Programmierung I

Einführung in das Programmieren mit JAVA

Prof. Christian Lengauer

Fakultät für Informatik und Mathematik
Universität Passau

Sommersemester 2013

Organisatorisches

- Durchführung
  - Vorlesung: Prof. Christian Lengauer
  - Übung: Dr. Vanessa el Khoury, Dr. Janet Siegmund, Dr. Norbert Siegmund
- Webseiten in Stud.IP
  - http://www.intelec.uni-passau.de/
  - Folien
  - Anzeigen/Ausdrucken mit Adobe Reader `acroread datei.pdf`
    http://get.adobe.com/de/reader/otherversions/
  - Kein Skript ⇒ Sekundärliteratur
  - Übungsblätter auf Übungsseiten
  - Ankündigungen
- Folien
  - Autor: PD Dr. Christian Bachmaier
  - Leichte Aktualisierungen für dieses Semester: Prof. Christian Lengauer

Organisatorisches. . .

- Termine
  - 15 Wochen (15 Doppelstunden)
  - Vorlesung
    - Dienstag 10:15–12:00 Uhr (HS 13 IM)
  - Übungen
    - Gruppe A: Mittwoch 08–10 Uhr (R 028 IM), Vanessa
    - Gruppe B: Mittwoch 10–12 Uhr (R 028 IM), Vanessa
    - Gruppe C: Mittwoch 12–14 Uhr (R 028 IM), Norbert
    - Gruppe D: Mittwoch 14–16 Uhr (R 028 IM), Janet
    - Start Mittwoch ??.?.13 ab 08 Uhr

Organisatorisches...

- Übungsblätter
  - Ausgabe online in Stud.IP Dienstag 14:00 Uhr
  - Abgabe Montag 10:00 Uhr in Übungskasten neben R 030 IM
  - Quellcode zusätzlich per E-Mail an den jeweiligen Übungsleiter
    - berndle@fim.uni-passau.de (R 240 ITZ)
    - Bei Schülern reicht E-Mail
  - Arbeitsgruppen von 2-3 Teilnehmern sind verpflichtend
  - Es gibt Punkte auf Übungsaufgaben
  - Mit 50% der Punkte ist man zur Klausur zugelassen
    - Studenten die Zulassung schon einmal hatten (Wiederholer) brauchen die Punkte nicht unbedingt
  - Auf Nachfrage gibt es Hilfe bei den Aufgaben
  - Dringender Appell: Lösen!
- Anmeldung zur Übung jetzt
  - Arbeitsgruppe muss komplett in einer Übungsgruppe sein

Organisatorisches. . .

- Auf Abgaben schreiben
  - Übungsgruppe
  - Namen und Matrikel-Nummern aller beteiligten Studenten
  - Nummer des Übungsblattes
- Klausur
  - Freitag 6.6.2012 10:00–11:30 Uhr (HS 9 AM)
  - Anmeldung in HisQis in jedem Fall erforderlich
    - https://qisserver.uni-passau.de/
    - Verbindliche Frist: 29.4.2013–26.5.2013
  - Nachholklausur nach Vorlesungsende WS 2013/2014

Aufwand

- c't Magazin 05/11 S. 148:

  Man kann viel über die Entwicklung von Software lesen oder sich zeigen lassen, aber zu einem Könner wird man nur durch häufiges Studieren fremden Codes, Ausprobieren und Anwenden.

- c't kompakt Programmieren 02/11 S. 10:

  Aller Anfang ist schwer. . . . Programmieren ist vielmehr ein ständiges Lernen durch Tun. Das bedeutet z. B., sich mit fremdem Code auseinanderzusetzen, ihn verstehen zu versuchen und abzuwandeln.
  . . . Wie das Erlernen einer Fremdsprache erfordert auch das Erlernen einer Programmiersprache sehr viel Eifer – und zusätzlich Freude an kreativer Arbeit und ein ausgeprägtes Abstraktionsvermögen. Letzteres bedeutet, Probleme so weit aufdröseln zu können, dass man die daraus entstehenden Teilprobleme idealerweise direkt als Programm formulieren kann. Der kreative Anspruch ergibt sich daraus, dass es häufig mehrere Wege zur Lösung gibt, etwa besonders elegante oder besonders kurze – oder vermeidbar umständliche.

Aufwand...

- Die Vorlesung zeigt Techniken und Tipps
- Kann praktische Erfahrung nicht ersetzen
- Es gibt kein generelles Patentrezept ein Programm zu erstellen
- Gutes Programmieren ist eine kreative Tätigkeit
  - Eine Kunst
  - Sonst würde es ein Roboter bzw. ein (anderes) Programm machen
- Motivation, Spaß und persönlicher Einsatz sind unabdingbar

Überblick

- Inhalt der Vorlesung
  - Einführung in das Programmieren mit JAVA
  - Vornehmlich imperativ (Objektorientierung vertieft in Programmierung II)
- Aufbau
  - Einführung
    0. Vorbemerkungen
    1. JAVA
  - Grundlagen
    2. Datenstrukturen
    3. Kontrollstrukturen
    4. Programmstrukturen
  - Programmieren
    5. Benutzerdefinierte Datenstrukturen
    6. Dynamische Datenstrukturen
    7. Benutzung von Datenstrukturen aus der Funktionsbibliothek (API)
    8. Einfache Algorithmen
    9. Ausnahmebehandlung
- Programmierung I und Grundlagen der Informatik
  - Prog I: Fertigkeiten in der z.Zt. populärsten Sprache
  - GdI: Grundlagen, Optionen und Zusammenhänge

Literatur

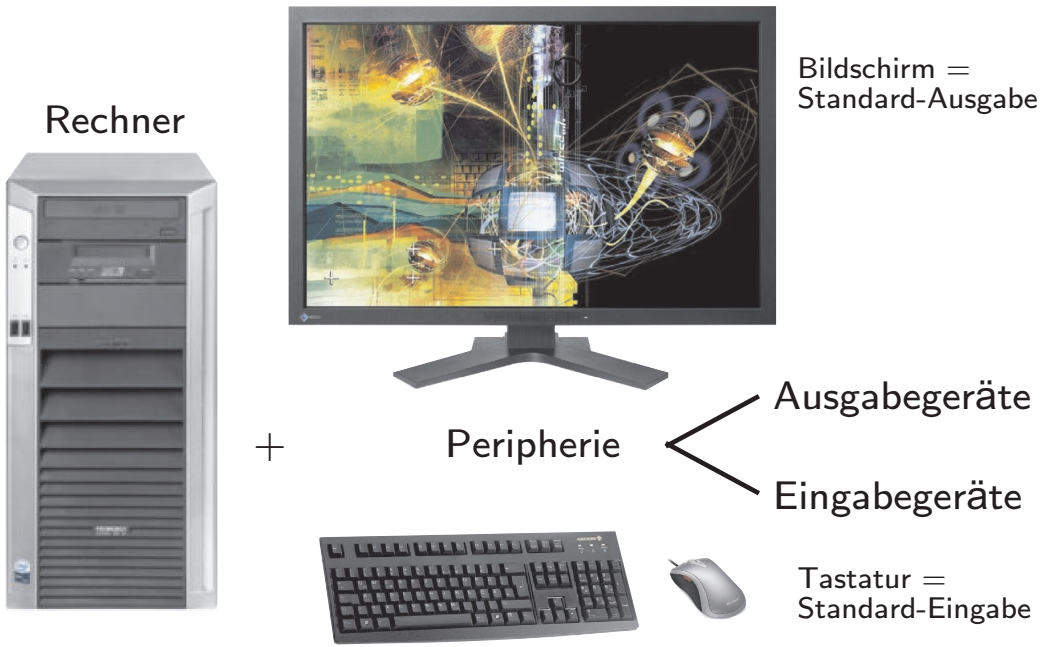
- Einführungen/Tutorials
  - Peter Pepper, *Programmieren lernen*,
    3. Auflage, Springer, 2007, 80+17/ST 250 J35 P4(3),
    http://www.springerlink.com/content/l67236/
  - Reinhard Schiedermeier, *Programmieren mit* JAVA,
    2. Auflage, Pearson Studium, 2010, 80+17/ST 250 J35 S3(2)
  - Dietmar Abts, *Grundkurs* JAVA*: Von den Grundlagen bis zu Datenbank- und Netzanwendungen*,
    6. Auflage, Vieweg, 2010, 80+17/ST 250 J35 A1(6),
    http://www.springerlink.com/content/978-3-8348-1277-3/
  - *The* JAVA *Tutorial*, Oracle Corporation,
    http://download.oracle.com/javase/tutorial/
  - Christian Ullenboom, JAVA *ist auch eine Insel*,
    10. Auflage, Galileo Computing, 2012, 80/ST 250 J35 U4(10),
    http://openbook.galileocomputing.de/javainsel10/
- API Referenz
  - Klassenbibliothek von JDK 7, Oracle Corporation,
    http://download.oracle.com/javase/7/docs/api/

Literatur für Spezialisten

- Spezifikation
  - James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, *The* JAVA *Language Specification*, Java SE 7 Edition, Oracle Corporation, 2012, http://docs.oracle.com/javase/specs/index.html
- Programmierstil
  - *Code Conventions for the* JAVA *Programming Language*, Oracle Corporation, http://www.oracle.com/technetwork/java/codeconv-138413.html
  - *How to Write Doc Comments for the Javadoc Tool*, Oracle Corporation, http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html
- Entwicklungsumgebung
  - Gottfried Wolmeringer, Thorsten Klein, *Profikurs Eclipse 3*, 2. Auflage, Vieweg 2006, http://www.springerlink.com/content/tk04q4/

Aufbau von Computersystemen

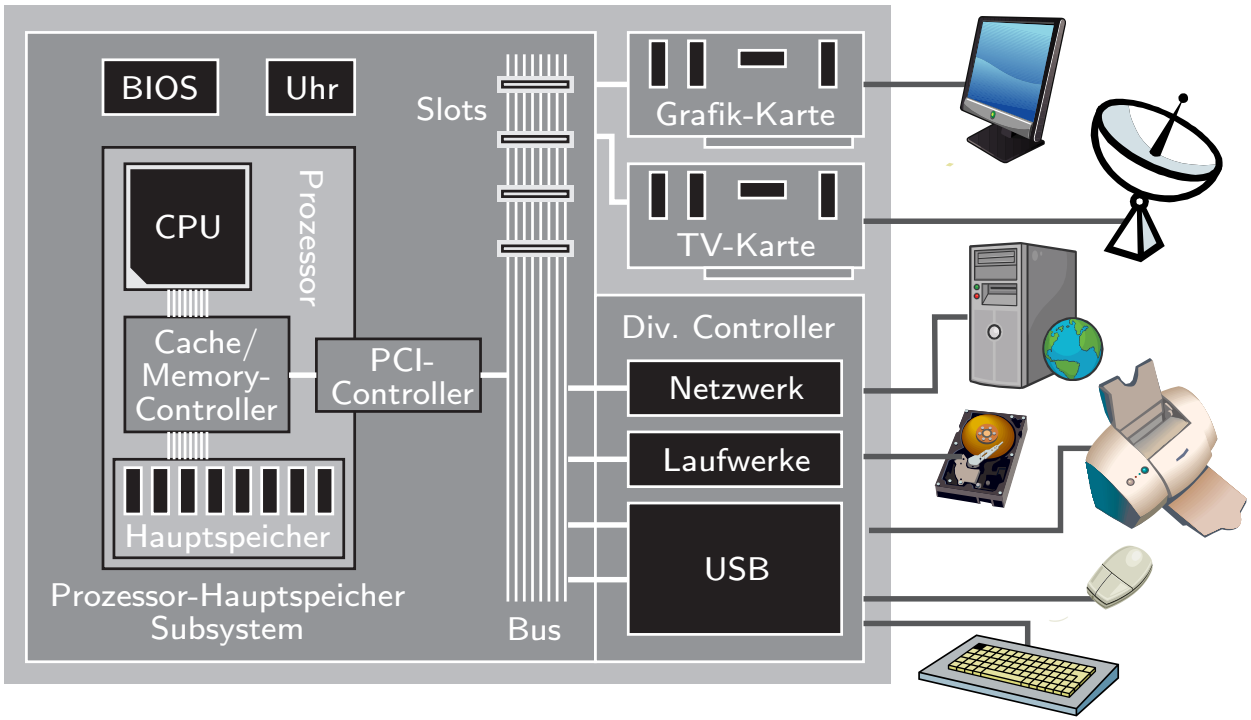
- Rechner und Peripherie



- Andere Eingabegeräte
  CD-ROM/DVD-Laufwerke, Scanner, Microfon, Webcam, Joystick, Datenhandschuh, Stift, ...
- Andere Ausgabegeräte
  Drucker, USB-Laufwerke/-Sticks, Streamer, Videogeräte, BD-Brenner, Modem, Netzwerkkarte, 3D-Brille, ...
- Oft Kombigeräte für Ein- und Ausgabe

Komponenten

- Schematischer Überblick



- Kommunikation der Komponenten über einen Bus
  - Nur einer kann gleichzeitig „sprechen“
- Prozessor ist der „Motor“
- Hauptspeicher ist das „Kurzzeitgedächtnis“
- Festplatte ist das „Langzeitgedächtnis“

Prozessor

- *Central Processing Unit* (CPU)

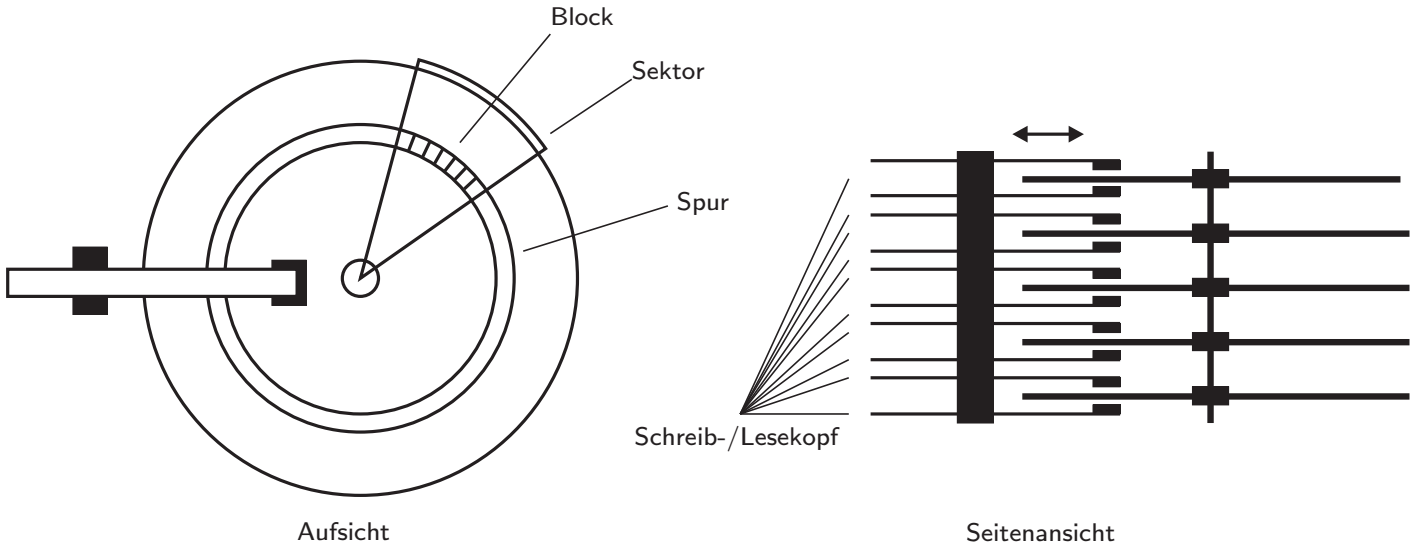
- Arbeitet Befehl für Befehl ab
- Hat einige wenige integrierte sehr schnelle Speicherzellen (*Register*)
- Nur rudimentäre Befehle (*Maschinenbefehle*) wie z. B.
  - `LOAD`: Laden eines Registers mit einem Wert aus dem Speicher
  - `STORE`: Speichern eines Registerinhalts im Speicher
  - `ADD`, `SUB`, `MULT`, `DIV`: Arithmetische Operationen auf Registern
  - `NOT`, `OR`, `AND`, `XOR`: Logische Operationen auf Registern
  - `MOVE`: Verschieben von Datenblöcken im Speicher
- Befehlszähler zeigt auf den als nächstes auszuführenden Befehl
  - Wird defaultmäßig nach Befehlsausführung erhöht
- Heute üblich $3,5$ GHz
  - Arbeitet 3,5 Milliarden Mikrobefehle pro Sekunde ab
- Details → VL Rechnerarchitektur

Hauptspeicher (= Arbeitsspeicher)

- Durchnummerierte Speicherzellen à 8 Bit
- Enthält typischerweise Daten und Anweisungen
- Bit 1 oder 0 ist elementare Einheit
  - Strom bzw. Ladung vorhanden oder nicht
- Einheiten
  - 8 Bit bilden ein Byte
  - 4 Byte, also 32 Bit bilden die *Wortlänge*
    - Grundverarbeitungsdatengröße auf einem 32-Bit-Rechner
    - Bus- und Registerbreite
    - Der Inhalt eines Worts – also 4 Speicherzellen auf einmal – ist mit einem Maschinenbefehl verarbeitbar
  - Kilobyte (KB) sind $1024 = 2^{10}$ Byte
  - Megabyte (MB) sind 1024 Kilobyte ($= 2^{20}$ Bytes)
  - Gigabyte (GB) sind 1024 Megabyte ($= 2^{30}$ Bytes)
- Heute übliche Größe 8 GB
- Neu: Norm der SI-Einheiten (Système international d'unités)
  - Hier werden 2er-Potenzen mit KiB, MiB, usw. bezeichnet
  - Ohne „i“ 10er Potenzen
    - Z. B. ein GB sind dann $10^3 = 1000$ MB statt 1024 MB
    - Festplatten/SSDs werden heutzutage mit diesen Einheiten verkauft
  - Bezeichnung hat sich aber bei den Informatikern noch nicht durchgesetzt
    - Im Folgenden bezeichnen also Einheiten ohne „i“ die 2er-Potenzen

Festplatte

- Organisation



- *Spuren* übereinander *Zylinder*
  - Durch konkreten Schreib-/Lesekopf unterscheidbar
- Elementare Einheit *Block*
- Adressierung durch Angabe von Zylinder, *Kopf*, *Sektor* und Blocknummer
- Dateien belegen in der Regel mehrere Blöcke
- Heute übliche Größe 4 TB
  (1 TB = 1024 GB = 2^{40} Bytes)
- Verbundsysteme aus mehreren Festplatten haben viele Terabytes
- Heute auch Solid State Disks (SSD) mit Flash-Speicher < 1 TB
  - Komplett ohne mechanisch bewegliche Teile

Betriebssystem

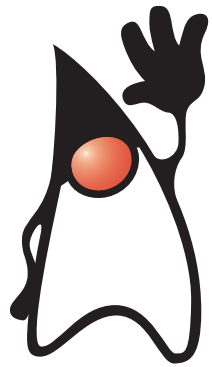
- Abstraktion von „Low-Level Maschinenoberfläche“
- Komfortable Programmausführung, meist mit graphischer Oberfläche
  - Programm wird auf Klick in den Speicher geladen und ausgeführt
- Dateisystemverwaltung auf der Festplatte
  - Abstraktion von Zylinder, Köpfen, Sektoren, Blöcken (CHS) bzw. logischer Blockadressierung (LBA)
  - Inhaltsverzeichnis für Dateien (z. B. FAT, MFT, Inodes)
  - Komfortable Zugriffswerkzeuge
- Linux, Android, Solaris, Windows, MacOS, ...

Programme in Maschinensprache

- Vorteile
  - Schnelle Programme
  - Gute Ausnutzung der Hardwaremöglichkeiten
- Nachteile
  - Mühsame, langsame und fehlerträchtige Erstellung
  - Komplexe Programme unmöglich
  - Nicht portabel auf andere Maschinen
- Lösung: *Hochsprachen*
  - Mächtigere Befehle, z. B. „Zeichne eine Linie“
  - Für den Menschen gut verständliche Sprache
  - Automatische Übersetzung in eine für die CPU verständliche Sprache (*Maschinensprache*)
- Noch besser: *domänenspezifische Sprachen*
  - Nur für einen speziellen Anwendungsbereich gedacht und einsetzbar
  - Enthalten Konzepte der Anwendung, nicht des Rechners
  - In der Regel nicht „imperativ“
- Literatur für Interessierte: H.-P. Gumm, M. Sommer, *Einführung in die Informatik*, 9. Auflage, Oldenburg Verlag, 2011, 80+17/ST 110 G974(9)

1. Java

Historie



- 1991 entwickelt von Sun Microsystems, Inc.
  - Codename *Oak* (Object Application Kernel)
  - James Gosling, Bill Joy, Guy Steele, u. v. a. m.
- Durchbruch erst mit der Verbreitung des *World Wide Web* (WWW)
- 1995 von Sun als JAVA vorgestellt (inkl. Quellcode)
- 1996 Freigabe der Version 1.02 des JAVA Development Kit (JDK)
- Vorgänger
  - Smalltalk, Eiffel, C, C++
- 1998 JAVA 2 mit Swing
- 2010 Oracle Corporation übernimmt Sun
- Juli 2011 JDK 7.0
  - http://www.oracle.com/technetwork/java/javase/downloads/
- Es gibt auch andere JAVA Umgebungen/Compiler
  - OpenJDK
  - Apache Harmony
  - IBM JAVA Developer Kit (jikes)
  - gcj aus der GNU-Compiler-Kollektion (gcc)
  - ecj der Entwicklungsumgebung Eclipse
  - Dalvik VM in Android

Ausgaben von JAVA

- JAVA Standard Edition (JAVA SE)
  vorwiegend für Client-Anwendungen auf PCs und Workstations
  - Das benutzen wir
- JAVA Enterprise Edition (JAVA EE)
  für Unternehmensanwendungen auf der Basis von Client-Server-Architekturen
- JAVA Micro Edition (JAVA ME)
  für kleine Geräte wie Handys und für Embedded Systems

Programmiersprachliche Merkmale von JAVA

- General purpose
  - Eine universelle Programmiersprache
- Plattformunabhängig
  - Konkrete Hardware und konkretes Betriebssystem soll nicht entscheidend sein
- Applikation, Applet oder Servlet
  - Eigenständiges Programm, im Browser oder auf einem Server lauffähig
- Strikt getypt, statische Typprüfung
- Basiert auf Klassen
- Objektorientiert
- Sicher
  - Sandbox
  - Gut geeignet für Internetanwendungen
- Kommunizierend
  - Streams (IO), Events
- Parallele Abläufe
  - Threads
- Automatische Speicherverwaltung
- Ausnahmebehandlung
  - Strukturierte Behandlung von Laufzeitfehlern
- Umfangreiche Bibliotheken

Hallo Welt

- Traditionelles Beispiel

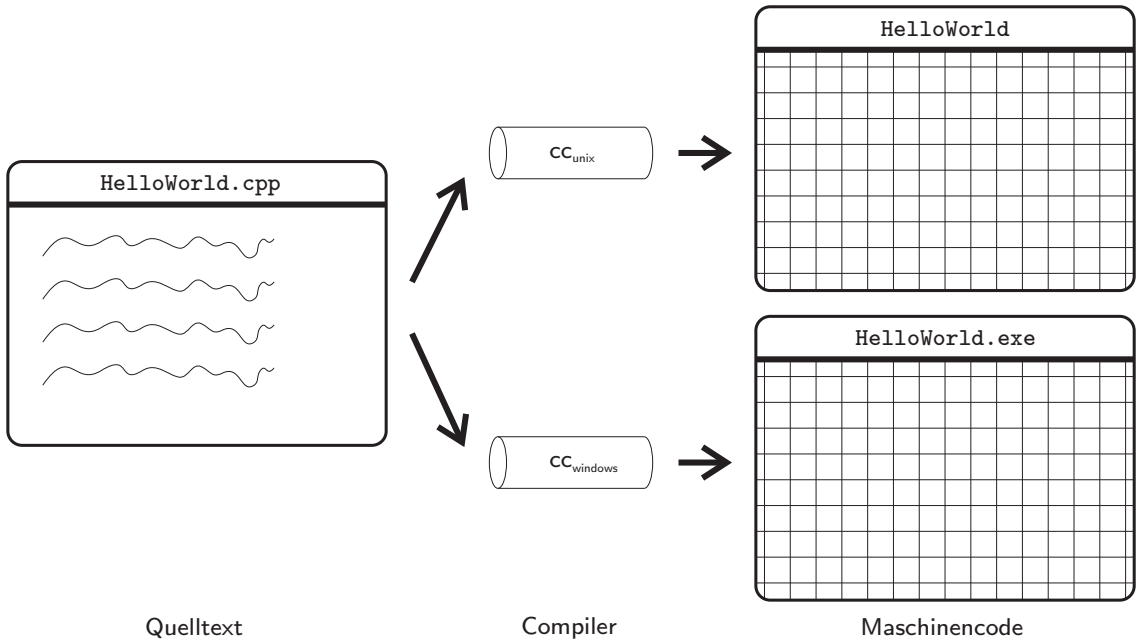
```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello, World!");
  }
}
```

Programm

- Datei muss HelloWorld.java heißen
- Erstellung
  - Editor `xemacs` von `http://www.xemacs.org/`
  - Integrierte Entwicklungsumgebung `eclipse` von `http://www.eclipse.org/` (mindestens Version 3.7.1 notwendig)
- Groß- und Kleinschreibung relevant, d. h. `println` ≠ `Println`
- Gerüst für eigene Programme
- Programm besteht aus
  - *Deklarationen*
  - *Anweisungen*
  - *Kommentaren*

Compiler

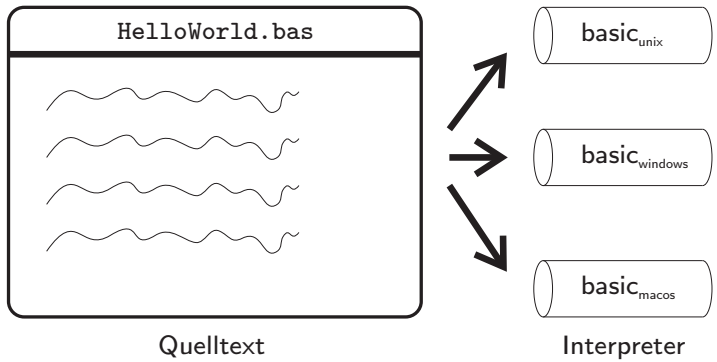
- Ein *Compiler* übersetzt kompletten Quelltext in eine Folge von Maschinenbefehlen



- Vorteile
  - Schnelle Programme
  - Direkt ausführbar
  - Programm wird vorher auf Fehler geprüft
- Nachteile
  - Programme laufen nur auf der Plattform für die sie erstellt worden sind
  - Portierung notwendig

Interpreter

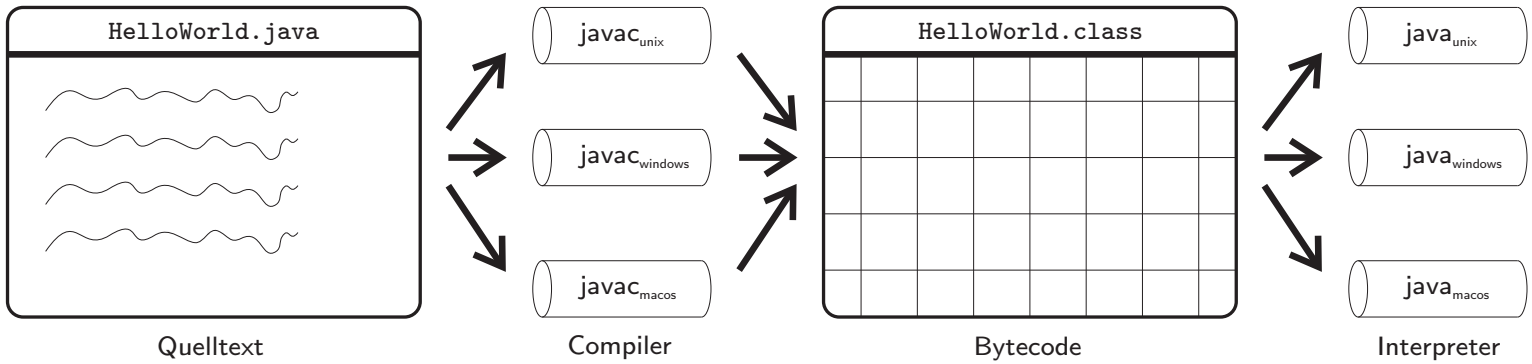
- Ein *Interpreter* übersetzt zur Laufzeit immer nur eine Programmanweisung in ein kleines Unterprogramm aus Maschinenbefehlen und führt dieses sofort aus



- Vorteile
  - Keine explizite Übersetzung notwendig
- Nachteile
  - Langsam
  - Zur Ausführung notwendiger Interpreter meist nicht für jede Plattform verfügbar
  - Portierung meist notwendig

Java

- Kompromiss durch Kombination



- Vorteile
  - Platformunabhängigkeit
  - Standardbibliothek (darf in jeder Laufzeitumgebung vorausgesetzt werden)
    - *Application Programming Interface* (API)
  - Bytecode muss nicht auf der Plattform ausgeführt werden, auf der er erzeugt wurde
  - Paradigma „write once – run everywhere“
  - Schnelle Übersetzung
  - Interpreter (= Java *Virtual Machine*/JVM) und meist auch Compiler für alle relevanten Plattformen verfügbar
    - Performance durch in JVM eingebauten „Just In Time“ (JIT) Compiler
    - Bytecode-Optimierung während der Laufzeit
  - Einfachheit, Sicherheit
  - ...

Übersetzung und Ausführung

- `javac HelloWorld.java`
  - Im Verzeichnis wo Datei gespeichert wurde
  - Erzeugte Datei `HelloWorld.class` enthält Bytecode
  - Bei mehreren java-Dateien im Verzeichnis auch `javac *.java`
- `java HelloWorld`
  - Ohne Endung .class !
  - Alternativ Classpath
    `java -cp /pfad/zum/Verzeichnis/der/class-Datei HelloWorld`
- Eclipse hat dazu eine graphische Oberfläche
- Bei mehreren Dateien
  - Buildtool
  - Apache Ant von `http://ant.apache.org/`

Kommentare

- Verbesserung der Lesbarkeit für den Menschen
- Immer notwendig
- Einzeilig
  - Alle Zeichen hinter `//` bis zum Zeilenende werden ignoriert
  - `System.out.println("Hello, World!");  // greetings`
- Mehrzeilig
  - Alle Zeichen zwischen `/*` und `*/` werden ignoriert
  - ```
    public static void main(String[] args) {
      /*
      System.out.print("Hello, ");
      System.out.println("World!");
      */
    }
    ```
  - Schachtelung nicht zulässig
    ```

Kommentare. . .

- Dokumentation z. B. in html oder pdf
  - ```
    /**
     * This is our first Java program.
     *
     * @author Chris
     * @version 0.2
     * @since 0.1
     */
    public class HelloWorld {
      /**
       * This is the main method where the program execution starts.
       *
       * @param args Parameters given on the command line.
       * @see java.io.Printstream#println
       */
      public static void main(String[] args) {
         System.out.println("Hello, World!");
      }
    }
    ```
  - Details http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html
  - Aufruf `javadoc *.java` im Verzeichnis mit den Quelldateien
  - API-Dokumentation http://download.oracle.com/javase/7/docs/api/
    ```

Anweisungen und Deklarationen

- Anweisungen (= *Statement*)
  - Stellen kleinste ausführbare Einheiten eines Programms dar
  - Elementare Anweisungen durch ; beendet

    ```
    System.out.println("Hello, World!");
    ```
  - Mehrere Anweisungen können durch { und } zu einer *Blockanweisung* (= *Sequenz*) zusammengefasst werden

    ```
    {
      System.out.print("Hello, ");
      System.out.println("World!");
    }
    ```
    - Semantik: ein Block wird als eine (große) Anweisung gesehen
    - Kein ; hinter } notwendig
  - Leere Anweisungen ; oder { } sind zulässig
- Deklarationen (= spezielle Anweisungen) siehe unter Datenstrukturen

Einlesen von Eingaben

- Grundsätzlich werden von Tastatur nur Zeichenketten eingelesen
- Falls Zahl gewünscht ist Umwandlung nötig
- Gerüst

```
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class Input {
  public static void main(String[] args) {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    try {
      String s = in.readLine();
      System.out.println("You have entered: " + s);
      int i = Integer.parseInt(s);
      System.out.println("which is an integer.");
    } catch (Exception e) {
      System.out.println("Invalid entry.");
    }
  }
}
```

  - Details später bzw. in der API-Dokumentation

2. Datenstrukturen

Basisdatentypen (= *primitive Datentypen*)

- Arithmetik

Typ	Länge in Bytes	Wertebereich
byte	1	-2^7 bis 2^7-1 ($-128 \ldots 127$)
short	2	-2^{15} bis $2^{15}-1$ ($-32768 \ldots 32767$)
int	4	-2^{31} bis $2^{31}-1$ ($-2147483648 \ldots 2147483647$)
long	8	-2^{63} bis $2^{63}-1$ ($-9223372036854775808 \ldots 9223372036854775807$)
float	4	$\pm$(1.40239846E–45f ... 3.40282347E+38f) (Gen. ca. 7 Stellen), 0.0f
double	8	$\pm$(4.9406564584124654 4E–324 ... 1.7976931348623157 0E+308) (Gen. ca. 15 Stellen), 0.0

  - Darstellung wie gewohnt nach üblichen Konventionen
  - Formal exakt (bei Zweideutigkeit explizite Typangabe)

`int`	`10`, `-5`, `1_000`, `0xA` (hexadezimal), `032` (octal), `0b11010` (binär)
`long`	`10l`, `10L`, `1234_5678_9012_3456L`, `0b11010010_01101001_10010100_10010010L`
`float`	`1f`, `1.2f`, `.12E1f`, `12.e-1F`
`double`	`.5`, `1.`, `1.2`, `12e-1`, `0.12d`, `0.12D`, `1.001_352`

  - Trennzeichen _ zur besseren Lesbarkeit (nur) zwischen zwei Ziffern zulässig
  - Interne Speicherung im *Zweierkomplement*

Basisdatentypen (= *primitive Datentypen*)...

- Symbol

Typ	Länge in Bytes	Wertebereich
char	2	16-Bit Unicode Zeichen (\u0000 ... \uffff)

  - Alle Zeichen auf der Tastatur z. B. 'a', '2'
  - Nationale Sonderzeichen z. B. 'ä', 'Ü', 'ß'
  - Alle Zeichen können auch über Nummer angesprochen werden
    - Notwendig weil nicht alle Zeichen auf der Tastatur vorhanden sind
    - 4-stellig hexadezimal (im 16er System)
    - Z. B. 'a' = '\u0061'
    - Vollständige Tabelle(n) http://www.unicode.org/
    - Ausgabeschriftart muss graphische Zeichenrepräsentation (= *Glyph*) enthalten!
  - *Escapezeichen* z. B.

'\t'	'\u0009'	Tabulator
'\n'	'\u000a'	Zeilenvorschub
'\b'	'\u0008'	Backspace
'\r'	'\u000d'	Return (Waagenrücklauf)
'\\'	'\u005c'	Backslash
'\''	'\u0027'	Apostroph
'\"'	'\u0022'	Anführungszeichen
	'\u0000'	Nullzeichen ≠ Whitespace ' '

  - Interne Speicherung binär-kodiert in UTF-16

Basisdatentypen (= *primitive Datentypen*)...

- Logik

Typ	**Länge in Bytes**	**Wertebereich**
boolean	1	`true` oder `false`

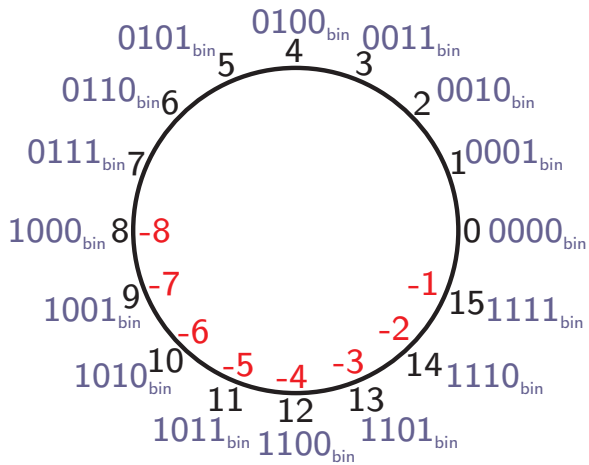
- Referenz (später)

Interne Repräsentation von ganzen Zahlen

- Ann: Register/Speicherzelle kann $b = 4$ Bits speichern
- Darstellung im Binärsystem
  - $10 = 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 = 1010_{\text{bin}}$
- Kein explizites Vorzeichen mgl. (nur 0 und 1)
  - Höchstwertiges (linkestes) Bit bestimmt das Vorzeichen: 0 pos. und 1 neg.
  - Konsequenz: Darstellung von 10 nicht mehr mgl.
- *Einerkomplement*
  - Bitkomplement des Absolutbetrags
  - $+5 = 0101_{\text{bin}}$
  - $-5 = 1010_{\text{bin}}$
  - Wertebereich $[-2^{b-1} + 1; 2^{b-1} - 1]$
  - Vorteil: Wertebereich symmetrisch
  - Nachteil: zwei Nullrepräsentationen $0 = 0000_{\text{bin}} = 1111_{\text{bin}}$

Interne Repräsentation von ganzen Zahlen...

- *Zweierkomplement*
  - Bitkomplement des Absolutbetrags (= Einerkomplement) und anschl. +1
  - $-5 = 1010_{\text{bin}} + 1_{\text{bin}} = 1011_{\text{bin}}$



  - Wertebereich $[-2^{b-1}; 2^{b-1} - 1]$
  - Vorteil: einfache Subtraktion durch gewöhnliche Addition mit dem negierten Wert des Subtrahenden
    - Überlauf-Bit wird einfach verworfen
    - $4 + (-4) = 0100_{\text{bin}} + 1100_{\text{bin}} = \not{1}0000_{\text{bin}} = 0$
  - So macht es JAVA
- Fließkommazahlen werden intern mit Mantisse m und Exponent e jeweils im Zweierkomplement zur Basis 2 repräsentiert
  - $d = m \cdot 2^e$
  - IEEE 754 Standard (http://de.wikipedia.org/wiki/IEEE_754)

Variablen

- Veränderliche Werte aus dem zulässigen Bereichs eines Datentyps
- *Deklaration* (= spezielle Anweisung) notwendig
  - Typ Name;
    - `int i;`
    - `char c;`
- *Zuweisung* (*Assignment*, = spezielle Anweisung) eines Wertes an eine Variable
  - Neuer Wert kann beliebiger *Ausdruck* (= *Expression*) sein
    - Alles was einen Wert hat oder zu einem ausgewertet werden kann
    - Z. B. auch eine Variable alleine, ein Literal, oder eine Funktion
  - Ein *Literal* ist eine Repräsentation eines Wertes von primitivem Typ
  - `i = 1;`
  - `int j; j = 42; i = j;`
  - `i = j * 7;`
  - `c = 'a';`
- Deklaration mit *Initialisierung*
  - `int i = 1; char c = 'a';`
- Achtung: keine automatische Initialisierung von (lokalen) Variablen
  - Compilefehler bei Zugriff auf möglicherweise nicht initialisierte Variable
- Bei gleichem Typ gemeinsame Deklaration mgl.
  - `int i = 0, j;`

Bezeichner (= Identifikatoren)

- Selbstdefinierte Namen aller Art
- (Muss-)Konventionen für Bezeichner (z. B. Variablen)
  - Große und kleine Buchstaben inkl. Umlaute
  - Zahlen außer an erster Stelle (`value12`)
  - Sonderzeichen nur Unterstrich und Dollarzeichen (`$COLOR_VALUE`)
  - Keine Leerzeichen
- Dringende Empfehlung: „sprechende“ Namen
  - Erhöhen Lesbarkeit von Programmen (für Menschen ☺)
- Reservierte Wörter

  ```
  abstract assert boolean break byte byvalue case cast catch char class const
  continue default do double else enum extends false final finally float for
  future generic goto if implements import instanceof int inner interface long
  main native new null operator outer package private protected public rest
  return short static strictfp super switch synchronized this throw throws
  transient true try var void volatile while
  ```

  - Nicht alle davon werden momentan tatsächlich auch benutzt
- Zusätzliche (Soll-)Namenskonventionen für Variablen
  - Kleingeschriebenes Substantiv
  - Namensbestandteile mit Großbuchstaben hervorheben (= *CamelCase*)
    - Z. B. `giroBankAccount`
  - → JAVA Code Conventions

Gültigkeitsbereich von Variablen (*Scope*)

- Grundsätzlich nur in dem Block wo sie und nachdem sie deklariert wurden und dessen Unterblöcke nach der Deklaration

```
{
  int i = 1;
  {
    i = 2;
    int j = 3;
    System.out.println(i + j);  // 5
  }
  System.out.println(i);        // 2
  System.out.println(j);        // compile error: j cannot be resolved
}
```

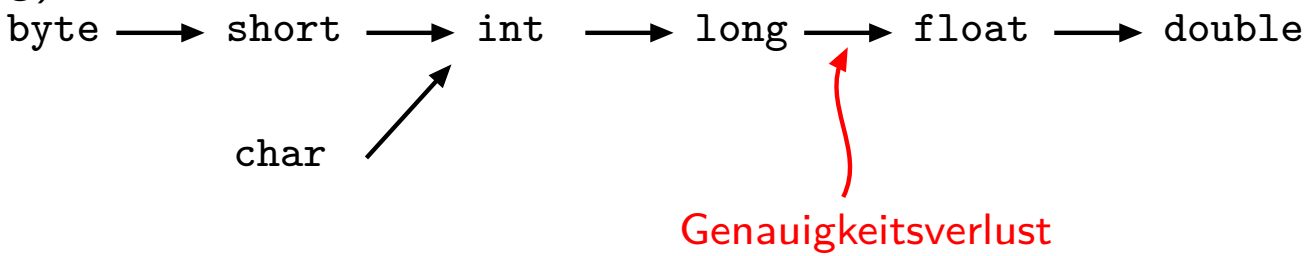
- Variablen mit gleichem Namen deklariert in einem Unterblock sind verboten

```
{
  int i = 1;
  {
    double i;  // compile error: duplicate local variable
  }
}
```

  - Ausnahme: Überdecken von Klassen-/Instanzvariablen (später)

Typkonversion

- JAVA ist *typstreng*
  - Zuweisung nur von gleichen/kompatiblen Typen
  - `int i = 10L;  // error: cannot convert from long to integer`
- Aber *implizite Konvertierung*
  - `short s = 3; int i = s;`
  - `double x = 1 / 3.0;  // x = 0.333333333333333 by double division`
  - `double y = 1 / 3;  // y = 0.0 by int division`
  - (*Up-casting*) Hierarchie



  - `long l = 1234567899; float f = l;  // f = 1.23456794E9`
  - Umgekehrt Compilefehler als Warnung für potentiellen Datenverlust

Typkonversion...

- *Explizite Konvertierung* (*cast*, *down-cast*)
  - `double d = 1.55; int i = (int) d;  // i = 1`
  - `float f = (float) d;  // f = 1.55`
  - `double x = (float) 1 / 3;  // x = 0.3333333 mit float-Division`
  - Vorsicht: bei *Überlauf* kein Programmabsturz, es wird mit falschem Wert weitergerechnet!
    - `short s = 257; byte b = (byte) s;  // b = 1`
    - Höherwertiges Byte wird einfach abgeschnitten
      $257 = 2^8 + 2^0 = 0\text{b}\underbrace{00000001}\underbrace{00000001}_{\texttt{b}}$

Konstanten

- Unveränderliche Werte
- Deklaration durch vorangestelltes Schlüsselwort `final` (*Modifier*)
  - `final int ONE = 1;`
- Wertezuweisung nur durch Initialisierung
- (Soll-)Namenskonvention für Konstantenbezeichner
  - Substantiv
  - Alle Buchstaben groß
  - Namensbestandteile durch Unterstrich hervorheben
    - `MAX_INT`, `NULL_VALUE`

Einschub: Zeichenketten

- Kein einfacher Datentyp
- Aneinanderreihung von Symbolen
- `String s = "Hello, World!";`
- Doublequotes anstatt Singlequotes wie bei Symbolen

Einfache Ausdrücke

- Literale, Konstanten und Variablen durch *Operationen* zusammensetzen
- Assoziativität (wenn Klammerung fehlt)
  - Linksassoziativ $1/2/4 = (1/2)/4 = 0,125 \neq 1/(2/4) = 2$
  - Rechtsassoziativ $a \mathrel{/}= b \mathrel{/}= c$ entspricht $a \mathrel{/}= (b \mathrel{/}= c)$ und nicht $(a \mathrel{/}= b) \mathrel{/}= c$
- Arithmetische und logische Operatoren (linksassoziativ)

# Operatoren	arithmetisch	logisch
unär	+ −	!
	++ −− (gleichzeitig Zuweisung)	
	~	
binär	* / %	&
	+ −	^
		|
	<< >> >>>	
	&	&&
	^	
	|	||

  - Typen von Operatoren wie Typ der/des allgemeinsten Operanden
    - Operatoren sind *überladen*

Einfache Ausdrücke...

- Relationale Operatoren
  `<` `>` `>=` `<=` `==` `!=`
  - Typ: `boolean`
  - Vergleichsoperator == nicht mit Zuweisungsoperator = verwechseln!
- Bedingter Ausdruck (ternärer Operator)
  logischer Ausdruck ? true-Rückgabewert : false-Rückgabewert
  - `int a = 3, b = 5; int c = a > b ? 20 : 10;  // c = 10`
  - Typ entspricht einheitlichem Typ der Rückgabewerte
- Zuweisungsoperatoren/Verbundoperatoren (rechtsassoziativ)
  `=` `+=` `-=` `*=` `/=` `%=` `>>=` `<<=` `>>>=` `&=` `^=` `|=`
  - `x <op>= y` Kurzform für `x = x <op> y`

Einfache Ausdrücke...

- Besonderheiten
  - Präfix- und Postfix-Inkrement bzw. -Dekrement
    −− ++
    - `b = <op>a` Kurzform für `a = a ± 1; b = a`
    - `int a = 2; int b = ++a;  // a = b = 3`
    - `int a = 2; int b = ++a * 3;  // a = 3, b = 9`
    - `b = a<op>` Kurzform für `b = a; a = a ± 1`
    - `int a = 2; int b = a++;  // b = 2, a = 3`
    - `int a = 2; int b = a++ * 3;  // b = 6, a = 3`
  - Überladener + Operator für Strings
    - `String a = "Hello, ", b = "World!"; String c = a + b;`
      `// c = "Hello, World!"`
  - Ganzzahlige Division / und Modulo %
    - $8/3 = 2$, $-8/3 = -2 \neq -3$, $8/-3 = -2$, $-8/-3 = 2 \neq 3$
    - $8\%3 = 2$, $-8\%3 = -2 \neq 1$, $8\%-3 = 2$, $-8\%-3 = -2 \neq 1$
    - Allgemein: $a\%b = a - (a/b) * b$

Einfache Ausdrücke...

- Bitweise Verknüpfungen

a	b	$\sim a$	$a\&b$	$a\|b$	$a\hat{}b$
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

  - $\sim 10 = \sim 0b01010 = 0b\underbrace{1\ldots1}_{27}10101 = -11$ (führende Nullen werden invertiert!)
  - $10 \,\&\, 7 = 0b1010 \,\&\, 0b0111 = 0b0010 = 2$
- Logische Verknüpfungen

a	b	$!a$	$a\&b$	$a\|b$	$a\hat{}b$
`false`	`false`	`true`	`false`	`false`	`false`
`false`	`true`	`true`	`false`	`true`	`true`
`true`	`false`	`false`	`false`	`true`	`true`
`true`	`true`	`false`	`true`	`true`	`false`

  - Bei Verkettung mit && oder || werden nur notwendige Teile ausgewertet
    - *Verkürzte Auswertung* von links nach rechts
  - Vorsicht: Zuweisungen `i = 5` werden nie ausgeführt
    - `boolean b1 = false && 5 == (i = 5);`
    - `boolean b2 = true || 5 == (i = 5);`
  - Mit Operatoren zur *vollständigen Auswertung* schon
    - `boolean b1 = false & 5 == (i = 5);`
    - `boolean b2 = true | 5 == (i = 5);`

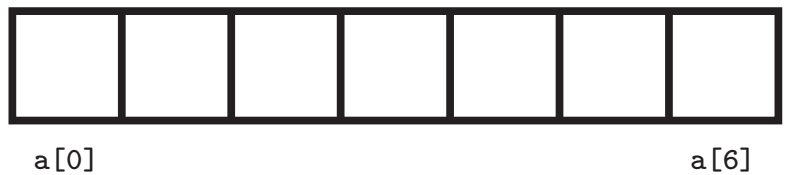
Einfache Ausdrücke...

- Bitweises Verschieben
  - $8 << 2 = 0b1000 * 2^2 = 0b100000 = 32$ „zieht von rechts Nullen nach“
  - $8 >> 2 = 0b1000/2^2 = 0b0010 = 2$ „zieht von links Nullen nach“
  - $-8 >> 2 = 0b11000/2^2 = 0b11110 = -2$ „zieht von links Einsen nach“
    - Negative Zahlen beginnen mit 1 (zumindest im höchstwertigen Bit)
    - $-7 >> 2 = \lfloor 0b11001/(\text{double})\ 2^2 \rfloor = 0b11110 = -2$
    - *Bitshift* nach rechts macht die (vgl. zu JAVA) andere ganzahlige Division
  - $-8 >>> 2 = 0b\underbrace{1\ldots1}_{29}000 >>> 2 = 0b00\underbrace{1\ldots1}_{29}0 = 1073741822$

    „zieht von links Nullen nach“
- Unäre Cast-Operatoren (siehe Typkonversion)
- Präzedenzen
  - Arithmetik vor Vergleich vor Logik vor Zuweisung
    - `boolean b = x + 1 < 10 & x >= 5;`
    - `boolean b = (((x + 1) < 10) & (x >= 5));  // false with x = 2`
  - Unär vor Punktrechnung vor Strichrechnung
    - `-1 * x++ + 3;`
    - `((-1) * (x++)) + 3;  // 1 with x = 2`
  - 13 Präzedenzklassen für stärkere Bindung
- Klammerung (`Ausdruck`) verändert Prioritäten
  - `(2 + 4) * 5;  // = 30 ≠ 22`
  - Erhöht ggf. die Lesbarkeit!

Arrays

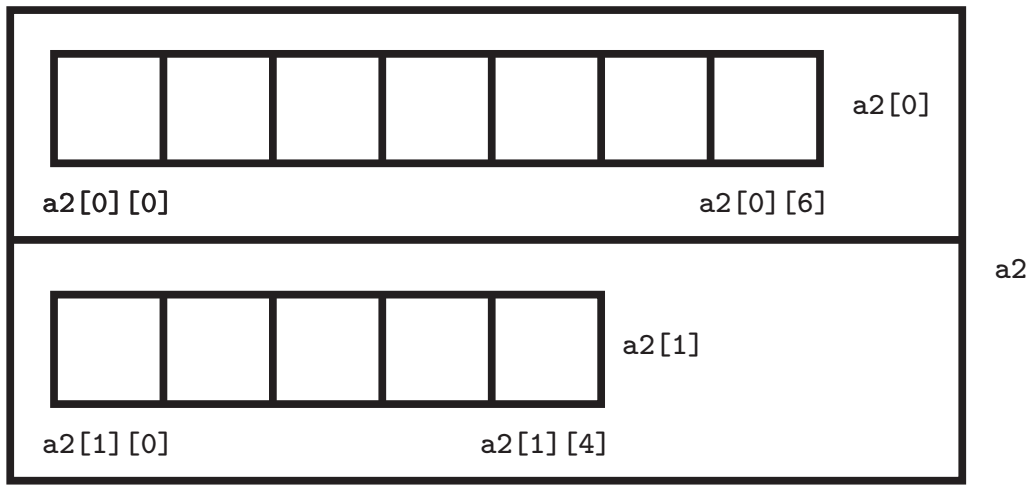
- Einfachste zusammengesetze Datenstrukturen
- Zusammenfassung mehrerer Werte eines Typs zu einem neuen Typ
- Deklaration eines Arrays, welches 7 Zahlen aufnehmen kann
  - `int[] a = new int[7];`



  - `int[] a; a = new int[7];`
- Initialisierung
  - `char[] c = new char[3]; c[0] = 'a'; c[1] = 'b'; c[2] = 'd';`
  - `char[] c = {'a', 'b', 'd'};`
  - `char[] c = new char[] {'a', 'b', 'd'};`
    - Diese Schreibweise ist überall wo ein Array verlangt wird benutzbar
    - Nicht nur zur Initialisierung wie hier
- Abfrage der Größe mit `a.length` `// 7`
- Zugriff
  - Elemente von 0 bis $n-1$ durchnummeriert (n = length)
  - `c[0] = 'z'; System.out.println(c[1]);` `// b`
  - `int i = 8; System.out.println(c[i / 4]);` `// d`
  - Klammeroperator kann bel. Ausdruck der Typ `int` liefert beinhalten

Arrays. . .

- Mehrdimensionale Arrays
  - Zusammenfassung von Arrays zu Arrays
  - Deklaration
    - `int[][] a2; a2 = new int[2][];`
      `a2[0] = new int[7]; a2[1] = new int[5];`



  - 2D Array mit uniformer Länge der „Sub-Arrays“ entspricht einer Matrix
    - `int[][] m = new int[2][7];  // 2x7 matrix`
  - Dimension nicht begrenzt
  - Initialisierung
    - `int[][] x = {{1, 2, 3}, {4, 5}};`
    - `int[][] x = new int[][] {{1, 2, 3}, {4, 5}};`
  - Zugriff
    - `x[0][2] += x[1][1]; ++x[0][2]; System.out.println(x[0][2]);  // 9`

Tausch/Swap

- Vertauschen der Inhalte von `a[i]` und `a[j]`
  - int clip = a[i]; a[i] = a[j]; a[j] = clip;
  - Hilfsvariable `clip` notwendig
- Auch beim Vertauschen der Inhalte zweier normaler Variablen
  - `int a = 0, b = 1, clip; clip = a; a = b; b = clip;`
- Alternative in-place mit exklusivem Oder (xor)

```
a = a ^ b;  // 1
b = a ^ b;  // 2
a = a ^ b;  // 3
```

  - Korrektheit folgt aus Rechenregeln für xor
    1. $a = a \hat{} b$
    2. $b \overset{1.\text{ in }2.}{=} (a \hat{} b) \hat{} b \overset{\text{Ass.}}{=} a \hat{} (b \hat{} b) \overset{\text{Inv.}}{=} a \hat{} 0 \overset{\text{Id.}}{=} a$
    3. $a \overset{1.\text{ in }3.}{=} (a \hat{} b) \hat{} b \overset{2.\text{ in }3.}{=} (a \hat{} b) \hat{} a \overset{\text{Kom.}}{=} a \hat{} (a \hat{} b) \overset{\text{Ass.}}{=} (a \hat{} a) \hat{} b \overset{\text{Inv.}}{=} 0 \hat{} b \overset{\text{Id.}}{=} b$

    - Nachteil a und b müssen verschiedene Variablen (mit unterschiedlichem Speicherplatz) sein
    - Sonst „nullt“ erstes xor a und b
    - Wert darf aber gleich sein
  - Variante mit `clip`-Variable schneller (und klarer!)
    - Keine Prüfung auf $a \neq b$ notwendig
    - Besser optimierbar (parallelisierbar) für den Compiler

3. Kontrollstrukturen

Bedingte Verzweigung

- Beeinflussung der normalen sequentiellen Ausführungsreihenfolge
  - Bis jetzt nur Anweisung für Anweisung von oben nach unten
  - Ablaufsteuerung
- Einfache Verzweigung

```
if ( logischer Ausdruck )
   Anweisung   // if this expression is true
else if ( logischer Ausdruck )
   Anweisung   // if this expression is true and previous ones were false
 ⋮
else
   Anweisung   // if every previous expression was false
```

- Anweisung kann einzelne Anweisung oder Blockanweisung sein
- Es darf beliebig viele Sonst-Wenn-Teile geben
- Der Sonst-Teil (alles ab einschließlich else) kann auch weggelassen werden
- Beachte
  - Kein `then`
  - Abschluss der *Zweige* mit ; außer bei Blockanweisung
  - Logische Ausdrücke müssen in Klammern ( ) stehen

Bedingte Verzweigung...

- Beispiel

```
int a = 2, b, c = -1;

if (a >= 0) {
  System.out.println("a is greater than or equal to 0");
  b = 1;
} else {
  System.out.println("a is smaller than 0");
  b = -11;
}

if (b == 1)
  if (c > 0)
    System.out.println("c is greater than 0");
  else
    System.out.println("c is smaller than or equal to 0");
```

Bedingte Verzweigung...

- Problem „dangling else“
  - ```
    int i = 1;
    if (i <= 0)
      if (i == 0) System.out.println("i is zero");
    else System.out.println(i);
    ```
  - `else` wird an das „nächste“ `if` gebunden
    - Hier also keine Ausgabe!
  - Innere Bindung stärker als äußere
  - Gegenmaßnahme Blockanweisungen, d. h. Klammerung

    ```
    int i = 1;
    if (i <= 0) {
      if (i == 0) {
        System.out.println("i is zero");
      }
    } else {
      System.out.println(i);  // 1
    }
    ```
  - Empfehlung: immer Klammern verwenden!
    ```

Auswahl

- Mehrfache Verzweigung

```
switch ( Ausdruck ) {
case Konstante/Literal :
   Anweisungen
   break;
case Konstante/Literal :
   Anweisungen
   break;
...
default :
   Anweisungen
}
```

- Ausdruck kann nicht alle Typen liefern, sondern nur
  - `char` oder `Character` (später)
  - `byte`, `short`, `int`, `long` oder `Byte`, `Short`, `Integer`, `Long` (später)
  - `String`
    - Vergleich hier nicht mit == sondern mittels `equals`-Methode (später)
  - `Enum` (später)
- Wenn einer der case-Vergleiche zutrifft, werden die Anweisungen aller nachfolgenden case-Fälle ohne Fall-Prüfung abgearbeitet (= *Fall-Through*)
  - `break;` jeweils als letzte Anweisung unterbricht diesen (Oder-)Mechanismus

Auswahl

- Default-Teil
  - Optional
  - Soll nicht den letzten Fall abdecken sondern „alles andere“
  - Als (zusätzliche) Sicherheit dass kein Fall vergessen wurde

Auswahl...

- Beispiel

```
char choice = 's';
switch (choice) {
case 'n' :
case 'N' :
  System.out.println("Option new");
  break;
case 'o' :
case 'O' :
  System.out.println("Option open");
  break;
case 's' :
case 'S' :
  System.out.println("Option save");
  break;
case 'q' :
case 'Q' :
  System.out.println("Option quit");
  break;
default :
  System.out.println("Invalid option");
}
```

  - Ohne `break` Ausgabe `Option save\nOption quit\nInvalid option`
  - Besser `switch (Character.toLowerCase(choice)) { ... }` verwenden

Auswahl...

- Beispiel 2: case-Ausdrücke

```
int digit = ...;
final int ONE = 1;
int three = 3;
switch (digit) {
case ONE:
  System.out.println("One");     // constant ok
  break;
case ONE + 1:
  System.out.println("Two");     // constant expression ok
  break;
case three:
  System.out.println("Three");   // compile error: no constant expression
  break;
...
default:
  System.out.println("Not a digit");
}
```

Schleifen

- Abweisende Schleife (= kopfgesteuerte Schleife = vorprüfende Schleife)

  `while (` `logischer Ausdruck` `)`

  `Anweisung`

  - Solange `logischer Ausdruck` erfüllt ist führe `Anweisung` aus
- Nicht-abweisende Schleife (= fußgesteuerte Schleife = nachprüfende Schleife)

  `do`

  `Anweisung`

  `while (` `logischer Ausdruck` `);`

  - Führe `Anweisung` solange aus wie `logischer Ausdruck` erfüllt ist
  - `Anweisung` wird in jedem Fall mindestens einmal durchlaufen
  - Semantisch äquivalent zu

    `Anweisung A`

    `while (` `logischer Ausdruck` `)`

    `Anweisung A`
- `Anweisung` meistens Blockanweisung
  - Ansonsten Abschluss mit ;

Schleifen...

- Anweisung ist *Schleifenrumpf* (= *Body*), Rest *Schleifenkopf*
- Beispiel

```
int[] numbers = new int[5];
int i = 0, sum = 0;

do {
  numbers[i] = i;
  ++i;
} while (i < 5);

while (i > 0) {
  --i;
  sum += numbers[i];
}
System.out.println(sum);  // 10 = 4 + 3 + 2 + 1 + 0
```

Schleifen...

- Beispiel mit Schachtelung

```
int line = 1;
while (line <= 5) {
  int star = 1;
  while (star <= 2 * line) {
    System.out.print("*");
    ++star;
  }
  System.out.println();
  ++line;
}
```

  - Ausgabe

```
**
****
******
********
**********
```

Schleifen...

- Wichtig: Terminierung, Laufzeit, Abstieg (wie oft wird Schleife durchlaufen)
  - Unendlich laufende Schleifen heißen *Endlosschleife*
  - `while (true) ; // empty body`
- Sichere (Schleifen-)*Abbruchbedingung*
  - Überlauf generierende Aufzählschleife bei `value > 9`

```
int i = value;
while (i != 9) { ++i; }
```

    - Robustere Variante

```
int i = value;
while (i < 9) { ++i; }
```

  - Berücksichtigung von evtl. Rechenungenauigkeiten

```
double d = 0.0;
while (d != 1.0) {
  d += 0.1;
  System.out.print(d + " ");
}
// 0.1 0.2 0.30000000000000004 0.4 0.5 0.6 0.7 0.7999999999999999
// 0.8999999999999999 0.9999999999999999 1.0999999999999999 1.2 ...
```

    - Bessere Variante testet auf `(d >= 0.99999 && d <= 1.00001)`
    - Immer mit ϵ-Umgebung testen (nicht nur bei Schleifen)

Schleifen...

- Zählschleife

  `for (` `Initialisierung` `;` `logischer Ausdruck` `;` `Zuweisung` `)`
  `Anweisung`

  - `Initialisierung` wird anfangs einmal ausgeführt
  - Solange `logischer Ausdruck` erfüllt ist wird `Anweisung` ausgeführt
  - Nach jeder Anweisungsausführung wird `Zuweisung` ausgeführt
  - Eine Variable deklariert in `Initialisierung` ist im Schleifenkopf und -rumpf gültig, nicht außerhalb

    ```
    for (int i = 0; i < 10; ++i) {
      System.out.println(i);
    }
    System.out.println(i);  // error: i cannot be resolved
    int i = 20;             // valid
    ```

  - `i` kann uneingeschränkt im Rumpf gebraucht werden
    - Auch links bei einer Zuweisung
    - In vielen anderen Sprachen verboten

Schleifen...

- Zählschleife...
  - Semantisch äquivalent zu

```
Initialisierung ;
while ( logischer Ausdruck ) {
  Anweisung
  Zuweisung ;
}
```

- Beispiel

```
int sum = 0;
for (int i = 1; i <= 100; ++i) {
  sum += i;
}
System.out.println(
  "The sum of 1 to 100 is " + sum);
```

```
int sum = 0;
int i = 1;
while (i <= 100) {
  sum += i;
  ++i;
}
System.out.println(
  "The sum of 1 to 100 is " + sum);
```

Schleifen...

- Beispiel mit Array

```
int[] values = {3, 1, 5, 23};
int[] squares = new int[values.length];
for (int i = 0; i < values.length; ++i) {
  squares[i] = values[i] * values[i];
}
for (int i = 0; i < squares.length; ++i) {
  System.out.print(squares[i] + " ");
}
// 9 1 25 529
```

  - Wie sieht (bessere) Version ohne unnützes hinteres Leerzeichen aus?
  - Wie sieht Version mit `while`- anstatt `for`-Schleifen aus?

- Beispiel mit Schachtelung (verbesserte Version)

```
for (int line = 1; line <= 5; ++line) {
  for (int star = 1; star <= 2 * line; ++star) {
    System.out.print("*");
  }
  System.out.println();
}
```

Schleifen...

- `Zuweisung` bei for-Schleife kann aus mehreren Anweisungen bestehen
  - Durch Komma getrennt
  - Ohne Strichpunkt
  - ```
    for (int j = 0, i = 1;
         i < 5 && j < 4;
         ++i, System.out.print(j + " "), j = 3) ;  // empty body
    // 0 3 3 3
    ```
  - Variablendeklarationen allerdings nicht erlaubt
    ```

Schleifen...

- Beispiel mit 2D-Array

```
final int SIZE = 5;
int[][] values = new int[SIZE][SIZE];

for (int i = 0; i < SIZE; ++i) {
  for (int j = 0; j < SIZE; ++j) {
    values[i][j] = (i + j) * i;
  }
}

for (int i = 0; i < SIZE; ++i) {
  for (int j = 0; j < SIZE; ++j) {
    System.out.print(values[i][j]);

    // separating spaces except after last values of lines
    if (j < SIZE - 1) {
      System.out.print(" ");
    }
  }
  System.out.println();
}
```

- Ausgabe

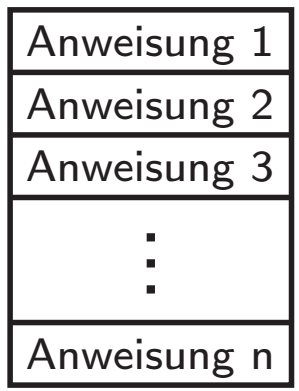
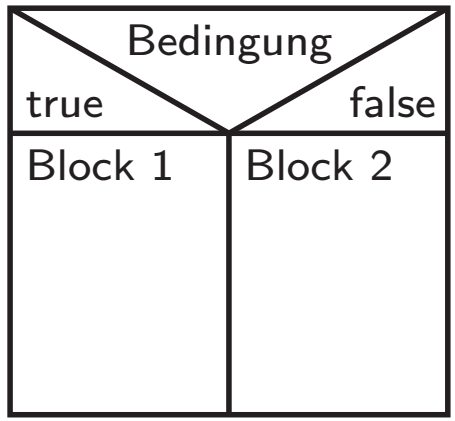
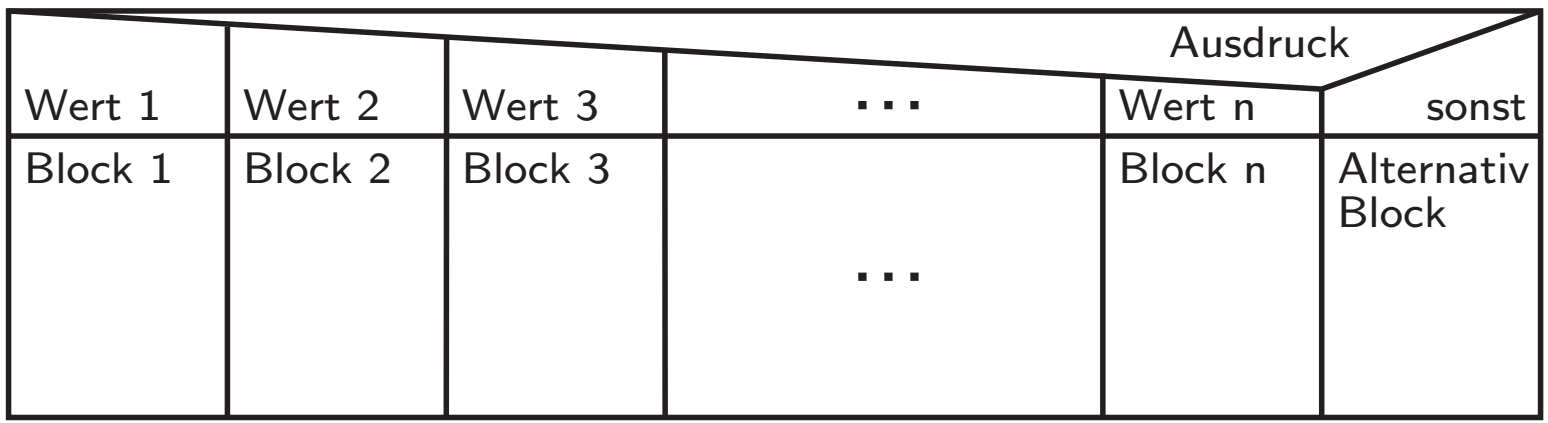
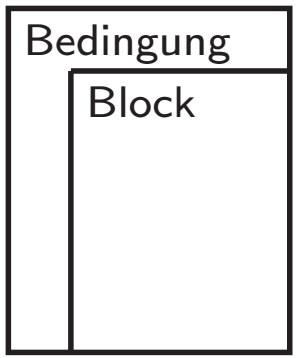
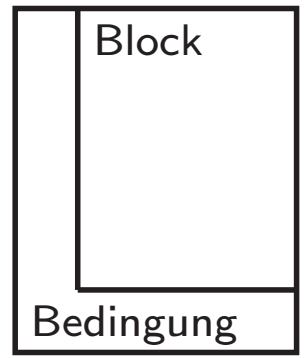
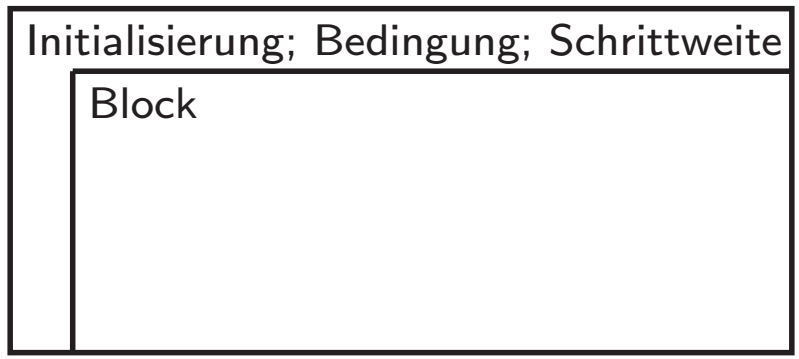
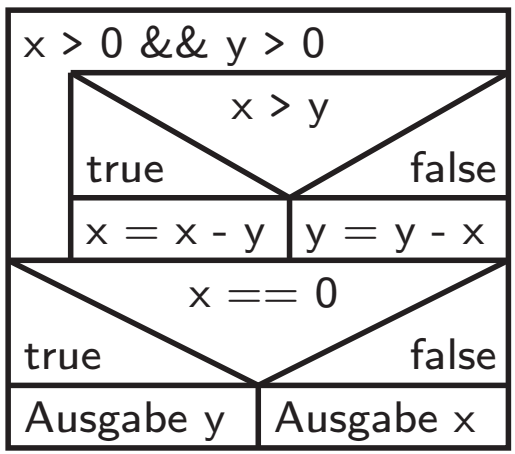
```
0 0 0 0 0
1 2 3 4 5
4 6 8 10 12
9 12 15 18 21
16 20 24 28 32
```

Schleifen...

- *Foreach*-Schleife
  - ```
    int[] a = {1, 2, 3, 4, 5};
    for (int elem : a) {
      System.out.print(elem + " ");
    }
    // output 1 2 3 4 5
    ```
  - Äquivalent zu

    ```
    for (int i = 0; i < a.length; ++i) {
      System.out.print(a[i] + " ");
    }
    ```
  - Iteration vorwärts durch das komplette Array
    - Implizite Zählvariable beginnt immer bei 0, schreitet in 1er Schritten fort und endet immer bei `a.length - 1`
  - Oft Vorteil wegen kurzer und prägnanter Schreibweise
  - Kein schreibender Zugriff auf durchlaufenes Array über Durchlaufvariable möglich

    ```
    for (int elem : a) {
      elem *= 2;
    }
    System.out.println(a[3]);  // 4
    ```

    - `elem` ist eine Kopie des Inhalts der aktuellen Zelle von `a`
    ```

Schleifen...

- `for`- statt `while`-Schleife (aber nur dann)
  - Abbruchbedingung darf nicht erst am Ende des Bodies feststehen
    - Sonst `do-while`-Schleife
  - Zählen mit einer Zählvariable
  - Alle drei Ausdrücke im Schleifenkopf beziehen sich auf dieselbe Variable
  - Zuweisungen an Zählvariablen tauchen nicht im Body auf
  - Terminierung meist leichter zu sehen/garantieren
  - Falls mgl. foreach-Schleife

Sprunganweisungen

- Vorzeitige Beendigung eines Schleifendurchlaufs
- `break`
  - ```
    int i = 10;
    while (true) {
      if (i-- == 0) {
        break;
      }
      System.out.println("Number: " + i);
    }
    ```
  - Zahlen von 9 bis 0
  - Schleife wird so lange ausgeführt, bis $i = 0$ (einschl.)
- `continue`
  - ```
    for (int i = 0; i <= 10; ++i) {
      if (i % 2 == 1) {
        continue;
      }
      System.out.println("Number: " + i);
    }
    ```
  - Gerade Zahlen von 0 bis 10
  - Überspringt nachfolgende Anweisungen im Rumpf der Schleife
- Beziehen sich auf lokalste umgebende Schleife
  - Innere Bindung ist stärker als äußere
- Sollten beide vermieden werden

Struktogramme

- Verdeutlichung der Struktur eines Programms mittels Diagramm
  - Unabhängig von der konkreten Programmiersprache
  - Liefert graphische Sicht auf die Kontroll- und Blockstruktur
  - Meistens vergröbert mit sprechenden Anweisungen
  - Hilft konkrete Aufgabenstellung in ein Programm umzusetzen
- Struktogramm (= Nassi-Shneidermann-Diagramm = DIN 66261)
  - Weitgehend aus dem allgemeinen Gebrauch verschwunden
  - Evtl. für Programmieranfänger als Zwischenschritt hilfreich beim Umsetzen einer Aufgabe in konkreten Code
- Sequenz / linearer Ablauf



Struktogramme...

- Verzweigung / `if`



  - `else if`-Zweig muss simuliert werden
  - Durch Einsetzen einer weiteren Verzweigung in Block 2

- Fallauswahl / `switch`



Struktogramme...

- Abweisende Schleife / `while`



- Nicht-abweisende Schleife / `do while`



- Zählschleife / `for`



  - Graphisch wie normale abweisende Schleife
  - Nur kompletter Schleifenkopf mit durch ; getrennte Bestandteile

Struktogramme...

- Beispiel: größter gemeinsamer Teiler zweier Zahlen mit dem Euklidischen Algorithmus
  - Struktogramm



  - JAVA-Code

```
while (x > 0 && y > 0) {
  if (x > y) {
    x -= y;
  } else {
    y -= x;
  }
}
if (x == 0) {
  System.out.println(y);
} else {
  System.out.println(x);
}
```

4. Programmstrukturen

Funktionen

- Blöcke mit Namen und Rückgabe
- ```
  public class Programmname {
    Funktionsdeklaration {
      Funktionsdefinition
    }
    ⋮
    public static void main(String[] args) {
      Hauptprogramm
    }
  }
  ```
- *Kopf* (= *Signatur* = Funktionsdeklaration)
- *Rumpf* (= Funktionsdefinition)
- Erwartet Eingabewerte (*Parameter* = *Argumente*) und gibt *Rückgabewert* zurück

  ```
  public static Rückgabetyp Funktionsname(Argumente) {
    Unterprogramm
    return Ausdruck;
  }
  ```
  ```

Funktionen...

- Argumente (*formale Parameter*) werden wie Variablen deklariert

  `Typ` `Name`, `Typ` `Name`, ...

  - Hinter letzter Deklaration steht kein Komma
  - Kann auch leer sein
  - Abkürzende Schreibweise `int a, b, String s` nicht erlaubt
- Eine Funktion ist eindeutig durch ihren Namen und die Argumente definiert
  - Wenn nur Argumente unterschiedlich *Überladung*
  - Nur verschiedener Rückgabetyp reicht nicht
  - Innerhalb einer Klasse keine „doppelten“
- *Lokale* Variablen
  - Innerhalb einer Funktion (innerhalb eines Blocks) definiert
  - Werden bei Beginn der Funktions- bzw. der Blockausführung angelegt
    - Bei der Deklaration (nicht vorher)
  - Zerstörung beim Verlassen
  - Formale Parameter sind lokale Variablen
    - Werden beim Funktionsaufruf initialisiert

Funktionen...

- Funktionsaufruf (*Invoke*)
  - Mit typkompatibler Belegung der Parameter (*aktuelle Parameter*)
  - ```
    public class AverageComputation {
      public static double average(double a, double b) {
        double two = 2.0;
        return (a + b) / two;
      }

      public static void main(String[] args) {
        double avg = average(2.5, 2.7);  // 2.6
        double avgOf4 = average(average(1.0, 2.0), average(3.0, 4.0)); // 2.5
      }
    }
    ```
  - Auch hier implizite Up-Casts

    ```
    public static void main(String[] args) {
      int a = 2, b = 3;
      double avg = average(a, b);  // 2.5
    }
    ```
  - Ergebnis einer Funktion muss nicht verwertet werden
  - Nach Abarbeitung wird das Programm an der Aufrufstelle fortgesetzt
    ```

Funktionen...

- Jede Funktion hat einen Funktionswert
  - Schlüsselwort `return` zur Rückgabe des Funktionswerts
  - Mit `return` `Wert` ; endet Funktionsausführung
    - Oft als letzte Zeile einer Funktion
    - Kann aber auch vorzeitig aus einer Funktion springen
    - Darf mehrfach vorkommen
    - (Primitiver) Rückgabewert wird kopiert (analog zu Call-by-Value)
    - Deshalb kann problemlos lokale Variable zurückgegeben werden
  - Jeder mögliche Durchlauf muss mit einem `return` enden
    - Verboten

```
public static int produceCompileError(int i) {
  if (i % 2 == 0) { return 0; }
  else { System.out.println("i is odd"); }
}
```

    - Funktion verhält sich wie Wert (ist ein Ausdruck)
    - Wert darf nicht undefiniert sein
    - Kein unerreichbarer Code
- (Soll-)Namenskonvention (Bezeichner)
  - kleingeschriebenes Verb
  - CamelCase
  - Sonst wie Variablen

Parameterübergabe

- *Call-by-Value*
  - ```
    public static int pseudoIncrease(int i) {
      i = i + 1;
      System.out.println("i = " + i);  // 6
      return i;
    }

    public static void main(String[] args) {
      int i = 5;
      System.out.println("i = " + i);  // 5
      int j = pseudoIncrease(i);
      System.out.println("j = " + j);  // 6
      System.out.println("i = " + i);  // 5
    }
    ```
  - Variablen werden kopiert
    - Aktuelle Parameter in die formalen Parameter
    - Für jedes Argument wird Variable des deklarierten Typs mit dem übergebenen Wert initialisiert
    - Wert (*Aktionsparameter*) der Variablen im Aufruf bleibt im Gegensatz zu Argument-Variablen (Formalparameter) unverändert
  - Nur einfache Datentypen
    - Arrays (zumindest deren Inhalt im Speicher) werden nicht kopiert
    ```

Parameterübergabe...

- *Call-by-Reference*
  - ```
    public static int[] increase(final int[] a) {
      for (int i = 0; i < a.length; ++i) {
        ++a[i];
      }
      System.out.println("a[3] = " + a[3]);   // 5
      return a;
    }

    public static void main(String[] args) {
      int[] a = {1, 2, 3, 4, 5};
      System.out.println("a[3] = " + a[3]);   // 4
      int[] b = increase(a);
      System.out.println("b[3] = " + b[3]);   // 5
      System.out.println("a[3] = " + a[3]);   // 5
    }
    ```
  - Auch `final` hilft nicht gegen Inhaltsveränderung
    - Primitiver Typ bzw. hier Referenz konstant
    - Vereinfachte Vorstellung: a speichert nur wo Array im Speicher beginnt
  - Evtl. ungewollte Seiteneffekte
    - Nutzer kann der Funktion das von außen nicht ansehen
    - Inhaltsveränderungen mit Bedacht einsetzen und dokumentieren
    ```

Variable Parameteranzahl (*Varargs*)

- Beispiel

```
public static int output(int j, String... args) {
  for (int i = 0; i < args.length; ++i) {
    System.out.print(args[i] + " ");
  }
  System.out.println();
  args[0] = "modified";  // only local
  return 0;
}

public static void main(String[] args) {
  String s = "Hello,";
  output(10, s, "World!");
  System.out.println(s);  // Hello,
}
```

- Erlaubt nicht beschränkte Anzahl an Parametern für eine Funktion
- Alle (Var-)Parameter müssen den gleichen Typ haben
  - Innerhalb der Funktion als lokales Array behandelt
  - Nur Inhalte und kein Array übergeben: Call-by-Value (bei primitiven Typen)
- Gleichzeitige Verwendung mit normalen Parametern
  - Nur einmal und als letztes Argument
  - Ansonsten Trennung zwischen Varargs und normalen Parametern nicht klar

Variable Parameteranzahl (*Varargs*)...

- Auch Übergabe eines Arrays zulässig

```
public static void main(String[] args) {
  String[] words = {"Hello,", "World!"};
  output(10, words);
  System.out.println(words[0]);  // modified
}
```

  - Arrays werden mit Call-by-Reference übergeben

- Konsistente Vorstellung: (anonymes) Array schon beim Aufruf zu sehen

```
output(10, new String[] {s, "World!"});
```

  - Änderung nur im Array
  - `s` bleibt unverändert

Prozeduren

- „Funktionen ohne Rückgabewert“
- Nur Seiteneffekte, Gruppierung oder Wiederverwendbarkeit interessant
  - Grundsatz der Wiederverwendbarkeit: kein doppelter Code
- Rückgabetyp `void`
  - Heißt „kein Rückgabetyp“
  - Nicht „irgendetwas“ nicht näher spezifiziertes wie z. B. `void*` in C
- ```
  public static void printArrayInOneLine(int[] a) {
    if (a.length > 35) { return; }  // too long to fit into one line
    for (int i = 0; i < a.length - 1; ++i) {
      System.out.print(a[i] + ", ");
    }
    System.out.println(a[a.length - 1]);
    return;
  }
  public static void main(String[] args) {
    int[] a = {1, 2, 3, 4, 5};
    printArrayInOneLine(a);
  }
  ```
- Auch hier kann es (mehrere) `return`s geben, aber ohne Parameter
  - Ist aber kein Muss
  - Ohne wird nach letzter Anweisung zum aufrufenden Code zurückgesprungen
  ```

Prozeduren...

- Parameterübergabe wie bei Funktionen
- *Methode*
  - Oberbegriff für Funktion oder Prozedur in einer Klasse
  - Reihenfolge innerhalb der Klasse unerheblich

Klassenvariablen

- Deklaration außerhalb von Methoden
- Innerhalb der Klasse
  - Reihenfolge unerheblich
  - Per Konvention vor den Methoden
- Für alle Methoden der Klasse *global* verfügbar
- `public static` `Deklaration (optional mit Initialisierung)` `;`

Klassenvariablen...

- Seiteneffekte

```
public class SideEffects {
  public static int i = 5;
  ⋮
  public static void printString(String s) {
    System.out.println(s);
    i = 0;
  }

  public static void main(String[] args) {
    System.out.println("10 / i = " + 10 / i);  // 2
    printString("Hello, World!");
    System.out.println("10 / i = " + 10 / i);  // crash
  }
}
```

  - Lokale Variablen in Methoden oder Parameter meist besser

Sichtbarkeit von Klassenvariablen

- Lokale Variablen mit gleichem Namen *verdecken* (= *verschatten*) globale Klassenvariablen

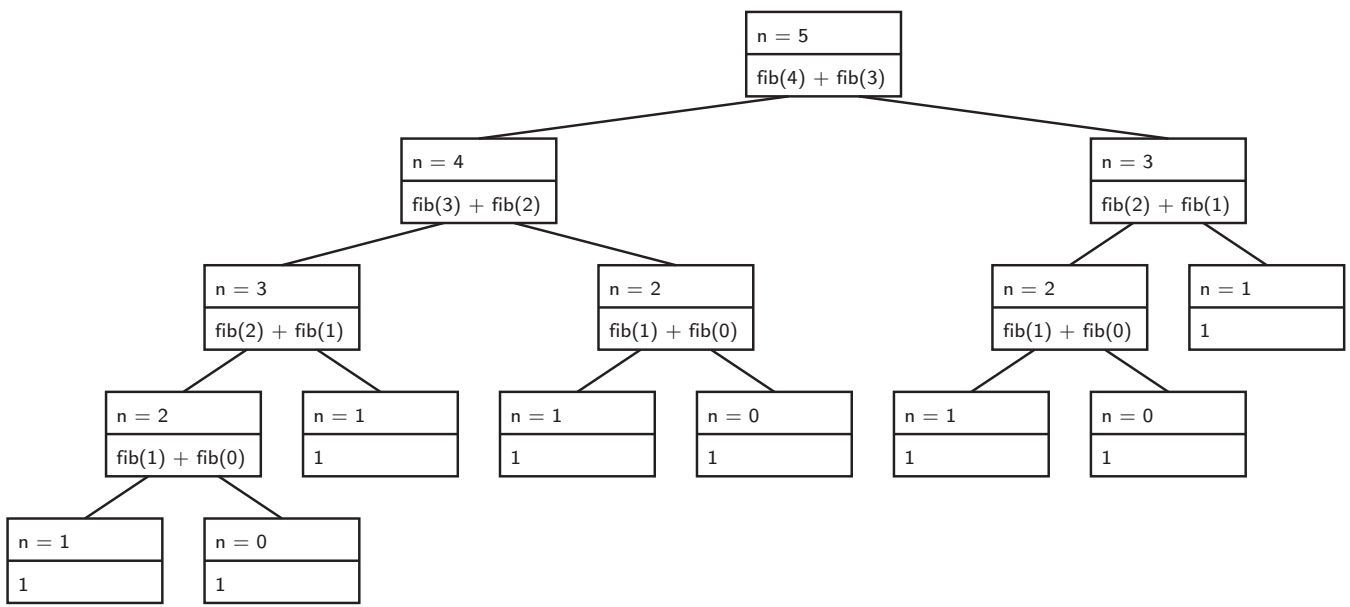
```
public class HideAndSeek {
  public static int i = 1;

  public static void increase() {
    ++i;
    System.out.println(i);  // 2
    int i = 4;
    ++i;
    System.out.println(i);  // 5
  }

  public static void main(String[] args) {
    System.out.println(i);  // 1
    increase();
    System.out.println(i);  // 2
  }
}
```

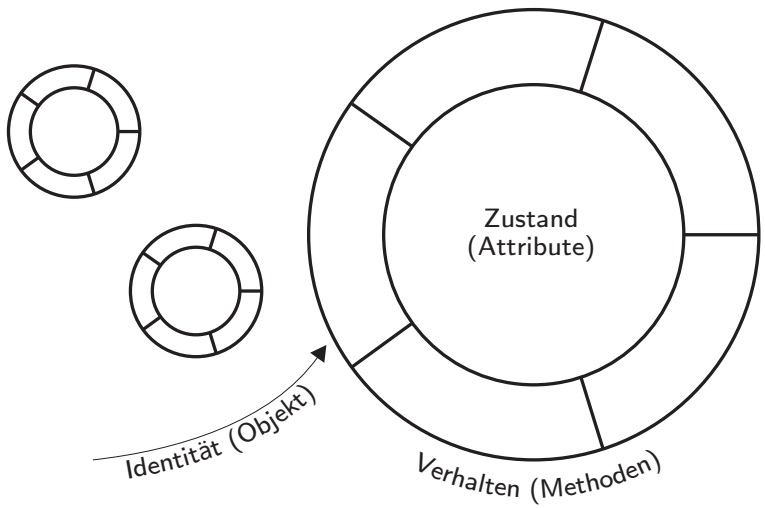
Sichtbarkeit von Klassenvariablen...

- Zugriff auf verdeckte globale Variable
  - `Klassenname`.`Variablenname`
  - ```
    public class HideAndSeek2 {
      public static int i = 1;

      public static void increase(int i) {
        ++i;
        System.out.println(i);                // 5
        ++HideAndSeek2.i;
        System.out.println(HideAndSeek2.i);   // 2
      }

      public static void main(String[] args) {
        int i = 4;
        increase(i);
        System.out.println(i);                // 4
        System.out.println(HideAndSeek2.i);   // 2
      }
    }
    ```
  - Hier 3 unterschiedliche Variablen `i` mit je eigenem Speicherbereich
- Auch formale Parameter verdecken Klassenvariablen mit gleichem Namen
    ```

Rekursion

- Methoden rufen sich selber (*elementar*) oder gegenseitig (*verschränkt*) auf

```
public class BinomialCoefficient {
  public static int numberOfCalls = 0;

  public static int binomial(int n, int k) {
    return (int) (fac(n) / (fac(k) * fac(n - k)));
  }

  public static long fac(int n) {  // 21! > 2^63
    ++numberOfCalls;
    if (n < 2) {
      return 1;
    } else {
      return n * fac(n - 1);  // recursion
    }
  }

  public static void main(String[] args) {
    System.out.println("6 choose 3: " + binomial(6, 3));           // 20
    System.out.println("called fac " + numberOfCalls + " times");  // 6+3+3
  }
}
```

Rekursion...

- Rekursive Funktion zur Berechnung der Fibonacci-Zahlen

```
public static int fib(int n) {
  if (n < 2) {
    return 1;
  } else {
    int res = fib(n - 1) + fib(n - 2)
    return res;
  }
}
```

  - Aufruf von `fib(5)` initialisiert neue Variable `n` vom Typ `int`
  - Zur Berechnung werden `fib(4)` und `fib(3)` aufgerufen, die wiederum neue Variablen `n` initialisieren, usw.



Rekursion...

- Jeder Aufruf von `fib` hat zwei weitere Aufrufe zur Folge
  - $A(n)$ Anzahl der Aufrufe generiert von `fib(n)`
  - $A(n) = 1$, falls $n < 2$
  - $A(n) = A(n-1) + A(n-2) + 1$, falls $n > 1$
- Per Induktion gilt $A(n) \geq 2^{\lfloor n/2 \rfloor}$

  $n = 0, 1, 2$: $A(0) = A(1) = 1 \geq 2^0$, $A(2) = A(1) + A(0) + 1 \geq 2^1$

  $n \to n+2$:
  $$A(n+2) = A(n+1) + A(n) + 1 > 2A(n) \underset{\text{I.V.}}{\geq} 2 \cdot 2^{\lfloor n/2 \rfloor} = 2^{\lfloor n/2 \rfloor + 1} = 2^{\lfloor (n+2)/2 \rfloor}$$
  - Laufzeit (Anzahl der Aufrufe) der rek. Funktionsauswertung von `fib` steigt exponentiell mit der Größe der Eingabe n
  - Damit auch der Speicherverbrauch
    - Grund: Zwischenergebnisse werden hier mehrfach berechnet

Rekursion...

- Iterative Methode

```
public static int fib(int n) {
  int pen = 1, ult = 1, res = 1;
  for (int i = 2; i <= n; ++i) {
    res = pen + ult;
    pen = ult; ult = res;
  }
  return res;
}
```

  - Anzahl der Schritte hängt nur von der Anzahl an Schleifendurchläufen ab
  - Linear in der Größe der Eingabe n
  - Speicherverbrauch konstant und klein
  - Für $n = 100$ macht rek. Funktion $2^{50} \approx 1.1 \cdot 10^{15}$ („Billiarden“) Schritte, Schleife hingegen nur 99
  - Noch Fragen?

Rekursion...

- Zugegeben: es gibt auch eine effiziente rekursive Variante von `fib`
  - Aber nicht ganz so offensichtlich

```
public static int fib(int n) {
  return n == 0 ? 1 : fibRec(n, 1, 1);
}

public static int fibRec(int n, int pen, int ult) {
  return n == 1 ? ult : fibRec(n - 1, ult, pen + ult);
}
```

- Rekursion nicht gleichzusetzen mit Ineffizienz
  - Kann sehr nützlich sein!
  - Meistens sehr elegante und kurze Programme
  - Aber man muss sich über Laufzeit, Terminierung, Abstieg und Konvergenz im klaren sein

Einschub: Übergabe von Kommandozeilenargumenten

- Argumentübergabe an ein JAVA Programm „von außen“
  - Argumente schon vor dem Programmstart klar
  - Keine Abfrage während des Programmablaufs mit `BufferedReader` nötig
- Analog zur Parameterübergabe bei Methoden
- Syntax eines JAVA-Programms
  - `public static` (Beginn einer Methodendeklaration) `void` (kein Rückgabewert) `main` (standardisierter Methodenname) `(String[] args)` (Argumentdeklaration)
  - JAVA Interpreter ruft obligatorische Methode `main` auf
  - Argument `String`-Array
- Aufruf
  - `java` `Programmname` `args[0]` `args[1]` `...`
  - Argumente durch Leerzeichen getrennt
  - Falls Argument Leerzeichen enthält `"..."` verwenden (*quoten*)
- `String` kann wie bei Benutzereingabe in einen `int`-Wert umgewandelt werden
  - `public static int Integer.parseInt(String str)`

Einschub: Übergabe von Kommandozeilenargumenten...

- Beispiel

```
public class CmdLineArgsPrinter {
  public static void main(String[] args) {
    for (int i = args.length - 1; i >= 0; --i) {
      System.out.print(args[i]) + "\t";
    }
    System.out.println();
  }
}
```

  - `java CmdLineArgsPrinter World! "Hello, "`
  - Ausgabe: `Hello, World!`
- Übergabe von Kommandozeilenparamter in Eclipse
  - Run → Run Configurations... → (x)= Arguments → Program arguments

5. Benutzerdefinierte Datenstrukturen

Objektorientierte Programmiersprachen

- Erweiterung des Konzepts strukturierter imperativer Programmiersprachen
  - Programmstrukturierung
  - Verbesserung/Vereinfachung des Entwurfs
  - Erhöhung der Wiederverwendbarkeit von Programmen bzw. Programmteilen
- Fast alle (strukturierten imperativen) Programmiersprachen erlauben Definition eigener Datentypen
  - Komposition bereits vorhandener Datentypen
  - Z. B. Array (nur Variablen gleichen Typs), Record, Struct
- Variablen zusammengesetzen Typs
  - Gebunden an den Wert (Inhalt) des für sie reservierten Speicherbereichs
  - Beschreiben *Zustand*
    - Inhalt der enthaltenen Variablen bzw. des enthaltenen Speichers
  - Haben eindeutige *Identität*
    - Vereinfacht: Adresse der ersten belegten Speicherzelle
- Bisher
  - Kapselung häufiger benötigter oder funktionell abgeschlossener Abläufe in *Subroutinen* (= Funktionen, Prozeduren)

Klassen

- Zusammenfassung mehrer Werte (auch unterschiedlichen Typs) und Funktionalität zu einem neuen *abstrakten Datentyp* (*Klasse*)
- Kapselung konkret
  - (Instanz-)Variablen (*Attribute*) und Konstanten
  - Routinen (*Methoden*)
  - Jetzt jeweils ohne Schlüsselwort `static`
- Variablen dieses Datentyps (*Objekte*, *Instanzen*, *Exemplare*)



  - Beschreiben Zustand definiert durch die Werte der Attribute
  - Haben Verhalten definiert durch die Methoden
    - „Aktive“ Objekte statt „passive“ Daten
    - Jedes Objekt besitzt die Fähigkeit zu speichern und zu rechnen
  - Haben Identität

Deklaration einer Klasse

- Angabe der Bauanleitung/Konstruktionsvorschrift
- `public class` **`Name`** `{`

  `Attributdeklaration`

  ⋮

  `Methodendeklaration`

  ⋮

  `}`
- Reihenfolge Attribute dann Methoden als (Soll-)Konvention
- (Soll-)Namenskonvention für Klassen (Bezeichner)
  - Substantiv
  - Erster Buchstabe groß und CamelCase
- Jede Klasse in eigene Datei `Name.java`
- Von Methoden aus zugreifbar
  - Lokale Variablen der Methode
  - Formale Parameter der Methode
  - Attribute ihrer Objekte
  - Mit dem .-Operator auf Attribute dieser Attribute (später)

Deklaration einer Klasse...

- Klasse die ein (vereinfachtes) Bankkonto modelliert

```
public class Account {
  public int balance;

  public void deposit(int amount) { balance += amount; }

  public void withdraw(int amount) { balance -= amount; }

  public int getBalance() { return balance; }
}
```

- Verwendung des Typs `Account`
  - `Account[] savings;` speichert alle Sparkonten einer Bank
- Klassen allgemein dienen zur allgemeinen Beschreibung von Dingen (Objekten)
  - Verschiedene Ausprägungen
    - Jedes Konto hat eigenen Kontostand
  - Aber gemeinsame Struktur und Verhalten

Referenzen

- Weiterer Basisdatentyp
- „Zeiger“ auf Instanzen zugehöriger Klassen (Objekte bestimmten Typs)
  - Referenzen werden in Illustrationen als Pfeil gezeichnet
- Deklaration
  - `Klassenname` `Variablenname`;
  - `Account giroAccount1, giroAccount2;`
- Intern ist eine Referenz als Integer realisiert
  - Zeigt auf Identität des Objekts
  - Enthält Nummer (Adresse) der 1. vom Objekt belegten Speicherzelle
  - Man sagt eine Referenz(-variable) *zeigt auf*/*referenziert* ein Objekt
- `giroAccount1 == giroAccount2` vergleicht nur Referenzen nicht Zustände/Inhalte/Ausprägungen
  - Zustand der Objekte ist hier jeweils der Kontostand `balance`
- Auch Arrays (die Variablen an sich) sind Referenzen
  - Vergleich zweier Arrayvariablen vergleicht nicht Inhalt

Referenzen...

- Parameterübergabe von Objekten an Methoden ist immer Call-by-Reference
  - Deshalb der Name ☺
  - Nur Referenz an sich (der „Zeiger“) wird kopiert
    - So gesehen in JAVA Parameterübergabe eigentlich nur mittels Call-By-Value
  - Referenziertes Objekt im Speicher wird nicht kopiert!
  - Der Ausdruck eines Kontoauszugs ist nicht umsonst:

```
public void printStatement(Account a) {
  System.out.println("Your current balance is " + a.getBalance());
  --a.balance;  // charge
}
```

    - Besser gleich als Methode der Klasse `Account`
    - Anstatt `Account` als Parameter zu übergeben
    - Direkter Zugriff auf Attribute: `--balance;`
- Schlüsselwort `null` ist Referenz, die per Definition auf kein Objekt zeigt

Instanziierung

- Konstruktion eines Objekts mit dem `new`-Operator
  - `new` `Klassenname()`
  - Legt Objekt physikalisch im Speicher an
- Rückgabewert des `new`-Ausdrucks ist Referenz auf neue Instanz der Klasse
  - `Account giroAccount1 = new Account();`
- Mehrere Referenzen können auf das gleiche Objekt zeigen
  - `Account giroAccount2 = giroAccount1;`
- Zugriffsoperator `.`
  - Zugriff über Referenz auf Methode oder Attribut (*Dereferenzierung*)

```
Account account = new Account();
account.balance = 200;
System.out.println(account.getBalance());  // 200
```

- Schlüsselwort `this` innerhalb Klasse ist Referenz des Objekts (*Selbstreferenz*)
  - Entspricht Rückgabewert von `new`
  - Oft notwendig um auf verdeckte Attribute zuzugreifen
  - Zur Erinnerung: bei Klassenvariablen dient Klassenname zum Zugriff

Zustandsvergleich

- Manuell
  - `giroAccount1.getBalance() == giroAccount2.getBalance()`
  - Bei einer (Zustands-)Variablen kein Problem
  - Objekte haben i. A. aber viel mehr als eine
- Bessere Objekte können das von sich aus
- Methode `equals`

```
String s1 = "Hello, World!", s2 = "Hello, World!";
s1 == s2;       // false
s1.equals(s2);  // true
```

- Aber Klasse muss Methode `equals` enthalten (*implementieren*)

```
public class Account {
  public int balance;

  ⋮
  public boolean equals(Object other) {
    return balance == ((Account) other).balance;
  }

  public void open(int balance) {
    this.balance = balance;  // this necessary, alt. deposit(balance)
  }
}
```

Zustandsvergleich...

```
  ⋮
Account giroAccount1 = new Account();
giroAccount1.open(100);
Account giroAccount2 = giroAccount1;
giroAccount1.equals(giroAccount2);  // trivially true
giroAccount2 = new Account();
giroAccount2.open(200);
giroAccount1.equals(giroAccount2);  // false
giroAccount2.withdraw(100);
giroAccount1.equals(giroAccount2);  // true
giroAccount1 == giroAccount2;       // false
```

Konstruktoren

- Spezielle Methoden die bei Erzeugung eines Objekts aufgerufen werden
- ```
  public class Point {
    public int x, y;

    // explicit default constructor (without parameters)
    public Point() {
      x = y = 1;
    }

    public Point(int x, int y) {
      this.x = x; this.y = y;
    }
  }
  ```
- Konstruktoren haben denselben Namen wie Klasse
- Konstruktoren haben keinen expliziten Rückgabetyp
  - Rückgabetyp ist implizit die Klasse selbst und -Wert Instanz davon
- Mehrere Konstruktoren mgl. (Unterscheidung anhand Parameter/Signatur)
- Falls kein Konstruktor vorhanden
  - Compiler ergänzt automatisch leeren Standard-Konstruktor (= *impliziter Default-Konstruktor*)
  - Ohne Parameter und Anweisungen
  - Siehe Klasse `Account`
  ```

Konstruktoren...

- Erklärt Syntax des Operators `new`
  - `Point p = new Point();`
  - `Point p = new Point(10, 10);`
- Initialisierung von (Instanz-)Attributen
  - Zum Zeitpunkt des Konstruktoraufrufes
    - Genauer: vor dessen Anweisungen
  - Macht in diesem Fall dasselbe wie expliziter Default-Konstruktor

```
public class Point {
  public int x = 1, y = 1;
  ⋮
}
```

  - Wird aber immer ausgeführt, d. h. auch bei Ausführung des anderen Konstruktors
  - Im Gegensatz zu lokalen Variablen werden Attribute automatisch/implizit initialisiert
    - Falls keine explizite Initialisierung angegeben ist
    - Variablen eines Zahlendatentyps mit `0` bzw. `0.0`
    - `boolean`-Variablen mit `false`
    - Referenzen mit `null`
  - Nur schlechter Programmierstil verlässt sich darauf

Speicherverwaltung

- Elemente die mit `new` Angelegt werden befinden sich im *Heap*- (= *dynamischer* = *Frei*-) Speicher
  - Explizite Anforderung von Speicher notwendig
- Alles andere befinden sich im *Stack*-Speicher
  - Z. B. lokale Variablen und formale Parameter
  - Aufrufstack von Methoden
  - Automatische Anforderung von Speicher
  - Angeforderte Bereiche müssen in umgekehrter Reihenfolge wieder aufgegeben werden
    - Geschieht in allen Sprachen automatisch

Garbage Collection

- Operator `new` legt den für Objekte notwendigen Speicher an
- Automatischer Aufräumdienst des Heap-Speichers
  - *Garbage Collector*
  - Dursucht periodisch Speicher
  - Entfernt nicht mehr benötigte Objekte
- Was nicht mehr gebraucht wird kann genau nur der Programmierer wissen
- Aber Objekte ohne Referenz darauf können definitiv entfernt werden
  - Zugriff darauf sowieso nicht mehr möglich
- Spezielle Methode `protected void finalize()` ist Gegenstück zum Konstruktor
  - Wird vor Speicherplatzfreigabe vom Garbage Collector aufgerufen
  - Aufräumarbeiten (aber besser mit eigener Methode)
  - Aufrufzeitpunkt und ausführender Thread (und dessen Priorität) unspezifiziert
  - Besser erst gar nicht verwenden
  - In C++ gibt es anstatt dessen einen *Destruktor*
- JAVA sieht keine explizite Freigabe von Objekten vor
  - Wie z. B. `delete` in C++
  - Man kann aber Garbage Collector mit `System.gc()` anstoßen
    - Gibt aber nur einen Hinweis
    - Kein echter Aufruf

Garbage Collection...

- Vorteile
  - Man braucht sich nicht kümmern
  - Man kann i. d. R. keine Objekte vergessen („Speicherlöcher“)
    - Außer bei unnötigen Referenzen auf nicht mehr gebrauchte Objekte
    - Manchmal ist es notwendig sie auf `null` zu setzen
    - In vielen Fällen reicht aber den Gültigkeitsbereich der Variablen lokaler zu wählen
  - `finalize`-Methoden werden von einem anderen Thread ausgeführt (später)
- Nachteile
  - Performanceverlust
  - Tatsächlicher Zeitpunkt der Destruktion nicht spezifiziert und daher unbekannt
  - `finalize`-Methoden werden von einem anderen Thread ausgeführt (später)

Echte Kopien von Objekten

- Nicht nur Kopie der Referenz sondern zwei getrennte Objekte im Speicher
- In C++ Copy-Konstruktor `Rectangle::Rectangle(const Rectangle& r)`
- In JAVA Methode `public` `Typ` `clone()`
- Flach (= *Shallow Copy*)

```
public class Rectangle {
  public Point p1, p2;

  public Rectangle clone() {
    Rectangle c = new Rectangle();
    c.p1 = p1; c.p2 = p2;
    return c;
  }
}
  ⋮
Rectangle r = new Rectangle();
r.p1 = new Point(10, 10); r.p2 = new Point(50, 30);  // better by constructor
Rectangle c = r.clone();
c.p1.x = 20;
System.out.println(r.p1.x);  // 20
```

Echte Kopien von Objekten...

- Tief (= *Deep Copy*)

```
public class Rectangle {
  ⋮
  public Rectangle clone() {
    Rectangle c = new Rectangle();
    c.p1 = new Point(p1.x, p1.y);
    c.p2 = new Point(p2.x, p2.y);
    return c;
  }
}
```

Einschub: Anonyme Klassen

- Haben keinen eigenen Namen
- Werden direkt in `new`-Anweisungen eingesetzt

```
Point p = new Point(10, 12) {
  public String toString() {
    return "(" + x + ", " + y + ")";
  }
};
System.out.println(p.toString());  // (10, 12)
```

- Erweiterung der Klasse `Point` um die Methode `toString`
- Hat keine eigenen Konstruktoren
- Einziges Exemplar dieser Klasse lässt sich über die Referenz `p` ansprechen
- Braucht man hauptsächlich graphische Bedienoberflächen (AWT, SWING) um auf Ereignisse wie Mausklicks zu reagieren
  - Falls (momentan) kein triftiger Grund vorliegt keine anonymen Klassen verwenden
  - Klasse `Point` wie gehabt in eigener Datei anlegen
- Es gibt noch 3 weitere Typen sog. *innerer Klassen*

Statische Attribute und Methoden

- Schlüsselwort `static` (Modifier)
  - Kennzeichnet Attribute und Methoden
  - Unabhängigkeit von Objektidentitäten
  - Zur Klasse selbst gehörend, nicht zu ihren Instanzen
  - Nutzung ist nicht an Existenz von Objekten gebunden
  - *Klassenvariablen* und *-methoden*
- Alle Instanzen einer Klasse „teilen sich“ diese Elemente
- Zugriff (von außen) nur über Klassenname
  - `Klassenname`.`Variablenname`
  - `Klassenname`.`Methodenname(...)`
  - Als (Soll-)Konvention auch innerhalb der Klassen
- Klassenvariablen werden initialisiert beim Laden der Klasse durch das Laufzeitsystem/den *Class Loader*
- Nicht-statische Attribute sind pro Instanz vorhanden
  - Zugriff über Instanzname
  - Jede Instanz hat eigenen Wert für dieses Attribut
  - *Instanzvariablen*, *Instanzmethoden*

Statische Attribute und Methoden...

- Beispiel Instanzenzähler

```
public class InstanceCounter {
  public static int counter = 0;

  public InstanceCounter() {
    ++InstanceCounter.counter;
  }

  protected void finalize() {
    --InstanceCounter.counter;
  }
}
```

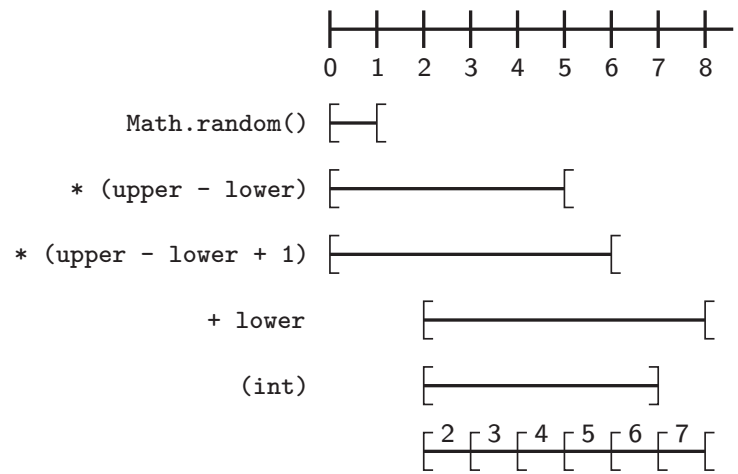
  - Verwendung

```
InstanceCounter c1 = new InstanceCounter();
InstanceCounter c2 = new InstanceCounter();
System.out.println(InstanceCounter.counter);  // 2
```

Statische Attribute und Methoden...

- Beispiel gemeinsame Verwendung von statischen und dynamischen Elementen

```
public class Numbers {
  public int i;
  public static int j;

  public void output() {
    System.out.println(i + j);  // better use FQN Numbers.j
  }
}
  ⋮
  Numbers n1 = new Numbers();
  Numbers n2 = new Numbers();
  n1.i = 1;
  n2.i = 2;
  Numbers.j = 3;
  n1.output();  // 4
  n2.output();  // 5
```

- Zugriff aus statischer Methode auf nicht-statische Elemente nur durch erzeugen eines Objekts möglich
  - ```
    public static void output() {
      System.out.println(i + j);  // compile error: i non-static
    }
    ```
  - Umgekehrt immer möglich
    ```

Wichtige vordefinierte Klassen für Mathematik

- Math (alles statisch)
  - Trigonometrie `Math.sin`, `Math.cos`, ...
  - Runden `Math.round`, `Math.ceil`, `Math.floor`
    - `ceil` und `floor` liefern `double` !
  - Expontentation `Math.pow`, `Math.sqrt`, `Math.exp`, `Math.log`
  - Extrema `Math.min`, `Math.max`
  - Absolutbetrag `Math.abs`
  - Vorzeichen `Math.signum`
  - (Pseudo-)Zufallszahlen mit `double Math.random()` $\in [0;1[$

```
int lower = 2, upper = 7;
int r = (int) (lower + Math.floor(Math.random() * (upper - lower + 1)));
```

    - Jede Zahl $r \in \{2,3,4,5,6,7\}$ gleich wahrscheinlich wegen gleich großen Intervallen



  - Konstanten `Math.PI`, `Math.E`

Wichtige vordefinierte Klassen für Zeichenketten

- Strings
  - Speicherung als unveränderliche Instanzen der Klasse `String`
  - `String s = "Hello, World!";` ist abkürzende Schreibweise für `String s = new String("Hello, World!");`
  - Operator == vergleicht Referenzen
  - Aber `String` implementiert die `equals`-Methode
  - Andere Methoden
    - `int length()`
    - `char charAt(int i)`
    - `static String valueOf(int i);`
    - API Dokumentation http://docs.oracle.com/javase/7/docs/api/
  - Binärer Operator + konkateniert zwei `String`-Objekte und liefert (neu erzeugtes) Ergebnisobjekt zurück
  - `static String format(String s, Object... args)` für Formatierung
    - `String s = String.format("H%s, %sld!", "ello", "Wor");`
    - `s = String.format("Today %1$te.%1$tm.%1$tY", java.util.Calendar.getInstance()); // Today 10.01.2012`
    - `s = String.format("e = %+6.2f", Math.E); // e = ␣+2,72`
    - % [arg_index$] [flags] [width] [.precision] conversion

Wichtige vordefinierte Klassen für Zeichenketten...

- Klasse `StringBuilder`
  - Instanzen speichern veränderliche Strings
  - Benötigter Speicherplatz wird automatisch einer neuen Größe angepasst
  - ```
    final StringBuilder b = new StringBuilder("Hallo");
                            System.out.println(b);  // Hallo
    b.setCharAt(1, 'e');    System.out.println(b);  // Hello
    b.insert(5, ',');       System.out.println(b);  // Hello,
    b.append(" World!");    System.out.println(b);  // Hello, World!
    ```
  - Buchstaben von 0 bis `b.length() - 1` indiziert (wie bei Strings)
  - `final` fixiert Objektreferenz/Identität, nicht Zustand
    - `append()` legt im Vergleich zu + bei Strings kein neues Objekt an
  - ```
    String s = new String();
    for (int i = 0; i < 100_000; ++i) {
      s += " ";
    }
    ```
    - Laufzeit 12 Minuten auf Pentium 1,6 GHz unter JAVA 5.0
    - Seit JAVA 6.0 optimiert und etwas entschärft
  - ```
    StringBuilder b = new StringBuilder();
    for (int i = 0; i < 100_000; ++i) {
      b.append(" ");
    }
    ```
    - Laufzeit < 1 Sekunde
    ```

Wichtige vordefinierte Klassen für Zeichenketten...

- StringTokenizer
  - Zerlegung von Strings
  - ```
    import java.util.StringTokenizer;  // explained in next section
      ⋮
    String s = "this text will be dissected into every single word";
    String delim = " ";
    StringTokenizer t = new StringTokenizer(s, delim);
    while (t.hasMoreTokens()) {
      System.out.println(t.nextToken());
    }
    ```
  - Alternative: Zerlegung mittels regulärer Ausdrücke

    ```
    String[] tokens = s.split("\\s+");
    ```

    - Irgend ein Leerzeichen als Trennzeichen (Tab, Whitespace, ...)
    - Bzw. beliebig viele (mindestens eines) von denen hintereinander
    - Backslash als Escape-Zeichen für Backslash, der normalerweise Sonderzeichen einleitet
    ```

Wichtige vordefinierte Klassen für Arrays

- Arrays verhalten sich wie Objekte
  - Erklärung für Syntax
    - Erzeugung durch den `new`-Operator, z. B. `int[] a = new int[7];`
    - Zugriff auf das (finale) Längenattribut `a.length`
    - Übergabe von Arrays durch Call-by-Reference und nicht durch Call-by-Value
  - Schreibweise `a[i]` lässt sich als Abkürzung für `a.get(i)` bzw. `a.set(i, ...)` deuten
- Es gibt auch eine Klasse `java.util.Arrays`
  - Zum komfortablen Arbeiten mit Arrays
  - Nur statische Methoden
  - `static void fill(`Typ`[] a,` Typ `data)` füllt `a` mit `data`
  - `static int binarySearch(`Typ`[] a,` Typ `data)`
    - Liefert den Index des gesuchten Elements
    - `a` muss sortiert sein, sonst undefiniertes Ergebnis
    - Wenn `data` kein Schlüssel ist, d. h. es mehrere Elemente mit diesem Wert gibt, dann wird irgendein passender Index geliefert
  - `static void sort(`Typ`[] a)` sortiert Elemente aufsteigend
  - `static boolean equals(`Typ`[] a,` Typ`[] b)` prüft auf gleichen Inhalt

Wichtige vordefinierte Klassen für Arrays...

- Es gibt auch eine Klasse `java.util.Arrays`...
  - `static String toString(Typ[] a)` liefert den Inhalt als Zeichenkette
    - Stringrepräsentationen der gespeicherten Elemente

      ```
      int[] a = 1, 2, 3, 4, 5;
      String s = Arrays.toString(a);  // [1, 2, 3, 4, 5]
      ```
  - Nicht zu verwechseln mit `java.lang.reflect.Array` zur Repräsentation eines Arraydatentyps
    - Brauchen wir (bis auf Weiteres) nicht!

Pakete

- Gruppierung mehrerer (verwandter) Klassen
- Insbesondere zur Vermeidung von Namenskonflikten
  - Definition eigener *Namensräume* (= *name spaces*)
- Verwendung von Klassen aus Paketen
  - `Paketname`.`Klassenname` bei jeder Verwendung
    - `java.util.Currency c;`
  - `import` `Paketname`.`Klassenname`;
    - `import java.util.Currency; ... Currency c;`
    - Am Quelltextbeginn (vor der Klassendefinition)
    - Erlaubt Klassen direkt mit Namen anzusprechen
    - Im Paketnamen sind Punkte enthalten
  - `import` `Paketname`.* importiert alle Klassen im Paket
    - `import java.util.*; ... Currency c; Date d;`
    - Verboten: `import java.*; ... util.Currency c;`
- `package` `Paketname`; am Anfang des Quelltexts fügt eigene Klasse zu diesem Paket hinzu
  - Per (Soll-)Konvention beginnen Namen mit rückwärtsgelesenem Domainnamen
    - `de.uni_passau.fim.chris.utils`
    - `-` in JAVA-Bezeichnern nicht erlaubt
    - `.` trennt Unterpakete voneinander ab
  - Paketstruktur bildet Dateistruktur eines Projekts ab
    - `.` entspricht dabei / im Dateisystem

Pakete...

- (Soll-)Namenskonventionen werden nicht hart vom Compiler durchgesetzt

```
package a;

public class a {
  a.a x;  // compile error: a.a cannot be resolved to a type
}
```

  - Klassenname beginnt nicht mit Großbuchstaben
  - Klassenname ist mit Paketname identisch
  - Verschattung des Paketnamens
    - Es gibt keinen Mechanismus um auf den Paketnamen (FQN) innerhalb der Klasse zuzugreifen
- Kompilieren
  - `javac de/uni_passau/fim/chris/utils/*.java`
  - Class-Dateien werden per Default neben JAVA-Dateien erstellt
  - Eclipse macht das automatisch
  - Spezielles Build-Tool für die Konsole: ant (`http://ant.apache.org/`)
- Besonderes Paket `java.lang`
  - Enthält elementare Klassen wie `String`, `StringBuilder`, `Math`, `System`
  - Wird automatisch eingebunden

Pakete...

- Verwendung von statischen Methoden aus Klassen
  - `import static` `Paketname` . `Klassenname` . `Methodenname`
  - `import static java.lang.Math.sqrt;`
    erlaubt `sqrt(9.0)` anstatt `Math.sqrt(9.0)` zu schreiben
  - `import static java.lang.Math.*;` erlaubt alle statischen Methoden einfach mit Ihrem Namen zu verwenden

Kapselung in Klassen

- Sichtbarkeit von Attributen und Methoden eines Objekts bzw. einer Klasse
- Durch Modifier
- Schlüsselwort `public`
  - Element von extern und intern uneingeschränkt sichtbar und zugreifbar
  - `Account a = new Account(); a.balance = -200;`
  - Grundregel: für alle Klassen, Konstruktoren und allgemeine Methoden
- Schlüsselwort `private`
  - Nur innerhalb der Klasse zugreifbar
  - Beispiel Bankkonto

```
public class Account {
  private int balance;

  public void deposit(int amount) { balance += amount; }
  public void withdraw(int amount) { balance -= amount; }
  public int getBalance() { return balance; }
}


  ...
  Account account = new Account();
  account.balance = -200; // compile error: field invisible
  account.withdraw(200);  // ok
```

    - Nur Bankangestellter darf in den Tresorraum und Geld zur Auszahlung holen
    - Kunde nicht

Kapselung in Klassen...

- Schlüsselwort `private`
  - Aber Zugriff auch auf Attribute anderer Objekte gleichen Typs/Klasse

```
public class Account {
  private int balance;
  ...
  public boolean equals(Object other) {
    return balance == ((Account) other).balance;
  }
}
```

  - Grundregel: für alle Attribute und Hilfsmethoden
    - Manipulation von Attributen nur über Methoden, z. B. `public void setX(int x)` und `public int getX()`
    - Vorteil: leichte Überprüfung auf Gültigkeit von Werten
  - *Kapselung* trennt Klassen (Code) voneinander
- Sichtbarkeit auch für statische Attribute und Methoden
  - Z. B. für Konstanten oder Hilfsmethoden die nicht auf dynamische Attribute zugreifen

```
public class Math {
  public static final double PI = 3.14159265358979323846;

  private static int calcPrecision(int n, int i) { ... }
  ...
}
```

Kapselung in Klassen...

- Ohne Modifier für Sichtbarkeit
  - Innerhalb aller Klassen des Pakets zugreifbar
- Andere Modifier (hier nicht explizit behandelt)
  - Für Sichtbarkeit höchstens einer aus `public`, `private`, `protected`
  - Für andere Zwecke teils nur für Methoden und/oder Klassen `abstract`, `native`, `strictfp`, `synchronized`, `transient`, `volatile`
  - Mehrere gleichzeitig getrennt durch Leerzeichen

Aufzählungen

- Erzeugen eines Aufzählungstyps mit Schlüsselwort `enum`
  `enum` `Name` `{` `Eintrag1` `, ..., ` `EintragN` `};`
- Nur als Klassenmember oder auf Ebene der Klassen in eigener Datei
  - Nicht lokal in Methoden
  - Wird vom Compiler intern in (*Sub*-)Klasse `Name` umgewandelt mit Einträgen als statische konstante Attribute
- Klasse `Enum` ist der Typ von `Name` . `EintragX`

Aufzählungen...

- enum-Konstanten in switch-Anweisungen

```
package utils;

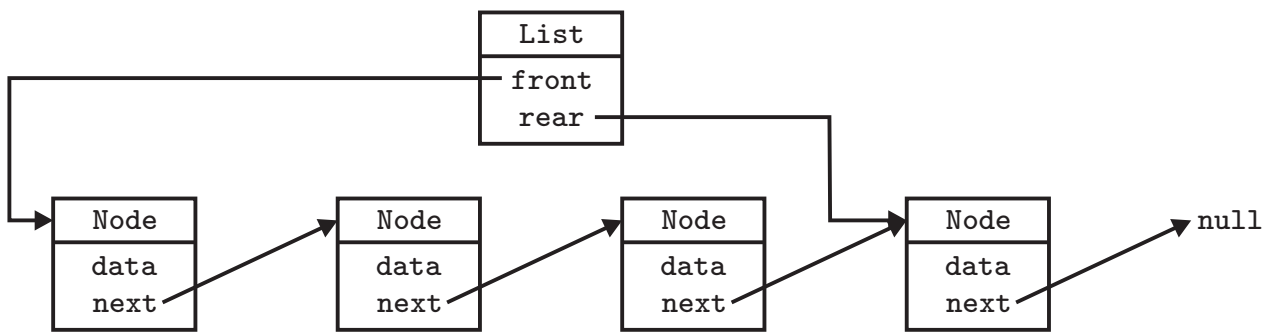
public class Calendar {
  public enum WeekDay { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
                        SATURDAY, SUNDAY }

  public static void main(String[] args) {
    WeekDay w = WeekDay.SATURDAY;
    switch (w) {
    case SATURDAY :
      System.out.println("It's party time");
      break;
    }
  }
}
```

  - case WeekDay.SATURDAY nicht notwendig
  - Sogar verboten
  - Wird automatisch aufgelöst, da Typ von *w* bekannt ist
  - Das ist nur bei switch so, sonst nicht

Aufzählungen...

- `import static` `Paketname` . `Klassenname` . `Name` `.*` erlaubt direkte Benutzung von Eintägen in anderen Klassen
  - ```
    import utils.Calendar.Weekday;
    import static utils.Calendar.Weekday.*;
      ⋮
      Weekday w = SATURDAY;
    ```
  - `import static Calendar.Weekday.*` im gleichen Package (ohne vorangestelltem `utils.`) verboten
    ```

6. Dynamische Datenstrukturen

Einfach verkettete Listen

- Nachteile von Arrays
  - Einfügen und Löschen erzwingt Verschieben aller Elemente mit höherem Index
  - Umkopieren notwendig falls zu klein
- Vorteil von Arrays
  - Einfacher Zugriff auf Elemente mit Klammeroperator (*random access*)
- Listen: Zugriff sequentiell
  - Ggf. Iteration durch alle Listenelemente
- Aber dynamische Größenveränderungen ohne Umkopieren mgl.
- Verkettung der Elemente (*Knoten*) zum Nachfolger
  - Zeiger/Referenzen



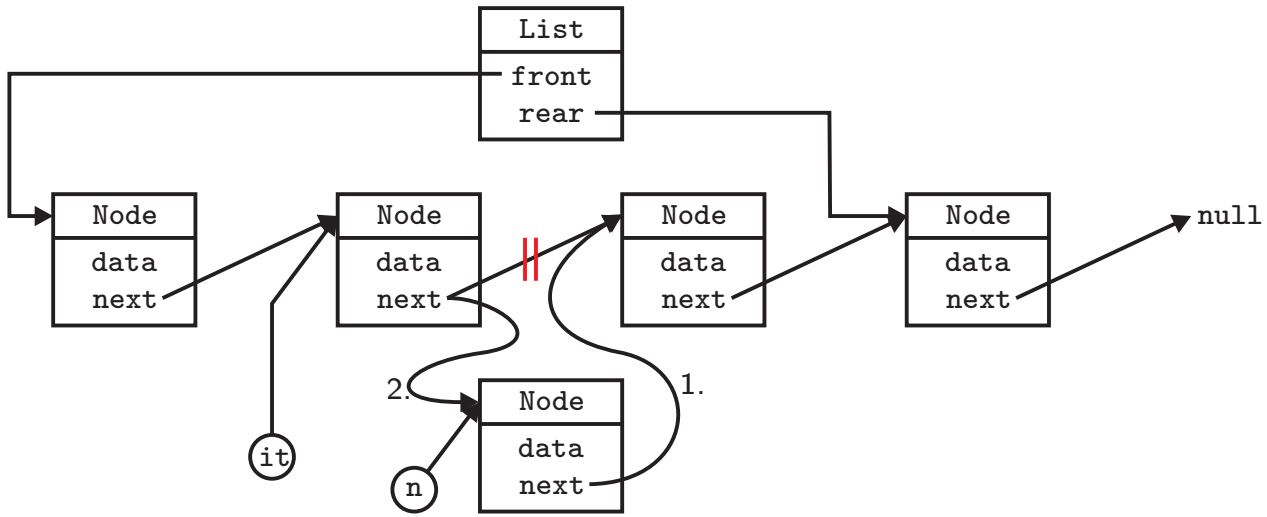
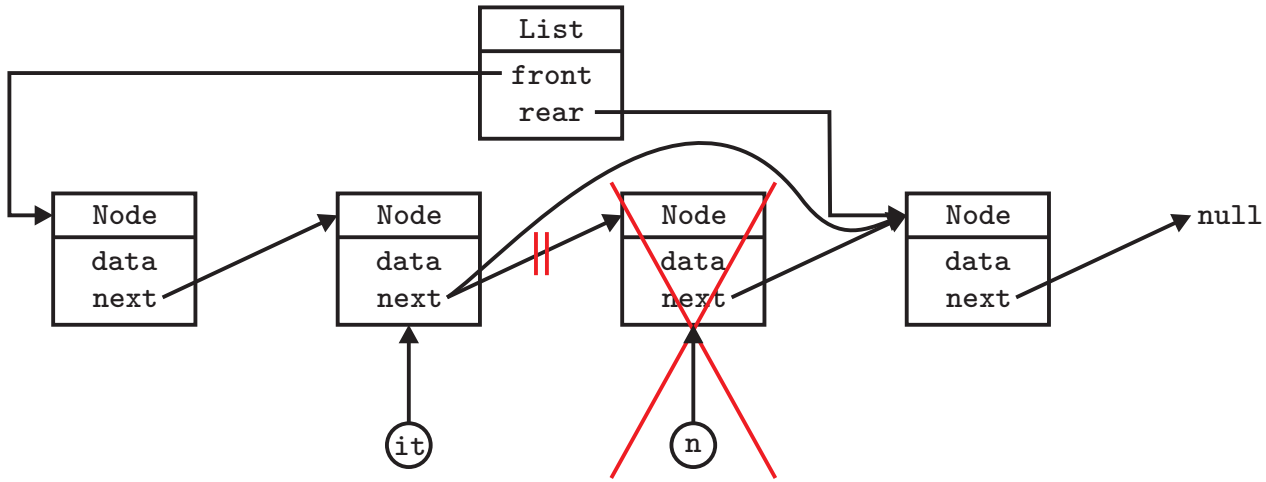
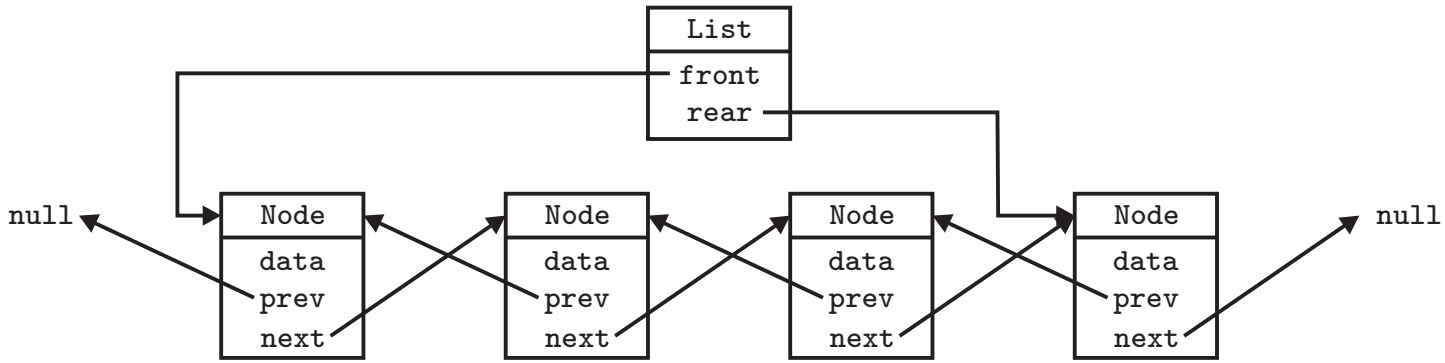
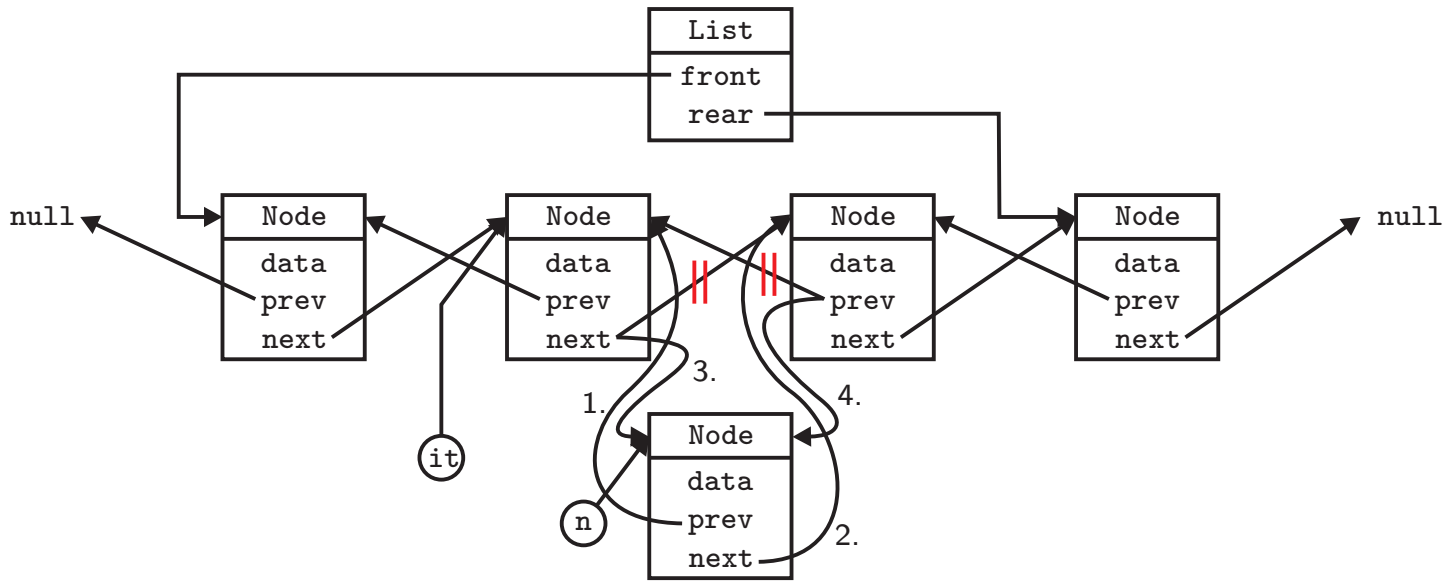
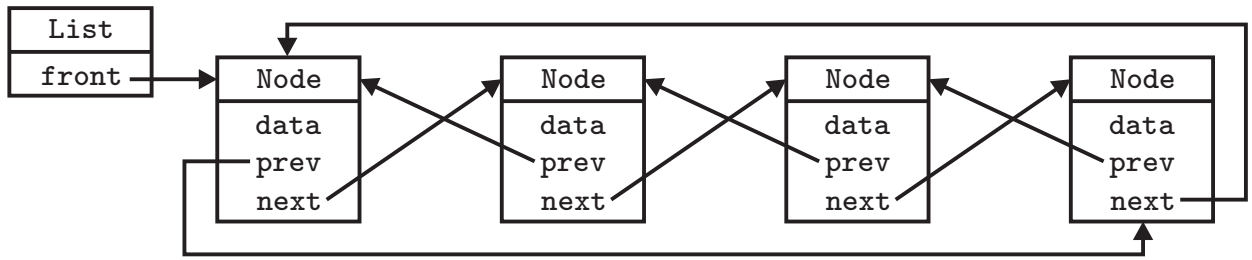
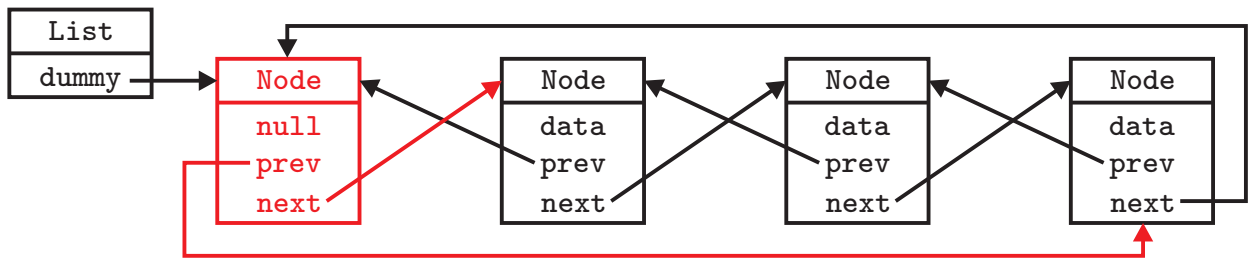
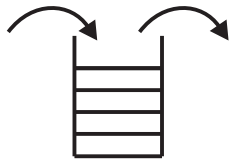
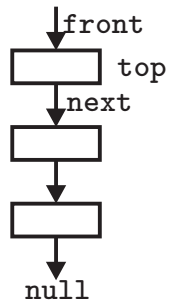
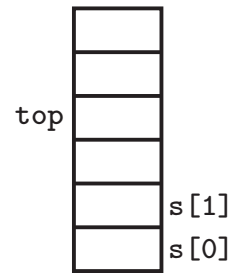
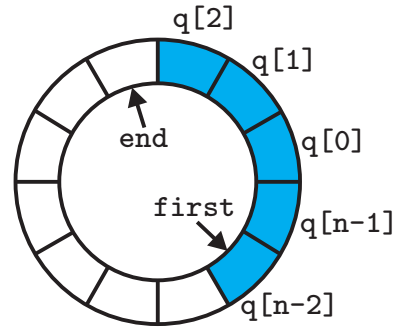
Knoten

- ```
  package de.uni_passau.fim.chris.list;

  public class Node {
    public int data;
    public Node next;

    public Node(int dataValue) {
      data = dataValue;
      next = null;  // initially null reference
    }

    public String toString() {
      return "{" + data + "} ";
    }
  }
  ```
- `toString()` ist Spezialmethode

  ```
  Node node = new Node(10);
  System.out.println(node);  // {10}
  ```
  ```

Liste

```
package de.uni_passau.fim.chris.list;

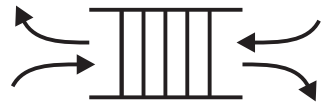
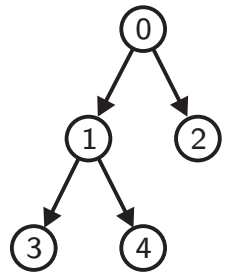
public class List {
  private Node front;  // reference to first node of list
  private Node rear;   // reference to last node of list

  public List() {
    front = null;      // initially the list is emtpy
    rear = null;
  }

  public boolean isEmpty() {
    return front == null;
  }
  ⋮
  public void display() {
    for (Node it = front; it != null; it = it.next) {
      System.out.print(it);
    }
    System.out.println();
  }
}
```

Einfügen von Knoten

- Einfügen nach einem bestimmten Knoten




```
public void add(Node it, int dataValue) {
  assert it != null : "cannot insert after null";
  Node n = new Node(dataValue);
  n.next = it.next;
  if (n.next == null) { rear = n; }  // new end
  it.next = n;
}
```

  - Reihenfolge der Befehle relevant!
  - Schlüsselwort `assert` beim Aufruf aktivieren
    - `java -ea ListDemo`
    - Wird ansonsten aus Performancegründen einfach ignoriert
    - Details später

Einfügen von Knoten am Anfang

- Sonderbehandlung

```
public void addFront(int dataValue) {
  Node n = new Node(dataValue);
  n.next = front;
  if (n.next == null) { rear = n; }  // new end
  front = n;
}
```

- Beim Einfügen vor einem bestimmten Knoten nicht notwendig
  - Nachteil: Vorgänger muss dann gesucht werden
  - Dessen `next`-Zeiger muss auf das neue Element gesetzt werden
  - Kostet im schlimmsten Fall fast einen ganzen Listendurchlauf

Löschen von Knoten

- Löschen über Datenelement (*Schlüssel*)



- „Vorauslesen“ notwendig
  - *Look-Ahead* 1 Knoten
  - Ansonsten Vorgänger und dessen zu aktualisierender `next`-Zeiger unbekannt
  - Alternative: Vorgänger suchen
- Auf Sonderfälle am Anfang und am Ende aufpassen!

Löschen von Knoten...

```
public void remove(int dataValue) {
  if ((front != null) && (front.data == dataValue)) {  // && is lazy
    front = front.next;
    if (front == null) { rear = null; }
  } else if (front != null) {
    // list not empty
    Node it = front;  // it.data != dataValue
    while (it.next != null) {
      Node n = it.next;
      if (n.data == dataValue) {
        if (n.next == null) { rear = it; }
        it.next = n.next;
        break;
      } else {
        it = it.next;
      }
    }
  }
}
```

- Hier wird bei mehreren passenden Knoten nur der erste gelöscht
- Alle: ohne `break`, statt erstem `if` ein `while` und erstes `else` weg

Weitere typische Operationen auf Listen

- Löschen am Listenanfang und -Ende bzw. eines bestimmten Elements
- Iteration (siehe `List.display()`)
- `clear`, `getFirst`, `getLast`, `add` (hinten/an vorg. Position einfügen), `concat`, `isEmpty`, `subList` (Teilliste zwischen zwei bestimmtem Elementen), `contains`, `find`, `size`, `get`
- Für hinten Anfügen und Konkatenation ist `rear`-Zeiger sehr nützlich
  - Erübrigt einen kompletten Durchlauf der Liste
  - Aber für Funktionalität nicht notwendig

Doppelt/zweifach verkettete Listen

- Zusätzlich `prev`-Zeiger im Listenelement



- Vorteil: Iteration/Navigation in beide Richtungen möglich
- Einfügen (danach)



  - Sonderbehandlung `addFront`

Zyklische Listen

- Zyklische Verkettung



- Mit Dummy-Element



  - Trick, der Programmierung erleichtert und vereinheitlicht
  - Meist als künstliches Ende-Element vom Typ `Node` das keine Daten enthält
  - Keine Sonderfälle beim Einfügen/Löschen
  - JAVA/C++ API benutzt dieses Konzept auch für lineare Listen

Stack (Keller, Stapel, Last-In-First-Out, LIFO)

- Kopfbahnhof







- Realisierung mittels Liste
  - Nur vorne/oben neues Element einfügen `push(...)`
  - Zugriff nur auf erstes/oberstes Element `int peek()`
  - Löschen jeweils nur erstes/oberstes Element `int pop()`
  - Sonstige Operationen `clear`, `size`, `isEmpty`
- Realisierung mittels Array
  - Kellerpegel entweder auf oberstes Element oder auf erstes freies Element
  - Vorsicht bei Über- oder Unterlauf
  - Bei Speicherung von Objekten in `pop()`: `s[top] = null` setzen
    - Sonst kann Garbage Collection nicht aufräumen (*Memory Leak*)

Stack (Keller, Stapel, Last-In-First-Out, LIFO)...

- Anwendungen
  - (Rekursive) Methodenaufrufe (= *Aufrufstack*)
    - Nächster auszuführender Befehl nach dem Methodenaufruf (= *Rücksprungadresse*)
    - Aktuelle (lokale) Methoden-Parameter
  - Abprüfen korrekter Klammerung ((([[()]([])]))())
  - Auswerten von Ausdrücken in Postfixnotation
  - Traversieren von Bäumen/Graphen

Queue ((Warte-)Schlange, First-In-First-Out, FIFO)

- Durchgangsbahnhof



- Realisierung mittels Liste
  - Nur hinten neues Element einfügen `offer(...)`
  - Zugriff nur auf erstes Element `int peek()`
  - Löschen jeweils nur erstes Element `int poll()`
  - Sonstige Operationen `clear`, `size`, `isEmpty`
- Realisierung mittels Array
  - Ringpuffer (der Länge `n`)
  - Auf `q[n - 1]` folgt `q[0]` (Indexrechnung mod `n`)
  - Zwei Zeiger (Cursor) `int first` und `int end`
    - `end` zeigt auf nächste Einfügeposition und nicht auf letztes Element
    - Macht Implementierung einfacher
  - `size` ist Anzahl echter Elemente zwischen den beiden Cursors
  - Vorsicht bei Über- oder Unterlauf

Queue mittels Ringpuffer

```
public class ArrayQueue {
  private int[] q;
  private int first = 0;
  private int end = 0;  // position for next insertion
  private boolean empty = true;

  public ArrayQueue(int bufferSize) {
    q = new int[bufferSize];
  }

  public void offer(int newData) {
    if (((end == first) && (!empty)) || (q.length == 0)) {
      System.out.println("Queue already full: " + newData + " not inserted.");
      return;
    }
    q[end] = newData;
    end = (end + 1) % q.length;
    empty = false;
  }

  public boolean isEmpty() {
    return (first == end) && empty;
  }
}
```

Queue...

- `poll`, `peak` und `size` als Übungsaufgabe
- Anwendungen
  - Warteschlange vor der Mensa
  - Stau auf der Autobahn
  - Puffern von Jobs (Scheduling)
  - Traversieren von Graphen

Deque (double ended queue)

- Mischung aus Stack und Queue



- Realisierung als Liste oder Ringarray
- Operationen
  `offerFront`, `offerBack`, `pollFront`, `pollBack`, `peekFirst`, `peekLast`, `size`, `isEmpty`
- Anwendungen
  - In der Praxis keine
  - Außer bei Klausuren ☺

Priority Queue (Vorrangschlange, Prioritätswarteschlange)

- Unterstützt Operation „gib mir extremales Element einer Menge bzgl. einer Ordnung“
  - `peek`, `poll`
- Array oder Liste, und extremales Element bei Zugriff suchen
  - Im schlimmsten Fall ganze Liste durchlaufen
- Effizienter
  - Spezielle Baumdatenstruktur: *Heap*
  - Hier nicht behandelt (→ VL Algorithmen & Datenstrukturen)
- Für manche Algorithmen notwendig `changeKey(Node node, int newKey)`
- Weitere Operationen wie normale Queue
  `offer, clear, size, isEmpty`
- Anwendungen
  - Sortieren (Heapsort)
  - Kürzeste Wegesuche (Navigationssystem)

Einschub: $\mathcal{O}$-Notation

- Beschreibungsform für Aufwand
  - Z. B. Laufzeit (= Anzahl ausgeführter Anweisungen), Speicherverbrauch
  - Nach Paul Bachmann 1894, Edmund Landau 1909
- Zeigt „nur“ asymptotische Größe/Komplexität an
- 1000-fach mal so große Eingabe bedeutet

Name	**Notation**	**Intuition**	**Beispiel**
konstant	$\mathcal{O}(1)$	gleiche Arbeit	Zugriff auf i-tes Element im Array
logarithmisch	$\mathcal{O}(\log n)$	nur $\approx$ zehnfache Arbeit	binäre Suche
linear	$\mathcal{O}(n)$	tausendfache Arbeit	Durchlauf einer Liste
„$n \log n$“	$\mathcal{O}(n \log n)$	$\approx$ zehntausendfache Arbeit	Sortieren
quadratisch	$\mathcal{O}(n^2)$	millionenfache Arbeit	2 geschachtelte `for`-Schleifen
kubisch	$\mathcal{O}(n^3)$	milliardenfache Arbeit	3 geschachtelte `for`-Schleifen
polynomial	$\mathcal{O}(n^c)$	gigantisch viel Arbeit (für großes c)	Strassen-Matrixmultiplikation $\mathcal{O}(n^{\log_2 7})$
exponentiell	$\mathcal{O}(2^n)$	hoffnungslos viel Arbeit	Erfüllbarkeit von boolschen Ausdrücken (SAT)

- Hier keine formale Definition
  - $\mathcal{O}(f) = \{ g \mid \exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0 : g(n) \leq c \cdot f(n) \}$

Binärbäume

- Verallgemeinerung einer Liste
- Jeder Knoten kann zwei `next`-Pointer haben
- Man spricht hier von *Kindern* (= *Söhnen*)
- Doppelte Verkettung manchmal von Vorteil
  - Kinder kennen *Elter* (= *Vater*)
  - Analog zu `prev` bei Listen
  - Ermöglicht Baum bottom-up zu durchlaufen
- (Einziger) elterloser Knoten ist die *Wurzel*
- Kinderlose Knoten sind *Blätter*
  - Alle anderen sind *innere Knoten*
- Geordneter Binärbaum
  - Reihenfolge der Kinder spielt eine (semantische) Rolle
  - `leftChild`, `rightChild`

Binärbäume...

- Typische Operationen auf Bäumen
  - `clear`, `root`, `leaves`, `add`, `remove`, `attachSubtree`, `detachSubtree`, `contains`, `find`, `getMax`, `getMin`
- Implementierung mittels Verzeigerung
  - Ähnlich wie Liste
  - Jeder Knoten hat Referenzen `leftChild`, `rightChild` bzw. (geordnete) Liste der Kinder
  - Jeder Knoten hat Referenz auf seinen Elter (Wurzel `null`)
- Implementierung mittels Array
  - Knoten an Position i
  - Linkes Kind an Position $2i + 1$
  - Rechtes Kind an Position $2i + 2$
  - Elter an Position $\lfloor \frac{i-1}{2} \rfloor$
  - Falls der Baum unvollständig ist existieren „Löcher“ die gekennzeichnet werden müssen
    - Durch `null`-Eintrag
  - Speicherplatzverbrauch wie entsprechender vollständiger Baum

Traversieren (Iteration) von Binärbaumen

- Ausgehend von der Wurzel



- *Inorder*: rekursiv linker Unterbaum, aktueller Knoten, rekursiv rechter Unterbaum
  - 3, 1, 4, 0, 2
  - ```
    public static void traverse(Node node) {
      if (node == null) { return; }
      traverse(node.leftChild);
      System.out.print(node + " ");  // toString()
      traverse(node.rightChild);
    }
    ```
- *Preorder*: aktueller Knoten, rekursiv linker Unterbaum, rekursiv rechter Unterbaum
  - 0, 1, 3, 4, 2
- *Postorder*: rekursiv linker Unterbaum, rekursiv rechter Unterbaum, aktueller Knoten
  - 3, 4, 1, 2, 0
    ```

Binäre Suchbäume

- Wichtiger Sinn von Bäumen als Datenstruktur ist schnelle Suche
- Paarweise Vergleichbarkeit (Ordnung) auf `data`-Elementen notwendig
- Invariante beim Suchbaum
  - `data`-Element aller Knoten des linken Unterbaums ist kleiner(-gleich) als das im aktuellen Knoten
  - `data`-Element aller Knoten des rechten Unterbaums ist größer als das im aktuellen Knoten
- *Höhe* sollte $\mathcal{O}(\log n)$ und nicht $\mathcal{O}(n)$ sein
  - Längster Weg von der Wurzel zu einem Blatt
- Maßnahmen gegen *Degeneration* zu einer Liste
  - AVL-Baum mit Rotationen
  - B-Baum mit Knotensplits
  - Rot-Schwarz-Baum
  - Hoffen auf Gleichverteilung der einzufügenden Elemente
    - Hier Strategie der Wahl

Binäre Suchbäume...

- ```
  package de.uni_passau.fim.chris.tree;

  public class Node {
    public int data;
    public Node leftChild;
    public Node rightChild;

    public Node(int value) {
      leftChild = null;
      rightChild = null;
      this.data = value;
    }
  }
  ```
- ```
  package de.uni_passau.fim.chris.tree;

  public class BinTree {
    private Node root;

    public BinTree() { root = null; }

    public boolean isEmpty() {
      return root == null;
    }
  ```

Binäre Suchbäume...

```
private Node add(Node node, int value) {
  if (node == null) {
    node = new Node(value);
  } else if (value < node.data) {
    node.leftChild = add(node.leftChild, value);
  } else if (value > node.data) {
    node.rightChild = add(node.rightChild, value);
  } else {
    // ignore value and return node as is (here data values unique)
  }
  return node;
}

public void add(int value) { root = add(root, value); }

private int getMax(Node node) {
  if (node.rightChild == null) {
    return node.data;
  } else {
    return getMax(node.rightChild);
  }
}

public int getMax() { assert !isEmpty(); return getMax(root); }
```

Binäre Suchbäume...

```
  private Node remove(Node node, int value) {
    if (node == null) {
      // not found -> nothing to do
    } else if (value < node.data) {
      node.leftChild = remove(node.leftChild, value);
    } else if (value > node.data) {
      node.rightChild = remove(node.rightChild, value);
    } else {
      if (node.leftChild == null) {
        node = node.rightChild;
      } else if (node.rightChild == null) {
        node = node.leftChild;
      } else {
        int max = getMax(node.leftChild);
        node.data = max;
        node.leftChild = remove(node.leftChild, max);
      }
    }
    return node;
  }

  public void remove(int value) { root = remove(root, value); }
}
```

7. Benutzung von Datenstrukturen aus der Funktionsbibliothek (API)

Vorgefertigte Listen im API

- Doppelt verkettete Liste

```
import java.util.Iterator;
import java.util.LinkedList;
  ⋮
  LinkedList<String> liLst = new LinkedList<String>();
  liLst.add("Hello,"); liLst.add("World!");

  for (Iterator<String> it = liLst.iterator(); it.hasNext(); ) {
    String s = it.next();
    System.out.println(s);
  }
```

- Zugriffsmethodennamen wie bei unseren selbstdefinierten Listen
- Kurzform für Iteration

```
for (String s : liLst) {
  System.out.println(s);
}
```

  - Foreach wie bei Arrays
  - Vorteile
    - Einfacher und kürzer
    - Iterationsvariable s wird nur im Schleifenkopf gesetzt

Generische Typen

- Bei Definition von `LinkedList` bzw. von `Node` ist nicht klar, welchen Typ `data` haben soll
  - Wir haben bei unseren selbstgeschriebenen Listen meist `int` verwendet
- Java API überlässt dies dem Nutzer
  - Angabe des Typs in spitzen Klammern
  - `LinkedList<String> l`
  - `l` ist vom Typ `LinkedList<String>` und nicht vom Typ `LinkedList`
  - Bei der Implementierung der Klasse nur Platzhalter (*Generic*) für den Typ von `data`
    - Generics (= *Typ-Parameter*)
- Java lässt hier keine primitive Datentypen wie `int`, `boolean`, `double`, usw. zu
  - Lösung: *Wrapper*-Klassen

Generische Typen...

- Implementierung

```
public class Node<T> {
  public T data;
  public Node<T> next = null;

  public Node(T dataValue) {
    data = dataValue;
  }
}

public class List<T> {
  private Node<T> front;
  ⋮
  public void add(Node<T> it, T dataValue) {
    // insert after it
    Node<T> n = new Node<T>(dataValue);
    n.next = it.next; it.next = n;
  }
}
```

- Konvention
  - Generics werden wie Klassen groß geschrieben
  - Oft einzelne Buchstaben (`T` wie Template)

Wrapper

- Vordefinierte Hüllklassen aus dem Paket `java.lang`
- `Byte`, `Short`, `Integer`, `Long`, `Boolean`, `Float`, `Double`, `Character`
- Automatische/implizite Umwandlung (*Auto(un)boxing*)

```
Integer i = new Integer(10); i = 11;
int j = i + 1;
System.out.println(i + " " + j);  // 11 12
```

- Explizit
  - `i.intValue()`
- Aber: keine implizite Umwandlung für Methodenaufrufe
  - `j.toString()` statt `(new Integer(j)).toString()` liefert Compilefehler
- „Rechnen“ mit `null` liefert Laufzeitfehler

```
Integer i = null;
int j = i + 1;
```

  - Bei offensichtlichen Fällen liefert der Compiler Warnung Null Pointer Access
- Eigene Wrapper gut für Parameterüberabe mittels Call-by-Reference
  - Vordefinierte konstant/*immutable* da keine Methoden zur Werteänderung vorhanden
    - Analog zur Klasse `String`

Wrapper...

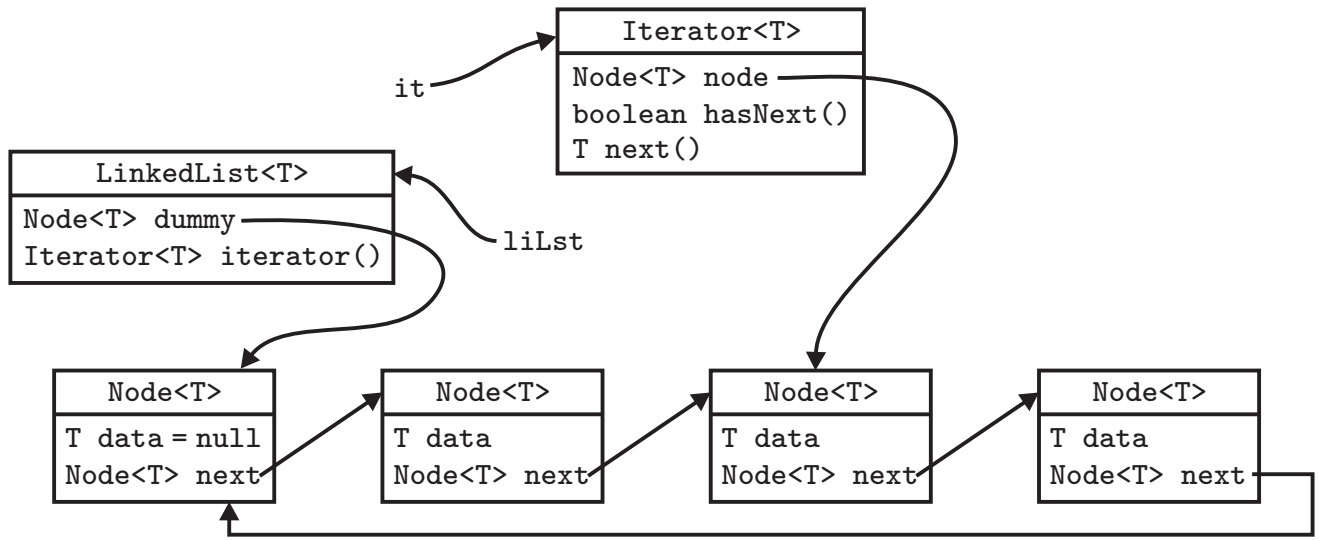
- Jede nummerische Wrapperklasse stellt zwei Konstanten zur Verfügung
  - `static final` `Typ` `MIN_VALUE`
  - `static final` `Typ` `MAX_VALUE`
- `Float` und `Double` haben zusätzliche Konstanten (= *Fluchtwerte*)
  - `NaN` (0.0 / 0.0), `NEGATIVE_INFINITY` (neg. Zahl / 0.0), `POSITIVE_INFINITY` (pos. Zahl / 0.0)
- Falle

```
Integer i = new Integer(32);
Integer j = new Integer(42);
Integer k = new Integer(32);

if (i < j) {    // autounboxing
  System.out.println("i < j");
}
if (i == k) {  // comparing references
  System.out.println("i == k");
}
```

  - Anwenden des ==-Operators auf boxed Primitiven ist fast immer falsch!
  - Zur Erinnerung: jedes Objekt hat eindeutige Identität (= Referenz darauf)
  - Besser

```
if (i.equals(k)) { ... }
```

ArrayList

- Array-Implementation des List Interfaces mit dynamischer Größenänderung
  - Hülle um ein Array, die Zugriffsmethoden einer Liste bietet
- ```
  import java.util.ArrayList;
    ⋮
  ArrayList<Integer> arLst = new ArrayList<Integer>(5);
  arLst.add(new Integer(17));
  arLst.add(10);  // autoboxing
  ```
- Vorteil
  - $\mathcal{O}(1)$ Indexzugriff mit `get(int i)` bzw. `set(int i, Typ elem)`
  - Random Access wie auf einem Array
  - Auf verketteten Listen gibt es diese Zugriffsmethoden auch
    - Ineffizient
    - Element bzw. Index muss immer erst gesucht werden $\mathcal{O}(n)$
- Nachteil
  - `add(Typ elem)` nur amortisiert (= ähnlich durchschnittlich) in $\mathcal{O}(1)$
  - `add(int i, Typ elem)` meistens $\mathcal{O}(n)$
    - Alle Elemente dahinter müssen um eins nach hinten geschoben werden
    - Bei verketteter Liste $\mathcal{O}(1)$ wenn man die Position kennt
- Nur add verändert tatsächliche Größe `arLst.size()`
  - `arLst.get(3)` bzw. `arLst.set(3, 25)` liefert Laufzeitfehler
  - Dagegen hilft auch Setzen der Initialgröße 5 nicht
  ```

Iteratoren

- ```
  for (Iterator<String> it = liLst.iterator(); it.hasNext(); ) {
    String s = it.next();
    System.out.println(s);
  }
  ```
- `Iterator<T>` ist ein „komfortabler Durchlaufcursor“-Typ



  - Definiert im Paket `java.util`
  - *Kollektionen* stellen Instanz davon mit `iterator()` bereit
  - Interface `Iterable<T>` stellt das sicher
    - Erzwingt Vorhandensein der Methode `Iterator<T> iterator()` z. B. in `LinkedList<T>`
  ```

Iteratoren...

- ```
  for (Iterator<String> it = liLst.iterator(); it.hasNext(); ) {
    String s = it.next();
    System.out.println(s);
  }
  ```
- Ein Iterator speichert eine Referenz auf das aktuelle Element
  - `next()` liefert den Datenwert des nächsten Elements/Listenknotens
    - Listenknoten `node` selber ist nicht zugreifbar
  - Schaltet Iterator gleichzeitig/zuerst ein Element weiter

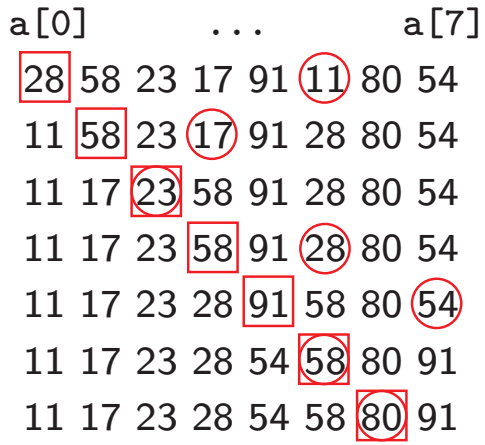
    ```
    public T next() {
      assert hasNext() : "No next element.";
      node = node.next;
      return node.data;
    }
    ```
  - Um Rückgabetyp von `next()` zu definieren muss Typ des Listenelements bekannt sein (`Iterator<String>`)
- Der Iterator weiß bzw. findet heraus, ob es noch ein weiteres Element gibt (`hasNext()`)
  - Intern: wenn `it.hasNext() == false` zeigt `it.node.next` auf das Dummy-Endeelement
- Iteration vorwärts und rückwärts mittels `java.util.ListIterator`
  - Alle Listen haben auch eine Methode `getListIterator()`
    ```

Vorgefertigte Stacks und Queues im API

- `Stack` intern realisiert als `Vector` (≈ `ArrayList`)

```
import java.util.Stack;
  ⋮
  Stack<String> stack = new Stack<String>();
  stack.push("Hello"); stack.push("World");
  String str = stack.pop();     // "World"
  String top = stack.peek();    // "Hello"
  boolean e = stack.isEmpty();  // false
```

  - In neueren JAVA-Versionen wird Verwendung von `Deque` empfohlen

```
Deque<Integer> stack = new ArrayDeque<Integer>();
```

  - Zugriffsmethoden bleiben gleich

- `Queue` intern realisiert als `LinkedList`

```
import java.util.LinkedList;
import java.util.Queue;
  ⋮
  Queue<String> queue = new LinkedList<String>();
  queue.offer("Hello"); queue.offer("World");
  String str = queue.poll();   // "Hello"
  String head = queue.peek();  // "World"
  int s = queue.size();        // 1
  boolean e = queue.isEmpty(); // false
```

Vorgefertigte Priority Queue im API

- ```
  import java.util.PriorityQueue;
    ⋮
  PriorityQueue<Integer> pq = new PriorityQueue<Integer>();
  pq.offer(3); pq.offer(5); pq.offer(1); pq.offer(6);  // autoboxing
  Integer top =  pq.peek();                            // 1
  Integer i =  pq.poll();                              // 1
  Integer newTop =  pq.peek();                         // 3
  ```
- Auf Elementen in der Priority Queue muss Ordnung definiert sein
- `Integer` untereinander vergleichbar
  - Normale $\leq$ Ordnung
  - Was aber mit selbstdefinierten Typen/Objekten?
  - Lösung `Comparable` und `Comparator`
  ```

Comparable

- Objekte müssen `compareTo(Typ o)` Methode implementieren und anzeigen, dass sie (paarweise) vergleichbar sind
- ```
  public class Person implements Comparable<Person> {
    public String name;
    public int persNo;
    public int age;

    public int compareTo(Person other) {
      return age - other.age;
    }
  }
  ```
- Typ des Objekts, mit dem aktuelles Objekt verglichen wird, ist der gleiche
- Rückgabewert von `compareTo` ist
  - < 0 wenn aktuelles Objekt kleiner ist als übergebenes
  - $= 0$ wenn beide Objekte gleich groß sind
  - > 0 wenn aktuelles Objekt größer ist als übergebenes
- Ordnung auf Elementen auch für Sortieren wichtig
  - Statische Methode `Collections.sort(...)`
  - ```
    LinkedList<Person> list = new LinkedList<Person>(); ...
    Collections.sort(list);  // sorted by ascending age
    ```

Comparator

- Weitere Möglichkeit eine Ordnung zu definieren
  - Z. B. wenn kein Zugriff auf den Quellcode der Elemente besteht
  - Oder wenn mehrere Ordnungen auf den Elementen existieren sollen
- ```
  public class PersonComparator implements Comparator<Person> {
    public int compare(Person p1, Person p2) {
      return p1.persNo - p2.PersNo;
    }
  }
  ```
- Methode `compare` für paarweisen Vergleich
  - Rückgabewerte analog zu `compareTo` bei `Comparable`
- Verwendung

  ```
  PriorityQueue<Person> pq
    = new PriorityQueue<Person>(10, new PersonComparator());

  LinkedList<Person> list = new LinkedList<Person>(); ...
  PersonComparator persComp = new PersonComparator();
  Person senior = Collections.min(list, persComp);
  Collections.sort(list, persComp);
  ```
- Falls Objekte nicht `Comparable` sind und kein `Comparator` angegeben wurde, wird anhand Objekt-ID sortiert
  - Das ist in den seltesten Fällen das, was man will
  ```

8. Einfache Algorithmen

Selectionsort

- Suche kleinstes Element im verbleibenden Array und tausche es an die aktuelle Position




- ```
  public static void selectionSort(int[] a) {
    for (int i = 0; i < a.length - 1; ++i) {
      int minIndex = i;
      // search position of smallest element in a[i + 1] to a[a.length - 1]
      for (int j = i + 1; j < a.length; ++j) {
        if (a[j] < a[minIndex]) { minIndex = j; }
      }
      swap(a, i, minIndex);
    }
  }
  ```
  ```

Selectionsort...

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

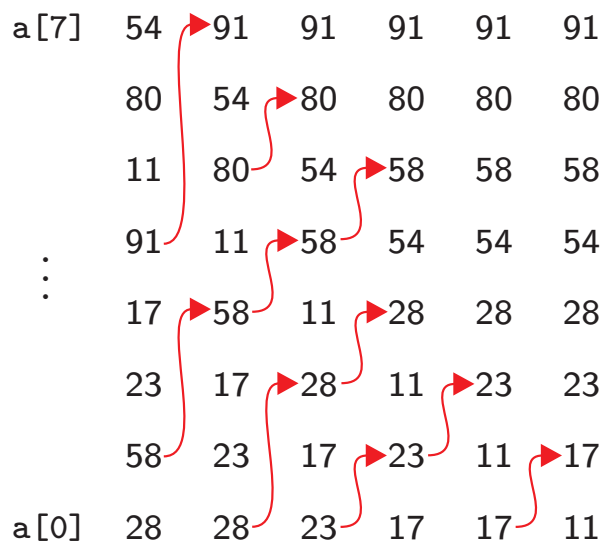
- Eigenschaften: in-situ, nicht stabil

  in-situ Verfahren braucht keinen (bzw. nur $\mathcal{O}(1)$) Hilfsspeicher

  stabil Verfahren ändert Reihenfolge von gleichen Elementen nicht

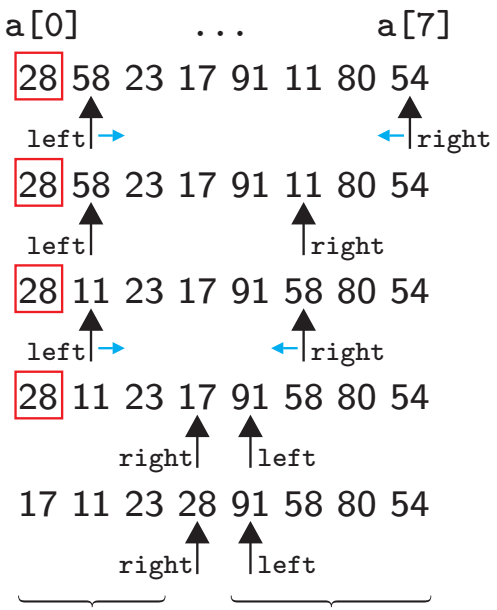
- Alternative
  - Zwei Arrays mit Löschen und Umkopieren
  - Oder ohne Löschen mit drittem `boolean`-Array zum markieren
- Quadratische Laufzeit $\mathcal{O}(n^2)$
  - Für jedes Element muss restliches Array durchgelaufen werden um Minimum zu finden

Insertionsort

- Nimm der Reihe nach das 2., 3., 4., ... Element
  - Wenn es für seine Position zu klein ist
  - Füge es links davon an den richtigen Platz in die schon sortierte Teilfolge ein
  - Schiebe dazu alle Elemente zwischen neuer und alter Position eins nach rechts
    - Bereits kombiniert mit Platzsuche
- ```
  public static void insertionSort(int[] a) {
    for (int i = 1; i < a.length; ++i) {
      int key = a[i];
      int j = i;
      // search new position for key in a[0] to a[i]
      while (j > 0 && key < a[j - 1]) {
        a[j] = a[j - 1];  // shift right
        --j;
      }
      a[j] = key;
    }
  }
  ```
- Eigenschaften: in-situ, stabil
- Quadratische Laufzeit $\mathcal{O}(n^2)$
  - Vergleiche pro Element $1 + 2 + \ldots (n-2) + (n-1)$ (worst case)
- Alternative
  - Zwei Arrays mit Löschen und Umkopieren
  ```

Insertionsort...

- Erweiterung Shellsort
  - Versucht Nachteil dass Elemente oft über weite Teile bewegt werden auszugleichen
  - Praktische Laufzeit besser aber asymptotisch nicht
  - Details http://de.wikipedia.org/wiki/Shellsort

Bubblesort

- Zwei nebeneinander stehende Elemente werden vertauscht, wenn das linke größer ist
- Danach werden die beiden um eins weiter rechts stehenden Elemente verglichen
- Invariante danach
  - Größtes Element steht hinten
- Wiederhole das Ganze für vorderen unsortierten Teil des Arrays
- Größte Zahl steigt auf wie Blasen in einem Wasserglas bis größere Blase (Zahl) sie stoppt



Bubblesort...

```
public static void bubbleSort(int[] a) {
  for (int i = a.length - 1; i > 0; --i) {
    for (int j = 0; j < i; ++j) {
      if (a[j] > a[j + 1]) {
        swap(a, j, j + 1);
      }
    }
  }
}
```

- Eigenschaften: in-situ, stabil
- Quadratische Laufzeit $\mathcal{O}(n^2)$
  - Vergleiche und Vertauschungen pro Element $(n-1)+(n-2)+\cdots+2+1$ (worst case)
  - $\mathcal{O}(n)$ falls schon in richtiger Reihenfolge sortiert (best case)

Bubblesort...

- BubbleSort mit vorzeitigem Abbruch
  - Fertig, falls keine Vertauschung in einem kompletten Durchlauf der äußeren Schleife
  - ```
    public static void bubbleSortWithEarlyTerm(int[] a) {
      boolean change = true;
      for (int i = a.length - 1; i > 0 && change; --i) {
        change = false;
        for (int j = 0; j < i; ++j) {
          if (a[j] > a[j + 1]) {
            swap(a, j, j + 1);
            change = true;
          }
        }
      }
    }
    ```
  - Worst-case Laufzeit wird dadurch nicht besser
  - In der Praxis aber normalerweise schneller
    ```

Quicksort

- Wähle Pivot-Element
  - Z. B. erstes Element
  - Oder Median of Three
- Zerlege Array in zwei Teile (*Divide*)
  - Vorderer Teil nur Elemente $\leq$ Pivot
  - Hinterer Teil nur Elemente $>$ Pivot
  - Linear $\mathcal{O}(n)$ mittels zweier zur Mitte aufeinanderzulaufender Zeiger
- Sortiere rekursiv die beiden Teile mit Quicksort (*Conquer*)
  - Ende der Rekursion bei 1-elementigen Mengen
  - Die sind trivialerweise schon sortiert
- Beispiel



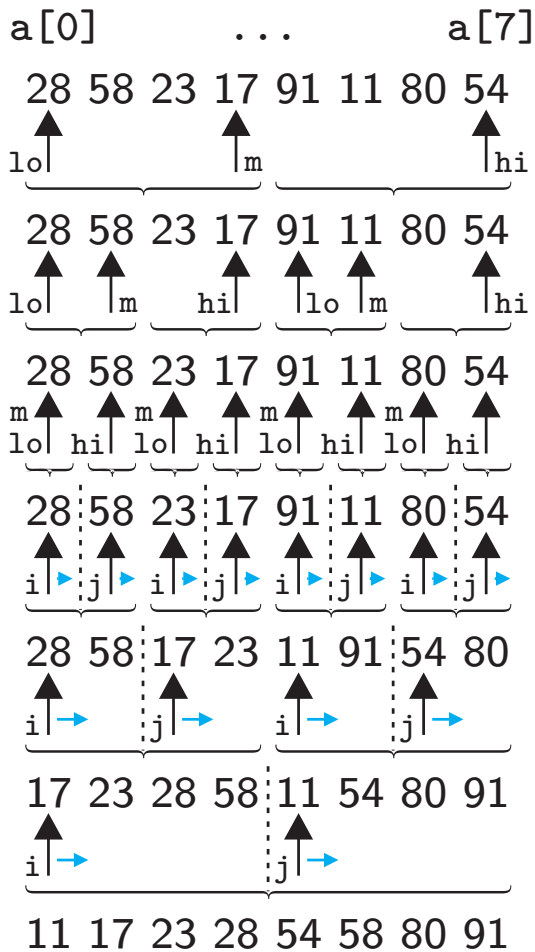
Quicksort...

- Eigenschaften: in-situ, nicht stabil
- Quadratische Laufzeit $\mathcal{O}(n^2)$ im Worst-Case
  - Listen werden nicht in etwa gleich große Hälften aufgeteilt
  - Schlimmstenfalls wird immer nur ein Element „abgespalten“
  - Wenn Sortierung vorhanden ist und Pivot-Element schlecht gewählt wird
- Im Schnitt $\mathcal{O}(n \log n)$ und in der Praxis sehr schnell

Quicksort...

```
public static void quickSort(int[] a) {
  quickSort(a, 0, a.length - 1);
}

private static void quickSort(int[] a, int lo, int hi) {
  if (lo < hi) {  // more than one element
    int pivot = a[lo]; int left  = lo + 1; int right = hi;
    do {
      while ((left <= hi) && (a[left] <= pivot)) { ++left; }
      while ((right > lo) && (a[right] > pivot)) { --right; }

      if (left < right) {
        swap(a, left, right);
      }
    } while (left < right);

    // pivot to final position (now a[right] <= pivot)
    swap(a, lo, right);

    quickSort(a, lo, right - 1);  // rec. sort left part
    quickSort(a, right + 1, hi);  // rec. sort right part
  }
}
```

Mergesort

- Sortiere rekursiv die linke und die rechte Hälfte des Arrays unabhängig voneinander (Divide)
- Kombiniere beide Ergebnisse durch einen linearen Durchlauf (Conquer)
- Laufzeit in jedem Fall $\mathcal{O}(n \log n)$
- Eigenschaften: nicht in-situ, stabil
- ```
  public static void mergeSort(int[] a) {
    mergeSort(a, 0, a.length - 1);
  }

  private static void mergeSort(int[] a, int lo, int hi) {
    if (lo < hi) {  // more than one element
      int m = (lo + hi) / 2;
      mergeSort(a, lo, m);
      mergeSort(a, m + 1, hi);
      merge(a, lo, m, hi);
    }
  }
  ```
  ```

Mergesort. . .

- Beispiel



Mergesort...

```
private static void merge(int[] a, int lo, int m, int hi) {
  int len = hi - lo + 1;
  int[] buffer = new int[len];

  // copy both halfs into a helper array
  System.arraycopy(a, lo, buffer, 0, len);

  int i = 0, j = m  - lo + 1;  // cursors for buffer
  int k = lo;                  // cursor for a
  // copy each time the least element back into a
  while ((i <= m - lo) && (j <= hi - lo)) {
    if (buffer[i] <= buffer[j]) { a[k++] = buffer[i++]; }
    else { a[k++] = buffer[j++]; }
  }

  // if available, copy rest of the left half back
  while (i <= m - lo) { a[k++] = buffer[i++]; }
  // if available, rest of right half is already in the right place in a
}
```

- Alternativ zur Aufteilung in linke Hälfte und rechte Hälfte
  - Teilarrays mit Elementen bzgl. gerader/ungerader Position im Array füllen

Weitere Sortierverfahren

- Heapsort
  - Siehe Datenstruktur PriorityQueue
  - So lange `poll` aufrufen und Ergebnis hinten an ein neues Array anhängen bis PQ/OrignalArray leer ist
    - Alternativ entgegengesetzt sortieren und hinten in frei gewordene Position im Array schreiben
  - $\mathcal{O}(n^2)$ mit naiver PQ (= Selectionsort)
  - $\mathcal{O}(n \log n)$ mit Heap
  - Eigenschaften: in-situ, nicht stabil
- Bucketsort
  - Für jeden möglichen Wert muss ein Bucket bereit gestellt werden
    - `boolean[] bucket = new boolean[max(a)];  // max(...) to be defined`
      `Arrays.fill(bucket, false);`
    - `for (int elem : a) { bucket[elem] = true; // have it }`
  - Durchlauf des Arrays liefert Sortierung
    - Indizes von Zellen mit `true` werden dabei ausgegeben, Indizes mit `false` nicht
  - Nur bei diskreten Werten und bei beschränktem Wertebereich möglich
  - Wenn Einträge nicht einmalig sind braucht man ein `int`-Array zum Zählen
    - $\rightarrow$ Countingsort
  - Laufzeit $\mathcal{O}((\max(a) - \min(a)) + n)$
    - Macht Sinn wenn $n \gtrapprox \max(a) - \min(a)$
    - Dann Sortieren linear
  - Eigenschaften: nicht in-situ, nicht stabil

9. Ausnahmebehandlung

Exceptions

- Programmierfehler sind unvermeidlich
  - Division durch 0, Index bei Arrayzugriff außerhalb der Grenzen, etc.
- Was passiert nachdem ein Fehler aufgetreten ist?
  - Beendigung des Programms (*die*)
  - Fortführung (*recover*)
    - Aber nicht mit falschen Werten!
- C-Stil: Anzeige eines Fehlers durch speziellen Rückgabewert z. B. `null`
- In JAVA wird *Exception* erzeugt (*geworfen*)
  - Kontrollierte Ausnahmen
    - Exceptions für Benutzungsfehler, die abgefangen werden müssen
    - `FileNotFoundException`, `InterruptedException`, ...
  - Nicht kontrollierte Ausnahmen
    - RuntimeExceptions für Benutzungsfehler, die abgefangen werden können
    - `ArrayIndexOutOfBoundsException`, `IllegalArgumentException`, `NumberFormatException`, ...
- Ohne RuntimeExceptions Overkill (Programmieraufwand, Performance)
- Es gibt auch Fehler, die gar nicht abgefangen werden können/sollen
  - Stromausfall
  - Softwarefehler (*Error*)
- *Throwable* als Oberbegriff für Exception und Error

Fehler erkennen

```
public void myMethod(...)
    throws XxxException, YyyException, ... {
  ⋮
  if (...) {
    throw new XxxException();
  }
  ⋮
  if (...) {
    throw new YyyException("message");
  }
  ⋮
}
```

- `throw` und `throws` sind Schlüsselwörter
- API definiert viele Exceptions
  - Passende wählen (nicht zweckentfremdet!)
- Oder: Definition eigener Exceptions
  - Exception die gefangen werden muss
    `public class XxxException extends Exception { ... }`
  - Exception die gefangen werden kann
    `public class YyyException extends RuntimeException { ... }`

Fehler erkennen...

- Bei eigenen (Runtime-)Exceptions müssen mindestens 2 Konstruktoren bereitgestellt werden

```
public class MyException extends Exception {
  public MyException() {
    super();
  }

  public MyException(String message) {
    super(message);
  }

  ...
}
```

  - Schlüsselwort `super` ruft hier jeweils passenden Konstruktor der Klasse `Exception` bzw. `RuntimeException` auf

Fehler behandeln

- ```
  BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
  try {
    String s = in.readLine();
    int i = Integer.parseInt(s);
    System.out.println("ok");
  } catch (IOException e) {
    // must be treated
    System.err.println("read problems");
  } catch (NumberFormatException e) {
    // can be treated
    System.err.println("entry is no integer");
  } finally {
    // optional
    System.out.println("anyway");
  }
  ```
- Statements nachdem eine Methode eine Exception geworfen hat werden nicht mehr ausgeführt
- Mehrere `catch`-Blöcke möglich
  ```

Fehler behandeln . . .

- Anweisungen im `finally`-Block werden immer ausgeführt
  - Wenn `try`-Block ohne Auftreten einer Ausnahme normal beendet wurde
  - Wenn eine Ausnahme in einem `catch`-Block behandelt wurde
  - Wenn auftretende Ausnahme von keinem `catch`-Block behandelt wurde
  - Wenn der `try`-Block durch `break`, `continue` oder `return` verlassen wurde
- Z. B. zum Schließen von Dateien
- Wenn `finally`-Block vorhanden ist muss nicht zwingend ein `catch`-Block vorhanden sein

Fehler weitergeben

- Nach oben in der Aufrufhierarchie

```
public int input() throws IOException, NumberFormatException {
  BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
  String s = in.readLine();
  int i = Integer.parseInt(s);
  System.out.println("ok");
  return i;
}

public void caller() throws IOException {
  try {
    input();
  } catch (NumberFormatException e) { ... }
}
```

- Auch hier werden Statements nachdem eine Methode eine Exception geworfen hat nicht mehr ausgeführt
- Aufrufende Methode muss sich um Fehler kümmern
- `IOException` am besten nie fangen bzw. auch aus `main` werfen
  - Man kann nicht sinnvoll darauf reagieren, z. B. Hardware-Defekt
  - Kontrolliertes Programmende noch das Beste
- Errors und RuntimeExceptions brauchen zum Weiterwerfen nicht mit `throws` deklariert werden

Fehler ausgeben

- `System.err.println` schreibt auf *Standard-Fehler-Ausgabe*
  - *Standard-Ausgabe* `System.out.println`
  - Können unterschiedlich sein
  - `java MainClass 2> error.log` schreibt (unter Unix) Fehlermeldungen in eine Datei
- Exceptions sind auch nur Objekte
- Spezielle Methoden
  - `String getMessage()` liefert Fehlertext wenn im Konstruktor der Exception einer übergeben wurde, `null` sonst
  - `String toString()` liefert Name der Exception und evtl. übergebenen Fehlertext
  - `printStackTrace()` schreibt Ergebnis von `toString()` plus Informationen wo Fehler (Code-Zeile) bei welchen Aufrufen aufgetreten ist
- Am besten JAVA-Logging API benutzen (führt hier zu weit)
- Nicht abgefangene (Runtime-)Exceptions zeigt Laufzeitsystem mit `printStackTrace()` in der Aufrufkonsole an und beendet das Programm

Assertions

- Beim Einfügen eines Elements nach einem anderen in Liste

```
public void add(Node it, int dataValue) {
  assert it != null : "cannot insert after null";
  ⋮
}
```

- Option `-ea` beim Aufruf der JVM nicht vergessen
  - Ansonsten werden `assert`-Anweisungen ignoriert
  - Performance in produktiv laufenden Programmen
- Zeile mit Schlüsselwort `assert` ist äquivalent zu

```
if (it == null) {
  throw new AssertionError("cannot insert after null");
}
```

  - So Prüfung allerdings immer
- Errors können aber sollen nicht gefangen werden
  - Schwerwiegende Fehler können i. d. R. nicht von einem Anwendungsprogramm behandelt werden
  - Programm wird (kontrolliert) beendet
- Hier besser `IllegalArgumentException` werfen