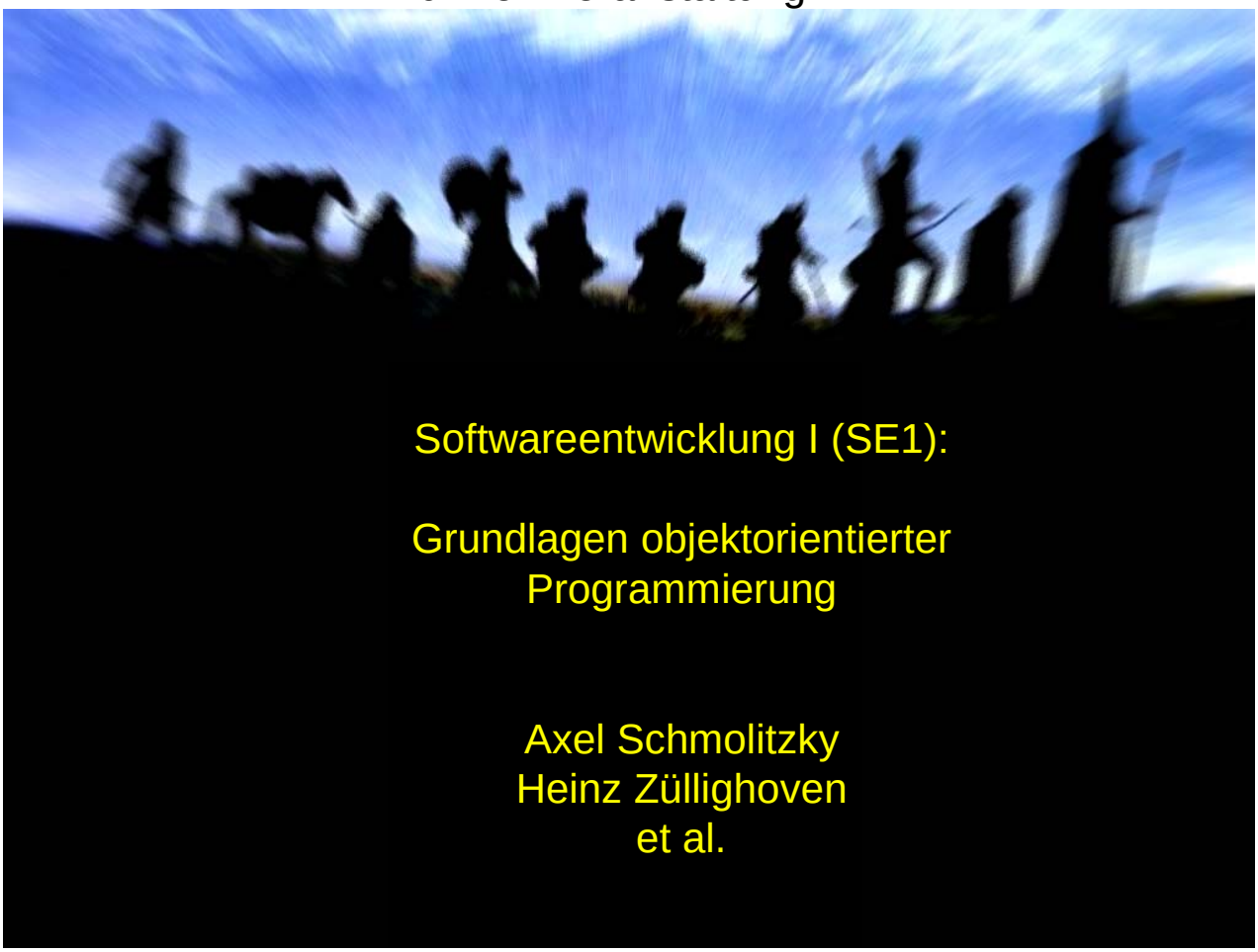
Prüfungsunterlagen
zur Lehrveranstaltung



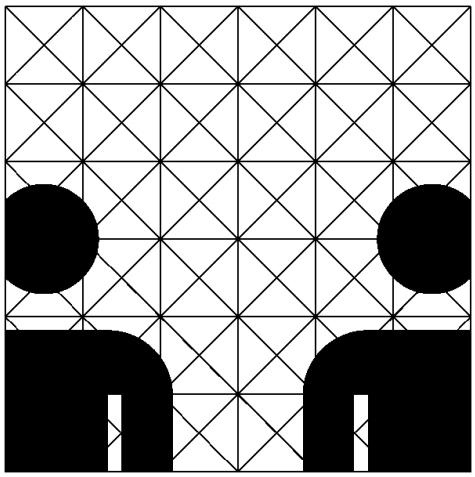
Softwareentwicklung I (SE1):

Grundlagen objektorientierter Programmierung

Axel Schmolitzky
Heinz Züllighoven
et al.

Teil 1

Universität Hamburg
MIN-Fakultät
Department Informatik
WS 2011 / 2012



Softwareentwicklung I
SE1
Grundlagen objektorientierter Programmierung

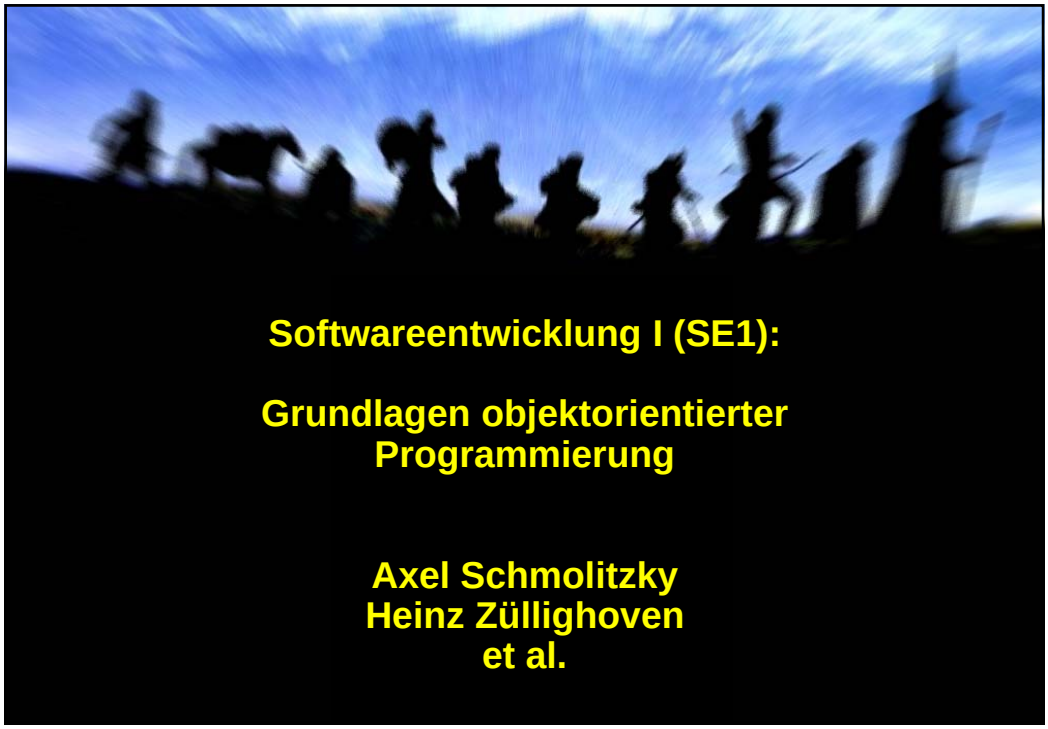
Axel Schmolitzky
Heinz Züllighoven
et al.

Teil 1

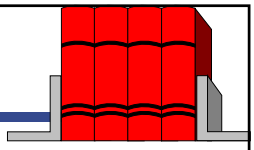
Verzeichnis der Folien

39. Die Struktur von Klassendefinitionen
40. Unsere erste selbst geschriebene Klassendefinition
41. Merkmale unserer ersten Klasse
42. Abgleich mit den Prinzipien der Objektorientierung
43. Auswertung: Grobstruktur einer Klassendefinition
44. Auswertung: allgemeine Struktur einer Klassendefinition
45. Objekte erzeugen
46. Konstruktoraufruf und Konstruktor
47. Methoden aufrufen
48. Die Punktnotation der Objektorientierung
49. Die Punktnotation in Java
50. Struktur der Methodendefinition in Java
51. Verändernde Methoden
52. Sondierende Methoden
53. Von der Klassendefinition zur Ausführung
54. Verarbeitung von Programmen im Rechner
55. Hybride Verarbeitung in Java
56. Die Java Virtual Machine
57. Virtuelle Maschinen: alles nur Software
58. Syntaktische Struktur von Klassendefinitionen
59. Syntax, Semantik und Pragmatik
60. Syntax, Semantik, Pragmatik am Beispiel
61. Syntax, Semantik, Pragmatik in Java
62. Syntaxbeschreibungen von Programmiersprachen
63. Historie der Backus-Naur-Form
64. Grundkonzepte der BNF: Nichtterminale und Terminale
65. Grundkonzepte der BNF: Nichtterminale und Terminale (II)
66. BNF: Startsymbol, Konkatenation und Auswahl
67. Ableiten von Wörtern einer Sprache
68. EBNF: Option und Wiederholung
69. Syntaxdiagramme
70. Java Level 1: Syntax für Einsteiger
71. Ausschnitt aus der Syntax von Java Level 1
72. Ausschnitt aus der Syntax von Java Level 1 (II)
73. Beispiel einer Ableitung
74. Beispiel einer Ableitung (II): i = j + 3;
75. Beispiel einer Ableitung (III): i = j + 3;
76. Beispiel einer Ableitung: Zusammenfassung
77. Zusammenfassung

78. Imperative Grundkonzepte
79. Konzept (fast) aller Computer: der von Neumann-Rechner
80. Imperative Programme auf von Neumann-Maschinen
81. Ablaufsteuerung im Vergleich
82. Hintergrund: der Prozedurbegriff
83. Methoden/Prozeduren als Grundeinheiten eines imperativen Programms
84. Ein erster Algorithmus-Begriff
85. Die Grundidee einer Methode / Prozedur
86. Parametrisierung
87. Formen der Parameterübergabe
88. Formale und aktuelle Eingabe-Parameter
89. Regeln bei der Parameterübergabe
90. Ergebnisprozedur
91. Formales und aktuelles Ergebnis in Java
92. Kontrollfluss bei Prozeduraufrufen
93. Kontrollfluss und Prozedur-Mechanismus
94. Unterschiede zwischen Prozeduren und Methoden
95. Zwischenergebnis Prozedur/Methode
96. Drei Arten von Variablen
97. Deklaration der Variablenarten
98. Anfangsbelegung der Variablenarten
99. Veränderung eines Variablenwerts



Grundlegende Lehrbücher



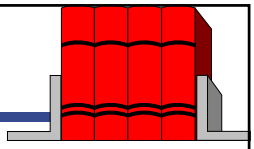
David J. Barnes, Michael Kölling: **Java lernen mit BlueJ – Eine Einführung in die objektorientierte Programmierung**, 4. Aufl., Pearson Studium, 2009. (deutsche Übersetzung von: **Objects First with Java - A Practical Introduction using BlueJ**, 4. Aufl., Pearson Education, 2009.)
[Der aktuelle „Objects First“ Ansatz mit BlueJ. Teilweise Grundlage für die Übungen. Gut geeignet zum Selbststudium.]

Cornelia Heinisch, Frank Müller-Hofmann, Joachim Goll: **Java als erste Programmiersprache**, 6. überarb. u. erw. Aufl., Teubner, Stuttgart, 2010.
[Eine gute konventionelle Einführung in Java.]

Khalid Mughal, Torill Hamre, Rolf W. Rasmussen: **Java Actually: A First Course in Programming**. Thomson, 2007.
[Englischsprachige Einführung in das Programmieren mit Java, die – ähnlich wie wir – Vererbung bewusst ausklammert.]

SE1 – Level 1 2

Mehr zu Java



Reinhard Schiedermeier, Klaus Köhler: **Das Java-Praktikum**, 2. überarb. u. erw. Aufl., dpunkt Verlag, 2011.
[Eine sehr nützliche Sammlung von Aufgaben zu Java.]

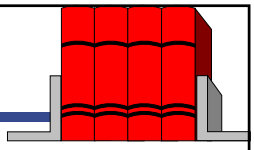
Ken Arnold, James Gosling, David Holmes: **The Java Programming Language**, Fourth Edition, Addison-Wesley, 2005.
[Der Java-Klassiker. Knapp und ohne didaktischen Anspruch. Eher zum Einlesen für erfahrene Programmierer.]

David Flanagan: **Java in a Nutshell**, 5. Aufl., O'Reilly Media, 2005.
[Der Java-Nachschlage-Klassiker. Kurz und knapp (auf 1224 Seiten) durch die wesentlichen Java-Bestandteile und -Packages.]

Joshua Bloch: **Effective Java Programming Language Guide**, 2. Aufl., Addison-Wesley Longman, 2008.
[Die Fallstricke von Java ausführlich und sehr kompetent. Eher für Fortgeschrittene.]

James Gosling, Bill Joy, Guy Steele: **The Java Language Specification**, Third Edition, Addison-Wesley, Juli 2005.
[Die offizielle Sprachdefinition. Für die, die es genau wissen wollen.]

Weitere Grundlagenwerke

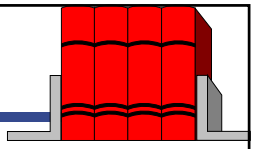


Peter Rechenberg, Gustav Pomberger, **Informatik-Handbuch**, Hanser-Verlag, 4., aktualis. u. erw. Aufl., 2006.
[Handbuch der wesentlichen Gebiete der Informatik.]

Duden Informatik, Dudenverlag, Ausgabe 2006.
[Grundbegriffe kurz und grundlegend definiert.]

Grady Booch, James Rumbaugh, Ivar Jacobson, **The Unified Modeling Language Reference Manual**, Addison-Wesley, 2004.
[Die Referenz von den „Erfindern“ der UML.]

Englischsprachig und weiterführend



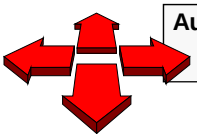
Robert W. Sebesta, **Concepts of Programming Languages**, Addison-Wesley Educational Publishers, 9. Auflage, 2009.
[Gute und verständliche Einführung in die Definition von Programmiersprachen.]

Bertrand Meyer: **Object-Oriented Software Construction**, Second Edition, Prentice Hall, 1997.
[Der Klassiker unter den Programmierbüchern zur Objektorientierung (am Beispiel der Sprache Eiffel). Viele allgemeingültige und wertvolle Hinweise. Engagiert und bissig.]

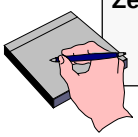
Heinz Züllighoven et al.: **Object-Oriented Construction Handbook**. dpunkt-Verlag, 2004.
[Unser Diskussionsbeitrag. Für Fortgeschrittene (und die, die es werden wollen :-)]

SE1 – Level 1 5

Symbole und Zeichen



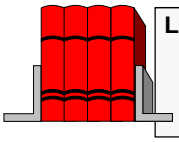
Ausblick: sagt, was kommt.



Zentraler Begriff: wird hier eingeführt oder erläutert.

Zwischensumme: hebt für einzelne Themen hervor, was uns wichtig ist.

Vertiefung: hier werden Hintergründe etwas genauer beleuchtet.



Literaturliste: Grundlage für den nachfolgenden Teil mit Bewertung.

© <Autor> **Literaturreferenz:** auf Bücher aus der Literaturliste.

SE1 – Level 1 6

Symbole und Zeichen (II)

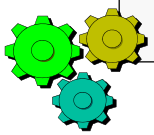
Merker:
kommentiert, stellt Bezüge her und hebt etwas hervor.

Negativbeispiel:
warnt vor häufigen, aber bedenklichen Konstruktionen.

Imperativer Begriff:
hier wird ein Begriff aus der imperativen Programmierung verwendet, der sich von der OO-Terminologie unterscheidet.

UML-Notation:
Diagramm in der Notation der Unified Modeling Language.

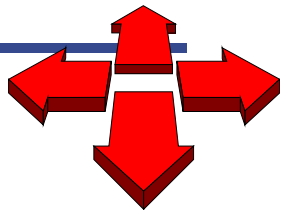
Java-Beispiel:
erläutert ein Konzept oder Konstrukt am Beispiel der Sprache Java.



Softwaretechnik:
diskutiert Erfahrungen und Prinzipien.

SE1 – Level 1 7

Die objektorientierte Sichtweise



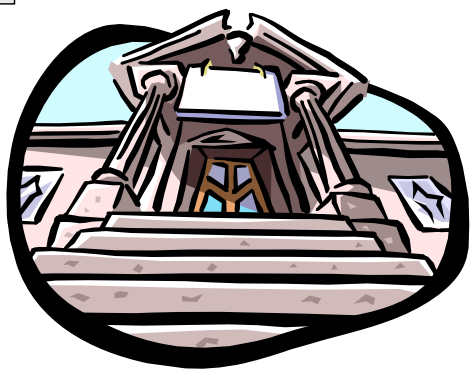
- Objektorientierte Modellierung und Programmierung
- Objekte: Klienten und Dienstleister
- Objekte zeigen Verhalten und haben Zustände
- Methoden und Zustandsfelder
- Klassen als Blaupausen für Exemplare
- Methoden bestehen aus imperativen Anweisungen
- Imperative Grundkonzepte in der Übersicht

- Andere Paradigmen: Funktionale und Logische Programmierung

SE1 – Level 1 8

Ein Szenario

Ein Bankkunde geht zu seiner Bank



SE1 – Level 1 9

Ein Szenario (II)

Der Bankkunde druckt sich am Bankautomat in der Schalterhalle seine Kontoauszüge aus.

SE1 – Level 1 10

Ein Szenario (III)

Dann lässt sich der Bankkunde mit seiner ec-Karte 300 EUR am Bankautomat von seinem Girokonto auszahlen.

SE1 – Level 1 11

Das Szenario: Kunde erledigt Bankgeschäfte

Personen erledigen Aufgaben, um ein Ziel zu erreichen.

Aufgaben werden durch verschiedene Tätigkeiten / Handlungen erledigt.

Handlungen sind charakterisiert durch die Art und Weise, wie die Personen mit Gegenständen umgehen.

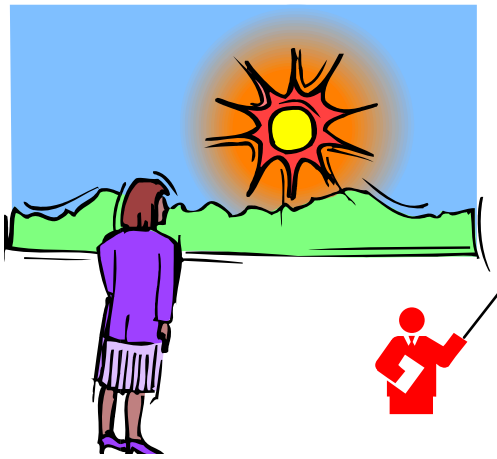
Ein Bankkunde geht zu seiner Bank.

Der Bankkunde druckt sich am Bankautomat in der Schalterhalle seine Kontoauszüge aus.

Dann lässt sich der Bankkunde mit seiner ec-Karte 300 EUR am Bankautomat von seinem Girokonto auszahlen.

SE1 – Level 1 12

Objektorientierung ist eine „Sicht der Welt“



Oft spricht man von einem **Paradigma**. Damit ist eine Sichtweise gemeint, die uns hilft, einen Sachverhalt zu interpretieren und zu verstehen.

Paradigma [*gr.-lat.*] *das*; -s, ...men (auch: -ta): ...
4. Denkmuster, das das wissenschaftliche Weltbild, die Weltsicht einer Zeit prägt.

SE1 – Level 1 13

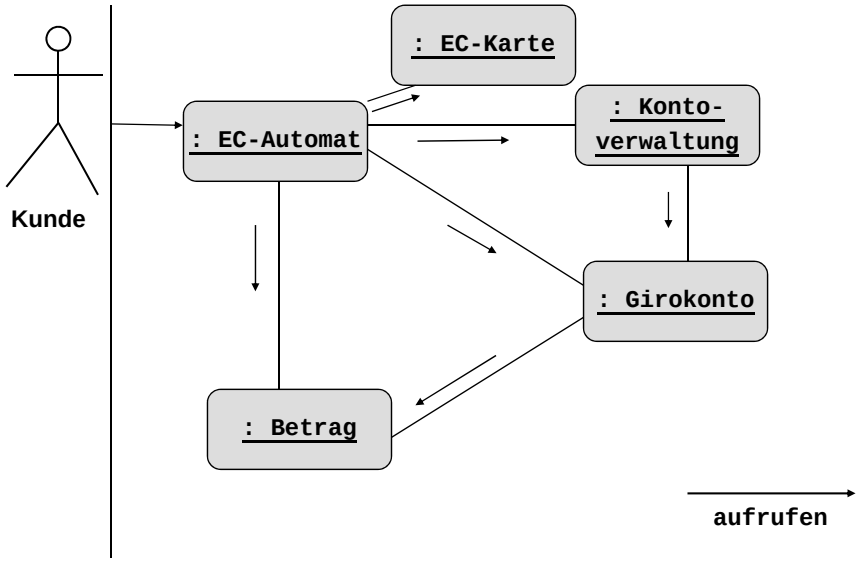
Modellieren und Programmieren

- Modellieren:
  - Ein Abbild von etwas machen, um damit etwas zu zeigen, prüfen, auszuprobieren.
  - Beim Modellieren verwenden wir oft vorgegebene „Modell-elemente“, Regeln und eine bestimmte Herangehensweise.
  - Objektorientierte Modellierung ist ein Beispiel für eine solche Modellierungsart.

- Programmieren:
  - Ein Programm für einen Rechner in einer Programmiersprache schreiben, testen, weiterentwickeln.
  - Beim Programmieren verwenden wir die Elemente der gegebenen Programmiersprache, Regeln der Programmkonstruktion und bestimmte Herangehensweisen.
  - Objektorientierte Programmierung in Java ist eine solche Art der Programmierung.

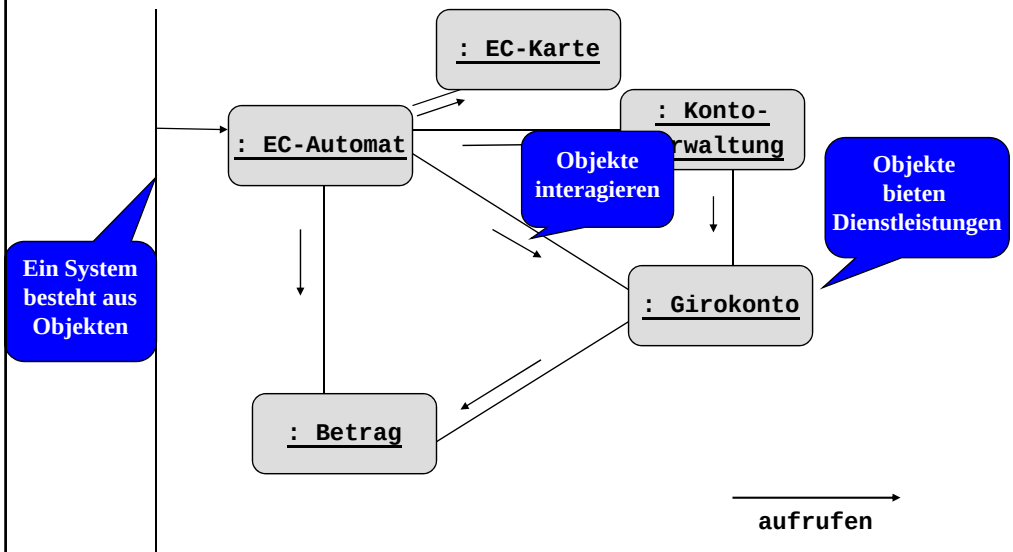
SE1 – Level 1 14

Akteure interagieren mit einem System von Objekten



SE1 – Level 1 15

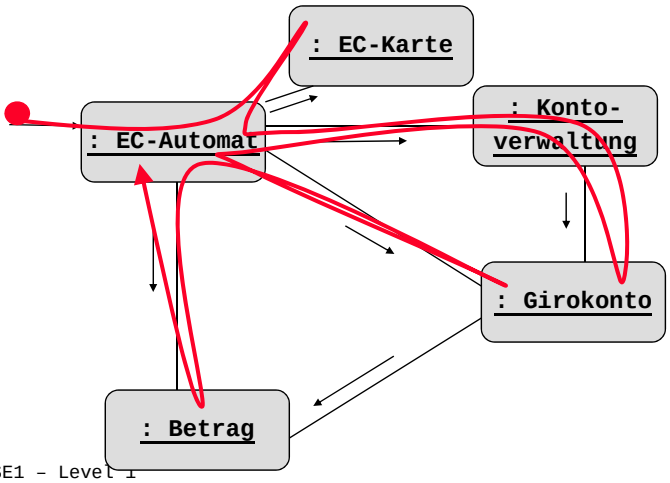
In Objektwelten laufen Prozesse auf Daten ab



SE1 – Level 1 16

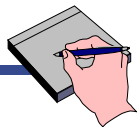
Vereinfachung: Ein Prozess „durchläuft“ Objekte

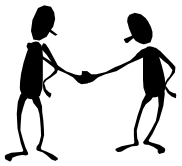
Unser **intuitives Verständnis** von miteinander arbeitenden Objekten ist, dass jedes Objekt unabhängig von anderen Objekten aktiv sein kann; es kann auf Anfragen von anderen Objekten warten oder es kann parallel beliebige Dinge tun.



Wir verwenden ein **vereinfachtes Modell** In SE1 (und SE2): **Alle Objekte sind passiv**; sie warten darauf, dass eine Dienstleistung angefordert wird, erbringen diese auf Anfrage und sind ansonsten untätig.

Dienstleister und Klienten

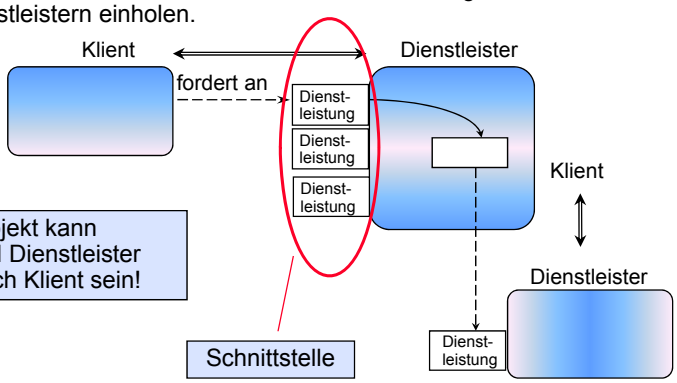




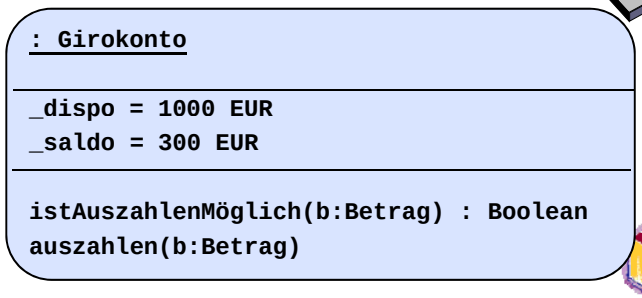
- Das Objekt, das bei einer bestimmten (Teil-)Aufgabe einen Dienst leistet, ist der **Dienstleister**.
- Das Objekt, das eine konkrete Dienstleistung eines anderen Objektes in Anspruch nimmt, wird als **Klient** bezeichnet.

Dienstleistungen an der Schnittstelle

- Objekte bieten **Dienstleistungen** als **Methoden** an ihrer **Schnittstelle** an.
- Diese Dienstleistungen werden von anderen Objekten, den Klienten, benutzt. Dazu fordert der Klient eine Dienstleistung des Anbieters an.
- Der Anbieter kann selbst wieder Teile seiner Dienstleistung von anderen Dienstleistern einholen.



Dienstleistungen: Verhalten und Zustand



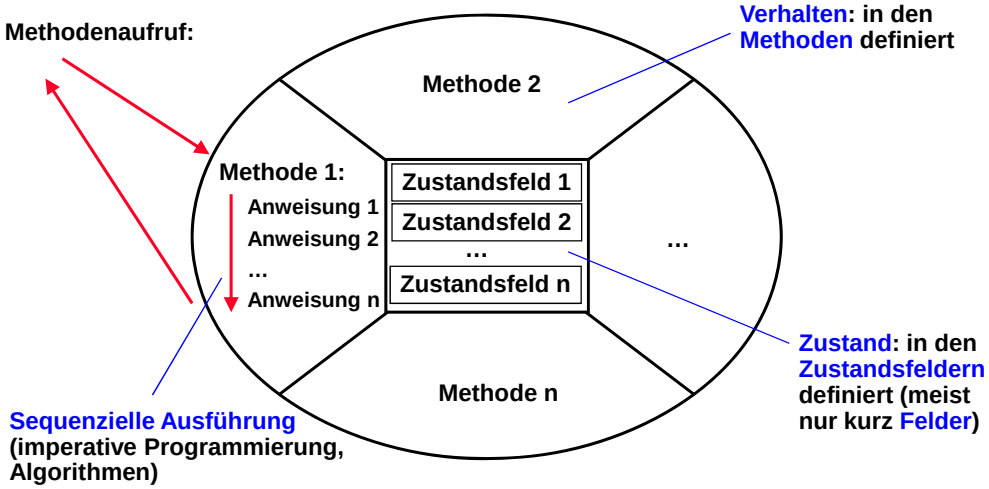
Zustand

Verhalten

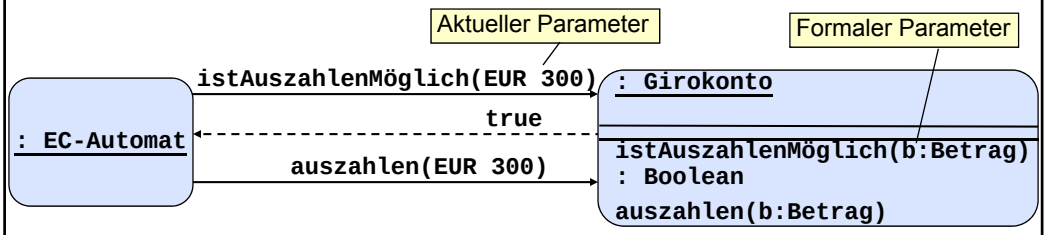
- Das **Verhalten** eines Objekts ist durch seine angebotenen Dienstleistungen, also durch seine **Methoden** bestimmt.
- Die Umsetzung dieser Dienstleistungen ist einem Klienten verborgen.
- Ein Objekt kann einen **Zustand** haben und seine Dienstleistungen von diesem Zustand abhängig machen.

Bsp.: Je nach Zustand eines Kontos ist das Auszahlen mal möglich und mal nicht…

Logische Sicht auf ein Objekt



Objekte interagieren über Methodenaufrufe



- Objekte interagieren, indem Objekte (Klienten) **Methoden** an anderen Objekten (Dienstleistern) **aufrufen**.
- Ein Methodenaufruf kann **parametrisiert** werden; der Klient gibt beim Aufruf konkrete Werte als sog. **aktuelle Parameter** an; der Dienstleister arbeitet dann auf Kopien dieser Parameter, die für ihn **formale Parameter** genannt werden.
- Der Dienstleister kann nach dem Ende einer Methodenausführung ein **Ergebnis** an den Klienten zurückgeben.

Signatur einer Methode

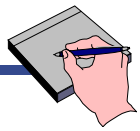
- Die **Signatur** einer Methode liefert die relevanten Informationen für einen **Methodenaufruf**. In Java umfasst dies:
  - **Name der Methode**
  - **Anzahl, Reihenfolge und Typen der Parameter**
- Bei der Beschreibung einer Methode werden für eine sinnvolle Benutzung zusätzlich weitere Informationen angegeben:
  - Parameternamen
  - Ergebnistyp
  - Methodenkommentar
- Diese Informationen sind in Java **formal nicht Teil der Signatur**.
- Beispiel:

```
boolean istAuszahlenMöglich(Betrag b)
```

Signatur

istAuszahlenMöglich(Betrag)

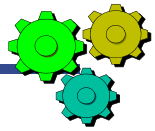
Klassen als Schablonen für Exemplare



Girokonto
_dispo : Betrag **_saldo : Betrag**
istAuszahlenMöglich(b:Betrag) : Boolean **auszahlen(b:Betrag)**

- Als **Exemplar** bezeichnet man das aus einer **Klasse** erzeugte **Objekt**.
- Eine Klasse definiert somit das **prinzipielle Verhalten** aller ihrer Exemplare.
- Von einer Klasse können **beliebig viele** Exemplare erzeugt werden.
- Aber: Jedes Exemplar hat einen **eigenen Zustand**, der verändert werden kann, und kann deshalb anders auf **dieselbe Anfrage** reagieren.

Programmieren im Kleinen



- Spricht man in der Softwaretechnik vom „Programmieren im Kleinen“, meinte man früher primär die Umsetzung eines **Algorithmus** in ein lauffähiges **Programm**, das aus **Anweisungen** besteht. Das ist ein zentrales Thema von SE1.

- Heutzutage bezieht „Programmieren im Kleinen“ auch die Realisierung von **Klassen** mit ihren **Methoden** und den Anweisungen innerhalb der Methoden mit ein. Dies ist ebenfalls ein zentrales Thema in SE1.

„**Programmieren im Großen**“ bedeutet, komplexe Anwendungsprobleme professionell im Team zu lösen. Dabei spielt die Strukturierung von sehr umfangreichen Programmen durch eine sog. **Softwarearchitektur** eine große Rolle. Einen Ausblick darauf geben wir in SE2.

Programmieren im Kleinen: ein imperativer Algorithmus

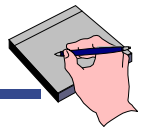
Beispiel:

```
(1) Vergleiche zwei natürliche Zahlen a und b.
(2) Wenn a größer als b ist, setze das Ergebnis
    max gleich dem Wert von a.
(3) Sonst setze das Ergebnis max gleich dem Wert
    von b.
```

Auswertung des Beispiels:

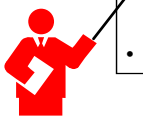
- Der Algorithmus besteht aus einer Folge von **Aktionen** (hier **`vergleiche`**, **`setze`**).
- Jede Aktion bezieht sich auf *Variablen* (hier **`a,b,max`**), die durch die Aktion gegebenenfalls verändert werden.
- Jeder Variablen ist ein *Typ* zugeordnet (hier natürliche Zahlen).

Imperative Programmierung



Das Paradigma **imperative Programmierung**:

- Programme werden als **Folgen von Anweisungen** formuliert.
- Die **Ausführungsreihenfolge** der Anweisungen ist durch die textuelle Reihenfolge oder durch Sprunganweisungen festgelegt.
- **Höhere Programmkonstrukte** fassen Anweisungsfolgen zusammen und bestimmen die Ausführungsreihenfolge.
- Benannte **Variablen** können Werte annehmen, die sich durch Anweisungen ändern lassen.

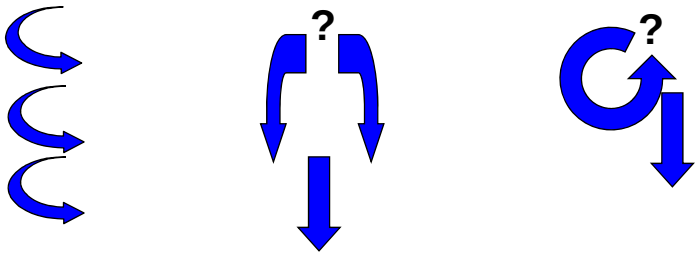


- Imperative Programmierung baut auf dem Konzept des *v. Neumann-Rechners* auf.
- Sie beruht auf dem *Zustandskonzept*.

SE1 – Level 1 27

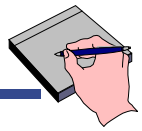
Ablaufsteuerung durch Kontrollstrukturen

- Ablaufsteuerung in **Programmiersprachen** durch **Kontrollstrukturen**:
  - Die Ausführungsreihenfolge der Anweisungen einer Methode entspricht zunächst der textlichen Anordnung (**Sequenz**). Davon kann aber abgewichen werden. Dazu gibt es spezielle Mechanismen der Ablaufsteuerung:
    - **Fallunterscheidung**
    - **Wiederholung**



SE1 – Level 1 28

Kontrollstruktur 1: Sequenz



Sequenz von Anweisungen:

- Eine Anweisung wird nach der anderen abgearbeitet. Dazu muss nur klar sein, wie zwei Anweisungen voneinander getrennt sind.
- Ein Anweisung kann auch die leere Aktion sein („tue nichts“).

Informeller Algorithmus **Telefonieren**:

hebe den Hörer ab;
wähle die Telefonnummer;
führe das Gespräch;
lege den Hörer auf.

```
...
int i = 4;
int j = 5;
int k = 6;
...
```

Kontrollstruktur 2: Fallunterscheidung

Der Mechanismus zur **Fallunterscheidung**:

- Abhängig vom Ergebnis einer Fallunterscheidung werden verschiedene Anweisungsfolgen ausgeführt.
- Das Grundschema der Fallunterscheidung ist:
  WENN … DANN … SONST … ENDE (*WENN*)

Informeller Algorithmus **Telefonieren**:

hebe den Hörer ab;
WENN Telefonnummer gespeichert
    DANN drücke Kurzwahltaste
    SONST wähle die Telefonnummer
ENDE (*WENN*)
WENN Gesprächspartner antwortet
    DANN führe das Gespräch
ENDE (*WENN*)
lege den Hörer auf.

```
if (a < b)
{
  min = a;
}
else
{
  min = b;
}
```

Kontrollstruktur 3: Wiederholung

Der Mechanismus zur **Wiederholung** von Anweisungen (Schleife):

- Anweisungsfolgen werden wiederholt ausgeführt.
- Das Ende der Wiederholung ist mit einer **logischen Bedingung** verknüpft.
- Wir unterscheiden konzeptionell:
  - "Solange-Noch"-Schleifen: **`SOLANGE`** … **`WIEDERHOLE`** … **`ENDE`**,
  - "Solange-Bis"-Schleifen: **`WIEDERHOLE`** … **`BIS …`** .

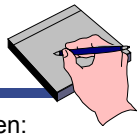
Aus dem Algorithmus **Telefonieren**:

```
SOLANGE  Geld da
  WIEDERHOLE
    hebe den Hörer ab;
    wirf Geld ein;
    führe Gespräch;
    lege den Hörer auf;
ENDE .
```

Aus dem Algorithmus **Telefonieren**:

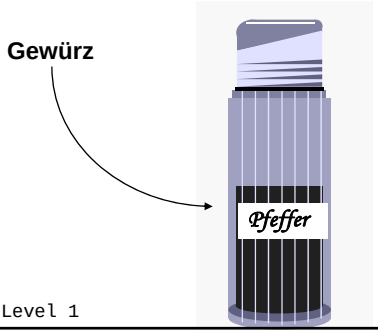
```
hole Liste der Gesprächspartner
WIEDERHOLE
    führe ein Gespräch;
    streiche Gesprächspartner;
BIS Liste abgehakt .
```

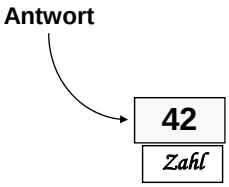

Imperative Variablen



Der Begriff **Variable** ist grundlegend für das Verständnis imperativer Sprachen:

- Eine Variable ist eine **Abstraktion eines physischen Speicherplatzes**.
- Sie hat einen **Namen** (häufig auch: **Bezeichner**), über den sie angesprochen werden kann.
- Eine **Variable** hat den Charakter eines Behälters:
  - Sie hat eine **Belegung** (ihren aktuellen Inhalt), die sich **ändern** kann;
  - und einen **Typ**, der Wertemenge sowie zulässige Operationen und weitere Eigenschaften festlegt.





Die Typen sind hier *Pfeffer* und *Zahl*.

Deklaration und Initialisierung

- **Vor der Verwendung** einer Variablen in imperativen Programmiersprachen muss sie bekanntgemacht, d.h. **deklariert** werden.
- Vereinfacht geschieht dies durch:
  - Angabe des **Typs**,
  - Vergabe eines **Namens** über einen **Bezeichner** (engl.: identifier).
- Durch die **reine Deklaration** von Variablen ist deren **Belegung** zunächst meist **undefiniert**.
- Erst bei der **Initialisierung** wird eine Variable erstmalig mit einem gültigen Wert befüllt.

Deklaration

```
int i;
boolean b;
```

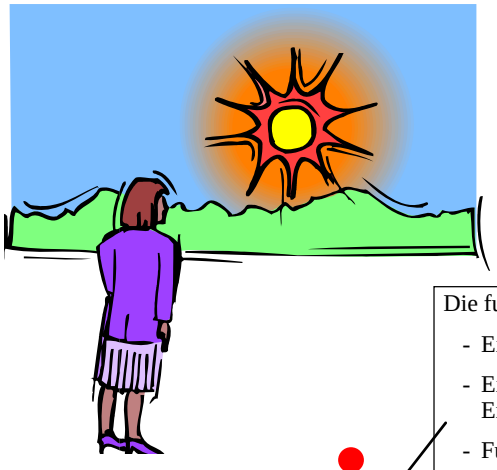
Deklaration und Initialisierung

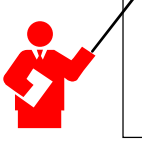
```
int i = 42;
boolean b;

b = true;
```

SE1 – Level 1 33

Es gibt andere informatische Sichten der Welt: z.B. die funktionale Sicht





Die funktionale Sicht

- Ein Programm besteht aus Funktionen.
- Eine Funktion berechnet einen Ergebniswert anhand von Eingabewerten.
- Funktionen können kombiniert werden.
- Ein (Wert-) Ausdruck kann immer durch einen wertgleichen anderen Ausdruck substituiert werden.

SE1 – Level 1 34

Beispiel für funktionale Programmierung

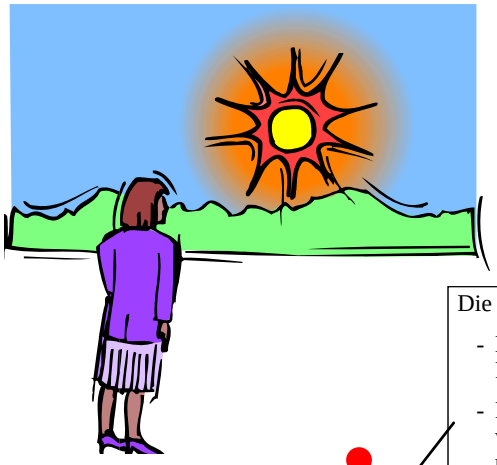
```
(define (max a b)
  (if (> a b) a b))
```

- Eine Funktion „max“ wird definiert.
- Sie nimmt die Werte **a** und **b**.
- Sie gibt den größeren der beiden Werte zurück.

Mehr dazu:
wählbar in SE3.

SE1 – Level 1 35

Es gibt andere informatische Sichten der Welt: z.B. die logische Sicht



Die logische Sicht

- Ein Programm besteht aus logischen Regeln und Fakten.
- Logische Anfragen können evaluiert werden. Dabei wird anhand der Regeln und Fakten abgeleitet, ob eine logische Anfrage wahr ist.

SE1 – Level 1 36

Beispiel für logische Programmierung

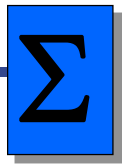
```
max(A,A,A).
max(A,B,A) :- A > B.
max(A,B,B) :- A < B.
```

- Ein dreistelliges Prädikat „**max**“ wird definiert.
- 1. Regel: Wenn zwei Zahlen gleich sind, ist auch ihr Maximum gleich.
- 2. Regel: Wenn **A** größer **B**, dann ist ihr Maximum gleich **A**.
- 3. Regel: Wenn **A** kleiner **B**, dann ist ihr Maximum gleich **B**.

Mehr dazu: wählbar in SE3.

SE1 – Level 1 37

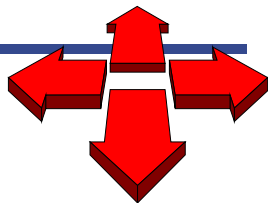
Zusammenfassung



- Softwaresysteme können als eine **Menge interagierender Objekte** modelliert und programmiert werden.
- Objekte haben einen **Zustand** und bieten **Dienstleistungen** an.
- Dienstleistungen werden in Form von **Methoden** angeboten.
- Der Zustand wird durch **Zustandsfelder** realisiert.
- Die für Klienten aufrufbaren Methoden eines Objektes bilden seine **Schnittstelle**.
- **Klassen** sind Schablonen zur Erzeugung von **Exemplaren**.
- Innerhalb einer Methode werden **Anweisungen sequenziell** ausgeführt.
- Die Anweisungen in einer Methode werden nach den Prinzipien imperativer **Kontrollstrukturen** ausgeführt: **Sequenz**, **Fallunterscheidung**, **Wiederholung**.
- **Variablen**, die ihre Belegung dynamisch ändern können, sind zentral in der imperativen Programmierung.
- Es gibt auch andere Sichtweisen der Programmierung: die **funktionale** und die **logische** Sichtweise.

SE1 – Level 1 38

Die Struktur von Klassendefinitionen



- **Logischer Aufbau von Klassen**
  - Konstruktoren
  - Zustandsfelder
  - Methoden
- **Übersetzung**
  - Hybride Sprachen
  - Virtuelle Maschinen
- **Syntaktischer Aufbau**
  - Formale Sprachen
  - Syntax in EBNF

Unsere erste selbst geschriebene Klassendefinition

```
class Girokonto
{
  private int _saldo;

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

- Ein Java-Programm besteht aus Textdateien.
- In jeder Textdatei ist eine Klasse beschrieben.
- Die textuelle Beschreibung einer Klasse nennen wir **Klassendefinition**.
- Wir bearbeiten Klassendefinitionen mit einem **Editor**.

Merkmale unserer ersten Klasse

```
class Girokonto
{
  private int  saldo;

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

- Java-Programme bestehen aus **Klassen** (hier: `Girokonto`).
- Die Klasse definiert eine **Methode** (hier: `einzahlen`).
- Die Methode erhält einen Parameter (hier: `betrag` vom Typ `int`) und hat keinen Rückgabewert (hier: Schlüsselwort `void`).
- Im Rumpf der Methode wird ein Wert einem **Zustandsfeld** zugewiesen (hier: `_saldo`).
- Das Feld muss **deklariert** sein (hier vom Typ `int`).
- Alternativ nennen wir die Felder in einer Klassendefinition auch **Exemplarvariablen**.

SE1 – Level 1 41

Abgleich mit den Prinzipien der Objektorientierung

```
class Girokonto
{
  private int _saldo;

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

- Das Verhalten eines Objekts ist durch seine angebotenen Dienstleistungen (Methoden) bestimmt.
  - ✓ `einzahlen` **ist durch** `public` **für Klienten aufrufbar.**
- Die Realisierung dieser (zusammengehörigen) Dienstleistungen (als Methoden) ist verborgen.
  - ✓ **Kein Zugriff durch Klienten auf die Implementierung von** `einzahlen`
- Ebenso sind die Zustandsfelder als interne Strukturen eines Objekts gekapselt.
  - ✓ **Das Feld** `_saldo` **ist durch** `private` **vor externem Zugriff geschützt.**
- Auf den Zustand eines Objektes kann nur über seine Dienstleistungen zugegriffen werden.
  - ✓ **Hier durch** `einzahlen`

SE1 – Level 1 42

Auswertung: Grobstruktur einer Klassendefinition

```
/**
 * Schnittstellenkommentar der Klasse
 */
class Girokonto
{
  private int _saldo;

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

Kopf der Klasse

Rumpf der Klasse

Klassenkopf: spezifiziert den Namen der Klasse und beschreibt mit dem Schnittstellenkommentar die Aufgabe der Klasse.

Klassenrumpf: beinhaltet Zustandsfelder, Konstruktoren und Methoden, die die Zuständigkeiten der Klasse realisieren.

Auswertung: allgemeine Struktur einer Klassendefinition

```
class Girokonto
{
  private int _saldo;

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

Konstruktor kann fehlen (Standardkonstruktor)

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

Objekte erzeugen

Objekterzeugung:
Objekte müssen zur Laufzeit durch einen expliziten Ausdruck erzeugt werden. Dazu wird ein eigenes **Schlüsselwort** (in Java **new**) verwendet.

```
class Zeichner {
  ...
  Quadrat wand = new Quadrat();
  Dreieck dach = new Dreieck();
  Quadrat fenster = new Quadrat();
  ...
  wand.vertikalBewegen(80);
  fenster.farbeAendern("blau");
  dach.horizontalBewegen(70);
  ...
}
```

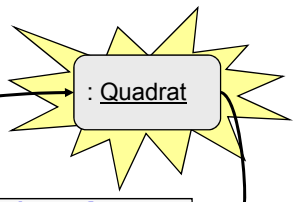
Schlüsselwort: Zeichenfolge, die in einer Programmiersprache eine feste Bedeutung hat (z.B. **if**). Schlüsselwörter sind (meist) reserviert, d.h. sie dürfen nicht als Namen von Variablen verwendet werden.

SE1 – Level 1 45

Konstruktoraufruf und Konstruktor

- Ein **Konstruktoraufruf** (in Java mit **new**) bewirkt zweierlei:
  1. Ein **neues Objekt** der genannten Klasse wird erzeugt.
  2. Bei diesem Objekt wird der angegebene **Konstruktor ausgeführt**; ein Konstruktor **initialisiert** ein neu erzeugtes Objekt.

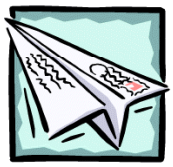
```
class Zeichner {
  ...
  Quadrat wand = new Quadrat();
  Dreieck dach = new Dreieck();
  Quadrat fenster = new Quadrat();
  ...
  wand.vertikalBewegen(80);
  fenster.farbeAendern("blau");
  dach.horizontalBewegen(70);
  ...
}
```




```
class Quadrat {
  ...
  public Quadrat()
  {
    <Initialisierungen>
  }
  ...
}
```

SE1 – Level 1 46

Methoden aufrufen



- Jeder **Methodenaufruf** richtet sich immer an ein bestimmtes Objekt, den **Adressaten** des Aufrufs.
- Der Adressat ist entweder explizit angegeben:

```
wand.vertikalBewegen(80);
```

  Die gerufene Methode ist dann üblicherweise Teil der **Schnittstelle** des gerufenen Objektes.
- Oder es wird eine Methode des aktuellen Objektes aufgerufen:

```
zeichneDach(80);
```

  Hilfsmethoden, die nur innerhalb einer Klasse verwendet werden, werden `private` deklariert.

Botschaft: Der Aufruf einer Methode wird oft auch als das Senden einer Botschaft oder Nachricht an das gerufene Objekt dargestellt. Dabei umfasst die Botschaft einen Bezeichner für das Objekt (als Adressaten), den Namen der Methode und die aktuellen Aufrufparameter.

Die Punktnotation der Objektorientierung

- Die Methoden eines Objekts werden in vielen objektorientierten Sprachen in der **Punktnotation** (engl.: dot notation) aufgerufen.

```
wand.vertikalBewegen 80
```

Das aufzurufende **Objekt** wird benannt.

Der **Punkt-Operator** sagt aus, dass auf einen Bestandteil der Schnittstelle (also eine Dienstleistung des Objektes) zugegriffen werden soll.

Die aufzurufende **Methode** wird benannt, eventuell gefolgt von **aktuellen Parametern** (hier die ganze Zahl **80**).

Die Punktnotation in Java

- Java folgt der objektorientierten Tradition und verwendet ebenfalls die Punktnotation für Methodenaufrufe an Objekten.

```
wand.vertikalBewegen(80);
```

Referenz auf ein Objekt

der Punkt

Die **aktuellen Parameter** werden in Java in **runden Klammern** eingeschlossen.

Auch wenn keine Parameter definiert sind, werden bei Methodenaufrufen die **runden Klammern immer** angegeben.

SE1 – Level 1 49

Struktur der Methodendefinition in Java

Methodenköpfe: Klassen spezifizieren mit den Köpfen ihrer öffentlichen Methoden Dienstleistungen, die festlegen, wie die Zustände der Objekte sondiert oder verändert werden können. Die öffentlichen Methoden bilden die Schnittstelle einer Klasse.

Methodenrümpfe: realisieren die versprochenen Dienstleistungen durch eine Implementierung (konkrete Deklarationen und Anweisungsfolgen).

Die Unterscheidung zwischen der Schnittstelle (dem „Kopf“) und der Implementierung (dem „Rumpf“) einer Methode spiegelt sich auch in ihrer Struktur wider.

Hier ist in Java die **Signatur** enthalten.

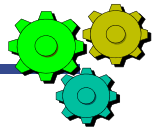
```
/**
 * Kommentar, der die Funktionalität der Methode beschreibt
 */
<Zugriffsmodifikator> <Ergebnistyp> <Name> ( <Parameter> )
{
        <(imperative) Anweisungen>
}
```

Kopf der Methode

Rumpf der Methode

SE1 – Level 1 50

Verändernde Methoden



- Pragmatik: Wenn Methoden den Zustand ihres Objektes verändern (**verändernde Methoden**, engl.: mutators), dann sollten sie keinen Wert zurück geben.
- Für Klienten sind nur die Methoden aufrufbar, die mit **`public`** als „öffentlich“ deklariert wurden; sie bilden die Schnittstelle einer Klasse.
- Neben den öffentlichen Methoden werden zur Implementierung oft interne Methoden verwendet. Sie werden in Java als **`private`** deklariert und entsprechen dem ursprünglichen imperativen Prozedurbegriff.

für Klienten sichtbar

benutzerdefinierter Bezeichner für die Methode (Pragmatik für Java: klein geschrieben)

```
public void einzahlen( int betrag )
```

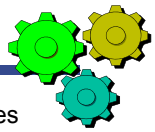
Parametername

Methode liefert keinen Rückgabewert

formaler Parameter vom Typ **`int`**

SE1 – Level 1 51

Sondierende Methoden



- **Sondierende Methoden** (engl.: accessor methods) sollen den Zustand des Objektes, an dem sie gerufen werden, nicht verändern.
- Sondierende Methoden liefern einen (Ergebnis-) Wert von einem vereinbarten (Ergebnis-) Typ.
- Das Ergebnis wird explizit (mittels der **`return`** Anweisung) zurückgegeben.
- Solche Methoden können deshalb an der Aufrufstelle als Teil von Ausdrücken verwendet werden.

Ergebnistyp **`int`**

benutzerdefinierter Bezeichner für die sondierende Methode

```
public int gibSaldo( )
```

für Klienten sichtbar

Methode hat keine Parameter

SE1 – Level 1 52

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 53

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 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.

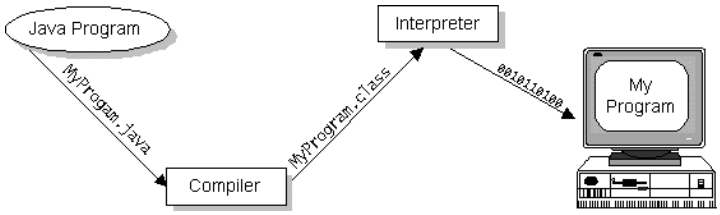
SE1 – Level 1 54

Hybride Verarbeitung in Java

Die hybride Verarbeitung bei Java ist eine Kombination:

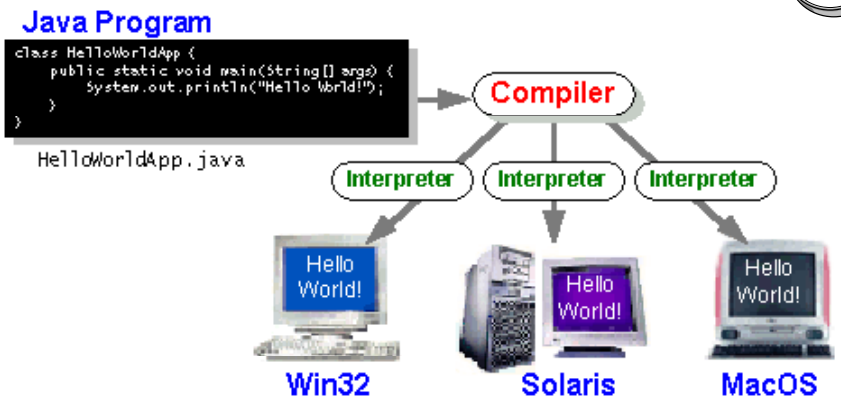
- Der **Quelltext** wird von einem Compiler in **Zwischencode** transformiert, der **Java Bytecode** genannt wird.
- Dieser Zwischencode wird dem Interpreter übergeben, der sog. **Java Virtual Machine**.

... 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.



SE1 – Level 1 55

Die Java Virtual Machine

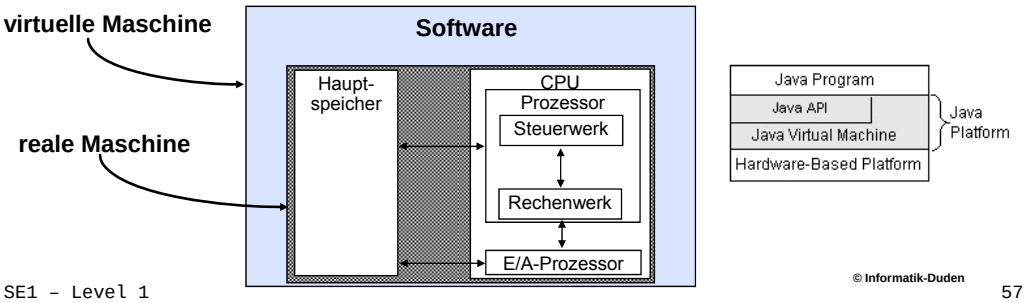


... You can think of **Java bytecodes** as the **machine code instructions** for the **Java Virtual Machine (Java VM)**.
... Java bytecodes help make **"write once, run anywhere"** possible.

SE1 – Level 1 56

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.



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

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

- 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.

SE1 – Level 1 61

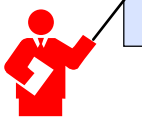
Syntaxbeschreibungen von Programmiersprachen

- 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.

SE1 – Level 1 62

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.

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:

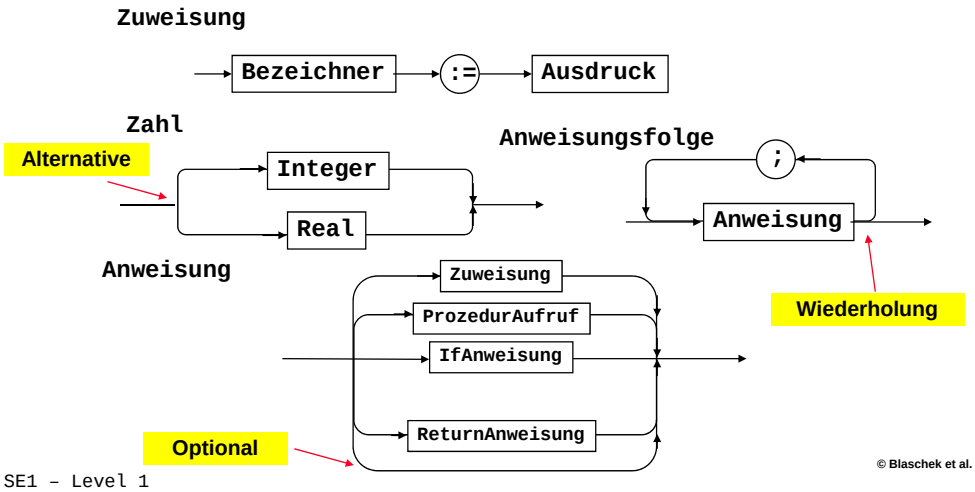
```
Anweisungsfolge → Anweisung { ';' Anweisung }
```

- **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.



SE1 – Level 1 69

Java Level 1: Syntax für Einsteiger

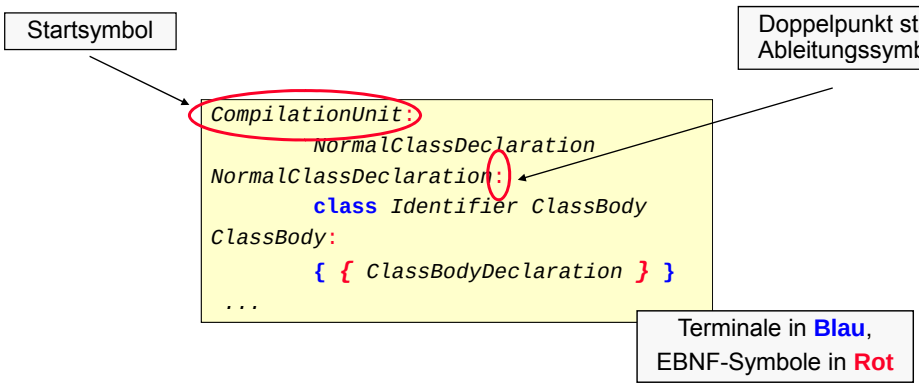
- Die Syntaxbeschreibung von Java ist sehr umfangreich; viele Ausnahmen und fortgeschrittene Konzepte erschweren das Einlesen für Programmieranfänger.
- Für SE1 haben wir deshalb eine Untermenge von Java namens **Java Level 1** definiert, die nur wenige Sprachkonzepte enthält, u.a. Klassen, Felder, Methoden, 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:

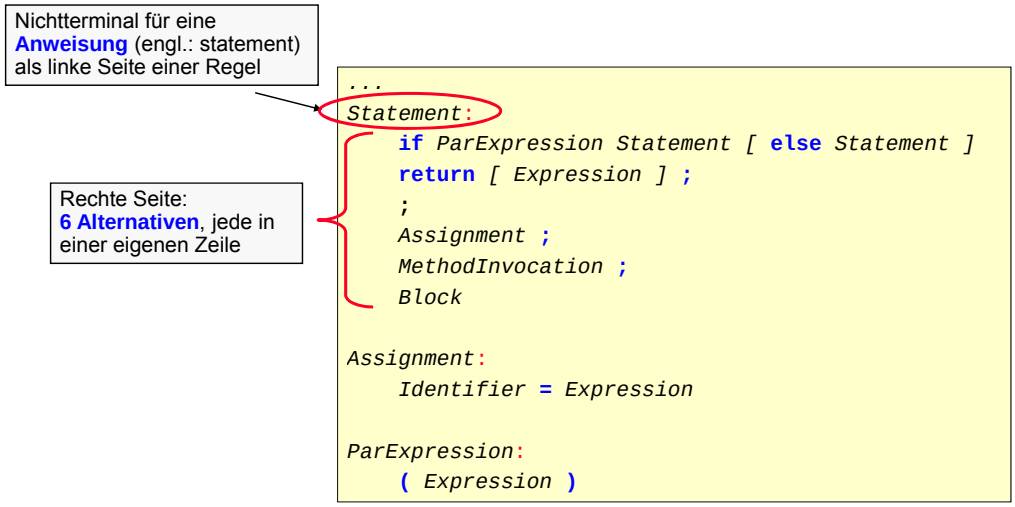
  http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html
- Die Grammatik ist leicht erweiterbar um weitere Konzepte, die im Lauf der Vorlesung vorgestellt werden, wie Schleifen, Interfaces etc.

SE1 – Level 1 70

Ausschnitt aus der Syntax von Java Level 1



Ausschnitt aus der Syntax von Java Level 1 (II)



Beispiel einer Ableitung

- Wir leiten als Beispiel folgende Anweisung mit Hilfe der Level 1-Syntax ab:
  `i = j + 3;`
- Eine solche einzelne **Anweisung** (von der es innerhalb eines Methodenrumpfes beliebig viele geben kann) wird aus dem Nichtterminal *Statement* (engl. für Anweisung) abgeleitet. Auf der vorigen Folie sind die sechs möglichen Alternativen für *Statement* aufgeführt.
- Von diesen Alternativen wählen wir diejenige für die **Zuweisung** (engl. assignment) aus:
  - *Assignment* `;`
- Wir haben somit einmalig eine linke Seite einer Grammatikregel durch eine der möglichen Alternativen ersetzt. Das tun wir nun für das Nichtterminal *Assignment* genauso. Für dieses Nichtterminal haben wir keine Wahl zwischen mehreren Alternativen, so dass wir in unserer bisherigen Ableitung direkt das Nichtterminal durch die eine rechte Seite ersetzen können:
  - *Identifier* `=` *Expression* `;`

Beispiel einer Ableitung (II): `i = j + 3;`

- Als nächsten Schritt leiten wir das Nichtterminal *Identifier* ab. Ein Identifier ist aus Sicht der Grammatik nicht weiter definiert. Für Java gilt, dass ein Identifier u.a. aus beliebig vielen Buchstaben bestehen kann. Also ist auch `i` ein gültiger Identifier und wir ersetzen das Nichtterminal:
  `i =` *Expression* `;`
- Nun folgt das Nichtterminal *Expression* (engl. für **Ausdruck**). Auch hier gibt es in der Grammatik nur eine Alternative:

```
Expression:
        Expression2 { InfixOperator Expression2 }
```

- Also setzen wir sie für das Nichtterminal in unsere Ableitung ein:
  `i =` *Expression2* `{` *InfixOperator Expression2* `}` `;`
- Hier haben wir nun eine spezielle Situation: Die Klammerung mit den Meta-Symbolen besagt, dass das Geklammerte beliebig oft auftreten kann. Mit Blick auf die Anweisung, die wir ableiten wollen, ersetzen wir dieses „beliebig oft“ in diesem konkreten Fall also mit „einmal“:
  `i =` *Expression2 InfixOperator Expression2* `;`

Beispiel einer Ableitung (III): `i = j + 3;`

- Die Reihenfolge, in der wir innerhalb einer Regel Nichtterminale ersetzen, ist nicht relevant. Wir können beispielsweise zuerst *InfixOperator* ersetzen. Eine mögliche Alternative für *InfixOperator* ist ein einfaches Pluszeichen:

  `i =` *Expression2* **`+`** *Expression2* **`;`**
- Für *Expression2* gibt es drei Alternativen. Die beiden ersten kommen nicht in Frage: die erste führt zu zwei Nichtterminalen, in unserer abzuleitenden Anweisung taucht aber links und rechts vom Pluszeichen nur noch ein Terminal auf; die zweite erfordert runde Klammern, die wir in unserer Zielanweisung auch nicht haben. Also bleibt nur die dritte, die wir hier gleich doppelt anwenden:

  `i =` *Primary* **`+`** *Primary* **`;`**
- Eine mögliche Alternative für *Primary* ist ein Identifier – das passt gut zu **`j`**, das auch ein gültiger Identifier ist. Die **`3`** ist ein Literal, also eine Zeichenkette, die direkt einen Wert darstellt. Also zwei Ersetzungen:

  `i =` *Identifier* **`+`** *Literal* **`;`**
- Schließlich ergibt sich somit durch zwei weitere Ersetzungen:

  `i = j + 3;`

Beispiel einer Ableitung: Zusammenfassung

- Wir haben insgesamt **zielgerichtet** eine bestimmte Anweisung aus einem Nichtterminal der Java Level 1-Syntax abgeleitet, indem wir schrittweise Nichtterminale durch eine ihrer möglichen Alternativen ersetzt haben.
- Die Ableitungsschritte sind hier noch einmal zusammengefasst:

  Statement

  Assignment **`;`**

  Identifier **`=`** *Expression* **`;`**

  `i =` *Expression* **`;`**

  `i =` *Expression2* **`{`** *InfixOperator Expression2* **`}`** **`;`**

  `i =` *Expression2 InfixOperator Expression2* **`;`**

  `i =` *Expression2* **`+`** *Expression2* **`;`**

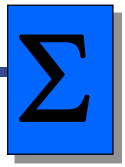
  `i =` *Primary* **`+`** *Primary* **`;`**

  `i =` *Identifier* **`+`** *Literal* **`;`**

  `i = j + 3;`

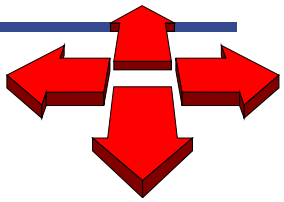
Zielgerichtet heißt: Wir haben das, was wir ableiten wollen (hier: **`i = j + 3 ;`**), ständig vor Augen, und versuchen es mit Hilfe der geeigneten Wahl von Syntaxregeln abzuleiten.

Zusammenfassung



- **Klassendefinitionen** beschreiben Klassen.
- Wir erzeugen Objekte durch **Konstruktoraufrufe**.
- Ein Konstruktor **initialisiert** den Zustand eines Objektes.
- Die (Zustands-)**Felder** eines Objektes halten seinen Zustand; in einer Klassendefinition bezeichnen wir die Definitionen der Felder auch als **Exemplarvariablen**.
- Eine Methode besteht aus einem **Kopf** und einem **Rumpf**.
- Wir unterscheiden **sondierende** (nur lesende) Methoden und **verändernde** Methoden.
- 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.

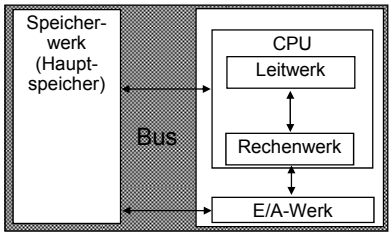
Imperative Grundkonzepte



- Grundlage: von Neumann-Rechner
- Der Prozedurbegriff
- Zuweisungen
- Ausdrücke
- Operatoren
- Basistypen

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

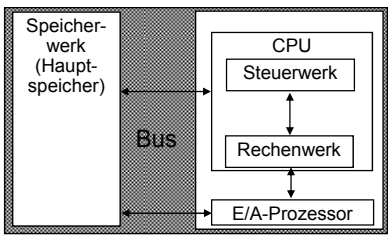
- Der Rechner besteht aus 4 Werken.
- Die Rechnerstruktur ist unabhängig vom bearbeiteten Problem.
- Programme und Daten stehen im selben Speicher.
- Der Hauptspeicher ist in Zellen gleicher Größe unterteilt, die durchgehend adressierbar sind.
- Das Programm besteht aus Folgen von Befehlen, die generell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



SE1 – Level 1 79

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.



SE1 – Level 1 80

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 81

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 82

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 83

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 84

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);
…
```

SE1 – Level 1 85

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**.

SE1 – Level 1 86

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.

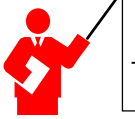
Java kennt ausschließlich Eingabe-Parameter! Wir werden uns Ausgabe-Parameter deshalb frühestens in SE2 näher ansehen.

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!

SE1 – Level 1 89

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.

SE1 – Level 1 90

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.

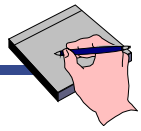
- An der aufrufenden Stelle kann der Name der Ergebnisprozedur **stellvertretend** für das Ergebnis des Aufrufs angesehen werden.

Aktuelle Ergebnisse

Aufrufenden Stellen:

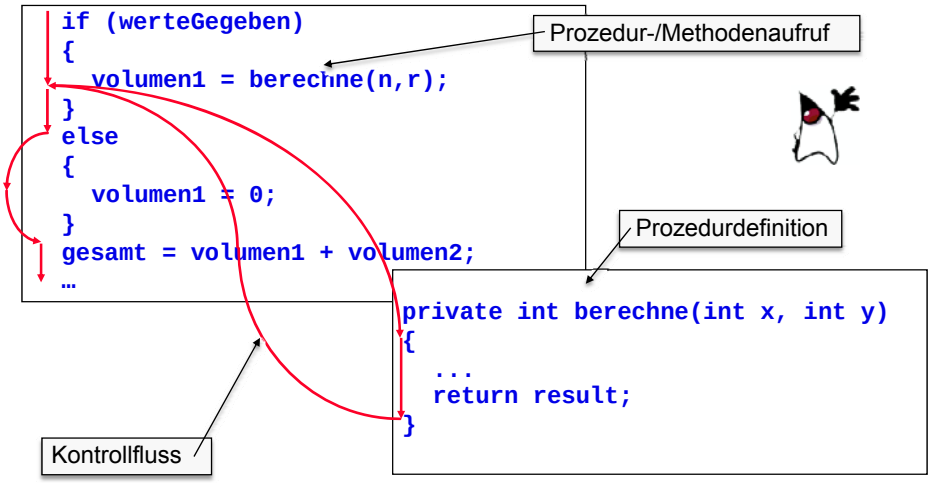
```
...
int ergebnis = maximum(6, 9);
int ergebnis2 = maximum(ergebnis, 2*x);
```


Kontrollfluss bei Prozeduraufrufen



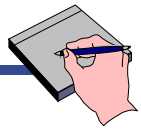
- Der **Prozeduraufruf** ist die explizite Anweisung, dass eine Prozedur ausgeführt werden soll.
- Eine Prozedur ist **aktiv**, nachdem sie gerufen wurde und in der Abarbeitung ihrer Anweisungen noch kein vordefiniertes Ende erreicht hat.
- Für den Prozeduraufruf in **sequenziellen imperativen** Sprachen 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

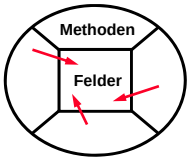


SE1 – Level 1 93

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 94

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.
- **Funktionsprozeduren** 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.

SE1 – Level 1 95

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;
  }
}
```

SE1 – Level 1 96

Deklaration der Variablenarten

- **Vor der Verwendung** einer Variablen in imperativen Programmiersprachen muss sie **deklariert** werden. Dies geschieht durch Angabe des **Typs** und die Vergabe eines **Bezeichners**.
  - **Exemplarvariablen** werden in einer Klassendefinition für alle Exemplare der Klasse deklariert.
  - **Formale Parameter** werden jeweils in den **Köpfen** von Methoden deklariert.
  - **Lokale Variablen** werden in den **Rümpfen** von Methoden deklariert.

SE1 – Level 1 97

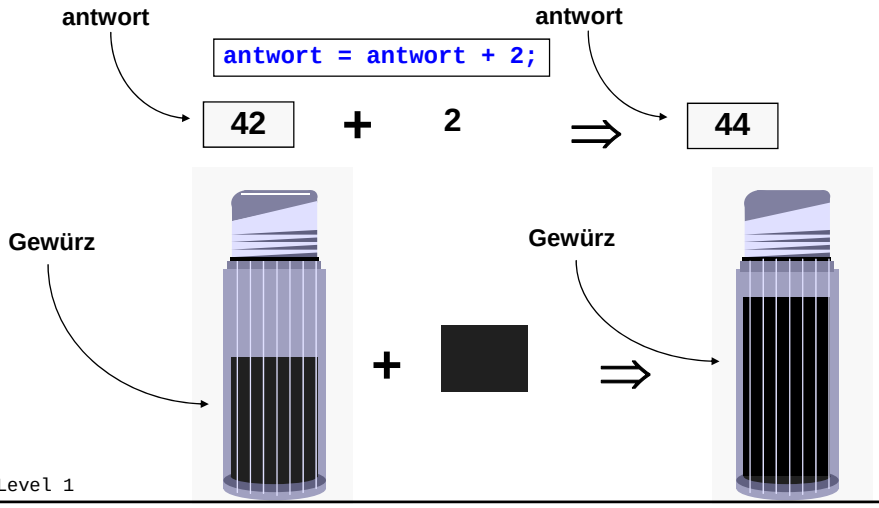
Anfangsbelegung der Variablenarten

- Die **Anfangsbelegung** einer Variablen kann bereits durch ihre Deklaration festgelegt sein:
  - **Exemplarvariablen** werden automatisch mit dem Standardwert (engl.: default value) des jeweiligen Typs belegt.
  - **Formale Parameter** werden mit den Werten der aktuellen Parameter eines Aufrufs belegt.
  - **Lokale Variablen** müssen explizit initialisiert oder erst zugewiesen werden, bevor ihre Belegung ausgelesen werden darf.

SE1 – Level 1 98

Veränderung eines Variablenwerts

- Wichtig ist, dass wir die **Belegung** einer Variablen **verändern** können.
- Diese Aktion, bei der eine Variable unter Beibehaltung ihres Namens und Typs eine (neue) Belegung erhält, heißt **Zuweisung**.



SE1 – Level 1 99

Zuweisung (1)

- Bei der **Zuweisung** wird ein **Ausdruck** ausgewertet und sein **Ergebnis** einer **Variablen** zugewiesen.
- **Syntax-Schema der Zuweisung**:

  `Linke-Seite Zuweisungsoperator Rechte-Seite`

  `Rechte-Seite`:
  - imperativ: meist arithmetische und boolesche Ausdrücke, Vergleiche und Zeichen oder Zeichenketten

  `Zuweisungsoperator`:
  - in Java (wie in C/C++) : `'='`
  - auch üblich (Pascal etc): `':='`

  `Linke-Seite`: Bezeichner einer Variablen

Typkompatibilität:

Der Typ der linken Seite muss zum Typ des Zuweisungsausdrucks passen, d.h. zunächst, die Typen müssen gleich sein.

Die Zuweisung in der Syntax von **Java Level 1**:

```
Assignment:
    Identifier = Expression
```

SE1 – Level 1 100

Zuweisung (2)

- Bei der **Zuweisung** sprechen wir oft von der **rechten** und der **linken Seite** einer Zuweisung *(*engl.: right-hand side - RHS, left-hand side - LHS*)* oder von dem **L-Wert** und **R-Wert** *(*engl.: lvalue, rvalue*)*.
- **Bedeutung**:
  - L-Wert:
    Ist ein Bezeichner einer **Variablen**, der ein Speicherplatz zugeordnet ist. Dort wird der neu berechnete Wert gespeichert.
  - R-Wert:
    Ist ein **Ausdruck**, der einen Wert liefert. Ein R-Wert kann nur rechts vom Zuweisungsoperator stehen.
  - Im folgenden Beispiel haben die beiden Auftreten des Bezeichners **a** unterschiedliche Bedeutung:

    ```
    a = a + (3*i);
    ```

    Auf der linken Seite ist das **a** das Ziel, in dem etwas gespeichert werden soll; auf der rechten Seite ist es die Quelle eines Wertes, der mit anderen Werten in eine Berechnung einfließt.

Merke: Die Zuweisung ist komplizierter, als man auf den ersten Blick vermutet.

Zuweisung in Java

```
antwort = 40
antwort += 2
korrekt = (antwort == 42)
```

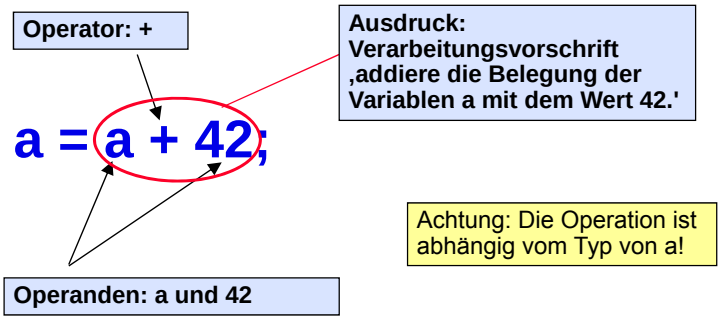
Operator	Funktion
==	Gleichheit
!=	Ungleichheit
=	Zuweisung

- Der Gleichheitstest wird häufig mit der Zuweisung verwechselt:

```
saldo =  0  // Zuweisung
saldo == 0  // Gleichheit
saldo != 0  // Ungleichheit
```

Ausdrücke und Operatoren

- In der imperativen Programmierung hat die Zuweisung einen zentralen Stellenwert. In vielen Fällen wird einer Variablen ein Wert dadurch zugewiesen, dass ein **Ausdruck** aus **Operanden** und **Operatoren** ausgewertet wird.
- Üblich sind arithmetische und boolesche Operatoren; in einigen „maschinennahen“ Sprachen kommen Operatoren zur Manipulation von Zeigern und Bit-Repräsentationen hinzu.



Ausdruck - nach Informatik-Duden

- **Ausdruck** (engl.: expression)
  - Synonym: Term
  - **Verarbeitungsvorschrift**, deren Ausführung einen Wert liefert. Ausdrücke entstehen, indem **Operanden** mit **Operatoren** verknüpft werden. In Programmiersprachen verwendet man häufig arithmetische und logische Ausdrücke.
  - Beispiel:
    Die Symbolfolge **5 * x + 3** ist ein arithmetischer Ausdruck, sofern **x** eine Zahl darstellt.

Operatoren

Als **Operator** bezeichnet man umgangssprachlich sowohl das **Operatorzeichen** (z.B. "+") als auch die damit verbundene **Operation** (z.B. "addieren"). Wir betrachten im folgenden vor allem die Operatorzeichen und ihre Verwendung in (Programm-) Texten.

Die Operatorenschreibweise ist im Zusammenhang mit Programmiersprachen allgemein gebräuchlich (es gibt andere Schreibweisen, z.B. als Funktion).

\+ - * /

SE1 – Level 1 105

Vereinbarungen über Operatoren

Die Operatorenschreibweise ist für uns deshalb einfach lesbar, weil wir bestimmte Vereinbarungen (implizit) kennen, die für die arithmetischen Operatoren gelten.

Bei der Einführung neuer Operatoren müssen diese Vereinbarungen explizit gemacht werden.

Vereinbarungen über Operatoren sind:

- Position,
- Stelligkeit,
- Präzedenz (Vorrangregel),
- Assoziationsreihenfolge.

Dazu kommt die Definition der mit dem Operator verbundenen Operation.

SE1 – Level 1 106

Position von Operatoren

Position, d.h. die Anordnung von Operator und Operanden:

- **Infix**, die häufigste Schreibweise, bei der arithmetische Operatoren zwischen ihren beiden Operanden stehen:
  z.B. : `3 * 4`
- **Präfix**: der Operator steht vor seinen Operanden. Gebräuchlich bei arithmetischen Operationen mit einem Operanden. Diese Form wird auch *Funktionsschreibweise* genannt.
  z.B.: `-2`
- **Postfix**: der Operator steht nach seinen Operanden. Gebräuchlich bei arithmetischen Operationen mit einem Operanden.
  z.B.: `3!` (im Sinne von "Fakultät von 3")

SE1 – Level 1 107

Stelligkeit von Operatoren

Stelligkeit, d.h., Anzahl der Operanden (auch Argumente oder Parameter) eines Operators:

- **einstellig**, oft: unär (engl.: unary)
  z.B. : `3!`
- **zweistellig**, oft: binär (engl.: binary)
  z.B. : `3 * 4`
- **dreistellig**, ternär (engl.: ternary), besser: triadisch.
  In Programmiersprachen kommt meist nur vor
  `if` Operand1 `then` Operand2 `else` Operand3

SE1 – Level 1 108

Präzedenz von Operatoren

Präzedenz (Vorrangregel): bezeichnet die Stärke, mit der ein Operator seine Operanden „bindet“.

- Der Wert eines Ausdrucks hängt oft von der Reihenfolge ab, in der Operatoren eines Ausdrucks angewendet werden.
- Die Präzedenz ist für die arithmetischen Operationen bekannt („Punkt vor Strich“).
- Wenn die Präzedenzen nicht passen, muss geklammert werden:
- Beispiele:

```
3 + 5 * 7 - 3
(3 + 5) * (7 - 3)
```

SE1 – Level 1 109

Assoziativität von Operatoren

Die **Assoziativität** regelt die implizite Klammerung von Ausdrücken bei Operatoren gleicher Präzedenz.

- Beispiel:

```
5 - 4 - 3
```

  ist gleichbedeutend mit

```
(5 - 4) - 3
```

  Sprechweise:

  Der Operator **–** ist **linksassoziativ**
  (er assoziiert von links nach rechts).

- Die Assoziationsreihenfolge ist uninteressant, wenn die Reihenfolge nichts am Wert des Ausdrucks verändert; z.B.:

```
(3 + 4) + 5
```

  ist synomym zu

```
3 + (4 + 5)
```

SE1 – Level 1 110

Zentrale Operatoren in Java

Operator	Funktion, arithmetisch
*	Multiplikation
/	Division
%	Modulo
+	Addition
-	Subtraktion
++	Inkrement
--	Dekrement

Sowohl prä- als auch postfix verwendbar

Operator	Funktion, boolesche
!	logisches NICHT
&&	logisches UND
||	logisches ODER
<	„kleiner als“
<=	„kleiner gleich“
>	„größer als“
>=	„größer gleich“
==	Gleichheit
!=	Ungleichheit

Operator	Funktion
=	Zuweisung

SE1 – Level 1 111

Alle Operatoren in Java: Präzedenz, Assoziativität et al.

Postfix-Operatoren:

1 **( ) . []** ++ -- links nach rechts

Unäre Operatoren (präfix):

2 ++ --

! **-** ~ + *rechts nach links*

3 **new** **(***Typname***)** *rechts nach links*

Binäre* Operatoren (infix):

4 ***** **/** **%** links nach rechts

5 **+** **-** links nach rechts

6 << >> links nach rechts

7 **<** **<=** **>** **>=** links nach rechts

8 **==** **!=** links nach rechts

9 & links nach rechts

10 ^ links nach rechts

11 | links nach rechts

12 **&&** links nach rechts

13 **||** links nach rechts

14 ? : *rechts nach links*

15 **=** += -= *= /=

%= &= ^= |=

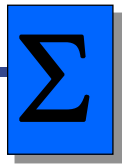
<<= >>= *rechts nach links*

*Ausnahme: ternärer Op.

fett blau hervorgehoben: Relevante Operatoren für Level 1

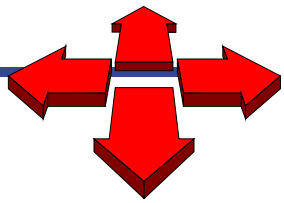
SE1 – Level 1 112

Zwischenergebnis: Zuweisungen et al.



- Die **Anweisungen** in den Rümpfen von Methoden folgen den Prinzipien der **imperativen Programmierung**:
  - Sie werden sequenziell nach der textuellen Reihenfolge im Quelltext ausgeführt.
  - Sie verändern üblicherweise die Belegungen von Variablen.
- Variablen werden durch **Zuweisungen** verändert; wir unterscheiden **Exemplarvariablen**, **formale Parameter** und **lokale Variablen**.
- Auf der **linken Seite** des **Zuweisungsoperators** steht immer eine Variable, auf der **rechten Seite** immer ein **Ausdruck**.
- Arithmetische und boolesche Ausdrücke setzen sich aus **Operanden** und **Operatoren** zusammen.

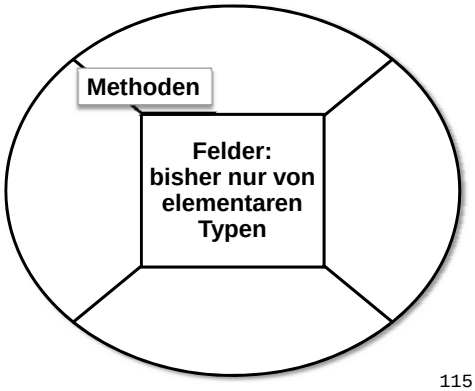
Elementare Typen als Grundbausteine



- Allgemeine Eigenschaften elementarer Typen
- Zahlendarstellung in Rechnern
- Elementare Typen in Java:
  - Ganze Zahlen
  - Wahrheitswerte
  - Zeichen
  - Gleitkommazahlen
- Typumwandlungen

Unsere bisherige Sicht auf Objekte

- **Objekte** sind Exemplare von **Klassen**; diese legen in ihren **Klassendefinitionen** die prinzipiellen Eigenschaften (**Verhalten** und mögliche **Zustände**) ihrer Exemplare fest.
- In einer Klassendefinition wird das Verhalten der Exemplare über die **Methoden** definiert, die möglichen Zustände über die **Zustandsfelder**.
- Als Typen für die **Felder** haben wir bisher nur vordefinierte Typen von Java gewählt, beispielsweise `int` und `boolean`. Die entstehenden Objekte waren deshalb weitgehend voneinander unabhängig (sie waren in ihrem Zustand nicht von anderen Objekten abhängig).
- Im Folgenden wollen wir uns die **elementaren Typen** genauer ansehen, mit denen wir die möglichen Zustände unserer Objekte definiert haben.



SE1 – Level 1 115

Erster Kontakt mit dem Typbegriff

- Der Typbegriff spielt eine sehr wichtige Rolle in der Programmierung.
- Ein **Typ** in einer Programmiersprache legt fest
  - eine **Wertemenge** (z.b. bei `int` nur eine Untermenge der ganzen Zahlen) und
  - die **zulässigen Operationen** (z.B. Addieren, Subtrahieren) auf den Werten der Wertemenge.

Typ: `int`
Wertemenge: { -2^{31} … $2^{31}-1$ }
Operationen: `Addieren`, `Subtrahieren`, `Multiplizieren`, …

SE1 – Level 1 116

Elementare Typen

- Imperative und objektorientierte Programmiersprachen bieten i.d.R. einen Satz **elementarer Typen** (engl.: *basic* or *primitive data types*) an:
  - für **ganze Zahlen** → Typ `Integer` o.ä.
  - für **reelle Zahlen** → Typ `Float` oder `Real` (Gleitkommazahlen)
  - für **Zeichen** → Typ `Char` o.ä. (Werte eines bestimmten **Zeichensatzes**)
  - für **Wahrheitswerte** → Typ `Boolean`.
- Die **Werte** dieser elementaren Typen können explizit im Quelltext hingeschrieben werden; jede Programmiersprache bietet zu diesem Zweck sog. **Literale** an. Ein Literal ist eine Zeichenfolge (wie `13` oder `"gelb"`) im Quelltext, die einen Wert eindeutig repräsentiert und deren Struktur dem Compiler bekannt ist.

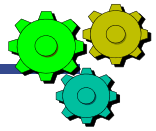
SE1 – Level 1 117

Zahlen und ihre Darstellung im Rechner

- Da es unendlich viele Zahlen gibt, aber der Rechner nur eine **begrenzte Kapazität und Wortlänge** hat, werden Zahlen auf einem Rechner meist nur begrenzt dargestellt.
- So gibt es z.B. eine größte und eine kleinste darstellbare ganze Zahl (bspw. `Maxint` und `Minint` genannt) und die Gleitkommazahlen haben eine **begrenzte Genauigkeit**.
- Während Operationen auf ganzen Zahlen, die wieder ganze Zahlen als Ergebnis liefern, **exakt** sind (soweit die Zahlen innerhalb der obigen Grenzen bleiben), gilt dies **nicht für die Gleitkommaarithmetik**.
- So ist z.B. nicht sichergestellt, dass der Ausdruck `(x+y)-y` als Ergebnis exakt den Wert `x` hat.
- Dies ist ein **fundamentales Problem der digitalen Datenverarbeitung**.

SE1 – Level 1 118

Gleitkommazahlen in Rechnern



- Prinzipiell gilt für SoftwaretechnikerInnen:
  Gleitkommazahlen sollten wegen ihrer potenziellen Ungenauigkeit möglichst vermieden werden!
- **Donald Knuth** beispielsweise hat das Satzprogramm TEX ausschließlich auf Basis von Festkommazahlen entwickelt!
- Wer dennoch meint, Gleitkommazahlen verwenden zu müssen, sollte sich gründlich mit dem Thema beschäftigen:
  - Goldberg, D.: "**What every computer scientist should know about floating-point arithmetic**", *ACM Computing Surveys,* 23:1, S. 5-48, 1991.
  - frei unter: http://docs.sun.com/source/806-3568/ncg_goldberg.html

Elementare Datentypen in Java

- Auch Java besitzt den üblichen Satz an elementaren Datentypen, die **primitive types** genannt werden. Ungewöhnlich ist die Festlegung der Wortlängen bei den numerischen Typen (jeweils in Klammern in Bit angegeben).
  - **Eine ganze Familie für ganze Zahlen**:
    - » `byte` (8), `short` (16), `int` (32), `long` (64)
  - **Zwei für Gleitkommazahlen**:
    - » `float` (32), `double` (64)
  - **Ein boolescher Datentyp**:
    - » `boolean`
  - **Ein Datentyp für Zeichen**:
    - » `char` (16), **0** bis **65535**

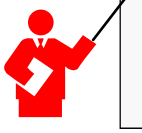
Datentyp	Bit	kleinster Wert	größter Wert
`long`	64	-2^{63}	$2^{63}-1$
`int`	32	-2^{31}	$2^{31}-1$
`short`	16	-32768 (-2^{15})	32767 ($2^{15}-1$)
`byte`	8	-128 (-2^{7})	127 ($2^{7}-1$)

Typ	Standardwert
byte, short int, long (ganze Zahlen)	0 bzw. 0L
boolean (Wahrheitswerte)	false
double, float (Gleitkommazahlen)	0.0 bzw. 0.0f
char (Zeichen)	'\u0000'

Wir geben auf den folgenden Folien nur einen Überblick.
Es ist empfehlenswert, in einem Java-Referenzbuch bei Bedarf weitere Details nachzulesen!

Literale für Zahlen in Java

- **Darstellung**:
  - » **Ganze Zahlen** (engl.: integer numbers) können wie gewohnt notiert werden, also etwa `542` oder `-1`; solche Literale werden dann als Dezimalzahlen vom Typ `int` aufgefasst, in diesem Fall mit den Werten `542` und `1` (das `-` ist ein Präfix-Operator, der die `1` hier negiert).
  - » **Gleitkommazahlen** (engl.: floating point numbers) werden in englischer Dezimalnotation mit einem Punkt notiert, z.B. `0.5`, oder mit einem expliziten Exponenten, z.B. `5e-1` (Wert in beiden Fällen `0,5`). Wenn kein `f` für `float` angehängt ist (wie beispielsweise bei `3.1415f`), wird für diese Literale der Typ `double` angenommen.



- Alternativ können ganze Zahlen auch **oktal** (beginnend mit einer `0`) oder **hexadezimal** (beginnend mit `0x`) angegeben werden.
  Bsp.: `29` u. `035` u. `0x1D` u. `0X1d` sind alternative Literale für die Dezimalzahl `29`.

Binäre Operatoren für ganze Zahlen in Java

Arithmetische Operatoren mit Ergebnistyp `int`

- Java bietet die vier Grundrechenarten über die Infix-Operatoren `+`, `-`, `*` und `/` an. Dabei ist zu beachten, dass der Divisionsoperator eine **ganzzahlige Division** durchführt:

  `20/6` ⇒ `3`

  Das Ergebnis dieser Operation ist also wieder ein int-Wert.
- Zusätzlich gibt es den Operator `%`, der bei einer ganzzahligen Division den Rest liefert:

  `20%6` ⇒ `2`
- Die Präzedenz dieser fünf Operatoren entspricht unseren Erwartungen aus der Mathematik („Punktrechnung vor Strichrechnung“):

  `3 + 2 * 2 + 5` ⇒ `12`

Binäre Operatoren für ganze Zahlen in Java (II)

Vergleichsoperatoren mit Ergebnistyp `boolean`

- Für Vergleiche ganzer Zahlen stehen Infix-Operatoren für die Operationen *Größer* (`>`), *Größer-gleich* (`>=`), *Kleiner* (`<`) und *Kleiner-gleich* (`<=`) zur Verfügung.

```
2 > 1 + 1          ⇒  falsch
2 >= 3 - 1         ⇒  wahr
2 * 2 < 1 * 1      ⇒  falsch
2 <= 2 / 1         ⇒  wahr
```

- Zwei weitere Infix-Operatoren erlauben die Abfrage, ob zwei `int`-Ausdrücke gleich (`==`) oder ungleich (`!=`) sind:

```
3 == 2 + 1         ⇒  wahr
4 != 2 * 2         ⇒  falsch
```

- Vergleichoperatoren haben gegenüber den arithmetischen Operatoren eine niedrigere Präzedenz; sie werden also in einem Ausdruck zuletzt ausgewertet.

Boolesche Literale und Operatoren

- **Boolesche Werte** oder **Wahrheitswerte** sind in Java vom primitiven Typ `boolean`.
- Die **Literale** für die booleschen Werte **wahr** und **falsch** werden als `true` und `false` notiert.
- Die Standardoperatoren der **booleschen Algebra** werden in Java folgendermaßen notiert:

  - logisches Und: `&&`
  - logisches Oder: `||`
  - logische Verneinung: `!`
- Die Java-Operatoren für Gleichheit (`==`) und Ungleichheit (`!=`) sind auch auf boolesche Werte anwendbar. Beispiele:

```
true == false      ⇒  falsch
true != false      ⇒  wahr
```


Boolesche Operationen: Wahrheitstafeln

Negation:

not	
false	true
true	false

Konjunktion:

and	false	true
false	false	false
true	false	true

Disjunktion:

or	false	true
false	false	true
true	true	true

Beispiele:

not not not true
not (false or true)
(true and false) and true
not (27 < 12)
3 < 6 = 7 > 5
false or (false = true) or 5 > 7

Boolesche Operationen: Einige Rechenregeln

Seien **P**, **Q** und **R** logische Variable, dann gelten die folgenden Identitäten:

Kommutativgesetze:
P or Q ≡ Q or P
P and Q ≡ Q and P

Distributivgesetze:
(P and Q) or R ≡ (P or R) and (Q or R)
(P or Q) and R ≡ (P and R) or (Q and R)

Assoziativgesetze:
(P or Q) or R ≡ P or (Q or R)
(P and Q) and R ≡ P and (Q and R)

De Morgans Gesetze:
not (P or Q) ≡ not P and not Q
not (P and Q) ≡ not P or not Q

Grundannahme: Der Operator **not** bindet stärker als die Operatoren **and** und **or**.

Zeichen und ihre Darstellung

- **Zeichen** werden im Rechner durch vordefinierte Werte (die sog. Codes) eines **Zeichensatzes** repräsentiert.
- In Java können wir einzelne Zeichen im Quelltext als Literale in Hochkommata notieren: `'*'`, `'0'`, `'A'`, `'z'`
- Merke: Das Literal `'4'` ist etwas anderes als das Literal `4`. Sie haben verschiedene Typen, im Fall von Java `char` und `int`.

```
char c = '4';
int i = 4;
```

Rückblick: Der ASCII-Zeichensatz

- Die meisten Programmiersprachen vor Java haben Zeichen durch die 128 vordefinierten Werte des sog. **ASCII-Zeichensatzes** dargestellt.
- **ASCII** (Akronym für **American Standard Code for Information Interchange**) ist laut Informatik-Duden:
  - Ein weit verbreiteter, besonders auf Heimcomputern üblicher 7-Bit-Code zur Darstellung von Ziffern, Buchstaben und Sonderzeichen.
  - Jeder ASCII-Codezahl zwischen 0 und 127 entspricht ein Zeichen. Beispiele:
    - ASCII 42 ⇒ *
    - ASCII 48 ⇒ 0
    - ASCII 65 ⇒ A
    - ASCII 122 ⇒ z

in gelb: die druckbaren ASCII-Zeichen

ASCII-Zeichensatz

+	0	1	2	3	4	5	6	7	8	9
30				!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~			

Java und der Unicode-Zeichensatz

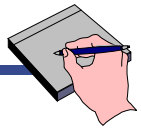
- Java war die erste weit verbreitete Programmiersprache, die vollständig auf dem **Unicode Standard UTF-16** aufsetzte, der für jedes Zeichen **16 Bit** verwendet (und somit 65.536 verschiedene Zeichen ermöglicht). Damit lassen sich die Zeichen und Zahlen der meisten bekannten Kultursprachen darstellen.
- Die ersten 128 Zeichen entsprechen dem ASCII-Zeichensatz.
- Inzwischen erlaubt der Unicode-Standard eine Kodierung in bis zu **32 Bit**. Vier Milliarden Zeichen sollten dann für alle irdischen Zwecke ausreichen…
- Zwei Drittel des 16-Bit Unicode-Zeichensatzes werden für chinesische Schriftzeichen verwendet.
- Informationen zu Unicode finden sich im Web unter **http://unicode.org**

In Java kann ein Unicode-Zeichen mit einer speziellen Schreibweise notiert werden: `'\uXXXX'`.

XXXX ist dabei der vierstellige, hexadezimale Unicode des Zeichens, eventuell mit führenden Nullen.

`'a'` beispielsweise bezeichnet wie `'\u0061'` das a.

Gleitkommazahlen in Java (I)



- Java definiert die primitiven Typen `float` und `double` für Gleitkommazahlen nach dem **IEEE Standard 754**. Dieser Standard legt einfache Genauigkeit mit 32 Bit fest (engl.: single precision, in Java durch `float` umgesetzt) und doppelte Genauigkeit mit 64 Bit (engl.: double precision, in Java durch `double` umgesetzt).
- IEEE 754 definiert **Summen von Zweierpotenzen** und nicht, wie wir es aus dem Alltag gewohnt sind, Summen von Zehnerpotenzen. Durch diesen Bruch zum Dezimalsystem können auch solche Werte nicht exakt dargestellt werden, von denen wir es intuitiv erwarten würden, z.B. der relativ glatte Dezimalbruch **0,1**. Als Dualbruch dargestellt sieht er so aus:

$$0{,}1_{10} = 0{,}0001100110011001100110011001100\ldots_2 = 0{,}0\overline{0011}_2$$

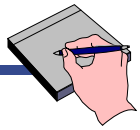
- Diese unendliche Periode lässt sich in einer begrenzten Anzahl von Bits nicht erfassen.

Gleitkommazahlen in Java (II)

- Beim Umgang mit Gleitkommazahlen muss aufgrund dieser Eigenschaften besonders auf Wertebereich und **Genauigkeitsgrenzen** geachtet werden.
- Wie bei den Typen für ganze Zahlen stehen in Java auch für `float` und `double` die vier Grundrechenarten über Infix-Operatoren zur Verfügung, ebenso wie die Vergleichsoperatoren.

Gleitkommazahlen sind ein umfangreiches Thema. Unter
http://tams-www.informatik.uni-hamburg.de/lectures/2009ws/vorlesung/rechnerstrukturen/
gibt es mehr Informationen zu IEEE 754, aber auch zum Zweierkomplement (`int` etc.).

Typumwandlungen



- Typprüfungen bewahren uns vor Fehlern. Die Zuweisung

  ```
  int i = true;
  ```
  ⇒ **Typfehler!**

  beispielsweise führt bei der Übersetzung zu einer Fehlermeldung, weil der Ausdruck rechts vom Typ `boolean` ist, auf der linken Seite der Zuweisung aber eine `int`-Variable steht.
- Andererseits erwarten wir, dass die Zuweisung

  ```
  double d = 5;
  ```

  funktioniert, obwohl auf der rechten Seite ein `int`-Ausdruck steht und auf der linken Seite eine `double`-Variable.
- Die Lösung sind so genannte **Typumwandlungen** (engl.: type conversion oder type cast), die in Programmiersprachen **implizit** (automatisch) oder **explizit** (durch den Programmierer) durchgeführt werden.
- Eine Typumwandlung bewirkt zur Laufzeit eine Umwandlung einzelner Bits. Die Art der Umwandlung hängt dabei von Ausgangstyp und Zieltyp ab.

Automatische Typumwandlungen in Java

- **Zieltyp hat höhere Genauigkeit als Ausgangstyp**
  Umwandlung kann automatisch vorgenommen werden (engl.: coercion), weil keine Genauigkeit verloren gehen kann (engl. auch: widening conversion). Die Zuweisung

  ```
  double d = 5;
  ```

  ist deshalb in Java zulässig, weil sich alle `int`-Wert auch als `double`-Werte darstellen lassen. Das Bitmuster für den Wert `5` im Zweierkomplement wird bei der Ausführung in die Gleitkomma-Darstellung nach IEEE 754 umgewandelt.

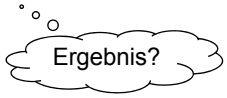
  Ein weiteres Beispiel:

  ```
  int i = 'a';
  ```

  Automatische Umwandlungen können auch mehrfach innerhalb eines Ausdrucks auftreten:

  ```
  double d = 3 + '4' - 3.1415f;
  ```

Explizite Typumwandlungen in Java

- **Zieltyp hat niedrigere Genauigkeit als Ausgangstyp**
  Weil bei der Umwandlung Genauigkeit verloren gehen kann (engl.: narrowing conversion), muss der Programmierer die Umwandlung explizit erzwingen (und wissen, was er tut).
  Die Zuweisung

  ```
  int i = 3.1415;
  ```

  ⇒ Fehlermeldung: possible loss of precision!

  ist in Java nicht zulässig, weil die Genauigkeit des Gleitkomma-Ausdrucks bei der Zuweisung an eine `int`-Variable verloren gehen kann.

  Wenn wir diesen Verlust bewusst in Kauf nehmen wollen, schreiben wir vor den Ausdruck **in runden Klammern** explizit den Zieltyp der Umwandlung:

  ```
  int i = (int)3.1415;
  ```

  Dies bewirkt eine Umwandlung in die ganze Zahl `3` vom Typ `int`.

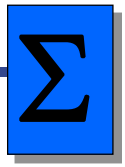
  Ein häufig gemachter Fehler in Java in diesem Zusammenhang:

  ```
  float f = 3.1415;
  ```

  ⇒ **???**

Zusammenfassung elementare Typen



- Programmiersprachen bieten üblicherweise einen Satz an **elementaren Typen**.
- Die Werte elementarer Typen werden im Quelltext mit **Literalen** benannt.
- **Java** verfügt über so genannte **primitive Typen** für
  - ganze Zahlen
  - Wahrheitswerte
  - Zeichen
  - Gleitkommazahlen
- Java basiert auf dem **Unicode-Zeichensatz**.
- Zwischen den primitiven Typen können explizite und implizite **Typumwandlungen** stattfinden.

