Go-Grundlagen

Reiner Hüchting

4. Dezember 2023

Inhalt

Hallo Welt

Das erste Programm

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

Hallo Welt

Das erste Programm

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

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

- Jedes Programm gehört zu einem Paket.
- Dient zur Strukturierung von komplexerem Code.

Hallo Welt

Das erste Programm

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

Zeile 3: Import-Statement

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

Hallo Welt

Das erste Programm

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

ab Zeile 5: `main`-Funktion

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

Hallo Welt

Das erste Programm

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

Zeile 6: Ausgabe

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

Ein-/Ausgabe

Wichtiger Aspekt: Interaktion mit dem Benutzer

- Geschieht über das Package `fmt`.

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.

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()`.

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.

Ein-/Ausgabe

Einlesen von Benutzereingaben

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

Ein-/Ausgabe

... mit Überprüfung der Eingabe.

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

Variablen

Wichtige Bestandteile von Programmen: Variablen

- Variablen sind Speicherplätze für Werte.

Variablen

Wichtige Bestandteile von Programmen: Variablen

- Variablen sind Speicherplätze für Werte.
- Müssen deklariert werden.

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.

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*.

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.

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.

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.

Variablen

Integer-Variablen

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

Variablen

Integer-Variablen

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

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

Variablen

String-Variablen

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

Variablen

String-Variablen

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

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

Variablen

Listen-Variablen

```
0  func ListVariables() {
1      var l []int // leere Liste
2      l = append(l, 10, 20, 30, 40, 50)
3
4      fmt.Println(l)       // komplett
           ausgeben
5      fmt.Println(l[1])    // Zweites
           Element ausgeben
6      fmt.Println(l[1:3]) //
           Teil-Liste ausgeben
7      l[1] = 42            // Wert
           ändern
8
9      fmt.Println(l)
10 }
```

Variablen

Listen-Variablen

```
0  func ListVariables() {
1      var l []int // leere Liste
2      l = append(l, 10, 20, 30, 40, 50)
3
4      fmt.Println(l)       // komplett
           ausgeben
5      fmt.Println(l[1])    // Zweites
           Element ausgeben
6      fmt.Println(l[1:3]) //
           Teil-Liste ausgeben
7      l[1] = 42            // Wert
           ändern
8
9      fmt.Println(l)
10 }
```

Beispiel: Fakultät einer Zahl

Ziel: Berechne 5!

- Es gilt: $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$

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

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.

Beispiel: Fakultät einer Zahl

Umsetzung der Schritt-Für-Schritt-Berechnung

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

Beispiel: Fakultät einer Zahl

Umsetzung der Schritt-Für-Schritt-Berechnung

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

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

Beispiel: Fakultät einer Zahl

Umsetzung der Schritt-Für-Schritt-Berechnung

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

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

Beispiel: Fakultät einer Zahl

Umsetzung der Schritt-Für-Schritt-Berechnung

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

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

Beispiel: Fakultät einer Zahl

Schrittweise Berechnung wie zuvor

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

Beispiel: Fakultät einer Zahl

Schrittweise Berechnung wie zuvor

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

Berechnung mit Schleife

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

Beispiel: Fakultät einer Zahl

Berechnung mit Schleife

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

Beispiel: Fakultät einer Zahl

Berechnung mit Schleife

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

Vorteile:

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

Beispiel: Fakultät einer Zahl

Berechnung von 5!

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

Beispiel: Fakultät einer Zahl

Berechnung von 5!

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

Berechnung von $n!$

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

Beispiel: Fakultät einer Zahl

Berechnung von $n!$

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

Beispiel: Fakultät einer Zahl

Berechnung von $n!$

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

Vorteile:

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

Beispiel: Fakultät einer Zahl

Berechnung von $n!$

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

Vorteile:

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

Nachteile:

- Code kann noch nicht wiederverwendet werden.
- Muss ggf. an mehrere Stellen kopiert werden.

Beispiel: Fakultät einer Zahl

Berechnung von $n!$

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

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

Beispiel: Fakultät einer Zahl

Berechnung von $n!$

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

Beispiel: Fakultät einer Zahl

Berechnung von $n!$

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

Beobachtungen:

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

Beispiel: Fakultät einer Zahl

Alternative: Rückwärts laufende Schleife

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

Beispiel: Fakultät einer Zahl

Alternative: Rückwärts laufende Schleife

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

Beobachtungen:

- Die Schleife hat einen Zähler und eine Abbruchbedingung.

Beispiel: Fakultät einer Zahl

Alternative: Rückwärts laufende Schleife

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

Beobachtungen:

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

Beispiel: Fakultät einer Zahl

Alternative: Rekursive Berechnung

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

Beispiel: Fakultät einer Zahl

Alternative: Rekursive Berechnung

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

Basiert auf folgender Beobachtung:

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$$
$$= n \cdot (n-1)!$$

Schleifen

Genereller Aufbau einer Schleife

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

Schleifen

Genereller Aufbau einer Schleife

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

Erläuterungen:

- Oft wird ein Zähler, der in jedem Schleifendurchlauf inkrementiert wird.

Schleifen

Genereller Aufbau einer Schleife

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

Erläuterungen:

- Oft wird ein Zähler, der in jedem Schleifendurchlauf inkrementiert wird.
- Die Schleife läuft solange, wie die Bedingung erfüllt ist.

Schleifen

Genereller Aufbau einer Schleife

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

Erläuterungen:

- Oft wird ein Zähler, der in jedem Schleifendurchlauf inkrementiert wird.
- Die Schleife läuft solange, wie die Bedingung erfüllt ist.
- Der Zähler ist meist eine `int`-Variable und startet bei 0.

Schleifen

Genereller Aufbau einer Schleife

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

Erläuterungen:

- Oft wird ein Zähler, der in jedem Schleifendurchlauf inkrementiert wird.
- Die Schleife 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.

Schleifen

Beispiel: Zahlen auflisten

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

Schleifen

Beispiel: Zahlen auflisten

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

Erläuterungen:

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

Schleifen

Beispiel: Zahlen rückwärts auflisten

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

Schleifen

Beispiel: Zahlen rückwärts auflisten

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

Erläuterungen:

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

Schleifen

Beispiel: Gerade Zahlen auflisten

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

Schleifen

Beispiel: Gerade Zahlen auflisten

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

Erläuterungen:

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

Schleifen

Beispiel: Vielfache auflisten

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

Schleifen

Beispiel: Vielfache auflisten

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

Erläuterungen:

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

Schleifen

Beispiel: Vielfache auflisten

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

Schleifen

Beispiel: Vielfache auflisten

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

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.

Schleifen

Beispiel: Summe berechnen

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

Schleifen

Beispiel: Summe berechnen

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

Erläuterungen:

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

Schleifen

Beispiel: Summe berechnen (rekursiv)

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

Schleifen

Beispiel: Summe berechnen (rekursiv)

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

Erläuterungen:

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

Schleifen

Beispiel: Primzahltest

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

Schleifen

Beispiel: Primzahltest

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

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`.

Schleifen

Beispiel: While-Schleife

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

Schleifen

Beispiel: While-Schleife

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

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.

Aufgaben: Funktionsaufrufe 1

```
0 func Foo() {
1     x := 3
2     y := Bar(x) + 2*3
3     fmt.Println(y + x)
4 }
5
6 func Bar(x int) int {
7     x = x*3 + 4
8     return x / 2
9 }
```

Aufgaben: Funktionsaufrufe 1

```
0 func Foo() {
1       x := 3
2       y := Bar(x) + 2*3
3       fmt.Println(y + x)
4 }
5
6 func Bar(x int) int {
7       x = x*3 + 4
8       return x / 2
9 }
```

Fragen:

- Was liefert ein Aufruf der Funktion `Bar()` für $x = 4, 5, 6$?
- Erklären Sie das Ergebnis für `Bar(5)`.
- Was für einen Effekt hat ein Aufruf von `Foo()`?

Aufgaben: Funktionsaufrufe 2

```
0 func Foo1() {
1       x := 5
2       Foo2(x)
3       fmt.Println(x)
4 }
```

Aufgaben: Funktionsaufrufe 2

```
0 func Foo1() {
1       x := 5
2       Foo2(x)
3       fmt.Println(x)
4 }
```

Fragen:

- Was für Effekte hat der Aufruf von `Foo1()`?

Aufgaben: Funktionsaufrufe 2

```
0 func Foo2(x int) {
1       x = x * 3
2       y := Foo3(x)
3       fmt.Println(y)
4 }
```

Fragen:

- Was für Effekte hat der Aufruf von `Foo1()`?
- Was für Effekte hat der Aufruf von `Foo2()`?

Aufgaben: Funktionsaufrufe 2

```
0 func Foo3(x int) int {
1       x = x + 4
2       fmt.Println(x)
3       return x / 2
4 }
```

Fragen:

- Was für Effekte hat der Aufruf von `Foo1()`?
- Was für Effekte hat der Aufruf von `Foo2()`?
- Was für Effekte hat der Aufruf von `Foo3()`?

Aufgaben: Zählfunktionen 1

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

Aufgaben: Zählfunktionen 1

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

Fragen:

- Was berechnet die Funktion `CountSomething()`?
- Was wäre ein sinnvoller Name?
- Was ist ein sinnvoller Wertebereich für `n`?

Aufgaben: Zählfunktionen 2

```
0 func CountElements(list []int) int {
1     result := 0
2     for i := 0; i < len(list); i++ {
3         x := list[i]
4         result += Value(x)
5     }
6     return result
7 }
```

Aufgaben: Zählfunktionen 2

```
0 func CountElements(list []int) int {
1     result := 0
2     for i := 0; i < len(list); i++ {
3         x := list[i]
4         result += Value(x)
5     }
6     return result
7 }
```

Fragen:

- Was berechnet die Funktion `CountElements()`?

Aufgaben: Zählfunktionen 2

```
0 func Value(x int) int {
1       x = x * x
2       return x
3 }
```

Aufgaben: Zählfunktionen 2

```
0 func Value(x int) int {
1       x = x * x
2       return x
3 }
```

Fragen:

- Was berechnet die Funktion `CountElements()`?
- Was berechnet die Funktion `Value()`?