Programmierung mit Go

Reiner Hüchting

4. Oktober 2023

Übersicht

Grundlagen – Übersicht

Grundlagen – Hallo Welt

Das erste Programm

```
1  package main
2
3  import "fmt"
4
5  func main() {
6    fmt.Println("Hello␣World!")
7  }
```

Zeile 1: Definition des Pakets, zu dem die Datei gehört.

- Jedes Programm gehört zu einem Paket.
- Dient zur Strukturierung des Codes, sobald er komplexer wird.

Grundlagen – Hallo Welt

Das erste Programm

```
1  package main
2
3  import "fmt"
4
5  func main() {
6    fmt.Println("Hello␣World!")
7  }
```

Zeile 3: Import-Statement

- Importiert ein anderes Paket. (Hier: `fmt` für *format*).
- Wird für die Ausgabe benötigt.

Grundlagen – Hallo Welt

Das erste Programm

```
1  package main
2
3  import "fmt"
4
5  func main() {
6    fmt.Println("Hello␣World!")
7  }
```

ab Zeile 5: `main`-Funktion

- Jedes Programm muss eine `main`-Funktion enthalten.
- Wird beim Start des Programms ausgeführt.

Grundlagen – Hallo Welt

Das erste Programm

```
1  package main
2
3  import "fmt"
4
5  func main() {
6    fmt.Println("Hello␣World!")
7  }
```

Zeile 6: Ausgabe

- `fmt.Println` gibt aus, was in den Klammern steht.
- `fmt` ist ein Paketname, `Println` eine Funktion.

Grundlagen – Ein-/Ausgabe

Wichtiger Aspekt: Interaktion mit dem Benutzer

- Geschieht über das Package `fmt`.
  - `fmt` steht für *format*.
  - Bietet Funktionen zum Einlesen und Ausgeben von Daten.
- Schon bekannt: `fmt.Println()`.
- `fmt.Scan()` liest eine Eingabe ein.

Grundlagen – Ein-/Ausgabe

Einlesen von Benutzereingaben

```
1  package main
2
3  import "fmt"
4
5  func main() {
6    var n int
7    fmt.Print("Bitte␣eine␣Zahl␣eingeben:␣")
8    fmt.Scan(&n)
9    fmt.Print("Ihre␣Lieblingszahl:␣", n)
10 }
```

Grundlagen – Ein-/Ausgabe

... mit Überprüfung der Eingabe.

```
1  func main() {
2    var n int
3    fmt.Print("Bitte␣eine␣Zahl␣eingeben:␣")
4    fmt.Scan(&n)
5
6    if n != 42 {
7      fmt.Println("Das␣war␣falsch!")
8      return
9    }
10   fmt.Print("Ihre␣Lieblingszahl:␣", n)
11 }
```

Grundlagen – Variablen

Wichtige Bestandteile von Programmen: Variablen

- Variablen sind Speicherplätze für Werte.
- Müssen deklariert werden.
- Anschließend können darin Werte gespeichert werden und man kann mit diesen Werten rechnen.

Technische Sicht

- Variablen sind Speicherbereiche im *Arbeitsspeicher*.
- Die Größe des Bereichs hängt vom Typ der Variable ab.
- Der Typ einer Variable muss bei der Deklaration klar sein.
  - Notwendig, um den Speicher korrekt zu reservieren.
  - Nützlich, um das Programm vorab auf Fehler zu überprüfen.

Grundlagen – Variablen

Integer-Variablen

```
1  func IntVariables() {
2    var n int // Variablendeklaration
3    n = 42    // Variablenzuweisung
4    k := 23   // Kurzschreibweise für
         Deklaration und Zuweisung
5
6    fmt.Println(n, k, n+k)
7  }
```

- Deklaration: Reservieren von Speicher
- Rechnen mit den Werten ist möglich.

Grundlagen – Variablen

String-Variablen

```
1  func StringVariables() {
2      s := "Hallo"
3      t := "Welt"
4
5      st := s + "␣" + t // Verkettung der
           Strings
6
7      fmt.Println(st)
8  }
```

- Wie bei Integern, nur der Typ ist anders.
- Auch mit Strings kann gerechnet werden.

Grundlagen – Variablen

Listen-Variablen

```
1  func ListVariables() {
2    var l []int // leere Liste
3    l = append(l, 1, 2, 3, 4, 5)
4
5    fmt.Println(l)       // komplett ausgeben
6    fmt.Println(l[1])    // Zweites Element
         ausgeben
7    fmt.Println(l[1:3]) // Teil-Liste ausgeben
8    l[1] = 42            // Wert ändern
9
10   fmt.Println(l)
11 }
```

- Listen sind (theoretisch) unbegrenzt.

Grundlagen – Beispiel: Fakultät einer Zahl

Ziel: Berechne 5!

- Es gilt: $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$
- Kann schrittweise mit Zwischenergebnissen berechnet werden:

Berechnung	**Zwischenergebnis**
1	1
$2 \cdot 1$	2
$3 \cdot 2$	6
$4 \cdot 6$	24
$5 \cdot 24$	120

- So ähnlich würde man es auf Papier berechnen.
- Ziel: Automatisiere die Berechnung.

Grundlagen – Beispiel: Fakultät einer Zahl

Umsetzung der Schritt-Für-Schritt-Berechnung

```
1    result := 1 // Startwert
2    result = result * 2
3    result = result * 3
4    result = result * 4
5    result = result * 5
```

Berechnung	Zwischenergebnis
1	1
$2 \cdot 1$	2
$3 \cdot 2$	6
$4 \cdot 6$	24
$5 \cdot 24$	120

Grundlagen – Beispiel: Fakultät einer Zahl

Umsetzung der Schritt-Für-Schritt-Berechnung

```
1    result := 1 // Startwert
2    result = result * 2
3    result = result * 3
4    result = result * 4
5    result = result * 5
```

- Problem: Die Berechnung ist sehr starr.
- Umständlich aufzuschreiben und anzupassen.
- Lösung: Schleifen

Grundlagen – Beispiel: Fakultät einer Zahl

Schrittweise Berechnung wie zuvor

```
1    result := 1 // Startwert
2    result = result * 2
3    result = result * 3
4    result = result * 4
5    result = result * 5
```

Berechnung mit Schleife

```
1    result := 1 // Startwert
2    for i := 2; i <= 5; i++ {
3      result = result * i
4    }
```

Grundlagen – Beispiel: Fakultät einer Zahl

Berechnung mit Schleife

```
1    result := 1 // Startwert
2    for i := 2; i <= 5; i++ {
3      result = result * i
4    }
```

Vorteile:

- kompakterer Code
- Nur an einer Stelle ändern, um n zu ändern.
- Nächster Schritt: n durch eine Variable ersetzen.

Grundlagen – Beispiel: Fakultät einer Zahl

Berechnung von 5!

```
1    result := 1 // Startwert
2    for i := 2; i <= 5; i++ {
3      result = result * i
4    }
```

Berechnung von n!

```
1    result := 1 // Startwert
2    for i := 2; i <= n; i++ {
3      result = result * i
4    }
```

Grundlagen – Beispiel: Fakultät einer Zahl

Berechnung von $n!$

```
1    result := 1 // Startwert
2    for i := 2; i <= n; i++ {
3      result = result * i
4    }
```

Vorteile:

- Flexibel, n kann z.B. eingelesen oder berechnet werden.

Nachteile:

- Code kann noch nicht wiederverwendet werden.
- Muss ggf. an mehrere Stellen kopiert werden.
- Nächster Schritt: Funktionen

Grundlagen – Beispiel: Fakultät einer Zahl

Berechnung von $n!$

```
1  func FactorialNLoop(n int) int {
2    result := 1 // Startwert
3    for i := 2; i <= n; i++ {
4      result = result * i
5    }
6
7    return result
8  }
```

Beobachtungen:

- Code ist in einer Funktion *eingepackt*.
- Die Funktion kann an anderer Stelle verwendet werden.

Grundlagen – Beispiel: Fakultät einer Zahl

Alternative: Rückwärts laufende Schleife

```
1  func FactorialNLoopBackwards(n int) int {
2    result := 1 // Startwert
3    for i := n; i >= 1; i-- {
4      result = result * i
5    }
6
7    return result
8  }
```

Beobachtungen:

- Die Schleife hat einen Zähler und eine Abbruchbedingung.
- Eines der wichtigsten Konzepte in der Programmierung!

Grundlagen – Beispiel: Fakultät einer Zahl

Alternative: Rekursive Berechnung

```
1  func FactorialNRecursive(n int) int {
2    if n == 0 {
3      return 1
4    }
5    return n * FactorialNRecursive(n-1)
6  }
```

Basiert auf folgender Beobachtung:

$$
\begin{aligned}
n! &= n \cdot \boxed{(n-1) \cdot (n-2) \cdots 2 \cdot 1} \\
&= n \cdot \boxed{(n-1)!}
\end{aligned}
$$

Grundlagen – Schleifen

Genereller Aufbau einer Schleife

```
1 for <Start>; <Bedingung>; <Schritt> {
2   // Schleifenkörper
3 }
```

Erläuterungen:

- Oft wird ein Zähler, der in jedem Schleifendurchlauf inkrementiert wird.
- Die Schläufe läuft solange, wie die Bedingung erfüllt ist.
- Der Zähler ist meist eine `int`-Variable und startet bei 0.
- Schleifen können aber auch Rückwärts laufen oder komplexere Bedingungen haben.

Grundlagen – Schleifen

Beispiel: Zahlen auflisten

```
1  func ListNumbers(n int) {
2    for i := 0; i < n; i++ {
3      fmt.Println(i)
4    }
5  }
```

Erläuterungen:

- Gibt die Zahlen von 0 bis $n - 1$ auf der Konsole aus.
- Hat dabei n Schleifendurchläufe.

Grundlagen – Schleifen

Beispiel: Zahlen rückwärts auflisten

```
1  func ListNumbersBackwards(n int) {
2    for i := n; i > 0; i-- {
3      fmt.Println(i)
4    }
5  }
```

Erläuterungen:

- Gibt die Zahlen von n bis 1 rückwärts auf der Konsole aus.
- Hat dabei n Schleifendurchläufe.

Grundlagen – Schleifen

Beispiel: Gerade Zahlen auflisten

```
1  func ListEvenNumbers(n int) {
2    for i := 0; i < n; i++ {
3      if i%2 == 0 {
4        fmt.Println(i)
5      }
6    }
7  }
```

Erläuterungen:

- Gibt die geraden Zahlen von 0 bis $n - 1$ auf der Konsole aus.

Grundlagen – Schleifen

Beispiel: Vielfache auflisten

```
1  func ListMultiplesOf(m, n int) {
2    for i := 0; i < n; i++ {
3      if i%m == 0 {
4        fmt.Println(i)
5      }
6    }
7  }
```

Erläuterungen:

- Gibt alle Vielfachen von m auf der Konsole aus, die kleiner als $n - 1$ sind.

Grundlagen – Schleifen

Beispiel: Vielfache auflisten

```
1  func ListMultiplesOfBigSteps(m, n int) {
2    for i := 0; i < n; i += m {
3      fmt.Println(i)
4    }
5  }
```

Erläuterungen:

- Gibt alle Vielfachen von m auf der Konsole aus, die kleiner als $n - 1$ sind.
- Wie zuvor, aber eine Schleife, die größere Schritte macht.

Grundlagen – Schleifen

Beispiel: Summe berechnen

```
1  func SumN(n int) int {
2    sum := 0
3    for i := 1; i <= n; i++ {
4      sum += i
5    }
6
7    return sum
8  }
```

Erläuterungen:

- Berechnet die Summe der Zahlen von 1 bis n.
- Gibt nichts aus, sondern hat ein Rechenergebnis, das mit `return` zurückgegeben wird.

Grundlagen – Schleifen

Beispiel: Summe berechnen (rekursiv)

```
1  func SumNRecursive(n int) int {
2    if n == 0 {
3      return 0
4    }
5    return n + SumNRecursive(n-1)
6  }
```

Erläuterungen:

- Berechnet die Summe der Zahlen von 1 bis n.
- Rekursiver Ansatz, ähnlich wie schon bei der Fakultät.

Grundlagen – Schleifen

Beispiel: Primzahltest

```
1  func IsPrime(n int) bool {
2    for i := 2; i < n; i++ {
3      if n%i == 0 {
4        return false
5      }
6    }
7    return n >= 1
8  }
```

Erläuterungen:

- Prüft für alle i zwischen 2 und $n-1$, ob n durch i teilbar ist.
- Gibt `true` zurück, wenn n eine Primzahl ist, sonst `false`.

Grundlagen – Schleifen

Beispiel: While-Schleife

```
1  func SumWhileN(n int) int {
2    sum, i := 0, 1
3    for i <= n {
4      sum += i
5      i++
6    }
7    return sum
8  }
```

Erläuterungen:

- Berechnet wieder die Summe der Zahlen von 1 bis n.
- Verwendet dafür eine `while`-Schleife.
- Die Schleife läuft solange, wie die Bedingung erfüllt ist.