PK

Abstrakte Maschinen (im weiteren Sinn)

Architektur versus Implementierung (Wiederholung)

Architektur beschreibt Maschine in dem Detailliertheitsgrad, den Programmierer bzw. Benutzer kennen muss

Implementierung der Architektur = tatsächliche Maschine

Alle Implementierungen derselben Architektur verstehen dieselben Programme (Portierbarkeit)

Abstrakte Maschine

abstrakte Beschreibung aller Implementierungen
wobei Implementierungen häufig in Software (keine Hardware)

Beispiel: **Java Virtual Machine (JVM)** – Bytecode

höherer Abstraktionsgrad erhöht Portabilität
niedriger Abstraktionsgrad erhöht Effizienz

JVM-Code für Hello

```
public class Hello extends java.lang.Object{
public Hello();          //Konstruktor (nicht verwendet)
  Code:
   0: aload_0
   1: invokespecial #1; //Method java/lang/Object."<init>"
   4: return

public static void main(java.lang.String[]);
  Code:
   0: getstatic #2;     //Field java/lang/System.out
   3: ldc #3;           //String Hello World!
   5: invokevirtual #4; //java/io/PrintStream.println
   8: return
}
```

Berechnungsmodell

abstrakte Maschine auf sehr hohem Abstraktionsgrad
als reines Gedankenmodell, keine Implementierungen nötig

Beispiele: Lambda-Kalkül, Turing-Maschine

zeigt Möglichkeiten und Grenzen der Programmierung bzw.
der gesamten Informatik auf

Berechnungsmodell hinter jeder Programmiersprache

Objekt als Maschine

jedes Objekt entspricht einer abstrakten Maschine

beschrieben in der Klasse des Objekts

Objektzustand	$\approx$	Speicherinhalt
Objektverhalten	$\approx$	Befehlsausführung
Objektidentität	$\approx$	Internetadresse

diese Sichtweise erlaubt Konzentration auf das Wesentliche

Unterscheidung **Schnittstelle** von **Implementierung**

Softwarearchitektur

beschreibt wichtigste Teile der Software (als Module, Komponenten, Objekte)

Schnittstellen bestimmen Kombinierbarkeit

Analogie: Auto mit Motor, Getriebe, Radaufhängung, . . .

macht Spezialisierung auf einzelne Teile möglich

gute Architektur erleichtert vieles

Aufgabe: Begriff der Schnittstelle

Was versteht man unter der Schnittstelle eines Objekts?

A: spezielle Form der Klasse in Java

B: Architektur des Objekts (im Gegensatz zur Implementierung)

C: Beschreibung, wie das Objekt mit anderen Objekten zusammenpasst

Aufgabe: Abstraktion von Maschinen

Warum programmiert man abstrakte Maschinen und keine realen Maschinen?

A: wegen der Portabilität

B: wegen der Effizienz, die ein Compiler durch Optimierungen erreicht

C: weil Berechnungsmodelle einen sehr hohen Abstraktionsgrad aufweisen

PK

Formale Sprachen

Beschreibung von Sprachen

im Prinzip ähnlich: formale und natürliche Sprachen

Syntax: Aufbau der Sätze bzw. Programme

Grammatik = Regelsystem zur Beschreibung der Syntax

Semantik: Bedeutung von Begriffen, Sätzen, Programmen

in formalen Sprachen: Semantik = komplexe Regeln

Pragmatik: praktische Aspekte der Sprachverwendung

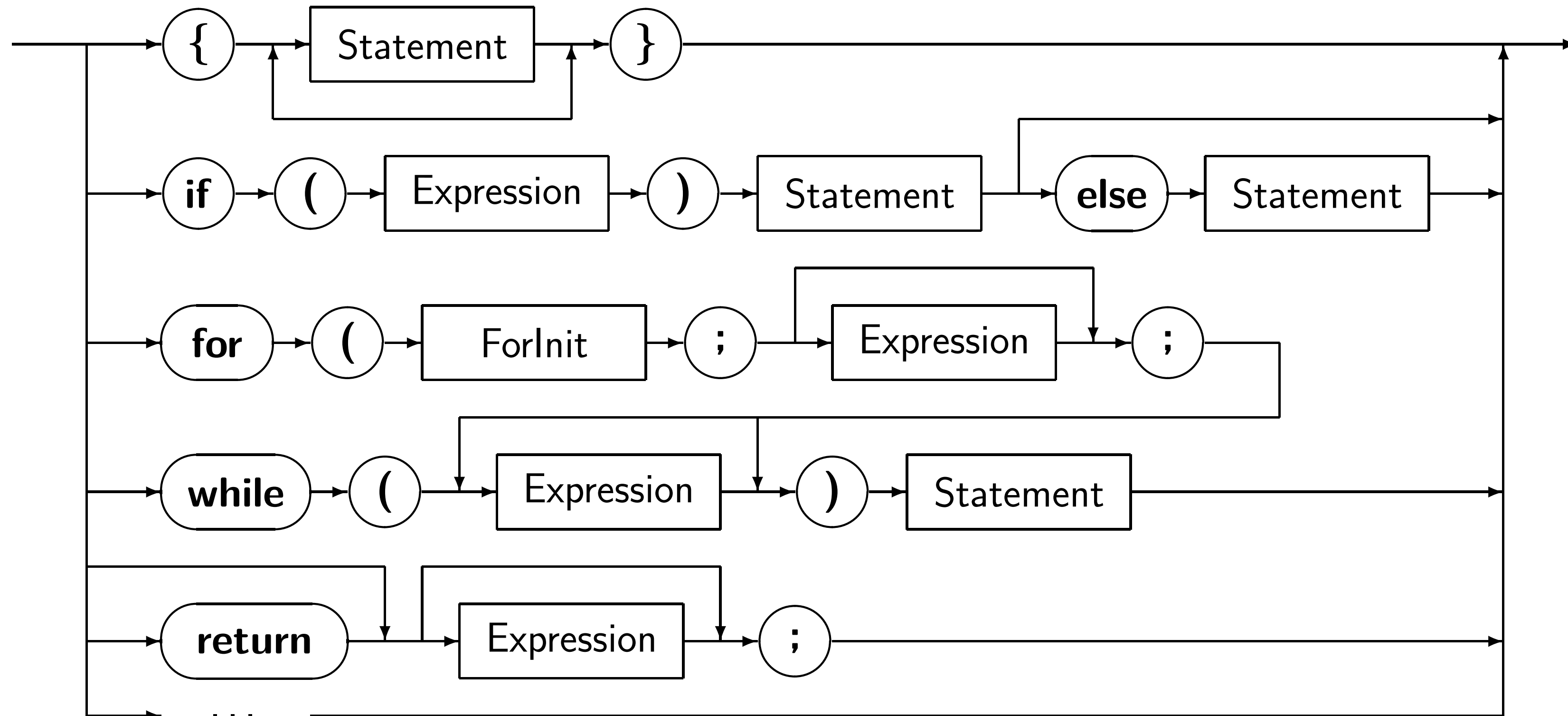
Verhältnis der Sprache zu Sprecher und Angesprochenem

Grammatik (EBNF)

Statement = '**{**' {Statement} '**}**'
| '**if**' '**(**' Expression '**)**' Statement ['**else**' Statement]
| '**for**' '**(**' ForInit '**;**' [Expression] '**;**' [Expression] '**)**' Statement
| '**while**' '**(**' Expression '**)**' Statement
| '**return**' [Expression] '**;**'
| [Expression] '**;**'
| ...

Alternative $A \mid B$:	X ('**+**'|'**−**'|'*****') Y	$\rightarrow$	X**+**Y, X**−**Y, X*****Y
Wiederholung $\{A\}$:	'**{**' $\{X\}$ '**}**'	$\rightarrow$	**{}**, **{**X**}**, **{**X X**}**, ...
Option $[A]$:	'**if**' '**(**' X '**)**' Y ['**else**' Z]	$\rightarrow$	**if(**X**)** Y, **if(**X**)** Y **else** Z

Grammatik (grafisch)



Beispiel: Syntax, Semantik, Pragmatik

Ausdruck entsprechend der Grammatik:

```
if(zahl<0){zahl=zahl+grenze;}
```

gleiche Syntax und Semantik, andere Pragmatik:

```
if (zahl < 0) {
    zahl = zahl + grenze;
}
```

gleiche Semantik, andere Syntax und Pragmatik:

```
if(zahl<0)zahl=zahl+grenze;
```

Bestandteile eines Programms

```
import java.util.Random;                    // Programmorganisation
public class UnbekannteZahl {
    private int zahl;                           // Datenstrukturen
    public UnbekannteZahl (int grenze) {
        zahl = (new Random()).nextInt() % grenze;
        if (zahl < 0) {
            zahl = zahl + grenze;
        }                                         // Algorithmen
    }
    public boolean gleich (int vergleichszahl) {
        return (zahl == vergleichszahl);
    }
    ...                                            // Umgebung
}
```

Statisch versus dynamisch

Algorithmen, Datenstrukturen, Programmorganisation und Umgebung **statisch** (ändern sich zur Laufzeit nicht)

Programmablauf und Daten in Datenstrukturen **dynamisch**

statische versus dynamische Sprachen:

- worauf mehr Wert gelegt wird (statische oder dynamische Aspekte)
- wobei Grenzen fließend sind

statisch: leichter lesbar, besser überprüft

dynamisch: leichter schreibbar, flexibler

Aufgabe: Statisch versus dynamisch

Warum sorgt man für *statische* Algorithmen und Datenstrukturen obwohl der Programmablauf und die Daten in den Datenstrukturen ohnehin *dynamisch* sind?

A: um die Flexibilität beim Programmieren zu erhöhen

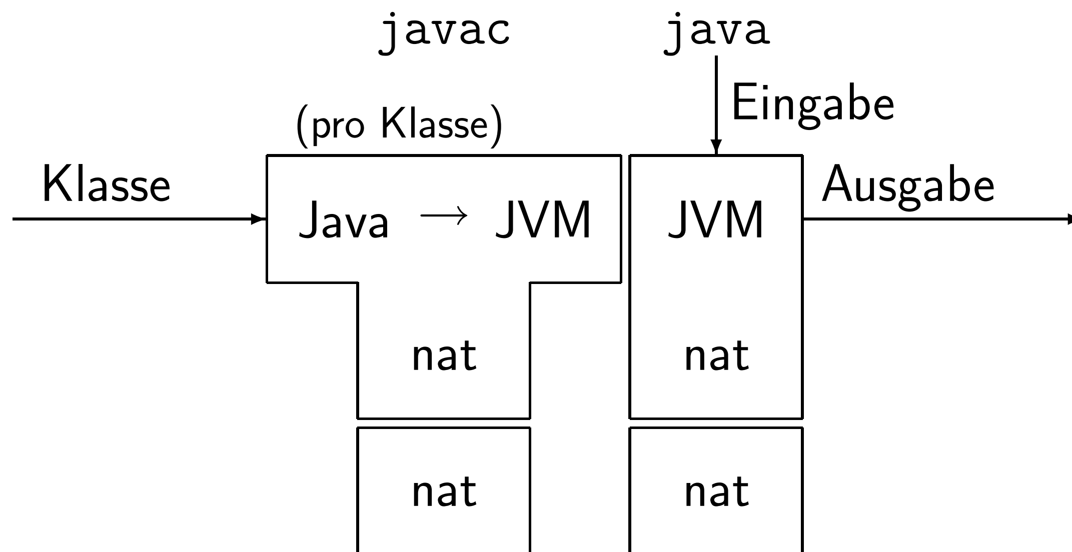
B: um Programme einfacher verständlich zu machen

C: um dem Compiler und Interpreter die Arbeit zu ermöglichen oder erleichtern

PK

Compiler und Interpreter

Java-Programme: übliche Ausführung



PK

TU WIEN

Funktionsweise eines Interpreters

Bei Ausführung ständig wiederholt (wie im Prozessor):

1. hole nächsten Befehl aus Speicher (durch PC adressiert)
2. erhöhe PC um 1
3. interpretiere Befehl und führe ihn aus

Vergleich mit Ausführung von `Zahlenraten`:

1. hole nächste Zahl von Eingabe
2. wobei darauffolgende Zahl zur nächsten Eingabe wird
3. vergleiche Zahlen und gib Meldung aus

Aufgaben eines Compilers

Zielcode erzeugen, semantisch äquivalent zu **Quellcode**

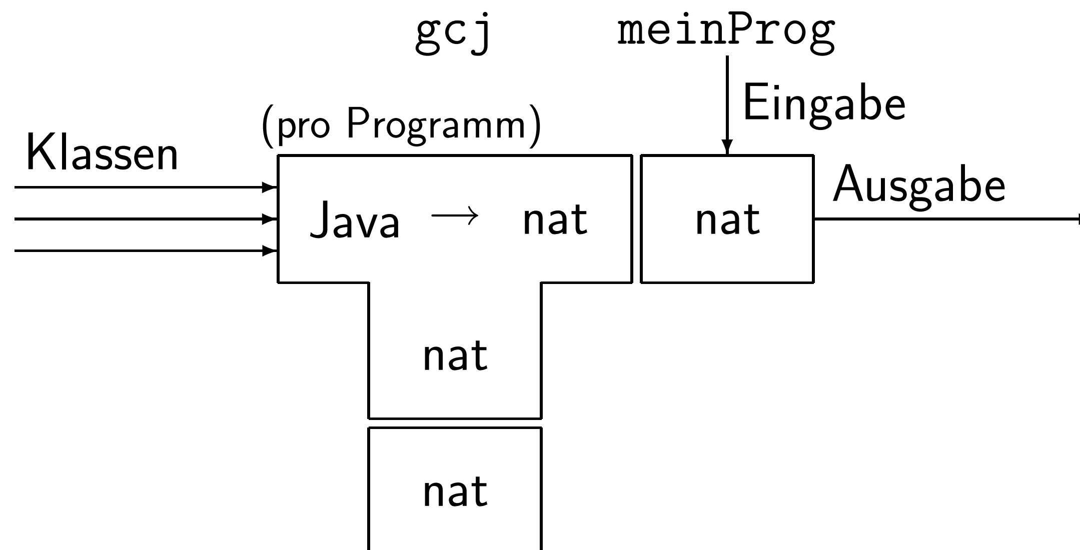
Vermeiden von Laufzeitfehlern

statische Fehlermeldungen

Warnungen

Optimierungen

Übersetzung ohne Zwischencode



Mehrere Übersetzungsschritte