

5. Klassen

- Klassendefinition
- Klasse als Typ
- Klasse als Objektfabrik
- Konstruktoren
- Objektmethoden
- Statische Felder und Methoden



Motivation

- In vielen Anwendungen besitzen reale Objekte unterschiedliche Eigenschaften:
 - Studierende besitzen einen Vornamen, Nachnamen, Matrikelnummer, Fachbereich und das Fachsemester.
 - Jeder dieser Eigenschaften kann ein Datentyp zugeordnet werden.
 - Nachname ist vom Typ String
 - Fachsemester vom Typ int
 - Ein Bankkonto hat eine Kontonummer, aktuellen Kontostand und eine Kundennummer.
 - Eine Kontonummer ist vom Typ String.
 - Der aktuelle Kontostand vom Typ double.
- Wünschenswert wäre, wenn wir alle Daten eines Studierenden bzw. eines Bankkontos über eine Variable ansprechen könnten.
 - Was ist der Typ einer solchen Variable

Max, Mustermann, 12345, 12, 1

54321, 2500.23, 77

Motivation

- In vielen Anwendungen besitzen reale Objekte unterschiedliche Eigenschaften:

- Studierende besitzen einen Vornamen, Nachnamen, Matrikelnummer, Fachbereich und das Fachsemester.

- Jeder dieser Eigenschaften kann ein Datentyp zugeordnet werden.

- Nachname ist vom Typ String

- Fachsemester vom Typ int

Max, Mustermann, 12345, 12, 1

- Ein Bankkonto hat eine Kontonummer, aktuellen Kontostand und eine Kundennummer.

- Eine Kontonummer ist vom Typ String.

- Der aktuelle Kontostand vom Typ double.

54321, 2500.23, 77

- Wünschenswert wäre, wenn wir für ein Student-Objekt bzw. eines Bankkontos überlegen, welche Datentypen wir verwenden.

- Was ist der Typ einer solchen Variable?

Achtung: wegen möglicher Rundungsfehler lassen sich nicht alle Geldbeträge als double darstellen. Mit Klassen lassen sich hier zutreffendere Datentypen erstellen.

Datentypen und Operationen

- Bestandteil von Datentypen
 - Wertemenge
 - Menge von erlaubten Operationen.
- Beispiel Bankkonto
 - Wertemenge
 - Das Kreuzprodukt $\text{String} \times \text{int} \times \text{double}$ repräsentiert die Eigenschaften eines Kontos: Kontonummer, Kundennummer und Kontostand.
 - Operationen
 - Geld abheben $\text{Bankkonto} \times \text{double} \rightarrow \text{double}$
 - Geld einzahlen $\text{Bankkonto} \times \text{double} \rightarrow$
 - Kontostand abfragen $\text{Bankkonto} \rightarrow \text{double}$
- Studierende haben andere Eigenschaften und benötigen andere Operationen.
 - Prüfe, ob ein Studierender an einem gegebenen Fachbereich eingeschrieben ist. $\text{Student} \times \text{String} \rightarrow \text{boolean}$

5.1 Klassen als eigene Datentypen

- Mit Hilfe von sogenannten **Klassen** können in objektorientierten Sprachen (wie Java) sehr flexibel **eigene Datentypen** definiert werden.
 - Datenelemente unterschiedlicher Typen werden dabei in einem neuen Typ zusammengefasst.
 - Die zu dem neuen Datentyp zugeordneten Operationen werden durch Angabe von Methoden zur Verfügung gestellt.
- Verwendung von Klassen
 - Wie bei Arrays können **Variablen mit Klassen-Typen** deklariert werden.
 - Wie bei Arrays muss der Speicherplatz für eine Instanz des neuen Datentyps **explizit reserviert** werden.
 - Reservierung geschieht ebenfalls auf dem Heap
 - Statt Instanz verwenden wir den Begriff **Objekt**.

Beispiel - Konto

```
class Konto{
```

```
    String kontoNr;  
    double kontoStand;  
    int kundenNr;
```

Datenfelder

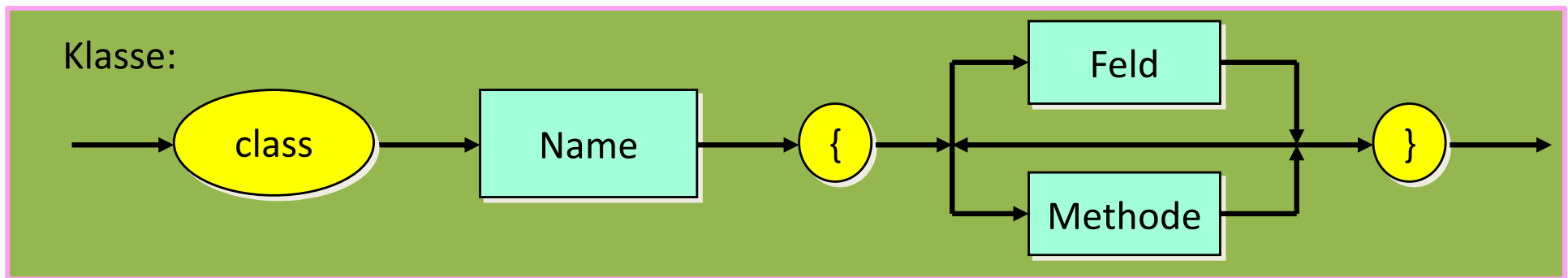
```
    /** ... */  
    void einzahlen(double geld) {  
        ...  
    }  
    /** ... */  
    double abheben(double wunschBetrag) {  
        ...  
    }  
    /** ... */  
    double getKontoStand() {  
        ...  
    }  
}
```

Objekt-
methoden

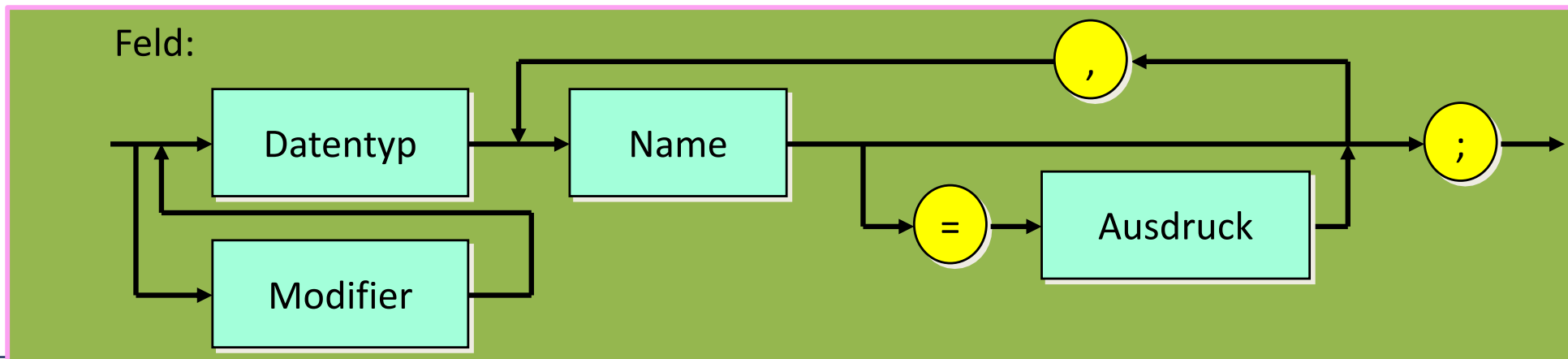
```
}
```

Formale Definition einer Klasse

- **Klassen**-Definitionen haben die folgende – vereinfachte – syntaktische Struktur

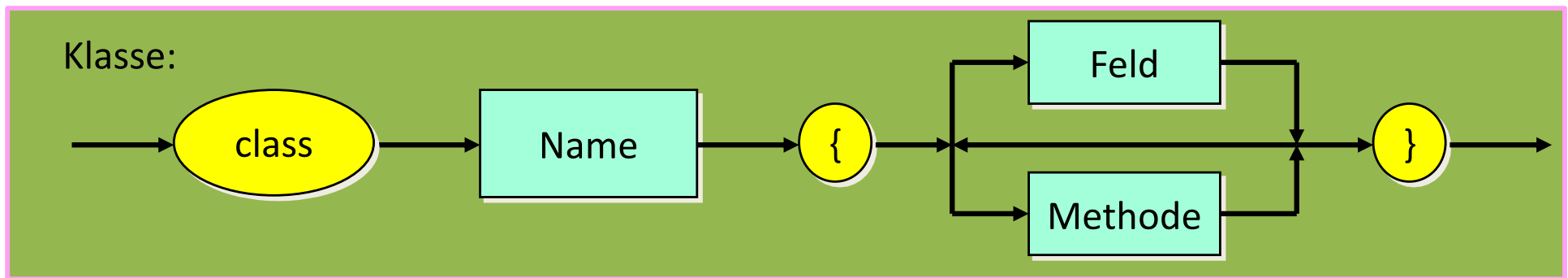


- Felder in einer Klasse werden ähnlich zu lokalen Variablen einer Methode deklariert.

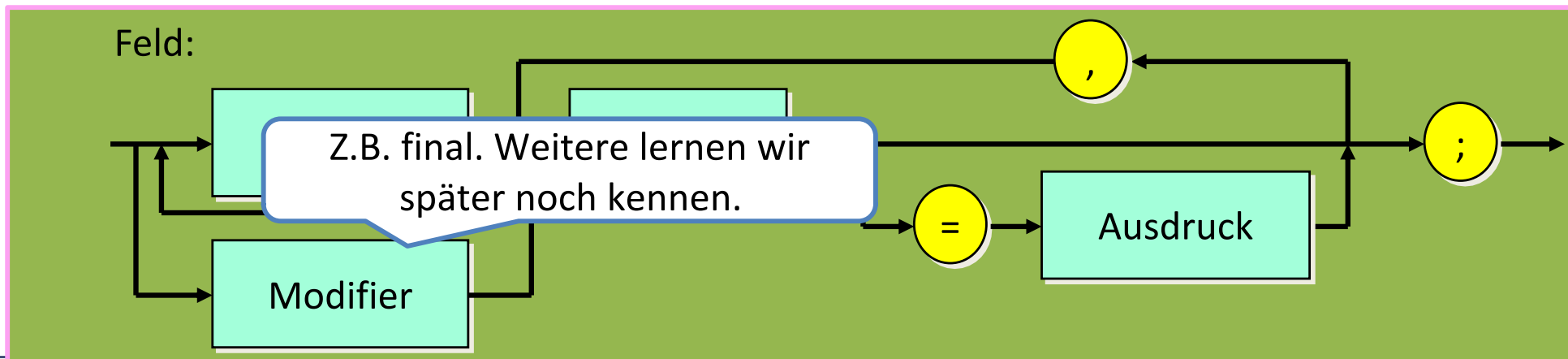


Formale Definition einer Klasse

- **Klassen**-Definitionen haben die folgende – vereinfachte – syntaktische Struktur



- Felder in einer Klasse werden ähnlich zu lokalen Variablen einer Methode deklariert.



Beispiel – Klasse Student

```
class Student {
    String vname;
    String nname;
    int matrnr;
    int fb;
    int fachsemester;

    /** Prüft die Zugehörigkeit zu einem Fachbereich
     * @param fb Fachbereich
     * @return true, wenn der Studierende vom Fachbereich fb
     * ist.
     */
    boolean istImFachbereich(int fb) {
        ...
    }
}
```

Datenfelder in Klassendefinition

- Ausgehend von beliebigen Typen T_1, \dots, T_m kann durch eine Klasse T ein neuer Typ definiert werden.

```
class T {  
    T1 a1;  
    T2 a2;  
    ...  
    Tm am;  
}
```

- Die Klasse T hat m Felder. Jedes Feld hat einen eindeutigen Namen.
- Beispiel

```
public class Student {  
    String vname;  
    String nname;  
    int matrnr;  
    int fb;  
    int fachsemester;  
}
```

oder



```
public class Student {  
    String vname, nname;  
    int matrnr, fb, fachsemester;  
}
```

Deklaration von Variablen

- Zu einer Klasse T können Referenzvariablen deklariert werden.

T myRefVar;

Beispiele:
Student fritz;
Konto vb;

- Analog zu Arrayvariablen verweisen diese Referenzvariablen nur auf den im Heap-Speicher liegenden Speicherplatz eines Objekts der Klasse.
 - Ein Objekt gibt es nach der Variablendeklaration nicht.
- Genau wie bei Arrays kann einer Referenzvariable mit Klassentyp der spezielle Wert null zugewiesen werden.
 - Es gelten dann dieselben Besonderheiten wie bei den Arrays vorgestellt

5.2 Erzeugung der Objekte

Klasse Student

Konstruktoren

Objekte vom
Typ Student



vorname:
nachname:
fb:
matrNr:



...



vorname: Max
nachname: Mustermann
fb: 12
matrNr: 12345



vorname: Ute
nachname: Musterfrau
fb: 13
matrNr: 54321

Klassen als Objektfabriken

- Objekte werden durch den **new-Operator** erzeugt.

```
fritz = new Student();
```

```
vb = new Konto();
```

- Bei dem new-Operator muss der Klassenname gefolgt von einem Klammerpaar angegeben werden.
 - Ähnlich zu einem Methodenaufruf können Parameter übergeben werden. Hierzu werden sogenannte **Konstruktoren** der Klasse benötigt.
 - Objekte werden fast immer über Variablen angesprochen, aber dies ist nicht zwingend erforderlich (→ wir werden später darauf eingehen).
- Der new-Operator liefert eine **Referenz** auf das erzeugte Objekt.
 - Diese Referenz wird in einer **Referenzvariable** hinterlegt.
 - Der Typ der Variable muss zu dem Typ des Objekts passen.
- Die Objekte selbst werden im Heap-Speicher gespeichert.

Klassen als Objektfabriken

- Objekte werden durch den **new-Operator** erzeugt.

```
fritz = new Student();
```

```
vb = new Konto();
```

- Bei dem new-Operator muss der Klassenname gefolgt von einem Klammerpaar angegeben werden.
 - Ähnlich zu einem Methodenaufruf können Parameter übergeben werden. Hierzu werden sogenannte **Konstruktor** der Klasse benötigt.
 - Objekte werden fast immer über **Referenzen** angesprochen, nicht zwingend erforderlich (→ w)
- Der new-Operator liefert eine **Referenz** auf das Objekt.
 - Diese Referenz wird in einer **Referenzvariable** gespeichert.
 - Der Typ der Variable muss **zu dem Typ des Objekts passen**.
- Die Objekte selbst werden im Heap-Speicher gespeichert.

Wenn der Typ des Objekts und der Variablen gleich sind, ist das *passend*. Es gibt noch weitere Regeln, die wir noch kennenlernen.

Stack- und Heap-Speicher

Variable	Wert (Referenz)
fritz	99
sparKonto	42

Stack-Speicher

Adresse	Wert
1	...
2	...
...	...
42	?
43	?
44	?
....	
99	?
100	?
101	0
102	0
103	0
...	...

Heap-Speicher

Objektinitialisierung

- Eine **Initialisierung** der Felder sollte entweder **direkt bei deren Deklaration** oder durch einen Konstruktor stattfinden.
 - Direkte Initialisierung

```
class Student {  
    String vname = "Max";  
    String nname = "Mustermann";  
    int matrnr = 12345;  
    int fb = 12;  
    int fachsemester = 1;  
  
    ...  
}
```

```
class Konto {  
    String kontoNr = "12345";  
    double betrag = 5.0;  
    int kundenNr = 42;  
}
```

- Damit bekommen alle Objekte der Klasse initial diese Werte.
 - Im Fall der Klasse Student ist dies nicht empfehlenswert, da typischerweise die Werte individuell gesetzt werden sollen.
 - Stattdessen empfiehlt sich die Benutzung von **Konstruktoren**.

Objektinitialisierung

- Eine **Initialisierung** der Felder sollte entweder **direkt bei deren Deklaration** oder durch einen Konstruktor stattfinden.
 - Direkte Initialisierung

```
class Student {  
    String vname  
    String nname  
    int matrnr =  
    int fb = 12;  
    int fachsemes  
    ...  
}
```

Spezielle Methoden, die nur bei Objekterzeugung ausgeführt werden.

```
class Konto {  
    String kontoNr = "12345";  
    double betrag = 5.0;  
    int kundenNr = 42;  
}
```

- Damit bekommen alle Objekte der Klasse initial diese Werte.
 - Im Fall der Klasse Student ist dies nicht empfehlenswert, da typischerweise die Werte individuell gesetzt werden sollen.
 - Stattdessen empfiehlt sich die Benutzung von **Konstruktoren**.

Konstruktoren (1)

- Konstruktoren dienen der Initialisierung neu erzeugter Objekte.
 - Ähnlich zu einem Formular müssen dabei Angaben gemacht werden, um die Objekte zu erzeugen.
- In Java besitzen Konstruktoren den **Namen der Klasse; ein Ergebnistyp wird nicht angegeben**.
 - Beim Erzeugen von Objekten der Klasse mit `new` wird stets ein Konstruktor aufgerufen.
 - Dies ist die einzige Möglichkeit Konstruktoren zu nutzen.
 - Sie dienen nur dazu, einem neuen Objekt einer Klasse einen **initialen Zustand zu geben**.

Konstruktoren (2)

- Eine Klasse kann **keinen, einen oder mehrere** unterschiedliche Konstruktoren besitzen.
 - Sollte **kein Konstruktor** zur Verfügung gestellt werden, wird der **Default-Konstruktor** (parameterlos und mit leerem Rumpf) automatisch hinzugefügt.
 - Wird **mindestens ein Konstruktor** in der Klasse zur Verfügung gestellt, wird kein Default-Konstruktor hinzugefügt.
 - Diese Konstruktoren verfügen dann i. A. über Parameter, die zur Erzeugung der Objekte genutzt werden.

Beispiel eines Konstruktors

```
class Student {
    String vorname;
    String nachname;
    int matrnr;
    int fb;
    int fachsemester = 1;

    /** Ein Konstruktor der Klasse Stud.
     */
    Student(String v, String n, int mnr, int f) {
        vorname = v;
        nachname = n;
        matrnr = mnr;
        fb = f;
    }
}

Student s = new Student("Max", "Mustermann", 12345, 12);
```

Ausgabe auf jshell:
s ==> Student@6b09bb57

Mehrere Konstruktoren

- In einer Klasse können mehrere Konstruktoren existieren.
 - Diese Konstruktoren müssen sich in Ihrer **Signatur** (Liste der Typen der Parametervariablen) unterscheiden.

```
class Student {  
    ...  
    Student(String v, String n, int mnr, int f, int fs) {  
        vorname = v;  
        nachname = n;  
        matrnr = mnr;  
        fb = f;  
        fachsemester = fs;  
    }  
    ...  
}
```

Initialisiertes Feld darf überschrieben werden, wenn es nicht final ist.

Standardwerte

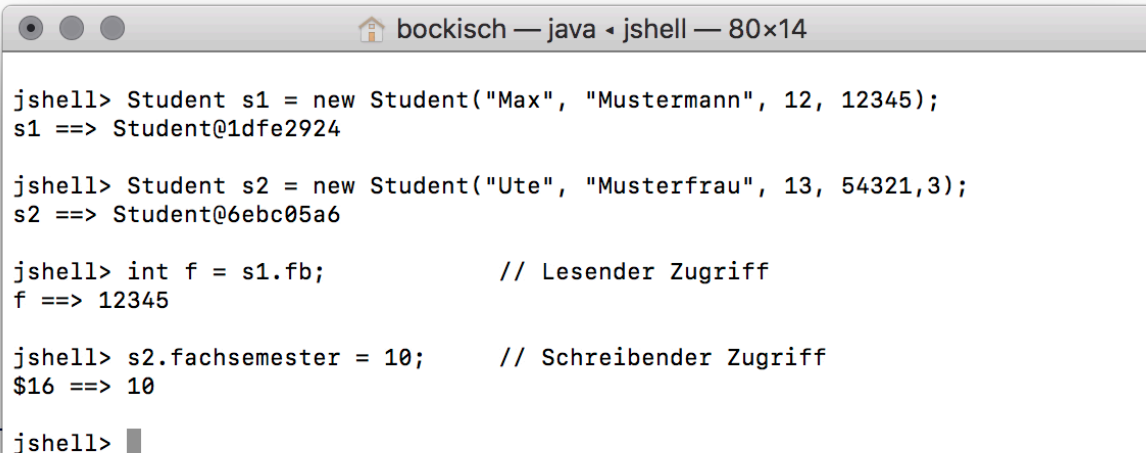
- Datenfelder ohne explizite Initialisierung werden implizit mit einem Standardwert (genau wie bei Arrayelementen) initialisiert
- Auf Datenfelder darf daher bereits vor der ersten Zuweisung zugegriffen werden.

```
class Student {  
    ...  
    Student() {  
        System.out.println(  
            "Vorname: " + vorname + ",\n" +  
            "Nachname: " + nachname + ",\n" +  
            "Matrikelnummer: " + matrnr);  
    }  
    ...  
}
```

5.3 Zugriff auf Datenfelder

- Ist ein Objekt erzeugt und initialisiert worden, ist ein Zugriff auf die Datenfelder schreibend und lesend möglich.
 - Hierzu verwendet man typischerweise eine Variable gefolgt von einem Punkt und den Namen des Datenfelds.

```
Student s1 = new Student("Max", "Mustermann", 12, 12345);  
Student s2 = new Student("Ute", "Musterfrau", 13, 54321,3);  
  
int f = s1.fb;                // Lesender Zugriff  
  
s2.fachsemester = 10;        // Schreibender Zugriff
```



```
bockisch — java • jshell — 80x14  
  
jshell> Student s1 = new Student("Max", "Mustermann", 12, 12345);  
s1 ==> Student@1dfe2924  
  
jshell> Student s2 = new Student("Ute", "Musterfrau", 13, 54321,3);  
s2 ==> Student@6ebc05a6  
  
jshell> int f = s1.fb;                // Lesender Zugriff  
f ==> 12345  
  
jshell> s2.fachsemester = 10;        // Schreibender Zugriff  
$16 ==> 10  
  
jshell> █
```

Zugriff auf Felder: ja oder nein?

- Oft ist es **gefährlich**, Datenfelder der Objekte direkt zu verändern.
 - Dadurch kann ein Objekt mit einem nicht erlaubten Zustand entstehen.
 - Beispiel (Klasse Konto)
 - Ein direkter Zugriff auf das Datenfeld `kontoStand` könnte dazu führen, dass dies **ohne Überweisung, Einzahlung oder Abheben** geändert wurde.
- Zugriffsrechte **public** und **private**
 - **public** erlaubt wie bisher den uneingeschränkten Zugriff auf die Datenfelder.
 - **private** nur dann, wenn man sich in der Klasse, z. B. in einer Methode, befindet.
 - Tatsächlich gibt es noch mehr Optionen für die Zugriffsrechte, auf die wir später zu sprechen kommen.

Geschützte Datenfelder

- Im Allgemeinen sollten **alle Datenfelder einer Klasse als private** deklariert werden.

```
class Konto{  
    private String kontoNr;  
    private double kontoStand;  
    private int kundenNr;  
  
    public void einzahlen(double geld) {  
        ...  
    }  
    public double abheben(double wunschBetrag) {  
        ...  
    }  
    public double getKontoStand() {  
        ...  
    }  
}
```

- Damit ist der Zugriff auf die Datenfelder nur noch innerhalb der Klasse möglich.
 - Den Zugriff von außerhalb ermöglichen wir indirekt über die public-Methoden.

5.4 Objektmethoden

- Die Operationen des neuen Datentyps, der durch die Klasse bereitgestellt wird, werden durch Objektmethoden realisiert.
- Objektmethoden werden innerhalb des Klassenrumpfs definiert
 - Diese Methoden haben stets **Zugriff auf alle Datenfelder eines Objekts** als wären es lokale Variablen der Methode.

Beispiel

```
class Konto {  
    private String kontoNr;  
    private double kontoStand;  
    private int kundenNr;  
  
    public void einzahlen(double betrag) {  
        if (betrag > 0.0)  
            kontoStand += betrag;  
    }  
  
    public double abheben(double wunschBetrag) {  
        ...  
    }  
  
    public double getKontoStand() {  
        return kontoStand;  
    }  
}
```

Aufruf von Objektmethoden

- Objektmethoden können nur im **Kontext eines Objekts** genutzt werden.
 - Beim Aufruf wird **erst das Objekt dann ein Punkt und dann der Methodenname** mit den Parametern angegeben.
 - Damit kann in der Methode auf alle Datenfelder des Objekts lesend und schreibend zugegriffen werden.
 - Beispiel

```
Konto k = new Konto("12345", 0.0, 7);  
// Hier wird ein neues Konto mit Nummer 12345 für Kunde mit  
// Kundennummer 7 erstellt. Der Kontostand ist am Anfang auf 0  
  
k.einzahlen(1000.0);  
// In der Methode einzahlen kann jetzt auf die Datenfelder des Kontos  
// zugegriffen werden, auf das k verweist.  
...  
System.out.println("Aktueller Kontostand: " + k.getKontoStand());
```

5.5 Das Schlüsselwort this

Wie referenziere ich mich selbst?

- Problem
 - Parametervariablen einer und Datenfelder einer Klassen können den gleichen Namen haben. → Namenskonflikt
 - Wie kann man den Namenskonflikt auflösen?
- Lösung: Verwendung von **this**
 - Durch das Schlüsselwort **this** bekommt man die **Referenz des Objekts**, in dem man sich befindet.
 - this kann in der Klasse **wie eine Variable vom Typ der Klasse** verwendet werden.

Beispiel

- Anwendung von this in der Klasse Konto zur Auflösung von Namenskonflikten

```
class Konto {  
    private String kontoNr;  
    private double kontoStand;  
    private int kundenNr;  
  
    Konto(String kontoNr, double ks, int kundenNr) {  
        this.kontoNr = kontoNr;  
        kontoStand = ks;  
        this.kundenNr = kundenNr;  
    }  
    ...  
}
```

Zugriff auf das
Datenfeld

Zugriff auf die
Parametervariable

Selbstreferenz eines Objekts

- Problem
 - Wie kann ein Objekt in einer Methode eine Referenz auf sich selbst als Ergebnis liefern?
- Lösung: Verwendung von **this**
 - Durch das Schlüsselwort **this** bekommt man die **Referenz des Objekts**, in dem man sich befindet.
 - **this** kann als **Ergebnis einer Methode** nach außen geliefert werden.

Beispiel

- Anwendung von this in der Klasse Konto als Selbstreferenz

```
class Konto {  
    private double kontoStand;  
    ...  
  
    /** ToDo: Kommentar fehlt! */  
    Konto einzahlen (double betrag){ // neuer Rückgabetypp!!  
        kontoStand += betrag;  
        return this;  
    }  
    ...  
}
```

this liefert die Referenz
auf das Objekt

Selbstbezug bei Konstruktoren

- Problem
 - Konstruktoren sehen oft sehr ähnlich aus und unterscheiden sich oft nur in einem Parameter.
 - Kann bei der Bereitstellung von Konstruktoren auch ein anderer Konstruktor benutzt werden?
- Lösung: Verwendung von **this**
 - Durch das Schlüsselwort this kann in einem Konstruktor ein **anderer Konstruktor** der gleichen Klassen **aufgerufen werden**.
 - this wird dann **wie ein Methodenname** genutzt.
 - Nebenbedingung
 - Der this-Konstruktoraufruf muss die **erste Anweisung** in einem Konstruktor sein.

Beispiel

- Anwendung von this zum Aufruf eines anderen Konstruktors

```
class Konto {  
    private String kontoNr;  
    private double kontoStand;  
    private int kundenNr;  
  
    Konto(String kontoNr, double ks, int kundenNr) {  
        this.kontoNr = kontoNr;  
        kontoStand = ks;  
        this.kundenNr = kundenNr;  
    }  
  
    Konto(String kontoNr, int kundenNr) {  
        this(kontoNr, 0, kundenNr);  
    }  
}
```

5.6 Das Schlüsselwort static

- Das Schlüsselwort **static** kann vor
 - einer Methode,
 - einem Datenfeld,
 - und einem Initialisierungsblock einer Klasse
 - wird nicht in dieser Vorlesung besprochenstehen.
- Alle mit static gekennzeichneten Komponenten einer Klasse, sind **Bestandteile der Klasse**.
 - Diese Datenfelder und Methoden gehören nicht zu einem Objekt der Klasse.

static Datenfelder

- Manchmal werden Datenfelder benötigt, die unabhängig von den Objekten einer Klasse sind.
 - Für alle Objekte der Klasse sollen die Felder den gleichen Wert haben.
- In der Klasse Konto soll der dispo als statisches Feld gespeichert werden.
 - Diese Felder können durch das Schlüsselwort static definiert werden.

```
static double dispo = 5000.0;
```

- Oft handelt es sich dabei um Konstanten, weshalb static zusammen mit final benutzt wird.

```
static final double PI = 3.14;
```

- Der Zugriff von außerhalb der Klasse erfolgt durch den Namen der Klasse oder einem Objekt der Klasse:

```
Konto.dispo
```

static Methoden

- **Klassenmethoden, die unabhängig von Objekten einer Klasse sind**, werden ebenfalls mit dem Schlüsselwort `static` deklariert.
 - In diesen Methoden steht kein „this“ zur Verfügung.
 - Sie dürfen nur auf static Felder und Methoden der Klasse zugreifen.
- So enthält Java eine Klasse `Math`, in der nützliche mathematische Funktionen wie z. B. `sin`, `cos`, `max`, `min`, `random`, etc. als statische Methoden implementiert sind:

```
static double random() { ... };
```

- Der **Aufruf der Methode** erfolgt über den **Klassennamen**, ohne ein Objekt von `Math` zu erzeugen:

```
double zufall = Math.random();
```

Unterschied zwischen Klassenmethoden und Objektmethoden

- **Klassenmethoden**

- Schlüsselwort `static`
- Zugriff nur auf mit `static`-deklarierten Methoden und Datenfelder der eigenen Klasse möglich.
- Aufruf (typischerweise) über den Klassennamen
 - Beispiel: `Math.sqrt(2)`
- Genauso für Klassen-Felder: `System.out`

- **Objektmethoden** besitzen **nicht** das Schlüsselwort `static`.

- Diese Methoden haben stets **Zugriff auf alle Datenfelder und Methoden eines Objekts**.
- Aufruf einer Objektmethode erfolgt über ein Objekt
 - Beispiel: `meinKonto.einzahlen(10000)`, `out.println("Hallo OOP")`
- Genauso für Objekt-Felder: `student.vorname`

static und private Datenfelder

- Problem
 - Man möchte static-Datenfelder nutzen, aber den Gebrauch außerhalb der Klasse verbieten.
- Lösung
 - Dann ist es sinnvoll **zusätzlich den Modifier private** zu benutzen.

```
private static double dispo = 5000.0;
```

- Damit wird eine unkontrollierte Veränderung des Datenfelds verhindert, da nur **static-Methoden der Klasse** Zugriff auf das Datenfeld haben.

Geheimnis gelüftet - print

- Bisher haben wir folgende Methode zur Ausgabe einer Zeichenkette benutzt.

```
System.out.println("Hallo Welt");
```

- Was steckt dahinter?
 - **System** ist eine Klasse
 - System hat ein statisches Datenfeld **out**
 - Der Typ des Datenfelds out ist die **Klasse PrintStream**.
 - Die Klasse PrintStream hat Objektmethoden **print** und **println**.

5.7 Die main-Methode in Java

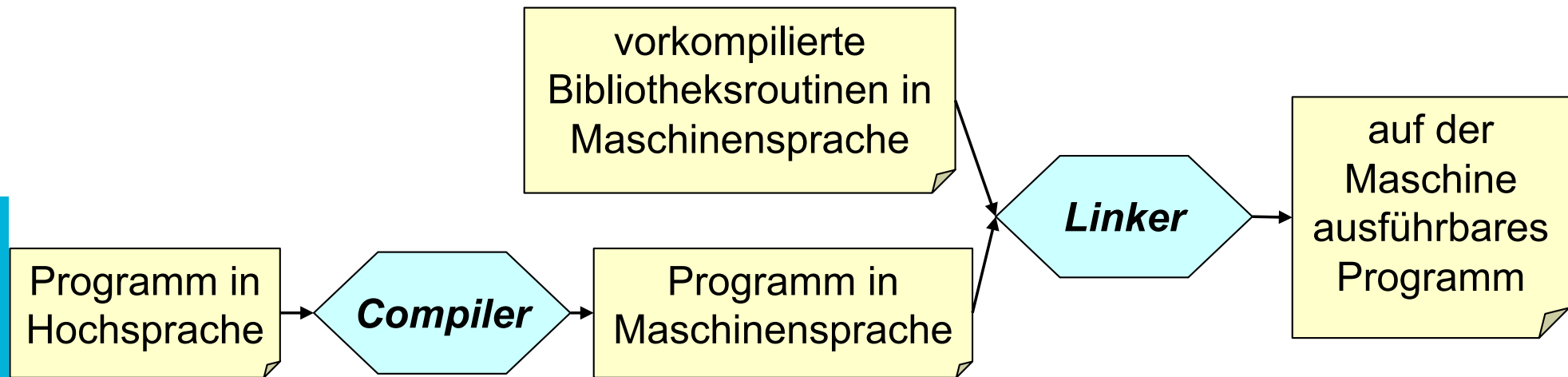
- Bisher haben wir in der Vorlesung jshell benutzt.
 - Vorteil: Schnelles und einfaches Erstellen von Java-Programmen
- Bevor es jshell gab, war die main-Methode der klassische Zugang zum ersten Java-Programm.

```
class Hallo {  
    public static void main(String[] args) {  
        System.out.println("Hallo Welt!");  
    }  
}
```

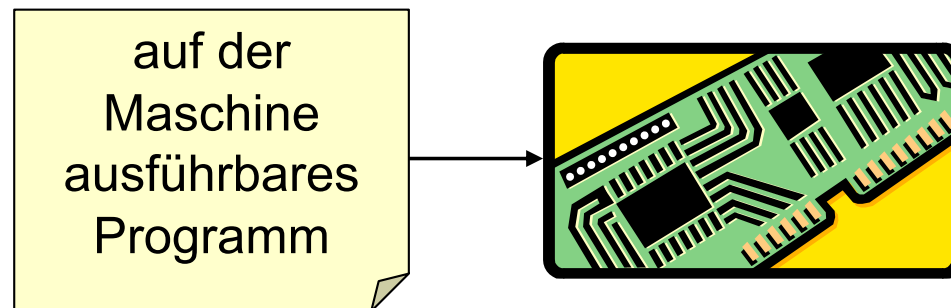
- Abspeicherung der Klasse in einer Datei mit dem Namen der Klasse und der Dateiendung "java".
 - In unserem Fall Hallo.java.

Der Weg zum ausführbaren C-Programm

■ Zur „Übersetzungszeit“



■ Zur „Laufzeit“



Vorteil:

Schnelle Ausführung des fertig übersetzten Programms zur Laufzeit!

Nachteil:

Nur auf einem Maschinentyp (Prozessortyp) ausführbar!

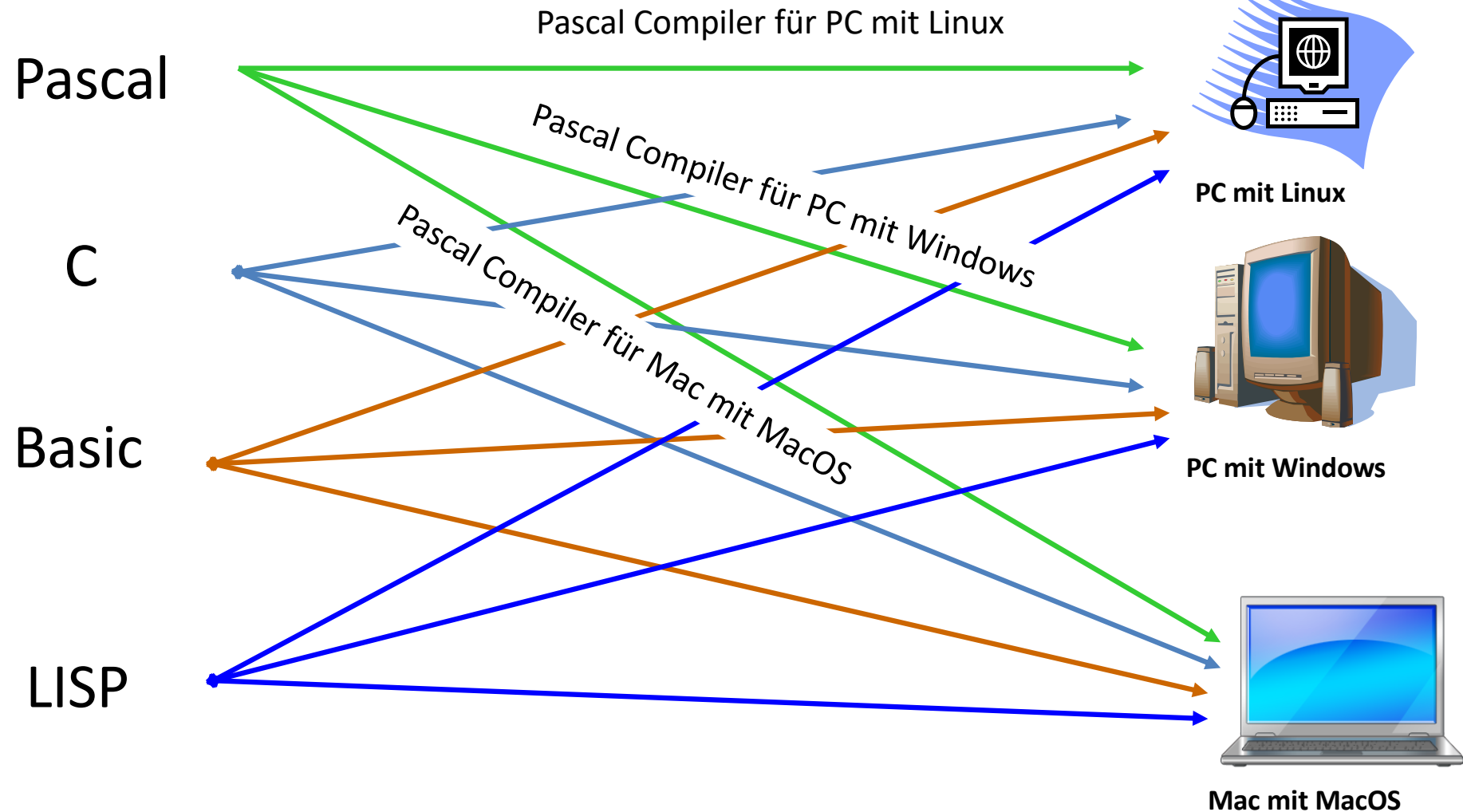
Maschinenabhängigkeit

- Jede Maschine hat andere Befehle
 - Ein Programm für einen **PC** läuft nicht auf dem **Mac** und umgekehrt
- **Betriebssysteme** **abstrahieren** von dem konkreten Rechner und stellen **Werkzeuge** bereit, die von Programmen genutzt werden können.
 - **Dateiverwaltung, Speicherverwaltung, Prozessverwaltung, Input/Output, Hilfsprogramme...**
- Programme rufen die **Funktionen** der Werkzeuge von Betriebssystemen auf.
 - **writefile, print, readfile, send, receive, out, ...**

Konsequenz:

- Jedes Programm läuft nur auf einem bestimmten Rechnertyp mit einem bestimmten Betriebssystem
 - z.B. nur auf **PC** mit **Linux**, oder nur auf **Mac** mit **MacOS**
- Einen Rechnertyp zusammen mit dem darauf laufenden Betriebssystem nennt man auch **Plattform**.

m Sprachen, n Plattformen $\rightarrow m \cdot n$ Compiler



Virtuelle Maschinen

- Eine Virtuelle Maschine ist ein gedachter Computer **VM**.
 - Eine VM bietet auch eine gedachte Maschinensprache an. Diese Sprache wird auch als *Bytecode* bezeichnet.
- **VM** wird auf jedem realen Computer emuliert.
- Für jede Sprache **L** benötigt man nur einen Compiler von **L** nach **VM Maschinencode**.

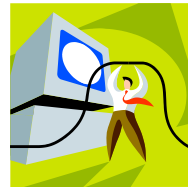
Virtuelle Maschine

Pascal

C

Basic

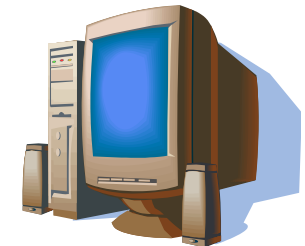
LISP



Virtuelle Maschine



PC mit Linux



PC mit Windows



Mac mit MacOS

M Sprachen

N Plattformen

➔ **M** Compiler, **N** Implementierungen der **VM**

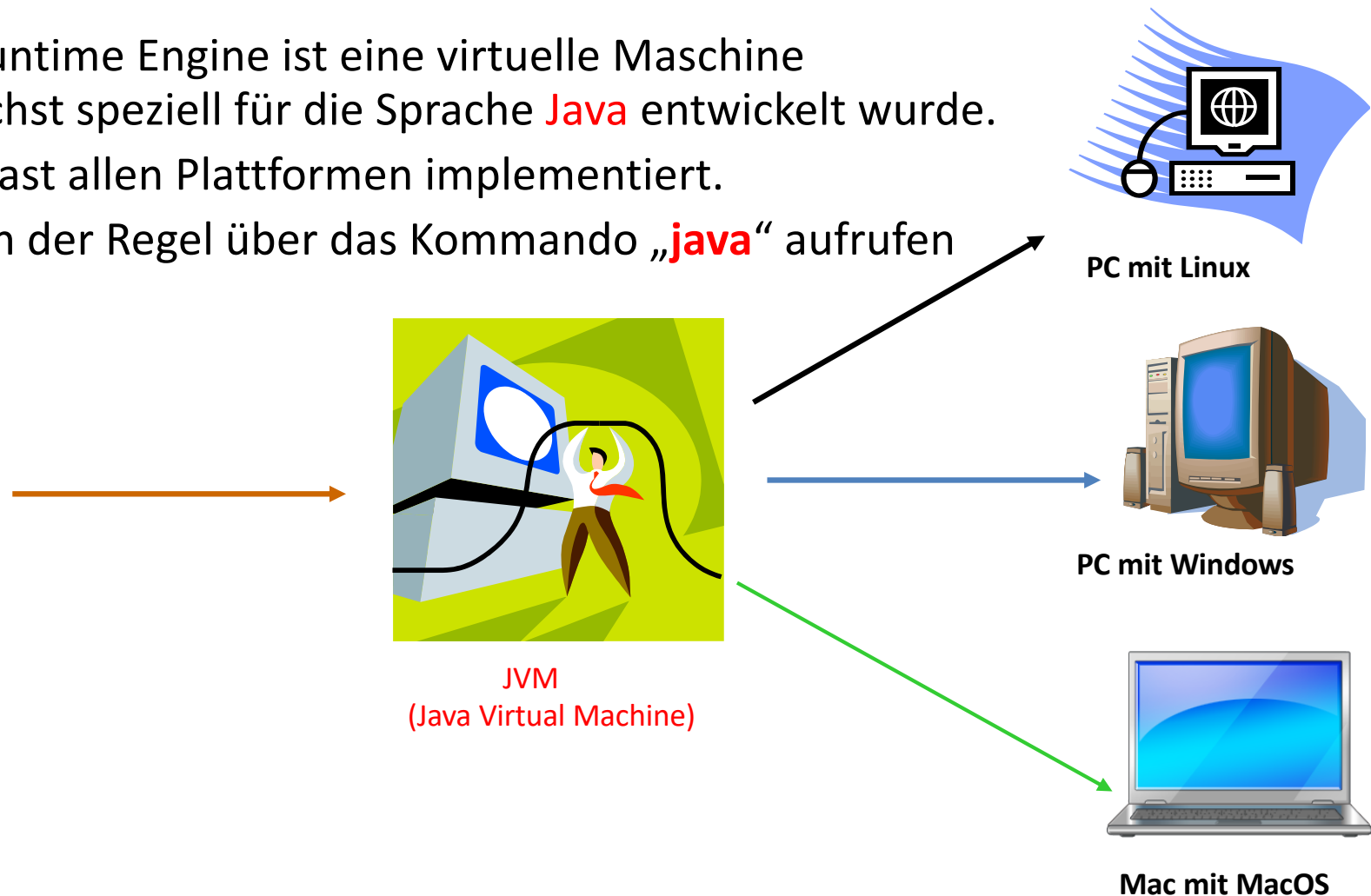
WORA: Write-Once-Run-Anywhere

- Durch die Verwendung einer VM als Zielsystem können lauffähige Programm portiert werden.
 - Entwicklung auf PC unter Windows
 - Ausführung eines Programms auf dem Mac unter Mac OS



Java Virtuelle Maschine

- Die Java-Runtime Engine ist eine virtuelle Maschine
 - die zunächst speziell für die Sprache **Java** entwickelt wurde.
- Sie ist auf fast allen Plattformen implementiert.
- Lässt sich in der Regel über das Kommando „**java**“ aufrufen



Übersetzen von Java-Programmen (1)

- Aus einer Textdatei mit der Endung **.java** erzeugt der Compiler **javac** eine Datei mit gleichem Namen, aber Endung **.class**
- Diese sogenannte class-Datei enthält den **Maschinencode** für die **JVM**

Bsp.java

```
public static void Bsp() {  
    int a = -1;  
    double b = 1;  
    double c = b + 3;  
}
```

Quellprogramm in
Textdatei **Bsp.java**



Compiler
javac

Bsp.class

```
Method void Bsp()  
0 iconst_m1    // -1  
1 istore_0     // store -> a  
2 dconst_1     // 1.0  
3 dstore_1     // store -> b  
4 dload_1      // b  
5 ldc2_w #2    <Double 3.0>  
8 dadd  
9 dstore_3     // store -> c
```

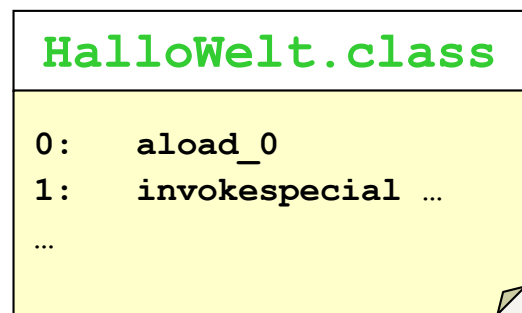
Bytecode in
Datei **Bsp.class**

Maschinencode für die JVM
wird „Bytecode“ genannt.

„Linken“ findet zur
Laufzeit statt.

Ausführung

- Die **class**-Datei mit dem Bytecode wird der **JVM** übergeben.
 - Der Bytecode wird zur Laufzeit in Maschinencode der zugrundeliegenden Plattform übertragen.



Bytecode in
Datei **HalloWelt.class**



Java Virtual Machine
java



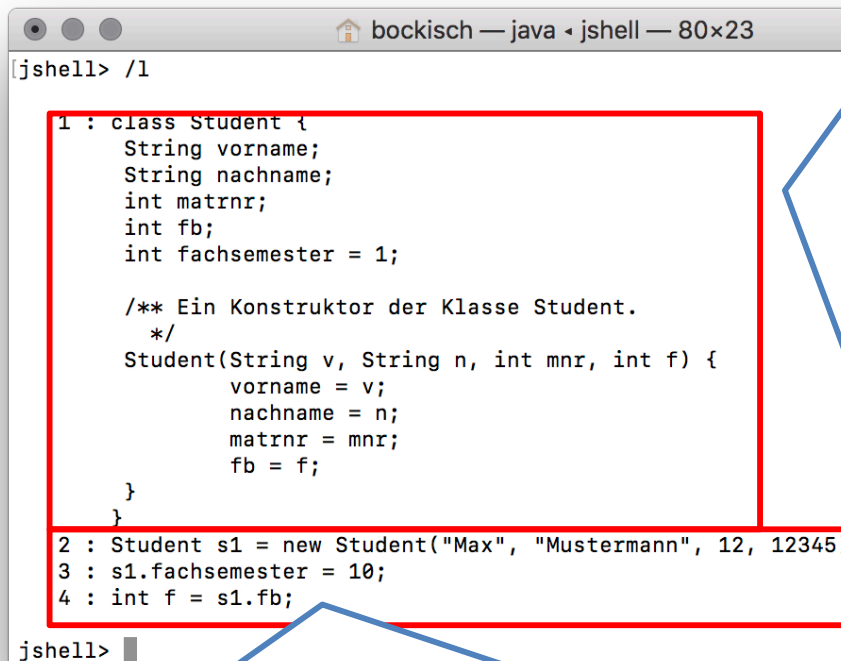
Unter der Haube der JShell

Student.java

```
class Student {  
    String vorname;  
    String nachname;  
    int matrnr;  
    int fb;  
    int fachsemester = 1;  
  
    /** Ein Konstruktor der Klasse Stud.  
     */  
    Student(String v, String n, int mnr, int f) {  
        vorname = v;  
        nachname = n;  
        matrnr = mnr;  
        fb = f;  
    }  
}
```

Temp.java

```
class Temp {  
    public static void main(String[] args) {  
        Student s1 = new Student("Max",  
            "Mustermann", 12, 12345);  
        s1.fachsemester = 10;  
        int f = s1.fb;  
    }  
}
```



```
bockisch — java • jshell — 80x23  
[jshell> /1  
1 : class Student {  
    String vorname;  
    String nachname;  
    int matrnr;  
    int fb;  
    int fachsemester = 1;  
  
    /** Ein Konstruktor der Klasse Student.  
     */  
    Student(String v, String n, int mnr, int f) {  
        vorname = v;  
        nachname = n;  
        matrnr = mnr;  
        fb = f;  
    }  
}  
2 : Student s1 = new Student("Max", "Mustermann", 12, 12345  
3 : s1.fachsemester = 10;  
4 : int f = s1.fb;  
[jshell>
```

Unter der Haube der JShell

Student.java

```
class Student {  
    String vorname;  
    String nachname;  
    int matrnr;  
    int fb;  
    int fachsemester = 1;  
  
    /** Ein Konstruktor der Klasse Stud.  
    */
```

Klassen werden behandelt
als ob sie in einer eigenen
.java-Datei stehen.

JShell führt aus:
javac Student.java
javac Temp.java
java Temp

Alle direkt eingegebenen Befehle
werden behandelt als ob sie in der
main-Methode einer eigenen Java-
Klasse stehen.

Gegebenenfalls werden Semikolons am
Ende der Anweisungen ergänzt.

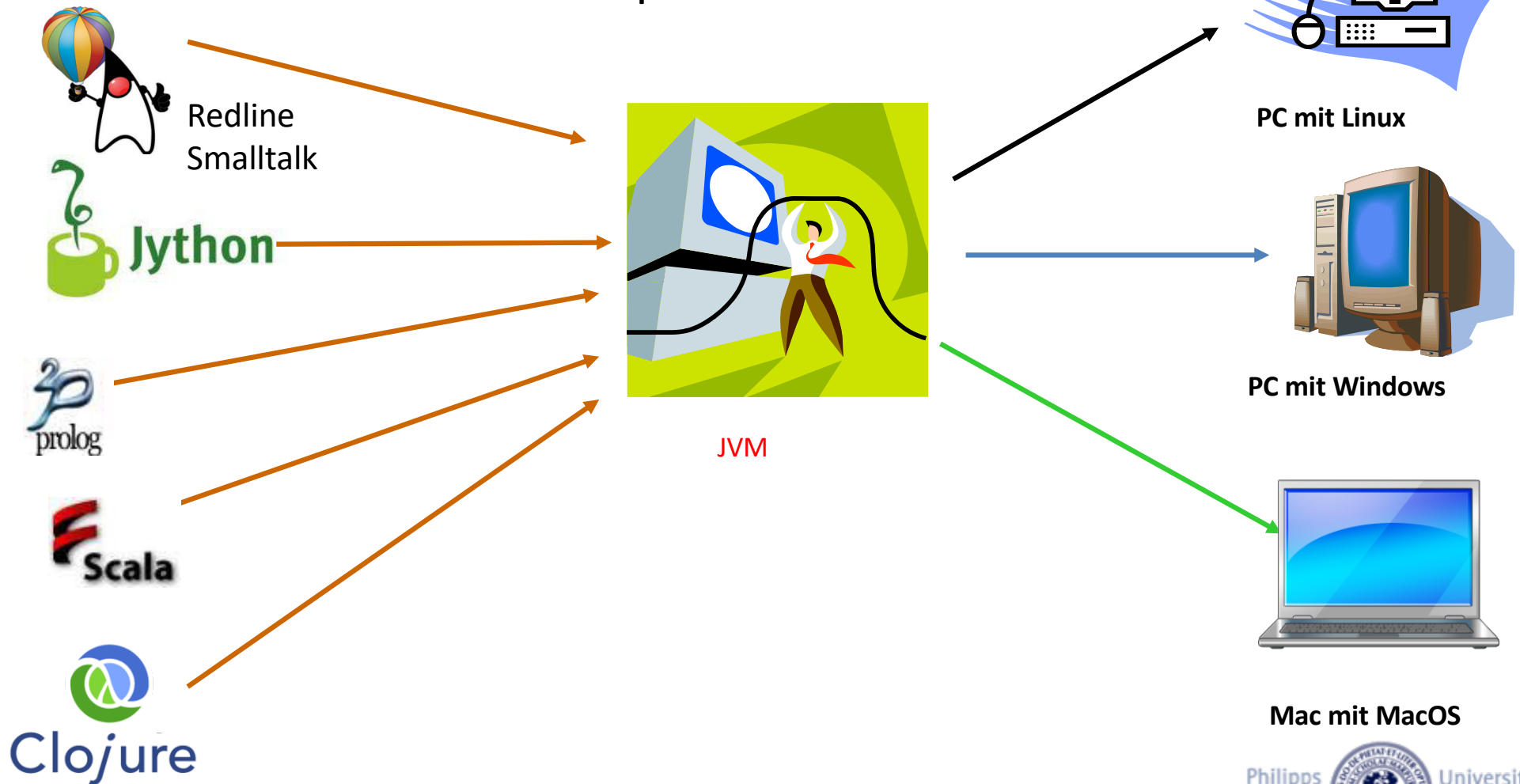
```
Temp  
  
public static void main(String[] args) {  
    Student s1 = new Student(  
        "Mustermann", 12, 12345);  
    s1.fachsemester = 10;  
    int f = s1.fb;  
  
}
```

Alles gab es schon mal ...

- Virtuelle Maschinen für
 - **eine** Sprache und
 - **multiple** Plattformengab es schon früher, aber sie haben sich nicht durchgesetzt.
- Früher existierende virtuelle Maschinen für verschiedene Sprachen:
 - Pascal : **p-Maschine** (Anfang der 80-er Jahre)
 - Smalltalk : **Smalltalk Bytecode Interpreter**
- **Smalltalk**
 - Vorläufer von Java
 - teilweise mit besserer Umsetzung der Objektorientierung im Vergleich zu Java
 - Konzepte waren ihrer Zeit viele Jahre voraus ...
 - ... leider auch den damaligen Möglichkeiten der Hardware.

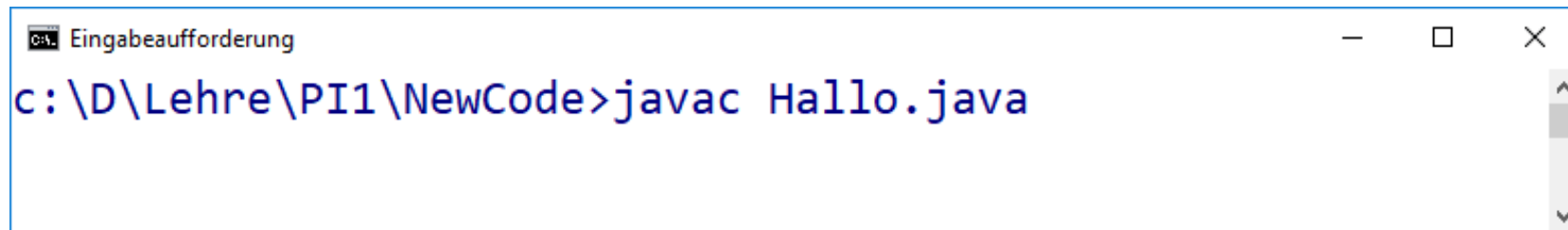
Erfolg steckt an ...

- Seit sich Java durchgesetzt hat, wird die **JVM** auch als Zielmaschine für andere Sprachen benutzt:



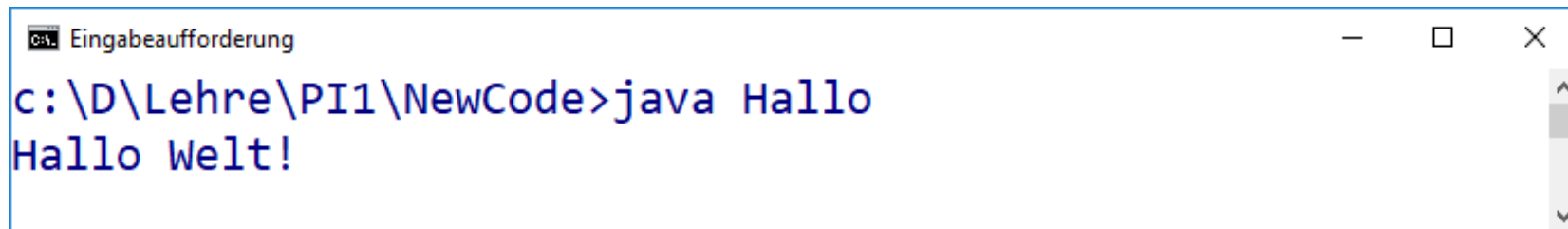
Übersetzen und Ausführen in der Kommandokonsole

- Übersetzen des Programms durch Aufruf des java-Compilers in cmd.



```
C:\D\Lehre\PI1\NewCode>javac Hallo.java
```

- In dem Verzeichnis wurde die Datei Hallo.class erzeugt.
- Ausführen des Programms durch Aufruf von java.



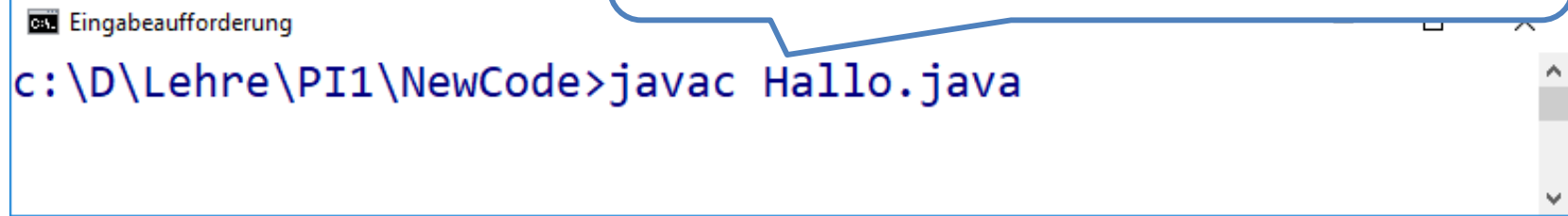
```
C:\D\Lehre\PI1\NewCode>java Hallo
Hallo Welt!
```

- Die Java-Laufzeitumgebung startet das Programm Hallo mit dem Aufruf der main-Methode.

Übersetzen und Ausführen in der Kommandokonsole

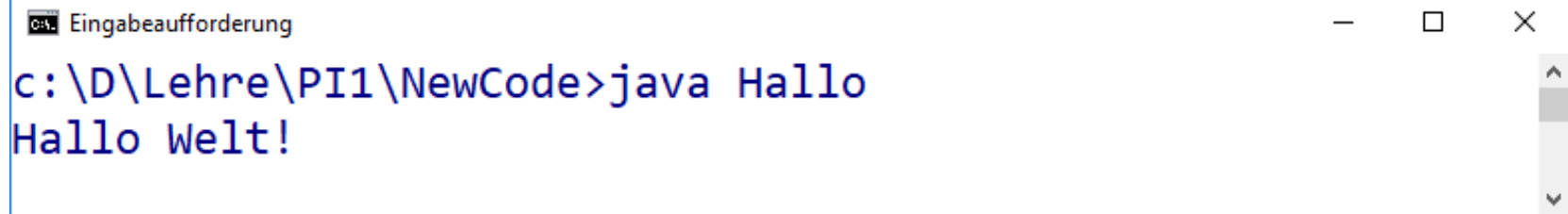
- Übersetzen des Programms mit dem Compiler in cmd.

Die .java-Dateien werden in einem einfachen Text-Editor (später: IDE) geschrieben.



```
c:\D\Lehre\PI1\NewCode>javac Hallo.java
```

- In dem Verzeichnis wurde die Datei Hallo.class erzeugt.
- Ausführen des Programms durch Aufruf von java.



```
c:\D\Lehre\PI1\NewCode>java Hallo  
Hallo Welt!
```

- Die Java-Laufzeitumgebung startet das Programm Hallo mit dem Aufruf der main-Methode.

Aufbau der main-Methode

```
public static void main(String[] args) {  
    ...  
}
```

- Was steckt hinter der main-Methode?
 - Die Methode ist öffentlich nutzbar.
 - Die Methode main ist eine statische Methode.
 - Die Methode liefert kein Ergebnis.
 - Die Methode besitzt eine Parametervariable vom Typ String[].
 - Damit kann man beim Aufruf des Programms beliebig viele Parameter vom Typ String der main-Methode übergeben.

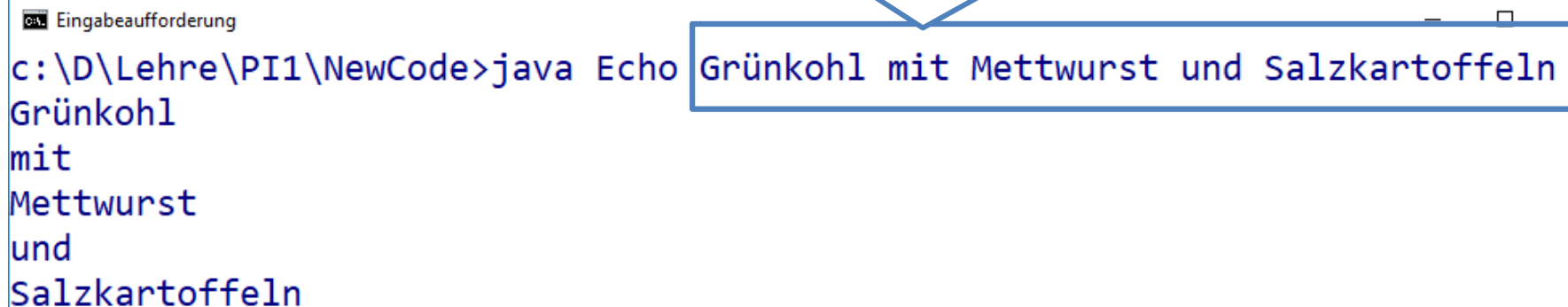
Parameter args in main

- Folgendes Programm gibt alles aus, was in der Kommandozeile nach dem Programmnamen kommt.

```
public class Echo {  
    public static void main (String[] args) {  
        for (String s: args)  
            System.out.println(s);  
    }  
}
```

Kommandozeilen-Parameter werden zu Elementen des Array-Parameters von main.

- Beispiel



The screenshot shows a Windows command prompt window titled "Eingabeaufforderung". The command entered is `c:\D\Lehre\PI1\NewCode>java Echo Grünkohl mit Mettwurst und Salzkartoffeln`. The output of the program is displayed on the next line: `Grünkohl`, `mit`, `Mettwurst`, `und`, and `Salzkartoffeln` on separate lines. A blue box highlights the command arguments "Grünkohl mit Mettwurst und Salzkartoffeln".

Eingabe von Zahlen

- Das Programm Euklid zur Berechnung des größten gemeinsamen Teilers soll zwei Zahlen übergeben bekommen.

- Folgender Aufruf

```
java Euklid 152343 7439823
```

gibt folgende Ausgabe:

```
ggt von 152343 und 7439823 ist gleich 9.
```

- Erforderlich ist dabei die Umwandlung von String nach int.
 - Statische Methode `parseInt(String s)` aus der Klasse `Integer`
 - Falls die Zeichenkette keine Zahl als Parameter hat, bekommt man eine **Fehlermeldung**, eine sogenannte **Exception**, geliefert.

Eingabe von Zahlen

- Das Programm Euklid zur Berechnung des größten gemeinsamen Teilers soll zwei Zahlen übergeben bekommen.
- Folgender Aufruf

`java Euklid 152343 7439823`

gibt folgende Ausgabe:

`ggT von 152343 und 7439823 ist gleich 9.`

- Erfordernis

- Statik
 - Falls eine
- ```
public static void main(String[] args) {
 int x = Integer.parseInt(args[0]);
 int y = Integer.parseInt(args[1]);
 System.out.println("ggT von " + x + " und " + y + " ist " + ggt(x, y));
}
```

## 5.8 JavaDoc Kommentare für Klassen

- Spezielle **Tags mit dem Präfix @** in Kommentaren für Klassen.
  - Allgemein verwendbare Tags
    - @author für Namen des Autors
    - @version für die Version der Klasse/Methode
    - @see für Verweise

# Lesbarkeit von Programmen

- Die Kommentare für javadoc dienen primär den **Benutzern von Klassen**, um die Klasse korrekt anzuwenden.
  - Die Kommentare werden im Normalfall nur für public Klassen, Methoden und Felder erzeugt.
  - Trotzdem auch private Klassen, Methoden und Felder kommentieren!
- Aktualisierte Coding Conventions: siehe ILIAS

# Zusammenfassung

- Klassen in Java
  - Definition eigener Datentypen
    - Wertemenge
    - Operationen
  - Verwendung von Klassen
    - Klassen als Datentypen
    - Klassen als Objektfabriken
- Konzepte von Klassen
  - Datenfelder und Methoden
  - Konstruktoren
- Schlüsselwort static