Objektorientierte Programmierung

Kapitel 1: Einführung

Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2014/15

http://www.informatik.uni-halle.de/~brass/oop14/

Inhalt

1. Computer, Programme
   - Computer, Hauptspeicher, Maschinensprache
   - Assembler, Texte, Editoren
   - Betriebssystem, Dateien, Verzeichnisstruktur
2. Programmiersprachen
   - Historische Bemerkungen
   - Compiler, Interpreter, Java Virtual Machine (Bytecode)
3. Erstes Programm
   - Minimales “Hello, World”-Programm
4. Compiler-Benutzung
   - Ausführung des Programms
   - Umgang mit Syntaxfehlern
   - Benutzung einer IDE am Beispiel von Eclipse

Computer: Minimal-Hardware (1)

- Der Kern eines Computers, der die Programme ausführt, heißt CPU oder Prozessor.

  Z.B. Pentium 4, UltraSparc IV. CPU = Central Processing Unit.

- Die auszuführenden Befehle entnimmt der Prozessor dem Hauptspeicher (RAM).

  RAM = Random Access Memory. Der Hauptspeicher enthält sowohl Programme (Maschinenbefehle) als auch Daten (die von den Programmen verarbeitet werden).

- Der Hauptspeicher besteht aus vielen Speicherzellen, die über Adressen (z.B. von 0 bis 268 435 455 bei 256 MByte) angesprochen werden (älterer PC, heute typisch 4 GByte).

  In der Informatik steht "M"/"Mega" meist für 2^{20}, d.h. $1024 * 1024$, also etwas mehr als eine Million. "G"/"Giga" entsprechend für 2^{30} (ca. 1.07 Milliarden).

Computer: Minimal-Hardware (2)

- Jede Speicherzelle enthält ein Byte (bestehend aus 8 Bits, die jeweils 0 oder 1 sein können, dies ergibt 256 verschiedene Werte).

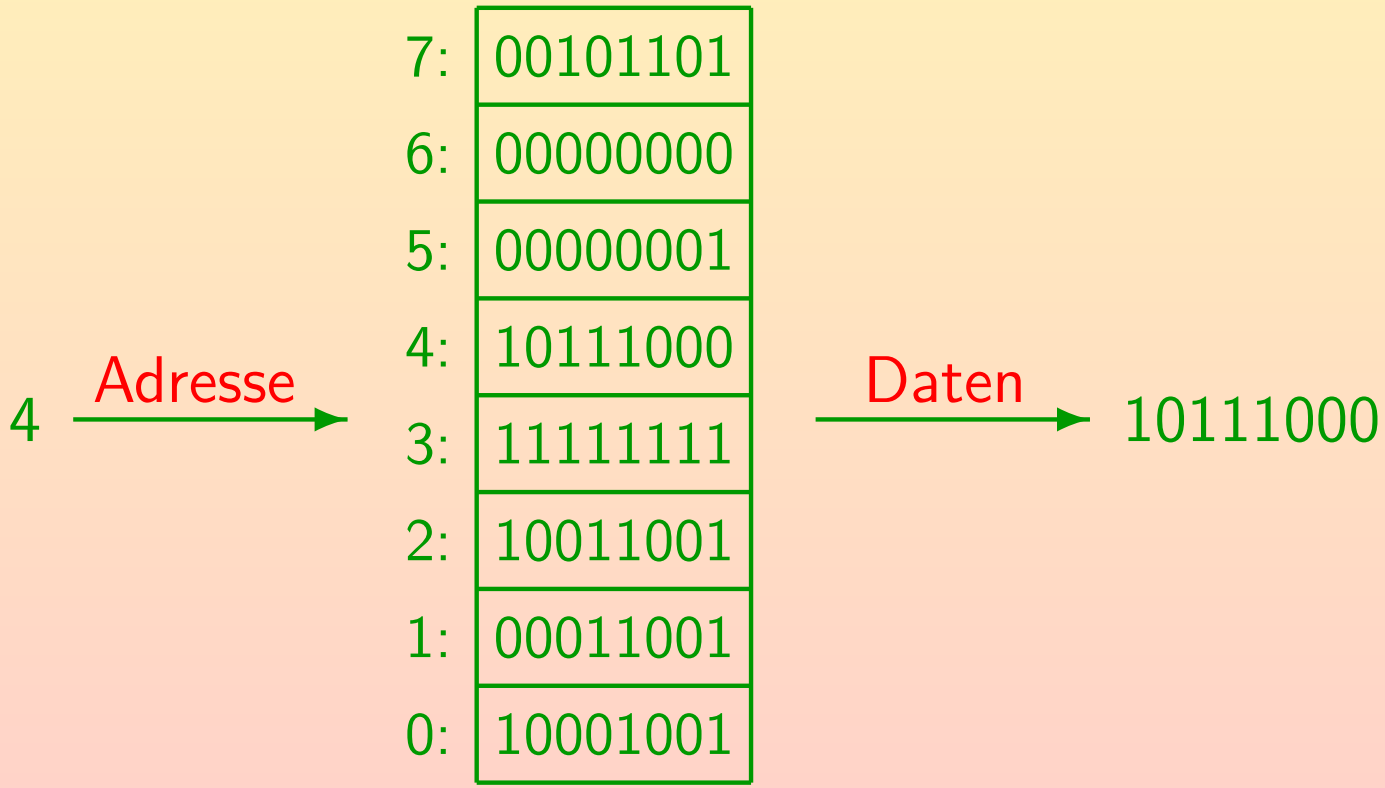
  Man kann Bytes zu größeren Einheiten (Worte) zusammenfassen, manche CPUs können auch einzelne Bits ansprechen. Deswegen kann man nicht genau sagen, was eine einzelne Speicherzelle ist.

- Man kann sich den Hauptspeicher also wie einen Schrank mit vielen Schubladen vorstellen. In jeder Schublade steckt eine Zahl zwischen 0 und 255.

  Informatiker beginnen häufig mit 0 zu zählen, da es ja eigentlich Folgen von Nullen und Einsen sind, und 00000000 einfach 0 entspricht.
  Das ist eine Interpretationsfrage. Z.B. auch möglich: -128 bis +127.
  Man kann die Bitmuster auch ganz anders interpretieren, z.B. als Buchstaben oder Maschinenbefehle (s.u.).

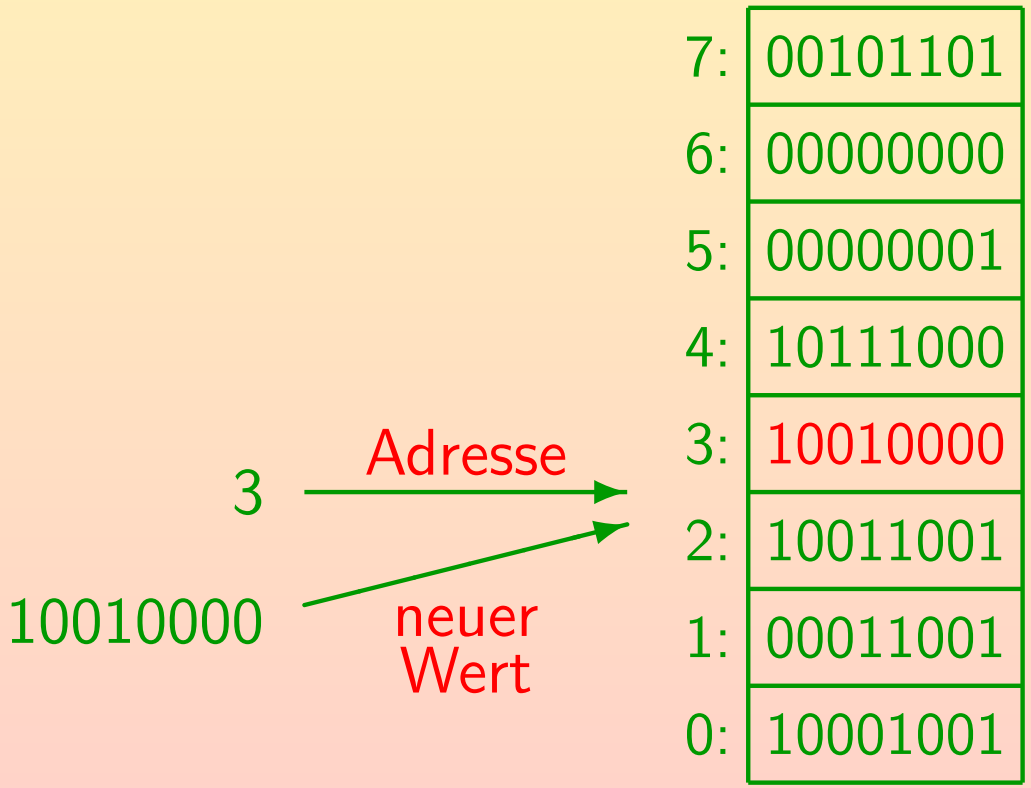
Computer: Minimal-Hardware (3)

Lesezugriff auf ein Byte des Hauptspeichers:



Computer: Minimal-Hardware (4)

Schreibzugriff auf ein Byte des Hauptspeichers:



Maschinencode (1)

- Die CPU enthält einen “Instruction Pointer” (oder “Program Counter”), der die Adresse des nächsten auszuführenden Befehls (in “Maschinencode”) enthält.
- Sie holt sich also den Wert aus dieser Speicherzelle.

  Eventuell auch die Werte aus einigen folgenden Speicherzellen: Viele Befehle sind länger als 1 Byte (z.B. 4–6 Byte). Typischerweise erkennt sie am ersten Byte des Befehls, wieviele Bytes noch nötig sind.
- Sie führt diesen Befehl aus, erhöht den Instruction Pointer, und holt sich den nächsten Befehl.

  Einige Befehle sind Sprungbefehle: Dann würde der Instruction Pointer auf einen neuen Wert gesetzt und nicht einfach der folgende Befehl geholt. Dies kann auch abhängig von Bedingungen geschehen.

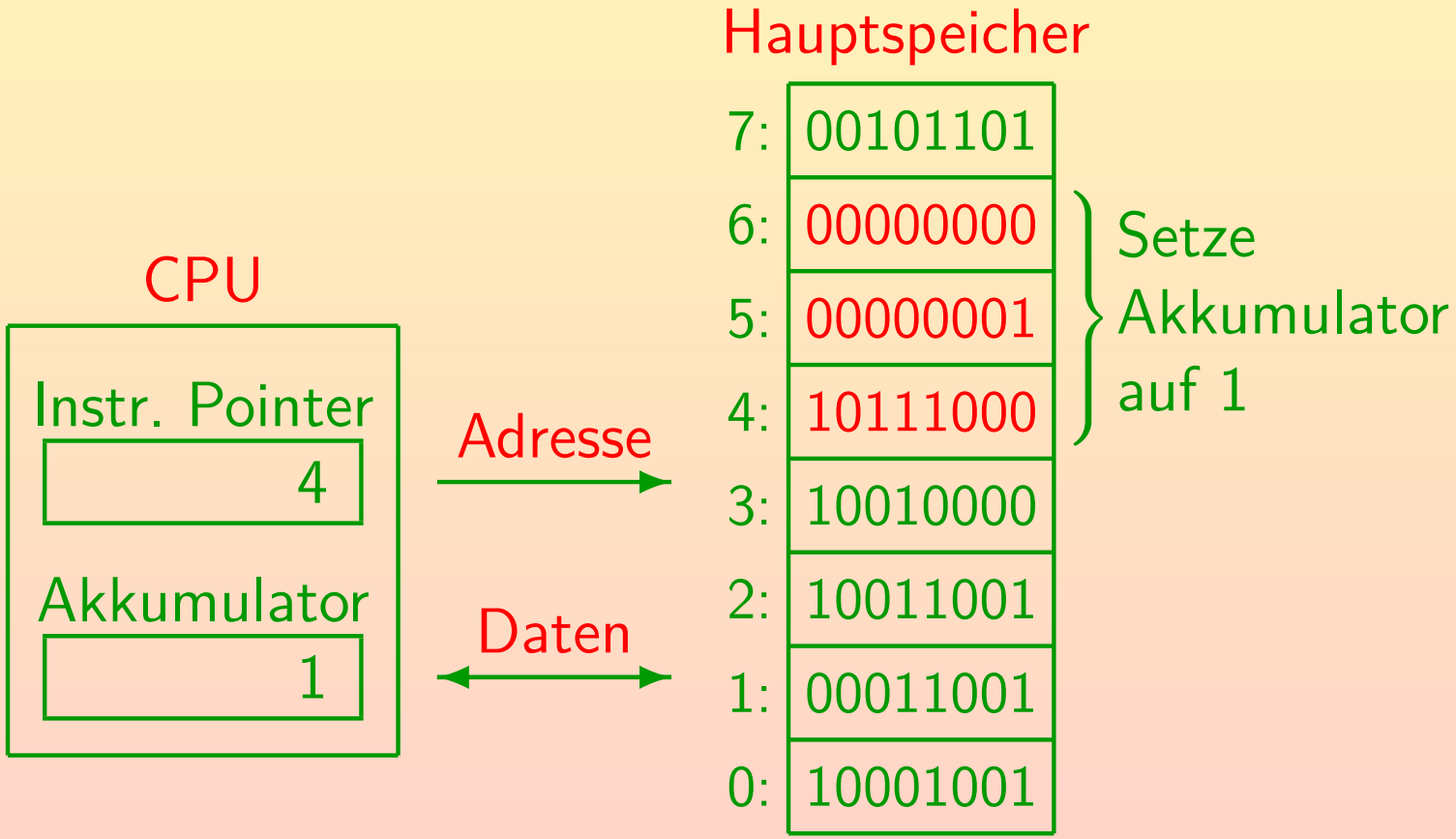
Maschinencode (2)

- Die einzelnen Befehle sind sehr einfach, z.B.
  - Lade den Inhalt von Speicherzelle X in eine spezielle Speicherzelle in der CPU ("Akkumulator").

    Viele CPUs haben mehr als eine solche Speicherzelle. Sie werden Register genannt. Es gibt aber normalerweise nur wenige Register. Die Register sind häufig größer als ein Byte, z.B. 4 Byte (32 Bit) oder 8 Byte (64 Bit).
  - Addiere den Inhalt von Speicherzelle X zum aktuellen Inhalt des Akkumulators hinzu.
  - Springe zu Speicherzelle X.

    Dies setzt also den Wert im Instruction Pointer, der auch ein spezielles Register der CPU ist. Der nächste Befehl wird dann von Speicherzelle X geholt.

Maschinencode (3)



Assembler

- Programme bestehen also letztendlich aus Folgen von Nullen und Einsen im Hauptspeicher.
- Anfangs musste man tatsächlich in dieser Form programmieren (Maschinensprache).
- Dann wurden Assemblersprachen (kurz "Assembler") erfunden. Sie sind ein 1:1 Abbild der Maschinensprache, aber mit für den Menschen lesbaren Befehlen:

```
mov AX, 1
```

  Speichere den Wert 1 in das Register AX, den Akkumulator.

  `mov` steht kurz für "move": Bewege den Wert 1 nach AX.

- Der Assembler ist ein Programm, das solche Programme (Texte) in die internen Bitmuster für die Befehle übersetzt.

Texte, Interpretation von Bitmustern

- Die Texte können auch im Hauptspeicher des Rechners repräsentiert werden.
- Dazu interpretiert man die Bytes (Bitmuster) einfach als Buchstaben/Zeichen. Z.B. wäre ein "a" nach dem ASCII-Code das gleiche Bitmuster wie die Zahl 97.

  ASCII = American Standard Code for Information Interchange.
- Auf Maschinenebene kann das gleiche Bitmuster also ganz unterschiedlich interpretiert werden.

  Das Bitmuster in der Speicherzelle, auf die der "Instruction Pointer" zeigt, wird als Maschinenbefehl für die CPU interpretiert. Ansonsten hängt die Bedeutung an der Programmierung, wie man die Daten verarbeitet.

Editor, Dateien

- Texte (z.B. ein Assembler-Programm) können mit einem weiteren Programm, dem Editor, eingegeben und geändert werden (über die Tastatur).
- Der Inhalt des Hauptspeichers geht verloren, wenn der Computer ausgeschaltet wird.
- Daher wird man den Text bzw. das Programm auf die Festplatte (oder einfach Platte, engl. Disk) abspeichern.
- Die Daten auf der Platte werden in Form von Dateien (Folgen von Bytes) verwaltet.

  Wenn Sie die Datei im Editor öffnen, kopiert er die Daten von der Platte in den Hauptspeicher. Wenn Sie auf “Speichern”/“Save” klicken, werden die ggf. veränderten Daten zurück in die Datei geschrieben.

Betriebssystem

- Die Verwaltung von Dateien ist eine der Funktionen des Betriebssystems (z.B. Windows, Linux).
- Das Betriebssystem
  - enthält eine Bibliothek von häufig verwendeten Funktionen (Programmcode),

    Programme können über einen Betriebssystemaufruf Daten aus einer Datei in den Hauptspeicher laden bzw. umgekehrt in die Datei schreiben. Sie brauchen nicht selbst Befehle zur Plattensteuerung zu enthalten.
  - ist eine Kontrollinstanz,

    Ein Rechner wird eventuell von mehreren Benutzern verwendet, dann darf man z.B. nicht beliebig auf Dateien anderer Benutzer zugreifen.
  - ermöglicht das Laden von Programmen aus Dateien (von der Platte) in den Hauptspeicher, um sie dort zu starten.

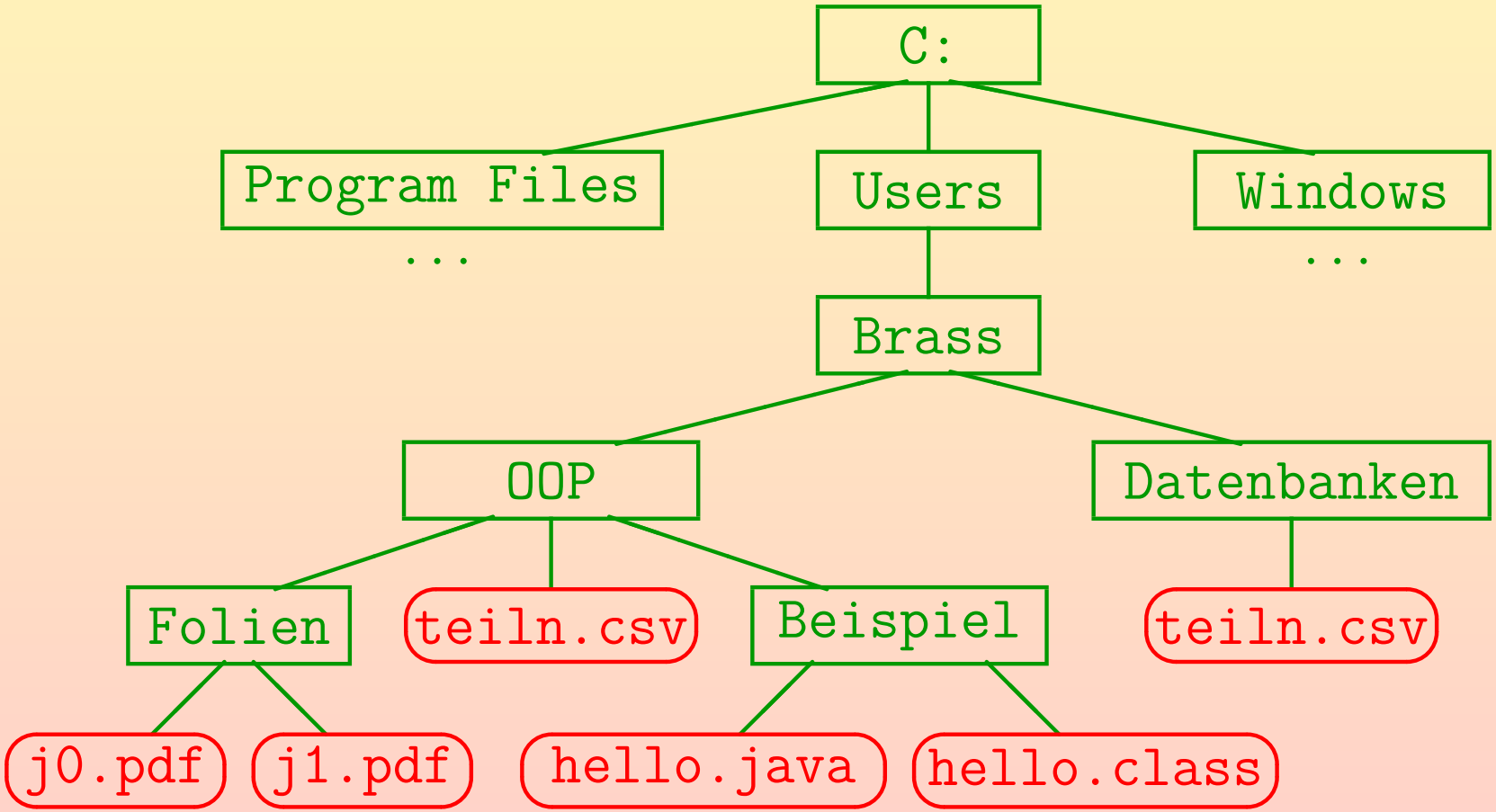
Dateien, Verzeichnisstruktur (1)

- Dateien haben einen Namen (Dateinamen).
- Dateien werden in Ordnern (Dateiverzeichnissen, Directories) strukturiert.

  Ordner und Dateiname identifizieren eindeutig die Datei.
- Ordner können selbst wieder Ordner enthalten, so dass eine hierarchische Struktur entsteht.

  In der Informatik ist so eine Datenstruktur als Baum bekannt. Allerdings wachsen in der Informatik die Bäume verkehrt herum: Die Wurzel (Hauptverzeichnis) wird oben dargestellt, die Verzweigung in Unterverzeichnisse und einzelne Dateien geschieht nach unten.
  Beim Betriebssystem Windows stehen auf oberster Ebene die Laufwerke, wie z.B. `C:`. Beim Betriebssystem UNIX (Linux, Solaris, ...) gibt es nur ein Hauptverzeichnis. Laufwerke können bei UNIX an beliebiger Stelle als Unterverzeichnisse in die Hierarchie integriert werden.

Dateien, Verzeichnisstruktur (2)



Dateien, Verzeichnisstruktur (3)

- Man kann Dateien über den vollständigen Namen (mit allen übergeordneten Ordnern) identifizieren:
  `C:\Users\Brass\OOP\Beispiel\Hello.java` (Windows)
  `/home/brass/OOP/Beispiel/Hello.java` (Unix)

  Solche Dateibezeichnungen heißen absolute Pfadnamen. Es gibt für jedes in Ausführung befindliche Programm ("Prozess") ein aktuelles Verzeichnis, von dem aus relative Pfadnamen möglich sind: Z.B. `Hello.java`, falls gerade `C:\Users\Brass\OOP\Beispiel` das aktuelle Verzeichnis ist, oder `..\Beispiel\Hello.java`, falls `C:\Users\Brass\OOP\Folien` akt. Verzeichnis.

- Es ist üblich, dass Dateinamen eine durch Punkt abgetrennte Endung haben ("Extension"), die die Art der Daten in dem Dokument anzeigt, z.B.:
  - `.pdf`: Textdokument (für Acrobat Reader / Evince).
  - `.java`: Java Programm (Eingabe für Compiler `javac`).

Inhalt

1. Computer, Programme
   - Computer, Hauptspeicher, Maschinensprache
   - Assembler, Texte, Editoren
   - Betriebssystem, Dateien, Verzeichnisstruktur
2. Programmiersprachen
   - Historische Bemerkungen
   - Compiler, Interpreter, Java Virtual Machine (Bytecode)
3. Erstes Programm
   - Minimales “Hello, World”-Programm
4. Compiler-Benutzung
   - Ausführung des Programms
   - Umgang mit Syntaxfehlern
   - Benutzung einer IDE am Beispiel von Eclipse

Höhere Programmiersprachen (1)

- Assembler-Sprachen hatten drei Nachteile:
  - Die Programme liefen nur mit dem CPU-Typ, für den sie geschrieben wurden (nicht portabel).
  - Die Befehle der CPUs sind sehr einfach, kompliziertere Programme also entsprechend lang.

    Es gibt Untersuchungen, nach denen Programme in der höheren Programmiersprache C dreimal kürzer sind als äquivalente Assembler-Programme, und auch entsprechend schneller entwickelt werden (d.h. Programmierer brauchen in diesen Sprachen die gleiche Zeit pro "Line of Code"). Die Produktivität verdreifachte sich also beim Schritt von Assembler zu der höheren Programmiersprache C.
  - Die Programme sind schlecht strukturiert und unübersichtlich, schwierig zu warten.

    Es geschehen auch leicht Fehler, da der Assembler alles zulässt.

Höhere Programmiersprachen (2)

- Daher wurden höhere Programmiersprachen erfunden, die erste erfolgreiche war Fortran (1954–57).

  Es hat auch etwas frühere Versuche gegeben.

  Fortran = FORmula TRANslator.

- Insbesondere konnte man jetzt die übliche mathematische Notation für Formeln verwenden, z.B.

```
X = 3 * Y + Z
```

  Dies entspricht einer Reihe von Maschinenbefehlen: Zuerst muss man den Wert von `Y` in den Akkumulator laden (der Variablen `Y` wird eine bestimmte Hauptspeicher-Adresse zugeordnet — man kann so mit Namen statt Adressen arbeiten). Dann muss man den Inhalt des Akkumulators mit 3 multiplizieren, anschliessend `Z` aufaddieren, und zum Schluß den aktuellen Inhalt des Akkumulators in die für `X` reservierte Speicherzelle schreiben.

Höhere Programmiersprachen (3)

- Java gehört zur Familie der C-ähnlichen Sprachen, und ist insbesondere von C++ beeinflusst.

  C ist seinerseits ein Ableger der Algol-Familie ("Algorithmic Language").

- C wurde 1969–1973 von Dennis Ritchie in den Bell Labs entwickelt (kleinere Änderungen 1977-1979), das wichtige Lehrbuch von Kernighan/Ritchie erschien 1978.

  Siehe: [http://cm.bell-labs.com/cm/cs/who/dmr/chist.html]

  C wurde parallel mit dem Betriebssystem UNIX entwickelt (zur Implementierung von UNIX). Die für UNIX zuerst benutzte PDP-7 hatte 8K 18-bit Worte RAM.

- C erreichte große Verbreitung.

  Es war eine kompakte/kleine Sprache, deren Befehle sehr direkt in Befehle der CPU übersetzt werden konnten (hardware-nah, effizient).

- Der ANSI-Standard für C erschien 1989 (ISO C90).

Kurze Java-Geschichte

- Für große Programme hat sich eine objektorientierte Struktur als meistens sehr übersichtlich herausgestellt. Als erste objektorientierte Programmiersprache gilt heute Simula-67.

  Weitere Meilensteine waren Smalltalk-80 und C++.

- Zur Steuerung/Vernetzung von Haushaltselektronik entwickelte James Gosling 1991/92 die objektorientierte Programmiersprache “Oak” (später in Java umbenannt).

- Das ursprüngliche Projekt war nicht erfolgreich, aber 1993 begann das World Wide Web begann seinen Siegeszug.

- Die Sprache Java wurde nun verwendet, um “Applets” über das Internet zu verteilen und im Browser auszuführen.

  Am 23.05.1995 stellt Sun den in Java geschriebenen Browser “HotJava” auf der “SunWorld” vor. Am 23.01.1996 erschien das JDK 1.0.

Compiler

- Programme sind also spezielle Texte (Folgen von Zeichen).

  Diese Texte müssen den Regeln einer Programmiersprache (wie z.B. Java) folgen. Z.B. hat Java eine Grammatik (recht einfach und absolut präzise).

- Solche Texte können mit Hilfe eines Programms, des Compilers, in Maschinensprache übersetzt werden.

  Der erste Compiler musste natürlich in Assembler geschrieben werden.
  "compile": zusammentragen, zusammenstellen (Maschinencode aus Mustern, Bibliotheken).

- Erst dadurch werden die Programme ausführbar:
  Die CPU selbst versteht ja nur Maschinenbefehle.

- Im Laufe der Zeit wurden viele Programmiersprachen vorgeschlagen (und Compiler für diese Sprachen entwickelt).

Begriffe

- Ein Algorithmus ist ein Verfahren, mit dem eine Aufgabe gelöst werden soll.

  Ein Algorithmus ist unabhängig von einer speziellen Programmiersprache.
  Z.B. wurden viele Algorithmen zum Sortieren vorgeschlagen.
  Kochrezepte, Bauanleitungen, Spieltaktiken sind ähnlich zu Algorithmen (in der Informatik aber eher nicht präzise genug).

- Einen Algorithmus kann man in einer Programmiersprache formal aufschreiben ("codieren").
- Quellcode (engl. "source code") ist die Eingabe für den Compiler (z.B. ein Programm in Java).
- Ziel der Übersetzung ist ein ausführbares Programm (Maschinenbefehle für die Ziel-Hardware).

Interpreter

- Eine weitere Möglichkeit, Programme eine höheren Programmiersprache auszuführen, sind Interpreter (z.B. typisch für die Sprachen Basic oder Lisp):
  - Sie laden den Programmtext in den Hauptspeicher,
  - verarbeiten ihn ggf. vor,
  - und führen es aus, ohne explizit Maschinensprache zu erzeugen.

    Die ausgeführten Maschinenbefehle sind Teil des Interpreter-Programms. Dies fragt jeweils den nächsten Befehl des gegebenen Programms ab und enthält entsprechende Fallunterscheidungen, in denen alle möglichen Befehle behandelt werden. Gewissermaßen simuliert es so die Ausführung des eigentlichen Programms.
- Unterschied zu Compiler: Keine getrennte Datei für Übersetzungsergebnis, man arbeitet nur mit Quellcode.

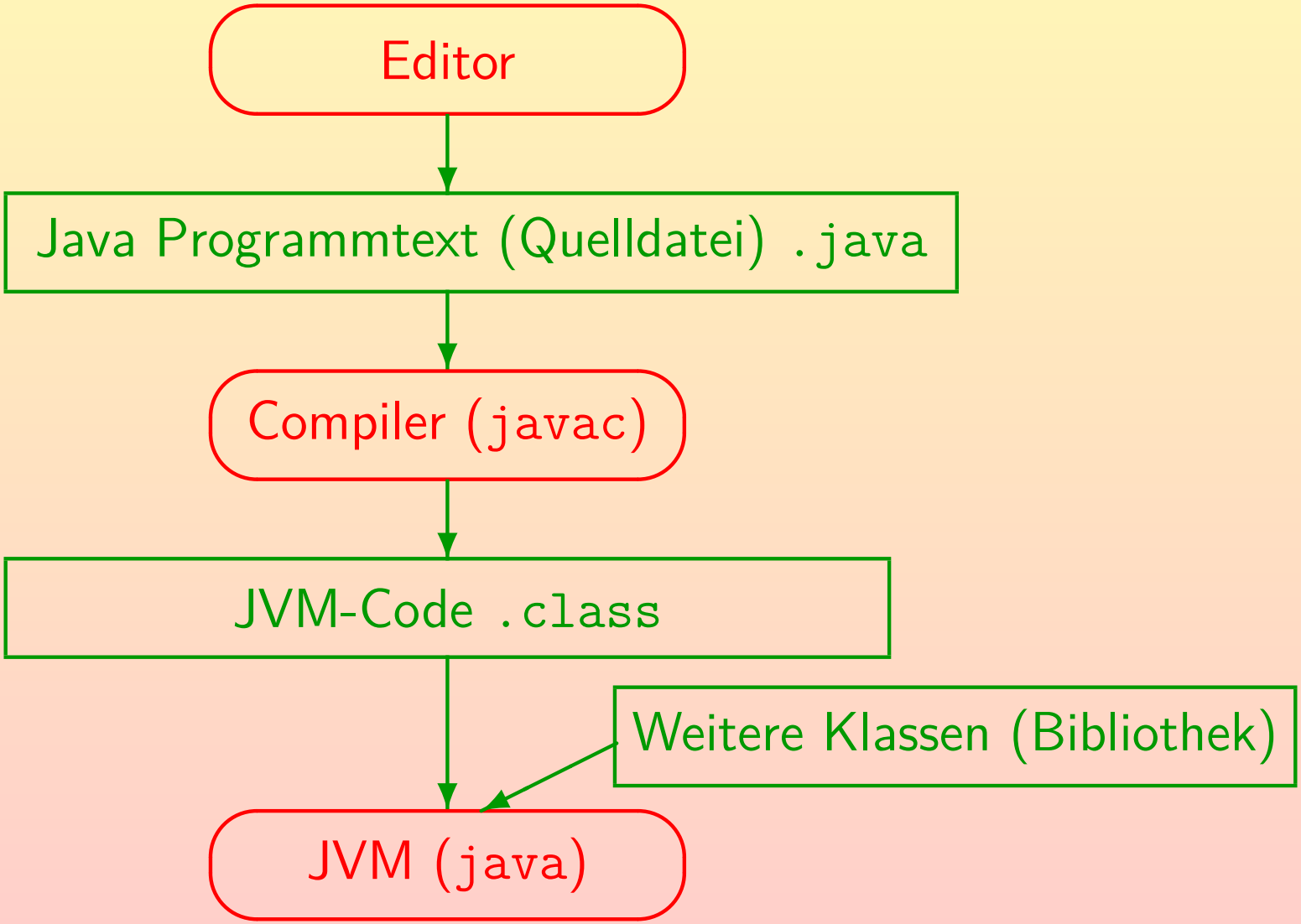
Java (1)

- Bei Java werden normalerweise Compiler und Interpreter kombiniert:
  - Der Compiler übersetzt nicht in Maschinencode, sondern in Instruktionen für die "Java Virtual Machine" (JVM). Dies wird auch "Java Bytecode" genannt.

    Das ist gewissermaßen schon Maschinencode, aber für eine gar nicht in Hardware (Elektronik) existierende Maschine.
  - Jeder unterstützte Rechner hat nun einen Interpreter für die JVM Instruktionen (das ist ja auch nur eine spezielle Programmiersprache).

    Der Interpreter muss natürlich in Maschinensprache vorliegen.
    Die Entwickler haben ihn in C++ geschrieben (250.000 Zeilen) und dann einen C++-Compiler benutzt.
    [http://openjdk.java.net/groups/hotspot/]

Java (2)


Editor
Java Programmtext (Quelldatei) .java
Compiler (javac)
JVM-Code .class
Weitere Klassen (Bibliothek)
JVM (java)

Java (3)

Vorteile dieses Verfahrens:

- Der Software-Anbieter muss das Programm nicht für jede Plattform (Hardware + Betriebssystem) einzeln bereitstellen: `class`-Dateien sind Plattform-unabhängig.

  Besonders wichtig für Ausführung in Browsern (Applets).

- Der Software-Anbieter muss den Quellcode nicht offenlegen.
- Mehr Prüfungen möglich, weniger Sicherheitslücken.

  In Browsern würde man die Ausführung von beliebigem Maschinencode, der von irgendeiner Webseite heruntergeladen wurde, ja gar nicht erlauben.

- Bessere Fehlermeldungen als bei reiner Maschinensprache.
- Oft kompakter als reiner Maschinencode (Dateien kleiner).

Java (4)

Zur Effizienz:

- Schneller als gewöhnlicher Interpreter.

  Ein Teil des Interpretationsaufwands wird zur Ausführungszeit vermieden (z.B. syntaktische Analyse des Quellprogramms).

- Langsamer als Ausführung eines Programms, das schon fertig in Maschinencode vorliegt (erzeugt von Compiler).

- Moderne Implementierungen der Java Virtual Machine enthalten aber einen "Just in Time Compiler", der den Bytecode zur Ausführung doch in Maschinencode übersetzt.

  Wenn die Befehle häufig ausgeführt werden, fällt der Overhead für die Übersetzung nicht mehr ins Gewicht. Es gibt auch Optimierungen, die so leichter werden (gegenüber normalem Compiler). Dafür leichte Verzögerung beim Starten. Insgesamt Laufzeitunterschied nicht mehr groß.

Zur Einschätzung von Java

- Java ist nicht nur eine Programmiersprache, es ist eine Programmier-Plattform.
- Die Bibliotheken, die mit Java mitgeliefert werden, sind mindestens so wichtig wie die Sprache selbst.

  Bibliotheken enthalten fertigen Programmcode für vielerlei Aufgaben, den man in eigenen Programmen aufrufen (mitbenutzen) kann.
- Die Anzahl von Klassen/Interfaces in der Bibliothek stieg von 211 in Version 1.0 auf 3777 in Version 6.

  Während in dieser Vorlesung die Sprache im Vordergrund steht, muss man als versierter Java-Programmierer einen nicht unwesentlichen Teil der Bibliothek kennen. Bei Sprachen wie C++ ist die Standard-Bibliothek im Vergleich zu Java eher klein, Erweiterungen (z.B. `Qt`, u.a. für portable GUI-Programmierung) sind halb-kommerziell und es gibt Konkurrenz.

Inhalt

1. Computer, Programme
   - Computer, Hauptspeicher, Maschinensprache
   - Assembler, Texte, Editoren
   - Betriebssystem, Dateien, Verzeichnisstruktur
2. Programmiersprachen
   - Historische Bemerkungen
   - Compiler, Interpreter, Java Virtual Machine (Bytecode)
3. Erstes Programm
   - Minimales “Hello, World”-Programm
4. Compiler-Benutzung
   - Ausführung des Programms
   - Umgang mit Syntaxfehlern
   - Benutzung einer IDE am Beispiel von Eclipse

"Hello, World" Programm (1)

- Seit dem Buch von Kernighan und Ritchie über die Programmiersprache "C" ist es üblich, als erstes Programm zur Illustration einer Programmiersprache eines zu schreiben, das "`hello, world`" ausgibt.

  Sprache und Buch waren sehr einflussreich und sind bis heute verbreitet.

- Man kann so mit einem minimalen Beispiel erst einmal sehen, dass alles funktioniert.

  Kernighan/Ritchie schreiben: "This is the big hurdle; to leap over it, you have to be able to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy."

- Außerdem hat man eine Basis für Erweiterungen, und kann z.B. weitere Berechnungen in diesen Rahmen einbauen.

"Hello, World" Programm (2)

```
(1)  // Erstes Java-Programm
(2)
(3)  class Hello {
(4)      static public void main(String[] args) {
(5)          System.out.println("Hello, World!");
(6)      }
(7)  }
```

Die Zeilennummern links gehören nicht mit zum Programm. Sie machen es einfacher, sich auf bestimmte Zeilen zu beziehen. In der Datei `Hello.java` stehen nur die anderen (grün gedruckten) Zeichen. Die genaue Einrückung mit beliebig vielen Leerzeichen oder Tabulator-Zeichen ist egal, die Zeilenaufteilung (größtenteils) auch.

Sie brauchen das Programm jetzt noch nicht zu verstehen, alles wird später noch systematisch und ausführlich erklärt. Sie müssen aber lernen, das Programm mit einem Editor einzugeben ("abzutippen"), und mit Compiler und JVM auszuführen. (Die erweiterte Version der Folien enthält einige Erklärungen für Neugierige.)

Weitere Beispiel-Programme

- Das erste Beispiel war eine “Konsolen-Anwendung”.
- Sie wirkt etwas antiquiert, weil sie keine eigene graphische Benutzeroberfläche (GUI: “Graphical User Interface”) hat, sondern nur Zeichen auf ein Terminal-Fenster ausgibt.

  Früher kommunizierte man mit Computern über solche Terminals in Hardware.
- Selbstverständlich kann man auch Programme mit graphischen Benutzeroberflächen in Java schreiben.

  Darin ist Java sogar besonders stark, weil es eine Bibliothek zur portablen Programmierung von Benutzeroberflächen enthält. Allerdings muss man mehr von Java wissen, und größere Bibliotheken kennen, um solche Programme wirklich zu verstehen. Es soll ja nicht (lange) beim stupiden “Abtippen” bleiben.
- Es können auch “Applets” programmiert werden, die in einem Webbrowser laufen. Und “Apps” für Android Smartphones.

Inhalt

1. Computer, Programme
   - Computer, Hauptspeicher, Maschinensprache
   - Assembler, Texte, Editoren
   - Betriebssystem, Dateien, Verzeichnisstruktur
2. Programmiersprachen
   - Historische Bemerkungen
   - Compiler, Interpreter, Java Virtual Machine (Bytecode)
3. Erstes Programm
   - Minimales “Hello, World”-Programm
4. Compiler-Benutzung
   - Ausführung des Programms
   - Umgang mit Syntaxfehlern
   - Benutzung einer IDE am Beispiel von Eclipse

Ausführung des Programms

- Schreiben Sie das obige Programm in die Datei "`Hello.java`".

  Natürlich ohne die Zeilennummern. Sie benötigen einen Editor. Minimale Lösung wäre `notepad`, aber es gibt natürlich viel mächtigere Editoren.

- Rufen Sie den Compiler mit folgendem Befehl auf:

  ```
  javac Hello.java
  ```

  Sie müssen den Befehl in der Eingabeaufforderung, Konsole, "Command Prompt" eingeben (in Windows unter "Zubehör"). Natürlich müssen Sie mit `cd` in das Verzeichnis wechseln, in dem "`Hello.java`" gespeichert ist.

- Führen Sie das Programm aus mit

  ```
  java Hello
  ```

  Beachten Sie, dass Sie die Endung "`.class`" weglassen müssen.

Syntaxfehler (1)

- Falls Sie beim Abtippen des Programms einen Fehler gemacht haben, so dass der Compiler die Eingabe nicht verstehen kann, erhalten Sie eine Fehlermeldung.
- Der Compiler versteht das Programm nur, wenn Sie die Regeln der Java-Syntax exakt befolgen.

  Es kommt auf jedes Zeichen an. Während ein Mensch den Sinn eines Textes trotz einigen Tippfehlern und Ungenaugigkeiten verstehen kann, ist der Compiler eine Maschine ohne Verstand. Er verarbeitet den Text, solange dieser genau den Vorgaben entspricht. Sonst verweigert er die Arbeit.
  Der Vorteil dabei ist, dass es keine Interpretations-Spielräume bleiben.
- Im positiven Fall sagt der Compiler dagegen nichts und erstellt die Datei `Hello.class`.

  Dies ist das Übersetzungsergebnis (die JVM-Maschinenbefehle).
  Später wird erklärt, wie diese dann ausgeführt werden.

Syntaxfehler (2)

- Wenn Sie z.B. das Semikolon in Zeile (5) weglassen, macht der Compiler folgende Ausgabe:

```
Hello.java:5: ';' expected
        System.out.println("Hello, World!")
                                           ^
1 error
```

  Mit etwas Erfahrung wird man das für eine sehr klare Fehlermeldung halten: Der Compiler hat genau an der richtigen Stelle im Programm eine Meldung ausgegeben, die deutlich sagt, was zu tun ist, nämlich dort ein "`;`" einzufügen.

- Korrigieren Sie den Fehler in der Quelldatei "`Hello.java`" und rufen Sie den Compiler erneut auf.

  Sie können das Editor-Fenster offen lassen, aber vergessen Sie nicht, abzuspeichern. Der Compiler liest das Programm ja aus der Datei.

  Gerade als Anfänger braucht man oft mehrere Anläufe, bis ein Programm fehlerfrei durch den Compiler läuft. Mit der Zeit macht man weniger Fehler.

Syntaxfehler (3)

- Leider sind die Fehlermeldungen nicht immer so klar. Schreibt man z.B. `static` groß, so erhält man:

```
Hello.java:4: <identifier> expected
    Static public void main(String[] args) {
           ^
1 error
```

Der Compiler liest das Programm von vorne nach hinten und gibt die Fehlermeldung an der ersten Stelle aus, an der keine gültige Fortsetzung mehr möglich ist. Das Wort "`Static`" wäre als Name einer Klasse möglich, und diese Klasse könnte als Datentyp eines Attributs oder Ergebnis-Datentyp einer Methode angegeben sein, deren Name (Bezeichner, "identifier") hier fehlt. Fügt man aber einen solchen Bezeichner ein (z.B. "`X`"), beschwert sich der Compiler über ein fehlendes Semikolon. Fügt man auch das ein, merkt er endlich, dass er gar keine Klasse "`Static`" kennt: Er analysiert zunächst die reine Syntax und schaut erst später die Bedeutung der Namen nach, dazu kommt er im Fehlerfall nicht mehr.

Syntaxfehler (4)

- Es ist also möglich, dass der Fehler nicht an genau der Stelle liegt, wo er gemeldet wird, sondern davor.

  Bei modernen Compilern kann er nicht dahinter liegen, da das Programm von vorne nach hinten verarbeitet wird, und der Fehler an der ersten Stelle gemeldet wird, wo keine gültige Fortsetzung mehr möglich ist.

- Meist ist er nur kurz davor.

  Theoretisch könnte er beliebig weit davor liegen, aber es muss dann doch eine Beziehung geben, z.B. eine Variable (Programm-Bestandteil), die man hier verwendet, und die man vorher falsch deklariert hat; oder eine Klammer, die man hier schließt, und vorher vergessen hat, zu öffnen.

- Es ist möglich, dass nach der Beseitigung eines Fehlers neue Fehler gemeldet werden.

  Eventuell auch weiter vorne, weil der Compiler den Text in mehreren Durchgängen ("Phasen") bearbeitet.

Syntaxfehler (5)

- Da der Compiler bei einem Fehler nicht aufhört, sondern auch den Rest des Programms noch analysiert, kann es "Folgefehler" geben: Wenn man den ersten Fehler beseitigt hat, verschwinden sie automatisch.

  Damit der Compiler den Rest des Textes weiter analysieren konnte, musste er Annahmen darüber machen, was Sie gemeint haben könnten.
  Wenn diese Annahmen falsch waren, führen sie zu weiteren Fehlermeldungen, weil der Rest des Textes nicht dazu passt.

- Kümmern Sie sich zuerst um den ersten angezeigten Fehler und ignorieren Sie den Rest.

  Früher dauerte ein Compilerlauf lange, eventuell musste man bei Großrechnern auch Stunden warten, bis der Programmlauf ausgeführt wurde.
  Da war es wichtig, nicht nur einen Fehler gemeldet zu bekommen.

Syntaxfehler (6)

- Beispiel: Schlüsselwort "`class`" vergessen:

```
Hello.java:3: class, interface, or enum expected
Hello {
^
Hello.java:4: class, interface, or enum expected
    static public void main(String[] args) {
                       ^
Hello.java:6: class, interface, or enum expected
    }
    ^
```

- Hier gibt es nur einen Fehler, aber drei Fehlermeldungen.

  Lassen Sie sich also von einer langen Liste von Fehlermeldungen nicht entmutigen. Lange Programme sollten Sie in mehreren Ausbaustufen eingeben, und jeweils die Compilierbarkeit testen. So vermeiden Sie, dass Sie wirklich viele Fehler auf einmal haben.

Aus Fehlern lernen!

- Es ist wichtig, Fehler wirklich aufzuklären: Geben Sie sich nicht damit zufrieden, dass es zufällig funktioniert.

  Wenn Sie den Fehler nicht verstanden haben, werden Sie ihn wieder machen. Außerdem funktioniert das Programm vielleicht nur für die eine Eingabe, die Sie ausprobiert haben. Es bleibt auch ein Gefühl der Unsicherheit.

- Wenn Sie eine unverständliche Fehlermeldung erhalten, kopieren Sie sich das Programm (um ggf. später zu fragen).

  Ansonsten ist eine übliche Methode, wenn man eine Fehlermeldung nicht versteht, dass man das Programm modifiziert. Die Hoffnung dabei ist, dass entweder der Fehler verschwindet, oder man beim modifizierten Programm eine Fehlermeldung bekommt, die man besser versteht. Es kann dann aber passieren, dass man am Ende nicht mehr weiss, was genau das ursprüngliche Programm war. Dann wird es natürlich schwierig, den Fehler aufzuklären.

Entwicklungsumgebungen

- Alternative zu Einzelwerkzeugen (Editor, Compiler, ...): IDE ("Integrated Development Environment").
- Vorteile IDE:
  - Wirkt moderner, macht mehr automatisch, enthält Hilfen.
- Nachteile IDE:
  - Funktionsumfang kann zu Beginn erschlagen.

    Es gibt viel, was man am Anfang nicht unbedingt braucht, und was eher im Wege steht (Konzentration auf das Wesentliche fällt schwerer).
  - Man muss auch ohne die Hilfen programmieren können.

    Die Hilfen dienen zur Produktivitätssteigerung für den Profi. Wenn man damit fehlendes Wissen über die Programmiersprache ausgleicht, erwirbt man dieses Wissen eventuell langsamer. Klausur ohne solche Hilfe!
  - Man kann nicht mit ggf. bekanntem Editor arbeiten.

Schlussbemerkung

- Machen Sie sich keine Sorgen, wenn Sie noch nicht alles voll verstanden haben.
- Ziel war es, Sie in die Lage zu versetzen, Programme auszuführen. Dazu brauchen Sie:
  - Betriebssystem (z.B. Windows oder Linux mit Konsole)
  - Editor (z.B. `notepad`, `notepad++`, `gedit`)
  - Compiler/Übersetzer (`javac`)
  - Bytecode-Interpreter: Java Virtual Machine (`java`)
  - Optional alles in einer IDE (`Eclipse`, `Netbeans`, `BlueJ`)

  Am Ende sollten Sie beide Möglichkeiten beherrschen.
- Außerdem: Erster Eindruck von Java-Programmen.

  Es wird später alles noch ausführlicher und präziser erklärt.