# Einführung in die Programmierung

**Dozent:** Prof. Dr. Michael Eichberg
**Kontakt:** michael.eichberg@dhbw-mannheim.de, Raum 149B
**Version:** 1.0




**Folien:** https://delors.github.io/prog-einfuehrung/folien.de.rst.html

https://delors.github.io/prog-einfuehrung/folien.de.rst.html.pdf

**Fehler melden:**

https://github.com/Delors/delors.github.io/issues

# 1. Einführung

# Ziele

- Einfache Programme (in Java) schreiben zu können
- Grundlegende Werkzeuge für die Softwareentwicklung kennen lernen
- Die Fähigkeit zum *Computational Thinking* (d. h. Denken wie ein Computer) zu erwerben
- Die Fähigkeit zu erwerben komplexe Probleme mittels des Prinzips *divide et impera* (teile und herrsche) zu lösen
- Ein paar grundlegende Begriffe der Softwareentwicklung kennen lernen

# AI Assistenten in der Softwareentwicklung

> ***AI accelerates software development to breakneck speeds, but measuring that is tricky***
>
> *[...]*
>
> *AI-assisted development is now the norm - 78% of survey respondents currently use AI in software development or plan to in the next two years, up from 64% in 2023.*
>
> *[...]*
>
> *Bringing in AI may be accelerating software development toward blinding speeds. Stunningly, most executives (69%) indicate they are shipping software twice as fast as last year.*
>
> *[...]*
>
> —*Juli 2024 - ZDNET*

# Auswirkungen von generativer KI

***KI: Kannibalisieren sich die Programmierer?***

*Viele Branchen und Berufe wurden in den letzten Jahrzehnten durch IT umgekrempelt. Laut McKinsey und IDC trifft es wegen generativer KI jetzt die IT-Branche.*

*Laut McKinsey sollen in den kommenden drei Jahren bis zu 250 Milliarden US-Dollar in generative KI (GenAI) [...] investiert werden. Das betrifft vor allem die Codeerstellung, den IT-Helpdesk und die Testautomatisierung. [...]*
*Damit [GenAI] ließe sich die Produktivität der Softwareentwickler um bis zu 50 Prozent steigern. Das betrifft vor allem die Dokumentation, die Anpassung an neue Anforderungen und Refactoring. Das wird dazu führen, dass mehr Eigenentwicklung betrieben wird und weniger Standardsoftware zum Einsatz kommt. Dieser Trend werde dadurch verstärkt, dass Citizen-Developer demnächst nicht mehr Low-Code/No-Code nutzen, sondern nur noch in natürlicher Sprache und GenAI programmieren. [...]*

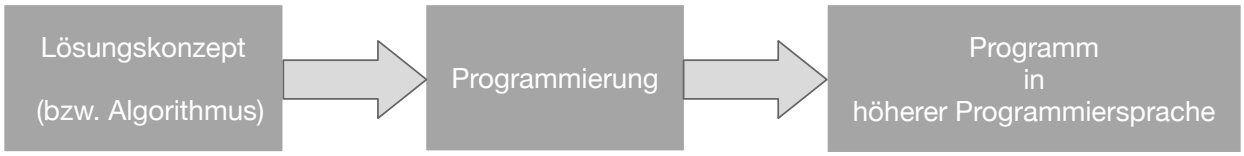
*—August 2024, Heise*

# AI Assistenten und Grundlagen der Programmierung

- Verwenden Sie keine AI Assistenten, um die Konzepte einer Programmiersprache oder Bibliothek zu erlernen.
- Später müssen Sie in der Lage sein, den Code, der von Assistenten generiert wurde, zu verstehen und zu validieren. Ohne ein tiefgreifendes Verständnis ist dies nicht möglich.
- Die Aufgaben werden immer nur Dinge verlangen, die gelehrt wurden.
- In der Klausur/Prüfung steht Ihnen auch kein AI Assistent zur Verfügung.

# Was ist Programmieren?

Programmieren bezeichnet das Formulieren eines Lösungskonzeptes (Algorithmus) in einer Programmiersprache.




**Hinweis**

Ohne Programmierkenntnisse ist es unmöglich zu beurteilen, wie komplex eine Aufgabenstellung ist.

# Programmiersprachen

## Tiobe Index für August 2024

| Programmiersprache | Anteil |
|---|---|
| Python | 18.04% |
| C++ | 10.04% |
| C | 9.17% |
| Java | 9.16% |
| C# | 6.39% |
| JavaScript | 3.91% |
| SQL | 2.21% |
| Visual Basic | 2.18% |
| Go | 2.03% |
| Fortran | 1.79% |

## Pypl Index für August 2024

| Programmiersprache | Anteil |
|---|---|
| Python | 29.6% |
| Java | 15.51% |
| JavaScript | 8.38% |
| C# | 6.7% |
| C/C++ | 6.31% |
| R | 4.6% |
| PHP | 4.35% |
| TypeScript | 2.93% |
| Swift | 2.76% |
| Rust | 2.58% |

# Was ist ein Algorithmus?

**Definition**

Ein Algorithmus ist eine exakte, endliche Vorschrift zur schrittweisen Lösung eines (lösbaren) Problems.

Eine Algorithmus erfolgt mit Hilfe eines wohl definierten Formalismus und ggf. mit Hilfe weiterer (elementarer) Algorithmen.



Es ist zum Beispiel nicht möglich alle reellen Zahlen aufzuzählen - das Problem ist nicht lösbar und es kann kein Algorithmus angegeben werden!

# Beispiel: Berechnung der Fakultät (rekursiv)

## Mathematisch

$$\begin{aligned}\text{Input} &: \text{natürliche Zahl (inkl. } 0) \\ \text{Output} &: \text{natürliche Zahl}\end{aligned}$$

$$\text{fak}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{fak}(n-1) & \text{if } n > 0 \end{cases}$$

## Java Programm bzw. Skript

```java
long fak(long n){
  if (n == 0) return 1;
  else return n * fak(n-1);
}
```

# Zentrale Eigenschaften von Algorithmen

**Allgemeinheit:**

Ein Algorithmus löst eine Klasse von Problemen. Die konkrete Instanz wird über die Eingabeparameter/Parametrisierung festgelegt.

**Endlichkeit:**

Die Beschreibung des Algorithmus bzgl. der Verarbeitungsschritte und (Eingabe-)Daten ist endlich und ermöglicht eine Ausführung mit endlichen Ressourcen.

**Determiniertheit:**

Der Algorithmus führt bei gleichen Eingaben immer zu gleichen Ausgaben.

**Ausführbarkeit:**

Der Algorithmus besteht aus einer Folge von elementaren (ausführbaren) Schritte. Diese werden vom Prozessor ausgeführt. Elementare Operationen sind z. B.:

- einfache arithmetische Operationen wie Addition, Subtraktion, Division etc.
- Vergleiche
- Zuweisungen
- etc.

**Terminiertheit:**

Das Ergebnis liegt nach endlich vielen Schritten vor. (Dies bedeutet aber nicht, dass das Problem auch praktisch lösbar ist.)

**Komplexität:**

Zeit und Platzbedarf sind endlich und in einem gewissen Rahmen abschätzbar.

Normalerweise versucht man den besten, schlechtesten und durchschnittlichen Fall zu bestimmen in Abhängigkeit von der Eingabegröße. (Insbesondere Thema des nächsten Semesters).

Diese Angaben erfolgen unabhängig von einer konkreten Implementierung bzw. Verwendung einer bestimmten Programmiersprache oder Hardware.

**Determinismus:**

Jeder (Teil-)schritt führt bei gleichen Eingaben immer zu gleichen Ausgaben.



Nicht jeder Algorithmus, der die Eigenschaft der **Determiniertheit** erfüllt, ist auch deterministisch. Bei einem deterministischen Algorithmus führt jeder (Teil-)schritt bei gleichen Eingaben immer zu gleichen Ausgaben, aber dies muss (in bestimmten Fällen) nicht immer erfüllt sein und der Algorithmus kann dennoch determiniert sein.

Insbesondere im Bereich der Kryptographie basieren viele Algorithmen darauf, dass die Ver-/Entschlüsselung nur dann effizient durchführbar ist, wenn man den Schlüssel kennt. Ist der Schlüssel nicht bekannt, dann kann immer noch ein **terminierender Algorithmus** angegeben werden, der verschlüsselte Daten entschlüsselt, aber dieser ist nicht effizient in sinnvoller Zeit durchführbar.

Die **Komplexität eines Algorithmus** bestimmt ganz maßgeblich wofür dieser Algorithmus eingesetzt werden kann. Wir können zum Beispiel Algorithmen wie folgt unterscheiden:

- konstante Komplexität (d. h. der Algorithmus benötigt unabhängig von der Größe der Eingabe immer gleich lange.)
- logarithmische Komplexität
- lineare Komplexität
- quadratische Komplexität
- exponentielle Komplexität (d. h. praktisch nicht anwendbar; häufig sucht man nach alternativen Algorithmen, die auf Heuristiken basieren. Zum Beispiel für das Erfüllbarkeitsproblem (🇺🇸 *Satisfiability*) in der Aussagenlogik.)

# Beispiel: Berechnung bzw. Approximation von $e$

## Mathematisch (exakt)

$$\text{Output : reelle Zahl}$$

$$e = \sum_{i=0}^{\infty} \frac{1}{i!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \dots$$

## Java Programm bzw. Skript

```
BigDecimal e(int steps) {
  BigDecimal e = BigDecimal.ZERO;
  while (steps >= 0) {
    e = e.add(
        BigDecimal.ONE.divide(
          new BigDecimal(fak(steps)
        ),
        MathContext.DECIMAL128));
    steps--;
  }
  return e;
}
```



$e$ steht hier für die eulersche Zahl.

In diesem Fall wurde folgende Implementierung der Fakultät verwendet:

```
import java.math.BigInteger;

BigInteger fak(int n) {
  if (n == 0)
    return BigInteger.valueOf(1);
  else {
    var bn = BigInteger.valueOf(n);
    return fak(n-1).multiply(bn);
  }
}
```

**Hinweis**

Diese Implementierung kann mit größeren Werten für **n** aufgrund der Art der Implementierung nicht umgehen. Die Details werden wir später besprechen.

# Übung: Primzahltest

- Schreiben Sie natürlich-sprachlich einen Algorithmus der eine beliebige natürliche Zahl testet ob diese eine Primzahl ist.

  Achten Sie darauf, dass der Algorithmus die vorher diskutierten Eigenschaften selbiger erfüllt.

- Beschreiben Sie die Komplexität Ihres Algorithmus.

# (Programmier-)sprachen - Unterteilung

**natürliche Sprachen:**

Dienen der Kommunikation zwischen Menschen und sind häufig mehrdeutig. In vielen Fällen ist die Bedeutung eines Satzes abhängig vom Kontext.

**formale Sprachen:**

Dienen der eindeutigen Beschreibung von Sachverhalten; sind präzise und eindeutig.

Können ggf. automatisch ausgewertet werden.

Programmiersprachen sind formale Sprachen zur Beschreibung von Algorithmen.

# Syntax und Semantik von formalen Sprachen

**Syntax:** definiert welche Sätze in der Sprache gültig sind. Die Syntax wird durch eine Grammatik formal und präzise beschrieben.

**Semantik:** definiert die Bedeutung der Sätze; wenn dies möglich ist. Nicht jeder syntaktisch korrekte Satz hat eine Bedeutung.
Häufig wird die Semantik „nur" in einem Standard oder „sogar nur" in durch eine Implementierung festgelegt.

# Formale Sprachen: Beispiel in EBNF

**Syntaktisch gültiger Satz**

Sie geht nach Hause.

1

**Syntaktisch gültig, aber semantisch falscher Satz**

Tim schwimmt auf den Mond.

2

**Syntaktisch ungültige Sätze**

Sie fährt nach Hause in die Schule.

Tim geht in die Schule

3

```
Satz = Subjekt Prädikat Objekt "."
Subjekt = "Tim" | "Sie"
Prädikat = "geht" | "fährt" | "schwimmt" | "fliegt"
Objekt = "nach Hause" | "in die Schule" | "auf den Mond"
```

16

Es gibt zahlreiche Varianten der EBNF (Extended Backus-Naur Form). Die grundlegenden Ideen und Konzepte sind jedoch überall gleich.

Beispiele für verschiedene Fehler in Java Programmen

```
int fak(long n){
  if (n == 0)
    return 1l // ';' expected
              // ⇒ "Syntaktischer Fehler"
  else
    return n * fak(n-1);
              // incompatible types: possible lossy conversion from long to int
              // ⇒ "Semantischer Fehler"
}
```

# Extended-Backus-Naur-Form (EBNF)

Die EBNF dient der Beschreibung kontext-freier Grammatiken.

| Verwendung | Notation | Bedeutung |
|---|---|---|
| Definition | = | |
| Konkatenation | , | |
| Terminierung | ; | Ende der Def. |
| Alternative | \| | |
| Optional | [ ... ] | 0 oder 1mal |
| Wiederholung | { ... } | 0 oder mehrfach |
| Gruppierung | ( ... ) | |
| Kommentar | (* ... *) | |
| Terminalsymbol | "Terminal" | |

## Beispiel

```
Ausdruck =
    Ziffer,
    {   ("+" | "-"),
        Ziffer };
Ziffer =
    "1" | "2" | "3" |
    "4" | "5" | "6" |
    "7" | "8" | "9" |
    "0";
```



Die Beschreibung einer Programmiersprache in EBNF besteht aus einer Startregel und einer Menge von weiteren Regeln sowie Terminalen, die die Syntax der Sprache beschreiben. Die Terminalen sind die Basiswörter der Sprache („reservierte Wörter”).

Auf der linken Seite einer Regel kommt genau ein Nichtterminal vor, auf der rechten Seite können beliebig viele Nichtterminale und Terminale vorkommen.

# Übung: EBNF verwenden

## Binärzahlen in Java

Folgend wird die Syntax von Binärzahlen (`BinaryNumeral`) in Java beschrieben.[1]

```
BinaryNumeral = "0", ("b" | "B"), BinaryDigits
BinaryDigits = BinaryDigit | (BinaryDigit [BinaryDigitsAndUnderscores] BinaryDigit)
BinaryDigit = "0" | "1"
BinaryDigitsAndUnderscores = BinaryDigitOrUnderscore {BinaryDigitOrUnderscore}
BinaryDigitOrUnderscore = BinaryDigit | "_"
```

Welche der folgenden Zahlen sind gültige Binärzahlen in Java?

```
0b1010                  0B1010_0
0b1_0_1_0               0B1010_0_
0B_1010                 0b1010_0_1
0b1010_                 0_b101
```

[1] Die von der JLS (Java Language Specification) verwendete Syntax wurde hier adaptiert an die zuvor gezeigte Variante. Weitere Änderungen wurden nicht vorgenommen.

## Binärzahlen in Java

Folgend wird die Syntax von Binärzahlen (`BinaryNumeral`) in Java beschrieben.[1]

```
BinaryNumeral = "0", ("b" | "B"), BinaryDigits
BinaryDigits = BinaryDigit | (BinaryDigit [BinaryDigitsAndUnderscores] BinaryDigit)
BinaryDigit = "0" | "1"
BinaryDigitsAndUnderscores = BinaryDigitOrUnderscore {BinaryDigitOrUnderscore}
BinaryDigitOrUnderscore = BinaryDigit | "_"
```

Welche der folgenden Zahlen sind gültige Binärzahlen in Java?

```
0b1010                  0B1010_0
0b1_0_1_0               0B1010_0_
0B_1010                 0b1010_0_1
0b1010_                 0_b101
```

# Übung: EBNF erweitern

## EBNF für einfache Ausdrücke

Erweitern Sie die EBNF für mathematische Ausdrücke, um die Möglichkeit Zahlen beliebiger Länge anzugeben und auch Ausdrücke (mathematisch korrekt) zu klammern. D. h. Ihre erweiterte Grammatik soll folgende Ausdrücke zulassen:

$$12 + 25$$
$$13 - 4 - (4 + 5)$$

## Bonus

Erweitern Sie die EBNF so, dass auch einfache Fließkommazahlen erlaubt sind (z. B. `1,2` oder `0,999`). Achten Sie darauf, dass keine ungültigen Zahlen wie `1,` oder `1,2,3` erlaubt sind.

## EBNF für einfache Ausdrücke

Erweitern Sie die EBNF für mathematische Ausdrücke, um die Möglichkeit Zahlen beliebiger Länge anzugeben und auch Ausdrücke (mathematisch korrekt) zu klammern. D. h. Ihre erweiterte Grammatik soll folgende Ausdrücke zulassen:

$$12 + 25$$
$$13 - 4 - (4 + 5)$$

## Bonus

Erweitern Sie die EBNF so, dass auch einfache Fließkommazahlen erlaubt sind (z. B. `1,2` oder `0,999`). Achten Sie darauf, dass keine ungültigen Zahlen wie `1,` oder `1,2,3` erlaubt sind.

# Voraussetzungen

Zu installieren (für den Anfang):

- (mind.) Java 23 JDK (Java Development Kit)

  https://adoptium.net/en-GB/temurin/releases/?version=23

  oder

  https://www.azul.com/downloads/?package=jdk#zulu
- Visual Studio Code inkl. Java Tools (oder Eclipse Theia)