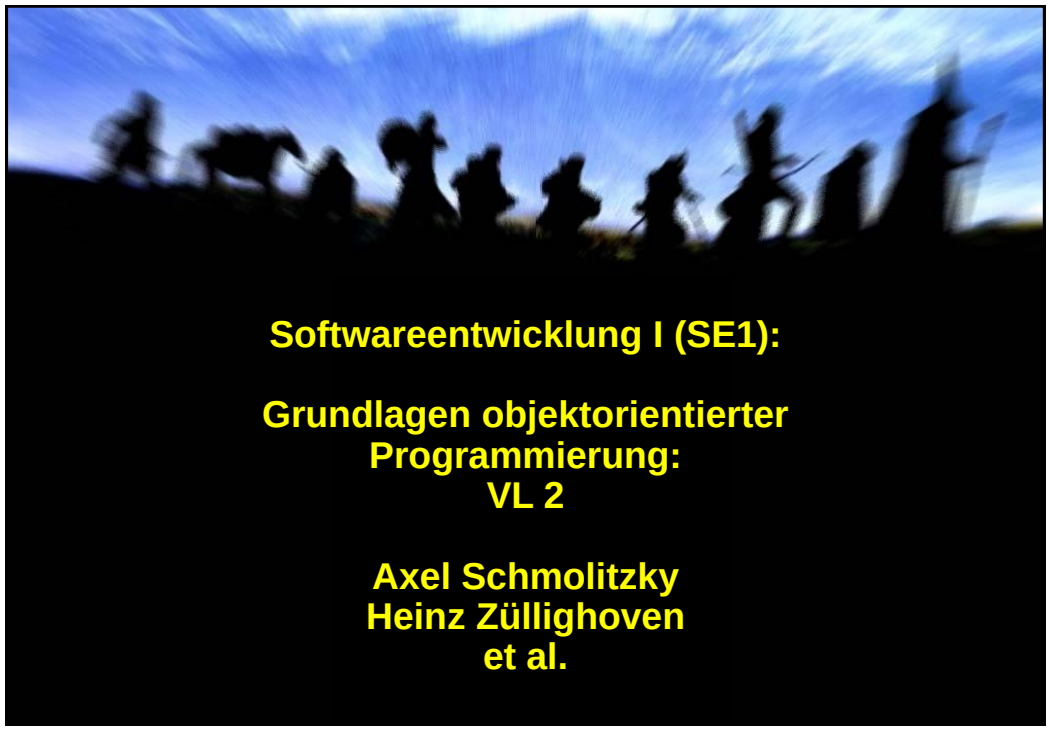


Softwareentwicklung I (SE1):

Grundlagen objektorientierter Programmierung: VL 2

Axel Schmolitzky
Heinz Züllighoven
et al.

CV AS

Informatikstudium Uni Bremen
Diplomabschluss 1993

Wissenschaftlicher Mitarbeiter Uni Ulm
Dissertation 1999

DAAD Postdoc-Stipendiat, 1999-2000
Monash University, Melbourne, Australien
Mitarbeit im BlueJ-Team

Wissenschaftlicher Assistent Uni Hamburg
2001 bis 2009, teilweise in Halbzeit

Senior-Entwickler/Software-Architekt bei der C1 WPS GmbH
Schulungen, Coaching, Beratung, Entwicklung
seit Anfang 2003 nur noch freiberuflich

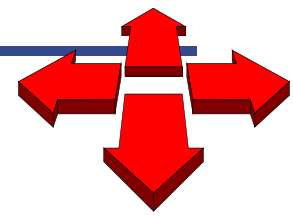
Akademischer Rat Uni Hamburg
seit 1.6.2009

CV AS (II)

Forschungsinteressen

- Design objektorientierter Programmiersprachen
- Agile Entwicklungsmethoden
- Software-Architektur (Schwerpunkt Ausbildung)
- Neue Lehrkonzepte in der Softwaretechnik
  - „Objects First“ (u.a. deutsche Übersetzung Barnes/Kölling)
  - Betreuter Laborbetrieb
  - Teachlets
  - Komplexitätsstufen von Objektsystemen in der Lehre

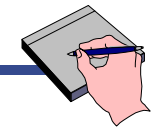
Objektorientierte Grundlagen

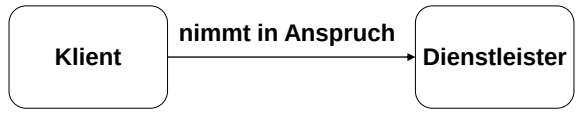


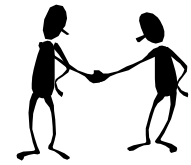
- Objekte: Klienten und Dienstleister
- Objekte zeigen Verhalten und haben Zustände
- Methoden und Zustandsfelder
- Klassen als Blaupausen für Exemplare
- Methoden bestehen aus imperativen Anweisungen
- Imperative Grundkonzepte in der Übersicht

- Logischer Aufbau von Klassendefinitionen
  - Konstruktoren
  - Zustandsfelder
  - Methoden

Immer wieder wichtig: Dienstleister und Klienten



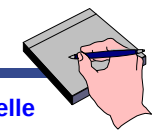




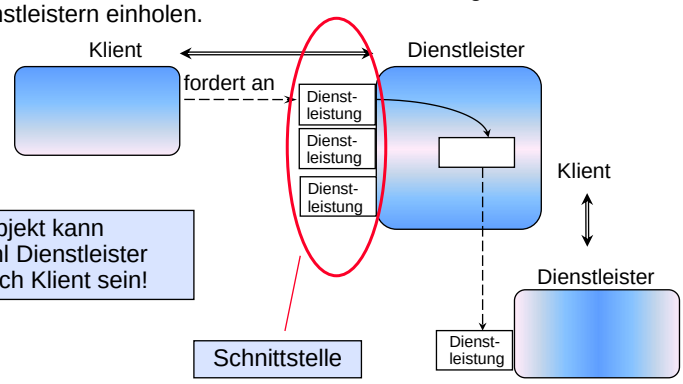
- Das Objekt, das bei einer bestimmten (Teil-)Aufgabe einen Dienst leistet, ist der **Dienstleister**.
- Das Objekt, das eine konkrete Dienstleistung eines anderen Objektes in Anspruch nimmt, wird als **Klient** bezeichnet.

SE1 – Level 1 5

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.



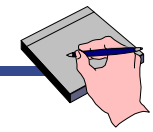
SE1 – Level 1 6

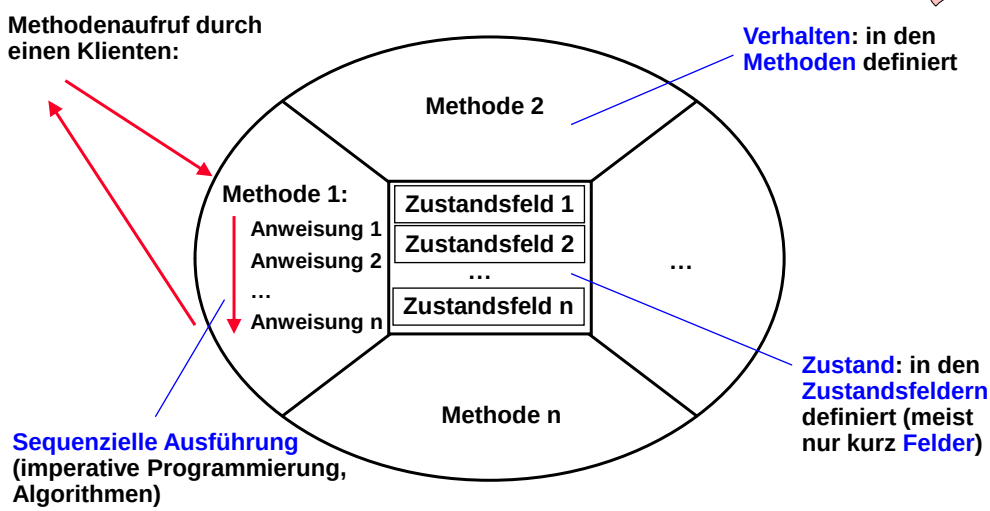
Dienstleistungen: Verhalten und Zustand

- 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.

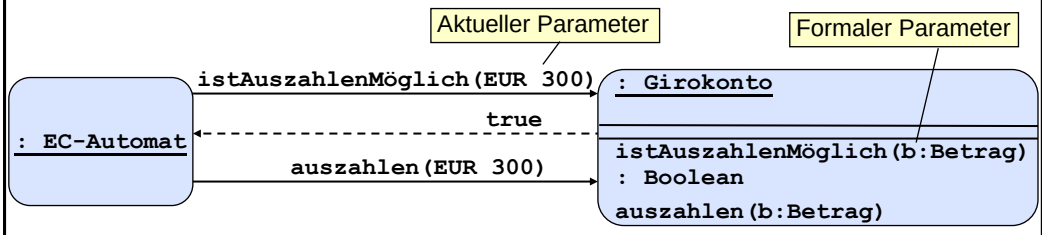
Bsp.: Je nach Zustand eines Kontos ist das Auszahlen mal möglich und mal nicht…

Logische Sicht auf ein Objekt





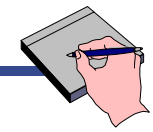
Objekte interagieren über Methodenaufrufe



- 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.

SE1 – Level 1 9

Signatur einer Methode



- Die **Signatur** einer Methode liefert die **für einen Klienten** 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:

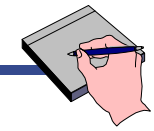
```
boolean istAuszahlenMöglich(Betrag b)
```

Signatur

istAuszahlenMöglich(Betrag)

SE1 – Level 1 10

Klassen als Schablonen für Exemplare

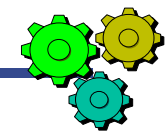


Girokonto
_dispo : Betrag _saldo : Betrag
istAuszahlenMöglich(b:Betrag) : Boolean auszahlen(b:Betrag)

- 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.

SE1 – Level 1 11

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.

SE1 – Level 1 12

Programmieren im Kleinen: ein imperativer Algorithmus

Beispiel:

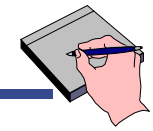
```
(1) Vergleiche zwei natürliche Zahlen a und b.
(2) Wenn a größer als b ist, setze das Ergebnis
    max gleich dem Wert von a.
(3) Sonst setze das Ergebnis max gleich dem Wert
    von b.
```

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).

SE1 – Level 1 13

Imperative Programmierung



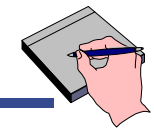
Das Paradigma **imperative Programmierung**:

- 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.

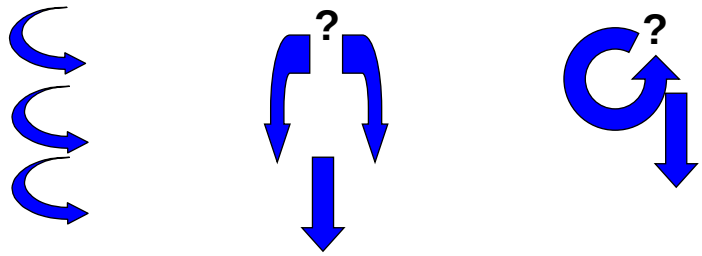
- Imperative Programmierung baut auf dem Konzept des *v. Neumann-Rechners* auf.
- Sie beruht auf dem *Zustandskonzept*.

SE1 – Level 1 14

Ablaufsteuerung durch Kontrollstrukturen

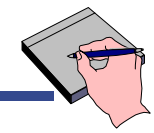


- Ablaufsteuerung in **Programmiersprachen** durch **Kontrollstrukturen**:
  - Die Ausführungsreihenfolge der Anweisungen einer Methode entspricht zunächst der textlichen Anordnung (**Sequenz**). Davon kann aber abgewichen werden. Dazu gibt es spezielle Mechanismen der Ablaufsteuerung:
    - **Fallunterscheidung**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



SE1 – Level 1 15

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;

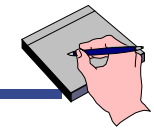
wähle die Telefonnummer;

führe das Gespräch;

lege den Hörer auf.

SE1 – Level 1 16

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`** … .

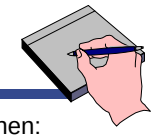
Aus dem Algorithmus **Telefonieren**:

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

Aus dem Algorithmus **Telefonieren**:

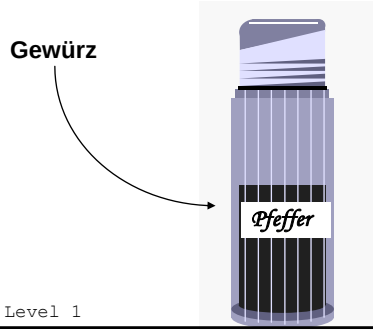
```
hole Liste der Gesprächspartner
WIEDERHOLE
      führe ein Gespräch;
      streiche Gesprächspartner;
BIS Liste abgehakt .
```

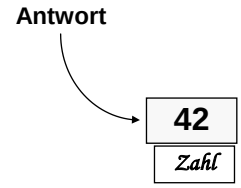

Imperative Variablen



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.

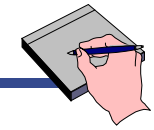




Die Typen sind hier *Pfeffer* und *Zahl*.

SE1 – Level 1 19

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;
```

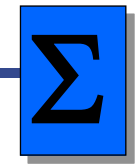
Deklaration und Initialisierung

```
int i = 42;
boolean b;

b = true;
```

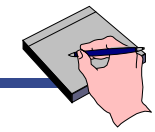
SE1 – Level 1 20

Zwischenfazit



- 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.

Unsere erste selbst geschriebene Klassendefinition

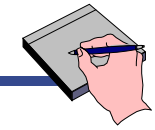


```
class Girokonto
{
  private int _saldo;

  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 23

Abgleich mit den Prinzipien der Objektorientierung

```
class Girokonto
{
  private int _saldo;

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

- 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 24

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

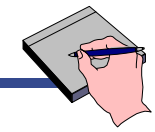
Objekterzeugung:
Objekte müssen zur Laufzeit durch einen expliziten Ausdruck erzeugt werden. Dazu wird ein eigenes **Schlüsselwort** (in Java `new`) verwendet.

```
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 27

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);
  ...
}
```

1

: Quadrat

2

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

SE1 – Level 1 28

Methoden aufrufen

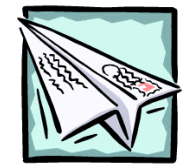
- Jeder **Methodenaufruf** richtet sich immer an ein bestimmtes Objekt, den **Adressaten** des Aufrufs.
- Der Adressat ist entweder explizit angegeben:

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

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

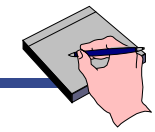
```
zeichneDach(80);
```

  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

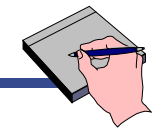
der Punkt

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 31

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.

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

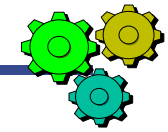
Hier ist in Java die **Signatur** enthalten.

Kopf der Methode

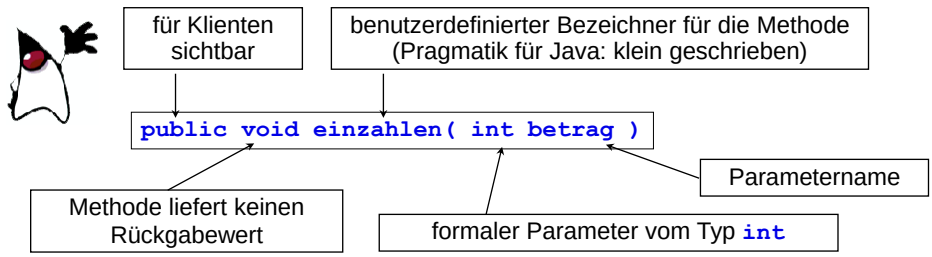
Rumpf der Methode

SE1 – Level 1 32

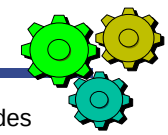
Verändernde Methoden



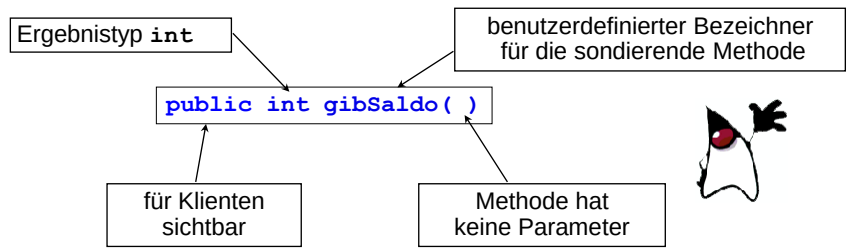
- Pragmatik: Wenn Methoden den Zustand ihres Objektes verändern (**verändernde Methoden**, engl.: mutators), dann sollten sie keinen Wert zurück geben.
- Für Klienten sind nur die Methoden aufrufbar, die mit `public` als „öffentlich“ deklariert wurden; sie bilden die Schnittstelle einer Klasse.
- Neben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



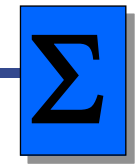
Sondierende Methoden



- **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.



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.

