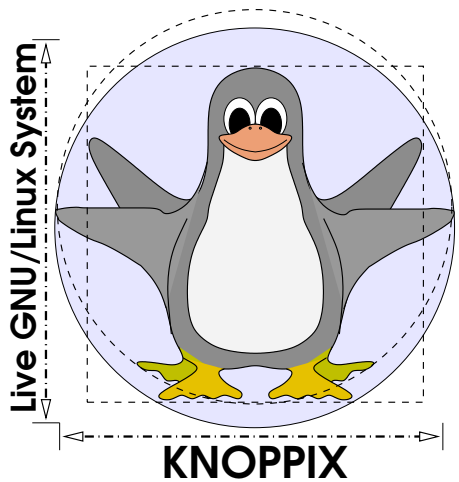
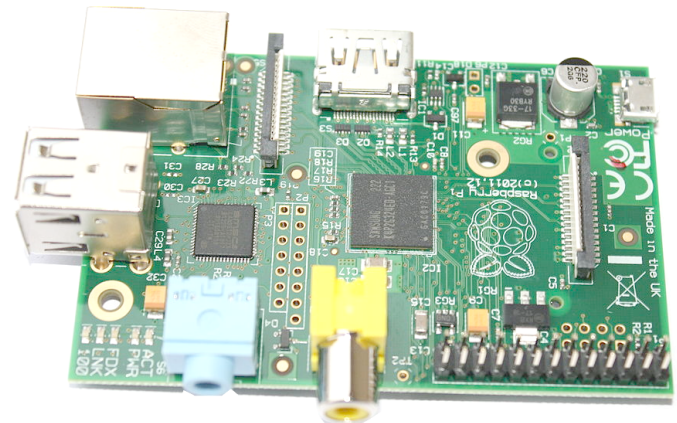
Grundlagen der Informatik

Prof. Dipl.-Ing. Klaus Knopper





Vorlesung mit Übungen an der HS Kaiserslautern im Sommersemester 2020

Organisatorisches (1)

☞ Vorlesung mit Übungen **Grundlagen der Informatik** SS2020 bis auf weiteres als Online-Vorlesung!

Vorläufige Terminplanung:

Fr. 10:00 - 12:00 Uhr	Vorlesung
Fr. 12:30 - 14:45 Uhr	Übungen

Organisatorisches (2)

Materialien:

☞ **http://knopper.net/bw/gdi/**

Kursziel

- **Informatik**, v.a. Softwaretechnik, **definieren und erklären** können,
- grundsätzlichen **Aufbau von Computerprogrammen** in Theorie und Praxis kennen und verstehen, **Syntax** und **Semantik** sog. **formaler Sprachen**,
- grundlegende Konzepte der Darstellung und Anwendung von **Daten** und **Algorithmen** nachvollziehen,
- eine spezielle höhere Programmiersprache - **JAVA** - im Detail lernen (später auch für Softwaretechnik und Software-Engineering benötigt!),
- kleine und **genau definierte Aufgabenstellungen** mit Hilfe **in JAVA umgesetzter Algorithmen** lösen.

Zur Benutzung der Folien

- Foliensätze werden nach Bedarf erstellt, und können sich bis zum Ende der Veranstaltung noch ändern. Daher bitte Vorsicht beim Ausdrucken.
- Verweise auf Handouts oder sinnvolle ☞ Sekundärliteratur sind entsprechend gekennzeichnet und i.d.R. direkt anklickbar.
- Prüfungsrelevant sind grundsätzlich alle in der Vorlesung behandelten Themen.
- Die Vorlesungsinhalte sind größtenteils mit denen der Vorlesungen von Prof. Schiefer identisch, einige Folien sind noch im alten Foliensatz (online) erhalten und werden hier nur verlinkt. Viele Themen werden auch nach dem Stand der Technik und aktualisiert, umgestellt, entfernt oder kommen neu hinzu (Vorschläge/Wünsche?).

Ein gut gemeinter Rat zur Prüfungsvorbereitung

- **Informatik** ist - wie Mathematik - ein **wissenschaftliches Fach**, in dem Begriffe **nicht einfach auswendig gelernt** sondern **verstanden werden müssen**, ansonsten verliert man schnell die Übersicht.
- Verfahren nicht „wörtlich seitenweise auswendig lernen“ , sondern **anwenden können**!
- Informatik besteht aus vielen Komponenten ☞ teilweise sehr komplexe Inhalte ☞ konsequente Zeiteinteilung für das Lernen notwendig. Überlasten Sie Ihr ☞ Gehirn nicht, es funktioniert anders als eine „Festplatte“. Erst ein paar Tage vor der Prüfung mit dem Lernen/Aufarbeiten anzufangen, hat erfahrungsgemäß überhaupt keine Aussicht auf Erfolg, obwohl Sie in der Prüfung auch Unterlagen, Bücher, alte Klausurlösungen etc. verwenden dürfen!
- Versuchen Sie, schon während der Übungen alle behandelten Themen nachzuvollziehen, und bitte **stellen Sie Fragen sofort**, wenn etwas unklar ist!*)

*) Mut zur Lücke“ ist kontraproduktiv, weil fast alles aufeinander aufbaut

Heute: ...

- Einführung GDI Grundlagen
- 1. Übung (bis nächste Woche vorzubereiten): ☞**http://knopper.net/bw/gdi/uebungen/**

Einführung

Einige der folgenden Inhalte wurden dem Foliensatz der Vorlesung 2014 von Herrn Prof. Dr. Schiefer entnommen, und geringfügig modifiziert. Die Abbildungen entstammen, wenn nicht anders angegeben, Wikipedia und stehen unter Creative Commons Lizenz.

Rechnergrundlagen Übersicht

- Informatik-Begriff (9)
- Information und Daten (47)
- Darstellung von Zahlen (62)
- Darstellung von Texten und Multimedia (62)
- Computer-Hardware (23)
- Betriebssystem (33)

Informatik und Algorithmen

- Womit beschäftigt sich die Informatik (engl. „Computer Science“)?
  - Informatik ist die Wissenschaft von der systematischen Verarbeitung von Informationen, besonders der automatischen Verarbeitung mit Hilfe von Rechnern.
- Was ist ein Algorithmus, was sind Daten?
  - Die systematische Verarbeitung der Information wird durch den Begriff Algorithmus präzisiert, Information durch den Begriff Daten.
  - Ein **Algorithmus** ist eine eindeutige **Beschreibung** eines in **mehreren Schritten** durchgeführten (Bearbeitungs-) **Vorgangs**.

Zentrale Grundbegriffe

- In der Informatik liegt der Schwerpunkt auf der Ausführbarkeit durch (abstrakte) Maschinen, die auch als Prozessoren, Rechner oder Computer bezeichnet werden:
- Ein Prozessor führt einen Prozess (Arbeitsvorgang) auf Basis einer eindeutig interpretierbaren Beschreibung (dem Algorithmus) aus.
- Zur Umsetzung muss der Algorithmus in eine Programmiersprache übersetzt bzw. im Detail **implementiert** werden.

Kann ein Rechner mehr als ein Mensch?

- Alles, was ein Computersystem kann, kann ein Mensch im Prinzip auch!
- Ein Computersystem hat jedoch 3 wesentliche Vorteile:
  - Hohe Speicherungsfähigkeit
  - Hohe Geschwindigkeit
    - Mensch: In 1 Sekunde: 2 Zahlen addieren
      – In 1 Jahr: 32 Mio. Zahlen addiert
    - Computer: In 1 Sekunde: 100 Mio. Zahlen addieren
      – Für 32 Mio. Zahlen: 0,3 Sekunden
  - Hohe Zuverlässigkeit
    - Ob alle 32 Mio. Additionen, die der Vergleichsmensch vorgenommen hat, wohl alle korrekt sein werden?

Untergliederungen der Informatik (1)

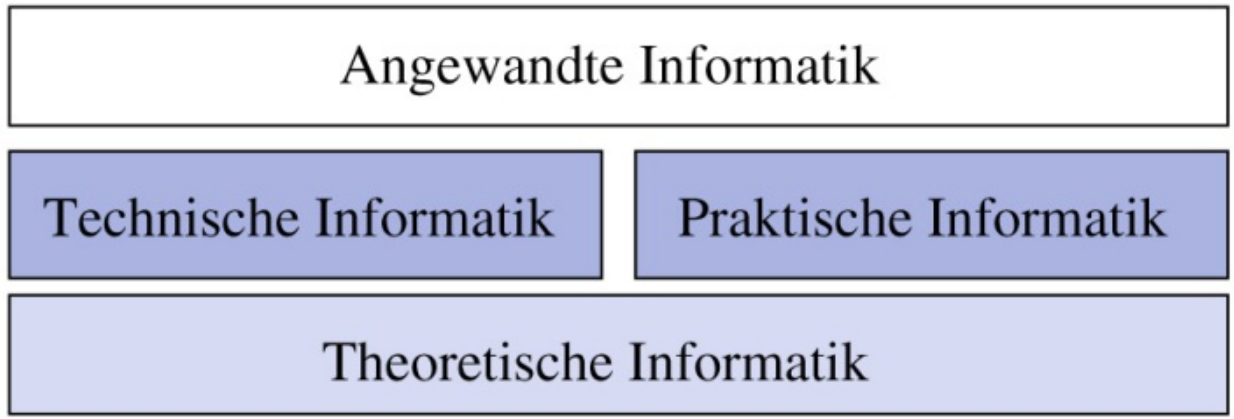


Abbildung 2.16: Die vier Hauptgebiete der Informatik

- Die Theoretische Informatik beschäftigt sich mit den abstrakten mathematischen Grundlagen aller Teilgebiete der Informatik. Sie stellt also einen Grenzbereich der Informatik zur Mathematik dar.
- Die Angewandte Informatik untersucht den Einsatz von Rechnersystemen in den verschiedenen Bereichen. Auf Grund der rasanten Technologie-Fortschritte eröffnen sich immer mehr Einsatzfelder für diese Systeme.

Untergliederungen der Informatik (2)

- Die Praktische Informatik beschäftigt sich mit den Programmen, die einen Rechner steuern, der Software. Die Aufgabe der Praktischen Informatik ist es, die Brücke zwischen der Hardware und der Anwendungssoftware zu schlagen.
- Die Technische Informatik beschäftigt sich mit der Konstruktion von Rechnern, Spreicherchips, schnellen Prozessoren oder Parallelprozessoren, aber auch mit dem Aufbau von Peripheriegeräten wie Festplatten, Displays oder Netzwerk-Komponenten. Die Technische Informatik ist der Grenzbereich zur Elektrotechnik und Physik.

Die Grenzen sind fließend!

Entwicklung der Informatik 1

Die Informatik hat sich auf Grund einer Vielzahl unterschiedlicher Entwicklungen in verschiedenen Bereichen herausgebildet. Diese betreffen unter anderen:

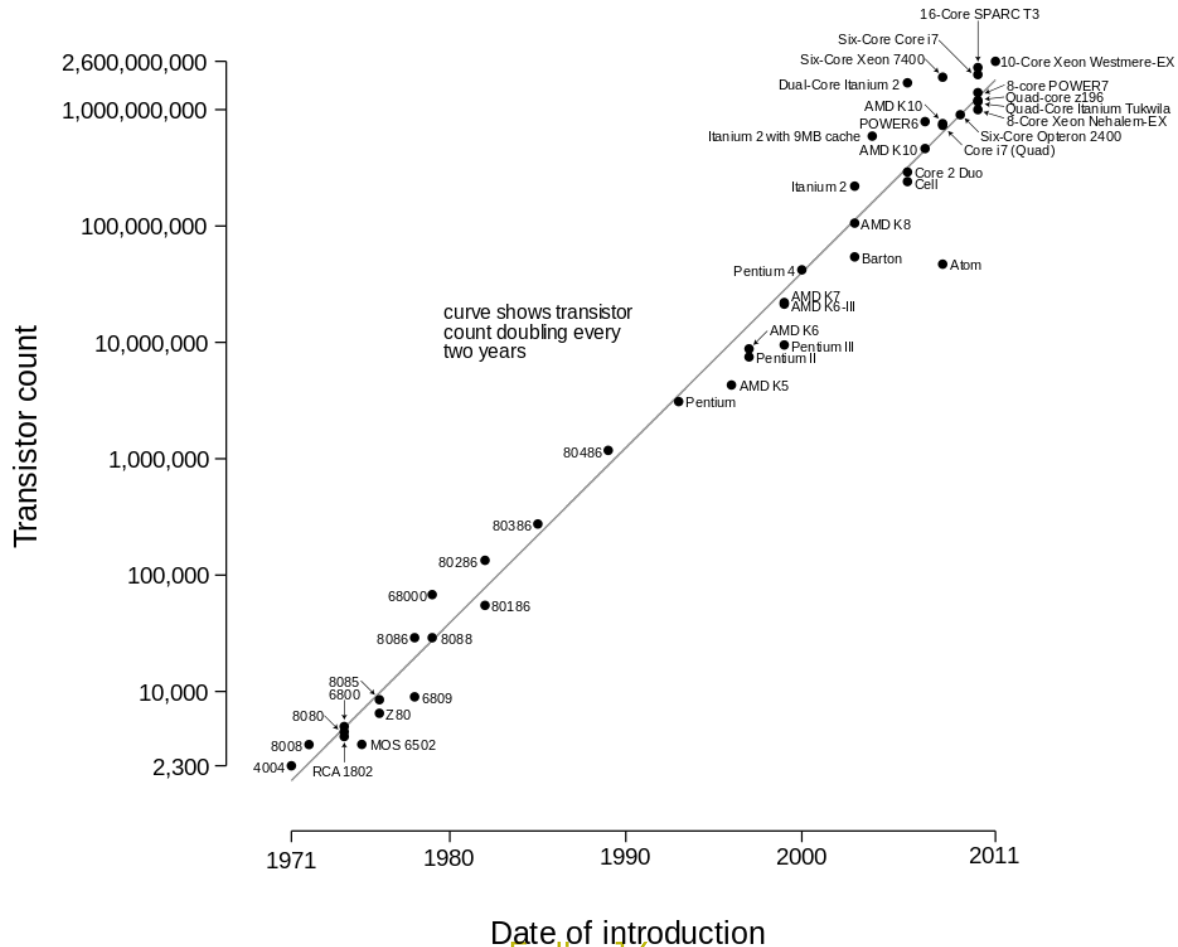
- die Mathematik: ca. 5000 v. Chr. erste Zahlensysteme, ca. 500 v. Chr. Grundbegriffe der Logik.
- den Algorithmus:
  ca. 825 n. Chr. erste algorithmische Verfahren.
  1. Beschreibung durch den arabischen Schriftsteller Abu Dshafar Muhammed Ibn Musa al-Chwarizmi

Entwicklung der Informatik 2

- den Computer:
  1600 - 1700 Wilhelm Schickard, Blaise Pascal und Gottfried Leibnitz entwickeln unabhängig voneinander erste mechanische Rechenmaschinen
  1941 baut Konrad Zuse die Z3, einen programmierbaren Relais-Rechner
  1946 entwickelt er das Konzept der Speicherung des Programms im Datenspeicher, welches bis heute die Grundlage der Computer ist
  ab 1950 beginnt die industrielle Rechnerproduktion

Das Gesetz von Moore (Moore's Law)

Das „Gesetz von Moore" ist eigentlich eher eine Faustregel: **Alle 12 - 24 Monate verdoppelt sich die Rechenleistung zum gleichen Preis.**



Entwicklung der Programmiersprachen

Die heute existierenden Programmiersprachen bilden eine große Familie (☞ Liste von Programmersprachen auf Wikipedia).

Als erste Programmiererin und damit Wegbereiterin computerimplementierbarer Programmiersprachen (100 Jahre vor der Erfindung des Computers, wie wir ihn heute kennen), wird häufig ☞ Ada Lovelace zitiert, die Algorithmen für die (nie fertiggestellte) „Analytical Engine“ des Mathematikers Charles Babbage in „Maschinen-Code“ umsetzte.

Entwicklung der Netze

Neben der Entwicklung der Rechner und Programmiersprachen haben sich seit 1969 die Rechnernetze im Weitverkehrsbereich ebenso rasch entwickelt und spielen heute eine zentrale Rolle für die Anwendung der neuen Technik:

- 1969 beginnt die Entwicklung der Weitverkehrsnetze mit dem ARPANET, das 4 Hochschulen in den USA vernetzte.
- 1972 ist das ARPANET auf 40 vernetzte US-Hochschulen angewachsen.
- 1982 wird das Netzwerkprotokoll TCP/IP (Transmission Control Protocol / Internet Protocol) eingeführt.
- 1990 hat sich das Internet durch die Einführung der Dienste Gopher und WWW zu einem einfach zu bedienenden Infosystem entwickelt.
- 1995 wurde es auch für kommerzielle Anwendungen geöffnet. Die Anzahl der Benutzer steigt seitdem (v.a. in den Industrienationen) exponentiell.

All-IP-Netze übertragen heute Telefonate, Daten und Multimedia kom-

TCP/IP

Der ☞ TCP/IP Protokollstandard wurde und wird durch die ☞ RFCs der ☞ Internet Engineering Task Force definiert. Alle Betriebssyteme bzw. Systemprogramme, die am Netzwerk teilnehmen, unterstützen diesen Standard.

- Alle am Internet teilnehmenden Rechner besitzen eine weltweit eindeutige **IP-Adresse**:
  Beispiel **IPv4**: 81.169.229.18 (4 8-bit Zahlen),
  Beispiel **IPv6**: 2001:0db8:85a3:08d3:1319:8a2e:0370:7344
- IPv4 unterstützt 4,3 Milliarden Adressen, IPv6 340 Sextillionen ($3{,}4 \cdot 10^{38}$).
- Lokale und Regionale Netze werden durch ☞ **Router** verbunden bzw. mit weiteren Netzen zusammengeschlossen.

Shared Media

Der theoretisch mögliche Datendurchsatz eines Netzwerkes hängt nicht nur von der Geschwindigkeit der Netzwerkanschlüsse ab.

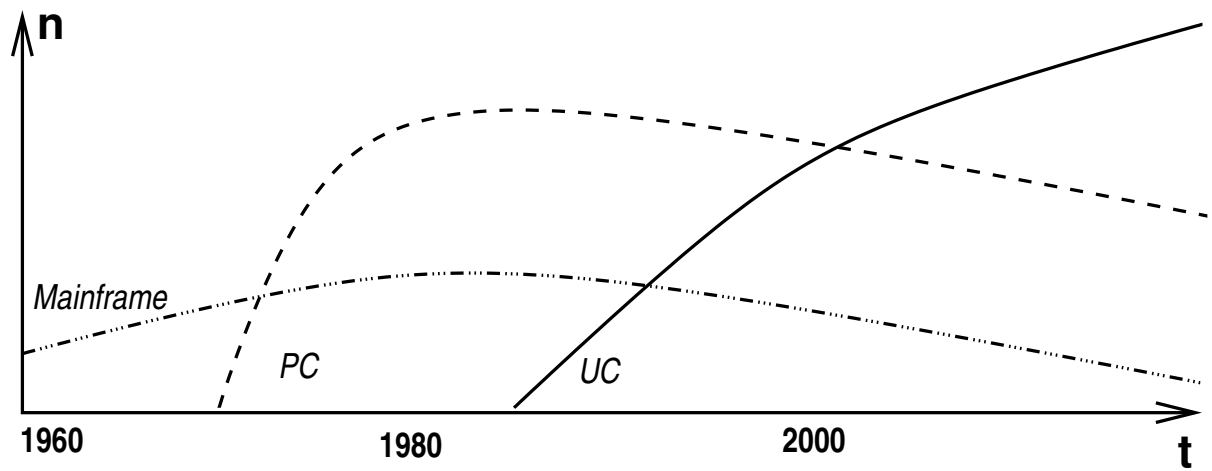
- Bei einer sternförmigen Vernetzung - jeder Computer hat eine direkte Verbindung zu jedem anderen - wäre ein unterbrechungsfreier Maximaldurchsatz möglich.
- Üblich ist **Shared Media** - mehrere Computer teilen sich eine „Leitung“ oder Funkstrecke. Es kommt zwangsläufig zu ☞ **Datenkollision**, die das ☞ **Transportprotokoll TCP** aber erkennt und durch Warten auf einen freien Zeitslot und Neuversenden der Datenpakete „repariert“.
- Treten **zu viele Kollosionen** auf, können die Datenverluste nicht mehr ausgeglichen werden, und die Verbindung bricht zusammen (TCP-Timeout max. 3 Minuten). ☞ **(ICMP) Ping** liefert **„Package Loss“** $>$ **50%**.

Quantitative Sichtweise - Zeitalter (1)

ca. 1938 (Z1)	Erste experimentelle, rein numerisch arbeitende Maschinen mit viel Mechanik, Relays, Röhren, ersten Transistoren. Programme bestehen eher aus „Hardware“ als aus leicht änderbarer „Software“
1960-1980	Mainframe - Ein Computer, viele Computernutzer
1980-2000	Personal Computer - Ein Computer pro Nutzer

Quantitative Sichtweise - Zeitalter (2)

2000- ☞ Ubiquitous Computing - Viele, auf bestimmte Aufgaben spezialisierte Computer pro Person



Hardware

Bei Computersystemen werden häufig folgende Varianten unterschieden:

- Personal Computer (PCs)
- Workstations (ähnlich PC, leistungsfähiger und robuster)
- Mainframes (Zentralrechner)
- Super-Computer (schnellste, aber auf bestimmte parallelisierbare Aufgaben spezialisierte Rechner der Welt)

... und viele weitere Bezeichnungen für Varianten.

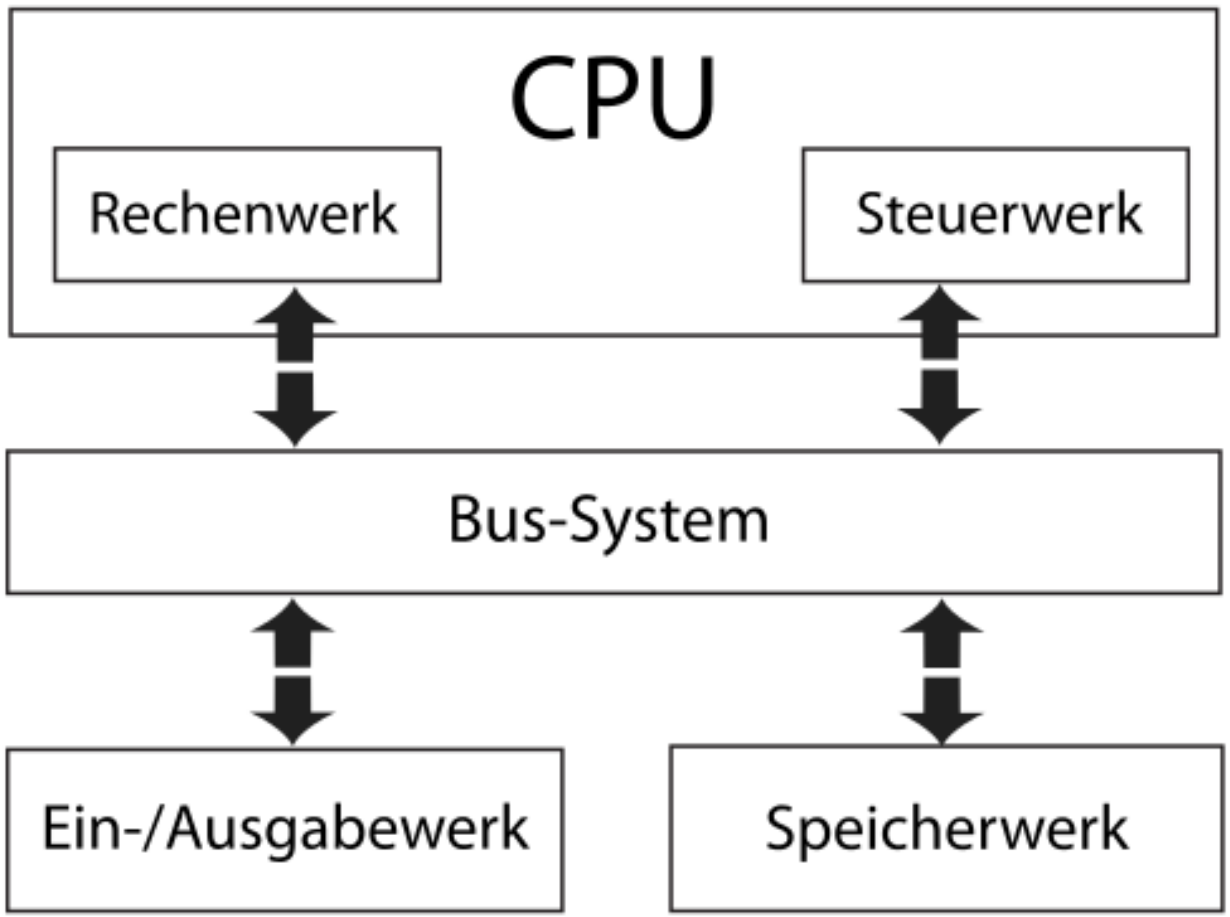
Von-Neumann-Architektur (1)

- Die von-Neumann-Architektur ist ein Referenzmodell für Computer:
  - ein gemeinsamer Speicher für Befehle und Daten
  - Bus-Systeme zur Verbindung aller Komponenten
- Geschichte / Bedeutung
  - Diese Architektur wurde bereits 1945 von John von Neumann (1903-1957) veröffentlicht.
  - Mit dem Ansatz lassen sich Programme für alle prinzipiell lösbaren Probleme formulieren – vorherige Ansätze basierten auf fest verdrahteter Logik
  - Alles was von einer ☞ Turingmaschine berechnet werden kann, kann auch mit diesem Rechner grundsätzlich gelöst werden.

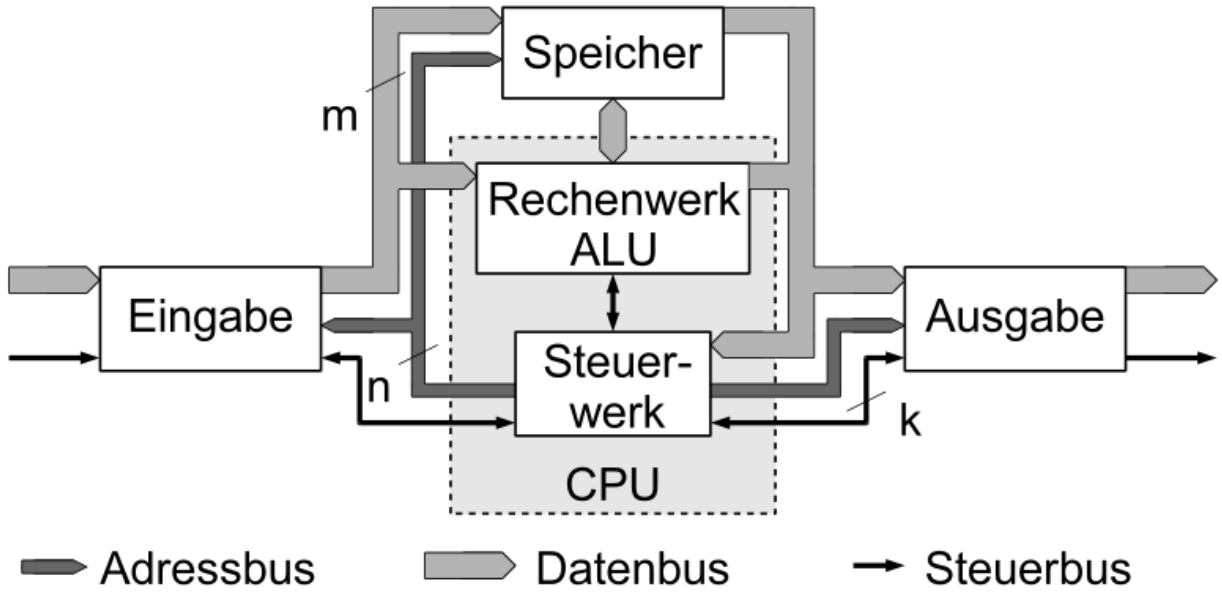
Von-Neumann-Architektur (2)

- Ein von-Neumann-Rechner enthält folgende Komponenten:
  - Rechenwerk: führt Rechenoperationen und logische Verknüpfungen durch. Auch Prozessor, Zentraleinheit oder ALU (Arithmetic Logical Unit) genannt.
  - Steuerwerk: interpretiert die Anweisungen eines Programms und steuert die Befehlsabfolge. Auch Control Unit oder Leitwerk genannt.
  - Speicherwerk (auch Memory genannt): speichert sowohl Befehle/Programme als auch Daten, welche für das Rechenwerk zugänglich sind.
  - Ein-/Ausgabewerk: steuert die Ein- und Ausgabe von Daten zum Anwender (Tastatur, Bildschirm) oder zu anderen Systemen (Schnittstellen). Auch I/O-Unit genannt.

Komponenten: von-Neumann-Rechner (1)



Komponenten: von-Neumann-Rechner (2)



Beispiel (1)

```
Load <4> // Hole den Inhalt der Speicherzelle 4
         // ins Rechenwerk
```

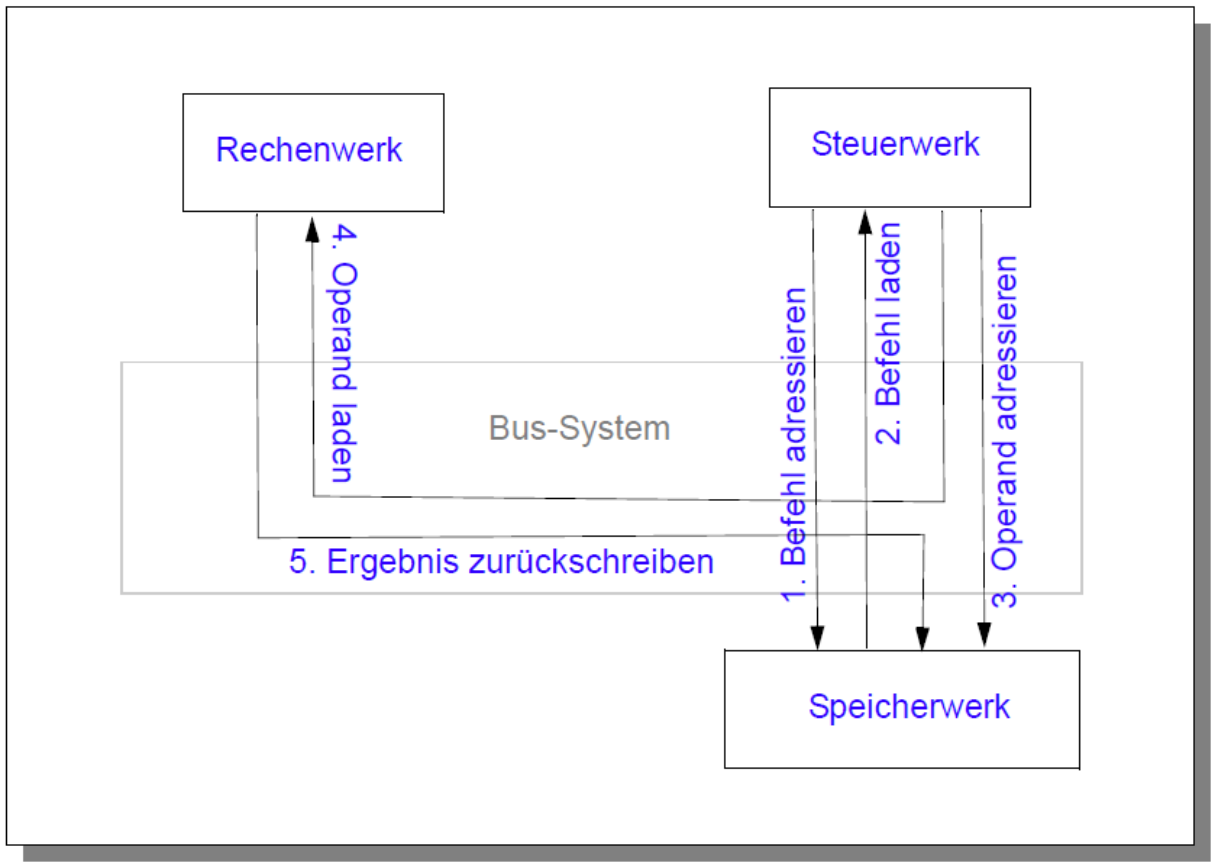
- Dazu werden vom Steuerwerk die folgenden Operationen durchgeführt:
  - Über die Adressleitung wird die Adresse 4 ausgegeben.
  - Über die Steuerleitung zum Speicher wird das Signal „Lesen“ gegeben.
  - Über die Steuerleitung zum Rechenwerk wird das Signal „Schreiben“ in das erste Operanden-Register gegeben.

Beispiel (2)

```
Add <5> // Addiere Inhalt der Speicherzelle 5
        // zum Inhalt d. Speicherzelle 4
```

- Dazu werden vom Steuerwerk die folgenden Operationen durchgeführt:
  - Über die Adressleitung wird die Adresse 5 ausgegeben.
  - Über die Steuerleitung zum Speicher wird das Signal „Lesen“ gegeben.
  - Über die Steuerleitung zum Rechenwerk wird das Signal „Schreiben“ in das zweite Operanden-Register und „Addition als Verknüpfung“ gegeben.

Verarbeitung eines Datenwortes



Weitere Komponenten (1)

- Der Arbeitsspeicher („Flüchtiges" RAM) eines Rechners verliert seinen Inhalt, wenn er nicht in regelmäßigen Abständen (z.B. alle 15 μs) „aufgefrischt" wird. Insbesondere gehen beim Abschalten alle Daten verloren.
- Zur langfristigen Speicherung werden daher andere Speichertechnologien benötigt. Ihre Kenngrößen sind Speicherkapazität und Zugriffszeit
  - Der wichtigste nichtflüchtige Massenspeicher ist die Festplatte. Ihre Kapazität hat inner- halb der letzten 15 Jahre um das Zehntausendfache zugenommen und nimmt weiter zu.
  - Prinzipiell sind Festplatten und Disketten sehr ähnlich aufgebaut.
  - Optische Platten wie CD und DVD schreiben Bits mithilfe von Löchern (pits), die beim Schreiben eingebrannt werden.

Weitere Komponenten (2)

- Daneben gibt es weitere Ein- u. Ausgabegeräte wie ISDN-Karte, Netzwerkkarte, externe Festplatten, Mikrofone, Lautsprecher, Drucker und vieles mehr. Einige davon sind bereits im Inneren eines Rechnergehäuses fest eingebaut.
- Der „von-Neumann-Flaschenhals“ in der von-Neumann-Architektur bezeichnet den Sachverhalt, dass das Bus-System zum Engpass zwischen dem Prozessor und dem Speicher wird. Da die CPU-Taktraten in jeder neuen Generation auch wesentlich schneller ansteigen als die der verwendeten Speicherbausteine, wird der Hauptspeicher ebenso zum Flaschenhals.

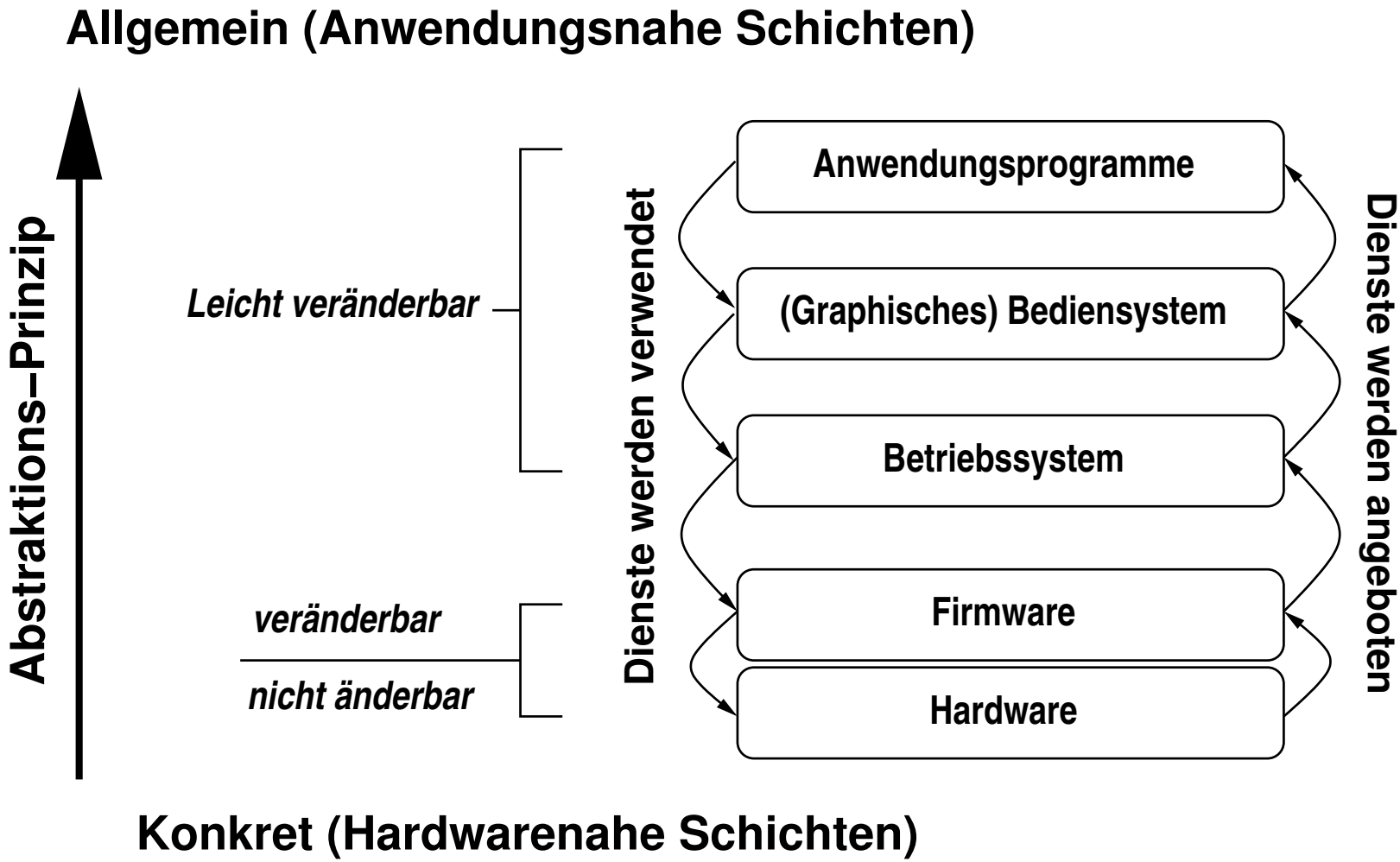
Betriebssystem (1)

- Bisher haben wir die Hardware und die Möglichkeiten der Datenrepräsentation diskutiert. Ein solcher „blanker" Rechner bzw. CPU kann nicht viel mehr als
  - Speicherinhalte in Register laden, Registerinhalte in Speicher ablegen,
  - Registerinhalte logisch oder arithmetisch verknüpfen,
  - mit Ein- oder Ausgabebefehlen Register in Peripheriegeräten lesen und schreiben.
- Um einen Rechner auf dieser sehr niedrigen Hardware-Ebene bedienen zu können, muss man alle technischen Details kennen.

Betriebssystem (2)

- Auf Benutzer/Anwenderebene will man eher folgende Dinge erledigen:
  - Briefe editieren und drucken,
  - E-Mail bearbeiten und versenden,
  - Fotos und Grafiken bearbeiten,
  - Simulationen ausführen u.v.m.
- Die Lücke zwischen dem (intuitiv handelnden) Anwender und den Fähigkeiten eines Rechners wird mit Hilfe von Zwischenschichten geschlossen, die über geeignete Schnittstellen in einem Schichtenmodell miteinander kommunizieren.

Hierarchisches Schichtenmodell



Aufgaben des Betriebssystems

- Ein Rechner mit seinen Peripheriegeräten stellt Betriebsmittel (Ressourcen) zur Verfügung, auf die Anwender-Programme zugreifen können:
  - CPU (Rechenzeit),
  - Hauptspeicher,
  - Plattenspeicherplatz,
  - externe Geräte wie Drucker, Modem oder Scanner.
- Zur Verwaltung dieser Betriebsmittel müssen viele Benutzerprogramme gleichzeitig auf diese Ressourcen zugreifen. Ein Betriebssystem muss daher die folgenden zentralen Aufgaben lösen:
  - Prozesse verwalten,
  - Speicher verwalten und
  - Dateien verwalten.

Prozessverwaltung (1)

- Zur Lösung einer Aufgabe müssen Programme auf einem Rechner ausgeführt werden.
  - Solche (dynamischen) Codeausführungen nennt man **Prozesse**.
- Häufig führt schon der Aufruf eines Programms zu vielen gleichzeitig und unabhängig voneinander laufenden (Teil-) Prozessen.
- Ein Prozess ist also ein eigenständiges Stück Programmcode mit eigenem Speicherbereich, der vor dem Zugriff durch andere parallel laufende Prozesse geschützt werden muss.

Prozessverwaltung (2)

- In der Regel laufen auf einem Rechner (eine oder mehrere CPUs) viele Prozesse gleichzeitig (Time Sharing).
  - Das Betriebssystem muss also alle nebenläufigen Prozesse möglichst fair verwalten.
  - Ebenso muss die Kommunikation zwischen den Prozessen realisiert werden und zwar so, dass sich die Prozesse nicht gegenseitig beinträchtigen oder sogar zerstören.
- Vorsicht: Zu viele parallel laufende Prozesse können das System verlangsamen (exzessives Task-Switching)!

Speicherverwaltung (1)

- In Analogie zu den Prozessen muss auch der Hauptspeicher verwaltet werden, in dem die Daten der vielen Prozesse gespeichert werden:
  - Neuen Prozessen muss freier Hauptspeicher zugewiesen werden und
  - der Speicher terminierter Prozesse muss wiederverwendet werden.
  - Die Speicherbereiche verschiedener Prozesse müssen vor gegenseitigen Zugriff geschützt werden.
  - Also: Betriebssystem-Aufgabe

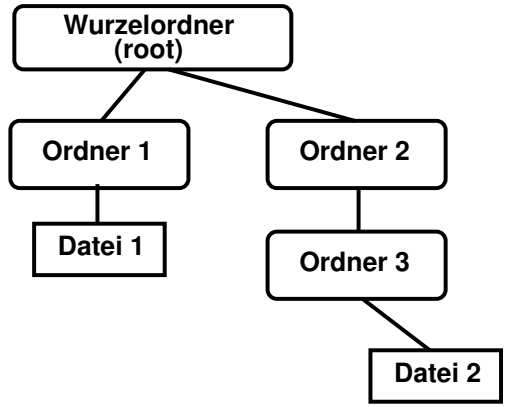
Speicherverwaltung (2)

- I.d.R. ist der Bedarf an Arbeitsspeicher größer ist als der verfügbare physische Speicher.
  - Dabei geht man von der Tatsache aus, dass zu einem beliebigen Zeitpunkt nur auf wenige Speicherplätze tatsächlich zugegriffen wird, während die anderen nur für einen späteren Zugriff (der u.U. nie erfolgt) bereitstehen.
  - Die Grundidee ist also eine Erweiterung der Speicherkapazität unter Zuhilfenahme externer Massenspeicher, von denen bei Bedarf die benötigten Informationen in den Arbeitsspeicher geladen werden.

Dateiverwaltung (1)

- Die Dateiverwaltung übernimmt die Aufgabe, Dateien auf die konkreten Gegebenheiten der Speichermedien abzubilden:
  - In welche Sektoren bzw. auf welche Spuren und Köpfe wird eine gerade geschriebene Text-Datei auf die Platte geschrieben, oder
  - wo ist die Version des Textes, der gestern gespeichert wurde?
- In diesem Sinne stellt das Betriebssystem das Konzept der Datei als Behälter für Daten aller Art zur Verfügung.
- Moderne Dateisysteme sind hierarchisch aufgebaut

Verzeichnisstrukturen



- Mehrere Dateien können zu einem Ordner zusammengefasst werden
  - Übliche Benennung: Verzeichnis (engl. directory)
  - Da Ordner sowohl normale Dateien als auch andere Ordner enthalten können, entsteht eine baumähnliche Struktur mit einem Wurzelordner (engl. root) an der Spitze.
- Jede Datei erhält einen Namen, unter dem sie gespeichert und wiedergefunden werden kann.
- Zusätzlich werden Erweiterungen verwendet, welche die Dateiinhalte spezifizieren, und Attribute.

DOS und Windows

Frühe Betriebssysteme für PCs waren in erster Linie Dateiverwaltungssysteme.

- Wichtigster Vertreter war DOS (Disk Operating System):
  - In diesem Fall kann immer nur ein Programm nach dem anderen ausgeführt werden.
  - Die Schnittstelle zum Benutzer (Bediensystem) ist die Kommandozeile: Der Benutzer tippt ein Kommando ein, das vom Betriebssystem sofort ausgeführt wird und zwar über den sog. Kommandointerpreter (shell).
    Um z.B. die Namen aller Dateien im aktuellen Verzeichnis anzuzeigen, gibt man **`dir`** ein, oder zur Umbenennung einer Datei **`ren alt.doc neu.doc`**.
    Eine DOS-ähnliche Kommandozeile kann in Windows immer noch über **`cmd.exe`** aufgerufen werden, mit Cygwin eine Unix-Shell-ähnliche.
- Erweiterungen (Prozess- und Speicherverwaltung, immer ähnlicher zu Unix): Windows 3.1, Windows 95, 98, ME, NT, 2000, XP, Windows 7, 10

Linux

- Das Betriebssystem ☞ Linux ist an ☞ UNIX angelehnt und wurde von dem finnischen Studenten Linus Torvalds entworfen und wird seitdem von tausenden Programmierern weltweit weiter entwickelt.
  - Der Quellcode ist frei zugänglich.
  - Es gilt als effizienter, schneller und robuster als Windows.
  - Heute ist es genau wie Windows sehr einfach zu bedienen. Der Benutzer kommuniziert über eine GUI (Graphical User Interface) mit dem System.
- Linux-Distributionen (Zusammenstellung von Betriebssystem und Anwendersoftware):
  - Ubuntu, Debian, SUSE, Fedora, CentOS, Redhat, ☞ Knoppix ...

Schnittstellen und Treiber (1)

- Damit eine CPU mit den Endgeräten (z.B. Laufwerk) verschiedener Hersteller zusammenarbeiten kann, muss man sich vorher auf eine gemeinsame Schnittstelle verständigt haben.
- Eine Schnittstelle ist eine Konvention, die eine Verbindung verschiedener Bauteile festlegt.
  - Man kann sich diesen Sachverhalt am Beispiel der elektrischen Steckdose verdeutlichen.
- Die Schnittstellen in der Informatik definieren nicht nur die räumlichen Ausmaße, sie können auch die Reihenfolge und Konvention des Signal- und Datenaustausches festlegen.

Schnittstellen und Treiber (2)

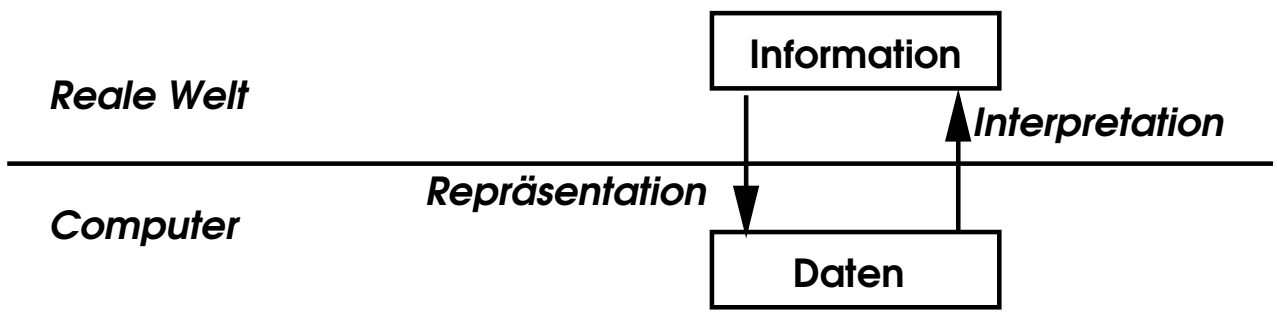
- Treiber unterstützen die Schnittstellen-Problematik auf der Komponenten- Seite:
  - Treiber sind allgemeine Übersetzungsprogramme zur Ansteuerung einer Software- oder Hardware-Komponente.
  - Treiber ermöglichen einem Anwendungsprogramm die Benutzung einer Hardware-Komponente, ohne den detaillierten Aufbau zu kennen.

Information und Daten (1)

- Was tut eigentlich ein Computer?
  - Soll „das Leben erleichtern“ durch eine maschinelle, automatische Informationsverarbeitung,
  - Konkret berechnen Computer Wettervorhersagen, steuern Raumfähren und unbemannte Fluggeräte, visualisieren medizinische Röntgenaufnahmen, erfassen und verarbeiten ständig sensorische Daten (z.B. Wearable Computing, Ubiquitous Computing, ...)

Information und Daten (2)

- Um die Aufgabe zu erfüllen, muss die Eingangsinformation als Datum (Singular von Daten) repräsentiert werden und die Ausgangsdaten wieder als Information interpretiert werden.
- Ein Computer kann auf Grund seines Aufbaus nie Information im eigentlichen Sinne interpretieren und direkt verarbeiten („verstehen"), sondern nur mit der in den Daten repräsentierten Information umgehen („rechnen").



Information und Daten (3)

- Die **Interpretation** der Daten kann auch als **Abstraktion** aufgefasst werden.
- Die Interpretation der Daten ist immer **abhängig vom Problem**. Das selbe digitale Codewort hat in einer anderen Umgebung (statt Wettervorhersage) eine ganz andere Bedeutung.

Bits

- Ein **Bit** ist die **kleinstmögliche Einheit der Information**.
- Ein Bit ist die Informationsmenge in einer Antwort auf eine Frage, die nur **zwei Möglichkeiten** zulässt.
- Die Information in einem Bit kann also durch nur zwei Symbole in Daten codiert werden.
- Man benutzt dazu meist die Zeichen `0` und `1`.

Codierung

- Eine **Codierung** ist deswegen nötig, weil die Information **technisch dargestellt** werden muss. Man bedient sich dabei etwa
  - elektrischer Ladungen (0 = ungeladen, 1 = geladen), oder
  - elektrischer Spannungen (0 = 0 Volt, 1 = 5 Volt) oder
  - Magnetisierungen (0 = unmagnetisiert, 1 = magnetisiert)
- Erst durch **Interpretation** werden **Daten** zu **Informationen**!

Bitfolgen (1)

- Mit einem Bit können $2^1 = 2$ verschiedene Möglichkeiten dargestellt werden. Um mehr Informationen zu codieren, müssen Bit**folgen** verwendet werden.
- Dazu werden in einem geordneten n-Tupel einfach n-viele Bits hintereinander gehängt.
- Mit n Bits können 2^n Werte dargestellt werden.

Bitfolgen (2)

- Auf Maschinenebene werden Bitfolgen verwendet um Daten und Befehle zu codieren.
- Die Daten stellen natürliche oder reelle Zahlen dar oder auch Felder von solchen Zahlen.
- Die (Maschinen-) Befehle führen Operationen auf den Daten aus. Dabei werden beispielsweise zwei Zahlen addiert oder multipliziert und als neues Datum ausgegeben.

Bytes und Worte (1)

- Innerhalb eines Rechners werden Bits in Gruppen verarbeitet, jeweils als Vielfaches von **8 Bit**: also 8 Bit, 16 Bit, 32 Bit oder 64 Bit.
  - Ein **Byte** ist eine Bitfolge der Länge **8**.
  - **Zwei Bytes** bilden ein **Wort**,
  - **4 Bytes** bilden ein **Doppelwort**.

Bytes und Worte (2)

In der Informatik werden i. d. R. 2er-Potenzen für die entsprechenden Vielfache verwendet. Daher ergeben sich abweichende Potenzen gegenüber dem Dezimalsystem, in der „traditionellen“ Schreibweise:

- 1 KiloByte = 1 KB = 2^{10} Byte = 1024 Bytes.
- 1 MegaByte = 1 MB = 2^{20} Byte = 1.048.576 Bytes.
- Diese Festlegungen entsprechen nicht dem üblichen dekadischen System und sind somit auch nicht streng einheitlich. Nach dem neueren ☞ IEC-Standard sollen hingegen die Dezimalpräfixe verwendet werden, also 1 KB = 1000 Bytes, während die zuvor genannten 2-er Potenzen als „Kibibytes“ und „Mebibytes“ benannt werden. Allerdings findet man diese Schreibweise noch eher selten.

Gängige Kapazitätsbezeichnungen

Kilobyte (kB)	2^{10} Byte =	1.024 Byte
Megabyte (MB)	2^{20} Byte =	1.048.576 Byte
Gigabyte (GB)	2^{30} Byte =	1.073.741.824 Byte
Terabyte (TB)	2^{40} Byte =	1.099.511.627.776 Byte
Petabyte (PB)	2^{50} Byte =	1.125.899.906.842.624 Byte
Exabyte (EB)	2^{60} Byte =	1.152.921.504.606.846.976 Byte
Zettabyte (ZB)	2^{70} Byte =	1.180.591.620.717.411.303.424 Byte
Yottabyte (YB)	2^{80} Byte =	1.208.925.819.614.629.174.706.176 Byte

Darstellung von Informationen

In Anlehnung an das Grundprinzip, wird in diesem Kapitel für verschiedene Arten von Informationen eine geeignete Repräsentation als Datum gefunden. Das betrifft:

- Wahrheitswerte
- Natürliche Zahlen
- Rationale und reelle Zahlen
- Text und
- andere Medien wie Audio oder Video-Daten.

Wahrheitswerte (1)

- Wahrheitswerte bzw. Kombinationen davon können als 0,1 Folgen dargestellt werden.
- Interpretation legt fest, dass z.B. **0** als **Falsch** und **1** als **Wahr** zu lesen ist
- Wahrheitswerte werden in der Mathematik auch als „boolsche Werte“ bezeichnet, benannt nach George Boole (1815-1864)

Boolesche Algebra

- eine spezielle algebraische Struktur mit logischen Operatoren: AND, OR, XOR, NOT
- mathematische Notation: $\wedge$, $\vee$, $\underline{\vee}$, $\neg$

Zwei- und einelementige boolesche Algebra

Elementare Rechenregeln

AND

$\wedge$	**0**	**1**
0	0	0
1	0	1

OR

$\vee$	**0**	**1**
0	0	1
1	1	1

XOR

$\veebar$	**0**	**1**
0	0	1
1	1	0

NOT

$\neg$	
0	1
1	0

Gesetze der booleschen Algebra

(Lassen sich ALLE aus den elementaren Rechenregeln ableiten!)

Kommutativgesetze	$a \wedge b = b \wedge a$	$a \vee b = b \vee a$
Assoziativgesetze	$(a \wedge b) \wedge c = a \wedge (b \wedge c)$	
	$(a \vee b) \vee c = a \vee (b \vee c)$	
Idempotenzgesetze	$a \wedge a = a$	$a \vee a = a$
Distributivgesetze	$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$	
	$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$	
Neutralitätsgesetze	$a \wedge 1 = a$	$a \vee 0 = a$
Extremalgesetze	$a \wedge 0 = 0$	$a \vee 1 = 1$
Doppelnegationsgesetz	$\neg(\neg a) = a$	
DeMorgansche Gesetze	$\neg(a \wedge b) = \neg a \vee \neg b$	
	$\neg(a \vee b) = \neg a \wedge \neg b$	
Komplementärgesetze	$a \wedge \neg a = 0$	$a \vee \neg a = 1$
Dualitätsgesetze	$\neg 0 = 1$	$\neg 1 = 0$
Absorptionsgesetze	$a \vee (a \wedge b) = a$	$a \wedge (a \vee b) = a$

To be done

Foliensatz gdi 02 Teil 1 / Schiefer: insert here (Ab Folie 2-13)...

Foliensatz gdi 03 / Schiefer: insert here...

Computer können nicht richtig rechnen?

Problem: Darstellung der 0,9 im Zehnersystem als IEEE-Fließkommazahl.

Reminder: Exponent gibt 2^x an, wobei x durch den „Bias"-Summanden auch negativ sein kann.

Wie stelle ich 0,9 also dar mit einer ganzzahligen Mantisse und einem ganzzahligen Exponent?

0,9 = 1,8 * 2^{-1} = 3.6 * 2^{-2} = ... = 15099494,4 * 2^{-24}

☞ Egal wie hoch der Exponent auch wird, die Nachkommastellen entfallen nie, daher ist bei ganzzahliger Mantisse der *genaue* Wert $0{,}9_{10}$ als Fließkomma-Dualzahl nicht abbildbar.

Programmierung - OpenSCAD

Kurzer Exkurs in eine sehr einfach gehaltene, imperative, prozedurale Programmiersprache: **OpenSCAD (Handout mit Übungen)** für 3D-Konstruktionsaufgaben.

Programmierung - JAVA (1)

☞ **Java** wurde 1994 von der Firma Sun Microsystems (Gruppe um James Gosling) entwickelt. Syntax und Semantik sind ähnlich der damals bereits bekannten Sprachen **C** und **C++**, Ziele der Neuentwicklung waren insbesondere:

- **Plattformunabhängigkeit:** Um Java-Programme ausführen zu können, genügt das **JRE** (Java Runtime Environment),
- Gute Unterstützung der Sprache für Netzwerkanwendungen und Systemfunktionen,
- Einfachere Anwendung und Wiederverwendbarkeit des Code im Vergleich mit C bzw. C++,
- Freie Verfügbarkeit der Entwicklungsumgebung und Dokumentation
  (☞ **`http://www.oracle.com/technetwork/java`**).

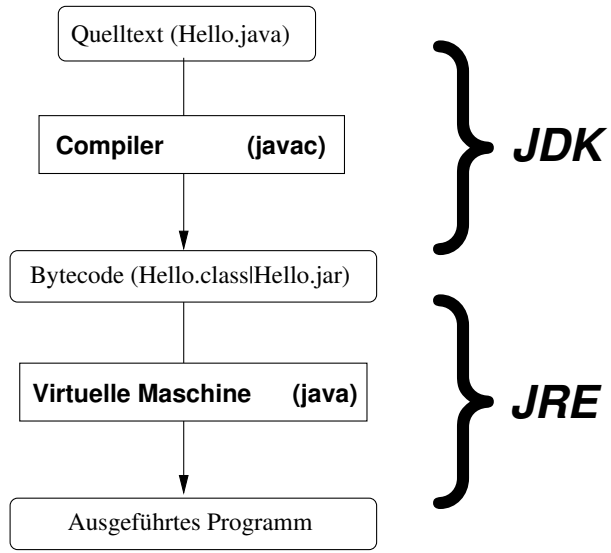
Programmierung - JAVA (2)

Um Java-Programme komfortabel entwickeln zu können, ist eine ☞ **IDE** (Integrated Development Environment) empfehlenswert, z.B. ☞ **Eclipse**.

Für die Programmiersprache Java steht eine große Menge von Literatur zum Lernen und als Nachschlagewerke zur Verfügung, einiges hiervon auch kostenlos, wie das Buch ☞ **„Java ist auch eine Insel“** (erschienen als Online-OpenBook im Rheinwerk-Verlag).

Ebenfalls empfehlenswert: Die „Original“ Java-Online Dokumentation im HTML-Format ☞ **`http://docs.oracle.com/javase`**

JAVA Installation



Um Java-Programme nicht nur ausführen, sondern auch entwickeln zu können, werden JDK (Java Development Kit) und JRE (Java Runtime Envoronment) in der gleichen Version benötigt!

Achtung: Seit 19. April 2019 gelten neue, proprietäre Lizenzbedingungen für das von Oracle herausgegebene JAVA! Eine Version, die Sie ohne Registrierung verwenden können, finden Sie hier: ☞ **`https://jdk.java.net/`**

Für Mac User leichter installierbar: ☞ **`https://adoptopenjdk.net/`**

JAVA erster Test (1)

1. Erstellen Sie mit einem beliebigen Texteditor (**`leafpad`** oder **`notepad++`** oder **`wordpad`**) den folgenden JAVA-Quelltext und speichern Sie ihn in der Datei **`Hello.java`** (genau so geschrieben!):

```
public class Hello {
 public static void main(String[] args){
  System.out.println("Hello, World!");
 }
}
```

   Bitte notieren Sie sich den Pfad, unter dem Sie die Datei gespeichert haben!

JAVA erster Test (2)

2. Öffnen Sie ein Terminalfenster bzw. die Eingabeaufforderung (Windows: **`cmd.exe`**, Linux/Mac: **`Terminal`** bzw. **`lxterminal`**). Wechseln Sie dort in das Verzeichnis, dass Sie zuvor notiert haben, in dem sich die gespeicherte Java-Quelltextdatei befindet.
   Windows/Linux/Mac: **`cd`** ***`Verzeichnispfad`***

3. Übersetzen Sie die Quelltextdatei mit dem Kommando **`javac Hello.java`**. Sollte dies zu einem **`"javac: Kommando nicht gefunden"`**-Fehler führen, dann ist JDK nicht richtig installiert! (Ggf. Installation wiederholen.)

JAVA erster Test (3)

4. Führen Sie das übersetzte Programm aus mit dem Kommando: **`java Hello`**. Beachten Sie, dass Sie diesmal die Dateiendung **`.class`** weglassen müssen, **`java`** sucht automatisch nach einer Datei mit der entsprechenden Endung, die so heißt, wie die angegebene Klasse.

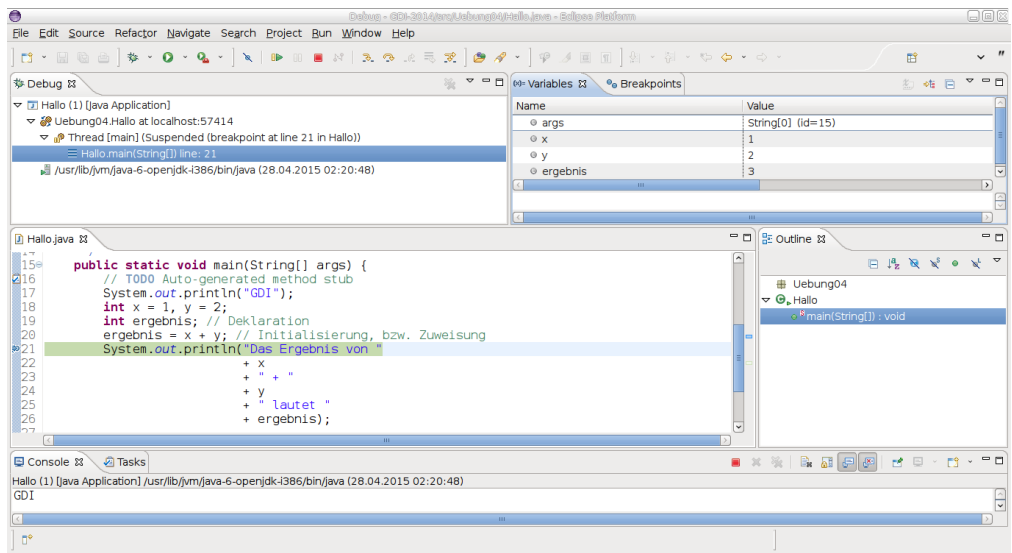
JAVA erster Test (4)

Beispiel:

Kommando	Was passiert?
`$ cd Java-Programme`	Wechsel ins Verzeichnis „Java-Prog...“
`$ leafpad Hello.java`	Bearbeiten der Datei „Hello.java“
`$ javac Hello.java`	Übersetzen des Java-Quelltextes
`$ java Hello`	VM führt **`Hello.class`** aus
`Hello, World!`	Programm gibt Text aus

Entwicklungsumgebung Eclipse (1)

- Integriert Quelltext-Editor, Java-Compiler und Java-Interpreter in einer Oberfläche,
- kann Programme auch schrittweise ausführen („tracen“, „debuggen“) und dabei Variablen „beobachten“,
- erleichtert die Fehlersuche durch Tooltips und Klassen-/Dokumentations-Browser,
- Organisiert Programm-Komponenten in Packages und trennt Quelltext und Compilat.



Entwicklungsumgebung Eclipse (2)

- Eclipse besitzt viel mehr Funktionen, als die meisten Einsteiger verwenden, und die überladenen Menüs machen die Suche nach einem bestimmten Feature oft komplizierter als notwendig.
- Das Verständnis für Zusammenhänge zwischen Quelltexten und Compilaten für verschiedene Projekte geht verloren,
- Manchmal unerwartete Effekte durch Starten des „falschen" Compilats (nicht der Quelltext im Editor), wenn mehrere Programme offen sind.

Fazit: JAVA erster Test

Was macht „**`java Hello`**“?

1. Die Java-VM sucht eine Datei mit dem Namen der angegebenen Hauptklasse (**`Hello`**) und Endung **`.class`**.
2. Innerhalb der Klasse **`Hello`** wird die Funktion **`public void main(String[] args)`** aufgerufen. Wenn weitere Parameter beim Aufruf angegeben sind, werden diese in das Übergabe-Parameter-Array **`String[] args`** übernommen (später!).
3. In **`main()`** wird (in diesem Beispiel) die Funktion **`println()`** aus der (Sub-)Package **`out`** aus der Klasse **`System`** aufgerufen, und gibt den übergebenen Text auf der Konsole aus.

Compilieren bei Java vs. Compilieren in C/C++

- Java: Beim Compilieren (**`javac`**) wird der „leicht von Menschen lesbare“ Quelltext in einen „leicht vom Rechner lesbaren“ Bytecode übersetzt, der von der VM (**`java`**) ausgeführt werden kann.
  C/C++: Beim Compilieren wird der „leicht von Menschen lesbare“ Quelltext in einen „direkt (nativ) ausführbaren Code“ übersetzt, und kann anschließend direkt gestartet werden.

- Konsequenz: Der compilierte Java-Bytecode läuft zwar überall, wenn eine Java-VM installiert ist, ist aber im Vergleich mit dem nativen Maschinencode-Compilat anderer Programmiersprachen langsamer, da er nur „interpretiert“ wird.

Imperative Programmiersprachen (1)

Eine **imperative** („befehlsorientierte") **Programmiersprache** besteht i.d.R. aus einer Folge von Anweisungen (**`Zuweisungen`** oder **Befehle**).

Die elementare Aktion **Zuweisung** besteht aus

1. einer Phase, in der ein Ausdruck **ausgewertet** wird, und
2. einer Phase, in der das **Ergebnis gespeichert** wird.

Imperative Programmiersprachen (2)

Zur Organisation des Programm-Ablaufs werden **Kontrollstrukturen** benötigt:

- Anweisungs-Blöcke (oder -gruppen),
- Bedingungen,
- Schleifen.

Wir werden im weiteren Verlauf diese Kontrollstrukturen im Detail kennen lernen.

Anweisungen in Java

- Jede Anweisung muss (!) mit einem Semikolon beendet werden. Ausnahmen sind Deklarationen von Klassen, die i.d.R. mit der schließenden geschweiften Klammer } enden.
- In einer Anweisung können u.U. mehrere Aktionen ausgeführt werden, z.B. eine Berechnung mit anschließender Zuweisung an eine Variable. Hierbei wird die Seite rechts vom Gleichheitszeichen zuerst ausgeführt!

```
int i = 1; // Deklaration mit Initialisierung
i = i + 1; // i speichert den Wert i+1 (d.h. 2)
```

Bezeichner (Namen) (1)

Die folgenden 47 Bezeichner sind (die kompletten!) Java-Schlüsselwörter, d.h. alle nicht in dieser Tabelle vorkommenden Wörter sind entweder selbstdefinierte Variablen und Funktionen, oder entstammen Java-Bibliotheken („`import ...`“).

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

*) *const* ist in Java ein reserviertes Wort ohne semantische Definition!

Bezeichner (Namen) (2)

Um neue Variablen, Funktionen und Klassen zu benennen, müssen die vom Programmierer gewählten Ausdrücke einer genau festgelegten Syntax (s.a. Chomsky-Hierarchie) entsprechen.

1. beliebig vielen Unicode-Buchstaben und Ziffern bestehen,
2. nur Unterstrich _ und Dollarzeichen **`$`** sind als Sonderzeichen erlaubt,
3. das erste Zeichen eines Namens darf keine Ziffer sein,
4. es wird zwischen Groß- und Kleinschreibung der Namen unterschieden („case sensitive"),
5. die Bezeichner dürfen nicht mit den Schlüsselwörtern der Sprache oder den Literalen **`true`** / **`false`** / **`null`** übereinstimmen.

Mit Hilfe der (selbsterklärenden) Java-Methoden
`Character.isJavaIdentifierStart (char)` und
`Character.isJavaIdentifierPart (char)`
kann ein Buchstabe auf syntaktische Konformität in einem Bezeichner geprüft werden.

Namenskonventionen

... erhöhen Lesbarkeit, Verständlichkeit und Wartbarkeit von Programmen. Folgende Richtlinien (kein Zwang!) haben sich in Java etabliert:

- Variablennamen beginnen mit einem Kleinbuchstaben (z.B. **`args`**). Sie sollten „selbsterklärend" gewählt werden!
- Namen von Konstanten (**`final`**) bestehen aus Großbuchstaben.
- Komponenten-Wörter werden durch _ getrennt.
- Methodennamen beginnen ebenfalls mit einem Kleinbuchstaben (z.B. **`main`** oder **`print`**) und werden i.d.R. nach Verben benannt.
- Klassen- und Interfacenamen beginnen mit einem Großbuchstaben und werden im weiteren Verlauf gemischt mit Groß- und Kleinbuchstaben geschrieben (z.B. **`HelloWorld`**).
- Paketnamen hingegen bestehen wieder nur aus Kleinbuchstaben.

Sonderzeichen Übersicht (1)

Fast alle Standard-Sonderzeichen haben eine spezielle Funktion, z.B. als Rechenoperator, Trenner, Zuweisungs- oder Vergleichsoperator, und erhalten in bestimmten Kombinationen auch neue Bedeutungen.

Zeichen	**Bedeutung**
=	Zuweisung
==	Vergleich
!	Logisches „Nicht“
~	Bitweises Invertieren
^	Bitweise XOR-Verknüpfung
|	Arithmetisches (bitweises) „Oder“
||	Logisches „Oder“
&	Arithmetisches (bitweises) „Und“
&&	Logisches „Und“
,	weitere Definition/Anweisung
;	Anweisungsende

Sonderzeichen Übersicht (2)

Zeichen	Bedeutung
`.`	Zugriff auf ...
`%`	MODULO (Teilungsrest)
`-`	Negativwert / Minus
`+`	Addition oder Aneinanderhängen
`*`	Multiplikation
`/`	Division
$>$	Größer (Vergleich)
$>>$	Bitshift nach rechts ($:2^x$)
$>=$	Größer oder gleich (Vergleich)
$<$	Kleiner (Vergleich)
$<<$	Bitshift nach links ($\cdot 2^x$)
$<=$	Kleiner oder gleich (Vergleich)
`(...)`	Funktionsparameter oder Berechnungsklammer
`{...}`	Implementation oder Anweisungsgruppe

„Interne Doku“ mit Kommentaren

Kommentare helfen, Quelltexte zu verstehen, indem der Autor hilfreiche Hinweise notiert oder Alternativteile des Quelltextes durch „Auskommentieren“ außer Funktion setzt.

Kommentar bis Zeilenende `//`:

```
i = 1 + 2;  // Einfache Addition
i = 1 - 2;  // Einfache Subtraktion
```

Kommentar mit Anfang und Ende `/* ... */`:

```
/*****************************
   entfernt 28.4.2015 -KK
   i = i + 1;
*****************************/

/* Kürzere Version: */ i++;
```

Variablen und Zuweisungen

- Eine **Variable** ist ein **Name für einen Speicherbereich für Datenwerte**.
- Mit einer Variablen können die folgende elementaren Operationen durchgeführt werden:
  - Variable lesen, also den unter dem jeweiligen Namen im Hauptspeicher gespeicherten Datenwert bestimmen.
  - Werte einer Variablen zuweisen, also den Speicherbereich der Variablen belegen. Falls dort bereits ein Datenwert abgelegt war, so wird dieser durch die neue Zuweisung überschrieben.
- Die Zuweisung erfolgt **von rechts nach links** durch den **=** Operator (einfaches Gleichheitszeichen), s.a. Folie 78.

Typisierung von Variablen

- **Java** zählt zu den **streng typisierten** Sprachen.
- Variablen können nur die zuvor bei der **Deklaration** festgelegten Datentypen aufnehmen.
- Der Datentyp bestimmt auch die **Operationen**, die mit der Variablen ausgeführt werden dürfen.
- Wie in vielen Programmiersprachen müssen auch in Java Variablen erst **deklariert** (vereinbart) werden, **bevor sie verwendet** werden dürfen.
- Bei der Deklaration muss jeder Variablen ein **Typ zugeordnet** werden.
- Bereits zum Compile-Zeitpunkt findet eine **Prüfung der Typverträglichkeit von Zuweisungen** statt.
- Der Compiler ermittelt bereits den Speicherbedarf der Variablen in Abhängigkeit vom Datentyps.

Deklaration und Gültigkeit

- Die **Position der Variablendeklaration** im Programm legt zugleich auch den **Gültigkeitsbereich der Variablen** fest.
- Eine Variable ist **nach Verlassen des Blocks, in dem sie vereinbart wurde, nicht mehr gültig**.
- Die **Deklaration (und optionale Initialisierung) einer Variablen** erfolgt in der Form:

  ```
  Datentyp Variablenname;
  ```

Beispiele:

```
int alter;
boolean nochAllesOk = true;
```

Basis-Datentypen

Typ	Bedeutung	Bits	Wertebereich
`boolean`	Wahrheitswert	-	`true` oder `false`
`char`	Zeichen	16	0 bis 65.535
`byte`	Sehr kurze Zahl	8	-128 ... 127
`short`	Kurze Ganzzahl	16	-32.768 ... 32.767
`int`	Ganzzahl (Integer)	32	-2.147.483.648...2.147.483.647
`long`	Lange Ganzzahl	64	-9.223.372.036.854.775.808 ... 9.223.372.036.854.775.807
`float`	Einfachgen. Fließk.	32	$1{,}4 \cdot 10^{-45}$ ... $3{,}4028 \cdot 10^{38}$
`double`	Doppeltg. Fließk.	64	$4{,}9 \cdot 10^{-324}$ ... $1{,}798 \cdot 10^{308}$

Noch einmal: Zuweisungen

Bei Ausdrücken der Form

```
a = b + c;
```

bezeichnet man die Seite **rechts** vom Gleichheitszeichen als **R-Value**, sie besitzt immer einen konkreten Wert (auch wenn hier Variablen stehen), der arithmethisch berechnet wird.
Die Seite **links** vom Gleichheitszeichen ist der **L-Value**, hier muss eine Variable angegeben sein, die den Wert aufnehmen / speichern kann.

Zuweisungskompatibilität

Faustregel: Die linke Seite der Zuweisung (L-Value) muss immer einen Datentyp besitzen, der gleich viele oder mehr Bits bzw. „Genauigkeit“ besitzt als der Datentyp auf der rechten Seite, sonst bricht der Compiliervorgang mit einem „possible loss of precision“ Fehler ab.

Zuweisungen in Richtung von rechts nach links in der folgenden Kette sind erlaubt und die Datentypen werden ggf. auch automatisch umgewandelt, umgekehrt nicht:

`double` $\leftarrow$ **`float`** $\leftarrow$ **`long`** $\leftarrow$ **`int`** $\leftarrow$ **`short`** $\leftarrow$ **`byte`**

`int` $\leftarrow$ **`char`**

Durch einen **cast** **`(datentyp)`** kann der Programmierer eine Umwandlung unter möglichem Verlust von Information erzwingen:

```
double d = 1.5;
int i = (int) d;
```

Modifizierende Operatoren

Syntax	Wert des Ausdrucks (R-Value)	Wirkung
`a++;`	`a` vor Erhöhung	`a` wird um 1 erhöht
`a--;`	`a` vor Erniedrigung	`a` wird um 1 kleiner
`++a;`	`a` nach Erhöhung	`a` wird um 1 erhöht
`--a;`	`a` nach Erniedrigung	`a` wird um 1 kleiner

Ausgabefunktion (Konsole)

Die Methode **`System.out.println( ... );`** wandelt automatisch sämtliche als Parameter übergebenen Werte und Variablen in Zeichenketten (**`String`**) um, und gibt diese anschließend auf der Konsole aus. Durch eine „Addition“ (**`+`**-Operator) können Zeichenketten aneinandergehängt werdfen (Vorsicht: Bei Zahlen bewirkt dies eine Addition, ggf. Klammern **`(...)`** setzen).

```
System.out.println("Das Ergebnis der Addition "
                   + a
                   + " + "
                   + b
                   + " lautet: " + (a+b) );
```

Eingabefunktion (Konsole) (1)

Das „Einlesen“ von Texten oder Zahlen ist in JAVA wesentlich komplizierter. Hier wird zunächst der Zugriff auf das „Eingabegerät“ mit Hilfe z.B. der „Scanner“-Klasse vorbereitet.

```
// Konsole zum Lesen von Eingabewerten vorbereiten
Scanner sc = new Scanner(System.in);
```

Das **`sc`**-Objekt kann nun mit Hilfe seiner eingebauten Funktionen Zahlen oder Texte lesen.

Eingabefunktion (Konsole) (2)

Beispiel Lesen einer Ganzzahl (mit Wandlung von Text nach Zahlenwert):

```
int zahl = sc.nextInt();
```

`zahl` enthält nun den eingegebenen Text, konvertiert in das **`int`**-Zahlenformat. Die Methode **`nextInt()`** ist in der Klasse **`Scanner`**, von der das Objekt **`sc`** mit **`new`** gebildet wurde, integriert.

Achtung: Je nach Spracheinstellung Ihres Betriebssytems wird in der Scanner-Klasse das Zeichen `,` in das bei **`float`** oder **`double`** vorgeschriebene Zeichen `.` für die Nachkommastellen konvertiert!

`Eingabe.java`: Klassenbibliothek zum Einlesen

In der für unsere Zwecke beim Arbeiten mit der Ein-/Ausgabekonsole recht nützlichen Datei **`Eingabe.java`** sind Funktionen integriert, die das Lesen von Zahlen vereinfachen, so dass Sie sich das o.g. Prozedere (obwohl Sie es kennen sollten) ersparen können.

Funktion	**Bedeutung**
`Eingabe.readInt()`	Liest eine **`int`**-Zahl
`Eingabe.readInt("text")`	Gibt den **`text`** aus, liest dann ein **`int`**
`Eingabe.readLong()`	Liest eine **`long`**-Zahl
`Eingabe.readLong("text")`	Gibt den **`text`** aus, liest dann ein **`long`**
...	

Achtung: Im Gegensazu zur **`Scanner`**-Klasse nimmt **`Eingabe`** keine automatische Konvertierung von `,` nach `.` vor.

`Eingabe.java`: Klassenbibliothek zum Einlesen

Sinngemäß können mit der gleichen Klasse auch **`float`**, **`double`** und **`String`** gelesen werden. Der **Rückgabewert** muss natürlich über eine Zuweisung jeweils an eine Variable übergeben werden, z.B.:

```
double d =
 Eingabe.readDouble("Fließkommazahl eingeben: ");
```

Besonderheit: Wird ein ungültiger Wert eingegeben (z.B. Buchstaben statt Zahlen), geben die Funktionen eine Fehlermeldung aus, und übergeben einen Null-Wert bei der Zuweisung. Anders als bei den in **`Scanner`** verwendeten Funktionen wird das Programm im Fehlerfall aber nicht beendet.

Bedingte Ausführung: `if`

```
if(Bedingung) Anweisung;

if(Bedingung) { Anweisung1; Anweisung1; ...; }
```

Eine Anweisung bzw. ein Block von mehreren Anweisungen wird nur dann ausgeführt, wenn die Bedungung **`true`** (WAHR) ist.

Bedingte Ausführung: `if` ... `else` ...

```
System.out.print(zahl + " ist ");
if( zahl%2 == 0 ) System.out.println("gerade");
else              System.out.println("ungerade");
```

Optional kann nach der auf die **`if()`**-Abfrage folgenden Anweisung bzw. nach dem geklammerten Anweisungsblock noch eine „ansonsten"-Verzweigung stehen. Die Anweisung bzw. der Anweisungsblock dahinter wird dann ausgeführt, wenn die Bedingung hinter **`if false`** (FALSCH) war.

Achtung: Jedes **`else`** bezieht sich, wenn nicht anders geklammert ist, auf das unmittelbar zuvor verwendete **`if()`**.

Der ternäre Operator...

... macht Programme kürzer.

```
if(bedingung) i = 1;
else i = 2;
```

kann mit dem ternären Operator abgekürzt werden als:

```
int i = bedingung ? 1 : 2;
```

`?` und `:` wirken hier zusammen und beherbergen DREI Argumente (daher der Name „ternär").

Mehrfach-Bedingungen mit `if()...else`

```
System.out.print("Der " + zahl + ". Wochentag ist ");
if(i==1) System.out.println("Montag");
else if(i==2) System.out.println("Dienstag");
else if(i==3) System.out.println("Mittwoch");
...
else System.out.println("ungültig");
```

Hierfür gibt es eine elegantere Möglichkeit...

Mehrfach-Entscheidung mit `switch()` (1)

```
System.out.print("Die Ampel ist ");
switch(zahl){
 case 0: System.out.println("grün"); break;
 case 1: System.out.println("gelb"); break;
 case 2: System.out.println("rot"); break;
 default: System.err.println("kaputt"); break;
}
```

Die in den Klammern hinter **`switch()`** stehende Variable wird in den **`case`**-Klauseln verglichen, und ggf. die Anweisungen dahinter ausgeführt.

Mehrfach-Entscheidung mit `switch()` (2)

Achtung: Fehlt das Schlüsselwort **`break;`**, so werden auch die folgenden **`case`**-Anweisungen ohne Überprüfung ausgeführt! Der Doppelpunkt kennzeichnet hier also eher eine *Einsprungadresse* als eine abgeschlossene Anweisungsfolge.

```
switch(taste){
 case 'r':
 case 'R':
  reboot(); break;
 case 's':
 case 'S':
  shutdown(); break;
 default:
  System.err.println("Bitte nur R oder S drücken.");
}
```

Schleifen: `while()`

```
while(Bedingung) Anweisung;
```

```
while(Bedingung) { Anweisung1; Anweisung2; ...;}
```

Die Anweisung(en) wird/werden ausgeführt, so lange die **`Bedingung`** **`true`** (WAHR) ist.

```
do { Anweisungen...; } while(Bedingung);
```

Die Anweisung(sfolge) wird mindestens einmal ausgeführt und danach, wenn die Bedingung wahr ist, wiederholt (bis die Bedingung falsch wird).

Schleifen: `for()`

```
for(Start; Bedingung; Aktualisierung) Anweisung;

for(Start; Bedingung; Aktualisierung) {
 Anweisung1;
 Anweisung2;
 ...;
}
```

Zu Beginn wird die **`Start`**-Anweisung *einmalig* ausgeführt, dies ist häufig die Deklaration und Initialisierung einer „Laufvariable", die nur innerhalb der Schleife gültig ist, z.B. **`int i = 0;`**.

Vor Ausführen der Anweisungsfolge wird die **`bedingung`** überprüft, und wenn diese **`true`** (WAHR) ist, kommt die Ausführung zustande.

Nach Abarbeiten des Anweisungsblocks wird die **`Aktualisierung`** ausgeführt, meist wird dabei die Laufvariable erhöht oder erniedrigt.

Schleifen: `for()` Beispiel

```
for(int i=0; i<10; i++) {
 System.out.println(i);
}
```

Es werden die Zahlen 0 .. 9 ausgegeben. Nach Ablauf der Schleife wäre die Laufvariable `i == 10`, wenn sie noch existieren würde.

Da hier nur eine Anweisung ausgeführt wird, sind die geschweiften Klammern `{ ... }` eigentlich unnötig.

Schleifen vorzeitig verlassen

Mit **`break`** kann eine Schleife sofort verlassen werden:

```
int summe = 0;
for(int i=0; i<100; i++){
 if(abbruchkriterium() == true)  break;
 summe += i;
}
```

Schleifen vorzeitig wiederholen

Mit **`continue`** wird der Rest der Anweisungen in der Schleife sofort wieder gestartet, dabei wird (bei der **`for()`**-Schleife) die Aktualisierungsanweisung ausgeführt und die Bedingung erneut überprüft.

```
int summe = 0;
for(int i=0; i<100; i++){
 if(diesmal_überspringen() == true)
  continue;
 summe += i;
}
```

„Verbotene“ Befehle: `goto`

Beispiel in der Programmiersprache **C**:

```
if(nenner == 0) goto fehler;
System.out.println(zaehler / nenner);
goto ende;
fehler:
 System.out.println("Durch 0 darf man nicht teilen!");
ende:
 System.out.println("Programmende.");
```

`goto Sprungmarke;` fährt mit der Programmausführung ab der genannten Marke fort. Dies gilt als schlechter Programmierstil, da hiermit eine konsistente Strukturierung in Anweisungsblöcke umgangen wird.

Java unterstützt derzeit die **`goto`**-Anweisung nicht, obwohl das Schlüsselwort reserviert ist (s. Tabelle auf Folie 79).

Aufzählungen (1)

- **Enum**s sind Datentypen, die als Werte nur selbstdefinierte Konstanten zulassen.
- Eine Variable eines Aufzählungstyps kann als Wert eine dieser Konstanten besitzen.
- Ein Aufzählungstyp trägt einen *Namen*. Bei der Definition des Typs werden die Aufzählungskonstanten in Form einer Liste mit geschweiften Klammern angegeben.

```
enum AmpelFarbe { ROT, GELB, GRUEN }
```

Aufzählungen (2)

```
public class EnumTest {
 enum Wochentag { MO, DI, MI, DO, FR }
 public static void main (String[] args) {
  Wochentag tag = Wochentag.MI;
  switch (tag) {
   case MO : System.out.println ("Ein Montag"); break;
   case DI : System.out.println ("Ein Dienstag"); break;
   case MI : System.out.println ("Ein Mittwoch"); break;
   case DO : System.out.println ("Ein Donnerstag"); break;
   case FR : System.out.println ("Ein Freitag"); break;
   default : System.out.println ("Anderer Tag"); break;
  } // switch
 } // main
} // class EnumTest
```

Strings

- Die Klasse String repräsentiert Zeichenketten.
- Obwohl es sich nicht um einen Basisdatentyp handelt, sind bestimmte Operatoren wie + hier zulässig.
- Die Klasse String bietet Funktionen, um komfortabel mit Zeichenketten zu arbeiten.
- Die Elemente (Zeichen, char) einer Zeichenkette sind Unicode-Zeichen (16bit = 2 Byte lang).

Strings

- Grundsätzlich sind Zeichenketten Konstanten („read-only“), die Methoden aus der String-Klasse liefern aber neue Objekte, die z.B. Zeichen in einem Eingabestring tauschen oder Strings aneinanderhängen.
- Eine String-Variable enthält nur die Adresse des Ortes im Speicher, an dem der Inhalt des Strings steht, was vor allem bei Vergleichsoperationen beachtet werden muss (gleicher Speicherinhalt vs. gleiche Adresse bzw. gleiches Objekt).

Strings

String-Objekte (unbekannter Länge) können auf verschiedene Arten erzeugt werden:

```
// Konstante
String nachname = "Hurtig";
// Kopierkonstruktor
String vorname = new String ("Harry");
```

Was passiert hier mit der ersten String-Konstanten nach der zweiten Zuweisung?

```
String nachname;
nachname = "Hurtig";
nachname = "Meier";
```

Strings

Eine Zeichenkette muss in Java in einer einzigen Zeile im Quelltext stehen, kann über mehrere Zeilen aber auch mit + verbunden werden. Der gleiche Operator wird generell dazu verwendet, zwei Strings zu einem zusammenzufügen.

```
String alphabet = "ABCDEFGHIJKLMNOPQRST"
                  + "UVWXYZ";
```

Besondere Strings:

`""`	Leerer String
`"\""`	String bestehend aus einem einzelnen Anführungsstrich
`"\n"`	Ein Zeilenumbruch

Die `String`-Klasse (1)

In der Java-Klasse **`String`** sind im Gegensatz zum aus C bekannten **`char *`** auch Methoden definiert, die es erlauben, mit dem aus der Arithmetik bekannten Operatorzeichen **+** Zeichenketten zusammenzuhängen, oder Umwandlungen zwischen Text und Zahlen vorzunehmen.

```
public class StringVerkett
{
  public static void main(String[] args)
  {
    int a = 5;
    double x = 3.14;

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

Die `String`-Klasse (2)

```
public class StringVergleich
 {
  public static void main(String[] args)
  {
    String a = new String("hallo");
    String b = new String("hallo");
    System.out.println("a == b liefert " + (a == b));
    System.out.println("a != b liefert " + (a != b));
  }
}
```

Die `String`-Klasse (3)

```
public class StringVergleich2
{
  public static void main(String[] args)
  {
    String a = new String("hallo");
    String b = new String("hallo");
    System.out.println("a.equals(b) liefert " +
                                        a.equals(b));
  }
}
```

Weitere Methoden der `String`-Klasse

Doku: ☞ JAVA 6 API

`boolean equalsIgnoreCase(String anotherString)` Vergleicht die beiden **`Strings`** unabhängig von Groß-/Kleinschreibung.

`String trim()` Entfernt Leerzeichen am Anfang und Ende.

`char charAt(int index)` Gibt das Zeichen an der Stelle **`index`** des **`String`** zurück.

`int indexOf(String str)` Gibt die Stelle des ersten Vorkommens von **`str`** in der Zeichenkette zurück.

`int length()` Gibt die Anzahl der im **`String`** enthaltenen Zeichen zurück. Achtung: Anders als bei Arrays ist dies eine Funktion!

`String replace (char old, char new)` Gibt einen neuen String zurück, bei dem alle Buchstaben **`old`** durch **`new`** ersetzt worden sind.

`String substring(int beginIndex, int endIndex)` Gibt den Teil des String zurück, der bei **`beginIndex`** beginnt und mit **`endIndex`** endet.

Von `String` nach `int`

Die Zeichenkette `"123"` ist etwas anderes als die Zahl `123`. Eine automatische Umwandlung von `String` nach `int` findet nicht statt.
In der JAVA-Bibliothek befinden sich z.B. in der Klasse `Integer` Hilfsfunktionen für die Umwandlung.

```
// Nicht möglich: String s = 123;
// Nicht möglich: int i = "123";

// OK:
String s = new Integer(123).toString();
int i = Integer.valueOf("123").intValue();
// Alternativ:
int i = Integer.parseInt("123");
```

Arrays - Datenfelder

Arrays in Java sind Variablen oder Konstanten, die mehrere Elemente des jeweils gleichen Datentyps speichern können. Gleichzeitig enthalten Arrays als Klasse auch Informationen, mit denen z.B. die bei der Instanzierung angegebene Größe festgestellt werden kann.

Zunächst muss zuerst eine Array-Variable deklariert werden, anschließend das mit **`new`** erzeugte Array der Variablen zugewiesen werden:

```
int[] a;  // Deklaration Array-Variable a
// Zuweisung eines Arrays mit 5 int-Elementen
a = new int[5];
```

Arrays - Datenfelder

Die Deklaration und Initialisierung von Arrays kann auch in einem Schritt durchgeführt werden.

```
int[] a = new int[5];
int[] a = { 1, 2, 3, 4, 5}; // Initialisierung
```

Arrays - Datenfelder

Ein Element eines Arrays kann daher auch selbst wieder ein Array sein, so dass auf diese Weise ein mehrdimensionales Array entsteht.

Beispiel-Anwendungsfälle für Arrays:

- Liste mit allen Matrikelnummern der Teilnehmer(innen) dieser Vorlesung
- Speichern einer ganzen Matrix von m x n Werten, wobei m die Anzahl der Zeilen und n die Anzahl der Spalten beschreibt.

Arrays - Datenfelder

In Java werden die Grenzen eines Arrays genau überwacht.

- Im Gegensatz zu anderen Programmiersprachen ist es in Java nicht möglich, über die Grenzen eines Arrays hinaus andere Speicherbereiche zu überschreiben oder auszulesen.
- Ein Zugriff außerhalb der Grenzen des Arrays führt zu dem Melden einer Ausnahme.

Zugriff auf Arrays (1)

Der Zugriff auf ein Array-Element erfolgt, wie in C, über seinen Index, beginnend mit 0.

```
public static void main(String[] args)
{
  int[] zahl = new int[2];
  zahl[0] = 2;
  zahl[1] = 3;
  System.out.println("zahl hat " + zahl.length +
                     " Elemente.");
  System.out.println(zahl[0]);
  System.out.println(zahl[1]);
  }
}
```

Zugriff auf Arrays (2)

Typisch ist der Zugriff auf Arrays mit einer **`for()`**-Schleife, die von 0 bis zur Arraygröße minus 1 zählt. Das folgende Beispiel gibt die Kommandozeilenparameter, mit denen das Java-Programm **`Beispiel`** aufgerufen wurde, nacheinander aus.

```
public static void main(String[] args) {
 for(int i=0; i<args.length; i++)
  System.out.println("args[" + i + "] ist: "
                     + args[i]);
}
```

Beispiel-Aufruf:

```
java Beispiel "Dies ist ein Test" und so weiter.
```

Zugriff auf Arrays (3)

Ein Spezialfall ist die sog. „Iterator-Schleife“ , die in einer Variable automatisch alle Elemente des Arrays nacheinander zuweist.

```
for (int val : feld) {
 System.out.println (val);
}
```

Dies entspricht in der Funktionsweise einer „normalen“ for-Schleife wie z.B.:

```
for (int i=0; i<feld.length; i++) {
 System.out.println (feld[i]);
}
```

Zugriff auf Arrays (4)

```
public class ArrayRandomTest {
 public static void main (String[] args) {
  int [] zufall = new int [10];
  for (int i = 0; i < zufall.length; i++) {
   zufall [i] = (int) (Math.random () * 100.0);
   // Math.random () liefert einen (zufälligen)
   // Double-Wert zwischen 0 und 1: 0 <= x < 1
  } // for
  for (int val : zufall) {
   System.out.print (val + " ");
  } // for
 } // main
} // class ArrayRandomTest
```

Matrix - Array (1)

Beispiel für eine zwei-dimensionale Matrix:

```
int [] [] matrix; // Deklaration
matrix = new int[2][3]; // Erzeugen
for (int x = 0; x < 2; x++)
 for (int y = 0; y < 3; y++)
  matrix [x][y] = -1;
```

Matrix - Array (2)

Die Anzahl der Elemente eines Arrays kann über das Attribut **`length`** (vergl. Funktion **`length()`** bei **`String`**s 118) auch im mehrdimensionalen Fall abgefragt werden.

```
int[][] matrix = new int[2][3];
System.out.println(matrix.length); // hier: 2
System.out.println(matrix[0].length); // 3
```

Beispiel: Pascal'sches Dreieck mit Array

```
// Wie viele Felder hat ein Pascalsches Dreieck
// mit n Zeilen?
public class ArrayPascalTest {
 public static void main (String[] args) {
  int n = 5; // Anzahl der Zeilen in Pascal-Dreieck
  // Deklaration und Erzeugung der ersten Dimension
  int [][] pascal = new int [n][];
  for (int i = 0; i < n; i++) // für n Zeilen
   pascal[i] = new int [i+1]; // Erzeugung 2. Dimensio
  int anzFelder = 0; // Zähler
  for (int i = 0; i < n; i++)
   for (int j = 0; j < pascal[i].length; j++)
    anzFelder++;
  System.out.println("Anzahl Felder bei " + n
   "Zeilen: " + anzFelder);
 } // Ende main
} // Ende class ArrayPascalTest
```

Wichtige Eigenschaften von Arrays in Java (1)

- Die Dimension (Länge des anzulegenden Arrays) kann zur Laufzeit angegeben werden.
- Ist das Array angelegt, dann kann seine Länge nicht mehr verändert werden.
- Die Definition einer Array-Variablen bedeutet in Java nicht das Anlegen eines Arrays, sondern die Definition einer Referenz-Variablen, die auf ein Array-Objekt zeigen kann.
- Das Anlegen selbst erfolgt auf dem Heap („Stapelspeicher" von Java) mit dem **`new`**-Operator
- Die Indizierung beginnt mit dem ersten Element bei 0.

Wichtige Eigenschaften von Arrays in Java (2)

- Im Grunde kennt Java nur (eindimensionales) Array
  - ...aus Elementen eines einfachen Datentyps oder
  - ...aus Elementen eines Referenztyps.
- Ein Element eines Arrays kann aber selbst auch wieder ein Array sein. Auf diese Weise können mehrdimensionale Arrays produziert werden.

Wichtige Eigenschaften von Arrays in Java (3)

Beim Anlegen einer Array-Variablen werden die Feldinhalte mit einem typabhängigen Standardwert belegt:

Datentyp	Standardwert
`boolean`	`false`
`byte`	`0`
`short, int, long`	`0`
`float`	`0.0F`
`double`	`0.0`
`char`	`\n0000`

Erinnerung: Bei einfachen Variablen war eine anwendungsspezifische Initialisierung notwendig

Achtung: In vielen anderen Programmiersprachen sind die Variablen stattdessen mit einem Zufallswert belegt!

Arrays sortieren - Arrays.sort()

```
import java.util.Arrays;
...
int[] array_1 = { 10, 5, 1, 8, -1, 4 };
Arrays.sort(array_1);
```

Arrays: java.util

Die Klasse **`Arrays`** aus dem Paket **`java.util`** bietet Methoden zum komfortablen Arbeiten mit Arrays. Beispiele:

`static void fill (Typ[] feld, Typ w)`
Alle Elemente aus feld erhalten den Wert **`w`**

`static void fill (Typ[] feld, int from, int to, Typ w)`
Alle Elementen ab Index from bis Index to-1 in feld erhalten den Wert **`w`**

`static void sort (Typ[] feld)`
die Elemente des Arrays feld werden aufsteigend sortiert

`static int binarySearch (Typ[] feld, Typ schluessel)`
durchsucht das Array **`feld`** nach dem Wert **`schluessel`**, im positiven Fall wird der entsprechende Index zurück geliefert, im negativen Fall wird ein negativer Index zurückgeliefert, die Elemente in **`feld`** müssen aufsteigend sortiert sein.

Prozeduren

Eine **Prozedur** fasst Anweisungen (i.d.R. mehrere) unter einem neuen Namen zusammen. Im Gegensatz zu einer ☞ **Funktion** (137) besitzt sie keinen Rückgabewert (was mit dem Schlüsselwort **`void`** gekennzeichnet wird), kann aber durchaus Übergabeparameter erhalten.

```
void print_int(double d){
 System.out.print("Gerundeter Wert: ");
 System.out.println((int) d);
}
```

Diese Beispiel-Prozedur kann später so aufgerufen werden:

```
print_int(radius * 2.0 * Math.PI);
```

Funktionen

Eine **Funktion** kann **Übergabeparameter** besitzen, und kann einen **Rückgabewert** liefern. Die Übergabeparameter werden innerhalb der Funktion wie Variablen behandelt. Der Rückgabewert wird mit der Anweisung **`return wert;`** beim Aufruf der Funktion übertragen.

```
int quadrat(int seite){
 return seite * seite;
}
```

Die Funktion kann, wenn sie Rückgabewerte liefert, später in einer **Zuweisung** aufgerufen werden:

```
int flaeche = quadrat(10);
System.out.println(flaeche);
```

Methoden

Im Gegensatz zu vielen anderen prozeduralen Programmiersprachen kennt JAVA keine „freien Funktionen“. Stattdessen befinden sich sämtliche Funktionen und Prozeduren, gleichgültig ob selbst-definierte oder solche aus der JAVA-Bibliothek, immer innerhalb einer Klasse, und können von anderen Klassen aus unter Angabe des Klassenpfads aufgerufen werden, wenn sie bereite instanziert wurden oder **`static`** sind, z.B.: **`System.out.println(...);`**

Analog zu „Klassenvariablen“ bezeichnet man die Funktionen und Prozeduren innerhalb einer Klasse als „Klassenmethoden“.

Deklaration vs. Implementation

Eine **Deklaration** gibt zunächst nur an, wie eine **Methode heißt** und **wie sie aufzurufen ist** („Funktionskopf").

```
double summe(double [] a);
```

In der **Implementation** wird die Funktionsweise der Methode **programmiert** („Funktionsrumpf" oder „-körper"). Dies erfolgt häufig (aus praktischen Gründen) kombiniert mit der Deklaration, kann aber auch später (z.B. vor der ersten Benutzung in einer abgeleiteten Klasse) stattfinden.

```
double summe(double [] a) {
  double tmp = 0.0;
  for(int i=0; i<a.length; i++) tmp += a[i];
  return tmp;
}
```

Übergabeparameter von Funktionen (1)

In JAVA (wie in den meisten anderen Programmiersprachen) können einer Funktion Variablen bzw. Werte „mitgegeben“ werden, diese werden in den runden Klammern hinter dem Funktionsnamen mit Typ und Variablennamen angegeben. Eine Übergabevariable existiert nur innerhalb der Funktion, andere Variablen mit dem gleichen Namen, die aus der Klasse stammen, werden dadurch „überschattet“.

Übergabeparameter von Funktionen (2)

Mit dem JAVA-Schlüsselwort **`this`** kann explizit eine Variable der Klasse (bzw. des von der Klasse gebildeten Objekts) angesprochen werden, so können auch gleichnamige Variablen innerhalb und außerhalb der Funktion unterschieden werden.

```
public class A {

 int i;

 int i1_mal_i2(int i){
  // this.i: Variable i in einem A-Objekt
  // A.i:    statische Variable i der Klasse A
  // i:      Übergabevariable i
  return this.i * i;
 }

}
```

Übergabeparameter von Funktionen (3)

Bei Übergabevariablen werden

- bei Basisdatentypen (**`int`**, **`float`**, ...) Werte in die neuen Variablen kopiert, d.h. eventuell aus dem Hauptprogramm übergebene Variablen bleiben unverändert. ☞ **Call by Value**
- bei Arrays oder komplexen Datentypen (Objekten) Referenzen auf den Speicherbereich übergeben, so dass von der Funktion aus Werte innerhalb der übergebenen Arrays oder Objekte im Hauptprogramm auch geändert werden können. ☞ **Call by Reference**

Übergabeparameter von Funktionen (4)

```
void f(int i){ i = 1; }
...
int x = 10;
f(x);
```

x bleibt unverändert

```
void f(int[] i){ i[0] = 1; }
...
int[] x = { 10 };
f(x);
```

x[0] wird auf 1 gesetzt

Übergabeparameter von Funktionen (5)

Eine Funktion oder Prozedur darf in JAVA mehrfach deklariert werden, sofern sich die gleichnamigen Methoden in den Übergabeparametern unterscheiden. Durch die Art des Aufrufs kann dann automatisch die „richtige" Methode ausgewählt werden. Man nennt dies **Überladen** von Funktionen.

```
void f(int i){ ...}
void f(String d){ ...}
...
f(10);
f("Hallo, Welt!");
```

Übergabeparameter von Funktionen (6)

Eine gleichnamige Methode mit gleichen Übergabeparametertypen, die sich nur im Rückgabedatentyp unterscheidet, ist allerdings nicht erlaubt, da JAVA beim Aufruf nicht unterscheiden kann, welche der beiden Methoden gewählt werden soll!

```
int f(int i){...}
float f(int i){...} // Verboten!
...
System.out.println(f(10));
```

Rekursion (1)

Rekursiv heißt eine Prozedur oder Funktion, die sich selbst aufruft.

```
void f_rek(int i){
 if(i>0){                    // Prüft auf Rekursions-Ende
  f_rek(i-1);                // Ruft sich selbst auf mit i
  System.out.println(i); // Ausgabe von i
 }
}
```

Vorteil: In der Mathematik bereits rekursiv definierte Funktionen wie ☞ Fibonacci können elegant beinahe 1:1 in die Programmiersprache übertragen werden; Schleifen-Konstrukte entfallen. ☞ Vereinfachung des Algorithmus.

Nachteil: In den klassischen Ingenieurs-Aufgabenstellungen ungewohnt; Rekursionsschritte sind anfangs schwer nachvollziehbar. („Ein 3D-Drucker druckt sich selbst aus?“)

Rekursion (2)

Der folgende Sachverhalt kann in der theoretischen Informatik nachgewiesen werden:

- Jede berechenbare Funktion kann grundsätzlich auch mit Hilfe der Rekursion dargestellt werden, also gänzlich ohne die Verwendung von lokalen Variablen (außer Übergabeparametern) oder Schleifen.
- Es gilt also: Die sog. Mächtigkeit der Rekursion entspricht der Mächtigkeit der imperativen Programmierung. ☞ s.a. Beispiel Fakultäts-Funktion.

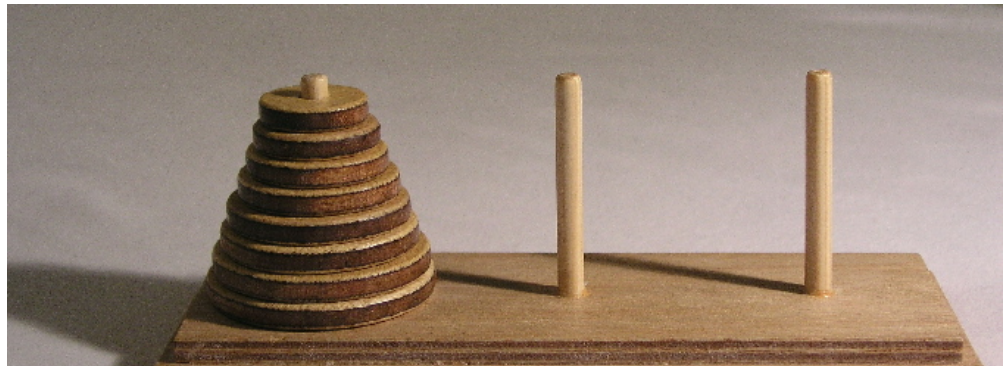
Rekursion (3a)

Beispiel: ☞ „Die Türme von Hanoi“ lösen

Ein geordneter Stapel von n der Größe nach sortierten Scheiben soll verschoben werden. Dazu

- darf immer nur eine Scheibe in einem Schritt bewegt werden und
- nie eine größere auf eine kleinere Scheibe abgelegt werden.

Löse dieses Problem mit insgesamt drei Ablageplätzen a, b und c, wobei der Stapel zu Beginn bei a steht und nach b verschoben werden soll. Es darf also nur ein Hilfsstapel (c) verwendet werden.



Rekursion (3b)

Mit Rekursion ist die Lösung sehr kurz! Das Verschieben eines beliebig hohen Stapels wird auf die Lösung zum Verschieben eines 2er Stapels bzw. einer einzigen Scheibe zurückgeführt.

```
public class TuermeVonHanoi {

 // Bewegt n Scheiben von Turm a nach Turm b und
 // benutzt als Zwischenspeicher Turm c.

 static void bewege (char a, char b, char c, int n) {
  if (n == 1) System.out.println("Bewege Scheibe von " + a + " auf " + b);
  else {
   bewege (a, c, b, n-1); // die oberen n-1 Scheiben von a nach c
   bewege (a, b, c, 1);   // Bewege größte Scheibe von a nach b
   bewege (c, b, a, n-1); // die oberen n-1 Scheiben von c nach b
  }
 } // bewege()

 public static void main (String[] args) {
  // Gib die notwendigen Züge für einen Stapel der Höhe 5 aus
  bewege('a', 'b', 'c', 5);
 } // main()

} // TuermeVonHanoi
```

Algorithmen-Komplexität (Theorie)

Ziele:

- Laufzeitverhalten des Programms erkennen, v.a. „Programm-Hänger“ durch unerwartet lange laufende Schleifen- oder Rekursionskonstrukte (Grenzwerte für $n \to \infty$ betrachten),
- Vereinfachung einer komplizierten Aufgabe durch Zerlegung in Teilaufgaben mit geringerer Komplexität und Kombination dieser Teilaufgaben zur Lösung des Problems.
- Laufzeitoptimierung durch Finden eines alternativen Algorithmus zur Lösung des Problems mit geringerer ☞ Komplexitätsklasse (152).

☞ Foliensatz Schiefer gdi_07_AlgorithmenKomplexitaet.pdf S. 18-31.

(Zeit-)komplexität

Faustregeln zur Bestimmung:

- Schleifen: Anzahl der Schleifendurchläufe mal Laufzeit der „teuersten“ Schleifenausführung.
- Geschachtelte Schleifen: Produkt der Größen aller Schleifen mal Laufzeit der innersten Anweisung
- Nacheinander-Ausführung: Zunächst Addition der Laufzeiten der Anweisungen der Sequenz. Dann werden konstante Faktoren weggelassen und nur der jeweils höchste Exponent berücksichtigt.
- Fallunterscheidung (**`if...else, switch`**): Laufzeit der Bedingungsanweisung plus Laufzeit der „teuersten“ Alternative.
- Rekursiver Prozeduraufruf: Anzahl der rekursiven Aufrufe mal Laufzeit der „teuersten“ Funktionsausführung.

Komplexitätsklassen in der **O-Notation**

Klasse	**Charakteristik**	**Beispiel**
$O(1)$	konstant	Sortierte Tabellen-Suchverfahren (Hashing, Arrays mit Zahlenindex)
$O(log\, n)$	logarithmisch	Allgemeine Tabellen-Suchverfahren, optimierte binäre Suche
$O(n)$	linear	sequentielle Suche, Suche in Texten, syntaktische Analyse (best case)
$O(n \cdot log\, n)$		teiloptimierte Suchverfahren
$O(n^2)$	quadratisch	Einfache Sortierverfahren, einige dynamische Optimierungsprobleme
$O(n^3)$	kubisch	Einfache Matrizen-Multiplikation
$O(2^n)$	exponentiell	Viele Optimierungsprobleme, Bestimmung aller Teilmengen einer Menge, ...

Konstanten entfallen bei der Bestimmung v. Komplexitätsklassen $\left(\lim_{n\to\infty} m \cdot n + c = n\right)$.

Was ist ein „größeres Software-Projekt“?

Die folgende Tabelle gibt einen groben Einblick in die Klassifikation von Software-Projekten hinsichtlich der Komplexität in der Bearbeitung.

Software-Klasse	**Codezeilen (LOC)**	**Bearbeitungs-aufwand (PJ)**
Sehr klein	0 - 1.000	0 - 0,2
Klein	1.000 - 10.000	0,2 - 2
Mittel	10.000 - 100.000	2 - 20
Groß	100.000 - 1.000.000	20 - 200
Sehr groß	1 Mio. und mehr	200 und mehr

LOC - Lines of code

PJ – Personenjahre

ENDE DER VORLESUNG SOMMERSEMESTER 2020

Prüfungsrelevant sind alle in der Vorlesung behandelten Themen, Übungen, Handouts, ...

Tipp für die Vorbereitung: Da Sie in der Klausur schriftliche Materialien (auch Musterlösungen alter Klausuren und Übungen, Bücher, ...) mitnehmen dürfen, handelt es sich größtenteils um **Transferaufgaben**, d.h. Sie sollen nicht Texte wörtlich auswendig lernen und „reproduzieren“ , sondern **das erworbene Wissen** (Methoden, Kompetenzen) **in neuen Aufgabenstellungen anwenden können**.

Viel Erfolg in der Prüfung!

Mit den Folien auf den nächsten Seiten steigen wir in Softwaretechnik im Wintersemester 20/21 wieder ein.

Objektorientierte Programmierung (OOP)

OOP (1)

Bisher erschien das Programmieren der **`main()`** umgebenden Klasse einfach nur umständlich, nun wird der Sinn und Zweck objektorientierter Programiersprachen untersucht.

```
public class OO {
 void hello() { System.out.println("Hello"); }
 public static void main(String[] args){
  OO oo = new OO();
  oo.hello();
 }
}
```

OOP (2)

Seit Beginn der 80er Jahre hat sich die Objektorientierung in der Softwaretechnik kontinuierlich ausgebreitet und ist heute beherrschend. Dahinter verbirgt sich zunächst ein neues Paradigma:

Die Anwendung (das Programm) besteht aus einer Menge miteinander agierender *Objekte*.
Alle Eigenschaften und Methoden dieser Objekte sind in ihrem Inneren gekapselt.

OOP (3)

Was soll der Vorteil der Objektorientierung sein?

- Während die verwendeten Methoden und Eigenschaften der Objekte durchaus komplex sein können, ist der Zugriff und die Zusammenschaltung sehr einfach, da die Objekte selbst „wissen", wie sie funktionieren.
- Kritische Parameter und Funktionen können „privat" gemacht und somit vor dem Zugriff von außerhalb des Objektes geschützt werden ☞ Geheimnisprinzip (170).
- Sobald die Logik eines Objektes in diesem implementiert ist, funktioniert es „von alleine".
- Wiederverwendbarkeit und Programmieren im Team an Teilaufgaben (Module) großer Programme wird durch die Kapselung in Objekte und Pakete unterstützt.
- Durch *Vererbung* und *Spezialisierung* können bestehende Programme leicht erweitert werden, ohne bei Änderungswünschen alles neu schreiben zu müssen.
- Der Programmierer wird von der Verwaltung des Hauptspeichers entbunden, durch dynamisches Erzeugen und automatisches Löschen von Objekten, was auch Fehlerquellen durch falsches Allozieren und Freigeben von Speicher ausschließt (eine der häufigsten Fehlerquellen z.B. in C).

OOP (4)

Die Vorteile der Objektorientierten Programmierung zeigen sich oft erst bei größeren, im Team bearbeiteten Projekten ☞ Software Engineering (übernächstes Semester).

Dennoch werden die Eigenheiten der Objektorientierung auch bei kleinen Programmen (s. nächste Folie) durch regelmäßige Übung bald zur Gewohnheit.

OOP (5)

Entscheidend für den objektorientieren Ansatz ist erst im zweiten Schritt das objektorientierte Programmieren, zunächst findet das **Denken in Objekten** statt.

- Es wird dazu in Konzepten und **Begriffen der realen Welt** anstatt in rechnernahen Konstrukten wie „Haupt- und Unterprogramm“ gedacht.
- Vor der Programmierung wird **analysiert**, **welche Objekte von Bedeutung sind**, um die **Aufgaben des Zielsystems zu erfüllen**.

Beispiel: Für die Entwicklung einer Bankanwendung müssen zunächst die relevanten Objekte gefunden werden:

1. Konto
2. Kunde
3. ...

Objekte

Jedes Objekt hat gewisse Attribute/Eigenschaften und ein Verhalten.

- Die **Eigenschaften** werden durch **Daten** beschrieben, z.B. beim *Konto*: *KontoNr*, *Saldo*, ...
- Das **Verhalten** wird durch Operationen **(Methoden)** beschrieben, die auf einem Objekt ausgeführt werden können, z.B. beim *Konto*: *Geld abheben* oder *Geld einzahlen*.

Klassen

Klassen stellen die **Baupläne für Objekte** dar. Von einer Klasse können i.d.R. beliebig viele gleichartige Objekte erzeugt **(instanziiert)** werden, die unterschiedliche konkrete **Belegungen ihrer Eigenschaften** besitzen können.

Beispiel: Es gibt mehrere von der Klasse **`Konto`** gebildete Objekte, die unterschiedliches **`Saldo`**, **`KontoNr`** usw. besitzen.

Klassen entsprechen damit **Datentypen**, die aber komplexer aufgebaut sind als die bekannten Basisdatentypen **`int`**, **`float`** usw.

Objekte stellen damit Variablen **(Instanzen)** dieser Datentypen dar.

Aufbau von Klassen

- Jede Klasse besitzt einen Klassen-**Namen**.
- Die Klasse legt die **Datenfelder** und die **Methoden** der von ihr gebildeten Instanzen fest.
- Klassen stellen außerdem einen **Namensraum** dar: Die gleichen Datenfelder und Methodensignaturen können in verschiedenen Klassen existieren, aber unterschiedliche Bedeutung bzw. Funktionsweise besitzen.
- Jede Klasse besitzt mindestens einen **Konstruktor** (164), mit dem sich Objekte von ihr instanziieren lassen, auch wenn dieser nicht explizit programmiert wird.

Konstruktor (1)

Der **Konstruktor** ist eine Funktion, die so heißt wie die Klasse, in der sie steht, und die ein von der Klasse instanziiertes Objekt zurückgibt. Sofern der Konstruktor nicht in der Klasse angegeben ist, gilt der **Default Konstruktor**, der lediglich Speicherbereich für das neue Objekt zur Verfügung stellt.

Wird der Konstruktor explizit programmiert, so kann er **Initialisierungsaufgaben** übernehmen, beispielsweise eine neue Kontonummer erzeugen und den Kontostand zunächst auf 0 setzen.

```
public Konto(){
 saldo_euro = saldo_cent = 0;
 iban = "DE" + (long) (Math.random() *
                  10000000000000000000000L);
}
```

Aufruf in `main()`: `Konto k1 = new Konto();`

Konstruktor (2)

Mit einem weiteren Konstruktor, der Übergabeparameter besitzt, lässt sich das Erzeugen eines Objektes besser steuern:

```
public Konto(String ki, String ib,
             int se, int sc){
 kontoinhaber = ki;
 iban = ib;
 saldo_euro = se;
 saldo_cent = sc;
}
```

Aufruf in `main()`:

```
Konto k2 = new Konto("K.K.","DE012345678...",1000,0);
```

Destruktor

Jede Klasse enthält, auch wenn es nicht explizit angegeben ist, eine Methode, die nach Beendigung der Lebenszeit eines Objektes dieser Klasse automatisch von der sog. „Garbage Collection“ der Java VM aufgerufen wird. Der genaue Zeitpunkt ist allerdings nicht vorhersehbar.

```
public class KonstruktorTest {
  public KonstruktorTest()  {
      System.out.println("Konstruktor wurde aufgerufen!");
  }
  protected void finalize()  {
      System.out.println("Destruktor wurde aufgerufen!");
  }
  public static void main(String[] args) {
      KonstruktorTest t1 = new KonstruktorTest();
  }
}
```

Garbage Collector aufrufen

```
[...]
System.gc();
[...]
```

Hiermit werden die Destruktoren aller nicht mehr benötigten Objekte aufgerufen, und der Speicher freigegeben.

Programmierung der Klasse `Konto`

Beispiel:

```
class Konto {
 String kontoinhaber;
 String iban;
 int saldo_euro;
 int saldo_cent;
 void einzahlen(int euro, int cent);
 boolean abheben(int euro, int cent);
 boolean ueberweisen(String iban, int euro, int cent);
}
```

Nachteil: Statt die Methoden **`einzahlen()`**, **`abheben()`** und **`ueberweisen()`** zu benutzen, könnte ein Programmierer auch einfach die Variablen **`saldo_euro`** und **`saldo_cent`** verändern, und jegliche Überprüfung in den vorgesehenen Methoden umgehen!

Geheimnisprinzip (1)

(Information Hiding)

Bei der Entwicklung großer Software-Systeme stellt die Kommunikation zwischen den Entwicklern eines der größten Probleme dar. Lösungsansatz:

- Jede Person bearbeitet einzelne Programmbausteine, auch Module genannt. Verknüpft sind diese miteinander sowohl über Kontroll- als auch Datenstrukturen.
- Diese Module müssen so definiert und gegeneinander abgegrenzt werden, dass jeder Kollege (Co-Entwickler) nur das für ihn Notwendige über diese erfährt, *aber nicht mit Detailwissen über die Module der anderen überfrachtet wird*.

Geheimnisprinzip (2)

☞ David Parnas schlug 1972 das ☞ Geheimnisprinzip vor, welches besagt, dass beim Entwurf eines großen Systems jedes Modul in zwei Teilen zu beschreiben ist:

1. Alle Vereinbarungen, deren Kenntnis für die Benutzung des Moduls durch andere Module notwendig sind: Der sogenannte „Visible Part“ oder auch **Spezifikation** genannt.
2. Alle Vereinbarungen und Anweisungen, die für die Benutzung des Moduls durch andere Module nicht benötigt werden: Der sogenannte „Private Part“ oder auch **Konstruktion** genannt.

Heute bekannt als: Prinzip der **Datenabstraktion**.

Datenabstraktion/Datenkapselung

- **Daten** und darauf operierende Funktionen **(Operationen)** sollen **immer gemeinsam** in einem unmittelbaren Zusammenhang definiert werden.
- **Datenstrukturen** sind so in Module zu verpacken (verkapseln), dass auf sie **von anderen Modulen** nur **über ihre Operationen** zugegriffen werden kann.
- Die **Beschreibung** (Schnittstelle bzw. **Signatur**) dieser Operationen macht die **Spezifikation** des Moduls aus, die für alle anderen sichtbar ist.
- Die **Programmierung** (Implementation) der konkreten Datenzugriffe erfolgt im **privaten Konstruktionsteil** und damit im alleinigen Verantwortungsbereichs des damit befassten Entwicklers

Lösung in JAVA: `public`, `private`, `protected`

`final` Variablen und Methoden in einem Objekt dürfen nur ein einziges Mal zugewiesen bzw. programmiert werden, unabhängig von den folgenden Zugriffsrechten.

`public` Variablen und Methoden in einem Objekt sind für alle anderen Objekte zugreifbar.

`private` Variablen und Methoden sind nur für das Objekt, in dem sie stehen, zugreifbar.

`protected` Variablen und Methoden sind für alle Objekte in der gleichen **`package`** (=Unterverzeichnis) zugreifbar. Ist keine **`package`** angegeben, ist die *default package* für die entsprechenden Objekte die gleiche.

Ist nichts anderes angegeben, dann gilt das Zugriffsrecht **`protected`** innerhalb der (default) **`package`**!

Verbesserte Programmierung der Klasse `Konto`

Beispiel:

```
public class Konto {
 final String kontoinhaber;
 final String iban;
 private int saldo_euro;
 private int saldo_cent;
 public int[] kontostand();
 public void einzahlen(int euro, int cent);
 public boolean abheben(int euro, int cent);
 public boolean ueberweisen(String iban,
                            int euro, int cent);
}
```

Nun ist ein Lesen oder Verändern des Kontostandes nur noch mit den als **`public`** deklarierten Methoden von anderen Objekten aus möglich!

`public` Methoden zum Auslesen von `private` Variablen

Die Methode **`kontostand()`** könnte so implementiert werden, um die aktuellen Werte der privaten Variablen **`saldo_euro`** und **`saldo_cent`** zurückzugeben:

```
public int[] kontostand(){
 int[] k = new int[2];
 k[0] = saldo_euro;
 k[1] = saldo_cent;
 return k;
}
```

Dabei werden die Werte der privaten Variablen in die „Öffentlichkeit“ kopiert, da **`kontostand()`** eine **`public`**-Funktion ist.

`public` Methoden zum Verändern von `private` Variablen

Die Methode `einzahlen()` könnte so implementiert werden, um die aktuellen Werte der privaten Variablen `saldo_euro` und `saldo_cent` kontrolliert zu verändern:

```
public void einzahlen(int euro, int cent){
 if(euro >=0 && cent >=0){
  saldo_cent += cent;
  saldo_euro += euro + (saldo_cent / 100);
  saldo_cent %= 100;
 } else {
  System.err.println("Falsche Beträge eingegeben!");
 }
}
```

Die `System`-Klasse

... enthält einige weitere Anweisungen, die das JAVA-Runtime-System betreffen.

Anweisung	**Wirkung**
`System.exit(int status)`	beendet das laufende Programm vorzeitig.
`System.currentTimeMillis()`	liefert die Anzahl der Millisekunden, die seit dem 1.1.1970 um 00:00:00 Uhr GMT bis zum Aufruf der Methode vergangen sind.
`System.getProperties()`	liefert die definierten Systemeigenschaften der Plattform als Objekt der Klasse Properties.

Vererbung

Eine Klasse, die eine Grundstruktur enthält, die an weitere Klassen *vererbt* wird kann, heißt **Basisklasse**.
Der Vorgang der Vererbung von Eigenschaften wird mit dem JAVA-Schlüsselwort **`extends`** gekennzeichnet.

```
public class B extends A{
 int zwei;
}

class A{
 int eins;
}
```

☞ Die Klasse **`A`** enthält die Variable **`eins`**.
☞ Die Klasse **`B`** enthält die Variablen **`eins`** und **`zwei`**, da wie von Klasse **`A`** erbt und diese **erweitert**.

Limitierung in JAVA

1. Eine Klasse kann immer nur von einer anderen Klasse erben, nicht von mehreren.
2. Variablen vom Typ der Basisklasse können Objekte vom Typ der erweiterten Klasse zugewiesen bekommen, umgekehrt nicht.
3. Objekte unterschiedlicher Klassen, die nicht die gleiche Basisklasse besitzen, sind untereinander zuweisungsinkompatibel.

Die Klasse `Object` (1)

- Wenn nichts anderes angegeben ist, erweitert eine Klasse automatisch die Basisklasse **`Object`**.
- Einer Variablen vom Typ **`Object`** kann ein Objekt von jeder beliebigen Klasse zugewiesen werden.

☞ Dadurch eignen sich Variablen vom Typ **`Object`**, wenn man sich nicht früh auf einen bestimmten Datentyp festlegen will.

Die Klasse `Object` (2)

`Object` enthält selbst bereits einige Methoden, die an alle Klassen vererbt werden.

Anweisung	**Wirkung**
`String toString()`	Zeigt Klassenname und Speicheradresse
`boolean equals(Object obj)`	Vergleicht den Inhalt
`int hashCode()`	eine Prüfsumme anzeigen
`Object clone()`	Eine Kopie erzeugen

instanceof

Überprüfen, ob eine Objekt-Variable eine Instanz einer bestimmten Klasse, oder von der Klasse abgeleiteten Klasse besteht.
Beispiel: Was wird ausgegeben? (**`Girokonto extends Konto`**, **`Sparkonto extends Konto`**)

```
k = new Girokonto ();
System.out.println("k ist ein Konto: " +
                   (k instanceof Konto));
System.out.println("k ist ein Girokonto: " +
                   (k instanceof Girokonto));
System.out.println("k ist ein Sparkonto: " +
                   (k instanceof Sparkonto));
```

`this` und `super`

`this` bezeichnet immer das **aktuelle Objekt**, in dem der Bezeichner steht.
`super` bezeichnet das Objekt der übergeordneten Klasse (bei Vererbung).

Überladen von Funktionen (ad-hoc Polymorphie)

Eine Methode, die in der Basisklasse bereits existiert, kann in einer erweiterten Klasse überschrieben („überladen“) werden. Hierdurch verliert die Methode der Basisklasse ihre Funktion, und es wird die der erweiterten Klasse verwendet, wenn auf das Objekt zuggriffen wird.
Beispiel: Die Funktion **`String toString()`** aus **`Object`**, die normalerweise nur den Klassennamen und die Speicheradresse kann durch eine tabellarische Ausgabe ersetzt werden, wenn die Methode entsprechend neu in der abgeleiteten Klasse definiert wird.