Westfälische
Wilhelms-Universität
Münster

Programmieren in Java

Einführung in die (imperative) Programmierung

wissen.leben
WWU Münster

Name: Patrick Förster, Michael Hasseler

ZENTRUM FÜR
INFORMATIONS
VERARBEITUNG

Programmierung

- Ziel: Zielsystem soll eine bestimmte Aktion ausführen
  - *Zielsystem*: Eine Plattform wie Windows oder MacOS oder ein bestimmter Prozessor
  - *Aktion*:
    - Addiere zwei Zahlen
    - Gebe einen Text auf dem Bildschirm aus
    - Sende Daten (an einen Drucker, über Netzwerkverbindung, ...)
    - ...
- Problemstellung: Wie verständigt man sich mit dem Zielsystem
- Antwort: Maschienencode

Maschienencode

- Sprachumfang des Zielsystems wird als Maschinencode bezeichnet
- Satz in der Maschinensprache besteht aus Folge von Nullen und Einsen

```
11111010
10110011
11000000
```

- Ein Satz beinhaltet den „Befehl“ (Opcode) sowie evtl. „Daten“
- Aufgeteilt in eine Reihe von Bits (im „Beispiel“: 8Bit)
- Ein oder mehrere Sätze zusammen ergeben ein „Programm“

- Maschinencode als Kommunikationsmittel ungeeignet:
  - Programmcode schwierig zu schreiben und ohne weitere Informationen überhaupt nicht zu lesen
  - Unterschiedlicher Sprachumfang einzelner Zielsysteme
  - Gleiche Reihenfolge von Bits auf verschiedenen Zielsystemen verschieden interpretiert

Assembler

- mnemonische Symbole (*Mnemonics*) für mehr Lesbarkeit:

```
10110000 01100001

movb $0x61, %al
```

Quelle: http://de.wikipedia.org/wiki/Assemblersprache

- Der hexadezimal Wert **61(=97)** soll in das untere (l = low) Ende des Registers (grob: Speicherbereich) **a** geschoben werden

- Nicht ideal, aber brauchbar
- Heute noch von Bedeutung: Performance

- Assembler?
  - Der mnemonische Code muss in Maschinencode übersetzt werden
  - Das Übersetzungsprogramm wird „Assembler“ genannt

Compiler

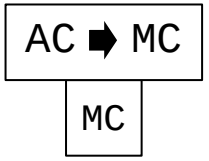
- Übersetzt Code einer bestimmten Sprache in eine andere Sprache
- Der Quellcode: *„Source-Programm“*
- Ergebnis: *„Object-Programm“*

Source	→	*Object*
`movb $0x61, %al`	→	10110000 01100001

- Den Übersetzungsprozess bezeichnet man als „compilieren“
- Der Compiler überprüft jeden Satz des Source-Programms auf de richtigen Satzbau (*Syntax*)
- Falscher Satzbau führt zu einem „Compile-Fehler“

- Der Compiler überprüft allerdings **nicht** , ob die Bedeutung (*Semantik*) des Satzes überhaupt Sinn macht

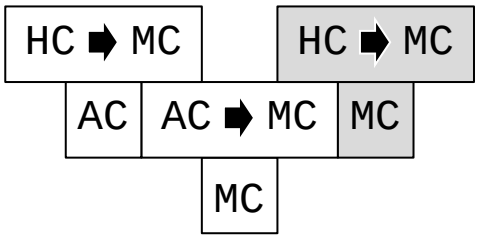
Bootstrap



• Wenn *MC* eine Maschinensprache ist und *AC* eine Assemblersprache, dann soll folgender T-Block den zugehörigen Compiler/Assembler darstellen (geschrieben in MC!):

• Assemblersprachen sind noch keine „Höhere Programmiersprache“ (*HC*)
• Der „Maschinen- zu Assemblercode“ Trick ist universell:

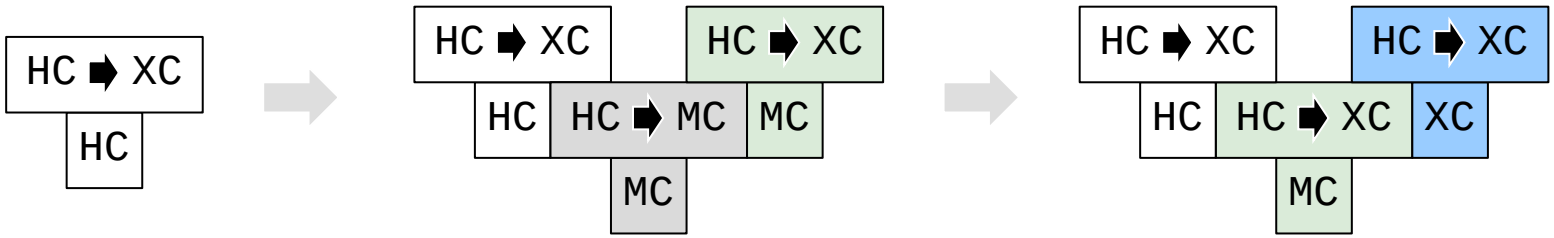
• Compiler in *AC* für eine neue Sprache *HC*, der *MC* produziert
• Den bereits existierenden Assembler nutzt man um einen neuen Compiler (*HC*→*MC*) zu erstellen
• Nun hat man einen Compiler, der die neue Sprache für ein Zielsystem mit der Sprache *MC* übersetzen kann und auf *MC* lauffähig ist



Bootstrap (II)

- Was aber, wenn *HC* auch auf einem anderen Zielsystem mit XC eingesetzt werden soll?

- Man schreibt in *HC* einen Compiler für *XC*
- Den Compiler übersetzt man mit dem *HC→MC* Compiler
- Man erhält einen HC→XC Compiler, der allerdings nur auf MC läuft
- Diesen kann man nun nutzen um den ursprünglichen Compiler zu XC zu übersetzen (*Bootstrap*)



- Nur der HC→XC in HC Compiler musste neu geschrieben werden!

Interpreter

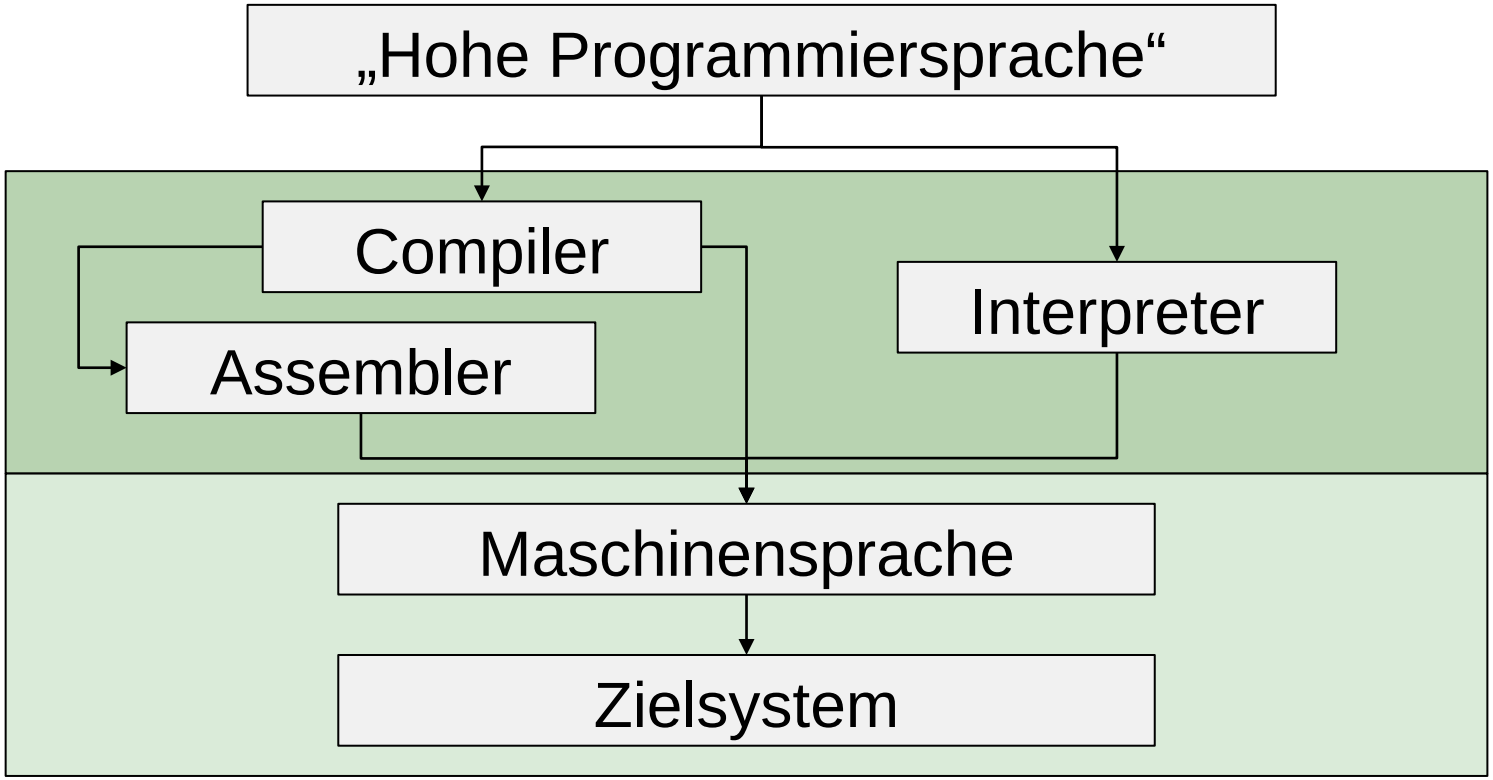
• Compiler: Vor dem Ausführen eines Source-Programms muss dieses compiliert werden

• Interpreter: Zur Laufzeit analysieren, übersetzen und ausführen
• Nachteil:
  • Interpreter muss zum Ausführen mitgeliefert werden
  • Analyse und Übersetzung zur Laufzeit kostet eben diese

• Vorteil:
  • Interpreter eignen sich hervorragend zum „Rapid Prototyping“

• Wichtig: Ob eine Sprache compiliert oder interpretiert wird, ist kein Merkmal der Sprache!

Übersicht



„Höhere Programmiersprachen“

- Die Anzahl an Programmiersprachen ist gefühlt kaum noch aufzählbar
- Sprachen unterscheiden sich nicht nur in verschiedenen „Dialekten“
- Verschiedene „Philosophien“:
  - Imperative Programmiersprachen (*C, Pascal*)
  - Objekt-orientierte Sprachen (*Java, C++, Objective-C*)
  - Funktionale Sprache (*LISP, Haskell*)
  - Deklarative Programmiersprachen (*SQL*)
  - Logische Sprachen (*Prolog*)
  - …
- Für diese Vorlesung: Imperative und OO-Programmiersprachen

Java

- Höhere Programmiersprache
- Aufgebaut auf dem Paradigma der „Objektorientierten Programmierung“
- Wie die meisten OOP-Sprachen aber auch imperativ
- Angelehnt an C++
- Allerdings „schlanker“

- Version 1.0 erschien 1996
- Mit Version 1.2 (1998) sprach man von Java 2
- Version 5.0 (eigentlich 1.5) oder auch Java 5 aus dem Jahr 2004 brachte umfangreiche Änderungen, die sich bis zur Sprache selbst durchschlugen (bspw. „Generics“, „Enums“)
- Die Versionen 6 (2006) und 7 (2011) brachten kleinere Erweiterungen
- Erst mit Version 8 (September 2013) kann mit der Einführung von „Closures“ wieder ein größere Schritt in der Evolution der Sprache erwartet werden

Bytecode

- Ziel: Ein Programm soll auf verschiedenen Zielsystemen lauffähig sein
- Durch Kompilieren nicht möglich:
  - Woher weiß man, auf welchem System das Kompilat ausgeführt werden wird?
- Lösung: „Bytecode“
- Ein Java-Compiler erstellt nicht Maschinen- sondern Bytecode
- JRE (Java Runtime Environment) ist „virtuelle Maschine“, die Bytecode versteht
- JRE nötig um Java-Programm überhaupt ausführen zu können
- Bytecode wird von der virtuellen Maschine für das jeweilige Zielsystem interpretiert

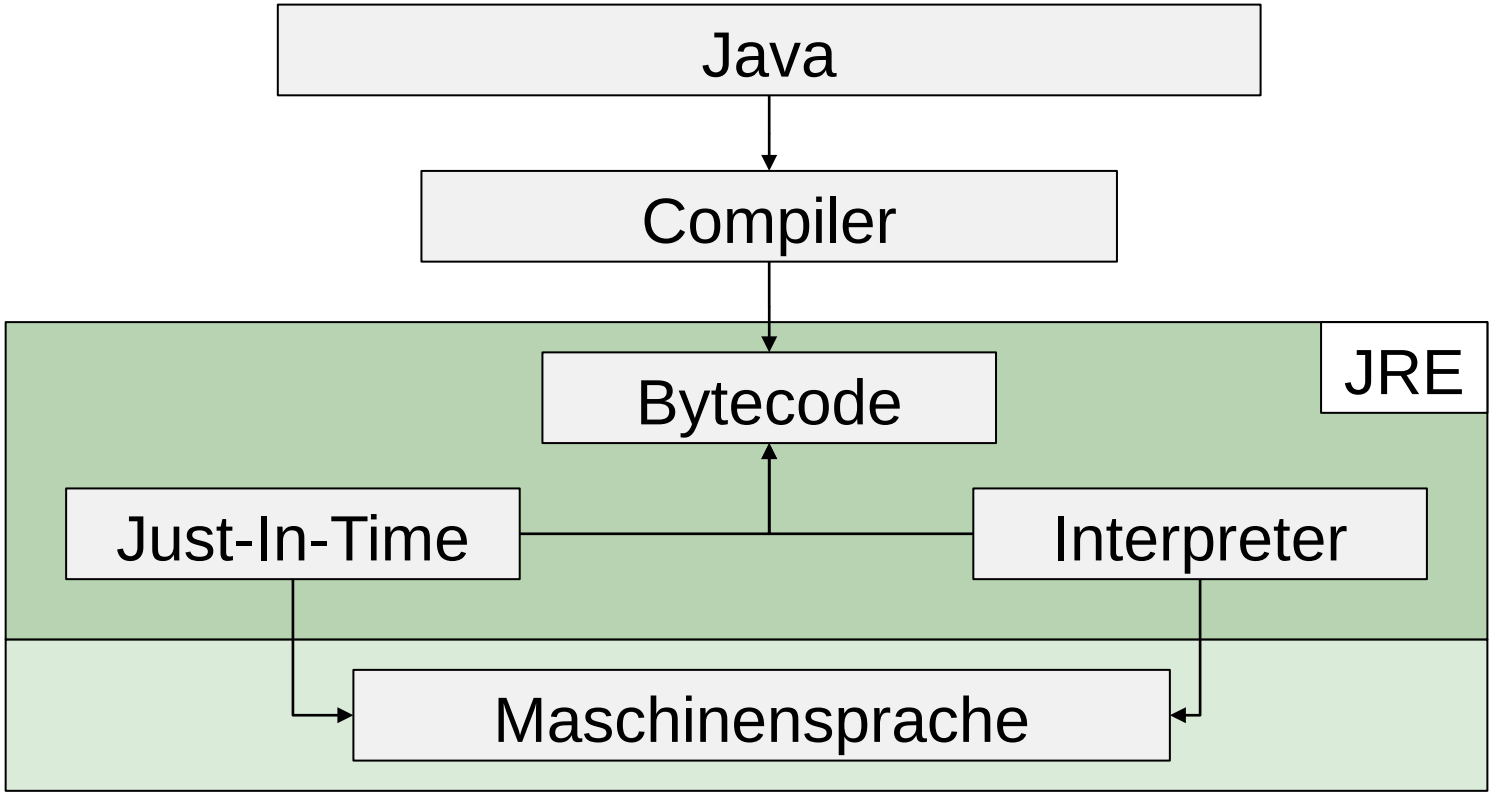
- **Achtung**: Natürlich kann man Java auch direkt zu Maschinencode kompilieren oder gänzlich interpretieren (siehe Folie 8)

Just-In-Time Compiler

- Ist Java langsam, weil Bytecode interpretiert wird?

- Mit Java 2 wurde der „Just-In-Time Compiler“ (HotSpot) eingeführt:
  - Häufig wiederkehrende Code-Einheiten werden zur Laufzeit erkannt und erst dann zu Maschinencode des jeweiligen Zielsystems übersetzt
  - Im nächsten Zyklus wird die Code-Einheit nicht mehr interpretiert, sondern der bereits fertig compilierte Maschinencode ausgeführt
  - Für die Code-Einheiten können zum Teil Optimierungen vorgenommen werden, zu denen normale Compilier nicht in der Lage wären (Stichworte: „Dynamische Optmierung“ oder „Closes World-Annahme“)

Übersicht: Java



Syntax

- Höhere Programmiersprachen bieten eine Reihe von Sprachkonstruktionen, die dazu dienen den Programmfluss zu steuern
- Es gibt Regeln, die festlegen, ob ein Satz überhaupt ein regulärer Ausdruck der Sprache ist
- Alle Regeln einer Sprachen beschreiben die „Syntax“ der Sprache
- Erst wenn ein Satz gänzlich syntaktisch korrekt ist, kann seine „Semantik“ (Bedeutung) festgestellt werden
- Sätze können in kleinere Einheiten, den sogenannten Tokens aufgeteilt werden
- Tokens werden zumeist durch „Whitespaces“ (Trennzeichen) von einander getrennt
- In den meisten Programmiersprachen gibt es zudem ein bestimmtes Trennzeichen, das einzelne Sätze von einander trennt (Java: „;“)

Syntax: Beispiel

- Java-Code:

```
class Program {
   public static void main(String[] args){
      println(27 * 37 + 1);
   }
}
```

Tokens		
class	program	{
public	static	void
main	(	string
[	]	Args
)	println	27
*	37	+
1	;	}

Token

- Jedes Token eines Satzes lässt sich in eine der folgenden Kategorien eingliedern:
  - **Bezeichner**: Das Token besteht aus Buchstaben eines zuvor definierten Alphabets (bspw.: A-Z und 0-9) und stellt einen Name dar, dessen Bedeutung sich aus dem Kontext des Codes ergibt
  - **Literal**: Ein Literal ist ein konstanter – also unveränderbarer – Ausdruck, bspw. eine Zahl
  - **Schlüsselwort**: Schlüsselwörter sind die Bausteine aller Sprachkonstruktionen und formen damit die Gestalt einer Sprache
  - **Separator**: Dies sind spezielle Trennzeichen, die nicht nur Tokens von einander trennen, sondern je nach Kontext eine bestimmte Semantik erfüllen (z.B: Klammern in mathematischen Ausdrücken)
  - **Operator**: Operatoren sind Vorschriften zur Verarbeitung einer bestimmten Anzahl von Operanden (bspw. „+“ für die Addition zwei Zahlen)

Token: Beispiel

- Java-Code:

```
class Program {
   public static void main(String[] args){
      println(27 * 37 + 1);
   }
}
```

Kategorie	Tokens
Bezeichner	Program, main, String, args, println
Literal	27, 37, 1
Schlüsselwort	class, public static, void
Separator	{, }, (,), ;
Operator	*, + , []

- Das Beispiel sollte klar machen: Bezeichner können nicht gleich einem Schlüsselwort sein

Semantik Satz für Satz

- Vorweg: Die Separatoren „{„ und „}“ definieren Code-Blöcke, die semantisch zum öffnenden Satz gehört

```
class Program
```

- Das Schlüsselwort **`class`** besagt, dass der nachfolgende Block eine *Klasse* bildet, die den Namen „Program“ tragen soll

```
public static void main(String[] args)
```

- Der Compiler weiß erst ab dem Separator „(“, dass der nächste Block eine *Methode* namens „main“ darstellen soll, die einen Parameter vom *Datentyp* **`String[]`** erwartet, der im Block „args“ heißen wird
- Die Schlüsselwörter **`public`** und **`static`** sind sogenannte *Modifier* und **`void`** beschreibt einen speziellen *Datentyp*

```
println(27 * 37 + 1)
```

- Hier signalisiert „(“, dass eine Methode namens **`println`** aufgerufen werden soll, dessen erster Parameter das Ergebnis des *Ausdrucks* **`27*37+1`** ist

Ein vorläufiger Rahmen

- Das Beispiel zeigt einen Code-Rahmen, der im folgenden des Öfteren zum Einsatz kommen wird:
- *Methoden*, *Datentypen*, *Modifier* und *Klassen* werden später erklärt

```
class Program {
   public static void main(String[] args){
      System out println("Hello World");
   }
}
```

- Erst einmal kann man davon ausgehen, dass die Klasse „Program“ nötig ist, um ein Programm ausführen zu können
- Das Programm startet mit dem in der Methode **`main`** definierten Code
- Der Parameter **`args`** beinhaltet Daten, die beim Programmstart mitgeben werden können
- Mit **`System.out.println`** können Ausgaben produziert werden (unter der Annahme, dass das Programm über eine Eingabeaufforderung gestartet wird, erscheinen die Ausgaben in genau dieser)
- **`“Hello World“`** ist eine Zeichenkette, die als *Literal* angesehen werden kann, dessen Wert der Zeichenkette ohne den Anführungszeichen entspricht

JDK

- Das Beispiel ist eine Umsetzung des „Hello World“-Programmes, welches als Pflichtprogramm bei der Einarbeitung in eine neue Sprache gilt
- Der Code ist vollständig und lässt sich zu einem Programm compilieren
- Allerdings benötigt man dazu einen Compiler
- Oracel bietet unter dem Kürzel JDK (Java Development Kit) alles nötige
  - Compiler
  - JRE
  - API
- http://www.oracle.com/technetwork/java/javase/downloads/index.html

Hello Word

- Mit dem JDK kann das Compilieren starten
  1. Man legt sich in einem beliebigen Verzeichnis die Datei „Program.java“ an und kopiert den Beispiel-Code in diese
  2. Start der Eingabeaufforderung (Windows: Start→Zubehör→ Eingabeaufforderung)
  3. Hier wechselt man in das eben angelegt Verzeichnis („cd ...“)
  4. Der Java-Compiler aus dem JDK hört auf dem Namen „javac“:

```
javac Program.java
```

  5. Ein Blick in das Verzeichnis verrät, dass eine neue Datei namens „Program.class“ angelegt wurde
  6. Diese lässt sich nun auf der virtuellen Maschine ausführen:

```
java Program
```

  7. Der Aufruf liefert das ersehnte:

```
Hello World
```