- Der Wert eines Ausdrucks $u$ in Zustand $z \in \mathbf{Z}$ lässt sich damit auch leicht definieren (jetzt $W_{\mathbf{Z}}$ statt $W_{\sigma}$)
- Dazu erweitern wir die rekursive Definition von Folie 57 (Wert eines Ausdrucks):
  - Ist $u$ eine Variable oder Konstante $x$ und

    ist $(x, d) \in \mathbf{Z}$, $d \neq \omega$, so ist $W_{\mathbf{Z}}(u) = \mathrm{d}$;

    ist $(x, \omega) \in \mathbf{Z}$ oder $x \notin N(\mathbf{Z})$, so ist $W_{\mathbf{Z}}(u) = \omega$ (d.h. undefiniert).

- Bemerkung 1: Für die Fälle dass $u$ ein Eingabeparameter, ein Literal, die Anwendung eines Operators oder ein Aufruf einer Funktion (also einer Prozedur mit Rückgabewert ohne Seiteneffekte) ist, ist $W_{\mathbf{Z}}(u)$ entsprechend wie auf Folie 57 definiert.

- Bemerkung 2: Wir schreiben wieder $W(u)$ statt $W_{\mathbf{Z}}(u)$, wenn $\mathbf{Z}$ aus dem Kontext klar ist

- Sein ***a*** eine Vereinbarung einer Variablen oder Konstanten oder eine Zuweisung. ***a*** bewirkt eine *Zustandsänderung* von Zustand **Z** in den Nachfolgezustand **Z'** wie folgt:
  - Ist ***a*** eine Vereinbarung einer Variablen oder Konstanten $x$ so ist

    $\mathbf{Z'} = \mathbf{Z} \cup \{ (x, \omega) \}$ (unter der Annahme $x \notin N(\mathbf{Z})$)
  - Ist ***a*** eine Zuweisung von einem Ausdruck $u$ zu einer Variablen oder zu einer Konstanten $x$, also „x = u;“, mit $x \in N(\mathbf{Z})$ und ist $W(u) \neq \omega$, so ist

    $\mathbf{Z'} = ( \mathbf{Z} \setminus \{(x, \omega)\} ) \cup \{ (x, W(u)) \}$
- Bemerkung: **Z** enthält bereits die Eingabevariablen $V(\mathbf{Z})$ des Algorithmus, der gerade ausgeführt wird
- Wir schreiben $\mathbf{Z} \rightarrow_{a} \mathbf{Z'}$, um auszudrücken, dass wir **Z'** aus **Z** durch Anwendung von ***a*** erhalten
- Ist ***A*** eine Folge von Variablen-/Konstantenvereinbarungen oder Wertzuweisungen ***a***1; … ***a***$k$; und es gilt $\mathbf{Z} \rightarrow_{a1} \ldots \rightarrow_{ak} \mathbf{Z'}$, so heißt **Z'** Nachfolgezustand von **Z** bzgl. ***A*** und wir schreiben $\mathbf{Z} \rightarrow_{A} \mathbf{Z'}$

# 4.4 Imperative Algorithmen

## 4.4.2 Prozeduren



- Beispiel

```
public void beispielMethode1(int a) {
  int x;
  int y = 5;
  x = 0;
  final int z = 3;
  x += 7 + y;
  y = y + z – 2;
  x = x * y;
  x = x/z;
  x += a;
  a = x + y – z;
}
```

- Ausführung des Aufrufs `beispielMethode1(1)`

| Anweisung | x | y | z | a | Zustand **Z** |
|---|---|---|---|---|---|
| Beim Aufruf | | | | 1 | $\mathbf{Z} = \{(a,1)\}$ |
| `int x;` | | | | 1 | $\mathbf{Z}_1 = \{(x, \omega),(a,1)\}$ |
| `int y = 5;` | | 5 | | 1 | $\mathbf{Z}_2 = \{(x,\omega),(y,5),(a,1)\}$ |
| `x = 0;` | 0 | 5 | | 1 | $\mathbf{Z}_3 = \{(x,0),(y,5),(a,1)\}$ |
| `final int z = 3;` | 0 | 5 | \| 3 \| | 1 | $\mathbf{Z}_4 = \{(x,0),(y,5),(z,3),(a,1)\}$ |
| `x += 7 + y;` | 12 | 5 | \| 3 \| | 1 | $\mathbf{Z}_5 = \{(x,12),(y,5),(z,3),(a,1)\}$ |
| `y = y + z - 2;` | 12 | 6 | \| 3 \| | 1 | $\mathbf{Z}_6 = \{(x,12),(y,6),(z,3),(a,1)\}$ |
| `x = x * y;` | 72 | 6 | \| 3 \| | 1 | $\mathbf{Z}_7 = \{(x,72),(y,6),(z,3),(a,1)\}$ |
| `x = x/z;` | 24 | 6 | \| 3 \| | 1 | $\mathbf{Z}_8 = \{(x,24),(y,6),(z,3),(a,1)\}$ |
| `x += a;` | 25 | 6 | \| 3 \| | 1 | $\mathbf{Z}_9 = \{(x,25),(y,6),(z,3),(a,1)\}$ |
| `a = x + y - z;` | 25 | 6 | \| 3 \| | 28 | $\mathbf{Z}' = \{(x,25),(y,6),(z,3),(a,28)\}$ |

= Zettel existiert  | | = Konstante (nicht radierbar)

# 4.4 Imperative Algorithmen

## 4.4.2 Prozeduren



- Beispiel:

```
public void beispielMethode2(int a) {
  int x = a;
  int y = a;
  beispielMethode1(a);
  return x + y + a;
}
```

- Was passiert hier bei der Ausführung des Aufrufs von `beispielMethode2(1)`?
  - Was für einen Effekt hat der Aufruf von `beispielMethode1` auf `a`?
  - Welche Variablen `x` und `y` sind wann sichtbar/gültig (beide Methoden benutzen Variablen mit den Namen `x` und `y`)?

# 4.4 Imperative Algorithmen

## 4.4.2 Prozeduren



- Zur Verarbeitung der Parameter von Prozeduren kennen Programmiersprachen zwei grundsätzliche Möglichkeiten:
- *Call-by-value*
  Im Aufruf `methodenName(parameter)` wird für `parameter` im Methoden-Block eine neue Variable angelegt, in die der Wert von `parameter` geschrieben wird. Auf diese Weise bleibt die ursprüngliche Variable `parameter` von Anweisungen innerhalb der Methode unberührt.
- *Call-by-reference*
  Im Aufruf `methodenName(parameter)` wird die Variable `parameter` weiter verwendet. Wenn innerhalb der Methode der Wert von `parameter` verändert wird, hat das auch Auswirkungen außerhalb der Methode.

***Achtung: call-by-reference ist daher eine potentielle Quelle unbeabsichtigter Seiteneffekte.***

# 4.4 Imperative Algorithmen

## 4.4.2 Prozeduren



- Aufruf von `beispielMethode2(1)` mit *call-by-value*:

```
1 public int beispielMethode2(int a) {
2     int x = a;
3     int y = a;
4     beispielMethode1(a);
5     return x + y + a;
  }
```

```
public void beispielMethode1(int a) {
   int x; int y = 5; x = 0;
   final int z = 3; x += 7 + y;
   y = y + z – 2; x = x * y;
   x = x/z; x += a; a = x + y – z;
}
```

  - Zustand nach Zeile 3 ist **Z** = {(a,1), (x,1), (y,1)}
  - Beim Aufruf von `beispielMethode1(a)` wird ein ***zweiter*** Zettel mit Namen a (zur Unterscheidung a‘ genannt) und Inhalt 1 angelegt (a‘,1), der unabhängig vom existierenden Zettel (a,1) ist und auch nur im Rumpf von `beispielMethode1` gilt
  - Innerhalb des Rumpfes von `beispielMethode1` werden neue Zettel x‘, y‘ und z angelegt, die unabhängig von x und y sind.
  - Nach Abarbeitung von `beispielMethode1` an Zeile 5 ist daher wieder der Zustand **Z** = {(a,1), (x,1), (y,1)}, die Zettel a, x und y wurden ***nicht*** verändert
  - Rückgabewert von `beispielMethode2` in Zeile 5 ist demnach 1+1+1 = 3

- Aufruf von `beispielMethode2(1)` mit *call-by-reference*:

```
1 public int beispielMethode2(int a) {
2     int x = a;
3     int y = a;
4     beispielMethode1(a);
5     return x + y + a;
  }
```

```
public void beispielMethode1(int a) {
   int x; int y = 5; x = 0;
   final int z = 3; x += 7 + y;
   y = y + z – 2; x = x * y;
   x = x/z; x += a; a = x + y – z;
}
```

- Zustand nach Zeile 3 ist wieder **Z** = {(a,1), (x,1), (y,1)}
- Beim Aufruf von `beispielMethode1(a)` wird ***kein*** neuer Zettel angelegt sondern der existierende Zettel (a,1) im Rumpf von `beispielMethode1` verwendet
- Innerhalb des Rumpfes von `beispielMethode1` werden neue Zettel x', y' und z angelegt, die unabhängig von x und y sind.
- Nach Abarbeitung von `beispielMethode1` an Zeile 5 ist der Zettel a nun durch `beispielMethode1` verändert worden (beim Aufruf `beispielMethode1(1)` wird am Ende 28 auf den Zettel a geschrieben)
- Daher ist nach Zeile 4 der Zustand **Z** = {(a,28), (x,1), (y,1)}, die Zettel x und y wurden ***nicht*** verändert
- Rückgabewert von `beispielMethode2` in Zeile 5 ist demnach 1+1+28 = 30

- ***Java wertet Parameter call-by-value aus***. Für den Aufruf der Methode swap im folgenden Beispiel werden also Kopien der Variablen x und y angelegt (im Rumpf von swap heißen diese Zettel dann i und j).

```
public class Exchange
{
  public static void swap(int i, int j)
  {
    int c = i;
    i = j;
    j = c;
  }

  public static void main(String[] args)
  {
    int x = 1;
    int y = 2;
    swap(x,y);
    System.out.println(x); // Ausgabe?
    System.out.println(y); // Ausgabe?
  }
}
```

- Aufruf von `main`:

  Zunächst werden (x,1),(y,2) angelegt
  (wir ignorieren „args“ im Kopf von `main`)

  Beim Aufruf von `swap` werden neue Zettel angelegt: i als Kopie für x und j als Kopie für y
  (i,1),(j,2)
  Dann
  (i,1),(j,2),(c,1)
  (i,2),(j,2),(c,1)
  (i,2),(j,1),(c,1)

```
public class Exchange
{
  public static void swap(int i, int j)
  {
    int c = i;
    i = j;
    j = c;
  }

  public static void main(String[] args)
  {
    int x = 1;
    int y = 2;
    swap(x,y);
    System.out.println(x); // Ausgabe?
    System.out.println(y); // Ausgabe?
  }
}
```

Alles schön und gut, aber nach dem Aufruf (beim `System.out`) ist

x = 1 und y = 2

d.h. `swap` hat keinerlei Wirkung auf x und y in `main`.

- Bemerkungen:
  - Unsere Definitionen für den Wert von Ausdrücken und Zustandsübergänge implementiert call-by-value
  - Prozeduren (in Java: statische Methoden) sind ein Mittel zur Abstraktion und daher zur Strukturierung von Programmen
    - Verschiedene Teilaufgaben eines Algorithmus können voneinander getrennt werden, indem dafür eigene Methoden geschrieben werden
    - Diese Methoden können theoretisch beliebig in anderen Algorithmen wiederverwendet werden
    - Wir müssen sogar noch nicht einmal die Details der Implementierung einer Methode kennen (den Methodenrumpf) solange die Semantik klar ist und die Signatur (d.h. wie ich die Methode benutzen/aufrufen kann) bekannt ist
    - (dies ist übrigens mit den Grundoperationen genauso)

# Übersicht

# 4.4 Imperative Algorithmen

## 4.4.3 Verzweigung und Iteration



- Um den Fluss von Anweisungen innerhalb eines Algorithmus steuern zu können, gibt es die bereits angesprochenen Konzepte der *Verzweigung* (bedingte Anweisung) und der *Iteration* (Wiederholungsanweisung)
- Wir lernen im folgenden die Umsetzungen dieser Konzepte in Java kennen
- Durch bedingte Anweisungen kann man nun auch Prozeduren rekursiv formulieren

- Java erlaubt zwei Formen von bedingten Anweisungen:

  Eine *einfache* Verzweigung:

```
if (<Bedingung>) <Anweisungsfolge>
```

  Eine *zweifache* Verzweigung:

```
if (<Bedingung>)
    <Anweisungsfolge1>
else
    <Anweisungsfolge2>
```

- Wobei
  - `<Bedingung>` ein Ausdruck vom Typ **boolean** ist,
  - `<Anweisungsfolge>`, `<Anweisungsfolge1>` und `<Anweisungsfolge2>` jeweils eine einzelne (ggfs. auch leere) Anweisung oder einen Block mit mehreren Anweisungen darstellen (dann mit {}).

- Beispiel: Der (rekursive) Algorithmus zur Berechnung der Fakultät einer natürlichen Zahl $n \in \mathbb{N}_0$ kann in Java durch folgende Methode umgesetzt werden:

```
public static int fakultaet01(int n)
{
  if(n==0)
  {
    return 1;
  }
  else
  {
    return n * fakultaet01(n-1);
  }
}
```

Auch wenn der Algorithmus auf den ersten Blick funktional ausschaut, er ist imperativ (*Warum*?)

- Bedingte Anweisungen mit mehr als zwei Zweigen müssen in Java durch Schachtelung mehrerer **`if`**-Konstrukte ausgedrückt werden:

```
if (<Bedingung1>)
      <Anweisungsfolge1>
else if (<Bedingung2>)
   <Anweisungsfolge2>
.
.
.
else if (<BedingungN>)
      <AnweisungsfolgeN>
else
      <AnweisungsfolgeN+1>
```

- *Achtung*: Fehlerquellen durch nicht gesetzte Blockklammern
  - Dangling Else

```
if (a)
  if (b)
      s1;
else
  s2;
```

Frage: Zu welchem if-Statement gehört der else-Zweig?

Antwort: Zur inneren Verzweigung `if (b)`. (Die (*falsche*!) Einrückung ist belanglos für den Compiler und verführt den menschlichen Leser hier, das Programm falsch zu interpretieren.)

Mit Blockklammern wäre das nicht passiert:

```
if (a) {
  if (b) { s1; }
  else { s2; }
}
```

oder

```
if (a) {
  if (b) { s1; }
}
else { s2; }
```

- *Achtung*: Fehlerquellen durch nicht gesetzte Blockklammern (Teil 2)
  - Beispiel: Ein Kunde hebt einen Betrag (`betrag`) von seinem Konto ab (die Variable `kontoStand` speichert den aktuellen Kontostand). Falls der Betrag nicht gedeckt ist, wird eine Überziehungsgebühr fällig (Konstante `UEBERZIEH_GEBUEHR`). Die fälligen Gebühren werden über einen bestimmten Zeitraum akkumuliert (`gebuehren`) und am Ende des Zeitraums in Rechnung gestellt. Was ist falsch in folgender Berechnung?

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

# 4.4 Imperative Algorithmen

## 4.4.3 Verzweigung und Iteration



- Problem im vorherigen Beispiel:
  Überziehungsgebühr wird immer verlangt, auch wenn das Konto gedeckt ist.
- Lösung: Blockklammern setzen.

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

- Nun wird die Überziehungsgebühr nur verlangt, wenn das Konto *nicht* gedeckt ist.

- Spezielle Mehrfachverzweigungen als *Sprunganweisungen*:
  - In Java gibt es eine weitere Möglichkeit, spezielle Mehrfachverzweigungen auszudrücken.
  - Die sog. **`switch`**-Anweisung funktioniert allerdings etwas anders als bedingte Anweisungen.
  - Syntax:

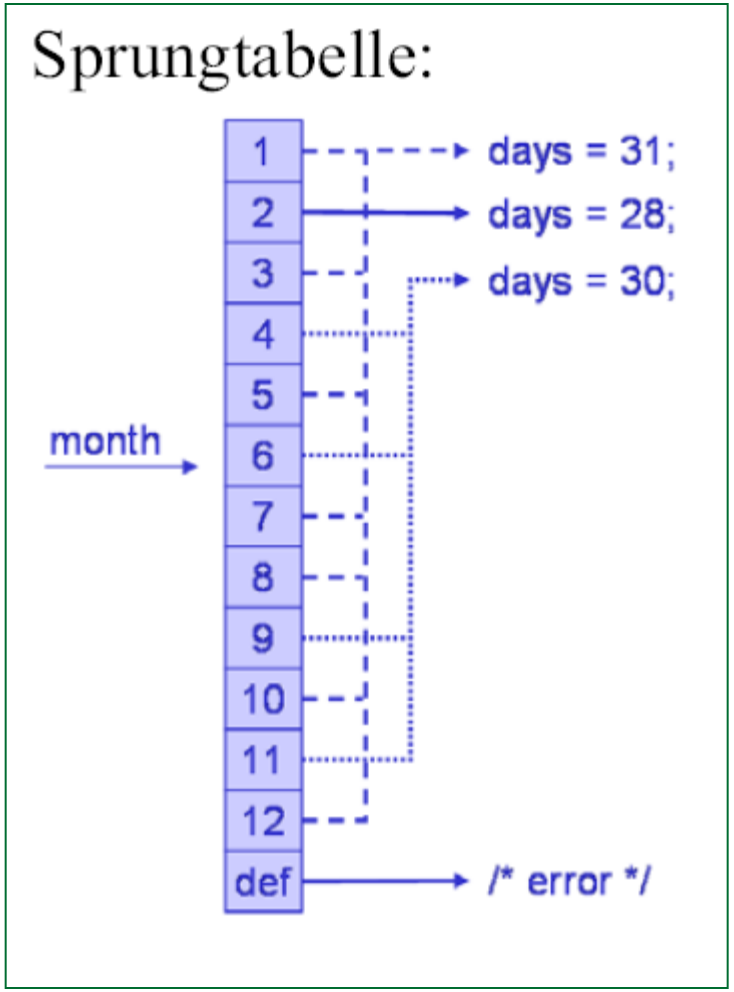
```
switch (ausdruck)
{
  case konstante1 : anweisung1
  case konstante2 : anweisung2
    .
    .
    .
  default : anweisungN
}
```

- Bedeutung:
  - Abhängig vom Wert des Ausdrucks ausdruck wird die *Sprungmarke* angesprungen, deren Konstante mit dem Wert von `ausdruck` übereinstimmt.
  - Die Konstanten und der Ausdruck müssen den selben Typ haben.
  - Die Anweisungen nach der Sprungmarke werden ausgeführt.
  - Die optionale **`default`**-Marke wird dann angesprungen, wenn keine passende Sprungmarke gefunden wird.
  - Fehlt die **`default`**-Marke und wird keine passende Sprungmarke gefunden, so wird keine Anweisung innerhalb der **`switch`**-Anweisung ausgeführt.

- Besonderheiten:
  - Der Ausdruck `ausdruck` darf nur vom Typ **`byte`**, **`short`**, **`int`** oder **`char`** sein.
  - Die **`case`**-Marken sollten alle verschieden sein, müssen aber nicht.
  - ***Achtung***: Wird zu einer Marke gesprungen, werden alle Anweisungen hinter dieser Marke ausgeführt. Es erfolgt *keine* Unterbrechung, wenn das nächste Label erreicht wird, sondern es wird dort fortgesetzt! Dies ist eine beliebte Fehlerquelle!
  - Eine Unterbrechung kann durch die Anweisung **`break;`** erzwungen werden. Jedes **`break`** innerhalb einer **`switch`**-Anweisung verzweigt zum Ende der **`switch`**-Anweisung.
  - Nach einer Marken-Definition **`case`** muss nicht zwingend eine Anweisung stehen.

- Beispiel

```
switch (month)
{
  case 1: case 3: case 5: case 7:
    case 8: case 10: case 12:
      days = 31; break;
  case 4: case 6: case 9: case 11:
      days = 30; break;
  case 2:
      if(leapYear)
      {
        days = 29;
      }
      else
      {
        days = 28;
      }
      break;
    default: /* error */
}
```



- Bei der Iteration gibt es zunächst die *bedingte Wiederholung* (*bedingte Schleifen*):
  - Java kennt mehrere Arten von bedingten Schleifen.
  - Schleifen mit dem Schlüsselwort `while`:
  - Die klassische While-Schleife:
    `while (<Bedingung>) <Anweisungsfolge>`
  - Die Do-While-Schleife:
    `do <Anweisungsfolge> while (<Bedingung>);`
  - Dabei bezeichnet `<Bedingung>` einen Ausdruck vom Typ **`boolean`** und `<Anweisungsfolge>` ist eine einzelne (ggfs. auch leere) Anweisung oder einen Block mit mehreren Anweisungen darstellen (dann mit {}).
  - Unterschied: `<Anweisungsfolge>` wird einmal *vor* (Do-While) bzw. *nach* (While) der Überprüfung von `<Bedingung>` ausgeführt.
  - Ist `<Bedingung>` **`false`**, wird die Schleife verlassen.

- Beispiele: nicht-rekursive Algorithmen für die Fakultät mit While:

```
public static int fakultaet02(int n)
{
  int erg = 1;
  while (n > 0)
  {
    erg = erg * n;
    n--;
  }
  return erg;
}
```

  - Variante

```
public static int fakultaet03(int n)
{
  int erg = 1;
  int laufVariable = 1;
  while (laufVariable <= n)
  {
    erg = erg * laufVariable;
    laufVariable++;
  }
  return erg;
}
```

– Variante mit Do-While

```
public static int fakultaet04(int n)
{
  int erg = 1;
  int laufVariable = 1;
  do
  {
    erg = erg * laufVariable;
    laufVariable++;
  }
  while (laufVariable < n);
  return erg;
}
```