LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Einführung in die Informatik:
Programmierung und Softwareentwicklung

Wintersemester 2018/19

Kapitel 4: Anweisungen und Kontrollstrukturen

Prof. Dr. David Sabel

Lehr- und Forschungseinheit für Theoretische Informatik

Institut für Informatik, LMU München

WS 2018/19

Stand der Folien: 7. November 2018

Die Inhalte dieser Folien basieren – mit freundlicher Genehmigung – tlw. auf Folien von Prof. Dr. Rolf Hennicker aus dem WS 2017/18 und auf Folien von PD Dr. Ulrich Schöpp aus dem WS 2010/11

Ziele

- **Kontrollstrukturen** in **imperativen** Programmen kennenlernen und verstehen
- Kontrollstrukturen:
  - Variablendeklarationen
  - Zuweisungen
  - Blöcke
  - Fallunterscheidung (Verzweigung)
  - Schleifen
- **Syntax** und **Semantik**
- **Realisierung** der Kontrollstrukturen in **Java**

Anweisungen

- **Anweisungen (Statements, Befehle)** sind die Grundelemente imperativer Programmiersprachen zur Festlegung (Kontrolle) des Ablaufs eines Programms.
- Wir unterscheiden folgende **grundlegende Arten** von Anweisungen:

  Syntax:

Statement	=	VariableDeclaration	(Deklarationsanweisung)
		| Assignment	(Zuweisung)
		| Block	(Block)
		| Conditional	(Fallunterscheidung)
		| Iteration	(Wiederholungsanweisung)

- Durch Ausführung einer Anweisung wird ein „alter“ Zustand in einen „neuen“ Zustand überführt (**Semantik!**)

Deklarationsanweisungen

Statement	= VariableDeclaration	(Deklarationsanweisung)
	| Assignment	(Zuweisung)
	| Block	(Block)
	| Conditional	(Fallunterscheidung)
	| Iteration	(Wiederholungsanweisung)

Deklarationsanweisungen (2)

Eine Deklarationsanweisung nennt man auch
lokale Variablendefinition

Syntax (Wiederholung):

VariableDeclaration =
Type VariableDeclarator { "," VariableDeclarator } ";"

VariableDeclarator = NamedVariable ["=" Expression]

Beispiel: `int x = 5, y = 7;`

Wirkung (Semantik):

Es wird ein Speicherplatz angelegt, auf den mit dem symbolischen **Namen** der deklarierten Variablen zugegriffen werden kann.

Dort wird ein „Default"-Wert oder der durch Expression bestimmte Initialwert gespeichert. Z.B. $[(\mathtt{x}, 5), (\mathtt{y}, 7)]$.

Deklarationsanweisungen (3)

Beachte:

Bereits deklarierte Variablen **dürfen nicht nocheinmal** deklariert werden!

Beispiel:

```
1   public class TestVarDeclTwice {
2    public static void main(String[] args) {
3      int x = 7;
4      int x = 8;
5      System.out.println(x);
6    }
7  }
```

```
javac TestVarDeclTwice.java
TestVarDeclTwice.java:4: error: variable x is already
                                defined in method main(String[])
    int x = 8;
        ^
1 error
```

Zuweisungen

Statement	= VariableDeclaration	(Deklarationsanweisung)
	| Assignment	(Zuweisung)
	| Block	(Block)
	| Conditional	(Fallunterscheidung)
	| Iteration	(Wiederholungsanweisung)

Zuweisungen (2)

Syntax:

$$\text{Assignment} = \text{Variable } \texttt{"="} \text{ Expression } \texttt{";"}$$

Wirkung:

1. Zunächst wird der Wert von Expression im „alten“ Zustand berechnet.
2. Dieser Wert wird im Nachfolgezustand der Variablen Variable als neuer Wert zugewiesen.

Beispiel: $[(\texttt{x}, 5), (\texttt{y}, 7)]$ `x = 2*x+y`

Zuweisungen (2)

Syntax:

$$\text{Assignment} = \text{Variable "="} \text{ Expression ";"}$$

Wirkung:

1. Zunächst wird der Wert von Expression im „alten“ Zustand berechnet.
2. Dieser Wert wird im Nachfolgezustand der Variablen Variable als neuer Wert zugewiesen.

Beispiel: $[(\texttt{x}, 5), (\texttt{y}, 7)]$ `x = 2*x + y ;`

`2*5+7 = 17`

Zuweisungen (2)

Syntax:

$$\text{Assignment} = \text{Variable } \texttt{"="} \text{ Expression } \texttt{";"}$$

Wirkung:

1. Zunächst wird der Wert von Expression im „alten“ Zustand berechnet.
2. Dieser Wert wird im Nachfolgezustand der Variablen Variable als neuer Wert zugewiesen.

Beispiel: $[(\texttt{x}, 5), (\texttt{y}, 7)]$ `x = 2*x + y ;` $[(\texttt{x}, 17), (\texttt{y}, 7)]$

`2*5+7 = 17`

Zuweisungen (2)

Syntax:

$$\text{Assignment} = \text{Variable "="} \text{ Expression ";"}$$

Wirkung:

1. Zunächst wird der Wert von Expression im „alten“ Zustand berechnet.
2. Dieser Wert wird im Nachfolgezustand der Variablen Variable als neuer Wert zugewiesen.

Beispiel: $[(\texttt{x}, 5), (\texttt{y}, 7)]$ `x = 2*x + y ;` $[(\texttt{x}, 17), (\texttt{y}, 7)]$

`2*5+7 = 17`

Beachte den Unterschied zwischen „=“ (Zuweisung) und „==“ (Vergleich)!

Zuweisungen (2)

Syntax:

$$\text{Assignment} = \text{Variable "="} \text{ Expression ";"}$$

Wirkung:

1. Zunächst wird der Wert von Expression im „alten“ Zustand berechnet.
2. Dieser Wert wird im Nachfolgezustand der Variablen Variable als neuer Wert zugewiesen.

Beispiel: $[(\texttt{x}, 5), (\texttt{y}, 7)]$ `x = 2*x + y ;` $[(\texttt{x}, 17), (\texttt{y}, 7)]$

`2*5+7 = 17`

Beachte den Unterschied zwischen „=“ (Zuweisung) und „==“ (Vergleich)!

Nebenbedingung für die Zuweisung Variable "=" Expression ";" :

- Der Typ der Variablen Variable muss mit dem Typ des Ausdrucks Expression verträglich sein.
- Die Variable Variable muss vorher deklariert sein.

Zuweisung: Abkürzende Schreibweisen

Abkürzungen:

`x++;`	steht für `x = x + 1;`
`x--;`	steht für `x = x - 1;`
`x operation= Ausdruck ;`	steht für `x = x operation Ausdruck;`

Beispiele:

`x += y;`	steht für `x = x + y;`
`b &= c;`	steht für `b = b & c;`
`x += 3* y;`	steht für `x = x + (3 * y);`

Blöcke

Statement	= VariableDeclaration	(Deklarationsanweisung)
	| Assignment	(Zuweisung)
	| Block	(Block)
	| Conditional	(Fallunterscheidung)
	| Iteration	(Wiederholungsanweisung)

Blöcke (2)

Ein **Block** fügt mehrere Anweisungen durch geschweifte Klammern zu einer einzigen Anweisung zusammen.

Syntax:

$$\text{Block} = \text{"\{"} \{ \text{Statement} \} \text{"\}"}$$

Wirkung:

- Die Anweisungen im Block werden in der Reihenfolge der Aufschreibung hintereinander ausgeführt.
- Der durch einen Block bewirkte Zustandsübergang erfolgt also durch Hintereinanderausführung der Zustandsübergänge der einzelnen Anweisungen.

Blöcke (3)

Beispiel:

$$[(\mathtt{x},5),(\mathtt{y},7)]\ \ \{\mathtt{x = 2*x+y;\ y = x-2;}\}\ \ [(\mathtt{x},17),(\mathtt{y},15)]$$

denn:

$$[(\mathtt{x},5),(\mathtt{y},7)]\ \ \mathtt{x = 2*x+y;}\ \ [(\mathtt{x},17),(\mathtt{y},7)]$$

und

$$[(\mathtt{x},17),(\mathtt{y},7)]\ \ \mathtt{y = x-2;}\ \ [(\mathtt{x},17),(\mathtt{y},15)]$$

Allgemein:

wenn σ_i $\quad$ Statement $_{i+1}$ $\quad$ σ_{i+1} für $i = 0, \ldots, n-1$,

dann σ_0 $\quad$ $\{$ Statement $_1; \ldots,$ Statement $_n\}$ $\quad$ σ_n

Gültigkeitsbereich

- Der **Gültigkeitsbereich** einer lokalen Variablen ist der Block, in dem die Variable deklariert wurde.
  Außerhalb dieses Blocks existiert die Variable nicht.
- Blöcke können geschachtelt werden.
- In einem untergeordneten Block sind Variable eines übergeordneten Blocks gültig und dürfen dort **nicht noch einmal** deklariert werden.

Beispiel:

```
{
    int wert = 0;
    wert = wert + 17;
    {
        int total = 100;
        wert = wert - total;
    }
    wert = 2 * wert;
}
```

Gültigkeitsbereich

- Der **Gültigkeitsbereich** einer lokalen Variablen ist der Block, in dem die Variable deklariert wurde.
  Außerhalb dieses Blocks existiert die Variable nicht.
- Blöcke können geschachtelt werden.
- In einem untergeordneten Block sind Variable eines übergeordneten Blocks gültig und dürfen dort **nicht noch einmal** deklariert werden.

Beispiel:

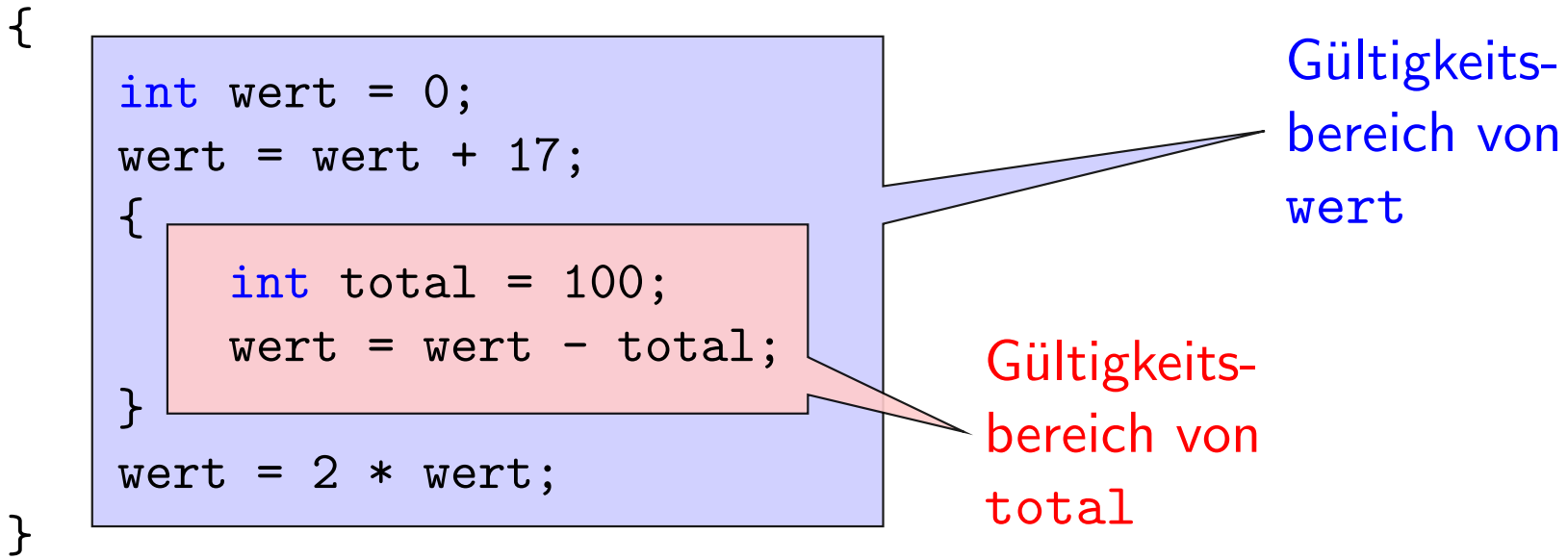
```
{
    int wert = 0;
    wert = wert + 17;
    {
        int total = 100;
        wert = wert - total;
    }
    wert = 2 * wert;
}
```

Gültigkeits-
bereich von
`wert`

Gültigkeitsbereich

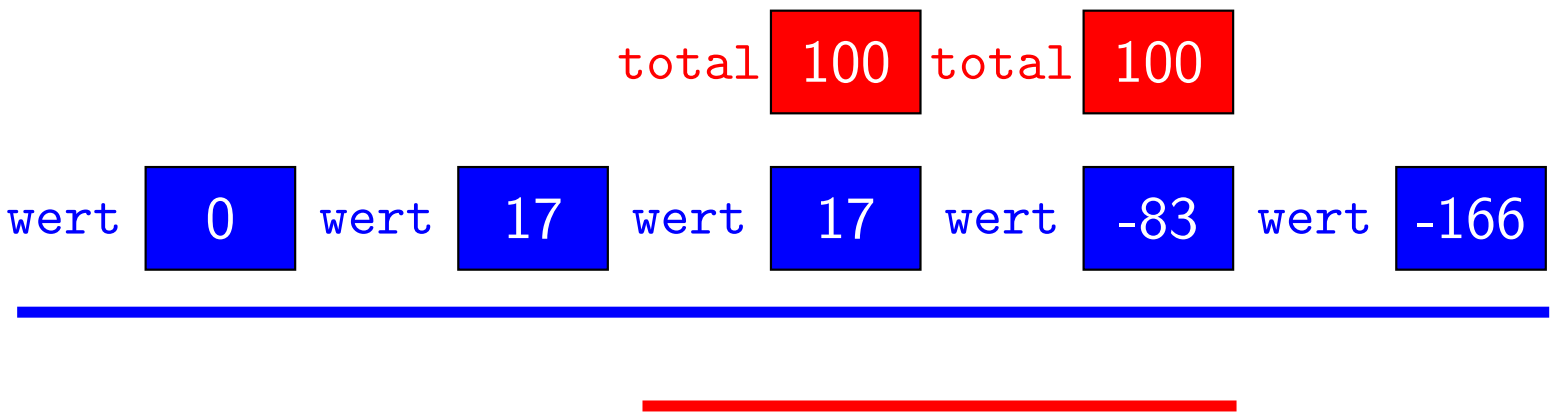
- Der **Gültigkeitsbereich** einer lokalen Variablen ist der Block, in dem die Variable deklariert wurde.
  Außerhalb dieses Blocks existiert die Variable nicht.
- Blöcke können geschachtelt werden.
- In einem untergeordneten Block sind Variable eines übergeordneten Blocks gültig und dürfen dort **nicht noch einmal** deklariert werden.

Beispiel:



Veränderung des Speichers

```
{
    int wert = 0;
    wert = wert + 17;
    {
        int total = 100;
        wert = wert - total;
    }
    wert = 2 * wert;
}
```



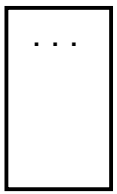
Lokale Variablen im Speicher

- Lokale Variablen werden **stapelartig** im Speicher abgelegt.
- Wird eine Variablendeklaration abgearbeitet, so wird ein neuer Speicherplatz für diese Variable oben auf den Stapel gelegt (allokiert).
- Am Ende eines Blocks werden alle Variablen (von oben) vom Stapel entfernt, die in diesem Block deklariert wurden.

Lokale Variablen im Speicher

- Lokale Variablen werden **stapelartig** im Speicher abgelegt.
- Wird eine Variablendeklaration abgearbeitet, so wird ein neuer Speicherplatz für diese Variable oben auf den Stapel gelegt (allokiert).
- Am Ende eines Blocks werden alle Variablen (von oben) vom Stapel entfernt, die in diesem Block deklariert wurden.

```
1 ...
      {
2      int y = 3;
3      double z = 2.0;
       ...
      }
4 ...
```

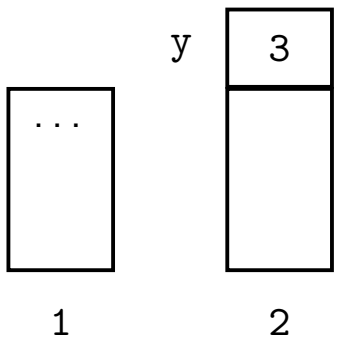


1

Lokale Variablen im Speicher

- Lokale Variablen werden **stapelartig** im Speicher abgelegt.
- Wird eine Variablendeklaration abgearbeitet, so wird ein neuer Speicherplatz für diese Variable oben auf den Stapel gelegt (allokiert).
- Am Ende eines Blocks werden alle Variablen (von oben) vom Stapel entfernt, die in diesem Block deklariert wurden.

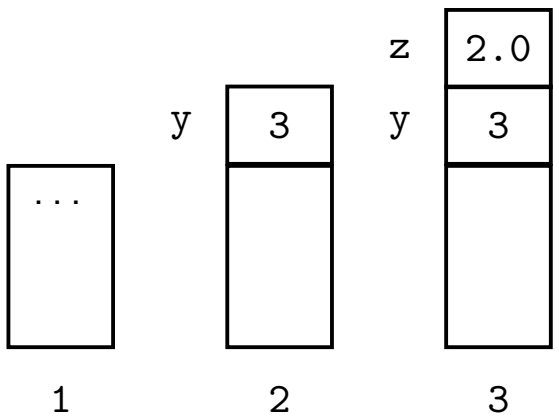
```
1 ...
      {
2       int y = 3;
3       double z = 2.0;
        ...
      }
4 ...
```



Lokale Variablen im Speicher

- Lokale Variablen werden **stapelartig** im Speicher abgelegt.
- Wird eine Variablendeklaration abgearbeitet, so wird ein neuer Speicherplatz für diese Variable oben auf den Stapel gelegt (allokiert).
- Am Ende eines Blocks werden alle Variablen (von oben) vom Stapel entfernt, die in diesem Block deklariert wurden.

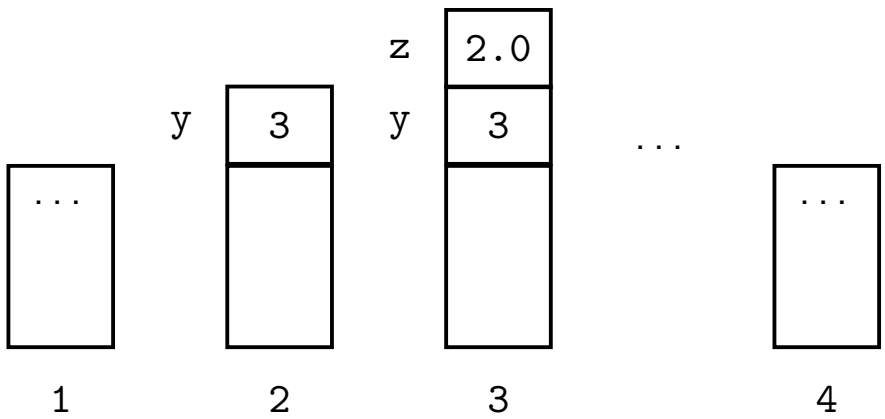
```
1 ...
      {
2      int y = 3;
3      double z = 2.0;
       ...
      }
4 ...
```



Lokale Variablen im Speicher

- Lokale Variablen werden **stapelartig** im Speicher abgelegt.
- Wird eine Variablendeklaration abgearbeitet, so wird ein neuer Speicherplatz für diese Variable oben auf den Stapel gelegt (allokiert).
- Am Ende eines Blocks werden alle Variablen (von oben) vom Stapel entfernt, die in diesem Block deklariert wurden.

```
1 ...
      {
2      int y = 3;
3      double z = 2.0;
       ...
      }
4 ...
```



Exkurs: Stack (auch Keller, Stapelspeicher) (2)

- Ein **Stack** ist eine Datenstruktur, in die Elemente eingefügt und in entgegen gesetzter Reihenfolge wieder herausgenommen werden können.

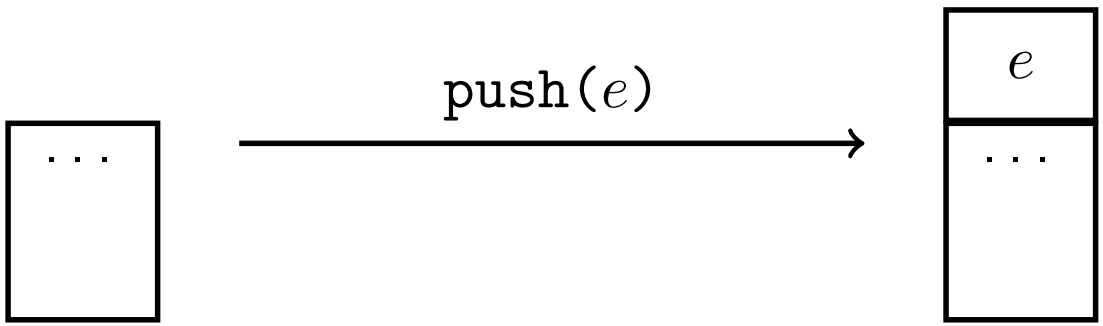
$$\text{LIFO} = \textbf{L}\text{ast } \textbf{I}\text{n, } \textbf{F}\text{irst } \textbf{O}\text{ut}$$

Exkurs: Stack (auch Keller, Stapelspeicher) (2)

- Ein **Stack** ist eine Datenstruktur, in die Elemente eingefügt und in entgegen gesetzter Reihenfolge wieder herausgenommen werden können.

$$\text{LIFO} = \textbf{L}\text{ast } \textbf{I}\text{n, } \textbf{F}\text{irst } \textbf{O}\text{ut}$$

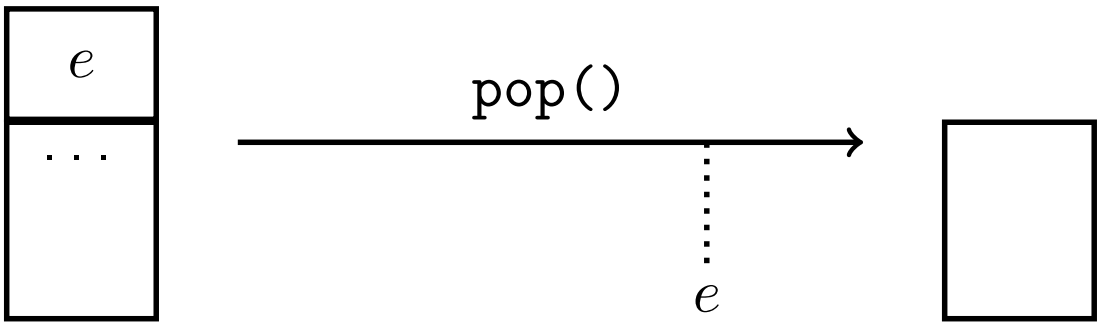
- Grundoperationen:
  - `push(`e`)`: legt das Element e oben auf den Stapel



...

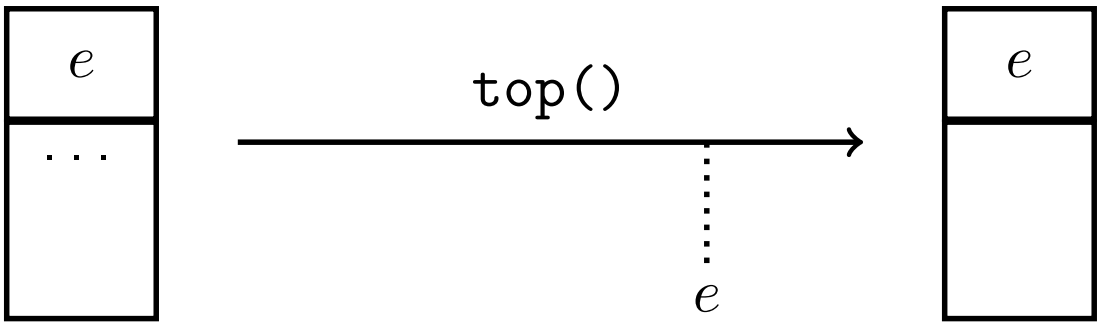
Exkurs: Stack (auch Keller, Stapelspeicher)

- `pop()`: entfernt das oberste Element
  (und liefert es als Ergebnis)



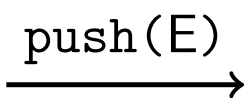
...

- `top()`: liefert das oberste Element als Ergebnis,
  ohne den Stapel zu verändern

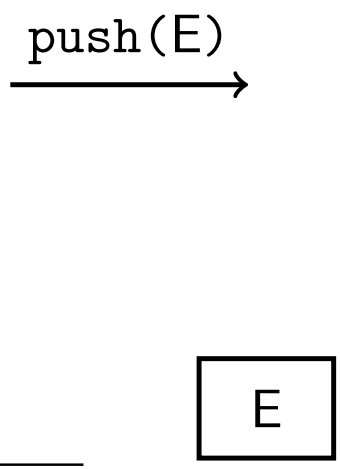


Stacks: Beispiel

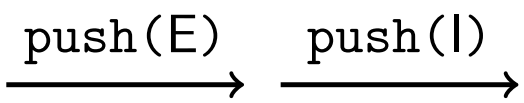
Stacks: Beispiel



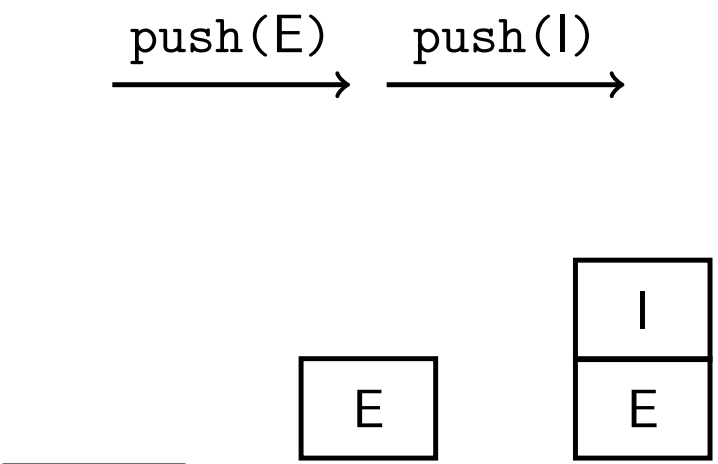
Stacks: Beispiel



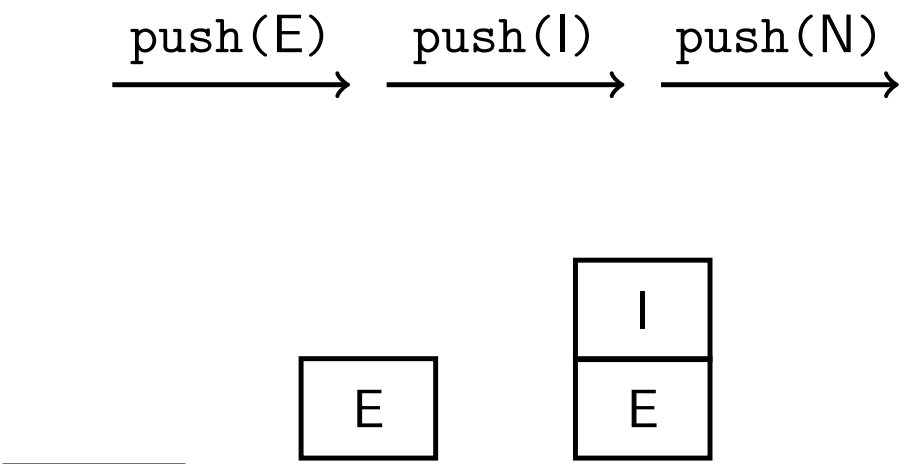
Stacks: Beispiel



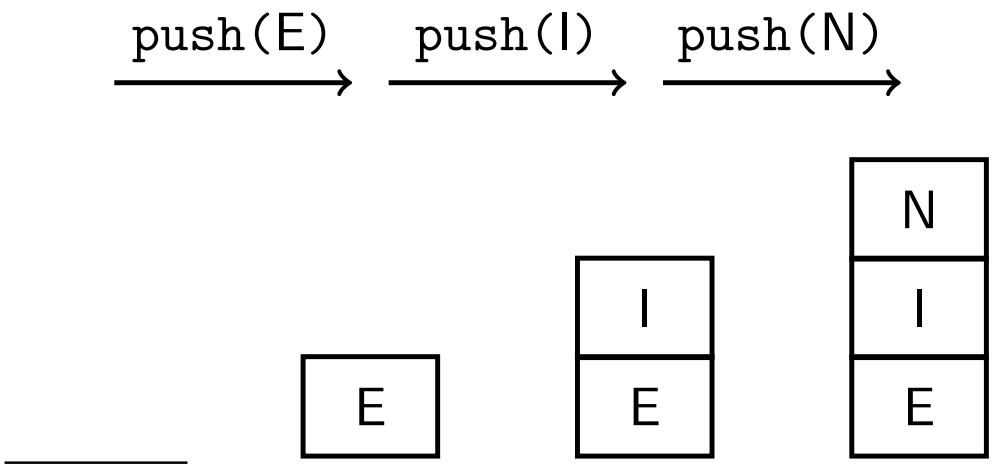
Stacks: Beispiel



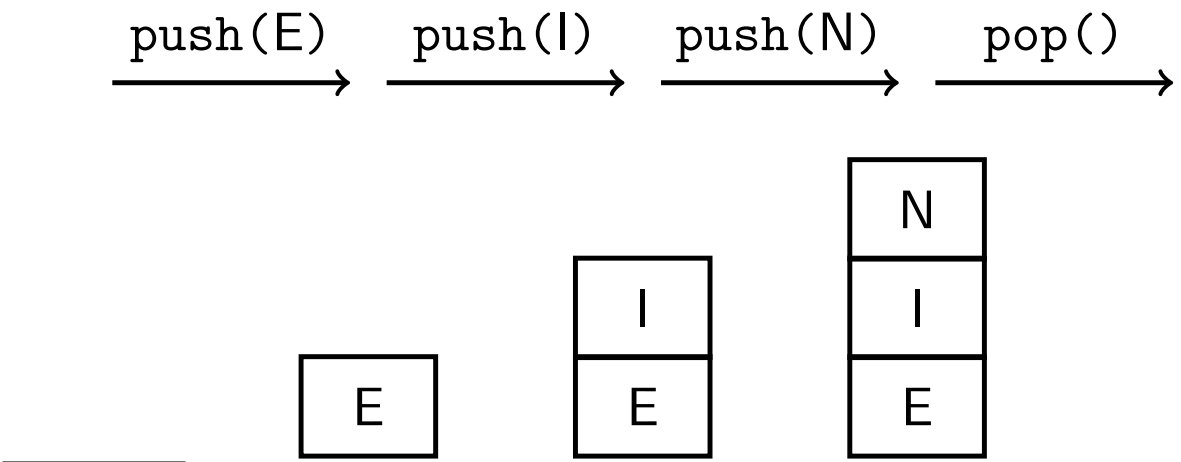
Stacks: Beispiel



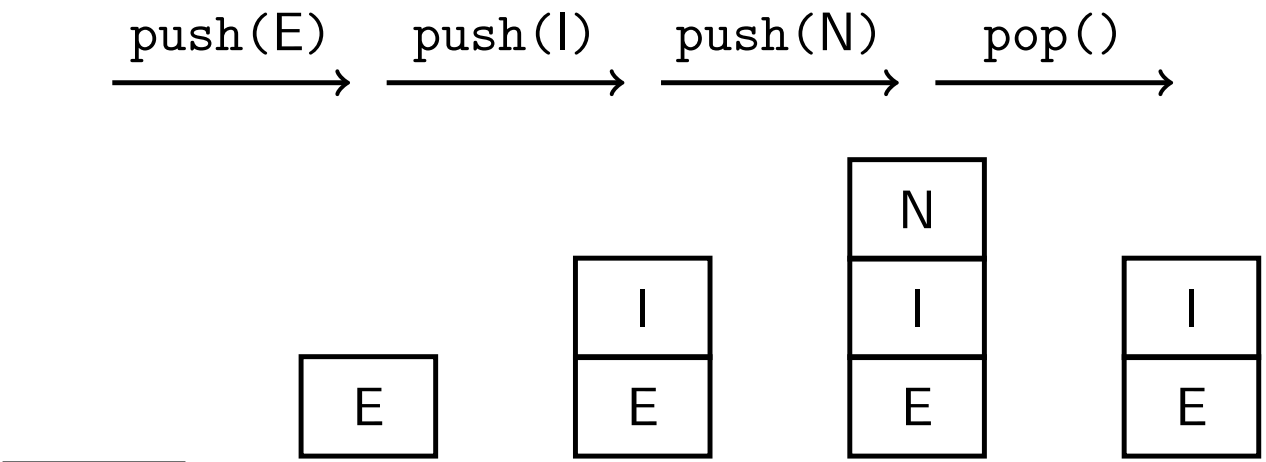
Stacks: Beispiel



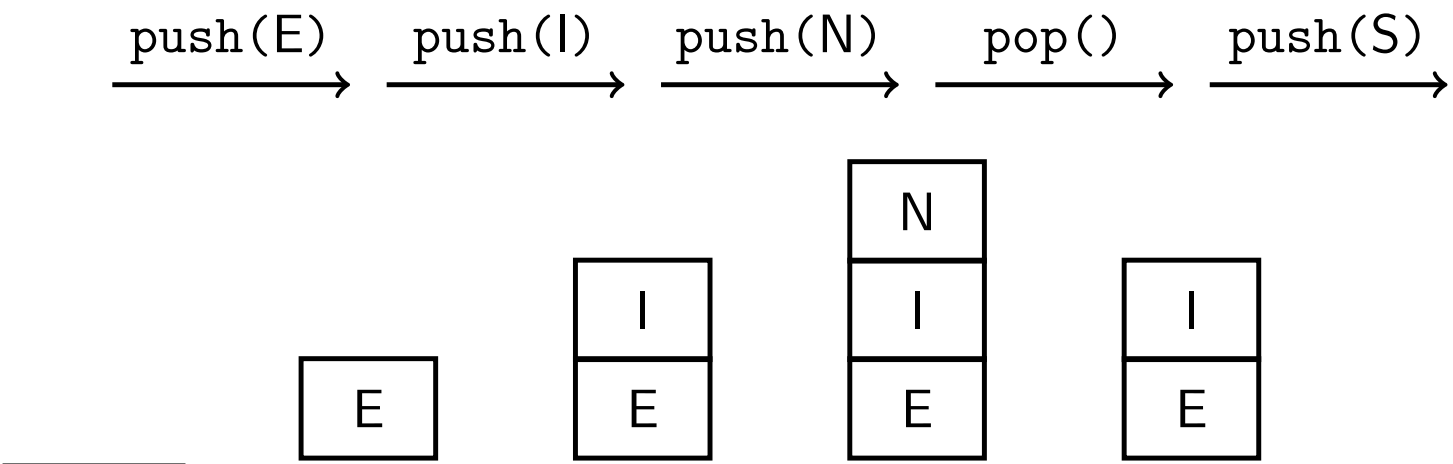
Stacks: Beispiel



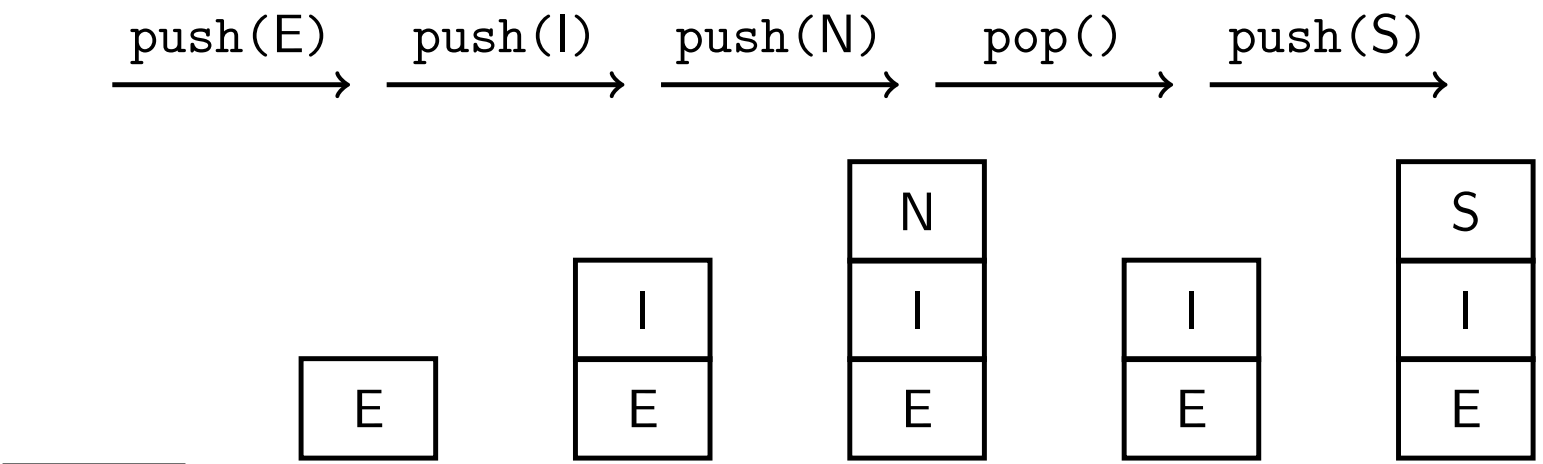
Stacks: Beispiel



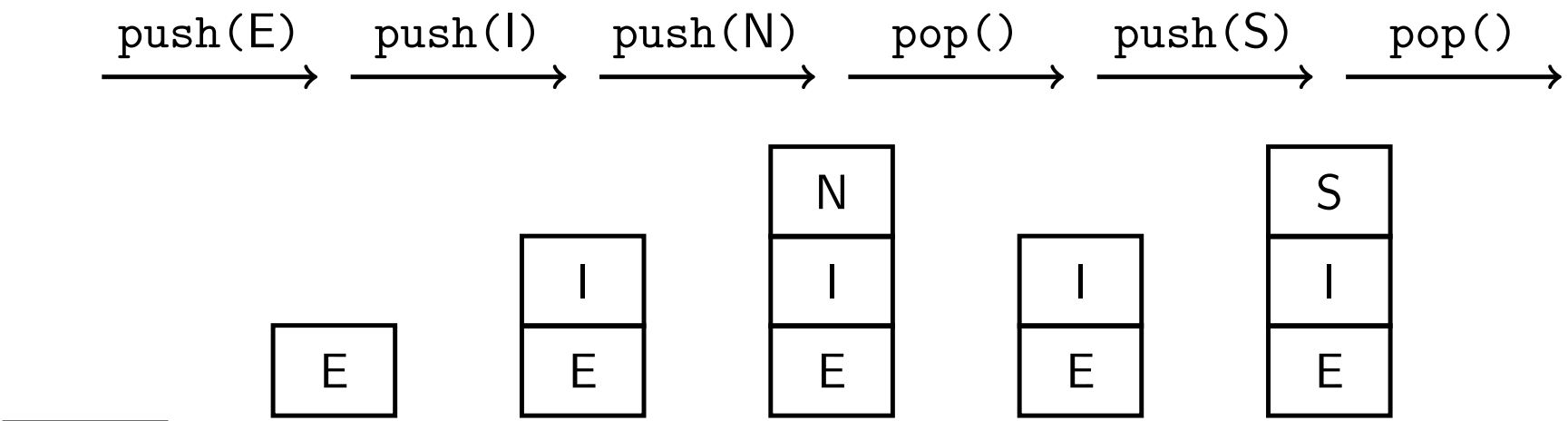
Stacks: Beispiel



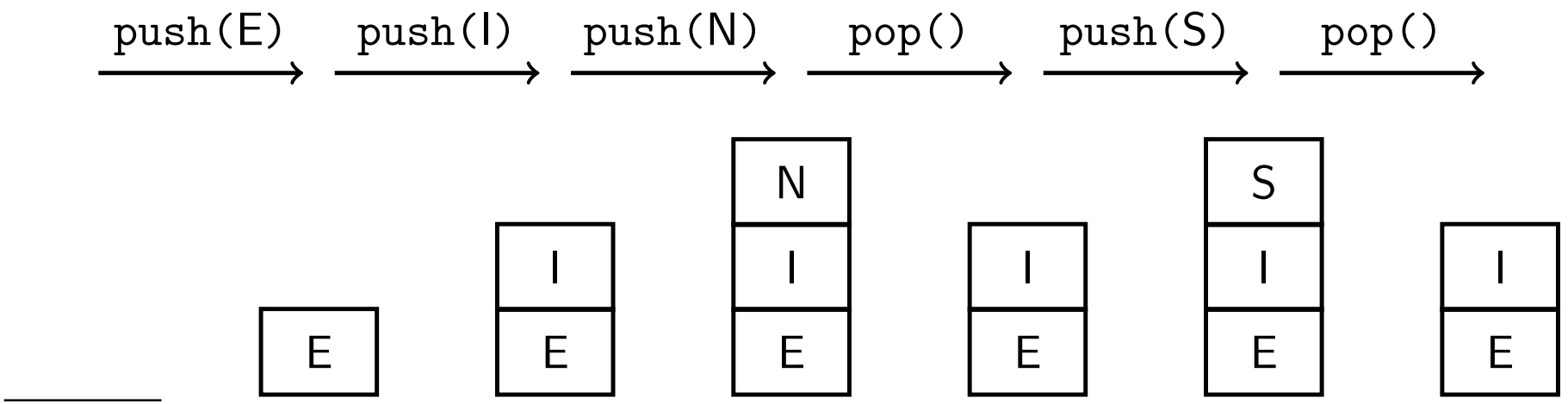
Stacks: Beispiel



Stacks: Beispiel



Stacks: Beispiel



Fallunterscheidungen (Bedingte Anweisung)

Statement	= VariableDeclaration	(Deklarationsanweisung)
	| Assignment	(Zuweisung)
	| Block	(Block)
	| Conditional	(Fallunterscheidung)
	| Iteration	(Wiederholungsanweisung)

Fallunterscheidungen (Bedingte Anweisung) (2)

Syntax:

Conditional = IfStatement | SwitchStatement
IfStatement = "if" "(" Expression ")" Statement ["else" Statement]

Beispiel

```
if (x >= 0) y = x;
else y = -x;
```

Nebenbedingung:
Der Type des Ausdrucks Expression muss `boolean` sein.

Wirkung:

- Wenn die Auswertung von Expression im aktuellen Zustand den Wert `true` ergibt, wird das erste Statement ausgeführt.
- Wenn die Auswertung von Expression im aktuellen Zustand den Wert `false` ergibt **und** ein `else`-Zweig vorhanden ist, wird das zweite Statement ausgeführt.

Fallunterscheidungen (Bedingte Anweisung) (3)

Formaler: (4 Fälle wegen `true`/`false` und mit else/ohne else)

- Wenn Expression im Zustand σ zu `true` auswertet, und σ Statement$_1$ σ', dann σ `if (` Expression `)` Statement$_1$ `else` Statement$_2$ σ'
- Wenn Expression im Zustand σ zu `false` auswertet, und σ Statement$_2$ σ', dann σ `if (` Expression `)` Statement$_1$ `else` Statement$_2$ σ'
- Wenn Expression im Zustand σ zu `true` auswertet, und σ Statement σ', dann σ `if (` Expression `)` Statement σ'
- Wenn Expression im Zustand σ zu `false` auswertet, dann σ `if (` Expression `)` Statement σ

Beispiele

Beispiel 1:

```
if (kontostand >= betrag)
    kontostand = kontostand
                 - betrag
                 - abhebe_gebuehr;
```

Beispiel 2:

```
if (kontostand >= betrag)
    kontostand = kontostand
                 - betrag
                 - abhebe_gebuehr;
else
    kontostand = kontostand
                 - betrag
                 - abhebe_gebuehr
                 - ueberzieh_gebuehr;
```

Was ist hier semantisch falsch?

```
if (kontostand >= betrag)
    kontostand = kontostand - betrag - abhebe_gebuehr
else
    kontostand = kontostand - betrag;
    kontostand = kontostand - abhebe_gebuehr
                            - ueberzieh_gebuehr;
```

Was ist hier semantisch falsch?

```
if (kontostand >= betrag)
    kontostand = kontostand - betrag - abhebe_gebuehr
else
    kontostand = kontostand - betrag;
    kontostand = kontostand - abhebe_gebuehr
                            - ueberzieh_gebuehr;
```

Die letzte Zeile wird immer ausgeführt! Sie gehört nicht mehr zum `else`-Zweig.

Was ist hier semantisch falsch?

```
if (kontostand >= betrag)
    kontostand = kontostand - betrag - abhebe_gebuehr
else
    kontostand = kontostand - betrag;
    kontostand = kontostand - abhebe_gebuehr
                            - ueberzieh_gebuehr;
```

Die letzte Zeile wird immer ausgeführt! Sie gehört nicht mehr zum `else`-Zweig.

Richtig ist: **Blockbildung!**

```
if (kontostand >= betrag)
    kontostand = kontostand - betrag - abhebe_gebuehr
else {
    kontostand = kontostand - betrag;
    kontostand = kontostand - abhebe_gebuehr
                            - ueberzieh_gebuehr;
}
```

Dangling else (1)

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

Vorsicht!

Das `else` bezieht sich auf das **zweite** `if` und wird nicht ausgeführt, wenn das Konto gesperrt ist.
(Dann ist die gesamte `if`-Anweisung ohne Wirkung).

Die Einrückung entspricht daher nicht der Semantik!
(`else` sollte eingerückt sein)

Dangling else (2)

```
if (!kontoGesperrt)
  if (kontostand >= betrag) {
     kontostand = kontostand - betrag -
         abhebe_gebuehr;
     System.out.println("Abhebung erfolgreich");
     }
  else
     System.out.println("Abhebung nicht erlaubt, da
         Konto nicht gedeckt.");
```

Dangling else (2)

```
if (!kontoGesperrt)
  if (kontostand >= betrag) {
     kontostand = kontostand - betrag -
         abhebe_gebuehr;
     System.out.println("Abhebung erfolgreich");
     }
  else
     System.out.println("Abhebung nicht erlaubt, da
         Konto nicht gedeckt.");
else
  System.out.println("Abhebung nicht erlaubt, da
      Konto gesperrt.");
```

Vergleich: Bedingungsoperator vs. Bedingte Anweisung

Statement = ... | Conditional | ...
Conditional = IfStatement | ...
IfStatement = `"if"` `"("` Expression `")"` Statement [`"else"` Statement]

vs.

Expression = ... | Expression `"?"` Expression `":"` Expression

Gemeinsamkeiten:

- Boolescher Ausdruck bestimmt wie die Verzweigung fortgesetzt wird

Unterschiede:

- `if`-Anweisung ist eine **Anweisung**, während der Bedingungsoperator zum Formen von **Ausdrücken** verwendet wird
- `else` Statement ist **optional**, aber `":"` Expression ist **zwingend erforderlich**

Vergleich: Bedingungsoperator vs. Bedingte Anweisung (2)

Beispiel mit Bedingungsoperator:

```
int max;
max = a > b ? a : b;
```

Äquivalent dazu mit bedingter Anweisung:

```
int max;
if (a > b)
      max = a;
else  max = b;
```

Falsch, da if keine Ausdruecke, sondern Anweisungen formt:

```
max = if (a > b) a; else b;
```

Falsch, da nach ? und : ein Ausdruck steht und keine Anweisung

```
(a > b)?max = a;:max = b;
```

Wiederholungsanweisungen

Statement	= VariableDeclaration	(Deklarationsanweisung)
	| Assignment	(Zuweisung)
	| Block	(Block)
	| Conditional	(Fallunterscheidung)
	| Iteration	(Wiederholungsanweisung)

Wiederholungsanweisungen (Iterationen, Schleifen)

Wir unterscheiden 3 Arten von Wiederholungsanweisungen:
Syntax:

$$\begin{aligned} \text{Iteration} &= \text{WhileStatement} \\ &\mid \text{ForStatement} \\ &\mid \text{DoStatement} \end{aligned}$$

Mit den dann zur Verfügung stehenden Anweisungen (insbesondere `while`-Anweisungen) können **alle berechenbaren Funktionen** programmiert werden!

While-Anweisungen

Syntax:

WhileStatement = `"while" "(" Expression ")"` Statement

Beispiel:

```
while (i <= 100) {
  s = s+i;
  i = i + 1; // oder i++;
}
```

Nebenbedingung:
Der Typ des Ausdrucks Expression muss `boolean` sein.
Wirkung:
Solange die Auswertung von Expression den Wert `true` ergibt, wird die Anweisung Statement ausgeführt.

While-Anweisungen (2)

Formaler:

- Falls die Auswertung von Expression im Zustand σ den Wert `true` ergibt, σ Statement σ' und σ' `while` Expression Statement σ'', dann σ `while` Expression Statement σ''
- Falls die Auswertung von Expression im Zustand σ den Wert `false` ergibt, dann σ `while` Expression Statement σ

While-Anweisungen: Beispiele

Beispiel: Zahlen von 1 bis 10 ausdrucken

```
public class WhileExampleNumbers {
  public static void main(String[] args) {
      int n = 1; // Iterator
      int end = 10;
      while (n <= end) {
       System.out.println(n);
       n++;
      }
  }
}
```

While-Anweisungen: Beispiele (2)

Beispiel: Quersumme einer Zahl berechnen

```
public class DigitSum {
  public static void main(String[] args) {
      int x = 352;
      int digitSum =   0; // Akkumulator
      while (x > 0) {
       digitSum += x % 10;
       x = x/10;
      }
      System.out.println(digitSum);
  }
}
```

Entwurf von While-Schleifen

Methodische Richtlinien

1. Bestimmung der **Anfangswerte** der Variablen **vor** Eintritt in die While-Anweisung
2. Bestimmung der **Schleifenbedingung**
3. Formulierung des **Schleifenrumpfes**
4. Prüfen, dass die Schleifenbedingung nach **endlich vielen** Ausführungen des Rumpfes nicht mehr erfüllt ist.
   Anderenfalls terminiert die While-Anweisung nicht!

```
int x = 352;
int digitSum =   0;
while (x > 0) {                  // Terminiert NICHT!
    digitSum += x % 10;
}
```

Beispiel: Fakultät einer Zahl

Fakultät $n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n$ für $n \geq 1$ und $0! = 1$
Entwurf eines Programms: while-Anweisung:

1. Variablen und Anfangswerte:
   Verwende Variablen `count` zum Zählen (initial 1), factorial für das Ergebnis (initial 1), und n für die Zahl n.
2. Schleifenbedingung: Zähler ist noch nicht größer als n, d.h. `count <= n`.
3. Schleifenrumpf: Multipliziere Ergebnis mit dem Zähler und erhöhe den Zähler:

   ```
   factorial = factorial * count;
   count++;
   ```

4. Terminiert die Schleife? Ja, denn count wird in jedem Durchlauf erhöht und n verändert sich nicht.

Beispiel: Fakultät einer Zahl (2)

```
public class Factorial {
  public static void main(String[] args) {
     int n = 5;
     int count = 1;
     int factorial = 1;
     while (count <=  n) {
      factorial = factorial * count;
      count++;
     }
     System.out.println(factorial);
  }
}
```

For-Anweisungen

- Häufige Form einer While-Schleife ist:

```
int i = start;          // Initialisierung einer
                        // Iteratorvariablen
while (i <= end) {      // Grenze fuer den Iterator
  ...
  i++; // konstante Aenderung des Iterators (hier +1)
}
```

- Abkürzende Schreibweise durch eine For-Anweisung:

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

For-Schleife: Beispiele

```
public class FactorialWithFor {
  public static void main(String[] args) {
      int n = 5;
      int factorial = 1;
      for (int count=1;count <= n; count++) {
       factorial = factorial * count;
      }
      System.out.println(factorial);
  }
}
```

For-Schleife: Beispiele (2)

```
public class FactorialWithForDownwards {
  public static void main(String[] args) {
      int n = 5;
      int factorial = 1;
      for (int count=n;count > 0; count--) {
       factorial = factorial * count;
      }
      System.out.println(factorial);
  }
}
```

For-Schleife: Beispiele (3)

Summe der ersten 100 Zahlen:

```
public class Sum  {
  public static void main(String[] args) {
      int summe = 0; // Akkumulator
      int factorial = 1;
      for (int i =0; i <= 100; i++) {
       summe = summe + i;
      }
      System.out.println("Ergebnis = " + summe);
  }
}
```

For-Schleife: Beispiel (4)

Im Schleifenrumpf einer For-Anweisung kann selbst wieder eine For-Anweisung stehen. Man spricht von geschachteltem For, bzw. allgemein von geschachtelten Schleifen.
Beispiel:

```
public class SternchenDrucken  {
  public static void main(String[] args) {
  int n = 5;
  for (int i = 0; i < n; i++) {
     for (int j = 0; j < i; j++) {
        System.out.print('*');
     }
     System.out.println(); // Zeilenvorschub
   }
  }
}
```

Zusammenfassung

- 5 wesentliche Kontrollstrukturen der imperativen Programmierung: Variablendeklaration, Zuweisung, Block, Fallunterscheidung, Iteration
- Syntax und Semantik
- Modellierung des Zustands als Stack der momentan definierten Variablen und ihrer Werte
- Realisierung der Kontrollstrukturen in Java