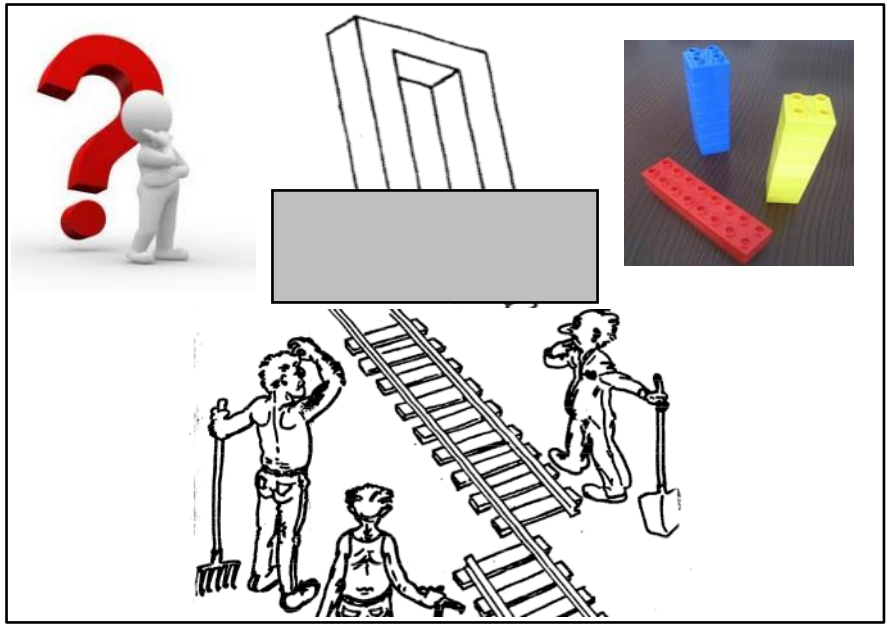
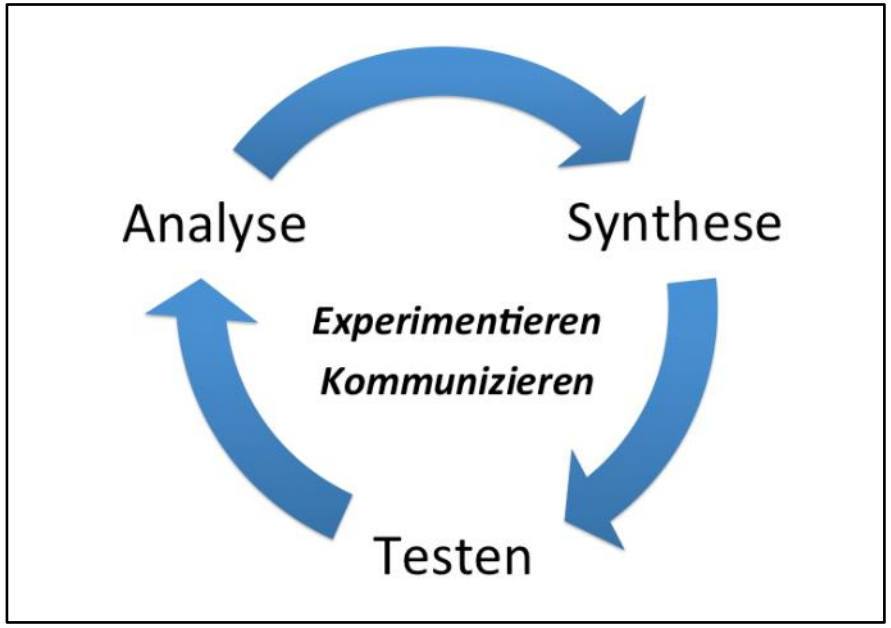
Software-Entwicklung 1
02: Grundlagen der imperativen Programmierung

Prof. Maalej & Team @maalejw



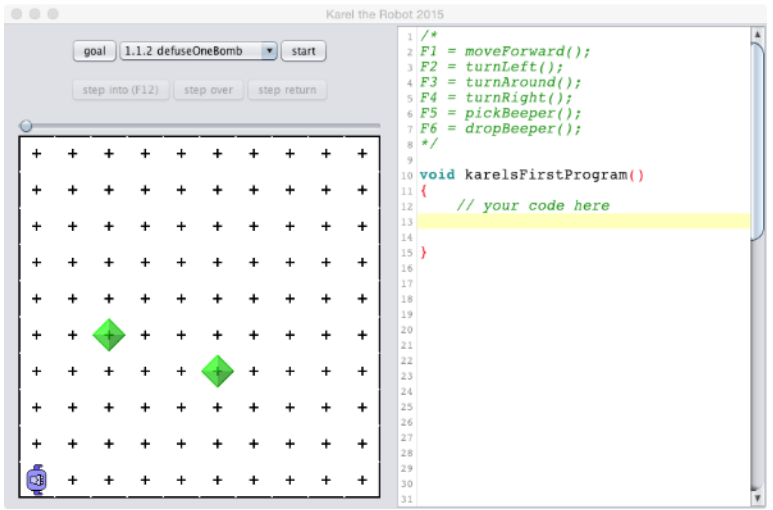


Was haben wir schon gemacht?

Wichtige Hilfsmittel

- **Algorithmen** (Lösungsbeschreibung)
- **Programmiersprachen**
- Techniken und **Notationen** (z.B., UML, Skizzen)
- **Methoden** (z.B. Pair-Programming, Scrum)
- **Werkzeuge** (z.B. Eclipse, BlueJ, Git, Issue Tracker)

Karel the Robot



Wichtige Hinweise

- Vorher mindestens **Kernbegriffe** lesen
- **Kriterien zum Bestehen der Übung** lesen
- **TutorInnen** bei Verständnisfragen einbinden

Übungsbetrieb Don’ts

Übungsbetrieb Do’s

Überblick

1 Organisatorisches

2 **Bedingungen in Kontrollstrukturen**

- Bedingte Schleifen
- Binärsystem mit Karel

Kontrollstrukturen

- In welcher Reihenfolge werden Karels elementare Befehle abgearbeitet?

- Die Abarbeitungsreihenfolge wird durch 5 **Kontrollstrukturen** beeinflusst:
  1. Sequenz
  2. Aufruf zusammengesetzter Befehle
  3. Zählschleife
  4. Fallunterscheidung
  5. Bedingte Schleife

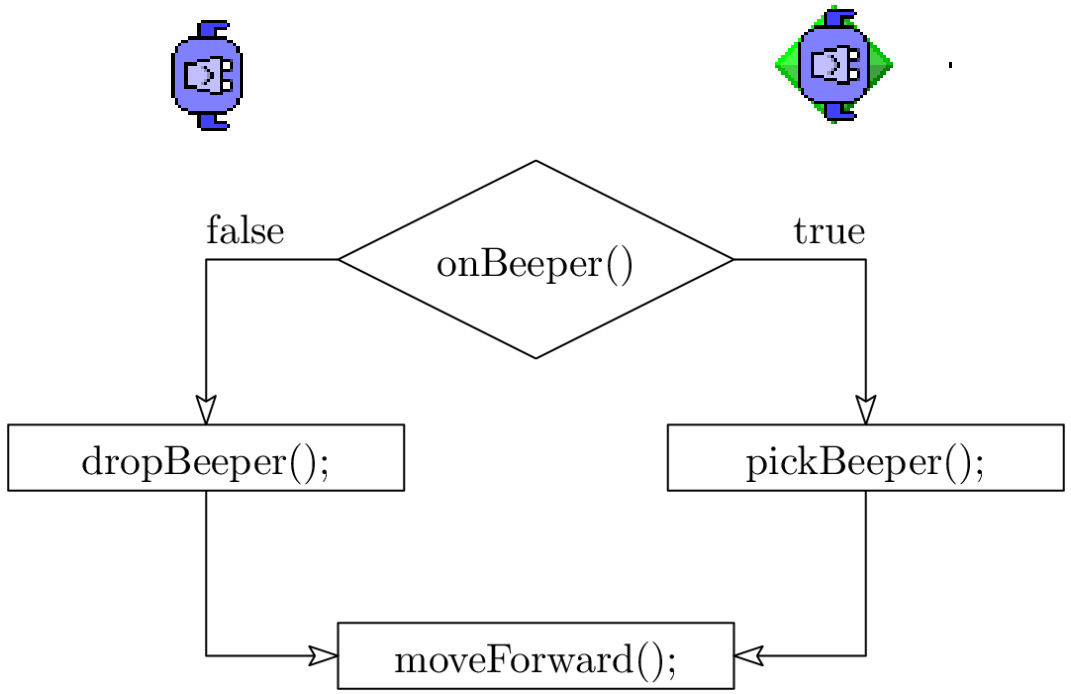
Fallunterscheidung mit Alternative

if Bedingung

```
if (onBeeper())
{
    pickBeeper();
}
else
{
    dropBeeper();
}
moveForward();
```

if Block

else Block



Negation mit dem !

```
if (frontIsClear())
{
}
else
{
    turnLeft();
}
```

=

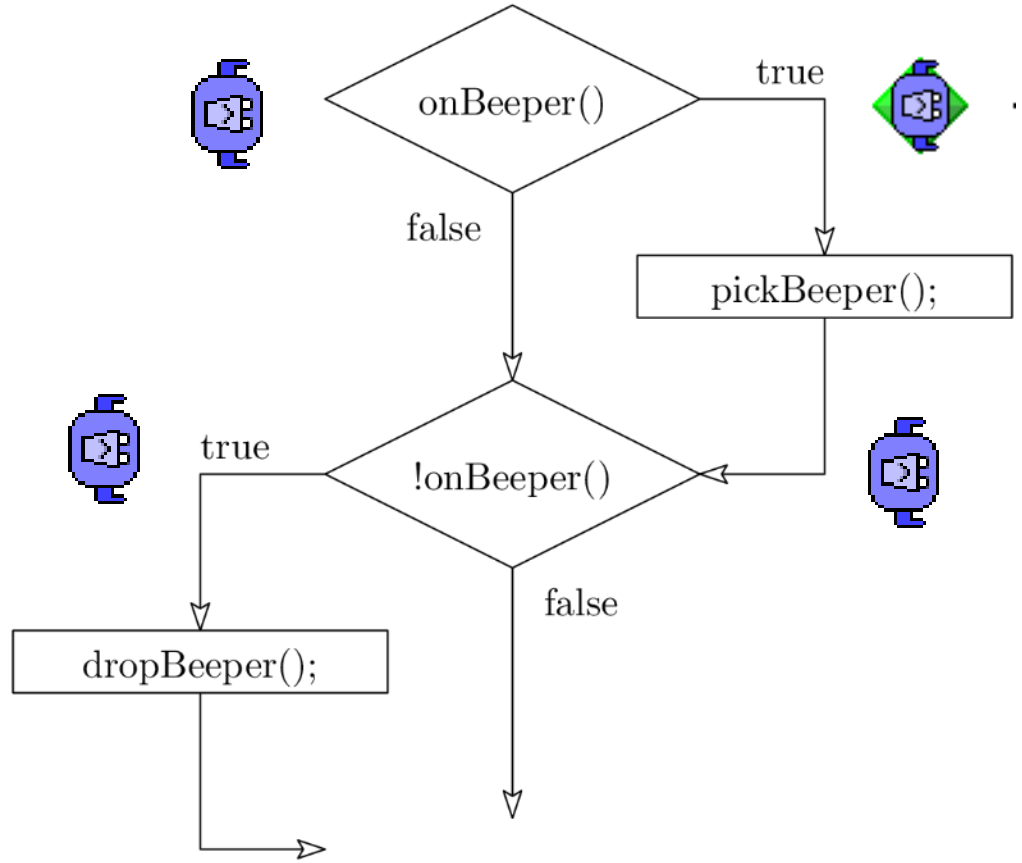
```
if (!frontIsClear())
{
    turnLeft();
}
```

- **`if/else`** mit leerem if-Block kann durch ein **`if`** ohne **`else`** ersetzt werden, indem man die Bedingung negiert

Negierte Fallunterscheidung statt “else“

- **Else** ist nicht immer durch ein negiertes **if** ersetzbar

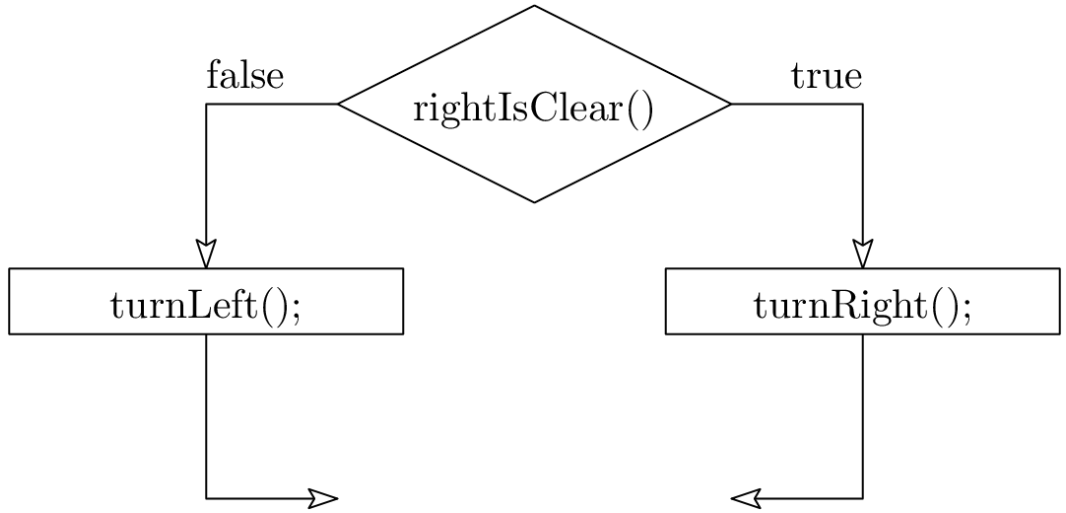
```
void pickOrDrop()
{
    if (onBeeper())
    {
        pickBeeper();
    }
    if (!onBeeper())
    {
        dropBeeper();
    }
}
```



Warum trifft die zweite Bedingung hier immer zu?

I: Wie viele verschiedene Pfade gibt es durch den Code?

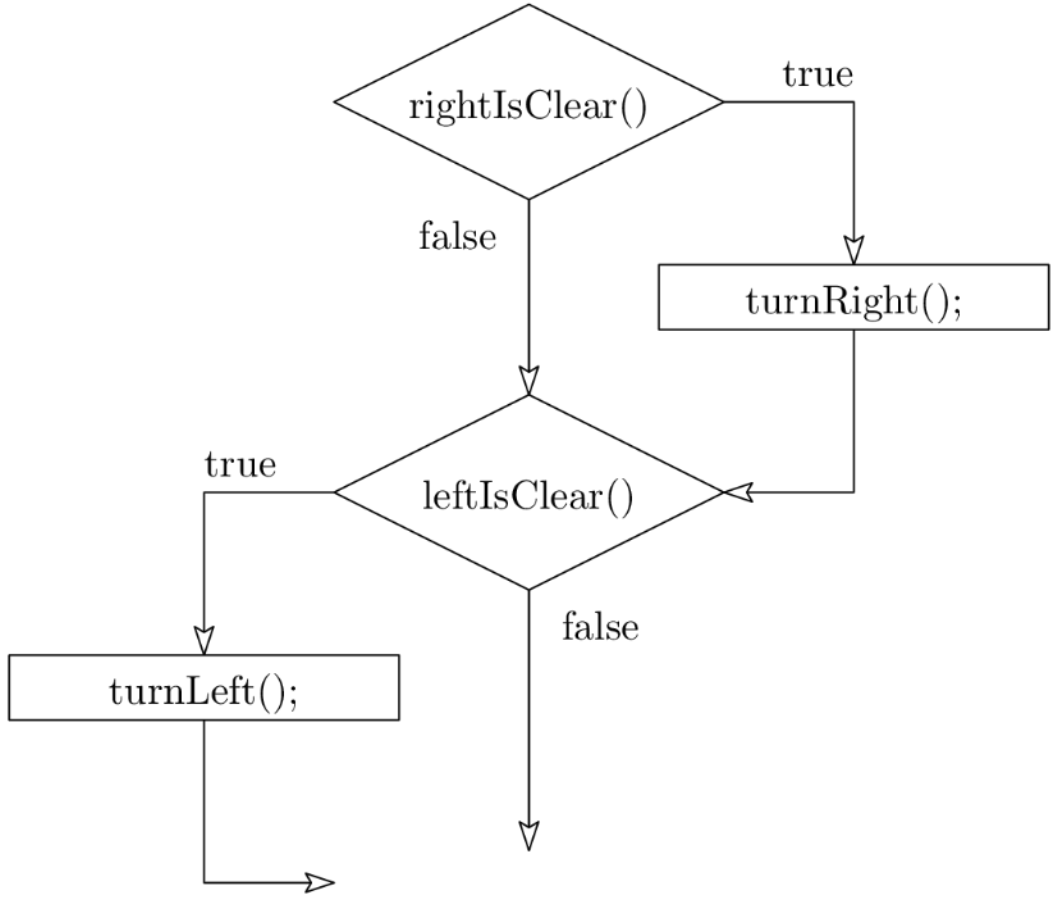
```
void ifelse()
{
    if(rightIsClear())
    {
        turnRight();
    }
    else
    {
        turnLeft();
    }
}
```



2 Pfade: entweder **turnLeft()** oder **turnRight()**

II: Wie viele verschiedene Pfade gibt es durch den Code?

```
void ifif()
{
    if (rightIsClear())
    {
        turnRight();
    }
    if (leftIsClear())
    {
        turnLeft();
    }
}
```



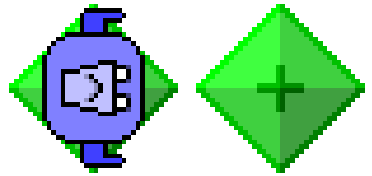
4 Pfade: entweder **keine Ausführung** oder **turnRight** oder **turnLeft** oder **beides**

Bedingungen kombinieren

- Konjunktion („**und**“)

```
onBeeper() && beeperAhead()
```

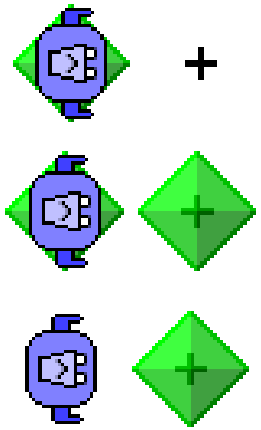
  - **Beide** Bedingungen müssen gelten



- Disjunktion („*inklusives* **oder**“)

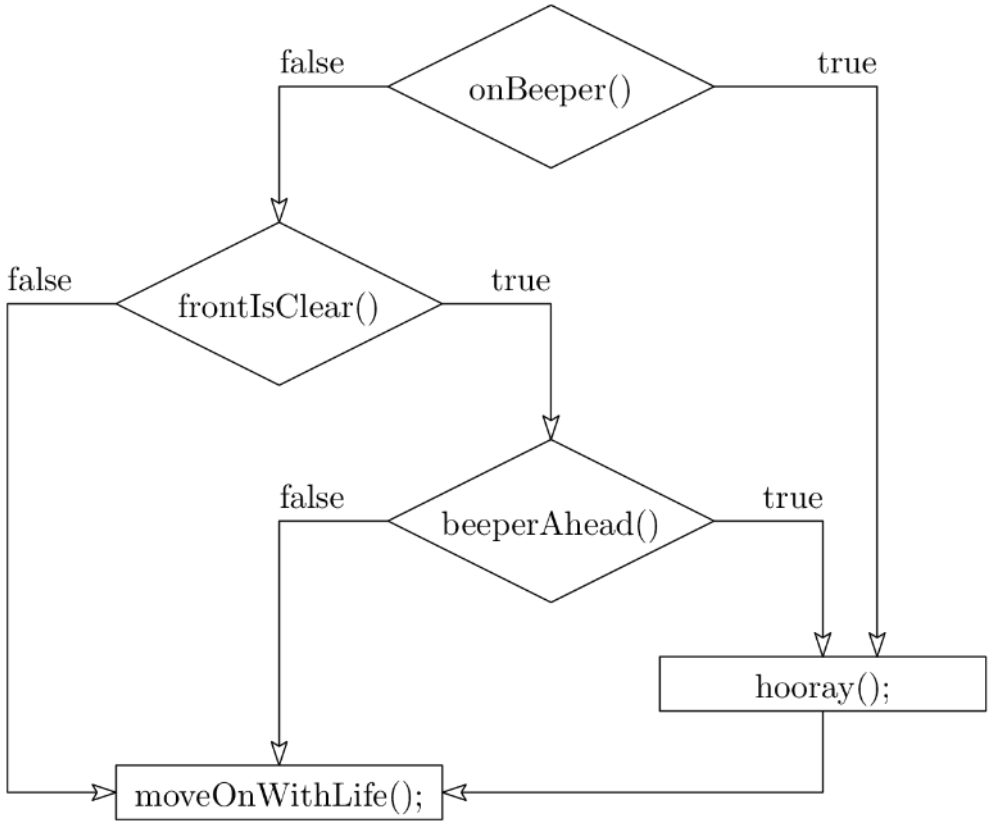
```
onBeeper() || beeperAhead()
```

  - **Mindestens eine** der beiden Bedingungen muss gelten



Konjunktion (“und”) bindet stärker als Disjunktion (“oder”)

```
if (onBeeper() ||  frontIsClear() && beeperAhead())
{
    hooray();
}
moveOnWithLife();
```



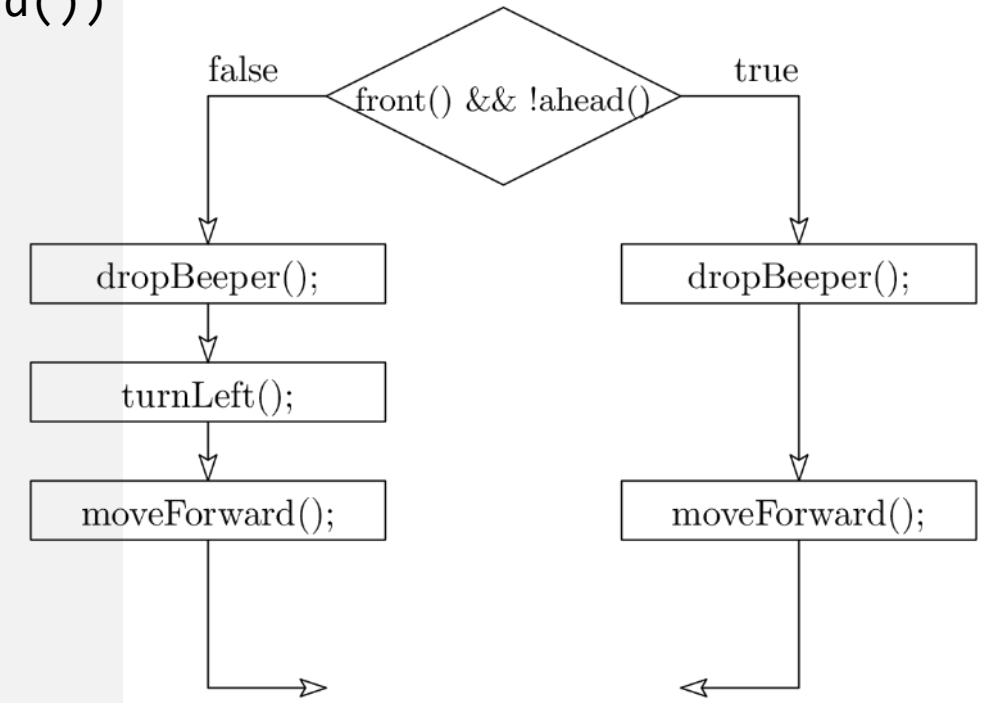
Merksatz:

“Punkt vor Strichrechnung”

Demo:
1.3.4 tileTheFloor

tileTheFloor: Fundstücke aus dem Übungsbetrieb

```
if (frontIsClear() && !beeperAhead())
{
    dropBeeper();
    moveForward();
}
else
{
    dropBeeper();
    turnLeft();
    moveForward();
}
```

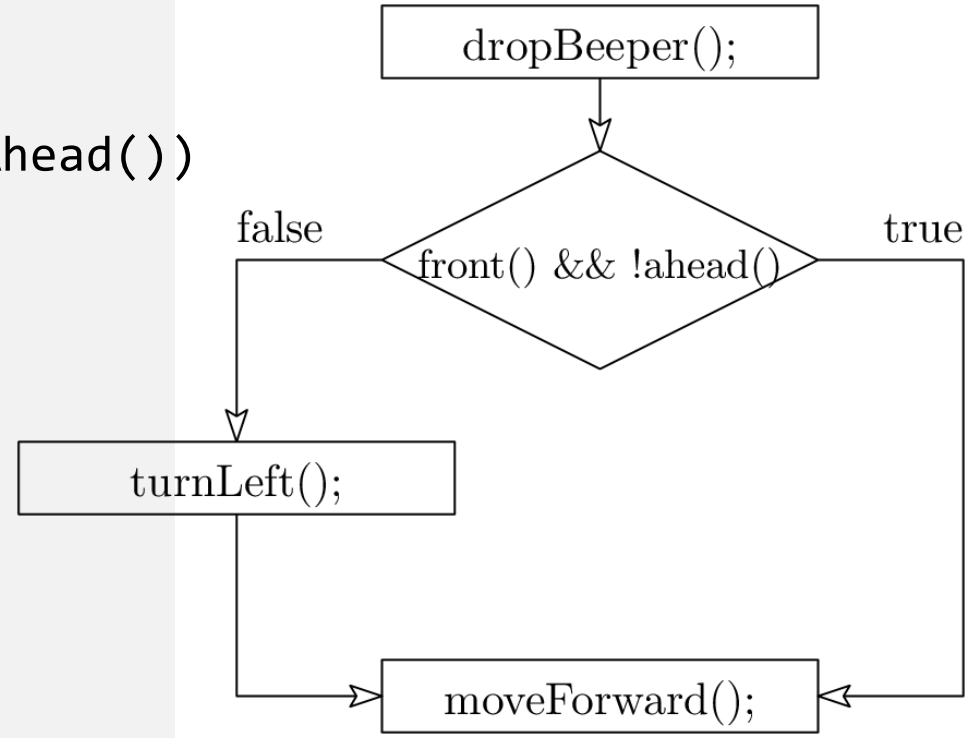


Beobachtung: Die beiden Blöcke sind sich recht ähnlich.

Frage: Kann man die Fallunterscheidung vereinfachen?

tileTheFloor: Fundstücke aus dem Übungsbetrieb

```
dropBeeper();
if (frontIsClear() && !beeperAhead())
{
}
else
{
    turnLeft();
}
moveForward();
```



Beobachtung: Eine Fallunterscheidung mit einem leeren Block ist ungewöhnlich.

Frage: Kann man das **`else`** loswerden, d.h. das **`if/else`** durch ein einfaches **`if`** ersetzen?

tileTheFloor

```
// vorher:
if (frontIsClear() && !beeperAhead())
{
}
else
{
    turnLeft();
}

// nachher:
if (!(frontIsClear() && !beeperAhead()))
{
    turnLeft();
}
```

Jetzt haben wir zwei Negationen; geht das nicht einfacher?

De Morgan Beispiel

- Alice schmeckt Kaffee nur
  - **mit Milch** und **mit Zucker**

```
withMilk() && withSugar()
```

- Wenn Alice den Kaffee **nicht** trinkt, dann ist der Kaffee:
  - **nicht** mit **(Milch** und **Zucker)** `! (withMilk() && withSugar())`
  - **ohne Milch** oder **ohne Zucker** `! withMilk() || ! withSugar()`

Gesetze von Augustus De Morgan

!(A && B) ⇔ !A || !B

Beide Bedingungen treffen **nicht** zu
bedeutet
mindestens eine Bedingung trifft **nicht** zu

!(A || B) ⇔ !A && !B

Mindestens eine Bedingung trifft **nicht** zu
bedeutet
Beide Bedingungen treffen **nicht** zu

Beispiel in Karel

```
if (!(frontIsClear() && !beeperAhead()))

if ( !frontIsClear() ||  beeperAhead() )
```

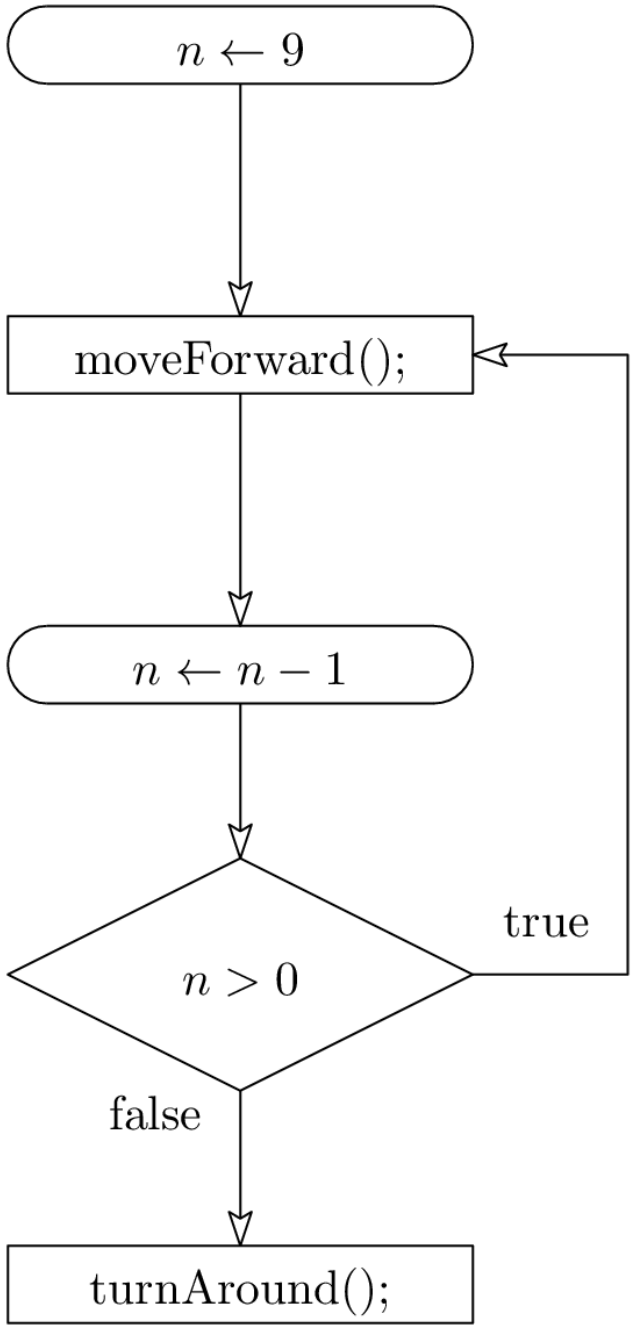
Kontrollstrukturen

- In welcher Reihenfolge werden Karels elementare Befehle abgearbeitet?

- Die Abarbeitungsreihenfolge wird durch 5 **Kontrollstrukturen** beeinflusst:
  1. Sequenz
  2. Aufruf zusammengesetzter Befehle
  3. Zählschleife
  4. Fallunterscheidung
  5. Bedingte Schleife

Zählschleife


```
repeat (9)
{
    moveForward();
}
turnAround();
```

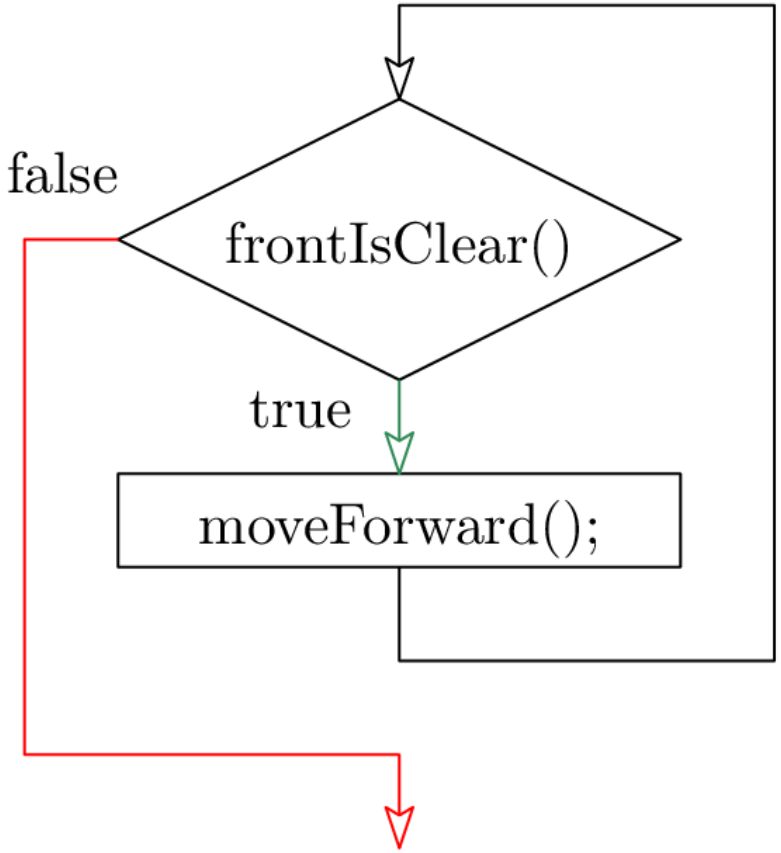


Problem:

- Bei variierender Feldgröße

Bedingte Schleife

```
void moveToWall()
{
    while (frontIsClear())
    {
        moveForward();
    }
}
```



- Solange die Bedingung wahr ist, wird der Block immer wieder ausgeführt
- Falls die Bedingung bereits am Anfang falsch ist, wird der Block nicht ausgeführt
- Schleifenbedingung wird vor dem nächsten Durchlauf geprüft
- Unbestimmte Anzahl an Ausführungen!

Demo while-Schleife:

2.1.1 hangTheLampions

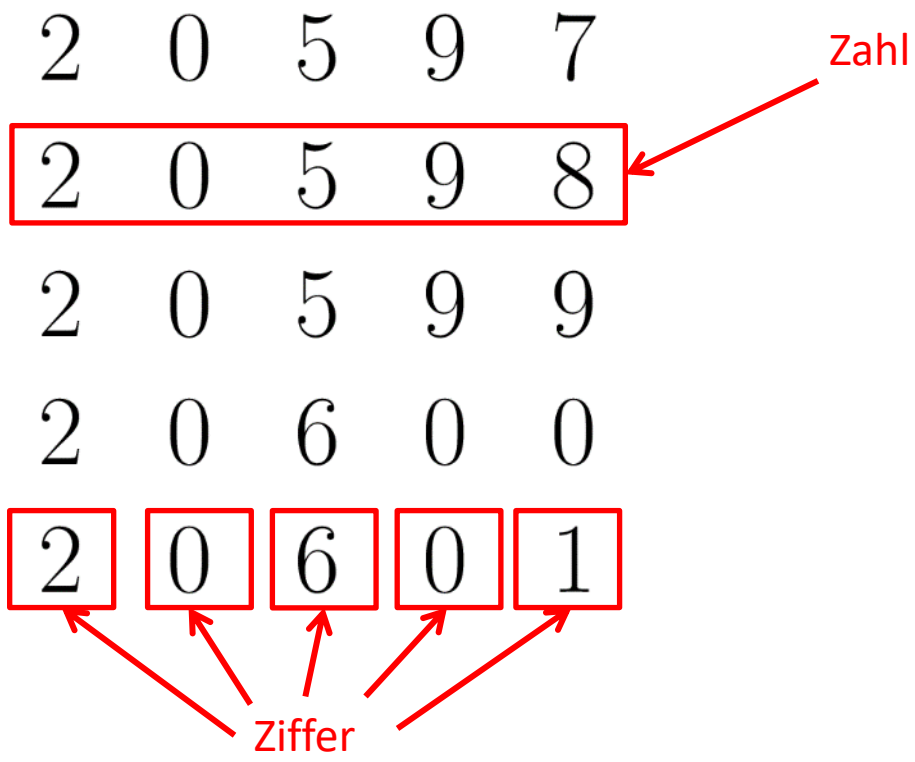
Überblick

1 Organisatorisches

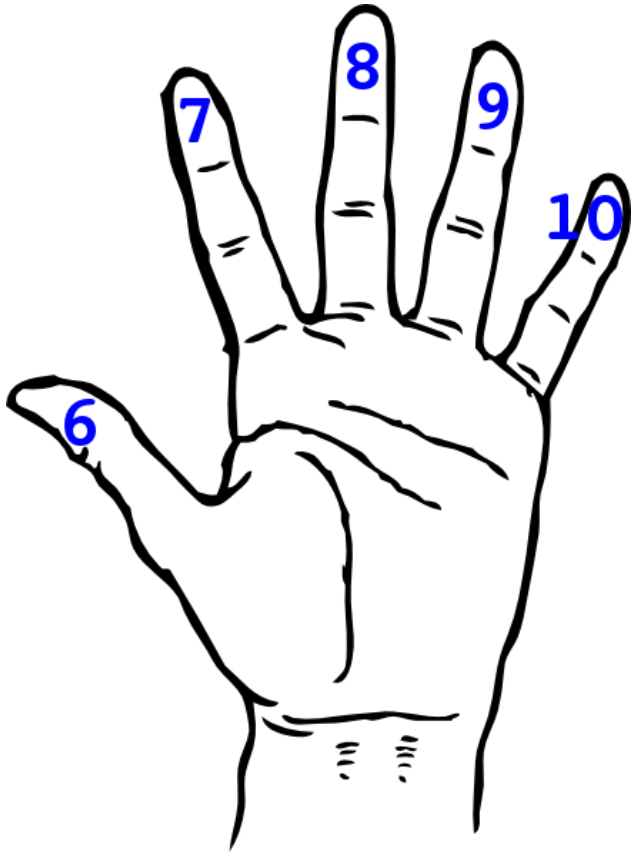
2 **Bedingungen in Kontrollstrukturen**

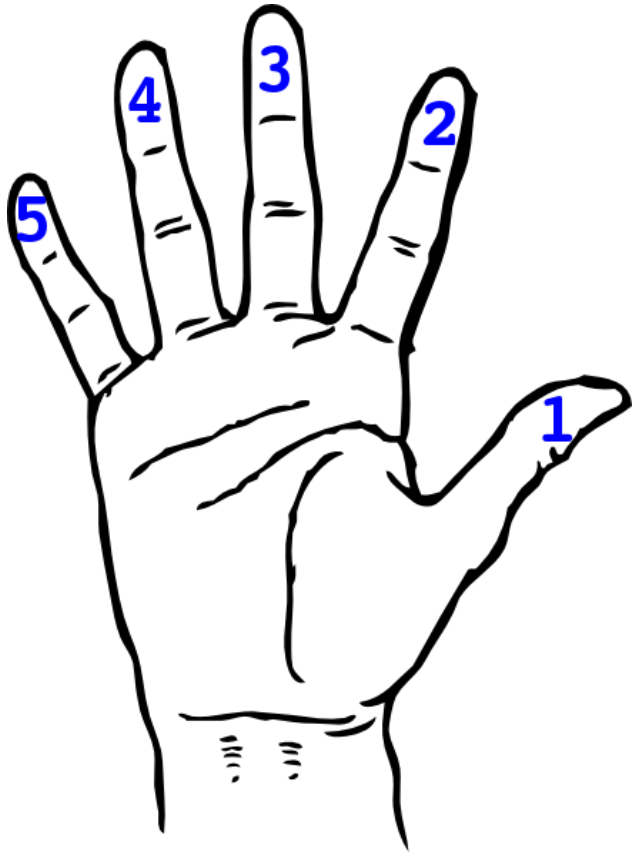
- Bedingungen und bedingte Schleifen
- **Binärsystem mit Karel**

Dezimalsystem

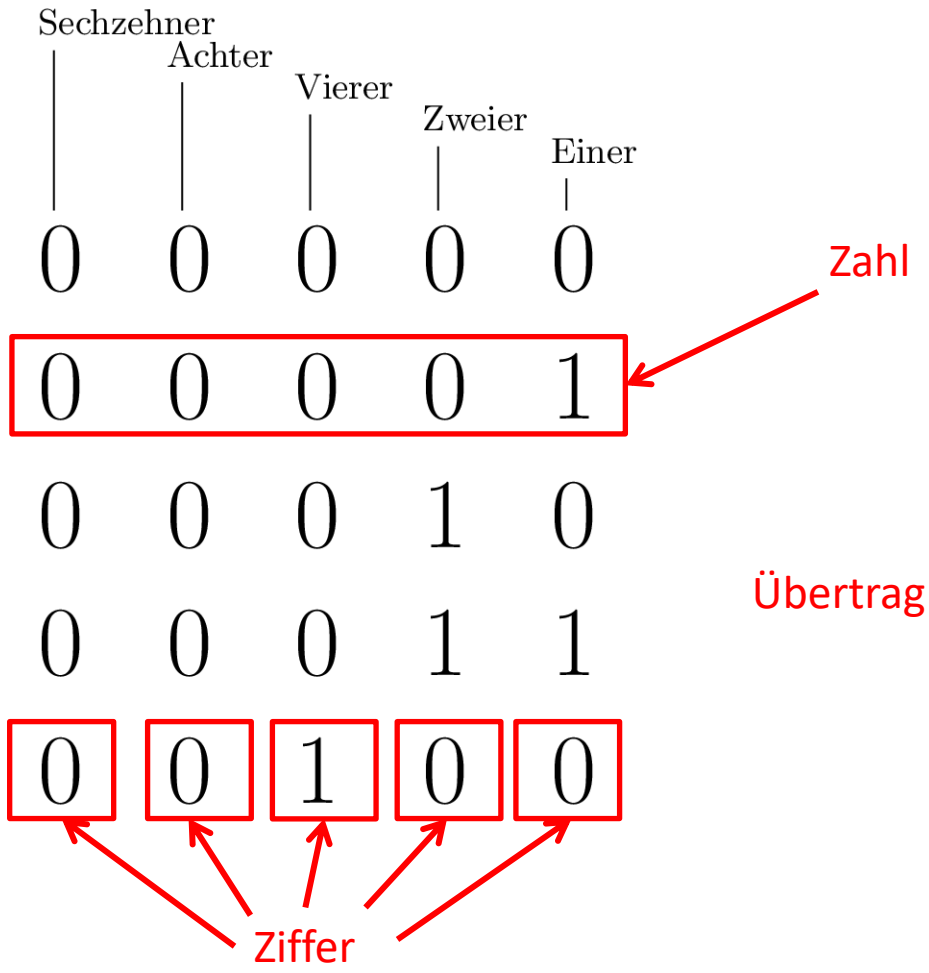


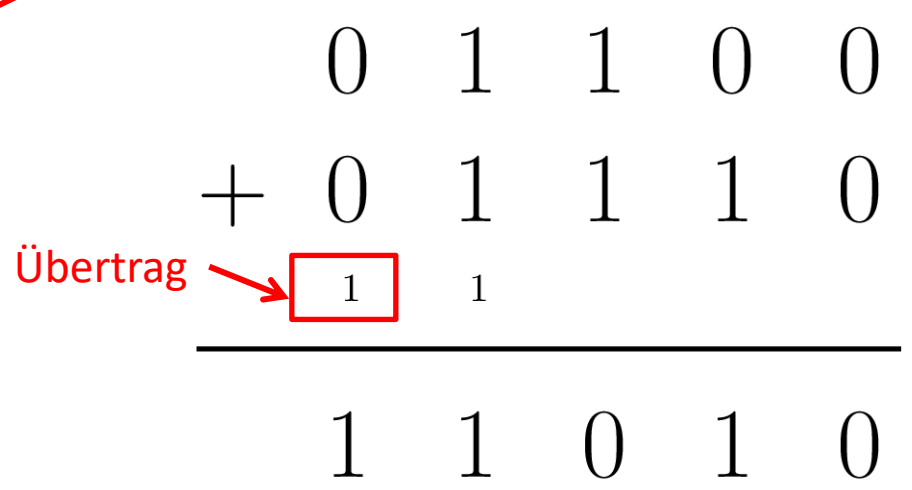
Warum gibt es eigentlich zehn verschiedene Ziffern?





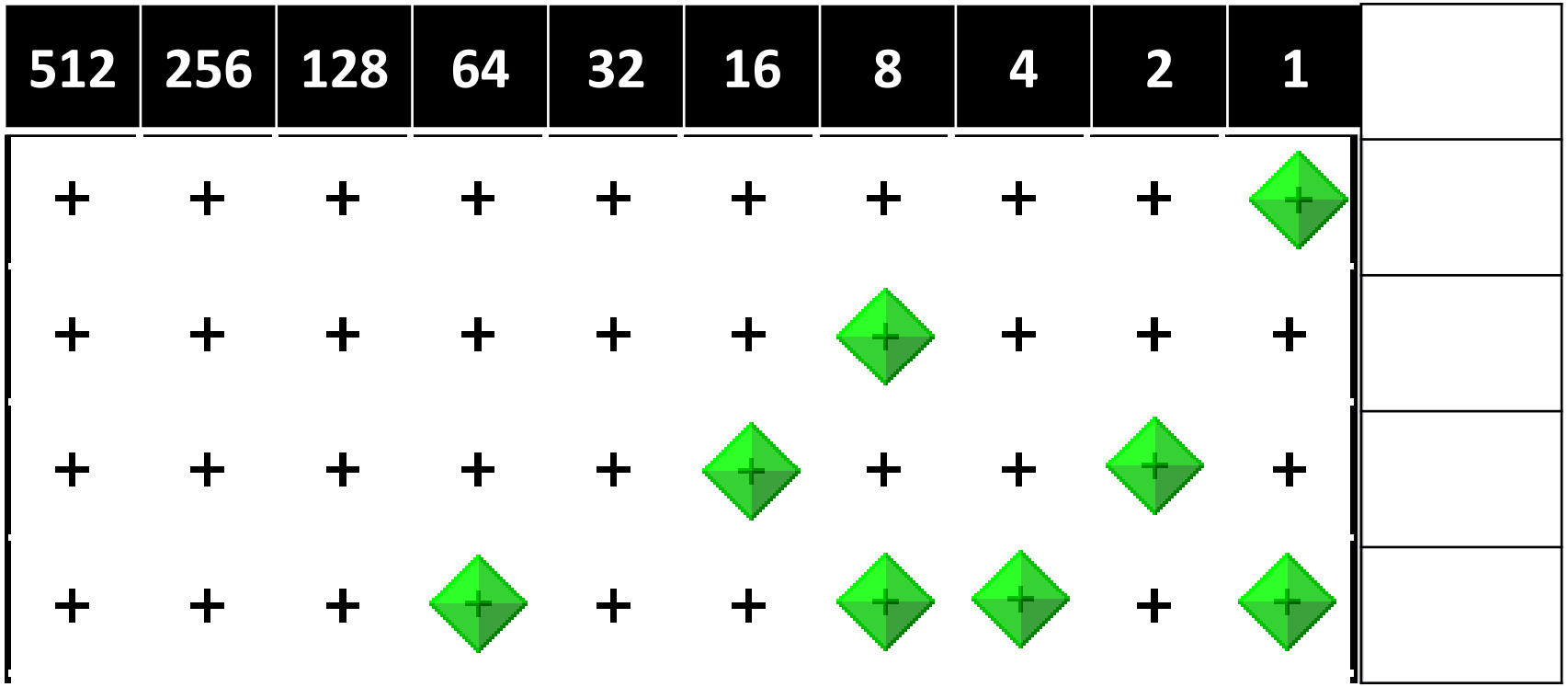
Binärsystem





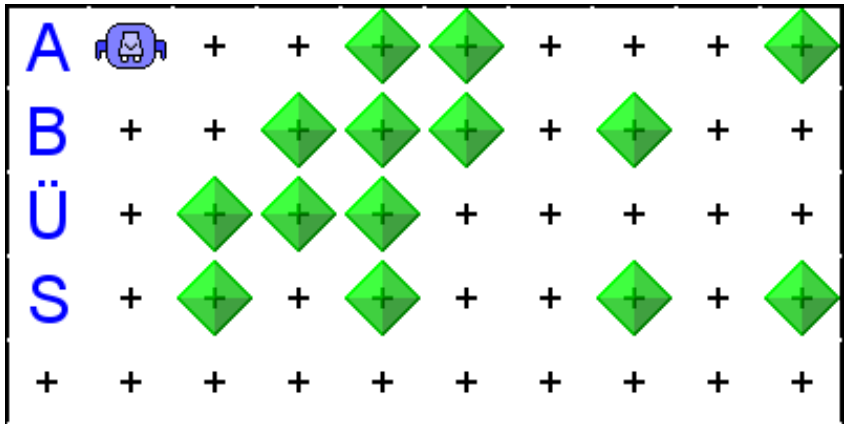
Binärsystem in Karels Welt

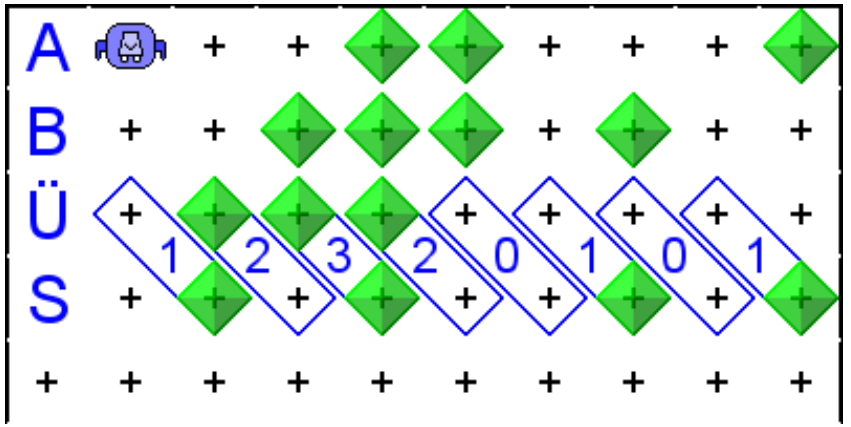
- Jede Zeile repräsentiert eine Binärzahl
  - 1 = Beeper
  - 0 = Leeres Feld



Schriftliche Addition im Binärsystem

- Addition im Binärsystem wie im Dezimalsystem
- A + B werden von rechts nach links aufsummiert
- Eventuell auftretende Überträge (Ü) berücksichtigen
- Blauer Kasten beinhaltet Summe A+B+Ü einer Spalte

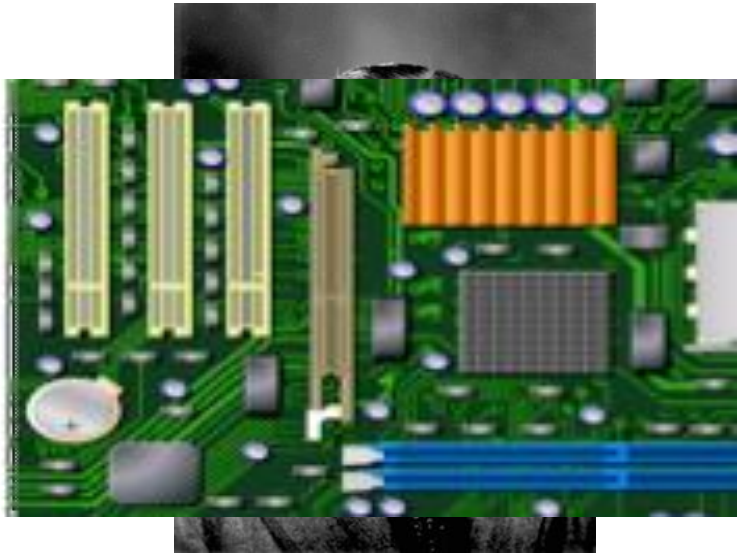




Überblick

1. Organisatorisches
2. Bedingungen in Kontrollstrukturen
3. **Imperative Programmierung**
   - Literale, Operationen und Ausdrücke
   - Variablen (und Zuweisungen)
   - Prozeduren

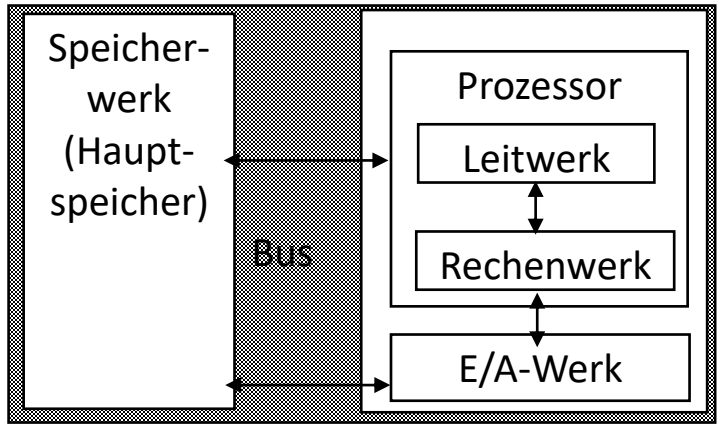
von Neumann-Rechner: Konzept (fast) aller Computer



- Rechner besteht aus **4 Werken**
- Rechnerstruktur ist **unabhängig** vom bearbeiteten Problem
- Programme und Daten stehen **im selben Speicher**
- Hauptspeicher ist in Zellen **gleicher Größe** unterteilt, die durchgehend **adressierbar** sind
- Ein Programm besteht aus Folgen von **Befehlen**, die generell **nacheinander** ausgeführt werden
- Von der sequenziellen Abfolge kann durch **Sprungbefehle** abgewichen werden
- Die Maschine benutzt **Binärcodes** für die Darstellung von Programm und Daten

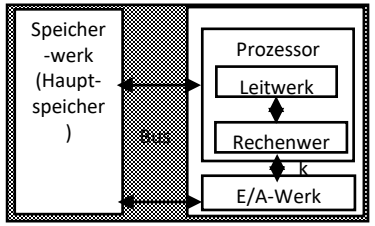
Imperative Programme auf von Neumann-Maschinen

- Die **elementaren Operationen** eines von Neumann-Rechners:
  - Prozessor führt **Maschinenbefehle** aus
  - Über den **Bus** werden **Befehle und Daten** vom Speicher in die CPU übertragen und die **Ergebnisse** zurück übertragen



- **Imperative Programmiersprachen** abstrahieren von diesen elementaren Operationen:
  - **Anweisungen** (engl.: statements) fassen Folgen von Maschinenbefehlen zusammen
  - **Variablen** (engl.: variables) abstrahieren vom physischen Speicherplatz

Ablaufsteuerung im Vergleich



von Neumann-Maschine

- Aufeinanderfolgende Befehle stehen hintereinander im Speicher
- Werden vom Steuerwerk in den Prozessor geholt und verarbeitet
- Mit Sprungbefehlen kann von der sequentiellen Reihenfolge der gespeicherten Befehle abgewichen werden

Programmiersprachen

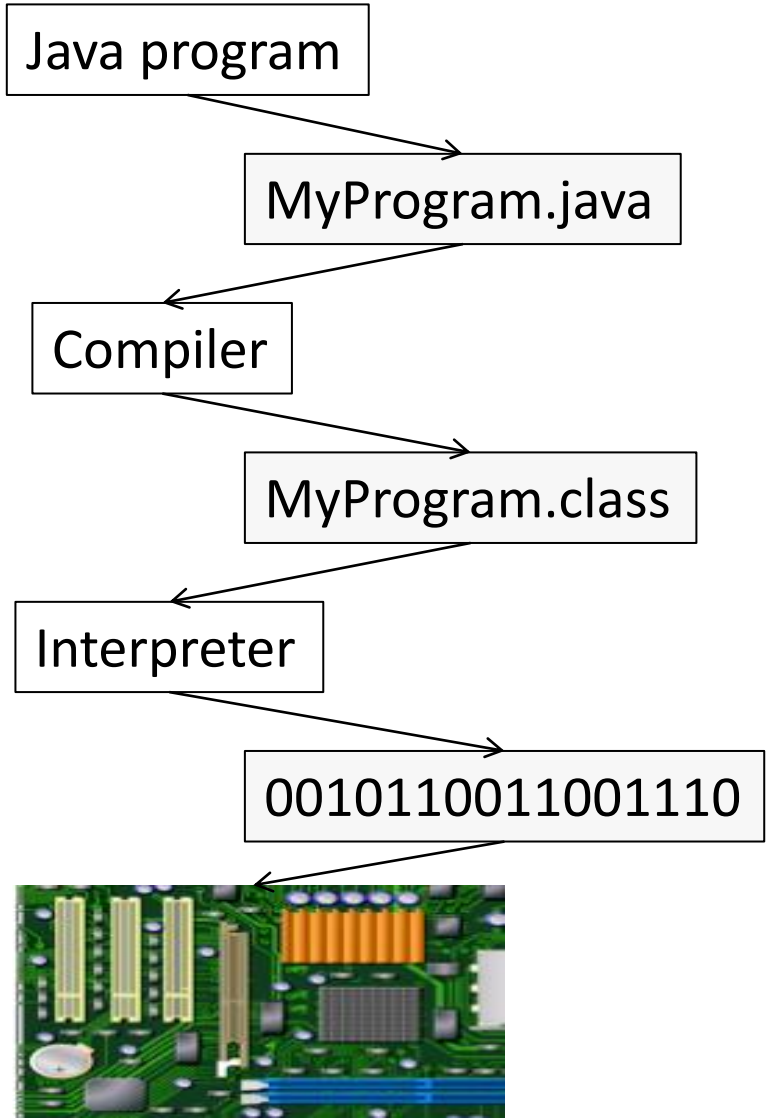
- *Kontrollstrukturen* innerhalb eines zusammengesetzten Befehls:
  - Sequenz
  - Fallunterscheidung
  - Wiederholung
- Aufrufe von zusammengesetzten Befehle greifen ebenfalls in den sequenziellen Ablauf ein

Übersetzung von Programmiersprachen

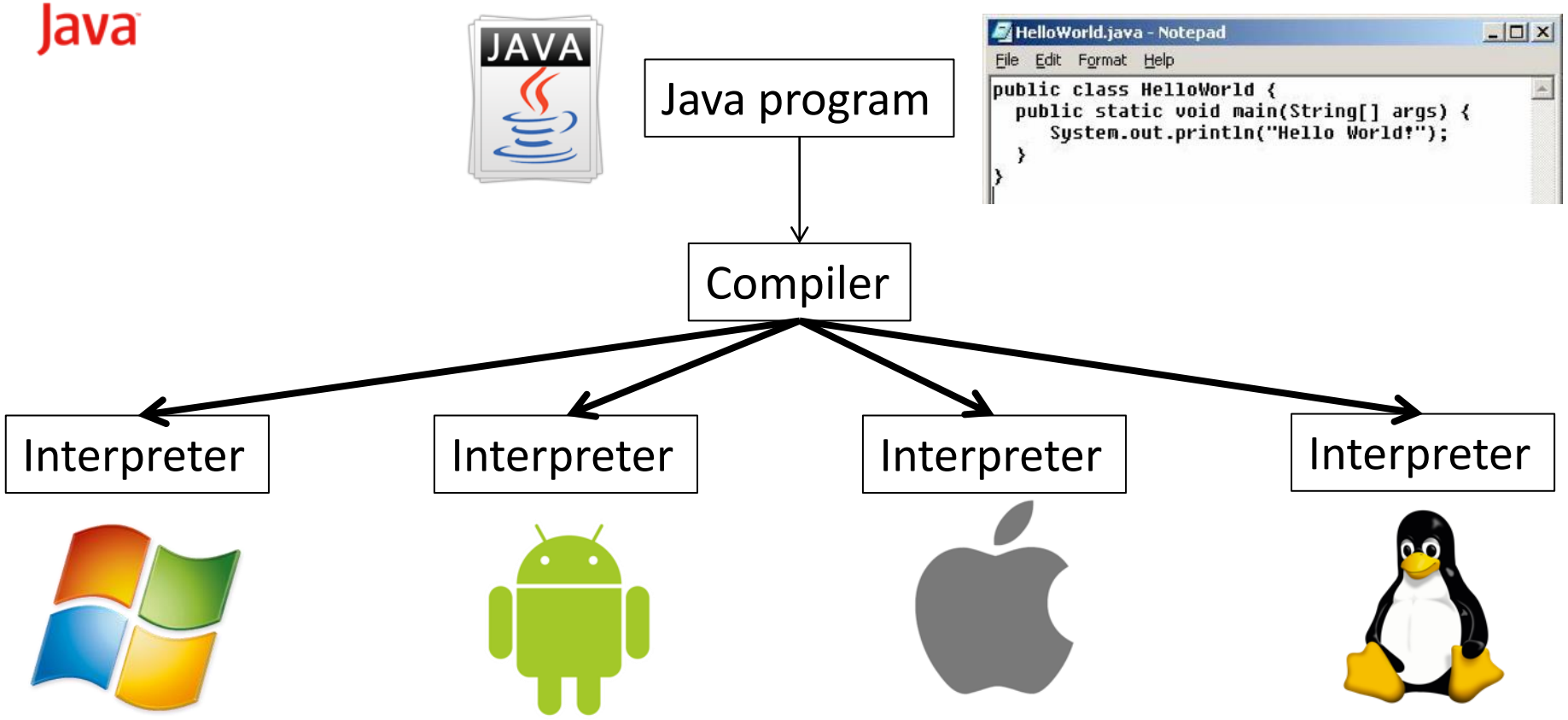
- **Compiler-Sprachen** (Bsp.: C++, Modula-2)
  - Anweisungen werden einmalig in Maschinensprache **übersetzt** und in dieser Sprache ausgeführt
- **Interpretersprachen** (Bsp.: Python, Lisp, PHP, Perl etc.)
  - Ein **Interpreter** übersetzt einzelne Anweisungen, wenn sie ausgeführt werden soll
- **Hybride Sprachen** (Bsp.: Java, C#)
  - Programme werden in eine Zwischensprache übersetzt, die sich gut für die Interpretation eignet

Hybride Verarbeitung in Java

- Der **Quelltext** wird in Zwischencode kompiliert
- Das ist der **Java Bytecode**
- Der Java Bytecode wird dem Interpreter übergeben, der sog. **Java Virtual Machine**

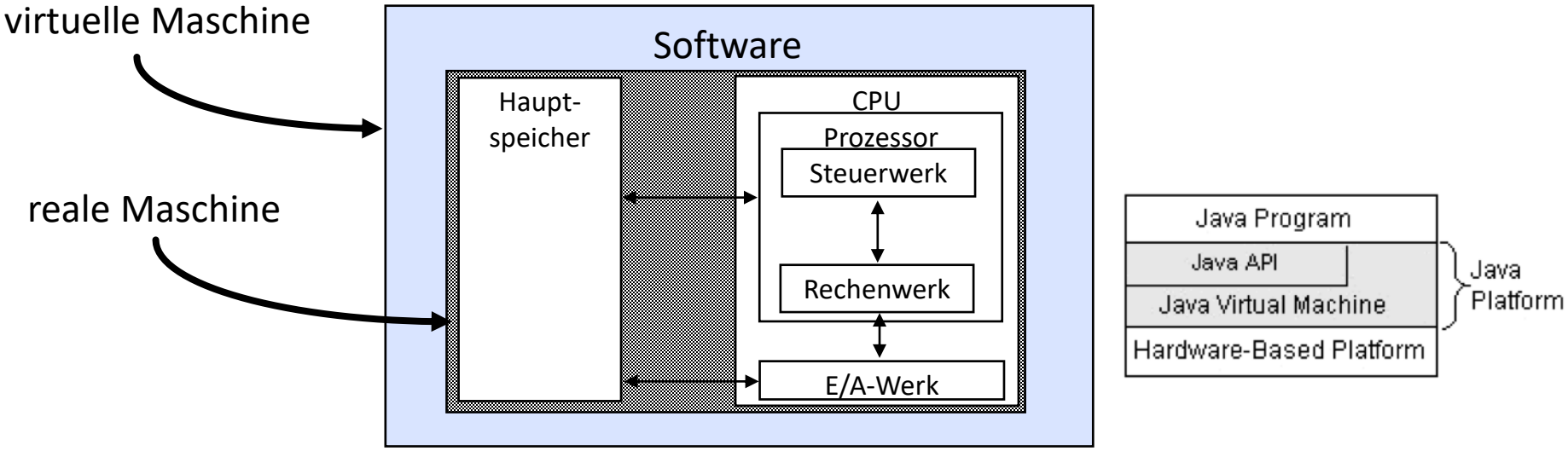


Java Virtual Machine



- **Java Bytecodes** ist quasi Maschinencode für die Java Virtual Machine (Java VM)
- „Write once, run anywhere“

Virtuelle Maschinen



- Ein **laufendes Programm** ist quasi eine Folge von Befehlen für eine **Maschine**, die diese Befehle schrittweise ausführen kann
- Diese Maschine kann auch Software sein
- Es genügt, ihren Satz an **Maschinenbefehlen**, ihre **Speicherbereiche** und ihre **Kontrollstrukturen** festzulegen
- Eine **virtuelle Maschine** ist eine durch Software definierte Maschine, die selbst auf einem Computer implementiert ist

Imperative Programmierung

- Imperative Programmierung baut auf dem Konzept des von Neumann-Rechners auf
- Programme sind **Folgen von Anweisungen**
- Ausführungsreihenfolge der Anweisungen ist durch die **textuelle Reihenfolge** oder durch **Sprunganweisungen** festgelegt
- Höhere **Programmkonstrukte** fassen Anweisungsfolgen zusammen und bestimmen die Ausführungsreihenfolge
- Benannte Variablen können Werte annehmen, die sich durch Anweisungen ändern lassen

Überblick

1. Organisatorisches
2. Bedingungen in Kontrollsturkturen
3. **Imperative Programmierung**
   - **Literale, Operationen und Ausdrücke**
   - Variablen (und Zuweisungen)
   - Prozeduren

Literale für ganze Zahlen

- Arithmetik in vielen Programmiersprachen verwendet Literale
- Literale sind feststehende **Werte** und stehen direkt im Quelltext
- Literale haben einen **Typ**
  - Ganze Zahlen sind vom Typ `int`

Beispiele für Literale vom Typ `int`:

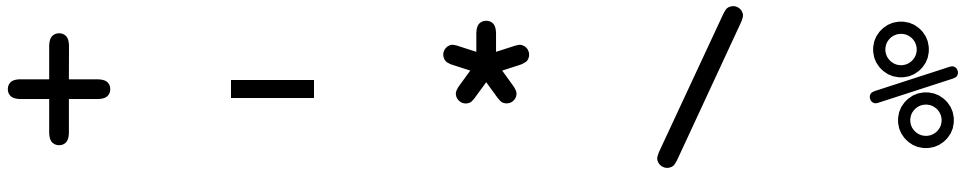
1 **14** **21** **-4** **-8**

0 **34567**

Operatoren

- Als **Operator** bezeichnet man umgangssprachlich
  - das **Operatorzeichen** (z.B. "+")
  - die damit verbundene **Operation** (z.B. "addieren“)

- Gängige Operatoren



Ausdruck

- **Ausdruck** (engl.: expression*)*
  - Synonym: Term

  - **Verarbeitungsvorschrift**, deren Ausführung einen Wert liefert
  - Ausdrücke entstehen, indem **Operanden** mit **Operatoren** verknüpft werden
  - In Programmiersprachen verwendet man häufig **arithmetische** und **logische** Ausdrücke

Beispiele für Ausdrücke:

```
5 * 4 + 3

true && false || true
```

Vereinbarungen über Operatoren

- Position
- Stelligkeit
- Präzedenz (Vorrangregel)
- Assoziationsreihenfolge
- Definition der mit dem Operator verbundenen Operation

- Operatorenschreibweise ist intuitiv

- Bei Neueinführung von Operatoren müssen Vereinbarungen explizit gemacht werden

Vereinbarungen über Operatoren: 1. Position

Infix

- Häufigste Schreibweise
- Operatoren stehen zwischen den beiden Operanden

  z.B. : **3 * 4**

Präfix (Funktionsschreibweise)

- Operator steht vor seinen Operanden
- Oft benutzt bei Operationen mit einem Operanden

  z.B.: **-2**

Postfix

- Operator steht nach seinen Operanden
- Oft benutzt für arithmetische Operationen mit einem Operanden

  z.B.: **3!** (“3 Fakultät")

Stelligkeit von Operatoren

- **Stelligkeit**, d.h., Anzahl der Operanden (auch Argumente oder Parameter) eines Operators

  - **einstellig**, oft: unär (engl.: unary)
    z.B. : **3!**

  - **zweistellig**, oft: binär (engl.: binary)
    z.B. : **3 * 4**

  - **dreistellig**, ternär (engl.: ternary), besser: triadisch
    In Programmiersprachen kommt meist nur vor
    if Operand1 **then** Operand2 **else** Operand3

Präzedenz von Operatoren

- **Präzedenz** (Vorrangregel): bezeichnet die Stärke, mit der ein Operator seine Operanden „bindet“
- Wert eines Ausdrucks ist oft abhängig von der Reihenfolge der Auswertung ab
- „**Punkt vor Strich**“ - für Arithmetische Operationen
  - Mit Klammern können Präzedenzen explizit bestimmt werden
  - Beispiele:
    - `3 + 5 * 7 – 3`
    - `(3 + 5) * (7 – 3)`

Assoziativität von Operatoren

- Bestimmt die implizite Klammerung von Ausdrücken bei Operatoren gleicher Präzedenz
  - Beispiel:
    - 5 - 4 - 3 ist gleichbedeutend mit (5 - 4) - 3
    - Sprechweise: Operator ist **linksassoziativ** (assoziiert von links nach rechts)

- Assoziationsreihenfolge irrelevant, wenn die Auswertungsreihenfolge nichts am Wert ändert
  - Beispiel:
    - (3 + 4) + 5
    - 3 + (4 + 5)

Zweistellige Operatoren für ganze Zahlen mit Ergebnistyp int

- Vier Grundrechenarten: + - * /
- Rest bei ganzzahliger Division mit dem % Operator
- Infixnotation
- Präzedenz ist „Punktrechnung vor Strichrechnung“

Operator	Bezeichnung	Ausdruck	Ergebnis (int)
+	Plus	14 + 3	17
-	Minus	24 - 13	11
*	Mal	7 * 4	28
/	Durch	20 / 6	3
%	Modulo	20 % 6	2

Zweistellige Operatoren für ganze Zahlen mit Ergebnistyp boolean

Operator	Bezeichnung	Ausdruck	Ergebnis (boolean)
>	Größer	2 > 1+1	false
>=	Größergleich	2 >= 3-1	true
<	Kleiner	2 * 2 < 1 * 1	false
<=	Kleinergleich	2 <= 2/1	true
==	Gleich	3 == 2+1	true
!=	Ungleich	4 != 2*2	false

- Vergleichoperatoren haben gegenüber den arithmetischen Operatoren eine niedrigere Präzedenz
- Werden im Ausdruck zuletzt ausgewertet

Boolesche Literale und Operatoren

- Boolesche Werte oder Wahrheitswerte sind in Java vom primitiven Typ `boolean`
- Literale für die booleschen Werte sind **`true`** und **`false`**

Operator	Bezeichnung	Ausdruck	Ergebnis (boolean)
&&	Logisches Und	true && false	false
||	Logisches Oder	true || false	true
!	Logische Verneinung	!true	false
==	Gleich	true == false	false
!=	Ungleich	true != false	true

Übersicht: Zentrale Operatoren in Java

Operator	Funktion, **arithmetisch**
*	Multiplikation
/	Division
%	Modulo
+	Addition
-	Subtraktion
++	Inkrement
--	Dekrement

Sowohl prä- als auch postfix verwendbar

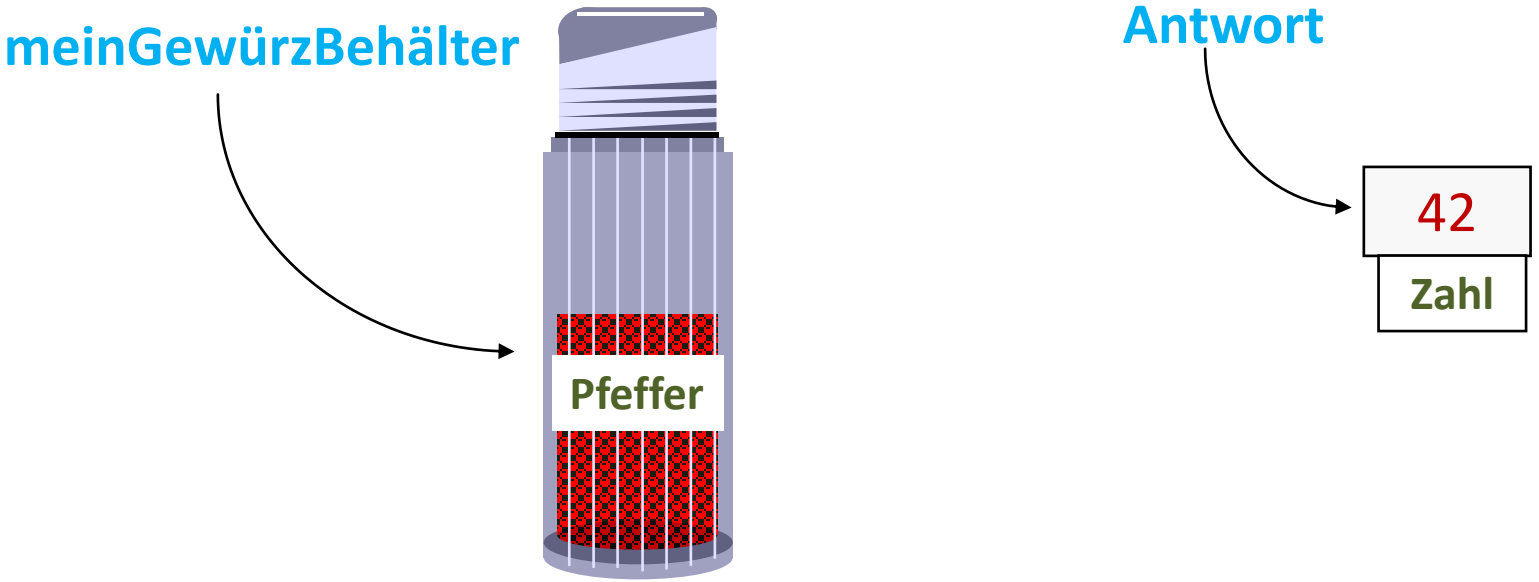
Operator	Funktion, **boolesch**
!	logisches NICHT
&&	logisches UND
||	logisches ODER
<	„kleiner als“
<=	„kleiner gleich“
>	„größer als“
>=	„größer gleich“
==	Gleichheit
!=	Ungleichheit

Überblick

1. Organisatorisches
2. Bedingungen in Kontrollsturkturen
3. **Imperative Programmierung**
   - Literale, Operationen und Ausdrücke
   - **Variablen (und Zuweisungen)**
   - Prozeduren

Variablen

- Abstraktion eines physischen **Speicherplatzes**
- Benutzen/ansprechen durch den **Namen** (auch: Bezeichner)
- **Variable** hat den Charakter eines Behälters:
  - **Belegung** (aktueller Inhalt) kann sich ändern
  - **Typ** legt Wertebereich, zulässige Operationen und weitere Eigenschaften fest



Deklaration und Initialisierung

- **Variablen** werden vor der Verwendung bekanntgemacht, d.h. **deklariert**

- Vereinfacht geschieht dies durch:
  - Angabe des **Typs**
  - Vergabe eines **Namens** über einen **Bezeichner** (engl.: identifier)

- Durch die reine Deklaration von Variablen ist deren **Belegung** zunächst meist **undefiniert**

- Erst bei der **Initialisierung** wird eine Variable erstmalig mit einem gültigen Wert befüllt

Deklaration

```
int i;
boolean b;
```

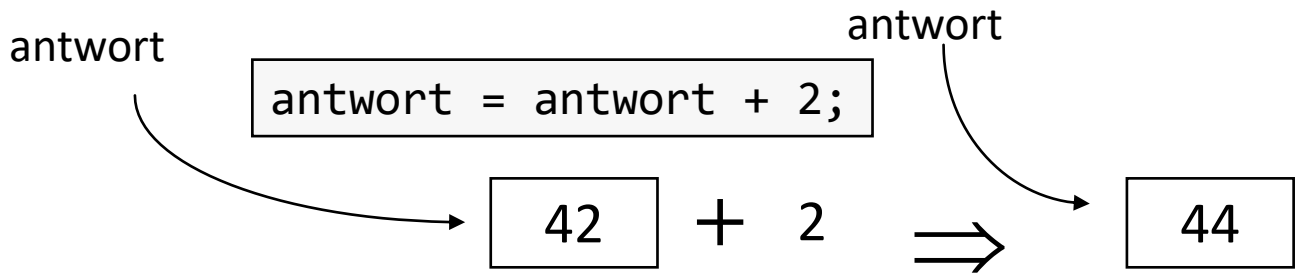
Deklaration und Initialisierung

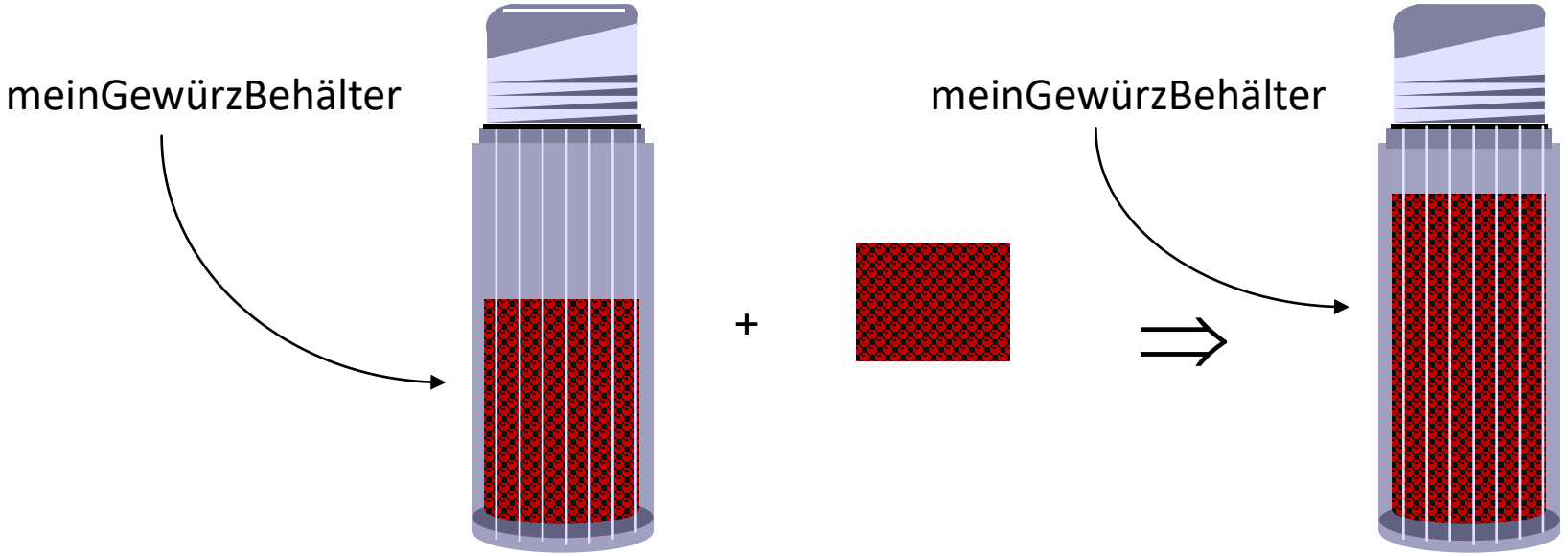
```
int i = 42;
boolean b;

b = true;
```

Veränderung eines Variablenwerts

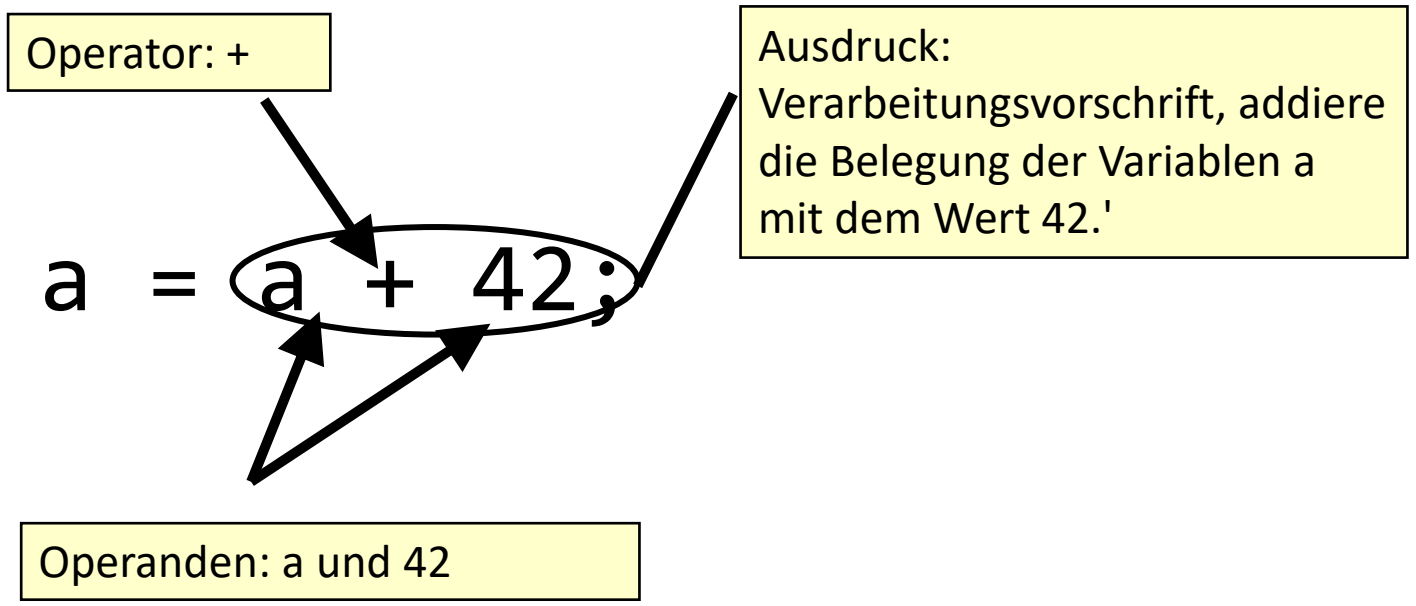
- **Zuweisung:** die Veränderung der **Belegung** eine Variable unter Beibehaltung ihres Namens und Typs





Neue Belegung mit einem Ausdruck berechnen

- Oft wird einer Variablen ein Wert zugewiesen indem ein **Ausdruck** aus **Operanden** und **Operatoren** ausgewertet wird
- **Arithmetische** und **boolesche** Operatoren sind üblich



Zuweisung

```
a = 3 + b;
```

- Bei der **Zuweisung** wird ein **Ausdruck** ausgewertet und sein **Ergebnis** einer **Variablen** zugewiesen

- Syntax-Schema der Zuweisung:

  <Linke-Seite> <Zuweisungsoperator> <Rechte-Seite>

  - **Rechte-Seite:** Ausdruck wird ausgewertet
  - **Zuweisungsoperator**: “=“ Java, C/C++ oder “:=“ in Pascal
  - **Linke-Seite**: Bezeichner einer Variablen

- **Typkompatibilität**:
  Typ der linken Seite muss zum Typ des Zuweisungsausdrucks passen

Variablen in Zuweisungen

- Wir sprechen von der **rechten** und der **linken Seite** einer Zuweisung *(*engl.: right-hand side - **RHS**, left-hand side - **LHS***)* oder von dem **L-Wert** und **R-Wert** *(*engl.: lvalue, rvalue*)*
  - L-Wert:
    Ist ein Bezeichner einer **Variablen**, der ein Speicherplatz zugeordnet ist. Dort wird der neu berechnete Wert gespeichert.
  - R-Wert:
    Ist ein **Ausdruck**, der einen Wert liefert. Ein R-Wert kann nur rechts vom Zuweisungsoperator stehen.
  - Im folgenden Beispiel haben die beiden Auftreten des Bezeichners **a** unterschiedliche Bedeutung:

```
a = a + (3*i);
```

    - Auf der linken Seite ist das **a** das Ziel, in dem etwas gespeichert werden soll
    - Auf der rechten Seite ist es die Quelle eines Wertes, der mit anderen Werten in eine Berechnung einfließt.

Zuweisung in C/C++, Java etc.

- Der Gleichheitstest wird häufig mit der Zuweisung verwechselt:

```
saldo =  0  // Zuweisung
saldo == 0  // Gleichheit
saldo != 0  // Ungleichheit
```

```
antwort = 40
antwort += 2 // kurz für „antwort = antwort + 2;“
korrekt = (antwort == 42)
```

Operator	Funktion
==	Gleichheit
!=	Ungleichheit
=	Zuweisung

Überblick

1. Organisatorisches
2. Bedingungen in Kontrollsturkturen
3. **Imperative Programmierung**
   - Literale, Operationen und Ausdrücke
   - Variablen (und Zuweisungen)
   - **Prozeduren**

Beispiel: ein imperativer Algorithmus

Beispiel

1. Vergleiche zwei natürliche Zahlen *a* und *b*
2. Wenn *a* größer als *b* ist, setze das Ergebnis *max* gleich dem Wert von *a*
3. Sonst setze das Ergebnis *max* gleich dem Wert von *b*

Grundidee einer Prozedur (bei Karel zusammengesetzter Befehl)

Eine Anweisungsfolge

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

… erhält einen Namen und Parameter:

```
int maximum(int a, int b)
{
    int max;
      if (a > b)
 {
    max = a;
 }
     else
 {
    max = b;
 }
     return max;
}
```

… und kann **aufgerufen** werden:

```
…
int ergebnis = maximum(6, 9);
…
```

Prozeduren:
Grundeinheiten eines Programms

- In der imperativen Programmierung sind Prozeduren das mächtigste Abstraktionsmittel
- Prozeduraufrufe bestimmen die Aktivitäten eines Programms
- Realisieren einen **Algorithmus** mit den Mitteln einer Programmiersprache
- Sie sind eine **benannte Folge von Anweisungen**
  - Der **Namen** ist **„stellvertretend“ für diese Anweisungsfolge**
  - Einer Prozedur können beim Aufruf unterschiedliche Informationen mitgegeben werden: **Parameterübergabe**

Parametrisierung

- Um die Anweisungsfolgen für verschiedene Fälle zutreffend zu formulieren

- Parameter sind Speicheradressen von Speicherzellen, an denen die Eingabe- oder Ausgabedaten stehen

- Höhere Programmiersprachen verwenden das Konzept **getypter formaler Parameter**

Formale und aktuelle Eingabe-Parameter

Definierende Stelle:

Formale Parameter

```
int maximum(int a, int b)
    {
      int max;
      if (a > b)
      {
          max = a;
      }
      else
      {
          max = b;
      }
      return max;
    }
```

Aufrufende Stellen:

Aktuelle Parameter

```
...
int ergebnis = maximum(6, 9);
int ergebnis2 = maximum(ergebnis, 2*x);
```

- Im Kopf hat eine Prozedur sog. **formale Parameter** zur Datenübergabe
- **Beispiel:** a und b; beide sind vom Typ int

- Beim Aufruf erhält eine Prozedur **aktuelle Parameter**
- Bei **Eingabe-Parametern** sind dies **Ausdrücke**, hier jeweils vom **passenden Typ** int
- Die Werte der Ausdrücke werden den formalen Parametern **zugewiesen**

Regeln bei der Parameterübergabe

Beim Prozeduraufruf werden die Werte der aktuellen Parameter an die formalen übergeben.

Zur Übersetzungszeit wird überprüft:

- Der **Name** im Aufruf definiert die zu rufende Prozedur
- Die **Anzahl** der aktuellen Parameter muss gleich der Anzahl der formalen sein
- Die Bindung der Parameter wird entsprechend ihrer **Position** vorgenommen
- Die aktuellen Parameter müssen **typkompatibel** zu den formalen Parametern sein (d.h. zunächst typgleich)

Diese Regeln werden üblicherweise von einem **Kompiler** überprüft!

Ergebnisprozedur

- Prozeduren, die die programmiersprachliche Form einer Funktion haben, nennen wir **Ergebnisprozeduren**
  - Das Ergebnis kann direkt in einem Ausdruck verwendet werden

Beispiel:

```
...
int max = maximum(gibWert(), berechneWert(a,b));
...
```

  - Prozeduren mit einem Ergebnistyp ungleich **`void`** sind (vorläufig) die einzige Möglichkeit, Informationen von der gerufenen Prozedur an den Aufrufer zurück zu liefern

Formales und aktuelles Ergebnis

Prozedurdefinition

Formales Ergebnis

```
int maximum(int a, int b)
{
    int max;
    if (a > b)
    {
        max = a;
    }
    else
      ...
    return max;
 }
```

- Im Kopf kann eine Prozedur ein **formales Ergebnis** definieren
- Bei **void**, ist die Prozedur keine Ergebnisprozedur
- Eine Ergebnisprozedur **muss** mit **return** ein Ergebnis liefern

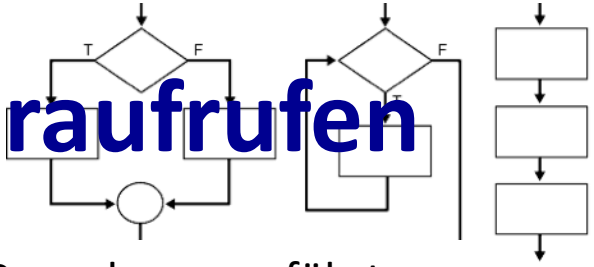
Prozeduraufruf

Aktuelle Ergebnisse

```
…
int ergebnis = maximum(6, 9);
int ergebnis2 = maximum(ergebnis, 2*x);
```

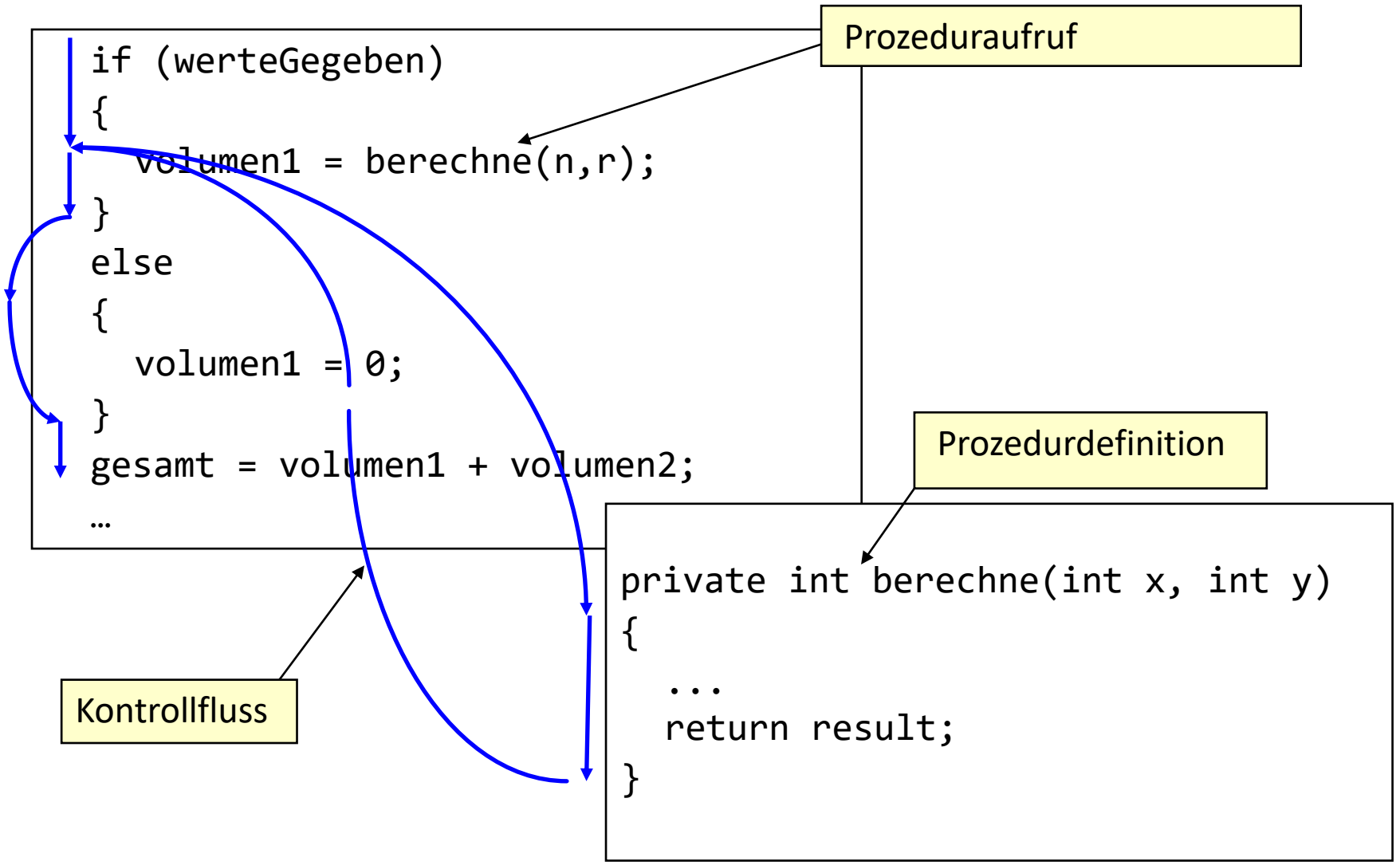
- Der Name der Ergebnisprozedur ist hier **stellvertretend** für das Ergebnis des Aufrufs

Kontrollfluss bei Prozeduraufrufen



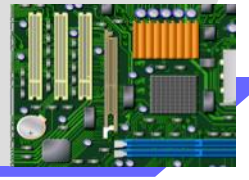
- Der **Prozeduraufruf** ist die explizite Anweisung, dass eine Prozedur ausgeführt werden soll.
- Eine Prozedur ist **aktiv**, nachdem sie gerufen wurde und in der Abarbeitung ihrer Anweisungen noch kein vordefiniertes Ende erreicht hat.
- Für den Prozeduraufruf in **sequenziellen imperativen** Sprachen ist charakteristisch:
  - Beim Aufruf wechselt die **Kontrolle** (d.h. die Abarbeitung von Anweisungen) vom Rufer zur Prozedur.
  - Dabei werden die (Werte der) **aktuellen Parameter** an die **formalen gebunden** (ihnen zugewiesen).
  - Prozeduren können **geschachtelt** aufgerufen werden. Dabei wird der Rufer unterbrochen, so dass die Kontrolle **immer nur bei einer Prozedur** ist; es entsteht eine **Aufrufkette**.
  - Nach der **Abarbeitung** der Prozedur kehrt die Kontrolle zum Rufer zurück; die Abarbeitung wird mit der Anweisung nach dem Aufruf fortgesetzt.

Kontrollfluss und Prozedur-Mechanismus

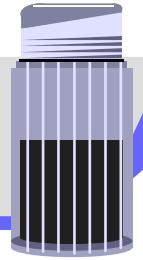


Zusammenfassung

1. Programme werden meistens in Maschinensprache (von Neumann Architektur) kompiliert, interpretiert und ausgeführt.



2. **Variablen** müssen deklariert werden und können dynamisch ihre Belegung durch Zuweisungen ändern.



3. **Literale** und **Variablen** werden für die Zuweisung einer **Variablen** oft durch **Operatoren** zu **Ausdrücken** verknüpft.

```
a = 3 + b;
```

4. Anweisungen und **Prozeduren** fassen Anweisungsfolgen zusammenfassen und bestimmen die **Ausführungsreihenfolge**.

5. **Prozeduren** können **parametrisiert** werden und können etwas zurückliefern.