolean b = d1 > d2 ;    // true
```

© H.Neuendorf

Operatoren - Logische Operatoren

Liefern **Ergebnis** vom **Typ boolean**

Wirkung auf Boolesche Operanden boolean a, b

Binär

And a **&&** b *(bzw **&**)* **Or** a || b *(bzw |)* **XOR** a **^** b *(entweder oder)*

Unär

Not **!** a

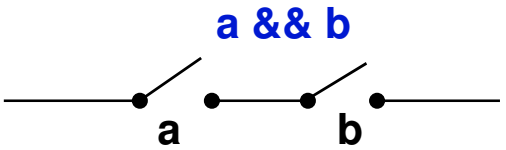
Bsp.: `int x = … ;` `boolean b = ( 0 <= x && x <= 10 || 100 <= x && x <= 110 ) ;`

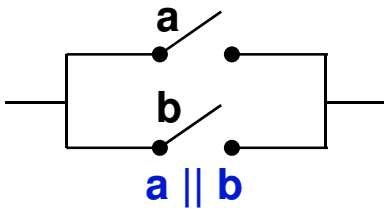
Vorrangregel : *Im Zweifel klammern !*

== bindet stärker als ! bindet stärker als && bindet stärker als ||

a **&&** b **==** c **&&** d *hat Bedeutung von* a **&& (**b **==** c **) &&** d

a	b	a && b	a || b	a^b
w	w	w	w	f
f	w	f	w	w
w	f	f	w	w
f	f	f	f	f





© H.Neuendorf

Logische Operatoren - Kurzschlussauswertung

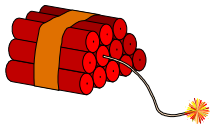
***lazy** evaluation*

Optimierungsleistung der JVM :

Auswertung Vergleichsausdruck **abgebrochen,** wenn **Gesamtergebnis** logisch **feststeht …**

d.h. unabhängig ist von Auswertung der noch **nicht** ausgewerteten Teile

&&-Verknüpfung :	Wenn schon **erster** Teil **false** , dann auch Wert des **Gesamt**ausdrucks **false**
|| - Verknüpfung :	Wenn schon **erster** Teil **true** , dann auch Wert des **Gesamt**ausdrucks **true**



```
int x = … ;    int y = … ;
boolean b1 = ( y == 10  &&  ++x * y > 10 ) ;   // Wenn y != 10, dann ++x * y nicht berechnet !
boolean b2 = ( x < 0  ||  --x * y  < 10  ) ;   // Wenn x < 0, dann --x * y nicht berechnet !
```

Umgehung durch **strict-**Operatoren :

& (statt &&) bzw. **|** (statt ||)

Bei Operanden vom Typ **boolean** wirken **&** und | als **logische** Operatoren, **nicht** als Bit-Operatoren (s.u.)

```
// Positiv-Beispiel :
// lazy verhindert LZ-Fehler
if( x!=0  &&  1/x > 0 ) …
```

© H.Neuendorf

Operatoren - Bitoperatoren

Bitoperationen (~, &, |, ^, <<, >>, >>>) **nur für Ganzzahltypen – und char**

Manipulation **Bitmuster** : Direkte Bit-Verarbeitung in Binärdarstellung

⇒ **Zurückliefern geändertes Bitmuster = Ganzzahltyp**

Negation :

a	~a
0	1
1	0

Und, Oder, Exclusiv-Oder :

a	b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Fiktives Bsp. :

x = 1010

y = 1100

x & y = 1000

x | y = 1110

x ^ y = 0110

~ x = 0101

~ y = 0011

Shift-Operatoren :

a << n Schiebt Bits von **a** um **n** Stellen nach **links** und füllt mit **0-Bits** auf

Bsp: a = 0110 ⇒ (a << 1) liefert 1100

a >> n Schiebt Bits von **a** um **n** Stellen nach **rechts** und füllt mit **höchstem Bit** von **a** auf

Bsp: a = 1100 ⇒ (a >> 2) liefert 1111

a >>> n Schiebt Bits von **a** um **n** Stellen nach rechts und füllt **0-Bit** auf

Bsp: a = 1100 ⇒ (a >>> 2) liefert 0011

© H.Neuendorf

Bitoperatoren

Negation : ~

byte b = **3** ; // ≡ 0000 0011

byte c = **(byte)**~b ; // ≡ 1111 1100 ≡ - 4

// Wert von c ist - 4 !

Addition : &

byte b1 = **3** ; // ≡ 0000 0011

byte b2 = **1** ; // ≡ 0000 0001

byte c = (byte)(b1 **&** b2) ; // ≡ 0000 0001 ≡ +1 ;

// Wert von c ist +1 !

Shift : Linksshift << und Rechtsshift >>

byte b = **2** ; // ≡ 0000 0010

byte c = (byte)(b **<< 1**) ; // ≡ 0000 0100 ≡ +4

// Wert von c ist +4 !

Schieben von Bits nach links bzw rechts entspricht Multiplikation bzw Division durch **2**

Zahldarstellung mit fiktiven **4** bits im **Zweierkomplement**

Höchstes Bit → Vorzeichen !

1 = - 2^3 **0** = + 0

Dezimal	**Bitmuster**
+8	nicht darstellbar
+7	0111
+6	0110
+5	0101
+4	0100
+3	0011
+2	0010
+1	0001
0	**0000**
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

Vorteil :
eindeutige Darstellung der **Null 0**

© H.Neuendorf

Operatoren - Zuweisungsoperatoren

Verschiedene ***Rückgabetypen*** *- je nach Operator*

Abkürzende Schreibweise für Operation **und** Zuweisung

Für **x** vom **Typ T** ist **x op= y** kürzere Schreibweise für :

x = (T) (x op y) ;

```
short x = 1,  y = 2 ;      x += y ;     //  x = (short)(x + y ) ;
x /= y ;     x -= y ;     x %= y ;     x *= y ;
```

```
// Eingebauter Cast:
int x = 2 ;
double d = 3.5 ;
x += d ;   //  in x : 5
```

Auch für andere arithmetische Operatoren - und *einige* (nicht alle) logische und Bit-Operatoren

Generell :

Bloße Schreibersparnis - generell kein schnellerer Bytecode

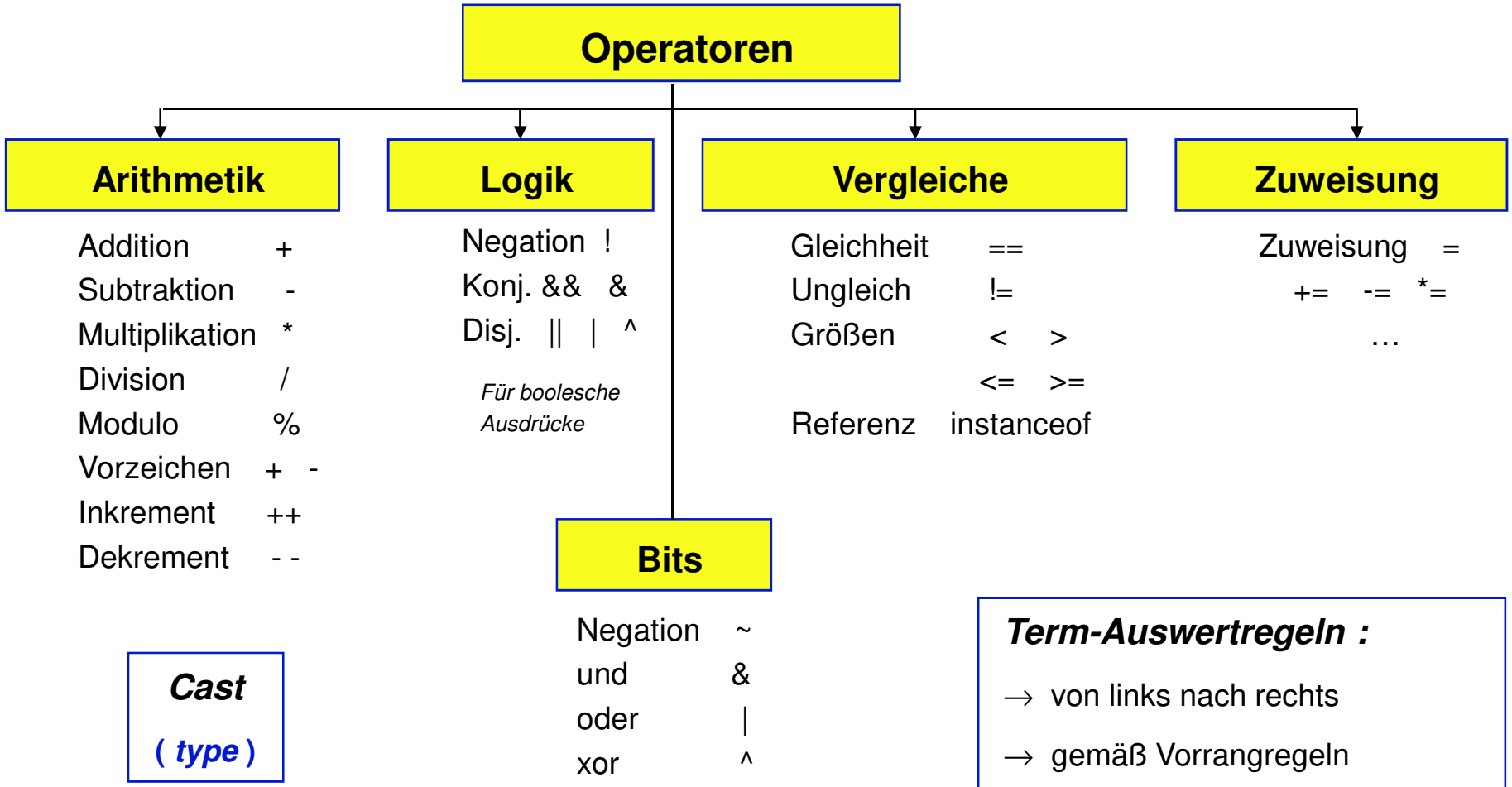
Nur in besonderen Fällen kann sich schnellere Auswertung ergeben

Zuweisungskette =

```
x = y = 2*z+1 ;        auch:    x = 3*( y = 4*z -1) ;
```

© H.Neuendorf

Java-Operatoren



© H.Neuendorf

Priorität der Operatoren - Vorrangregeln

Zusammengesetzter Ausdruck : **boolean b = 4 + 3 < 7 * 2 - 3 ;** *// liefert true*

Auswert-Reihenfolge der Operationen durch **Priorität** der Operatoren geregelt

Im Zweifel / zur Übersicht Klammern setzen !

Bezeichnung	Operator
Unäre Operatoren	++ -- + - ~ !
Expliziter Cast	(<type>)
Multiplikative Operatoren	* / %
Additive Operatoren	+ -
Schiebeoperatoren	<< >> >>>
Vergleichsoperatoren	< > <= >=
Vergleichsoperatoren	== !=
bitweise Und	&
bitweise XOR	^
bitweise Oder	|
logisches Und	&& &
logisches Oder	|| |
Zuweisungsoperator	= += -= …

hoch

niedrig

Operator höherer Priorität **vor** Operator niedrigerer Priorität angewandt !

Bsp : Punkt- vor Strichrechnung

Setzen von **Klammern** ändert Auswert-Reihenfolge !

boolean b = 4+3 < 7 * (2 - 3) ;

// liefert false !

© H.Neuendorf

Überblick Primitive Datentypen + Operatoren in Java

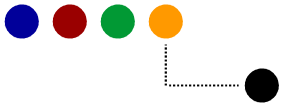
	+ - * / % ++ --	=	== !=	< > <= >=	(*type*)	&& (&) || (|) ^ ! ?	& | ^ ~ << >> >>>
byte short int long	●	●	●	●	●		●
float double	●	●	●	●	●		
char	●	●	●	●	●		●
boolean		●	●			●	

Weitere Operatoren im OO-Kontext mit Wirkung auf Referenztypen :

`instanceof` **`new`** **`•`** **`[ ]`** **`->`**

© H.Neuendorf

Grundstruktur prozeduraler Java-Programme



- Ausführbare Klasse mit **main()** - Methode
- Handling Ein-/Ausgabe-Operationen mit Tool **IO.java**

© H.Neuendorf

Grundstruktur Java-Programm = Blockstruktur

```
// Minimales ausführbares Programm :
class Name {
    public static void main( String[ ] args ) {
    // … Deklarationen + Sequenz von Anweisungen …
    }
}
```

*Zu jeder **öffnenden** Klammer (**Blockanfang**) gehört auch eine **schließende** Klammer (**Blockende**) **!***

Schlüsselwort ***class*** **:** Programm stellt selbst bereits **Klasse** dar

Programm-Start mit Ausführung der *zentralen Methode* ***main()***

String[ ] args **:** String-Array, das bei Programm-Aufruf gefüllt werden *kann*

void **:** main() liefert **keinen** Wert zurück

static **:** statische Methode - **direkt aufrufbar**, ohne erst Objekt zu erzeugen

public : Methode main() nach außen **sichtbar**, aufrufbar

Paar von geschweiften Klammern **{ }** mit null oder mehr Anweisungen = **Block**

Blöcke können **geschachtelt** werden

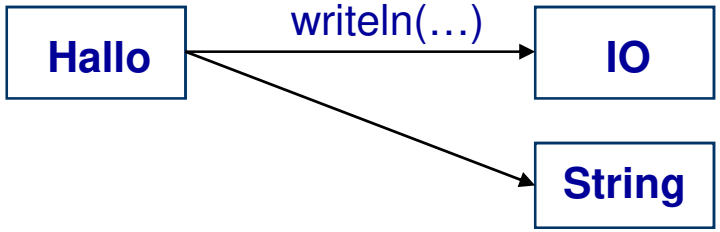
Ein **Block** kann u.a. überall da stehen, wo eine einzelne **Anweisung** verwendet werden kann - da **ein Block eine (zusammengesetzte) Anweisung ist.**

© H.Neuendorf

Grundstruktur Java-Programm


```
// Erstes Programm

class Hallo {

    public static void main( String[ ] args )   {

        String gruesse = "Hello World" ;

        IO.writeln( gruesse ) ;     // Methodenaufruf

    }

}
```

Quelldatei Hallo.java



Kollaborationsdiagramm (UML)

In Java-Programmen **kollaborieren** Klassen bzw. Objekte.

Klassen / Objekte erhalten Aufträge :

Klasse **Hallo** ruft Methode **writeln()** der Klasse **IO** auf

Hallo = Auftraggeber **Client**

IO = Auftragnehmer **Server**

Klasse **IO** stellt Methoden zur Vfg. **(s.u.)**

Methoden können **Parameter** haben oder parameterlos arbeiten.

Methoden können **Werte zurückliefern**.

Durch **Semikolon** werden Anweisungen abgeschlossen **;**

© H.Neuendorf

Ein- / Ausgabe (I/O)

Wertbelegungen von Variablen / Konstanten durch :

- Initialisierung und direkte Wertzuweisung
- Einlesen von **Tastatur**, **Datei, Netz**

Ein-/ Ausgabe **kein** integraler Teil der Sprache Java, sondern dargestellt in Java-Paketen :

→ zahlreiche (Zeichen-/Byte-Strom-) Klassen mit speziellen Methoden zur Ein- / Ausgabe

Handhabung nicht trivial ⇒ Vorerst Arbeiten mit Ein-/Ausgabeklasse **IO.java**

Deren Methoden verbergen / kapseln die Komplexität ...

```
class Hallo {
    public static void main( String[] args ) {
        IO.writeln( "Hallo Mosbach!" ) ;
        IO.advance(3) ;
        IO.writeln( ) ;
        IO.writeln( 123456 ) ;
    }
}
```

Tool-Klasse IO :

Methoden sind **public** + **static**

⇓

"Direkt" (= ohne Instanziierung) verwendbar !

© H.Neuendorf

Konsolen-Ein/Ausgabe mittels IO.java

Ausgaben :

```
void advance( int n )        // gibt n Leerzeilen aus
void write( String s )       // gibt String s aus
void write( int m )          // gibt Zahl m aus                    ... u.a.
void writeln( String s )     // gibt String s  aus  +  Zeilenvorschub
void writeln( int m )        // gibt Zahl m aus  +  Zeilenvorschub
void writeln( )              // Zeilenvorschub
```

Eingaben mit vorangestellter Eingabeaufforderung (prompt) :

```
String promptAndReadString( String s )        // gibt String s aus und liest String ein
int promptAndReadInt( String s )              // gibt String s aus und liest Integer ein
char promptAndReadChar( String s )            // gibt String s aus und liest Buchstaben ein
double promptAndReadDouble( String s )        // gibt String s aus und liest double-Zahl ein
// … und entsprechend für die restlichen primitiven Typen …
```

Runden von Kommazahlen auf n Kommastellen :

```
float round( float f, int n )      double round( double d, int n )      //  liefert gerundete float- bzw double-Zahl
```

© H.Neuendorf

Sequenzielles Java-Programm

Schon mit **reinen Sequenzen** lassen sich Aufgaben lösen.

Wechselgeld für Geldbetrag **zwischen 0 und 100 Cent** - wenn Münzen für 1,2,5,10,20,50 Cent und 1 Euro zur Verfügung stehen.

Ziel ist es mit möglichst wenig Münzen auszukommen ...

```
class Wechsler {
    public static void main( String[ ] args )  {
        int betrag = IO.promptAndReadInt( "Anfangsbetrag = " ) ;      // in Cent
        int rest = betrag ;
        IO.writeln( "Anzahl Euros =    " + rest / 100 );     rest = rest % 100 ;
        IO.writeln( "Anzahl 50 Cent = " + rest / 50 );       rest = rest % 50 ;
        IO.writeln( "Anzahl 20 Cent = " + rest / 20 );       rest = rest % 20 ;
        IO.writeln( "Anzahl 10 Cent = " + rest / 10 );       rest = rest % 10 ;
        IO.writeln( "Anzahl   5 Cent = " + rest / 5 );       rest = rest % 5 ;
        IO.writeln( "Anzahl   2 Cent = " + rest / 2 );       rest = rest % 2 ;
        IO.writeln( "Anzahl   1 Cent = " + rest ) ;
   }
}
```

*… **Sequenzen** reichen jedoch noch **nicht** aus, um eine Programmiersprache **vollständig** zu machen.*

SQL z.B. ist keine vollständige Sprache

© H.Neuendorf

Konventionen in Java-Programmen

*Making your code **readable** is as important as making it **executable**.*

R.C. Martin : Clean Code

Keine Vorschrift, jedoch üblich !

Einheitlichkeit + Konsistenz erhöht deutlich **Lesbarkeit** des Quellcodes

1. Nicht mehr als **einen** Befehl **pro Zeile** schreiben.
2. Öffnende geschweifte Blockklammer **"{"** steht am Ende des vorangegangenen Statements - oder in einer eigenen Zeile.
3. Wenn neuer Block mit **"{"** begonnen wird werden alle in diesem Block stehenden Zeilen um einige Stellen **nach rechts eingerückt.**
4. Das Blockendzeichen **"}"** wird stets so eingerückt, dass es mit der Einrückung der Zeile übereinstimmt, in der der Block geöffnet wurde.

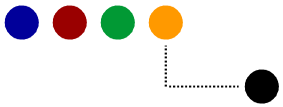
Tipps :

1. Zugehörige schließende und öffnende Klammern **{...} (...) [...]** immer gleich schreiben und ihren Inhalt erst nachträglich einfügen - damit schließende Klammern nicht vergessen werden.
2. Umgeben Sie Operatoren mit Leerzeichen - damit Formeln besser lesbar werden.

*Oft existieren firmeninterne **Style-Guides*** z.B. https://**google**.github.io/styleguide/javaguide.html

© H.Neuendorf

Java-Kontrollstrukturen



- **Verzweigungen**

  if else / switch case

- **Schleifen**

  while / do while / for

- **Sprunganweisungen**

  break / continue (return / …)

*Sequenzen von Anweisungen, Alternativverzweigung und while-Schleife machen Java bereits zur **universellen**, **vollständigen** Sprache …*

… in der alle Algorithmen formulierbar sind.

© H.Neuendorf

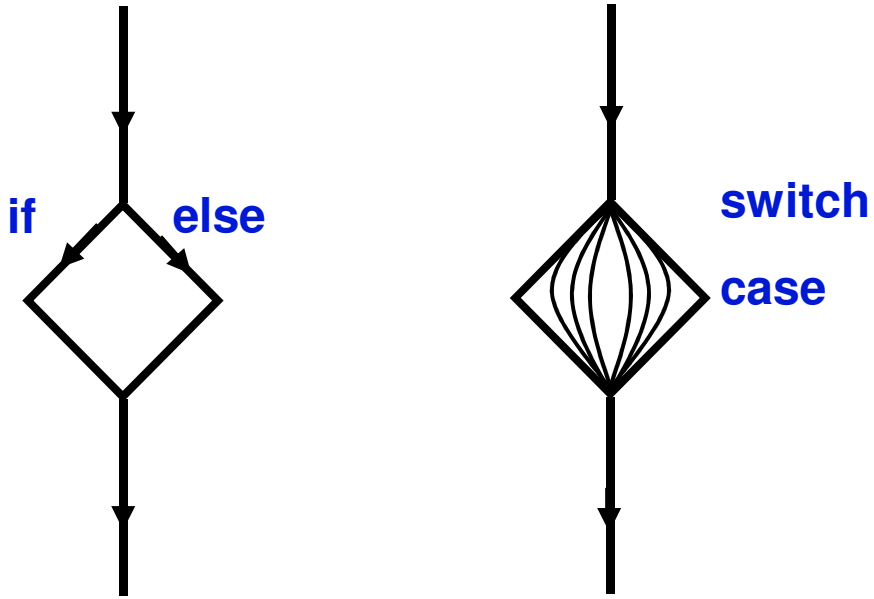
Kontrollstrukturen - Arten und Aufbau

Sinn : Kontrolle des Programmablaufs

Steuerflusskontrolle

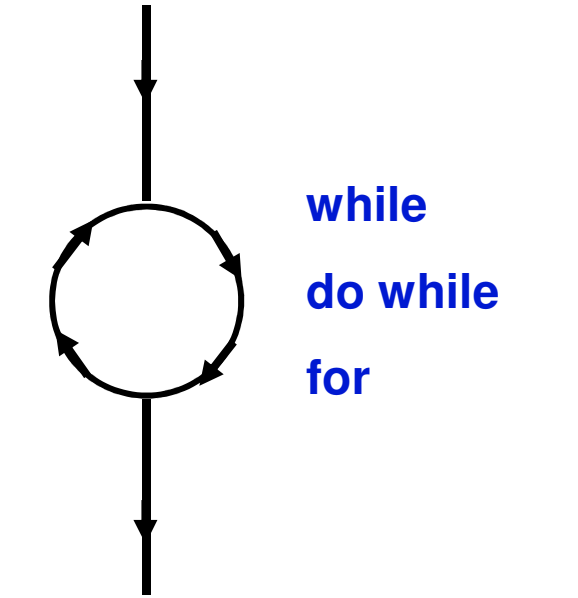
Genereller **Aufbau** :

Strukturkopf { Strukturrumpf }



Verzweigungen

Fallunterscheidungen



Schleifen

Anm: Auch **strukturiertes Exception-Handling** (try/catch/finally) dient der Ablaufkontrolle - wird erst später im OO-Kontext behandelt.

Ebenso : Rekursion und Multithreading

© H.Neuendorf

Kontrollstrukturen - Verzweigungen

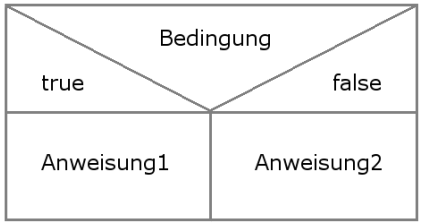
if - Bedingungsprüfung / Alternativverzweigung :

if (<Bedingung>) { <Anweisungen> } else { <Anweisungen> }

Überprüfung **Bedingung** auf Wahrheit / Falschheit bestimmt **Kontrollfluss**

Zwei **Zweige** : Bedingung wahr ⇒ ***if-Zweig*** Bedingung falsch ⇒ ***else-Zweig***

```
if ( x > y ) { max = x ; }   else  { max = y ; }

                                   //  else-Block darf leer sein  ⇒

                                   // else-Zweig kann fehlen – s.u.
```



Bedingung : Logischer **Ausdruck** vom Typ ***boolean***

Anweisungsblöcke : Ausführung **mehrerer** Anweisungen ⇒ in ***Block*** **{ }** *zu* fassen :

```
if  ( x < 0 )  {  x = x + 1 ;    IO.writeln (x) ;  }
else           {  x = x  - 1 ;   IO.writeln (x) ;  }
```

© H.Neuendorf

Kontrollstrukturen if-Verzweigung - Besonderheiten

1. else-Zweig kann leer sein - bzw. fehlen :

```
int x = IO.promtAndReadInt( "Eingabe: " );
if( x == 0 ) { /* spezielle Behandlung */ }
int y = 10*x ;
```

```
int tage = 365 ;   int jahr = … ;
if( jahr%4 == 0  &&  jahr%100 != 0  ||  jahr%400 == 0 )
    tage = 366 ;    // Schaltjahr !
// …
```

2. Dangling else :

Verschachtelte if-Anweisungen - ein else "hängt in der Luft" :

```
if ( x < y )
    if ( x > 0 )     x++ ;
  else   y++ ;          // auf welches if bezieht sich dieses else ??
```

Regel : Ein *else* gehört zum direkt vorausgehenden noch nicht zugeordneten *if*

3. Ternärer Operator : ?

```
int x = 10;   int y = 1;          int z =   x > 5   ?   2*y  :  3*y ;     // in z steht 2
```

© H.Neuendorf

Kontrollstrukturen if-Verzweigung - " else if "

Kein eigenes Sprachelement - nur **übersichtliche Formatierung**

Wird von **Eclipse** auch entsprechend formatiert …

Sinn :

Abfrage mehrerer Ausdrücke hintereinander mit davon abhängiger Fortsetzung

Kann durch **switch-Anweisung** (s.u.) jedoch manchmal **übersichtlicher** dargestellt werden …

```
class StrukturIfElse {
  public static void main( String[ ] args ) {
    int x = IO.promptAndReadInt( "Wert: " ) ;
    if( x > 10 )                        IO.writeln( " >10 " ) ;
    else if( x >=5 && x<=10 )   IO.writeln( "Zwischen 5 und 10" ) ;
    else if( x>=2 && x<=4 )     IO.writeln( "Zwischen 2 und 4" ) ;
    else if( x >=0 && x<=1 )    IO.writeln( "Zwischen 0 und 1" ) ;
    else                                IO.writeln( " <0 " ) ;
  }
}
```

© H.Neuendorf

Beispiel

```
class Pilzberatung  {
  public static void main( String[] args )  {

    IO.writeln( "Willkommen bei PILZEXPERT" ) ;

    char  c1  =  IO.promptAndReadChar( "Ist der Pilz rot?  (j/n)" ) ;
    boolean rot =   ( c1  ==   ' j'  ) ;

    char  c2  =  IO.promptAndReadChar( "Ist es ein Röhrenpilz?  (j/n)" ) ;
    boolean roehren =   ( c2  ==   ' j' ) ;

    if ( rot == true  &&  roehren == true )  {      // kürzer:    if( rot && roehren )
      IO.writeln( "Gibt keine roten Röhrenpilze" ) ;
    }
    if ( rot && !roehren )   {  // kürzere Formulierung für:  (rot==true)  &&  !(roehren == true)
      IO.writeln( "Fliegenpilz!" ) ;  }
    }
    if ( !rot && roehren )  {
      IO.writeln( "Speisepilz!" ) ;  }
    }
    if ( !rot && !roehren )  {
      IO.writeln( "Risiko!!" ) ;
    }
  }
}
```

© H.Neuendorf

Entscheidungsabfragen

```
// Ausführlich :
char antwort  =  IO.promptAndReadChar(  "Springt Auto an? ( j/n )"  ) ;      // Abfrage
if ( antwort == ' j'  )   IO.writeln( "prima!" ) ;
else                      IO.writeln( "schlecht!" ) ;
```

```
// Kompakte Formulierung :
// Zusammenfassung von Abfrage + Bedingungsprüfung
if (  IO.promptAndReadChar("Springt Auto an? ")   ==   ' j '   )   IO.writeln( "prima!" ) ;
else                                                               IO.writeln( "schlecht!" ) ;
```

Prinzip :

Abfrage wird in Bedingungsprüfung **eingebettet**

Abfrage wird **zuerst** ausgewertet $\rightarrow$ liefert Wert zurück, der für Vergleich z.Vfg. steht.

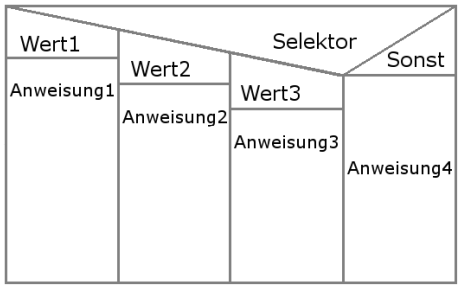
© H.Neuendorf

Kontrollstrukturen - switch *Mehrweg-Verzweigung*

Wert-Prüfung + **Verzweigung** gemäß **ganzzahligem Ausdruck** vom **Typ :**

int, short, byte, char (ab Java 7 auch **String** !)

(… oder entsprechendes Hüllklassen-Objekt oder enum-Typ)




```
int month ;     int days = 0 ;
month  =  IO.promptAndReadInt( "Monats-Nummer = " ) ;
switch( month )   {    // auch Term erlaubt
    // Case-Marker mit mehreren möglichen Alternativen  -  dank break
    case 1:   case 3 :   case 5 :   case 7 :   case 8 :   case 10 :  case 12 :
        days = 31;    // wenn Alternative zutrifft, wird Anweisung ausgeführt
        break ;        // wichtig:  damit switch verlassen wird !
                       // sonst :    folgende Anweisungen auch ausgeführt (Fall Through) !
    case 4 :   case 6 :   case 9 :   case 11 :
        days = 30;     break ;
    case 2 :
        days = 28;     break ;
    default :      // optionaler Default-Zweig - ausgeführt, wenn kein anderer zutrifft
        IO.writeln( "Kein gültiger Wert!" ) ;
}
```

Jeder case muss eindeutig sein !!

Es kann höchstens **einen** default-Zweig geben.

Wenn kein Fall zutrifft und kein default angegeben, bleibt **switch**-Anweisung **ohne** Wirkung.

© H.Neuendorf

Kontrollstrukturen - switch

Java 7

Ab **Java 7** auch mit **String** !

```
// …
String tagTyp ;
String tag = IO.promptAndReadString( "Wochentag: " ) ;
switch( tag ) {
   case "Montag" :
     tagTyp = "Wochenanfang" ;
     break ;
   case "Dienstag" :  case "Mittwoch" :  case "Donnerstag":  case "Freitag" :
     tagTyp = "Wochenmitte" ;
     break ;
   case "Samstag" :  case "Sonntag":
     tagTyp = "Wochenende" ;
     break ;
   default :
     tagTyp = "Ungültig" ;   // alles andere
}
IO.writeln( "Typ = " + tagTyp ) ;
```

Der Vergleich mit case-Werten funktioniert mit **equals( )** von **String**

Somit **Case Sensitive**

Vergleich mit if-else-Sequenzen :

Lesbarkeit + Effizienz !

Kompakterer Bytecode

In jeden **case** sind auch weitere Fallunterscheidungen oder Schleifen einbaubar

… wird rasch unübersichtlich !

© H.Neuendorf

switch

```
switch ( <Ausdruck> ) {
   case  <Konstante> :
        <Anweisung>
        // …
        <Anweisung>
        break ;
   case  <Konstante> :
        <Anweisung>
        // …
        <Anweisung>
        break ;
   default :
        <Anweisung>
        // …
        <Anweisung>
}
```

Ausdruck : Liefert ganzzahligen Wert vom **Typ** int, short, byte, char + String - **nicht** aber Typ **long** !

Case-Kombinationen durch Fall-**Aufzählung**

"Sonder"formen

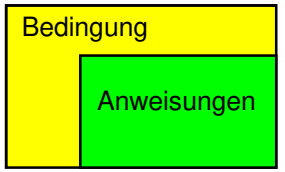
```
// case-Werte müssen Konstanten sein :
final int c1 = 10 ;    final int c2 = 20 ;
int n = 10 ;
switch( n ) {   // Benamte Konstanten (!) als case-Werte
   case c1 :   IO.writeln( "Die 10" ) ;   break ;
   case c2 :   IO.writeln( "Die 20" ) ;   break ;
   default :   IO.writeln( "Sonst was" );
}

public int getValue( int n ) {
    switch( n ) {   // Verlassen mit return-Anweisung
        case 1 :   return 100 ;
        case 2 :   return 200 ;
        default :  return 500 ;
    }
}
```

Aktuelle (nicht finale) Syntax-Erweiterungen (Java 11-13) nicht besprochen - noch Preview-Status.

© H.Neuendorf

Kontrollstrukturen - Schleifen

Wiederholte Ausführung



while - Schleife

Abweisschleife / Kopfgesteuerte Schleife :

while (<Bedingung>) { <Anweisungen> }

Prüfung Fortsetzungs-Bedingung **vor** jedem Schleifendurchlauf.

Bedingung ist Ausdruck, der **Booleschen Wert** liefert (**true** oder **false**)

(oder Boolean-Objekt)

⇒ Eventuell Schleifenrumpf *nie* durchlaufen, wenn Bedingung *nicht* zutrifft !

Zusammenfassung **mehrerer** Anweisungen in einen Anweisungs-Block **{ ... }**

```
int i = 1 ;   int sum = 0 ;    int n = 5 ;
while( i <= n ) {     // Abbruch :  wenn i größer als n  ⇒  Schleife verlassen
                      //  wird vor jedem erneuten Durchlauf mit aktuellen Werten geprüft.
    sum = sum + 1 ;
    i++ ;      // wenn  i >5  wird Schleife nicht mehr durchlaufen!
}
IO.writeln( sum ) ;    // hier wird Programm nach Verlassen der Schleife fortgesetzt.
```

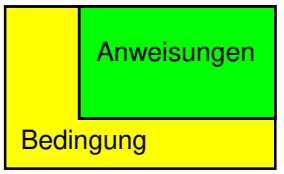
Vermeiden **Endlosschleifen :** Schleife muss **terminieren** ⇒ Abbruchbedingung checken !

© H.Neuendorf

Kontrollstrukturen - Schleifen

do-while - Schleife Durchlaufschleife / Fußgesteuert / nicht abweisend :

do { <Anweisungen> } while (Bedingung)



Prüfung der Fortsetzungsbedingung erst **am Ende** jedes Schleifendurchlaufs.

⇒ Schleifenrumpf somit stets **mindestens einmal** betreten und ausgeführt !

```
int i = 1 ;    int sum = 0 ;     int n = 5 ;


 do {
      sum = sum + 1 ;    // Anweisungen werden mindestens einmal ausgeführt !
      i ++ ;
 }  while( i <= n ) ;        // wenn  i >5  wird Schleife verlassen


 IO.writeln( sum ) ;      // hier wird Programm nach Verlassen der Schleife fortgesetzt
```

Anm: **do{ anweisungen } while(!bedingung)**

entspricht dem Konstrukt **do ... until** anderer Sprachen

© H.Neuendorf

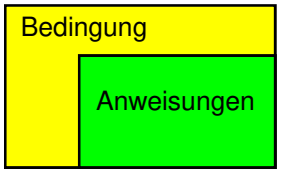
Kontrollstrukturen - Schleifen

for - Schleife **Kopfgesteuert - Gesamtanzahl Durchläufe steht fest.**

Aktuelle Anzahl Durchläufe durch **Laufvariable** gezählt = **Zählschleife**

Dreiteiliger Schleifen-Kopf :

for (*<In>; <Fb>; <V>*) { *<Anweisungen>* }



1. **Initialisierungsteil :** legt **Startwert** der **Laufvariablen** fest
2. **Fortsetzungsbedingung :** jedes mal **vor Betreten** der Schleife geprüft - wie bei while !
3. **Veränderungssteil :** **nach jedem Durchlauf** ausgeführt → verändert Laufvariable

```
int sum = 0 ;      int n = 5 ;
for( int i=1;  i<=n;  i++ ) {
    sum = sum + 1 ;
    IO.writeln( "Durchlauf Nr.  " + i ) ;
}
IO.writeln( sum ) ;   // Programm fortgesetzt
```

```
// Bsp - chars in UC-Tabelle angeordnet :
for( char n='A';  n <='Z';  n++ ) {
   IO.writeln( "Hier : " +  n ) ;
}
```

© H.Neuendorf

Kontrollstrukturen for - Schleife

In Schleife kann (u.a.) auch **dekrementiert** werden :

for (int i = 10 ; i > 3 ; **i = i - 3**) { /* … */ } // **i** nimmt Werte 10, 7, 4, 1 an

Alle Schleifen können **geschachtelt** werden :

Rumpf einer Schleife kann wieder Schleife enthalten - und auch andere Kontrollstrukturen.

```
// Multiplikationstabelle :
  int n = 10 ;
  int mult ;
  for( int i = 1; i <= n; i++ ) {
        for( int j = 1; j <= n; j++ ) {
              mult = i * j ;
              IO.write( mult + " ") ;
        }
        IO.writeln( ) ;
   }
 // i = 100 ;    j = 50 ;    Fehler !!
```

Die **im** Schleifenkopf **deklarierte Lauf-variable** ist **nur** *innerhalb* Schleife gültig !

© H.Neuendorf

Kontrollstrukturen for – Schleife Anmerkungen

Initialisierung + Veränderungsteil können aus **mehreren** Anweisungen bestehen :

Mit Komma trennen

Führt zu komplexen Schleifenkopf :

```
for ( int i = 1, sum = 0 ;   i <= n ;   i++ , sum = sum + 2 )  {  /* ... */ }
```

Verschieden typisierte Laufvariablen müssten **außerhalb** for-Schleife **deklariert** werden

Jeder Teil des **for-Schleifenkopfs** darf ***weggelassen*** werden :

Fehlende Initialisierung oder Veränderung $\Rightarrow$ quasi Übergang zur while-Scheife :

```
int i = 1;  int n = 10 ;
for( ; i<=n; ) { IO.writeln( i++ ) ; }
```

Keine Fortsetzungsbedingung $\Rightarrow$ **true** $\Rightarrow$ Endlosschleife :

```
for( int i=1 ; ; i++ ) { IO.writeln( i ) ; }
```

Endlosschleifen :

for(; ;) { … } **while(true)** { … }

© H.Neuendorf

Kontrollstrukturen break + continue

Strukturierte Schleifenabbrüche durch break-Anweisung : Kein **goto**

Sinn : Behandeln von Situationen, die weitere Ausführung sinnlos machen.

aber : Kontrollfluss wird unübersichtlicher $\Rightarrow$ vermeiden ! ("Verstreuung" der Logik)

Sprünge aus Schleifen : **break**

- **Kompletter** Abbruch des Schleifendurchlaufs
- Fortsetzung **hinter** dem Schleifenkonstrukt

```
int sum = 0 ;     int z =  100 ;
while( true ) {
  sum = sum + z ;      IO.writeln( sum ) ;
  if ( sum > 1000 )   break ;
}
// …
```

Sprünge aus Schleifen : **continue**

- Abbruch des aktuellen Schleifendurchgangs
- Sprung zur Fortsetzungsbedingung der Schleife
- Fortsetzung mit **nächstem** Durchgang

```
for( int n=-10;  n<=10;  n++ )  {
  if( n == 0 )  continue ;
  IO.writeln( "Wert = " + 1.0 / n ) ;
}
```

Anm: Auch **return-Anweisung** gilt als bedingte Sprunganweisung.

© H.Neuendorf

Schleifen - wann welches Konstrukt anwenden ?

```
while ( Ausführungsbedingung ) {          // Test der Ausführungsbedingung vor jedem Durchlauf
      Wiederholungsanweisungen
}
Anweisungen

do {                                      // Test der Ausführungsbedingung nach jedem Durchlauf
     Wiederholungsanweisungen
} while ( Ausführungsbedingung ) ;
Anweisungen

for ( Laufvariablenfestlegung ; Ausführungsbedingung ; Laufvariablenveränderung ) { // feste Zahl von
      Wiederholungsanweisungen                                                      // Wiederholungen
}
Anweisungen
```

Anwendungs-Kriterien :

Steht Anzahl der Wiederholungen **vor** Betreten der Schleife **fest** oder nicht ?

Wird **erst während** Schleifendurchläufe über Fortsetzung oder Abbruch entschieden ?

© H.Neuendorf

Schleifen - wann welche verwenden ?

1. **Anzahl der Wiederholungen steht von Anfang an fest oder kann berechnet werden :**

   ⇒ **for-Schleife verwenden :**

   Bsp:
   ```
   int n = IO.promptAndReadInt ( "Anzahl Durchläufe : " ) ;
   for( int i=1;  i<=n;  i++ )   {   /* etwas tun */  }
   ```

2. **Erst während Durchläufen wird über weitere Fortsetzung entschieden :**

   ⇒ **while- oder do ... while-Schleife verwenden :**

   Bsp:
   ```
   char c = IO.promptAndReadChar( "Aktion ausführen? (j/n) " ) ;
   while( c == ' j ' )  {
     //  etwas tun …
     c = IO.promptAndReadChar( "Nochmal (j/n) ? " ) ;
   }
   ```

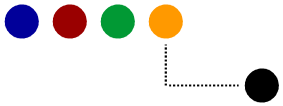
3. **Aber: Prinzipiell jede for-Schleife auch durch while-Schleife ausdrückbar !**

   Bsp:
   ```
   int n = IO.promptAndReadInt( "Anzahl Durchläufe : " ) ;      int i = 1 ;
   while( i <= n )  {   /* etwas tun … */    i++ ;    }
   ```

   Jedoch for-Schleife besser verständlich, da alle nötigen Infos **im Kopf** der for-Schleife konzentriert - und **nicht** über Schleifenkörper **verstreut** !

© H.Neuendorf

Java-Methoden



- **Methoden-Grundstruktur**
- Parameter
- Rückgabewerte (void + nicht void)
- Schachtelung von Methodenaufrufen
- Rekursion

© H.Neuendorf

Java-Methoden

Methoden sind elementare Mittel der **Modularisierung + Wiederverwendung !**

Eine Methode ist eine **"Maschine",** die definierte **Leistung** / **Service** erbringt.

Dazu muss Methode **aufgerufen** und mit erforderlichem **Input** versehen werden.

Methode = Kopf (header) + Rumpf (body)

- **Methodenkopf** :

  Deklaration → **Signatur** = Name / Schnittstelle / Parametrisierung / Rückgabeverhalten

  Kenntnis Methodensignatur befähigt, Methode aufzurufen.

- **Methodenrumpf** :

  Implementierung → Bei Aufruf der Methode auszführende Anweisungsfolge.

Methodendeklaration + Implementierung :

Methodenkopf **{** *Methodenrumpf* **}**

In Java-Klassen ist *keine* Trennung von Deklaration und Implementierung möglich !

(→ *Interfaces*)

© H.Neuendorf

Java-Methoden


```
class Groesstes {
   public static void printMax( int x, int y ) {  // Methodenkopf
      if ( x > y )    IO.writeln( x ) ;      // Methodenrumpf
      else            IO.writeln( y ) ;
      printGruss( ) ;
   }
   public static void printGruss( ) {        // Methodenkopf
      IO.writeln( "Hallo DH" ) ;    // Methodenrumpf)
   }
   public static void main( String[ ] args ) {
      int a = 10 ;    int b = 20 ;
      printMax( a , b ) ;      // Methodenaufruf ...
                               //… mit Parametern a und b
      printGruss( ) ;       // Aufruf parameterlose Methode
   }
}
```

Methoden **printMax()** und **printGruss()** stehen logisch auf gleicher **Ebene** wie die spezielle Methode **main( )**

Klassen können beliebig viele Methoden enthalten - **Reihenfolge egal !**

Methoden können **Parameter**-Werte entgegennehmen oder **parameterlos** sein.

Methoden können in ihrem **Rumpf andere Methoden aufrufen** →

Dabei wird Ausführung der aufgerufenen Methode an Aufrufstelle eingeschoben.

© H.Neuendorf

Signatur von Methoden

Methodenkopf : Signatur = Schnittstelle

```
public void  printMax( int x, int y )
   ↑     ↑     ↑          ↑
   1.    2.    3.         4.
```

1. Sichtbarkeit : legt fest, aus welchem Kontext Methode aufrufbar ist

a) **public** : auch jede andere Klasse darf die Methode aufrufen

b) **private** : Methode von "außen" nicht aufrufbar - nur aus Methoden der *selben Klasse* aufrufbar

2. Typ Rückgabewert : den Methode bei Aufruf zurückliefert

a) **void** ("leer") : Methode liefert **nichts** zurück (prozedural)

b) int, long, float, char ... : Methode **liefert Wert zurück** (funktional)

3. Methodenname : verschiedene Methoden einer Klasse dadurch unterschieden

4. Parameterliste : **Typ** + **Name** der Parameter, die bei Methoden-Aufruf zu initialisieren sind

a) Methoden können **mehr als einen Parameter** tragen : mit Komma getrennt aufzählen → s.o.

b) **parameterlose** Methoden : mit leeren Klammern deklarieren → void printGruss**( )** ;

© H.Neuendorf

Aufruf von Methoden

public void printMax(int x, int y)

Formale Parameter : in **Deklaration** „als Platzhalter“ angegeben

Aktuelle Parameter : bei **Aufruf** konkret übergeben

Keine Default-Werte definierbar

Aufruf von Methoden :

Nennung **Methodenname** + Mitgeben **typkorrekter Parameterwerte** **printMax(51, 10) ;**

Terme erlaubt : **printMax(10, 2*b-1) ;**

Aufruf parameterloser Methoden nur mit **leerer** Parameterliste **printGruss( ) ;**

Parameterübergabe : ***Kopie*** des **aktuellen** Parameters an **formalen** Parameter ⇒

Typkompatibilität gefordert !

© H.Neuendorf

Methoden mit Wertrückgabe

```
class Groesstes {
   public static int  calcMax( int x, int y ) {
       if( x > y )  return x ;    // Rückgabe durch return
       else         return y ;
    }
    public static void main( String[ ] args )  {
       int a = 10 ;    int b = 20 ;
       int z ;
       // Aufruf  + Zuweisung des Rückgabewerts an  z :
       z  =  calcMax(  a , b ) ;

       // Verwerfen des Rückgabewerts :
       calcMax( 2 , 3 ) ;
       // …
   }
}
```

Prozeduren geben **keinen** Wert zurück ⇒

Rückgabetyp **void** - **kein** return nötig.

Funktionen geben **einen Wert** zurück ⇒

Rückgabetyp ist im **Kopf** zu deklarieren !

Müssen mit **return-Anweisung** abschließen.

Ausführung der return-Anweisung **beendet** Methodenaufruf.

Compiler prüft korrekten Typ und korrekte Platzierung der return-Anweisung !

Nur ein einziger Rückgabewert möglich
Aber : Auch ganze Objekte !

Auch **quasi prozedural** aufrufbar - ohne Rückgabewert entgegen zu nehmen.

© H.Neuendorf

return-Anweisung in Prozeduren

```
class Test {
  public static void main( String[ ] args ) {
      gruesse( 't' ) ;

      int n = IO.promptAndReadInt( "Wert: " ) ;
      if( n== 0 ) return ;
      IO.writeln( 1.0 / n ) ;
  }

  public static void gruesse( char c ) {
      if (  c == '0' )  return ;
      IO.writeln( "Hallo " + c ) ;
  }
}
```

Verwendbar, um Abarbeitung der Methode **abzubrechen** und diese zu verlassen.

Anweisung : **return ;**

Natürlich **ohne** Rückgabewert !

Prinzipiell auch in **main()** nutzbar, um weitere Programmbearbeitung geordnet **abzubrechen**.

return ist im Grunde auch eine **bedingte Spunganweisung** - ebenso wie break und continue.

© H.Neuendorf

Methoden : Vier grundsätzliche Varianten

```
class MethTest {
  // prozedural - parameterlos
  public static void m1( ) {
    IO.writeln( "Hallo an der DH" ) ;
  }
  public static void main( String[ ] args )  {
    m1( ) ;
  }
}
```

```
class MethTest {
  // prozedural - parametertragend
  public static void m2( String s ) {
    IO.writeln( "Hallo in " + s ) ;
  }
  public static void main( String[ ] args )  {
    m2( "Mosbach" ) ;
  }
}
```

```
class MethTest {
  // funktional - parameterlos
  public static double m3( ) {
    return 3.1415927 ;
  }
  public static void main( String[ ] args )  {
    double pi = m3( ) ;
  }
}
```

```
class MethTest {
  // funktional - parametertragend
  public static double m4( double d ) {
    return d*d ;
  }
  public static void main( String[ ] args )  {
    double piQuad = m4( 3.1415927 ) ;
  }
}
```

© H.Neuendorf

Seiteneffekte von Methodenaufrufen

```
class Test {
  public static void main( String[ ] args ) {
     int a, b ;
     int c = max( a = 5, b = 10 ) ;
     // … Weiterrechnen mit  a, b, c …
  }

   public static int max( int n, int m ) {
       if (  n > m )  return n ;
       else           return m ;
   }
}
```

Die Variablen **a** und **b** werden erst im Methodenaufruf initialisiert.

Mit diesen Werten wird der **Methodenaufruf** durchgeführt.

Die beiden Variablen haben nach dem Aufruf die Werte **5** bzw. **10**.

In Java noch komplexere Formen (wie in C/C++) nicht möglich.

Typischer Seiteneffekt :

Eher unschöner und tendenziell unübersichtlicher Programmierstil.

© H.Neuendorf

Methoden - Wertrückgabe + Schachtelung


```
class Groesstes {
   public static int  calcMax( int x, int y ) {
       if( x >y )   return x ;     // Rückgabe
       else         return y ;
    }
   public static void gibAus( int a ) {
      IO.writeln( "Wert :  " + a ) ;
   }

    public static void main( String[ ] args )  {

       int a = 10 ;    int b = 20 ;
       int z  =  calcMax(  a , b ) ;
      gibAus( z ) ;

      a = 45 ;   b = 21 ;
      gibAus(  calcMax( a, b )   ) ;
   }
}
```

Als Input zum Aufruf einer Methode kann direkt ein anderer Methodenaufruf eingesetzt werden, der den ***korrekten*** **Rückgabetyp** hat !

Umweg über entsprechend typisierte Variable nicht erforderlich.

⇓

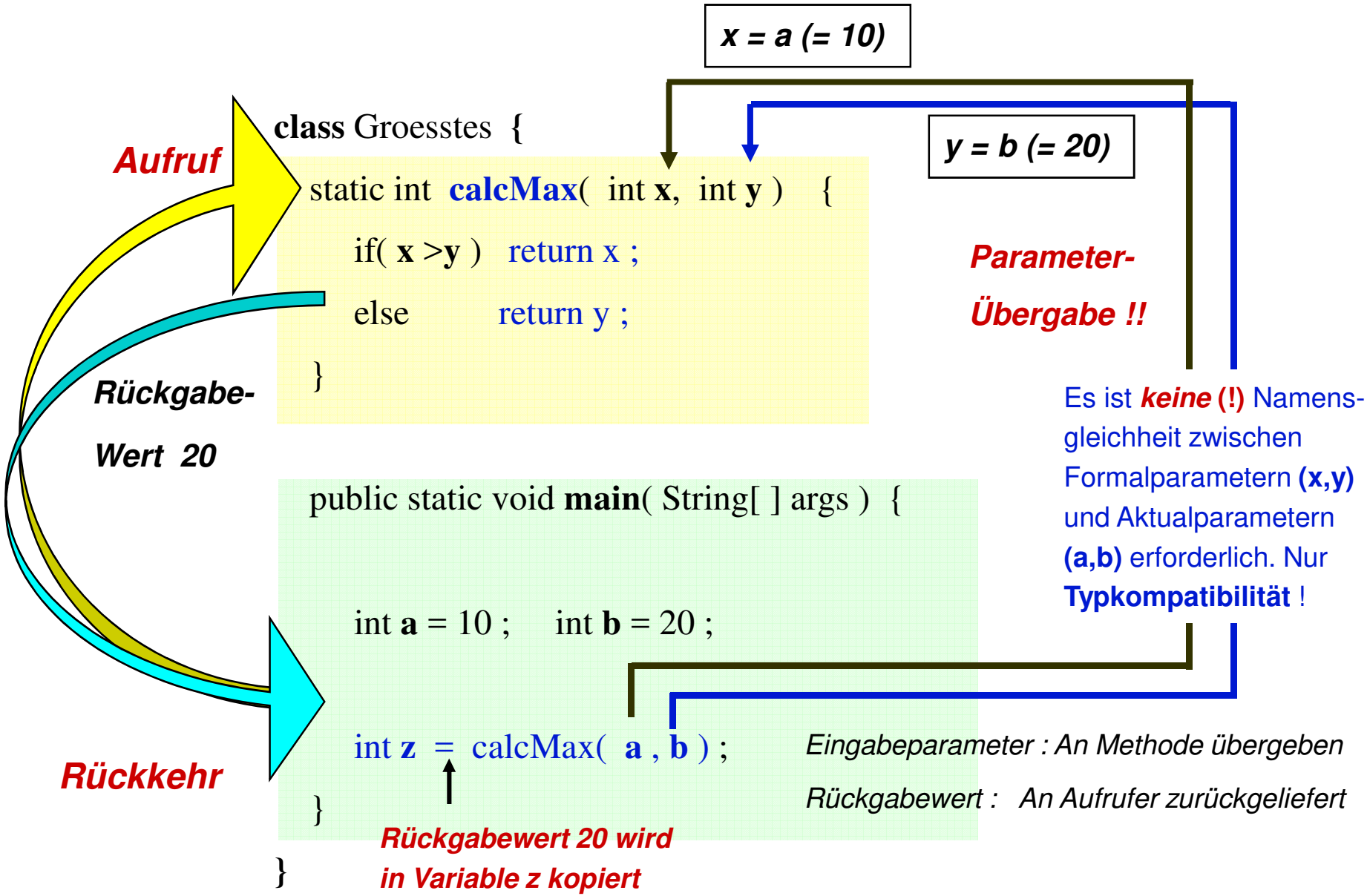
Methodenaufrufe können (beliebig tief) geschachtelt werden !

Methode **gibAus( )** muss mit int-Wert aufgerufen werden - Optionen :

- direkte Übergabe des Werts
- Übergabe Variable, die Wert trägt
- Methodenaufruf, der Wert zurückgibt

© H.Neuendorf

Methoden : Parameter-Übergabe + Wert-Rückgabe



© H.Neuendorf

Methoden - Finale Methodenparameter


```
class Kubisch {
   public static double hochDrei( final double x ) {
      //  x = 10.0 ;    Fehler !!
      double fx = x * x * x ;      // ok!
      return  fx ;      // oder :  return x * x * x ;
    }
   public static void main( String[ ] args ) {
      double y  =  hochDrei( 2.75 ) ;
      IO.writeln( "Wert = " + y ) ;
    }
}
```

Finale Methodenparamter dürfen in Methode **verwendet** aber **nicht im Wert verändert** werden !

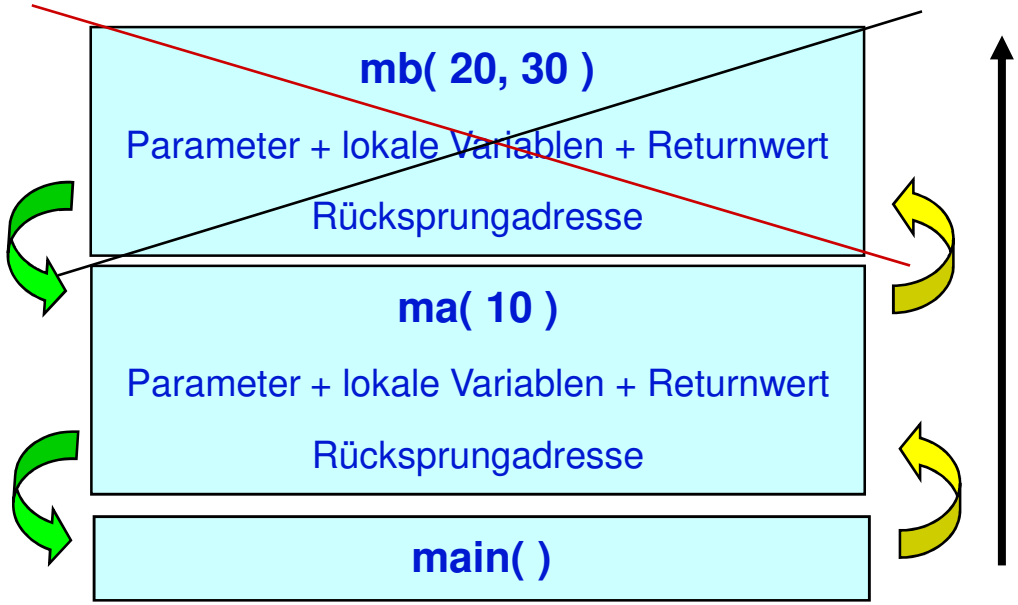
Kontrolle der Entwicklerintention durch Compiler.

Gute Methoden sollten …

- ✓ eine *einzige* klar definierte Aufgabe gut erledigen - und Seiteneffekte vermeiden.
- ✓ übersichtlich bleiben - *nicht zu viele Parameter und nicht zu viel Coding* aufweisen.
- ✓ nützlich und wiederverwendbar sein - *nicht zu wenig* Coding enthalten.
- ✓ alle nötigen *Daten als Parameter* erhalten - und nicht von globalen Variablen abhängen.
- ✓ gut *dokumentiert* werden.

© H.Neuendorf

Methoden - Verwaltung auf Stack-Speicher der JVM




```
class Methoden {
   public static void mb( int u, int v ) {
        // ...
   }
   public static void ma( int n ) {
        mb( 20,30 ) ;
        // …
   }
   public static void main( String[ ] args ) {
        ma( 10 ) ;
        // …
   }
}
```

Stackprinzip = Last In First Out (**LIFO**)

Methodenstack verwaltet **Parameter**, **lokale Variablen** und **Rücksprungadressen** der aufgerufenen Methoden.

Bereich der **zuletzt** aufgerufenen Methode liegt "oben" - wird nach Terminierung der Methode **freigegeben**.

Der Methodenstack oszilliert im Rhythmus der Methodenaufrufe.

Automatische Verwaltung durch **JVM** - ist in Größe **begrenzt.**

Stellt **zusätzlichen Aufwand** dar - im Vergleich zur direkten, sequenziellen Ausführung von Anweisungen.

Zweiter JVM-Speicherbereich :

Heap (s.u.)

"Ungeordneter" Haufenspeicher zur Ablage der Objekte

Beliebige Zugriffsreihenfolge

Lebensdauer durch Programm + **GC** verwaltet

© H.Neuendorf

Methodenstruktur – die "reine Lehre"

```
public static double vollTanken( double kapazitaet ,
                                 double anfangsstand ) {
  // precondition :
  if( fuellstand( ) / kapazitaet > 0.9 )
      throw new RuntimeException( "Voll" ) ;
  oeffneVentile( ) ;      startePumpe( ) ;
  while( fuellstand( ) / kapazitaet < 0.95 ) {
    // invariants :
    if( anfangsStand + zulauf( ) – fuellstand( ) > 2.0 ) {
      stopPumpe( ) ;  schliesseVentile( ) ;
      throw new RuntimeException( "Sensorfehler" ) ;
    }
    pumpe10Liter( ) ;
  }
  stopPumpe( ) ;    schliesseVentile( ) ;
  // postconditions :
  if( fuellstand( ) / kapazitaet < 0.9 )
      throw new RuntimeException( "Unvollständig" ) ;
  }
  return  fuellstand( ) ;   // alles ok !
}
```

Korrektes Arbeiten der Methode wird durch **Kontrolle dreier grundsätzlicher Beschränkungen** (Randbedingungen / Zusicherungen / Constraints) garantiert :

1. **Voraussetzungen (preconditions)**
2. **Konstanten (invariants)**
3. **Nachbedingungen (postconditions)**

Häufigste Verwendung :

Prüfung von **preconditions** und Werfen von RuntimeExceptions

Bsp: IllegalArgumentException

NumberFormatException …

© H.Neuendorf

Methodenstruktur – "Mikro-MVC"

```
class BMIRechner {
  // Model – kümmert sich nur ums Fachlogische :
  public static double bmi( double gewicht, double groesse ) {
    double bmi = gewicht / (groesse * groesse) ;
    return IO.round( bmi, 2) ;
  }
  // View – kennt nur die Präsentation :
  public static void datenAusgabe( double wert ) {
    IO.writeln( "**********" ) ;
    IO.writeln( "* Messwert = " + wert + " *" ) ;
    IO.writeln( "**********" ) ;
  }
  public static double datenEingabe( String abfrage ) {
    double wert = IO.promptAndReadDouble( abfrage + " :  " ) ;
    return wert ;
  }
  // Controller – koordiniert Datenflüsse zwischen M+V :
  public static void main( String[ ] args ) {
     double gew = datenEingabe( "Gewicht" ) ;
     double grs = datenEingabe( "Groesse" ) ;
     double bIndex = bmi( gew, grs ) ;
     datenAusgabe( bIndex ) ;
  }
}
```

Model-View-Controller Pattern

Klare **Aufgabenverteilung** für jede Methode – **keine Vermischung** von Modellogik und Präsentation :

Single Responsibilty Principle

⇓

Auswechselbarkeit der einzelnen Teile – ohne andere Teile zu beeinflussen.

*Das **MVC-Pattern** ist eigentlich auf **Klassenebene** angesiedelt und macht erst dort wirklich Sinn (View-, Model-, Controller-Klassen) …*

Dennoch lässt sich seine Idee schon im Kleinen erfassen.

Klare Aufgabenteilung + Modularisierung ergeben flexible Austauschbarkeit und Wiederverwendbarkeit.

© H.Neuendorf

Imperative versus Applikative Algorithmen

Imperative / Iterative Algorithmen

- Beruhen auf **Schleifen**
- Arbeiten mit **veränderlichen** Werten
- Bewirken **Zustandsänderungen** bei Ausführung
- Modell : Rechner, der Werte speichert und verändert

```
public static long fak( int n )  {
  long f = 1 ;
  for( int i=1; i<=n; i++ )   f = f * i ;
  return f ;
}
```

Applikative / Funktionale Algorithmen

- Beruhen auf **Funktionsdefinition und Rekursion**
- Arbeiten mit **unänderlichen** Werten
- Bewirken **keine** Zustandsänderungen bei Ausführung
- **Prinzip Divide & Conquer / Teile & Herrsche :**

  Schrittweise Vereinfachung bis zum Basisfall =

Rekursive **Rückführung** auf identisches Problem mit **kleinerer** Eingabemenge durch entspr. Deklaration.

```
public static long fak( int n )  {
  if ( n<=0 )   return 1 ;
  else   return  n * fak( n-1 ) ;
}
```

© H.Neuendorf

Rekursion Methoden-Selbstaufruf ⇒ Wiederholung ohne Schleifen !

Schleifen : Iteration ↔ **Rekursion : Selbstaufruf**

FAKULTÄT **n!** = 1*2*3* ....*(n-1)*n ***iterative Darstellung***

Bildungsgesetz ergibt ***rekursive Darstellung :***

n! = (n - 1)! * n

Basisfall → n=0 : 0! = 1

Vorteil : **Eleganz + Verständnis**

Problem : **Speicherbedarf + Performanz**

Daten **aller** nichtabgeschlossenen Aufrufe (incl. Rücksprungadressen) müssen auf **Laufzeitstack zwischengespeichert** werden **!**

Zahlreiche Methodenaufrufe involviert **!**

Prinzip:

Problem wird **auf sich selbst** zurückgeführt.

Rekursiv geführt bis zum trivialen **Basisfall**.

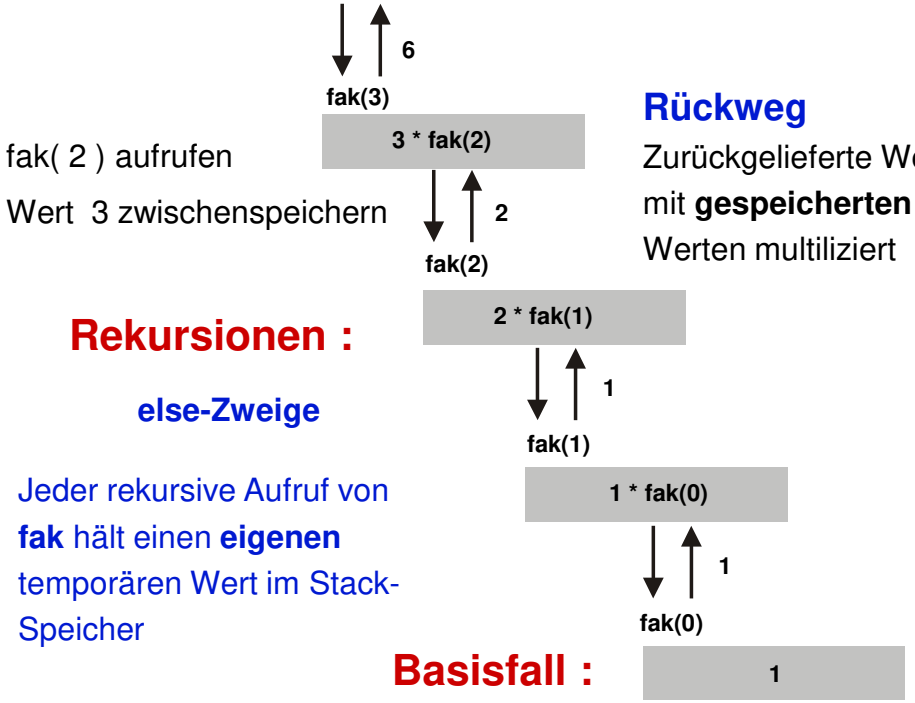
Rekursive Algorithmen sind **gleichmächtig** wie iterative Algorithmen.

Jedes iterative Problem kann auch rekursiv gelöst werden.

© H.Neuendorf

Rekursion Methoden-Selbstaufruf

```
public static long fak( int n ) {
  // Basisfall 0 oder 1 :
  if ( n<=0 ) { return 1 ; }
  // Rekursion:
  else { return n * fak( n-1 ) ; }
}
```



fak(2) aufrufen

Wert 3 zwischenspeichern

Rückweg

Zurückgelieferte Werte mit **gespeicherten** Werten multiliziert

Rekursionen :

else-Zweige

Jeder rekursive Aufruf von **fak** hält einen **eigenen** temporären Wert im Stack-Speicher

Basisfall :

if-Zweig

Voraussetzungen :

1. Methode muss **Basisfall** enthalten
2. Rekursion muss sich auf Basisfall **zubewegen**
3. Basisfall muss **erreicht** werden

Andernfalls **Stack-Overflow !**

Im **Gegensatz zur Iteration** (und generell dem Vorgehen imperativer Sprachen) kommt die **Rekrusion ohne Zustandsänderung** von Größen aus :

Statt in einer Schleife den Wert einer Laufvariablen und einer Produktvariablen ständig zu verändern, finden bei der Rekursion **weitere Methodenaufrufe** mit **neuen Parameterwerten** statt.

Im Beispiel : **n** wird **nicht** verändert !

© H.Neuendorf

Rekursion Beispiele

Rekursive Exponential-Definition für ganzzahlige positive Exponenten :

```
public static double power( double x, int y ) {
    if( y == 0 )  return 1 ;    // Basisfall
    else          return ( x * power( x, y-1 ) ) ;   // Rekursion
}
```

$$x^y = x \cdot x^{y-1}$$

Rekursive Summation ganzer positiver Zahlen :

```
public static int sum( int n ) {
    if( n==0 )  return 0 ;     // Basisfall
    else        return (n + sum( n-1 ) ) ;    // Rekursion
}
```

$$\sum_{i=0}^{n} i = \left(\sum_{i=0}^{n-1} i \right) + n$$

Rekursive Modulo-Definition :

```
public static int mod( int a, int b ) {
    if( a < b )  return a ;   // Basisfall
    else         return ( mod( a-b, b ) ) ;    // Rekursion
}
```

$$\mathrm{mod}(a,b) = \begin{cases} a & \text{falls } a < b \\ \mathrm{mod}(a-b,b) & \text{falls } a \geq b \end{cases}$$

© H.Neuendorf

Rekursion Beispiele

Produkt zweier positiver Zahlen :

```
public static long prod( int n, int m )  {
    if( n == 0 )   return 0 ;
    else           return  prod( (n-1),m ) + m ;
}
```

Test auf Geradzahligkeit positiver Zahlen :

```
public static boolean even( int n ) {
  if( n == 0 )  return true ;
  else          return  ! even( n-1 ) ;
}
```

Test auf Geradzahligkeit - wechselseitiger Aufruf :

```
public static boolean even( int n ) {
    if( n == 0 )  return true ;
    else          return  odd( n-1 ) ;
}
public static boolean odd( int n ) {
    if( n == 0 )   return false ;
    else           return  even( n-1 ) ;
}
```

© H.Neuendorf

Rekursion Rekursivierung von Iterationen

Iterative Form :

Operation **op** wird auf Variable **var** ausgeführt, solange Methode **test** true zurück liefert.

Annahme: Operation hat Einfluss auf Ergebnis von test(var).

```
while( test( var ) ) {
  op( var ) ;
}
```

Rekursive Form :

```
public static void rekursion( … var ) {
  if( test( var ) ) {
    rekursion( op(var) ) ;
  }
  else { … }
}
```

```
// Beispiel :

int i = 1 ;      // Iteration :
while( i <= 10 ) {
  IO.writeln( i ) ;   i++ ;
}

int i = 1 ;      // Rekursion :
public static void rekursion( int i ) {
    if( i <= 10 ) {
      IO.writeln( i ) ;
      rekursion( (i+1) ) ;
    }
    else IO.writeln( "Basis" ) ;
}
```

© H.Neuendorf

Rekursion Probleme

Hoher Speicherbedarf + zahlreiche Selbstaufrufe

FIBONACCI - Folge → jedes Glied ist Summe der beiden Vorgänger

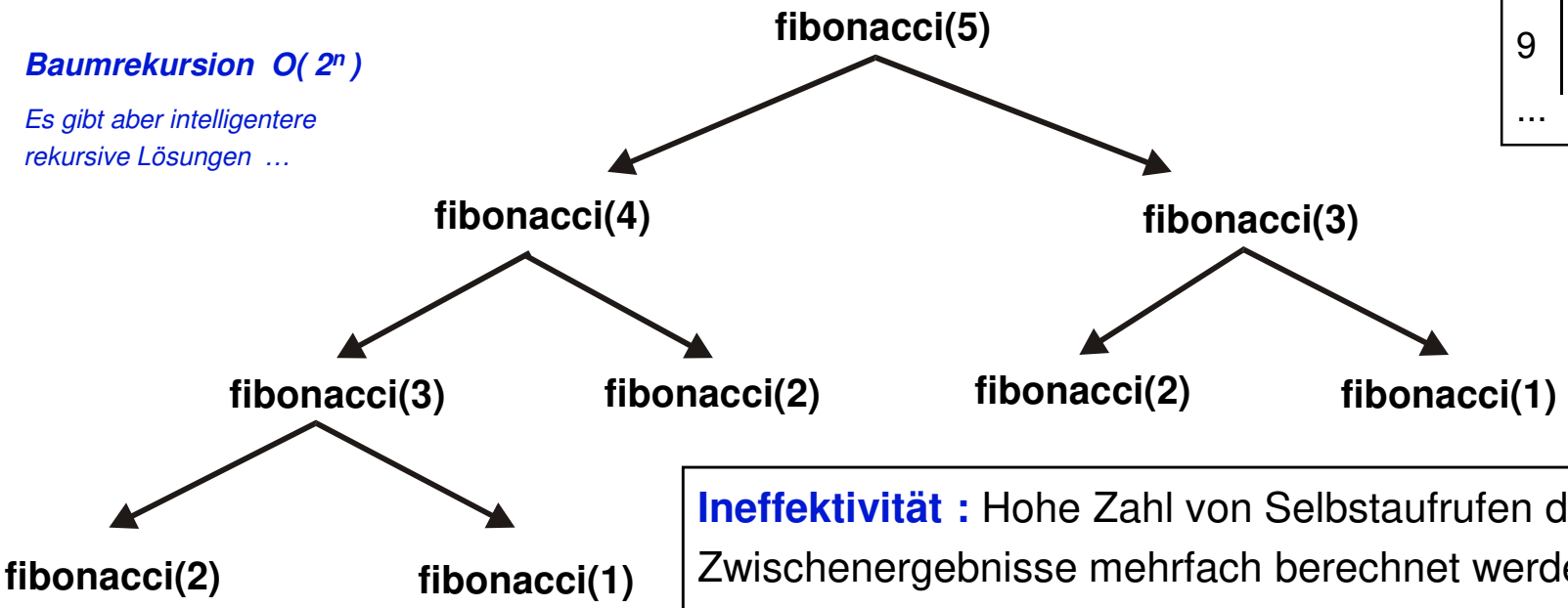
Glieder (1) und (2) haben Wert **1** = Basisfall

```
public static long fibonacci( int n )  {
    if ( n==1 || n == 2)    { return 1 ; }    // Basisfall
    else {   return  fibonacci( n-1 ) + fibonacci( n-2 ) ;  }   // Rekursion
 }
```

n	fibonacci(n)
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
...	...

Baumrekursion O(2^n)

Es gibt aber intelligentere rekursive Lösungen …

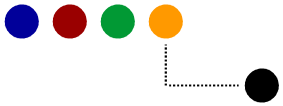


Ineffektivität : Hohe Zahl von Selbstaufrufen da zahlreiche Zwischenergebnisse mehrfach berechnet werden

⇒ **Simple Rekursion nicht das geeignete Mittel !**

© H.Neuendorf

Struktur prozeduraler Java-Programme



- **Kontext-Aspekte :**
- Lokale und "Globale" Namen (Variablen, Konstanten)
- Blockkonzept - Prinzip Lokalität / Kontext (Scope)
- Namensraum
- Sichtbarkeit und Lebensdauer
- Namenskonventionen
- Prinzipien der Strukturierten Programmierung

© H.Neuendorf

Lokale + Globale Namen

Sichtbarkeit + Lebensdauer


```
class Kreis  {

  // Lokale Variablen :   radius  und  f
  public static double flaeche( double radius ) {
    double f = 3.14 * radius*radius ;
    return f ;
  }

  // Lokale Variablen :   radius  und  kreisFlaeche
  public static void main( String[ ] args ) {
    double radius = IO.promptAndReadDouble( "r = " ) ;
    double kreisFlaeche = flaeche( radius ) ;
    IO.writeln( "Flaeche = " + kreisFlaeche ) ;
  }
}
```

Lokale Variablen :

Auf Methodenebene deklariert

Nur im dortigen Kontext verwendbar.

Außerhalb ihrer Methode nicht "sichtbar" / nicht verwendbar / nicht existent !

Ihr Speicherplatz wird jedes Mal beim Methodenaufruf **erneut auf dem Stack** angelegt.

Bei Methodenende wird ihr Speicherplatz freigegeben + der lokale Variablenwert gelöscht.

Lokale Variablen "leben" *nur* während der Ausführung *ihrer* Methode + in deren Kontext.

© H.Neuendorf

Lokale + Globale Namen

Sichtbarkeit + Lebensdauer


```
class Global  {

  public static int a = 5 ;       // "global"
  public static final double PI  =  3.14 ;

  public static void  tuWas( )  {
    int y = a  + x + PI;
    a = 15 ;  // leider erlaubt !
  }

  public static void main( String[] args ) {
    double d = 2.3 * PI ;
    a = 10 ;
    tuWas( ) ;
  }
}
```

*) *In Java gibt es **keine echten** globalen Variablen / Konstanten – es handelt sich um (**statische**) Attribute der Klasse.*

"Globale" Variablen und Konstanten : *)

Auf Klassenebene deklariert

Somit sichtbar + verwendbar in allen Methoden der Klasse.

Speicherplatz wird schon angelegt, wenn Klasse **geladen wird**.

Erst bei Programmende wird Speicherplatz freigegeben.

Bis dahin behält **globale** Variable / Konstante ihren Wert.

Default-Werte für Initialisierung wirksam.

Globale Variablen / Konstanten "leben" während *gesamter* Programmausführung.

© H.Neuendorf

Blockkonzept


```
class Bloecke  {
   public static double a = 1.0 ;     // global
   public static void  tuWas( )  {
      for( int i = 1 ;  i < 10 ;  i++ )  {
         double d = 9.0 ;      // lokal
         d = d + a  ;   // OK !
         for( int j = 1 ;  j < 5 ;  j++  )  {
            // OK : Tiefergeschachtelter Block !
            d = d + 1.0 ;
         }
         IO.writeln( " Wert = " + d ) ;    // OK !
      }
      // Fehler:  Zugriff außerhalb Block !
      d = d + 2.0 ;
      IO.writeln( " Wert = " + d ) ;
   }
   public static void main( String[ ] args )  {
         tuWas( ) ;
    }
}
```

Kontext Lokaler Variablen :

In Methoden-Blöcken deklariert bzw. in Unterblöcken von Methoden !

Nur **innerhalb** Block **zugreifbar**, **in dem** sie deklariert wurden **+** in darin enthaltenen **tiefer geschachtelten Unter-Blöcken.**

Variable **d** "lebt" **im** Block der for-Schleife in der **d** deklariert wurde.

Auch in darin **enthaltenen** Unter-Blöcken angesprechbar.

Jedoch **außerhalb** "ihres" Blocks **nicht** ansprechbar / verwendbar !

© H.Neuendorf

Namensräume

Methoden repräsentieren eigenen lokalen Namensraum

```
class Namen  {                                   Namensraum 3

                          Namensraum 1

    public static int  tuWas( int para )  {
        int x = 10 ;
        x = para ;
        return x ;
    }

                          Namensraum 2

    public static int  tuDas( int para )  {
        int x = 20 ;
        x = para ;
        return x ;
    }

     public static int x = 30 ;

}
```

Trotz Namensgleichheit kein Namenskonflikt :

Jeder Methode verfügt über eigene Variable **x .**

In Methode : Lokale Variable

→ Nur sichtbar + benutzbar in Methode

→ Außerhalb Methode verborgen

Die gleichnamigen Variablen leben in verschiedenen Namensräumen !

Namen in einem Namensraum **unabhängig** von Namen in anderem Namensraum wählbar !

Wird mehrfach vergebener Variablenname benutzt, wird immer **die** Variable angesprochen, **in deren Namensraum** man sich befindet !

© H.Neuendorf

Namensräume


```
class Bloecke  {
  public static void main( String[] args ) {

    // duplicate local variable …
    int x = 1 ;
    {  int x = 3 ;  }   //  Fehler !

    int y = 10 ;
    for( int i=0;  i<5;  i++ )  {
      int y = 5 ;      //  Fehler !
      // …
    }

  }
}
```

Unzulässig sind Variablen mit gleichem Namen im selben Namensraum !

Block allein oder bloße Kontrollstruktur **genügt *nicht*** zur Verdeckung !

Methodenrahmen erforderlich !

Aber :

Bei umgekehrter Anordnung werden die Konstrukte jeweils akzeptiert !

z.B.: `{ int x = 3; } int x = 10 ;`

Warum ?

© H.Neuendorf

Lokale + Globale Variablen


```
class Sichten  {
    public static int x = 10 ;
    public static int y = 20 ;

   public void m ( int par ) {
       int x ;
       x = par ;
   }
   //  ...
}
```

Sichtbarkeit der "**globalen**" Variablen **x** wird **unterbrochen** durch Sichtbarkeitsbereich der **lokalen** gleichnamigen Variablen **x**

Das lokale x verdeckt das gleichnamige globale x

Sichtbarkeit = Programmbereich, in dem auf Variable zugegriffen werden kann :

Von: **Deklaration der Variablen**

Bis: **Ende des Blocks, in dem Variable deklariert wurde**

Variablen innerer Blöcke verdecken gleichnamige globale Variablen.

Gutes **Mikro-Design** : ***Minimized Scope*** →

Lokale Variable sollte nur in den Blöcken sichtbar sein, in denen sie auch wirklich benötigt wird - z.B.:

⇒ Laufvariable der **for-Schleife** lebt nur in dieser.

⇒ Besser als **while-Schleife** mit stets externer Variablendeklaration.

© H.Neuendorf

Sichtbarkeit + Lebensdauer


```
class Sichten  {

     public static int g = 100 ;     // global!

    public static void P ( ) {
         int a = 30 ;
         …  [3]  …
    }
    public static void Q ( ) {
         int b = 20;
         [2] …  P( ) ;  … [4]
    }
    public static void main( String[ ] args ) {
         int m = 10 ;
          [1] …   Q( ) ;  …  [5]
    }
}
```

Sichtbarkeit *nicht* identisch mit Lebensdauer !

Variablen **existieren auch** während sie **nicht sichtbar** sind + **behalten** ihren **Wert …**

… mit dem sie **später wieder sichtbar / verwendbar** sind.

[1]	[2]	[3]	[4]	[5]	// Zeitpunkte
g	g	g	g	g	// **globale** Variable
m	~~m~~	~~m~~	~~m~~	m	// **lokale** Variablen
	b	~~b~~	b		
		a			

Zeitpunkte

[1]: Programmstart

Speicherplatz für globale Variable **g** angelegt

Aufruf von main( ) - Speicherplatz für lokale Var **m** angelegt

[2]: Aufruf von Methode Q( )

lokale Variable b lebt + ist sichtbar

m ist momentan nicht sichtbar, da Programm gerade in Q( ) !

[3]: Innerhalb von Q( ) Aufruf der Methode P( )

lokale Variable a lebt + ist sichtbar

b ist momentan nicht sichtbar, da Programm jetzt in P( ) !

[4]: P( ) ist beendet

Speicherplatz für lokale Variable a ist freigegeben

lokale Variable b ist wieder sichtbar

[5]: Q( ) ist beendet

Speicherplatz für lokale Variable b ist freigegeben

lokale Variable m ist wieder sichtbar

main( ) endet

Speicherplatz für lokale Variable m ist freigegeben

Programmende - globale Variable g wird freigegeben

© H.Neuendorf

Namenskonventionen in Java

Jeder Dummkopf kann Code schreiben, den ein Computer versteht. Gute Programmierer schreiben Code, den Menschen verstehen.

M. Fowler : Refactoring

Keine Vorschrift - aber allgemein **üblich** !

Einheitlichkeit + Konsistenz erhöht **Lesbarkeit !**

- **Klassennamen** beginnen mit Großbuchstaben *IO, String, Konto*
- In Klassennamen aus **mehreren Worten** beginnt jedes Wort mit Großbuchstaben :

  ZinsUndRatenRechner
- **Methodennamen** nach **Verben** benennen + mit Kleinbuchstaben *writeln()*

  In **zusammengesetzten Methodennamen** schreibt man Wortanfänge groß :

  promptAndReadDouble()
- **Variablennamen** beginnen mit Kleinbuchstaben *wert*
- **Konstanten** bestehen aus Großbuchstaben *MINIMUM = 50*

  Konstantennamen aus **mehreren Worten** durch Unterstriche zusammengesetzt :

  MAX_VALUE = 100
- **Methoden,** die **boolean** zurückliefern, sollten Frage formulieren *isOpen(), hasNext()*
- **Keine "Sonderzeichen"** (ä,ü,ö,ß) verwenden.

***CamelCase** für zusammengesetzte Namen*

© H.Neuendorf

Prinzipien der Strukturierten Programmierung

Ziel : Verbesserung Korrektheit, Lesbarkeit, Wartbarkeit

Regeln der Programmierung im Kleinen :

- Nur strukturierte / bedingte Anweisungen - kein *goto*
- Modulare Strukturen schaffen durch Verteilung auf separate Methoden
- Konstanten-Vereinbarungen nutzen
- Selbsterklärende und fachspezifische Bezeichner wählen

Prinzip der Programmierung im Großen :

- Schrittweise Verfeinerung – **Top Down-Entwurf** *divide et impera*
- Große, schwer überschaubare Problembereiche werden in kleiner Teilprobleme zerlegt und gesondert bearbeitet / als separate Module zur Vfg. gestellt.

Im Rahmen der **prozeduralen Programmierung** nur **eingeschränkt** möglich …

Objektorientierung liefert völlig neue Mittel der **Beherrschung von Komplexität !**

© H.Neuendorf