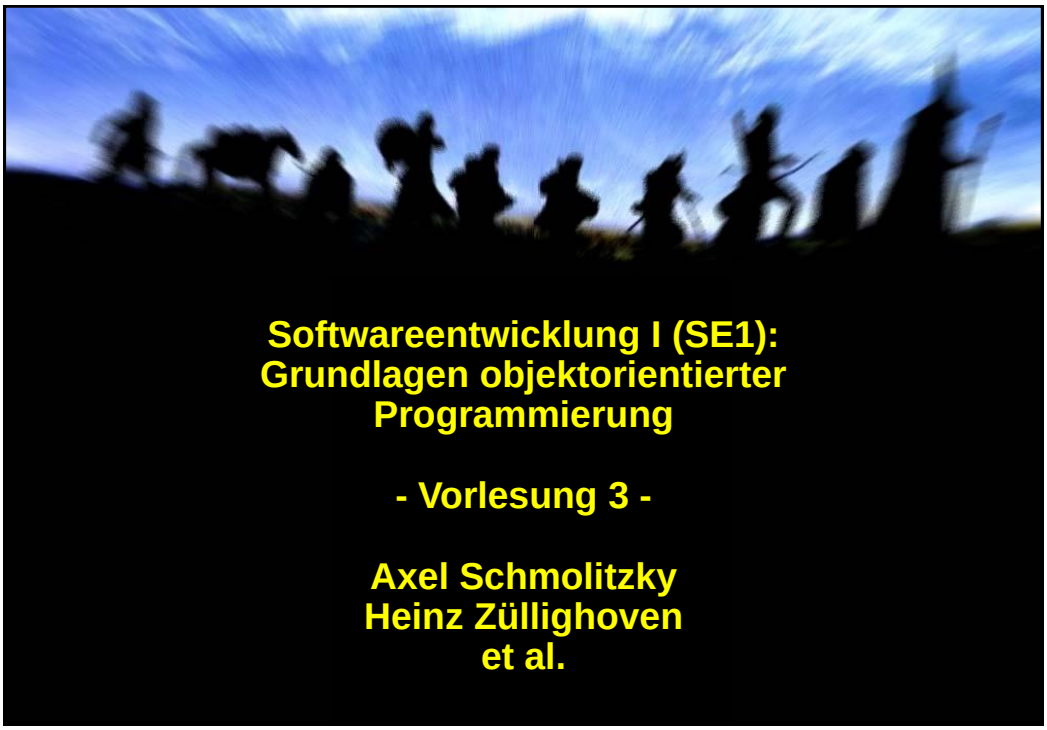


Softwareentwicklung I (SE1):
Grundlagen objektorientierter Programmierung

- Vorlesung 3 -

Axel Schmolitzky
Heinz Züllighoven
et al.

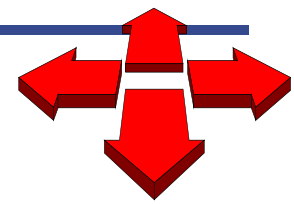
Begriffe:
Richtig oder falsch?

> ***„Der Unterschied zwischen dem richtigen Wort und dem beinahe richtigen ist der gleiche wie zwischen einem Blitz und einem Glühwürmchen.“***
>
> **Mark Twain**

?

Viel wichtiger:
konsistent!

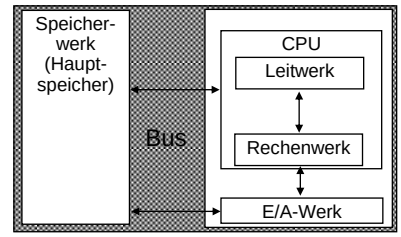
Imperative Grundkonzepte



- Grundlage: von Neumann-Rechner
- Der Prozedurbegriff
  - Parameter
  - Ergebnisse
  - Kontrollfluss

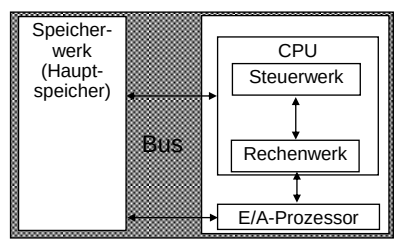
Konzept (fast) aller Computer: der von Neumann-Rechner

- Der Rechner besteht aus 4 Werken.
- Die Rechnerstruktur ist unabhängig vom bearbeiteten Problem.
- Programme und Daten stehen im selben Speicher.
- Der Hauptspeicher ist in Zellen gleicher Größe unterteilt, die durchgehend adressierbar sind.
- Das Programm besteht aus Folgen von Befehlen, die generell nacheinander ausgeführt werden.
- Von der sequenziellen Abfolge kann durch Sprungbefehle abgewichen werden.
- Die Maschine benutzt Binärcodes für die Darstellung von Programm und Daten.



Imperative Programme auf von Neumann-Maschinen

- Die **elementaren Operationen** eines **von Neumann-Rechners**:
  - Die CPU führt **Maschinenbefehle** aus.
  - Dabei werden über den sog. **Bus Befehle und Daten** vom Speicher in die CPU übertragen und die **Ergebnisse** zurück übertragen.
- **Imperative Programmiersprachen** abstrahieren von diesen elementaren Operationen:
  - **Anweisungen** (engl.: statements) fassen Folgen von Maschinenbefehlen zusammen,
  - **Variablen** (engl.: variables) abstrahieren vom physischen Speicherplatz.



Ablaufsteuerung im Vergleich

- Ablaufsteuerung in der **von Neumann-Maschine**:
  - Aufeinanderfolgende Befehle stehen hintereinander im Speicher, werden vom Steuerwerk nach einander in den zentralen Prozessor geholt und dort geeignet decodiert und verarbeitet.
  - Durch Sprungbefehle kann von der Reihenfolge der gespeicherten Befehle abgewichen werden.
- Ablaufsteuerung auf **Ebene der Programmiersprache** (**Kontrollstrukturen**):
  - Innerhalb einer Methode:
    - **Sequenz**
    - **Fallunterscheidung**
    - **Wiederholung**
  - Zusätzlich sind **Methodenaufrufe** spezielle Anweisungen, die ebenfalls in den sequenziellen Ablauf eingreifen.

SE1 – Level 1 7

Hintergrund: der Prozedurbegriff

- Die **Methoden** in der objektorientierten Programmierung sind spezielle Ausprägungen des klassischen imperativen **Prozedurbegriffs**.
- In der imperativen Programmierung sind Prozeduren das **mächtigste Abstraktionsmittel**.
- Viele Prinzipien von Prozeduren gelten analog für die Methoden objektorientierter Sprachen wie Java.
- Im Folgenden wird der Prozedurbegriff ausführlicher erläutert.

• **Pro|ze|dur** [lat.-nlat.] *die;* -, –en: Verfahren, [schwierige, unangenehme] Behandlungsweise.

SE1 – Level 1 8

Methoden/Prozeduren als Grundeinheiten eines imperativen Programms

- Methodenaufrufe (in objektorientierten Sprachen) oder Prozeduraufrufe (in klassischen imperativen Sprachen) bestimmen die Aktivitäten eines Programms.
- Hinter Methoden und Prozeduren steht das gleiche Konzept:
  - **Fachlich** realisieren sie einen **Algorithmus** mit den Mitteln einer Programmiersprache.
  - **Softwaretechnisch** sind sie eine **benannte Folge von Anweisungen**.
  - Die Grundidee ist, den **Namen** der Methode oder Prozedur **„stellvertretend" für diese Anweisungsfolge** anzusehen.
  - Einer Methode/Prozedur können beim Aufruf unterschiedliche Informationen mitgegeben werden. Dies geschieht durch Konzepte der **Parameterübergabe**.

In imperativen Sprachen sind Prozeduren „frei", d.h. sie sind nicht als Methoden einer Klasse und ihren Objekten zugeordnet.

SE1 – Level 1 9

Ein erster Algorithmus-Begriff

- Ein **Algorithmus** bildet die aus problembezogener Sicht kleinste Einheit beim Programmentwurf.
- Wir verstehen unter einem Algorithmus einen **endlichen Text**, in dem ein für einen Prozessor (Interpreter) eindeutiges allgemeines und schrittweises **Problemlösungsverfahren** aus **Aktionen**, die auf **sprachlichen Einheiten** arbeiten, beschrieben ist.
- Ein Algorithmus **terminiert**, wenn er nicht nur in einer endlichen Vorschrift beschrieben ist, sondern auch nach endlich vielen Schritten seine Bearbeitung beendet.
- Wir können einem Algorithmus einen **Namen** geben. Seine Verfahrensschritte können sich wieder auf weitere Algorithmen beziehen, die an anderer Stelle beschrieben sind.
- In der imperativen Programmierung setzen wir Algorithmen in **Prozeduren** um.

SE1 – Level 1 10

Die Grundidee einer Methode / Prozedur

Eine **Anweisungsfolge**:

```
{
   int max;
   if (a > b)
   {
      max = a;
   }
   else
   {
      max = b;
   }
}
```

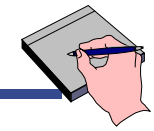
… erhält einen **Namen** und **Parameter**:

```
int maximum(int a, int b)
{
   int max;
   if (a > b)
   {
      max = a;
   }
   else
   {
      max = b;
   }
   return max;
}
```

… und kann **aufgerufen** werden:

```
…
int ergebnis = maximum(6, 9);
…
```


Parametrisierung

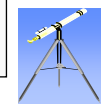


- Damit die Anweisungsfolgen von Prozeduren nicht nur für einen bestimmten Fall zutreffen, wird ein zweiter Abstraktionsmechanismus eingesetzt - **Datenaustausch durch Parameter**.
- In maschinennahen Sprachen werden als Parameter Speicheradressen von Speicherzellen übergeben, in denen die Eingabe- oder Ausgabedaten stehen.
- Höhere imperative Programmiersprachen verwenden das Konzept **getypter formaler Parameter**.

Formen der Parameterübergabe

- Parameter von Prozeduren lassen sich einteilen nach der Art, wie sie den **Informationsaustausch** zwischen der aufrufenden Programmstelle und der Prozedur regeln. Zwei zentrale Mechanismen zur Übergabe von Parametern in der imperativen Programmierung sind:
  - Über **Eingabe-** oder **Wert-Parameter** (engl.: call by value):
    Der Parameter dient zur Übergabe von Informationen an die Prozedur.
    Der Aufrufer gibt die **aktuellen Parameter** in Form von **Ausdrücken** an.
    Die Werte der Ausdrücke stehen der gerufenen Prozedur unter dem formalen Parameternamen zur Verfügung.
  - Über **Ausgabe-** oder **Variablen-Parameter** (engl.: call by reference):
    Der Parameter kann entweder zusätzlich oder ausschließlich ein **Ergebnis** an den Aufrufer **liefern**. An der Aufrufstelle muss eine **Variable** stehen, die das Ergebnis aufnehmen kann.

Java kennt ausschließlich Eingabe-Parameter! Wir werden uns Ausgabe-Parameter deshalb frühestens in SE2 näher ansehen.



Formale und aktuelle Eingabe-Parameter

Formale Parameter

Definierende Stelle:

```
int maximum(int a, int b)
{
  int max;
  if (a > b)
  {
      max = a;
  }
  else
  {
      max = b;
  }
  return max;
}
```

- An der definierenden Stelle hat eine Methode/Prozedur sog. **formale Parameter** zur Datenübergabe.
- Im Beispiel `a` und `b`; beide sind vom Typ `int`.

Aktuelle Parameter

Aufrufende Stellen:

```
...
int ergebnis = maximum(6, 9);
int ergebnis2 = maximum(ergebnis, 2*x);
```

- An der aufrufenden Stelle erhält eine Methode/Prozedur **aktuelle Parameter**.
- Bei **Eingabe-Parametern** sind dies **Ausdrücke**, hier jeweils vom **passenden Typ** `int`.
- Die Werte der Ausdrücke werden den formalen Parametern **zugewiesen**.

Regeln bei der Parameterübergabe

- Beim Prozeduraufruf werden die Werte der aktuellen Parameter an die formalen übergeben. Zur Übersetzungszeit wird überprüft:
  - Der **Name** im Aufruf definiert die zu rufende Prozedur.
  - Die **Anzahl** der aktuellen Parameter muss gleich der Anzahl der formalen sein.
  - Die Bindung der jeweiligen Parameter wird entsprechend ihrer **Position** im Aufruf und in der Prozedurdeklaration vorgenommen.
  - Die aktuellen Parameter müssen **typkompatibel** zu den formalen Parametern sein (d.h. zunächst typgleich).

Diese Regeln werden üblicherweise von einem Compiler überprüft!

Ergebnisprozedur

- Es gibt auch Prozeduren, die die programmiersprachliche Form einer Funktion haben, die wir **Ergebnisprozeduren** nennen.
- Sie liefern ein Ergebnis, das an der aufrufenden Stelle direkt in einem Ausdruck verwendet werden kann.
- In Java sind Methoden mit einem Ergebnistyp ungleich `void` für uns (vorläufig) die einzige Möglichkeit, Informationen von der gerufenen Prozedur an den Aufrufer zurück zu liefern.

Formales und aktuelles Ergebnis in Java

Formales Ergebnis

Definierende Stelle:

```
int maximum(int a, int b)
{
    int max;
    if (a > b)
    {
        max = a;
    }
    else
      ...
    return max;
}
```

- An der definierenden Stelle kann eine Methode ein **formales Ergebnis** definieren.
- Steht dort hingegen `void`, ist die Methode keine Ergebnisprozedur.
- Eine Ergebnisprozedur **muss** mit `return` ein Ergebnis liefern.

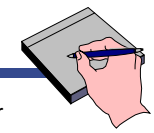
- An der aufrufenden Stelle kann der Name der Ergebnisprozedur **stellvertretend** für das Ergebnis des Aufrufs angesehen werden.

Aktuelle Ergebnisse

Aufrufenden Stellen:

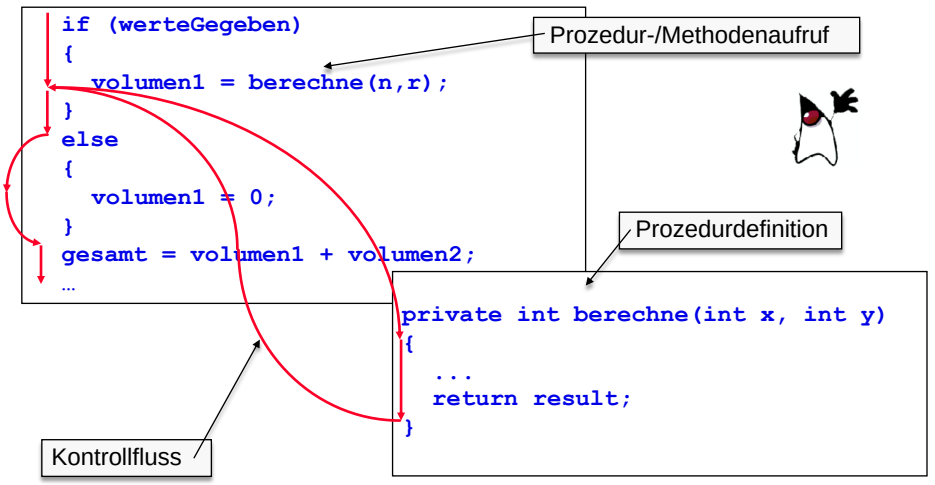
```
...
int ergebnis = maximum(6, 9);
int ergebnis2 = maximum(ergebnis, 2*x);
```


Kontrollfluss bei Prozeduraufrufen



- Der **Prozeduraufruf** ist die explizite Anweisung, dass eine Prozedur ausgeführt werden soll.
- Eine Prozedur ist **aktiv**, nachdem sie gerufen wurde und in der Abarbeitung ihrer Anweisungen noch kein vordefiniertes Ende erreicht hat.
- Für den Prozeduraufruf in **sequenziellen imperativen** Sprachen ist charakteristisch:
  - Beim Aufruf wechselt die **Kontrolle** (d.h. die Abarbeitung von Anweisungen) vom Rufer zur Prozedur.
  - Dabei werden die (Werte der) **aktuellen Parameter** an die **formalen gebunden** (ihnen zugewiesen).
  - Prozeduren können **geschachtelt** aufgerufen werden. Dabei wird der Rufer unterbrochen, so dass die Kontrolle **immer nur bei einer Prozedur** ist; es entsteht eine **Aufrufkette**.
  - Nach der **Abarbeitung** der Prozedur kehrt die Kontrolle zum Rufer zurück; die Abarbeitung wird mit der Anweisung nach dem Aufruf fortgesetzt.

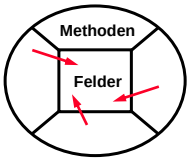
Kontrollfluss und Prozedur-Mechanismus



SE1 – Level 1 19

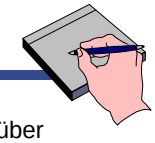
Unterschiede zwischen Prozeduren und Methoden

- Die **Methoden** der objektorientierten Programmierung haben die gleichen Eigenschaften wie die **Prozeduren** der imperativen Programmierung:
  - Beim Aufruf einer Methode **wechselt der Kontrollfluss** in die gerufene Methode, um nach dem Aufruf hinter die Aufrufstelle zurückzukehren.
  - Beim Aufruf besteht die Möglichkeit, **Parameter** zu übergeben.
- Darüber hinaus haben Methoden jedoch **weitere Eigenschaften**, die sie von simplen Prozeduren unterscheiden:
  - Eine Methode **gehört immer zu einem Objekt** (das beim Aufruf einer Methode angegeben werden muss);
  - Eine Methode kann immer auf die **Felder des zugehörigen Objektes** zugreifen.
  - Methoden haben eine **Sichtbarkeit** (in Java: `private` oder `public`).



SE1 – Level 1 20

Drei Arten von Variablen



- Eine imperative Variable hat einen **Namen** (häufig auch: **Bezeichner**), über den sie angesprochen werden kann, und einen **Typ**. Während der Ausführung eines Programms hat sie eine **Belegung** bzw. einen **Wert**.
- Wir kennen inzwischen drei Arten von Variablen:
  - **Exemplarvariablen** (Felder), die den Zustand von Objekten halten.
  - **Formale Parameter**, mit denen Methoden parametrisiert werden können.
  - **Lokale Variablen**, die als Hilfsvariablen in den Rümpfen von Methoden vorkommen können.

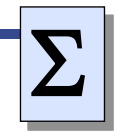
Exemplarvariable

Formaler Parameter

Lokale Variable

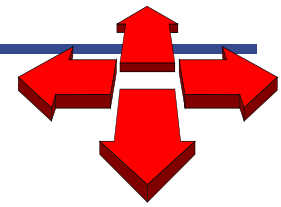
```
class Girokonto
{
  private int _saldo;

  public void einzahlen( int betrag )
  {
    int neuerSaldo;
    neuerSaldo = _saldo + betrag;
    _saldo = neuerSaldo;
  }
}
```


Zwischenergebnis Prozedur/Methode



- Aufrufe von Methoden entsprechen weitgehend **imperativen Prozeduraufrufen**.
- Prozeduren können **parametrisiert** werden; der Aufrufer übergibt **aktuelle Parameter**, die gerufene Methode bekommt diese als **formale Parameter**.
- **Java kennt nur Wert-Parameter**: Die Werte der aktuellen Parameter werden beim Aufruf kopiert; **die gerufene Methode arbeitet nur auf Kopien**, die den formalen Parametern zugewiesen wurden.
- **Ergebnisprozeduren** liefern ein Ergebnis, das an der Aufrufstelle direkt in einem Ausdruck verwendet werden kann.
- Der **Kontrollfluss** wechselt von der aufrufenden Prozedur zur aufgerufenen Prozedur; nach dem Ende der Ausführung kehrt die Kontrolle zum Aufrufer zurück.
- **Methoden** sind ein „reicheres“ Konzept als Prozeduren: eine Methode ist immer **einem Objekt zugeordnet** und hat **Zugriff auf die Felder** dieses Objekts.

Die Struktur von Klassendefinitionen

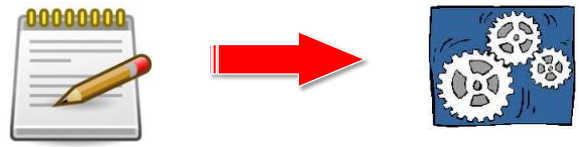


- **Übersetzung**
  - Hybride Sprachen
  - Virtuelle Maschinen
- **Syntaktischer Aufbau**
  - Formale Sprachen
  - Syntax in EBNF

SE1 – Level 1 23

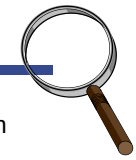
Von der Klassendefinition zur Ausführung

- Eine **Klassendefinition** ist lediglich die **textuelle Beschreibung** einer Klasse. Sie liegt in einer Textdatei vor und kann mit einem Editor bearbeitet werden.
- Wenn wir eine Klasse benutzen wollen (indem wir Exemplare von ihr erzeugen und diese Exemplare benutzen), müssen wir zuerst dafür sorgen, dass die menschenlesbare Klassendefinition in eine Form überführt wird, die ein Computer ausführen kann. Diesen Vorgang nennen wir **Übersetzen** bzw. **Compilieren**.
- Nur durch eine korrekt übersetzte Klassendefinition entsteht eine Klasse, die bei der Ausführung eines Java-Programms zur Erzeugung von Exemplaren benutzt werden kann.



SE1 – Level 1 24

Verarbeitung von Programmen im Rechner



Die Programme höherer Programmiersprachen werden nach drei Ansätzen verarbeitet:

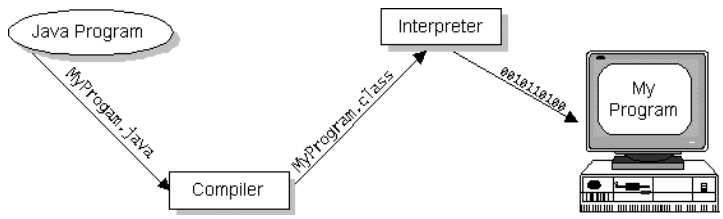
- **Compiler-Sprachen** (Bsp.: C++, Modula-2)
  - Alle Anweisungen eines Programms werden einmalig in eine Maschinensprache übersetzt und dann direkt in dieser Sprache ausgeführt.
- **Interpretersprachen** (Bsp.: Lisp, Skriptsprachen wie PHP, Perl etc.)
  - Eine einzelne Anweisung eines Programms wird von einem anderen Programm (dem **Interpreter**) immer erst dann interpretiert (übersetzt), wenn sie ausgeführt werden soll.
- **Hybride Sprachen** (Bsp.: Java, C#)
  - Programme werden in eine Zwischensprache übersetzt, die sich gut für die Interpretation eignet.

Hybride Verarbeitung in Java

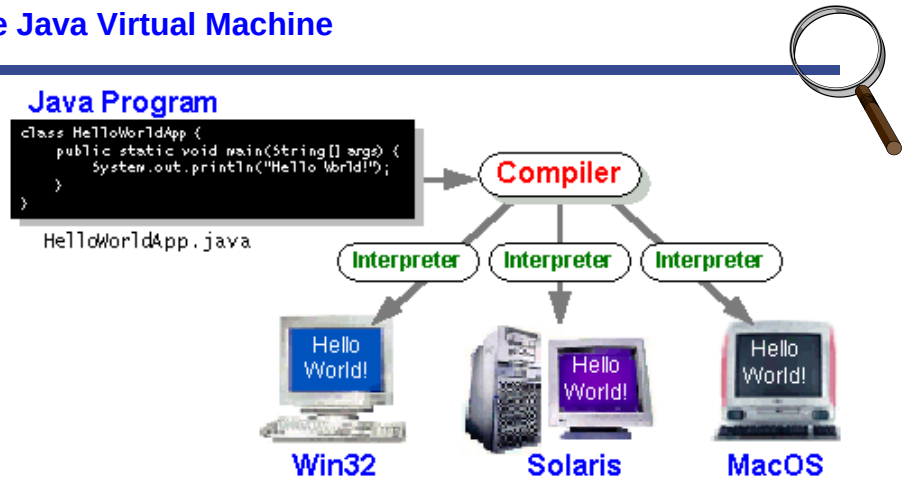
Die hybride Verarbeitung bei Java ist eine Kombination:

- Der **Quelltext** wird von einem Compiler in **Zwischencode** transformiert, der **Java Bytecode** genannt wird.
- Dieser Zwischencode wird dem Interpreter übergeben, der sog. **Java Virtual Machine**.

... each Java program is both compiled and interpreted. With a **compiler**, you translate a Java program into an intermediate language called **Java bytecode** -- the platform-independent code interpreted by the Java interpreter. With an **interpreter**, each Java bytecode instruction is parsed and run on the computer.
Compilation happens just once; interpretation occurs each time the program is executed.



Die Java Virtual Machine

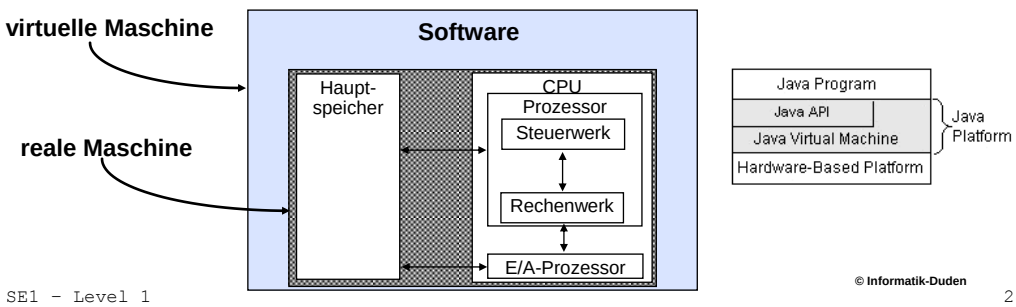


... You can think of **Java bytecodes** as the **machine code instructions** for the **Java Virtual Machine (Java VM)**.
... Java bytecodes help make **"write once, run anywhere"** possible.

SE1 – Level 1 27

Virtuelle Maschinen: alles nur Software

- Ein **laufendes Programm** lässt sich verstehen als eine Folge von Befehlen für eine **Maschine**, die diese Befehle schrittweise ausführen kann.
- Diese Maschine muss nicht mechanisch oder elektronisch konstruiert werden; es genügt, ihren Satz an **Maschinenbefehlen**, ihre **Speicherbereiche** und ihre **Kontrollstrukturen** festzulegen.
- Eine **virtuelle Maschine** ist eine solche, durch Software definierte Maschine, die selbst auf einem Computer implementiert ist.

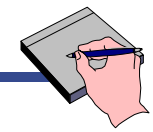


SE1 – Level 1 28

Syntaktische Struktur von Klassendefinitionen

- Die Struktur der Klassendefinitionen von Java folgt bestimmten Regeln, die wir als die **Syntax** von Java bezeichnen.
- Für die **Syntax von Programmiersprachen** gibt es spezielle **formale Beschreibungen**, die zwei Zwecken dienen:
  - Menschen können sie lesen, um syntaktisch korrekte Programme schreiben zu können.
  - Computer können sie ebenfalls lesen, um korrekte Programme verarbeiten zu können (und inkorrekte ablehnen zu können).
- Wir betrachten zwei gebräuchliche Darstellungsformen der Syntax von Programmiersprachen:
  - **Erweiterte Backus-Naur-Form (EBNF)**
  - **Syntaxdiagramme**

Syntax, Semantik und Pragmatik



- **Syntax** (nach Informatik-Duden):
  - Eine Sprache wird durch eine Folge von Zeichen, die nach bestimmten Regeln aneinandergereiht werden dürfen, definiert. Den hierdurch beschriebenen formalen Aufbau der Sätze und Wörter, die zur Sprache gehören, bezeichnet man als ihre Syntax.
  - Programme sind nach den festen **Syntaxregeln** ihrer jeweiligen Programmiersprache aufgebaut.
- **Semantik:**
  - Lehre von der **inhaltlichen Bedeutung** einer Sprache.
  - **Die Semantik eines Programms ist das, was das Programm beim Ablauf im Rechner (und darüber hinaus) bewirkt.**
  - Programmiersprachen definieren neben Syntax- auch **Semantikregeln**, beispielsweise, dass nur deklarierte Variablen verwendet werden dürfen.
- **Pragmatik**:
  - Lehre vom Gebrauch einer Sprache in einem bestimmten Zusammenhang.
  - Die Pragmatik eines Programms wird durch seinen Zweck, die Aufgabenstellung und die jeweilige Verwendung bestimmt.

Syntax, Semantik, Pragmatik am Beispiel

- Die **Syntax** einer Sprache wird üblicherweise in Grammatikregeln beschrieben. Eine Regel für den Aufbau von deutschen Sätzen könnte beispielsweise so aussehen:
  - <*Deutscher Satz*> ::= <Subjekt> <Prädikat> <Objekt> '.'
  - Ein Beispiel für einen syntaktisch korrekten Satz ist dann „Der Mann beißt den Hund.“.
- Die **Semantik** des Satzes (seine inhaltliche Aussage) ist uns klar, wenn wir den Satz verstehen. Es gibt aber syntaktisch korrekte Sätze, die wir nicht leicht verstehen, wie „Das Haus streichelt den Mann.“
- Der Unterschied zwischen **Semantik** und **Pragmatik** wird an folgendem Satz klar: „Da ist die Tür.“
  - Die **Semantik** ist der Hinweis, wo sich im Raum eine Tür befindet.
  - Die **Pragmatik** kann je nach Situation sein:
    - Eine höfliche Antwort auf die Frage einer orientierungslosen Person.
    - Eine sehr deutliche Aufforderung, den Raum zu verlassen.

Alle Arten von **Quelltextkonventionen** sind Hinweise zur Pragmatik.

Benutzerhandbücher geben oft Hinweise zur Pragmatik.

Syntax, Semantik, Pragmatik in Java

```
class Girokonto
{
   private int _saldo;

   public void einzahlen(int betrag)
   {
      _saldo = _saldo + betrag;
   }
}
```

class <*Klassenname*>
{
<*optional: Felder*>
<*optional: Konstruktoren*>
<*optional: Methoden*>
}

- Die **Syntax** für Klassen in Java ist hier (übervereinfacht!) dargestellt.

- Die **Semantikregeln** von Java definieren für die Klasse `Girokonto` beispielsweise, dass sie einen Standardkonstruktor hat.

- Die Reihenfolge ist ein Beispiel für die **Pragmatik** von Java („So machen wir es.“); die (echte) Syntax lässt es anders zu und auch die Semantik ist durch die Reihenfolge dieser logischen Bestandteile nicht beeinflusst.

Syntaxbeschreibungen von Programmiersprachen

- Um ein Programm korrekt notieren zu können, müssen wir zunächst die **Syntax** der entsprechenden Programmiersprache verstanden haben.
- Ähnlich wie auch bei natürlichen Sprachen werden Grammatikregeln verwendet, um die Syntax zu definieren. Die **formalen Grammatiken**, die für Programmiersprachen verwendet werden, bestehen aus **Regeln**, mit denen **Wörter** aus einem **Startsymbol** gebildet werden können.
- Den theoretischen Hintergrund dazu (die **kontextfreien Grammatiken** aus der Chomsky-Hierarchie) sehen wir uns hier nicht näher an. Er wird in den FGI-Modulen behandelt.
- Wir sehen uns als pragmatische Darstellung zunächst die **Backus-Naur-Form** an, da sie sehr hilfreich ist, um die Beschreibungen von Programmiersprachen verstehen zu können.

Historie der Backus-Naur-Form

- Mitte der 50-er Jahre entwickelte eine internationale Expertengruppe (die ACM-GAMM Gruppe) die Programmiersprache **ALGOL 58**. Eine formale Beschreibung der Syntax der Sprache wurde von **John Backus** (1959) vorgelegt.
- Diese Notation wurde von **Peter Naur** modifiziert und zur Beschreibung von **ALGOL 60** verwendet.
- Diese Version der Notation ist als **Backus-Naur-Form**, kurz **BNF**, bekannt und die wohl häufigste Form der Syntaxbeschreibung von Programmiersprachen.

Sebesta:

BNF gleicht einer Beschreibung der Syntax von Sanskrit durch Panini mehrere Jahrhunderte v.Chr.

© Sebesta

Grundkonzepte der BNF: Nichtterminale und Terminale

- Die BNF definiert syntaktische Strukturen mit Hilfe zweier Elementarten:
  - Syntaktische Variablen, häufig **Nichtterminale** genannt;
  - Syntaktische Basiselemente, auch terminale Symbole oder kurz **Terminale** genannt.
- **Nichtterminale** werden oft in spitzen Klammern notiert. Beispiel:
  `<Zuweisung>`
- Die **Definition** eines Nichtterminals heißt **Regel** oder **Produktion**. Beispiel:
  `<Zuweisung>` → `<Bezeichner> '=' <Ausdruck>`
- Eine solche Regel besagt, dass das Nichtterminal auf der linken Seite an allen Stellen, an denen es auftritt, durch die Verkettung der Elemente auf der rechten Seite ersetzt werden kann.
- Auf der linken Seite des **Ableitungssymbols** (→) steht immer genau ein Nichtterminal, auf der rechten Seite können beliebig viele Elemente stehen.
- Zu jedem Nichtterminal muss es mindestens eine Regel geben, in der das Nichtterminal auf der linken Seite steht.

Grundkonzepte der BNF: Nichtterminale und Terminale (II)

- Die **Terminale** einer Programmiersprachgrammatik (häufig auch **Lexeme** oder **Token** genannt) sind üblicherweise die
  - **reservierten Wörter**, (wie `if`, `begin`, `end` etc.)
  - **Bezeichner** (wie Namen von Variablen),
  - **Literale** (wie Zahlen und Zeichenketten)
  - und die **Sonderzeichen** (Operatoren, Satzzeichen, Klammern)

  einer Sprache.
- Terminale werden oft in einfachen Anführungsstrichen notiert:
  `<Zuweisung>` → `<Bezeichner> '=' <Ausdruck>`
- Terminale sind aus Sicht der Grammatik **unteilbare Elemente**, d.h. für sie existieren keine Regeln innerhalb der Grammatik; trotzdem können auch sie nach komplizierten Regeln aufgebaut sein, siehe etwa die Gleitkommazahlen.

BNF: Startsymbol, Konkatenation und Auswahl

- Ein Nichtterminal dient immer als **Startsymbol**; alle Sätze (korrekter ist eigentlich: Wörter) der definierten Sprache werden aus diesem Startsymbol abgeleitet.
- Die Elemente auf der rechten Seite einer Ableitungsregel werden konkateniert (verkettet):

```
<Zuweisung> → <Bezeichner> '=' <Ausdruck>
```

- Syntaktische Variablen können mehr als eine Definition haben:

```
<Zahl> → 'Integer' | 'Real'
```

  `Integer` und `Real` sind hier Terminale, d.h. für sie existieren in der Grammatik keine Regeln, die ihre Struktur definieren.

  "|" ist hier das Zeichen für die Wahl zwischen alternativen Regeln.

- Nichtterminale können rekursiv definiert sein:

```
<Anweisungsfolge> → <Anweisung>
                  | <Anweisung> ';' <Anweisungsfolge>
```


Ableiten von Wörtern einer Sprache

- Geg. die folgende einfache **Syntaxbeschreibung**:

```
<Zuweisung>  → <Bezeichner> ':=' <Ausdruck> .
<Bezeichner> → 'A' | 'B' | 'C'.
<Ausdruck>   → <Bezeichner> '+' <Ausdruck>
             | <Bezeichner> '*' <Ausdruck>
             | '(' <Ausdruck> ')'
             | <Bezeichner> .
```

- Durch **schrittweises Ersetzen der Nichtterminale** durch die entsprechenden Definitionen und Konkatenation der Terminale lassen sich alle Wörter der Sprache generieren. Beispiel für die **Ableitung** eines Wortes:

```
<Zuweisung> ⇒ <Bezeichner> ':=' <Ausdruck>
            ⇒ 'A :=' <Ausdruck>
            ⇒ 'A :=' <Bezeichner> '*' <Ausdruck>
            ⇒ 'A := B *' <Ausdruck>
            ⇒ 'A := B * (' <Ausdruck> ')'
            ⇒ 'A := B * (' <Bezeichner> '+' <Ausdruck> ')'
            ⇒ 'A := B * ( A +' <Ausdruck> ')'
            ⇒ 'A := B * ( A +' <Bezeichner> ')'
            ⇒ 'A := B * ( A + C )'
```


EBNF: Option und Wiederholung

- Es ist für die Darstellung praktisch, die BNF um **optionale** und **wiederholte Elemente** zu erweitern. Man spricht dann von der **erweiterten (extended) BNF**, kurz **EBNF**. Dies ist die heute gebräuchliche Form. (Im folgenden sind die spitzen Klammern für Nichtterminale ausgelassen.)
- **Wiederholbare Elemente** (einschließlich null-mal) schreibt man in **geschweiften** Klammern. Dadurch wird aus:

```
Anweisungsfolge → Anweisung
                | Anweisung ';' Anweisungsfolge
```

mit Hilfe der geschweiften Klammern:

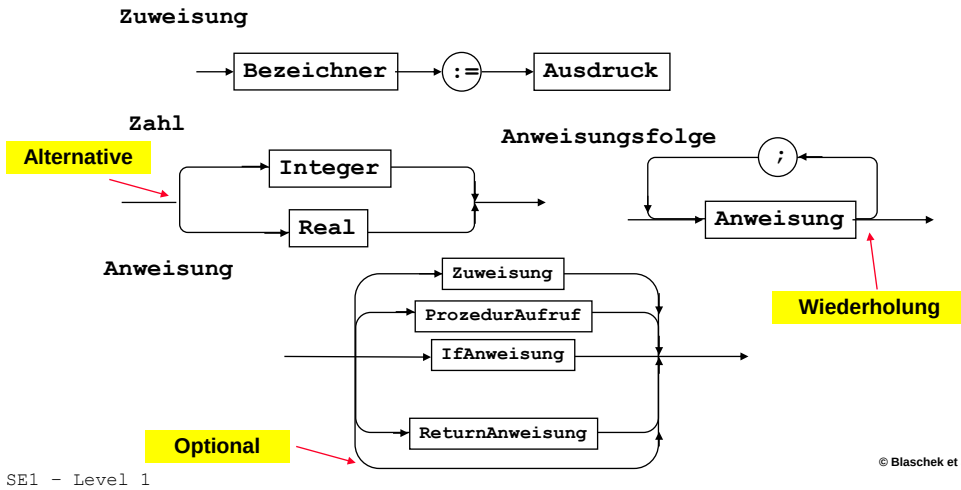
```
Anweisungsfolge → Anweisung { ';' Anweisung }
```

- **Optionale Elemente** schreibt man in **eckigen** Klammern. Beispiel:

```
Anweisung → [ Zuweisung | ProzedurAufruf
            | IfAnweisung | CaseAnweisung
            | WhileAnweisung | RepeatAnweisung
            | LoopAnweisung | WithAnweisung
            | ExitAnweisung | ReturnAnweisung ]
```


Syntaxdiagramme

- Häufig werden zur Darstellung von EBNF-Grammatiken auch **Syntaxdiagramme** verwendet.
- Terminale Symbole werden in diesen Diagrammen in Kreisen, Nichtterminale in Rechtecken dargestellt.



Java Level 1: Syntax für Einsteiger

- Die Syntaxbeschreibung von Java ist sehr umfangreich; viele Ausnahmen und fortgeschrittene Konzepte erschweren das Einlesen für Programmieranfänger.
- Für SE1 haben wir deshalb eine Untermenge von Java namens **Java Level 1** definiert, die nur wenige Sprachkonzepte enthält, u.a. Klassen, Felder, Methoden, Variablen, Anweisungen und Ausdrücke sowie einige Basistypen.
- Die Syntax von Java Level 1 in EBNF passt auf eine A4-Seite.
- Diese Minisprache ist dennoch sehr mächtig: Mit ihr können die ersten fünf Aufgabenblätter (fast vollständig) bearbeitet werden.

- Die Wahl der Namen für die Nichtterminale der Syntax hält sich eng an die Sprachdefinition von Java, die auch im Internet verfügbar ist:

  http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html
- Die Grammatik ist leicht erweiterbar um weitere Konzepte, die im Lauf der Vorlesung vorgestellt werden, wie Schleifen, Interfaces etc.

SE1 – Level 1 41

Ausschnitt aus der Syntax von Java Level 1

Startsymbol

Doppelpunkt statt Ableitungssymbol

```
CompilationUnit:
        NormalClassDeclaration
NormalClassDeclaration:
        class Identifier ClassBody
ClassBody:
        { { ClassBodyDeclaration } }
 ...
```

Terminale in **Blau**, EBNF-Symbole in **Rot**

SE1 – Level 1 42

Ausschnitt aus der Syntax von Java Level 1 (II)

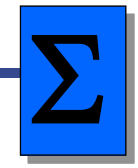
Nichtterminal für eine **Anweisung** (engl.: statement) als linke Seite einer Regel

Rechte Seite: **6 Alternativen**, jede in einer eigenen Zeile

```
...
Statement:
    if ParExpression Statement [ else Statement ]
    return [ Expression ] ;
    ;
    Assignment ;
    MethodInvocation ;
    Block

Assignment:
    Identifier = Expression

ParExpression:
    ( Expression )
```

SE1 – Level 1 43

Zusammenfassung



- Die syntaktische Struktur von Klassendefinitionen wird häufig in der **Erweiterten Backus-Naur-Form (EBNF)** beschrieben.
- Die Syntax von Java ist in einer Abwandlung der EBNF notiert, die wir auch für die Syntaxdefinition von Java Level 1 verwenden.

SE1 – Level 1 44