Einführung in die Programmierung für NF

Rückgabewerte, EBNF, Fallunterscheidung, Schleifen

FUNKTIONEN UND PROZEDUREN

Funktion und Prozedur

- Methoden bestehen aus Funktionen (Ausdrücken) und Prozeduren (Anweisungen)

Funktion

- Funktionen beschreiben mathematische Algorithmen
- Eine Funktion kann auf viele Werte Angewendet werden

Funktion - Beispiel

Kreiszylinder

- Mathematische Funktion
  volumen(r,h) = $r^2 * \pi * h$

- Methode

```
public double zylVolume (double radius,
double high){
   return radius * radius * Math.PI * high;
}
```

- Methodenaufruf

```
zylVolume (1.5, 1.0);
```
-> 4,71

Funktion

- Null, einer oder mehrere Parameter
- Der Funktion ist der **Ergebnistyp** vorangestellt, d.h. der Typ der Werte die als Resultat geliefert werden
- Der Rumpf einer Funktion ist ein Ausdruck in dem die Parameter vorkommen
- Das Ergebnis wird durch **return** gekennzeichnet
- Beim Aufruf müssen so viele Argumente, wie die Funktion Parameter hat, vorhanden sein

Prozedur

- Prozeduren sind Methoden, die keine Ergebnisse berechnen
- Prozeduren lösen Aktionen aus

Prozedur - Beispiel

Die Methode soll einen Punkt drucken

- Methode

```
public static void paintPoint () {
   System.out.print (".");
}
```

- Methodenaufruf

```
paintPoint();
```

Prozedur

- Prozeduren sind wie Funktionen, sie geben aber kein Ergebnis
- Da die Prozedur kein Ergebniss hat wird ihr der Pseudo-Typ **void** vorangestellt
- Im Rupf steht **kein** return

BACKUS-NAUR-FORM

Aufbau von Programmen

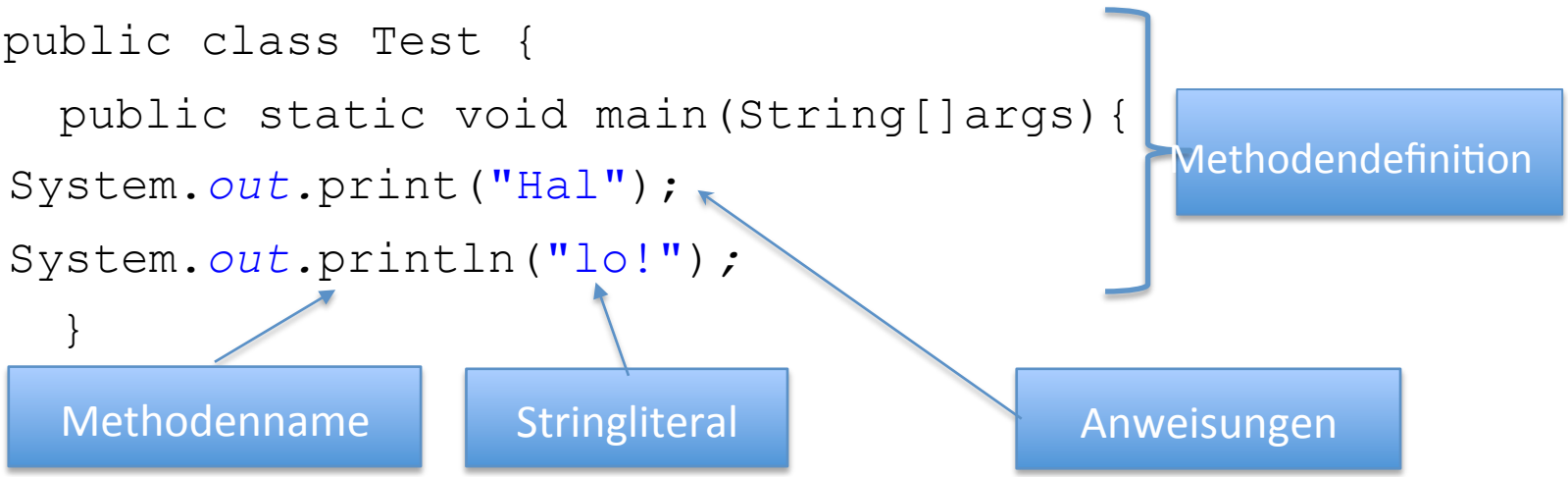
- Java-Programme sind Texte, die sich wie natürlich-sprachliche Texte in verschiedenartige Einzelteile zerlegen lassen.
- Natürliche Sprache:

  Ein Programmierer schreibt ein Programm.

  Subjekt Prädikat Objekt

- Java-Programm:

```
public class Test {
  public static void main(String[]args){
System.out.print("Hal");
System.out.println("lo!");
  }
```



Syntax von Programmiersprachen

- Der Aufbau von Programmen ist durch die *Syntax* der Programmiersprache festgelegt.

Syntax einer Programmiersprache:

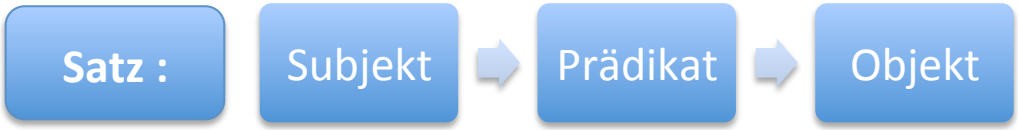
System von Regeln, das festlegt,

- *aus welchen Einzelteilen Programme bestehen können,*
- *wie man diese Einzelteile textuell aufschreibt und*
- *wie man sie zu einem Programm kombinieren kann.*

- Im Unterschied zu natürlicher Sprache ist die Syntax von Programmiersprachen vollständig und präzise festgelegt.

Syntax von Programmiersprachen

- Beispiel für eine Grammatikregel:
  Als Syntaxdiagramm:



  In BNF-Form: *Satz = Subjekt Prädikat Objekt*

- Das Syntaxdiagramm und die BNF *(Backus-Naur-Form)* erlauben das Bilden folgender syntaktisch korrekter Sätze:
  - *Ein Programmierer schreibt ein Programm.*
    - *Ein Programmierer* Subjekt
    - *schreibt* Prädikat
    - *ein Programm* Objekt
  - *Ein Programm schreibt einen Programmierer.*

    Dieser Satz ist zwar Unsinn, aber syntaktisch korrekt.

Syntax von Programmiersprachen

- Die BNF ist eine Notation für Grammatiken, die vor allem für die Beschreibung von Programmiersprachen verwendet wird.
- Heute ist die BNF (in bestimmten Varianten) die Standardbeschreibungstechnik für Programmier-sprachen und andere strukturierte Texte.
- Wir verwenden die „Erweiterte Backus-Naur-Form“ EBNF.
- Auch die Syntax von Java ist in der Backus-Naur-Form beschrieben.

Formale Grammatiken allgemein

- In formalen Grammatiken kommen zwei Arten von Symbolen vor:
- **Nichtterminalsymbole** sind Symbole, welche die verschiedenen syntaktischen Kategorien einer Sprache bezeichnen.
  Beispiele: *Subjekt, Prädikat, Objekt*
- **Terminalsymbole** (geschrieben in Anführungszeichen) stehen für die Symbole, die am Ende tatsächlich im Text vorkommen.
  Beispiele: "a", "b", "0"

Nichtterminalsymbole sind wie Platzhalter für Lücken im Text, die noch gefüllt werden müssen.

Subjekt " " "s" "c" "h" "r" "e" "i" "b" "t" " " *Objekt*

(*Subjekt* und *Objekt* müssen noch durch tatsächlichen Text ersetzt werden.)

Wir sind immer noch beim Satz:
Ein Programmierer schreibt ein Programm

Formale Grammatiken allgemein

Formale Grammatiken bestehen aus Regeln, die festlegen, was für die verschiedenen Platzhalter eingesetzt werden kann.

Beispiel: Die Regel

Satz = Subjekt Prädikat Objekt

sagt, dass wir einen *Satz-Platzhalter ersetzen können durch drei* aufeinander folgende neue Platzhalter: *Subjekt Prädikat Objekt.*

Danach werden diese drei Nichtterminalsymbole entsprechend der Regeln der Grammatik weiter ersetzt (d.h. die Lücken gefüllt), bis kein Nichtterminalsymbol mehr im Text vorkommt.

Auf diese Art erhält man einen grammatisch korrekten Text.

Subjekt Prädikat Objekt

„Ein Programmierer“ Prädikat Objekt

„Ein Programmierer“ „schreibt“ Objekt

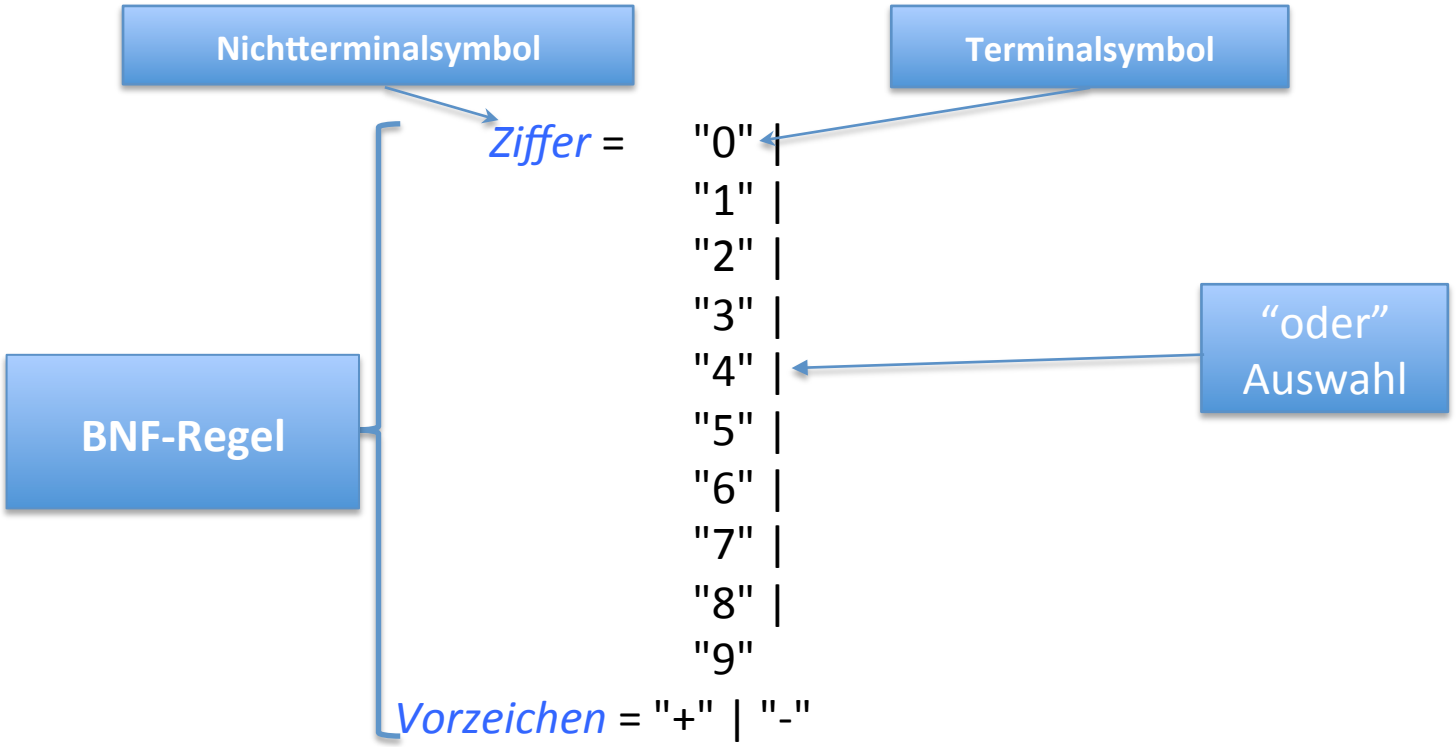
„Ein Programmierer“ „schreibt“ „ein Programm“

Syntax
Subjekt: „Ein Programmierer“
Prädikat: „schreibt“
Objekt: „ein Programm“

Backus-Naur-Form

- Beispiele für die Beschreibung von Teilen der Syntax einer Programmiersprache durch Grammatikregeln in BNF-Form.

- BNF-Regeln für Ziffern und Vorzeichen



Backus-Naur-Form

- BNF-Regeln für ganze Zahlen (Integer)
- Die textuelle Beschreibung einer ganzen Zahl besteht aus einer nichtleeren Folge von Ziffern, evtl. mit einem vorangestellten Vorzeichen.
- Nichtleere Folgen von Ziffern können durch eine BNF-Regel für ein Nichtterminalsymbol *Ziffern* erklärt werden.
- Idee:
  - Eine nichtleere Ziffernfolge ist entweder eine Ziffer
  - oder eine nichtleere Ziffernfolge gefolgt von einer Ziffer.
- **BNF-Regel: (benutzt Rekursion)**

  Ziffern = Ziffer |

  Ziffer Ziffern
  - Eine ganze Zahl beginnt mit einem optionalen Vorzeichen,
  - gefolgt von einer nichtleeren Ziffernfolge.
- **BNF-Regel:**

  GanzeZahl = [Vorzeichen] Ziffern

Beispiel

Eine Zahl -593 soll dargestellt werden

Regeln:

- *GanzeZahl* = [Vorzeichen] Ziffern
- *Ziffern* = Ziffer | Ziffer Ziffern
- *Vorzeichen* = "+" | "-"
- *Ziffer* = "0“ | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

Ganze Zahl

[Vorzeichen] Ziffern

- Ziffer Ziffern
  - "5" Ziffer Ziffern
  - "5" "9" Ziffer
  - "5" "9" "3"

Backus-Naur-Form

Die Backus-Naur-Form erlaubt es, die Syntax einer Programmiersprache durch **BNF-Regeln** festzulegen.

In BNF-Regeln kommen vor:

- Nichtterminalsymbole:
  - Begriffe, die durch BNF-Regeln erklärt werden.
  - Beispiele: *Ziffer, Ziffern*
- Terminalsymbole:
  - Begriffe, die so in den Programmtext übernommen werden.
  - Beispiele: "0", "1", "class"
- Operatorsymbole:
  - Terminal- und Nichtterminalsymbole werden durch Operatoren verknüpft.

Backus-Naur-Form

- Jede **BNF-Regel** hat die Form ***A = <Ausdruck>*** *wobei*
  - *A ist ein Nichtterminalsymbol*
  - Ein Ausdruck ist entweder
    - ein Terminalsymbol; oder
    - ein Nichtterminalsymbol; oder
    - ein zusammengesetzter Ausdruck.
- Aus gegebenen Ausdrücken E, E1 und E2 können zusammensetzte Ausdrücke durch folgende Operationen gebildet werden:
  - Auswahl E1 | E2 („E1 oder E2“)
  - Sequentielle Komposition E1 E2 (hintereinander schreiben)
    („E2 folgt direkt auf E1“)
  - Option [E] („E wird optional gemacht“)

Abgeleitete Operatoren

- Eine **Folge von Ausdrücken *gleicher* Bauart** lässt sich wie folgt definieren.
- E sei ein Ausdruck.
- {E} bedeutet: E kann 0-mal oder mehrmals vorkommen.
- Man kann {E} als unendliche Auswahl verstehen:

  {E} = [E] | [E] [E] | [E] [E] [E] | ...

  Allerdings sind unendliche Regeln **nicht** zulässig.
- Dennoch kann man {E} durch Rekursion definieren:

  {E} = [E] |

  [E] {E}

  (E kann beliebig, aber nur endlich, oft auftreten.)

Beispiel 2

Ist +31 eine *GanzeZahl?*

Wir bilden folgende Ableitung:

GanzeZahl	→ *(Ersetzen von GanzeZahl durch rechte Seite der Def.)*
[Vorzeichen] Ziffern	→ *(Ausführen des Operators [])*
Vorzeichen Ziffern	→ *(Ersetzen von Vorzeichen durch rechte Seite der Def.)*
("+"|"-") *Ziffern*	→ *(Ausführen der Auswahl)*
"+“ *Ziffern*	→ *(Def. von Ziffern)*
"+“ *(Ziffer | Ziffern Ziffer)*	→ *(Ausführen von |)*
"+“ *(Ziffern Ziffer)*	→ *(Def. von Ziffern)*
"+“ *((Ziffer | Ziffern Ziffer) Ziffer)*	→ *(Auswahl)*
"+" *Ziffer Ziffer*	→ *(Def. von Ziffer)*
"+“ ("0" | "1" | … | "9") *Ziffer*	→ *(mehrfache Auswahl)*
"+" "3“ *Ziffer*	→ *(Def. von Ziffer)*
"+" "3“ ("0" | "1" | … | "9")	→ (mehrfache Auswahl)
"+" **"3"** **"1"**	

Beispiel 3: Bezeichner

- Angabe:
  - Ein **Bezeichner** ist eine nichtleere Folge von Buchstaben und Ziffern, die mit einem Buchstaben beginnt.
  - Bezeichner: A, A2D2, wirsing
  - Keine Bezeichner: 23, 3x, F.D.P.

- BNF-Grammatik für Bezeichner:

  Buchstabe = "A" | "B" | … | "Z" |
  "a" | … | "z"

  Buzi = Buchstabe | Ziffer

  Bezeichner = Buchstabe {Buzi}

In Java müssen alle Variablennamen, Klassennamen usw. Bezeichner sein. Die Grammatikregel für Bezeichner ist etwas allgemeiner.

FALLUNTERSCHEIDUNG

Fallunterscheidung

- Die **Fallunterscheidung** (*Conditional*) in Java hat die Form
  - **if** (*BoolExpression*) *Statement*

  bzw.
  - **if** (*BoolExpression*) *Statement* **else** *Statement*

- **Beispiel:** Kontofluss 1

```
if (kontoStand >= betrag)
    kontoStand = kontoStand - betrag;
```

- **Beispiel:** Kontofluss 2

```
if (kontoStand >= betrag)
    kontoStand = kontoStand - betrag;
else
    kontoStand = kontoStand - betrag - UEBERZIEH_GEBUEHR;
```

Was ist hier falsch?

```
if (kontoStand >= betrag)
    gebuehren = gebuehren + TRANSAKTIONS_GEBUEHR;
    kontoStand = kontoStand - betrag;
```

Was ist hier falsch?

```
if (kontoStand >= betrag)
   gebuehren = gebuehren + TRANSAKTIONS_GEBUEHR;
   kontoStand = kontoStand - betrag;
```

Das if bezieht sich nur auf die erste Anweisung, der Betrag wird immer vom Kontostand abgezogen.

```
if (kontoStand >= betrag)
   gebuehren = gebuehren + TRANSAKTIONS_GEBUEHR;

kontoStand = kontoStand - betrag;
```

Mehrfache Anweisungen

Blockklammern sind nötig, wenn die Fallunterscheidung mehrere Anweisungen umschließen soll.

Beispiel:

```
if (kontoStand >= betrag) {
 kontoStand = kontoStand - betrag;
 gebuehren = gebuehren + ABHEBE_GEBUEHR;
}
else {
 kontoStand = kontoStand – betrag - UEBERZIEH_GEBUEHR;
 gebuehren = gebuehren + UEBERZIEH_GEBUEHR;
}
```

Dangling else

```
if (!kontoGesperrt)
 if (kontoStand >= betrag) {
    kontoStand = kontoStand - betrag;
    System.out.println("Abhebung erfolgreich.");
 }
else System.out.println("Abhebung nicht
erlaubt.");
```

Vorsicht! Das else bezieht sich auf das zweite if.

→ In Java schreibt man bei if (fast) immer Blockklammern, auch wenn der Block nur eine einzige Anweisung enthält.

SCHLEIFEN

Iteration

Drei Konstrukte zur Iteration

- while
- for
- do (behandeln wir nicht)

While-Schleifen

- Die **while**-Schleife (Nichtterminalsymbol *Iteration*) hat die Form

  while (*BoolExpression*) *Statement*

Beispiel:

```
int n = 1;
int end = 10;
while (n <= end) {
   System.out.println(n);
   n++;
 }
```

Beispiel:

```
int qs = 0;
int x = 352;
while (x > 0) {
    qs = qs + x % 10;
    x = x / 10;
}
```

for- Schleifen

- Die häufigste Form der while-Schleife ist:

```
int i = start;          // Initialisierung
while (i < end) {       // Bedingung
...
i++;                    // Zählerkorrektur durch konstante Änderung
}
```

- Abkürzende Schreibweise:

```
for (int i = start; i < end; i++) {
...
}
```

for-Schleifen

- Beispiel:

```
int end = 10;
for (int i = start; i < end; i++) {
 System.out.println(i);
}
```

- Allgemein hat eine **for**-Schleife die Gestalt

  for (Initialisierung; Bedingung; Zählerkorrektur) *Statement*

  Dabei wird zunächst die Initialisierung ausgeführt.

  Dann wird, solange die Bedingung wahr ist, *Statement* ausgeführt und der Zähler geändert (gemäß des Zählerkorrektur-*Statement*).

- Eine in der Initialisierung deklarierte Variable ist nur im Schleifenrumpf gültig.

for-Schleifen

- Guter Stil ist es, **for**-Schleifen nur folgendermaßen zu schreiben:

  for (Setze Zähler auf Startwert;

  Teste, ob Zähler den Endwert erreicht hat;

  ändere Zähler) {

  ...

  // Zähler, Startwert, Endwert und

  // Inkrement werden in diesem Block

  // nicht verändert.

  }

- Die Zählervariable sollte bei der Schleifeninitialisierung deklariert werden.

Vielen Dank für Ihre Aufmerksamkeit