

Programmier-Einführung mit Go

Reiner Hüchting

28. November 2024

Überblick

Grundlagen

Listen

Eigene Datentypen

Rekursion

Grundlagen – Überblick

Grundlagen

Hello World

Ratespiel

Kontrollfluss

Beispiel: Suche in einer Liste

Beispiel: Fakultät

Schleifen

Variablen

Listen

Eigene Datentypen

Rekursion

Hello World

Hello World in Go

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

Hello World

Hello World in Go

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

Zeile 1: Paketdefinition

- ▶ Pakete strukturieren den Code.
- ▶ main : Spezielles Paket für ausführbare Programme.

Hello World

Hello World in Go

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

Zeile 3: Import-Statement

- ▶ Macht Code aus anderen Paketen verfügbar (hier: `fmt`).
- ▶ `fmt` : Standardpaket für Ein- und Ausgabe.

Hello World

Hello World in Go

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

ab Zeile 5: Funktion `main`

- ▶ Einstiegspunkt des Programms.
- ▶ Jedes ausführbare Go-Programm enthält genau eine `main`-Funktion.

Hello World

Begrüßungsfunktion

```
1 package greet
2
3 import "fmt"
4
5 func Greet(name string) string {
6     return fmt.Sprintf("Hello %s!", name)
7 }
```


Hello World

Begrüßungsfunktion

```
1 package greet
2
3 import "fmt"
4
5 func Greet(name string) string {
6     return fmt.Sprintf("Hello %s!", name)
7 }
```

Funktionen machen den Code modular

- ▶ ermöglichen Wiederverwendung
- ▶ ermöglichen gleiches Verhalten für unterschiedliche Werte
- ▶ strukturieren den Code, verbessern die Lesbarkeit

Hello World

Begrüßungsfunktion

```
1 package greet
2
3 import "fmt"
4
5 func Greet(name string) string {
6     return fmt.Sprintf("Hello %s!", name)
7 }
```

Verhalten für verschiedene *Argumente*

`greet("Alice")` → Hello Alice!

`greet("Bob")` → Hello Bob!

Ratespiel

Ziel: Ein einfaches Ratespiel

- ▶ Der Benutzer wird bis zu drei Mal aufgefordert, eine Zahl zu raten.
- ▶ Falls die Eingabe richtig ist, endet das Programm sofort mit einer Nachricht.
- ▶ Falls die Eingabe drei Mal falsch ist, wird ebenfalls eine Nachricht ausgegeben und das Programm beendet.

Ratespiel

Einlesen einer Zahl

```
1 func ReadNumber() int {  
2     var n int  
3     fmt.Print("Rate eine Zahl: ")  
4     fmt.Scan(&n)  
5     return n  
6 }
```

Ratespiel

Einlesen einer Zahl

```
1 func ReadNumber() int {  
2     var n int  
3     fmt.Print("Rate eine Zahl: ")  
4     fmt.Scan(&n)  
5     return n  
6 }
```

Funktion ReadNumber

- ▶ erwartet keine Argumente
- ▶ gibt eine Zahl zurück

Ratespiel

Einlesen einer Zahl

```
1 func ReadNumber() int {  
2     var n int  
3     fmt.Print("Rate eine Zahl: ")  
4     fmt.Scan(&n)  
5     return n  
6 }
```

Zeile 2: Deklaration einer Variablen für die Zahl

- ▶ " var n int erzeugt eine Variable " n " vom Typ " int ".
- ▶ Soll die Eingabe des Benutzers speichern.

Ratespiel

Einlesen einer Zahl

```
1 func ReadNumber() int {  
2     var n int  
3     fmt.Print("Rate eine Zahl: ")  
4     fmt.Scan(&n)  
5     return n  
6 }
```

Zeile 4: Einlesen der Zahl

- ▶ Liest eine Eingabe von der Konsole.
- ▶ Speichert die Eingabe an der Adresse von `n`.

Ratespiel

Einlesen einer Zahl

```
1 func ReadNumber() int {  
2     var n int  
3     fmt.Print("Rate eine Zahl: ")  
4     fmt.Scan(&n)  
5     return n  
6 }
```

Zeile 5: Rückgabe der Zahl

- ▶ Beendet die Funktion mit `n` als Ergebnis.
- ▶ Das Ergebnis kann in anderen Funktionen verwendet werden.

Ratespiel

Verwendung von ReadInput

```
1 func GuessingGame() {  
2     for n := 0; n < 3; n++ {  
3         guess := ReadNumber()  
4         if NumberGood(guess) {  
5             fmt.Println("Richtig geraten! :-)")  
6             return  
7         }  
8     }  
9     fmt.Println("Zu viele falsche Zahlen! :-(")  
10 }
```

Ratespiel

Verwendung von ReadInput

```
1 func GuessingGame() {  
2     for n := 0; n < 3; n++ {  
3         guess := ReadNumber()  
4         if NumberGood(guess) {  
5             fmt.Println("Richtig geraten! :-)")  
6             return  
7         }  
8     }  
9     fmt.Println("Zu viele falsche Zahlen! :-(")  
10 }
```

Zeile 3: Aufruf von ReadNumber

- ▶ Führt die komplette Funktion aus.
- ▶ Speichert das Ergebnis in der Variable `guess`.

Ratespiel

Verwendung von ReadInput

```
1 func GuessingGame() {  
2     for n := 0; n < 3; n++ {  
3         guess := ReadNumber()  
4         if NumberGood(guess) {  
5             fmt.Println("Richtig geraten! :-)")  
6             return  
7         }  
8     }  
9     fmt.Println("Zu viele falsche Zahlen! :-(")  
10 }
```

Zeile 4: If-Anweisung

- ▶ Prüft die Eingabe mittels einer weiteren Funktion.
- ▶ Gibt eine Nachricht aus und beendet das Programm, falls die Eingabe korrekt ist.

Ratespiel

Verwendung von ReadInput

```
1 func GuessingGame() {  
2     for n := 0; n < 3; n++ {  
3         guess := ReadNumber()  
4         if NumberGood(guess) {  
5             fmt.Println("Richtig geraten! :-)")  
6             return  
7         }  
8     }  
9     fmt.Println("Zu viele falsche Zahlen! :-(")  
10 }
```

Zeile 2: For-Schleife

- ▶ Führt die Abfrage bis zu drei Mal aus.
- ▶ Verwendet einen Zähler, der mit 0 startet und bei jedem Durchlauf erhöht wird.

Ratespiel

Verwendung von ReadInput

```
1 func GuessingGame() {  
2     for n := 0; n < 3; n++ {  
3         guess := ReadNumber()  
4         if NumberGood(guess) {  
5             fmt.Println("Richtig geraten! :-)")  
6             return  
7         }  
8     }  
9     fmt.Println("Zu viele falsche Zahlen! :-(")  
10 }
```

Zeile 9: Ausgabe am Ende der Schleife

- Bis hier kommt das Programm, wenn der Benutzer drei Mal falsch geraten hat.

Ratespiel

Prüfung der Zahl

```
1 func main() {  
2     GuessingGame()  
3 }
```

main -Funktion des Ratespiels

- ▶ Einstiegspunkt: Muss vorhanden sein.
- ▶ Ruft die Funktion `GuessingGame` auf.

Ratespiel

Zusammenfassung

- ▶ Nutze `fmt.Scan` , um eine Benutzereingabe einzulesen,
- ▶ eine `If`-Anweisung, um auf die Eingabe zu reagieren,
- ▶ eine `For`-Schleife, um eine Aktion mehrfach auszuführen,
- ▶ Funktionen, um den Code zu strukturieren.

Ratespiel

Zusammenfassung

- ▶ Nutze `fmt.Scan`, um eine Benutzereingabe einzulesen,
- ▶ eine `If`-Anweisung, um auf die Eingabe zu reagieren,
- ▶ eine `For`-Schleife, um eine Aktion mehrfach auszuführen,
- ▶ Funktionen, um den Code zu strukturieren.

Die Schleife ist für sich alleine verständlich

```
1   for n := 0; n < 3; n++ {  
2       guess := ReadNumber()  
3       if NumberGood(guess) {  
4           fmt.Println("Richtig geraten! :-)")  
5           return  
6       }  
7   }
```


Kontrollfluss

Kontrollfluss

- ▶ Funktionen, Schleifen und If-Anweisungen
- ▶ Steuern den Ablauf eines Programms
- ▶ Erlauben automatisierte Berechnungen für unterschiedliche Eingaben

Kontrollfluss

Kontrollfluss

- ▶ Funktionen, Schleifen und If-Anweisungen
- ▶ Steuern den Ablauf eines Programms
- ▶ Erlauben automatisierte Berechnungen für unterschiedliche Eingaben

Beispiele

- ▶ Suche nach einem Element in einer Liste
- ▶ Berechnung der Fakultät einer Zahl
- ▶ Allgemeiner: Berechnung des aktuellen System-Zustands

Beispiel: Suche in einer Liste

Ziel: Finde die Position eines Elements in einer Liste.

Was wir nicht wollen...

```
1 func FindStepByStep(list []int, el int) int {  
2     if el == list[0] {  
3         return 0  
4     }  
5     if el == list[1] {  
6         return 1  
7     }  
8     if el == list[2] {  
9         return 2  
10    }  
11    // ...  
12    return -1  
13 }
```

Beispiel: Suche in einer Liste

Ziel: Finde die Position eines Elements in einer Liste.

Besser: Eine Schleife

```
1 func FindElementLoop1(list []int, e int) int {  
2     for i := 0; i < len(list); i++ {  
3         if list[i] == e {  
4             return i  
5         }  
6     }  
7     return -1  
8 }
```

Beispiel: Suche in einer Liste

Ziel: Finde die Position eines Elements in einer Liste.

Oder so:

```
1 func FindElementLoop2(list []int, e int) int {  
2     for i, el := range list {  
3         if el == e {  
4             return i  
5         }  
6     }  
7     return -1  
8 }
```

Beispiel: Suche in einer Liste

Ziel: Finde die Position eines Elements in einer Liste.

Oder so:

```
1 func FindElementLoop2(list []int, e int) int {  
2     for i, el := range list {  
3         if el == e {  
4             return i  
5         }  
6     }  
7     return -1  
8 }
```

Vorteil

- ▶ Die Schleife kann beliebig lange laufen.
- ▶ Die einzelnen Vergleiche wären in der Anzahl festgelegt.

Beispiel: Suche in einer Liste

Ziel: Finde die Position eines Elements in einer Liste.

Vorteil

- ▶ Die Schleife kann beliebig lange laufen.
- ▶ Die einzelnen Vergleiche wären in der Anzahl festgelegt.

Beispiel: Suche in einer Liste

Ziel: Finde die Position eines Elements in einer Liste.

Vorteil

- ▶ Die Schleife kann beliebig lange laufen.
- ▶ Die einzelnen Vergleiche wären in der Anzahl festgelegt.

Analogie

- ▶ Einzelne Vergleiche: „Dabei bleiben“
- ▶ Schleife ist *automatisiert*

Beispiel: Fakultät

Ziel: Berechne $5!$

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

Beispiel: Fakultät

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

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

Schritt-Für-Schritt-Berechnung

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

Beispiel: Fakultät

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 · 1	2
3 · 2	6
4 · 6	24
5 · 24	120

Beispiel: Fakultät

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.

Beispiel: Fakultät

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

Beispiel: Fakultät

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 von 5! mit Schleife

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


Beispiel: Fakultät

Berechnung von $5!$ mit Schleife

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

Beispiel: Fakultät

Berechnung von $5!$ 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.

Beispiel: Fakultät

Berechnung von 5! mit Schleife

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

Verallgemeinerte Schleife

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

Beispiel: Fakultät

Verallgemeinerte Schleife

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

Beispiel: Fakultät

Verallgemeinerte Schleife

```
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.

Beispiel: Fakultät

Verallgemeinerte Schleife

```
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.

Beispiel: Fakultät

Verallgemeinerte Schleife

```
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

Beispiel: Fakultät

Funktion für die Fakultät

```
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 }
```


Beispiel: Fakultät

Funktion für die Fakultät

```
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.

Beispiel: Fakultät

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 }
```

Beispiel: Fakultät

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**.

Beispiel: Fakultät

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!**

Beispiel: Fakultät

Alternative: Rekursive Berechnung

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

Beispiel: Fakultät

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:

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

Schleifen

Genereller Aufbau einer Schleife

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

Schleifen

Genereller Aufbau einer Schleife

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

Erläuterungen:

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

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 Schleife läuft solange, wie die **Bedingung** erfüllt ist.

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 Schleife läuft solange, wie die **Bedingung** erfüllt ist.
- ▶ Der Zähler ist meist eine `int`-Variable und startet bei 0.

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 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

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

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.

Schleifen

Beispiel: Zahlen rückwärts auflisten

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

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.

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 }
```


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.

Schleifen

Beispiel: Vielfache

```
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 }
```

Schleifen

Beispiel: Vielfache

```
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.

Schleifen

Beispiel: Vielfache

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

Schleifen

Beispiel: Vielfache

```
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.

Schleifen

Beispiel: Summe

```
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 }
```

Schleifen

Beispiel: Summe

```
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.

Schleifen

Beispiel: Summe Rekursiv

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


Schleifen

Beispiel: Summe 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.

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 }
```

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.

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 }
```

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.

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

```
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 }
```

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.

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 }
```

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.

Variablen

Listen-Variablen

```
1 func ListVariables() {
2     var l []int // leere Liste
3     l = append(l, 10, 20, 30, 40, 50)
4
5     fmt.Println(l)           // komplett ausgeben
6     fmt.Println(l[1])
7     // Zweites Element ausgeben
8     fmt.Println(l[1:3]) // Teil-Liste ausgeben
9     l[1] = 42           // Wert ändern
10    fmt.Println(l)
11 }
```

Variablen

Listen-Variablen

```
1 func ListVariables() {  
2     var l []int // leere Liste  
3     l = append(l, 10, 20, 30, 40, 50)  
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.

Listen – Überblick

Grundlagen

Listen

Arrays und Slices

Mehrdimensionale Arrays

Eigene Datentypen

Rekursion

Arrays und Slices

Arrays

- ▶ Basis-Datentyp für Listen von Elementen.
- ▶ Kommt in vielen Programmiersprachen vor.
- ▶ I.d.R. feste Größe/Länge und nur ein Element-Datentyp.
- ▶ Elemente liegen zusammenhängend im Speicher.

Arrays und Slices

Arrays

- ▶ Basis-Datentyp für Listen von Elementen.
- ▶ Kommt in vielen Programmiersprachen vor.
- ▶ I.d.R. feste Größe/Länge und nur ein Element-Datentyp.
- ▶ Elemente liegen zusammenhängend im Speicher.

Slices in Go

- ▶ Flexiblerer Listen-Datentyp.
- ▶ *Slices* sind *Views* auf *Arrays*.
 - ▶ Jede Slice hat ein zugrunde liegendes Array.
 - ▶ Mehrere Slices können auf das gleiche Array zeigen.

Arrays und Slices

Definition eines Arrays

```
1 func Example_arrayWithZeros() {  
2     var a [5]int  
3  
4     for i := 0; i < len(a); i++ {  
5         a[i] = i  
6     }  
7  
8     fmt.Println(a)  
9  
10    // Output:  
11    // [0 1 2 3 4]  
12 }
```

Arrays und Slices

Initialisierung eines Arrays

```
1 func Example_arrayWithValues() {  
2     b := [5]int{1, 2, 3, 4, 5}  
3  
4     fmt.Println(b)  
5  
6     // Output:  
7     // [1 2 3 4 5]  
8 }
```

Arrays und Slices

Leere Slice

```
1 func Example_emptySlice() {  
2     var a []int  
3  
4     fmt.Println(len(a))  
5     fmt.Println(a)  
6  
7     // Output:  
8     // 0  
9     // []  
10 }
```


Arrays und Slices

Slice mit Werten

```
1 func Example_sliceWithValues() {  
2     b := []int{1, 2, 3, 4, 5}  
3  
4     fmt.Println(len(b))  
5     fmt.Println(b)  
6  
7     // Output:  
8     // 5  
9     // [1 2 3 4 5]  
10 }
```

Arrays und Slices

Teil-Auschnitt einer Slice

```
1 func Example_subSlice() {  
2     a := []int{1, 2, 3, 4, 5}  
3     b := a[1:3]  
4  
5     fmt.Println(a)  
6     fmt.Println(b)  
7  
8     // Output:  
9     // [1 2 3 4 5]  
10    // [2 3]  
11 }
```

Arrays und Slices

Verändern einer Slice

```
1 func Example_modifySubSlice() {
2     a := []int{1, 2, 3, 4, 5}
3     b := a[1:3]
4
5     b[0] = 99
6
7     fmt.Println(a)
8     fmt.Println(b)
9
10    // Output:
11    // [1 99 3 4 5]
12    // [99 3]
13 }
```

Arrays und Slices

Append-Funktion

```
1 func Example_append() {  
2     a := []int{}  
3  
4     a = append(a, 1)  
5     a = append(a, 2)  
6     a = append(a, 3)  
7  
8     fmt.Println(a)  
9  
10    // Output:  
11    // [1 2 3]  
12 }
```

Arrays und Slices

Make-Funktion

```
1 func Example_make() {  
2     a := make([]int, 5)  
3  
4     fmt.Println(a)  
5  
6     // Output:  
7     // [0 0 0 0 0]  
8 }
```

Mehrdimensionale Arrays

Mehrdimensionale Arrays

- ▶ Listen können auch mehrere Dimensionen haben.
- ▶ Ansatz: Listen von Listen.

Mehrdimensionale Arrays

2x2-Matrix

```
1 func Example_matrix() {
2     a := [2][2]int{
3         {1, 2},
4         {3, 4},
5     }
6
7     fmt.Println(a[0])
8     fmt.Println(a[1])
9     fmt.Println(a[0][0])
10    fmt.Println(a[1][1])
11
12    // Output:
13    // [1 2]
14    // [3 4]
15    // 1
16    // 4
17 }
```

Mehrdimensionale Arrays

Schleife über Matrix

```
1 func Example_loopMatrix() {
2     a := [2][2]int{
3         {1, 2},
4         {3, 4},
5     }
6
7     for i := 0; i < len(a); i++ {
8         for j := 0; j < len(a[i]); j++ {
9             fmt.Print(a[i][j])
10        }
11        fmt.Println()
12    }
13
14    // Output:
15    // 12
16    // 34
17 }
```


Mehrdimensionale Arrays

Schleife über Spalte

```
1 func Example_loopMatrixColumn() {
2     a := [2][2]int{
3         {1, 2},
4         {3, 4},
5     }
6
7     for i := 0; i < len(a); i++ {
8         fmt.Print(a[i][1])
9     }
10
11     // Output:
12     // 24
13 }
```

Eigene Datentypen – Überblick

Grundlagen

Listen

Eigene Datentypen

- Definition eigener Datentypen

- Strukturierte Datentypen

Rekursion

Definition eigener Datentypen

Schlüsselwort type

- ▶ Definition neuer Namen für Datentypen.
- ▶ Bessere Lesbarkeit und Verständlichkeit.
- ▶ Modellierung von Domänen-spezifischen Typen.

Definition eigener Datentypen

Schlüsselwort type

- ▶ Definition neuer Namen für Datentypen.
- ▶ Bessere Lesbarkeit und Verständlichkeit.
- ▶ Modellierung von Domänen-spezifischen Typen.

Beispiel: Längeneinheiten

- ▶ Definiere Datentyp `Length` für Längenangaben.
- ▶ Ist i.W. ein `int`.
- ▶ Verhindert Verwechslung mit anderen `int`-Werten.

Definition eigener Datentypen

Längen-Datentyp

```
1 func ExampleLength() {  
2     var a Length = 10  
3  
4     fmt.Println(a)  
5  
6     // Output:  
7     // 10  
8 }
```

Definition eigener Datentypen

Methoden

- ▶ Spezielle Funktionen, die zu einem Typ gehören.
- ▶ Werden mit einem *Receiver* aufgerufen.
- ▶ Können Besonderheiten des Typs abbilden.

Definition eigener Datentypen

Exportmethoden

```
1 func (l Length) Centimeters() int {  
2     return int(l)  
3 }  
4  
5 func (l Length) Meters() int {  
6     return int(l / 100)  
7 }  
8  
9 func (l Length) Kilometers() int {  
10    return l.Meters() / 1000  
11 }
```

Definition eigener Datentypen

Exportmethoden

```
1 func ExampleLength_conversions() {  
2     var a Length = 500000  
3  
4     fmt.Println(a.Centimeters())  
5     fmt.Println(a.Meters())  
6     fmt.Println(a.Kilometers())  
7  
8     // Output:  
9     // 500000  
10    // 5000  
11    // 5  
12 }
```


Definition eigener Datentypen

Konstrukturen

- ▶ Funktionen, die ein Objekt eines Typs erstellen.
- ▶ Verbergen Initialisierungslogik.

Definition eigener Datentypen

Konstruktoren

```
1 func FromMeters(m int) Length {  
2     return Length(m * 100)  
3 }  
4  
5 func FromCentimeters(m int) Length {  
6     return Length(m)  
7 }  
8  
9 func FromKilometers(m int) Length {  
10    return Length(m * 1000 * 100)  
11 }
```

Definition eigener Datentypen

Konstruktoeren

```
1 func ExampleLength_from() {
2     a := FromMeters(5)
3     b := FromCentimeters(5)
4     c := FromKilometers(5)
5
6     fmt.Println(a)
7     fmt.Println(b)
8     fmt.Println(c)
9
10    // Output:
11    // 500
12    // 5
13    // 500000
14 }
```

Definition eigener Datentypen

Aufgabe: Entwerfen Sie einen Datentyp `Duration`

- ▶ Modelliert eine Zeitspanne.
- ▶ Speichert Sekunden.
- ▶ Bietet Export/Import als Stunden, Minuten und Sekunden.

Strukturierte Datentypen

Schlüsselwort `struct`

- ▶ Definition von zusammengehörigen Variablen.
- ▶ Modellierung von komplexen Datenstrukturen.

Strukturierte Datentypen

Schlüsselwort `struct`

- ▶ Definition von zusammengehörigen Variablen.
- ▶ Modellierung von komplexen Datenstrukturen.

Beispiel: GPS-Koordinaten

- ▶ Definiere `struct` für Längen- und Breitengrad.
- ▶ Beide sind `float64` -Werte.
- ▶ Methode, um Distanz zu einer anderen Koordinate zu berechnen.

Strukturierte Datentypen

Struct für Koordinaten

```
1 type Coordinate struct {  
2     Longitude float64  
3     Latitude  float64  
4 }
```

Strukturierte Datentypen

Verwendung

```
1 func ExampleCoordinate_usage() {  
2     a := Coordinate{0, 0}  
3     b := Coordinate{3, 4}  
4  
5     fmt.Println(a.Longitude)  
6     fmt.Println(b.Latitude)  
7  
8     a.Latitude = 1  
9     fmt.Println(a.Latitude)  
10  
11     // Output:  
12     // 0  
13     // 4  
14     // 1  
15 }
```


Strukturierte Datentypen

Distanz-Methode

```
1 func (c Coordinate) DistanceTo(other Coordinate) float64  
2     x := c.Longitude - other.Longitude  
3     y := c.Latitude - other.Latitude  
4     return math.Sqrt(x*x + y*y)  
5 }
```

Strukturierte Datentypen

Distanz-Methode

```
1 func ExampleCoordinate_DistanceTo() {  
2     a := Coordinate{0, 0}  
3     b := Coordinate{3, 4}  
4  
5     d := a.DistanceTo(b)  
6  
7     fmt.Println(d)  
8  
9     // Output:  
10    // 5  
11 }
```

Rekursion – Überblick

Grundlagen

Listen

Eigene Datentypen

Rekursion

- Einleitung

- Beispiele

- Türme von Hanoi

Einleitung

Was gibt diese Funktion für $n = 3$ aus?

```
1 func Countdown(n int) {  
2     if n <= 0 {  
3         return  
4     }  
5     fmt.Println(n)  
6     Countdown(n - 1)  
7 }
```

Einleitung

Beispiel: Addition als Gleichungen spezifiziert

$$x + 0 = x$$

$$x + s(y) = s(x + y)$$

Einleitung

Beispiel: Addition als Gleichungen spezifiziert

$$x + 0 = x$$

$$x + s(y) = s(x + y)$$

Anwendung der Gleichungen:

Einleitung

Beispiel: Addition als Gleichungen spezifiziert

$$x + 0 = x$$

$$x + s(y) = s(x + y)$$

Anwendung der Gleichungen:

$$s(s(0)) + s(s(s(0)))$$

Einleitung

Beispiel: Addition als Gleichungen spezifiziert

$$x + 0 = x$$

$$x + s(y) = s(x + y)$$

Anwendung der Gleichungen:

$$s(s(0)) + s(s(s(0)))$$

Einleitung

Beispiel: Addition als Gleichungen spezifiziert

$$x + 0 = x$$

$$x + s(y) = s(x + y)$$

Anwendung der Gleichungen:

$$s\left(\begin{array}{c} s(s(0)) + s(s(s(0))) \\ s(s(0)) + s(s(0)) \end{array} \right)$$

Einleitung

Beispiel: Addition als Gleichungen spezifiziert

$$x + 0 = x$$

$$x + s(y) = s(x + y)$$

Anwendung der Gleichungen:

$$s\left(\begin{array}{c} s(s(0)) + s(s(s(0))) \\ s(s(0)) + s(s(0)) \end{array} \right)$$

Einleitung

Beispiel: Addition als Gleichungen spezifiziert

$$x + 0 = x$$

$$x + s(y) = s(x + y)$$

Anwendung der Gleichungen:

$$s\left(\begin{array}{c} s(s(0)) + s(s(s(0))) \\ s(s(0)) + s(s(0)) \end{array} \right)$$

Einleitung

Beispiel: Addition als Gleichungen spezifiziert

$$x + 0 = x$$

$$x + s(y) = s(x + y)$$

Anwendung der Gleichungen:

$$\begin{array}{l} s(s(0)) + s(s(s(0))) \\ s(s(s(0)) + s(s(0))) \\ s(s(s(s(0)) + s(0))) \end{array}$$

Einleitung

Beispiel: Addition als Gleichungen spezifiziert

$$x + 0 = x$$

$$x + s(y) = s(x + y)$$

Anwendung der Gleichungen:

$$\begin{array}{l} s(s(s(0)) + s(s(s(0)))) \\ s(s(s(0)) + s(s(0))) \\ s(s(s(0)) + s(0)) \end{array}$$

Einleitung

Beispiel: Addition als Gleichungen spezifiziert

$$x + 0 = x$$

$$x + s(y) = s(x + y)$$

Anwendung der Gleichungen:

$$\begin{array}{l} s(s(s(0)) + s(s(s(0)))) \\ s(s(s(0)) + s(s(0))) \\ s(s(s(s(0)) + s(0))) \end{array}$$

Einleitung

Beispiel: Addition als Gleichungen spezifiziert

$$x + 0 = x$$

$$x + s(y) = s(x + y)$$

Anwendung der Gleichungen:

$$\begin{array}{lcl} & s(s(0)) + s(s(s(0))) & \\ s(& s(s(0)) + s(s(0)) &) \\ s(s(& s(s(0)) + s(0) &)) \\ s(s(s(& s(s(0)) + 0 &))) \end{array}$$

Einleitung

Beispiel: Addition als Gleichungen spezifiziert

$$x + 0 = x$$

$$x + s(y) = s(x + y)$$

Anwendung der Gleichungen:

$$\begin{array}{lcl} & s(s(0)) + s(s(s(0))) & \\ s(& s(s(0)) + s(s(0)) &) \\ s(s(& s(s(0)) + s(0) &)) \\ s(s(s(& s(s(0)) + 0 &))) \end{array}$$

Einleitung

Beispiel: Addition als Gleichungen spezifiziert

$$x + 0 = x$$

$$x + s(y) = s(x + y)$$

Anwendung der Gleichungen:

$$\begin{array}{lcl} & s(s(0)) + s(s(s(0))) & \\ s(& s(s(0)) + s(s(0)) &) \\ s(s(& s(s(0)) + s(0) &)) \\ s(s(s(& s(s(0)) + 0 &))) \end{array}$$

Einleitung

Beispiel: Addition als Gleichungen spezifiziert

$$x + 0 = x$$

$$x + s(y) = s(x + y)$$

Anwendung der Gleichungen:

$$\begin{array}{l} s(s(s(0)) + s(s(s(0)))) \\ s(s(s(s(0)) + s(s(0)))) \\ s(s(s(s(s(0)) + s(0)))) \\ s(s(s(s(s(s(0)) + 0)))) \\ s(s(s(s(s(s(s(0))))) \end{array}$$

Einleitung

Beispiel: Addition als Gleichungen spezifiziert

$$x + 0 = x$$

$$x + s(y) = s(x + y)$$

Anwendung der Gleichungen:

$$\begin{array}{lcl} & s(s(0)) + s(s(s(0))) & \\ s(& s(s(0)) + s(s(0)) &) \\ s(s(& s(s(0)) + s(0) &)) \\ s(s(s(& s(s(0)) + 0 &))) \\ s(s(s(& s(s(0)) &))) \end{array}$$

Einleitung

Rekursive Addition als Go -Programm:

```
1 func Add1(x, y int) int {  
2  
3     // Gleichungen für die Addition:  
4     //  $x + 0 = x$   
5     //  $x + (y+1) = (x+y) + 1$   
6  
7     if y == 0 {  
8         return x  
9     }  
10    return Add1(x, y-1) + 1  
11 }
```

Einleitung

Alternative Version (Tail-Recursion):

```
1 func Add2(x, y int) int {  
2  
3     // Gleichungen für die Addition:  
4     //  $x + 0 = x$   
5     //  $x + y = (x+1) + (y-1)$   
6  
7     if y == 0 {  
8         return x  
9     }  
10    return Add2(x+1, y-1)  
11 }
```

Einleitung

Wozu Rekursion?

- ▶ Manches lässt sich kürzer und eleganter schreiben.

Einleitung

Wozu Rekursion?

- ▶ Manches lässt sich kürzer und eleganter schreiben.
- ▶ Beispiel Fakultät:

$$fac(n) = \prod_{i=1}^n i \quad \text{oder} \quad \begin{aligned} fac(0) &= 1 \\ fac(n) &= n \cdot fac(n-1) \end{aligned}$$

Einleitung

Wozu Rekursion?

- ▶ Manches lässt sich kürzer und eleganter schreiben.
- ▶ Beispiel Fakultät:

$$fac(n) = \prod_{i=1}^n i \quad \text{oder} \quad \begin{aligned} fac(0) &= 1 \\ fac(n) &= n \cdot fac(n-1) \end{aligned}$$

- ▶ Als iteratives Go -Programm:

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


Einleitung

Wozu Rekursion?

- ▶ Manches lässt sich kürzer und eleganter schreiben.
- ▶ Beispiel Fakultät:

$$fac(n) = \prod_{i=1}^n i \quad \text{oder} \quad \begin{aligned} fac(0) &= 1 \\ fac(n) &= n \cdot fac(n-1) \end{aligned}$$

- ▶ Als rekursives Go -Programm:

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

Einleitung

Schema für rekursive Definitionen

- ▶ Ein oder mehrere Basisfälle (Rekursionsanfang, Anker).
- ▶ Ein oder mehrere rekursive Aufrufe (Rekursionsschritt).

Einleitung

Schema für rekursive Definitionen

- ▶ Ein oder mehrere Basisfälle (Rekursionsanfang, Anker).
- ▶ Ein oder mehrere rekursive Aufrufe (Rekursionsschritt).

Vergleich mit while -Schleifen

- ▶ Abbruchbedingung entspricht Rekursionsanfang.
- ▶ Schleifenrumpf entspricht Rekursionsschritt.

Beispiele

Beispiel: Fibonacci-Folge

$$\textit{fib}(1) = \textit{fib}(2) = 1$$

$$\textit{fib}(n) = \textit{fib}(n-1) + \textit{fib}(n-2)$$

Beispiele

Beispiel: Fibonacci-Folge

$$\text{fib}(1) = \text{fib}(2) = 1$$

$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$$

die ersten 10 Fibonacci-Zahlen:

$n :$	1	2	3	4	5	6	7	8	9	10
$\text{fib}(n) :$	1	1	2	3	5	8	13	21	34	55

Beispiele

Beispiel: Hailstone-Folge

- ▶ Beginne mit einer *natürlichen Zahl* n .
- ▶ Ist n gerade, so nimm als nächstes $n/2$.
- ▶ Ist n ungerade, so nimm als nächstes $3n + 1$.
- ▶ Wiederhole, bis der Zyklus 4, 2, 1 erreicht ist.

Beispiele:

Beispiele

Beispiel: Hailstone-Folge

- ▶ Beginne mit einer *natürlichen Zahl* n .
- ▶ Ist n gerade, so nimm als nächstes $n/2$.
- ▶ Ist n ungerade, so nimm als nächstes $3n + 1$.
- ▶ Wiederhole, bis der Zyklus 4, 2, 1 erreicht ist.

Beispiele:

$n = 1$: 1, 4, 2, 1

Beispiele

Beispiel: Hailstone-Folge

- ▶ Beginne mit einer *natürlichen Zahl* n .
- ▶ Ist n gerade, so nimm als nächstes $n/2$.
- ▶ Ist n ungerade, so nimm als nächstes $3n + 1$.
- ▶ Wiederhole, bis der Zyklus 4, 2, 1 erreicht ist.

Beispiele:

$n = 1$: 1, 4, 2, 1

$n = 2$: 2, 1, 4, 2, 1

Beispiele

Beispiel: Hailstone-Folge

- ▶ Beginne mit einer *natürlichen Zahl* n .
- ▶ Ist n gerade, so nimm als nächstes $n/2$.
- ▶ Ist n ungerade, so nimm als nächstes $3n + 1$.
- ▶ Wiederhole, bis der Zyklus 4, 2, 1 erreicht ist.

Beispiele:

$n = 1$: 1, 4, 2, 1

$n = 2$: 2, 1, 4, 2, 1

$n = 3$: 3, 10, 5, 16, 8, 4, 2, 1

Beispiele

Beispiel: Hailstone-Folge

- ▶ Beginne mit einer *natürlichen Zahl* n .
- ▶ Ist n gerade, so nimm als nächstes $n/2$.
- ▶ Ist n ungerade, so nimm als nächstes $3n + 1$.
- ▶ Wiederhole, bis der Zyklus 4, 2, 1 erreicht ist.

Beispiele:

$n = 1$: 1, 4, 2, 1

$n = 2$: 2, 1, 4, 2, 1

$n = 3$: 3, 10, 5, 16, 8, 4, 2, 1

$n = 4$: 4, 2, 1

Beispiele

Beispiel: Hailstone-Folge

- ▶ Beginne mit einer *natürlichen Zahl* n .
- ▶ Ist n gerade, so nimm als nächstes $n/2$.
- ▶ Ist n ungerade, so nimm als nächstes $3n + 1$.
- ▶ Wiederhole, bis der Zyklus 4, 2, 1 erreicht ist.

Beispiele:

$n = 1$: 1, 4, 2, 1

$n = 2$: 2, 1, 4, 2, 1

$n = 3$: 3, 10, 5, 16, 8, 4, 2, 1

$n = 4$: 4, 2, 1

$n = 5$: 5, 16, 8, 4, 2, 1

Beispiele

Beispiel: Hailstone-Folge

- ▶ Beginne mit einer *natürlichen Zahl* n .
- ▶ Ist n gerade, so nimm als nächstes $n/2$.
- ▶ Ist n ungerade, so nimm als nächstes $3n + 1$.
- ▶ Wiederhole, bis der Zyklus 4, 2, 1 erreicht ist.

Beispiele:

$n = 1$: 1, 4, 2, 1

$n = 2$: 2, 1, 4, 2, 1

$n = 3$: 3, 10, 5, 16, 8, 4, 2, 1

$n = 4$: 4, 2, 1

$n = 5$: 5, 16, 8, 4, 2, 1

$n = 6$: 6, 3, 10, 5, 16, 8, 4, 2, 1

Beispiele

Beispiel: Hailstone-Folge

- ▶ Beginne mit einer *natürlichen Zahl* n .
- ▶ Ist n gerade, so nimm als nächstes $n/2$.
- ▶ Ist n ungerade, so nimm als nächstes $3n + 1$.
- ▶ Wiederhole, bis der Zyklus 4, 2, 1 erreicht ist.

Beispiele:

$n = 1$: 1, 4, 2, 1

$n = 2$: 2, 1, 4, 2, 1

$n = 3$: 3, 10, 5, 16, 8, 4, 2, 1

$n = 4$: 4, 2, 1

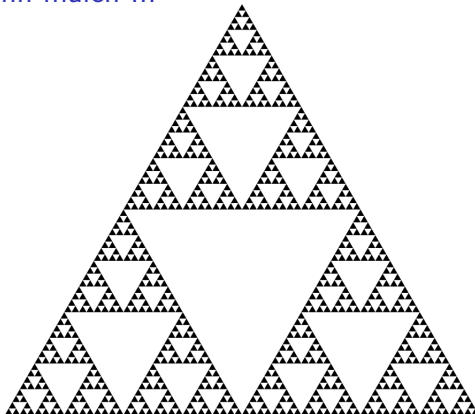
$n = 5$: 5, 16, 8, 4, 2, 1

$n = 6$: 6, 3, 10, 5, 16, 8, 4, 2, 1

$n = 7$: 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1

Beispiele

Rekursion kann malen ...



Dieses Bild wird **Sierpinski-Dreieck** genannt.

Beispiele

Beispiel: Ackermann-Funktion

$$A(m, n) = \begin{cases} n + 1 & \text{falls } m = 0 \\ A(m - 1, 1) & \text{falls } m > 0 \text{ und } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{falls } m > 0 \text{ und } n > 0 \end{cases}$$

Beispiele

Beispiel: Ackermann-Funktion

$$A(m, n) = \begin{cases} n + 1 & \text{falls } m = 0 \\ A(m - 1, 1) & \text{falls } m > 0 \text{ und } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{falls } m > 0 \text{ und } n > 0 \end{cases}$$

Hintergrund

- ▶ Die Werte dieser Funktion wachsen extrem schnell!
- ▶ Die Funktion wurde erdacht, um zu beweisen, dass Schleifen ohne Laufzeitschranke beim Programmieren notwendig sind.
- ▶ Der Beweis hat die Wachstumsgeschwindigkeit der Ackermann-Funktion verwendet.

Türme von Hanoi

Aufgabe: Bewege einen Turm aus Spielsteinen von A nach C

Gegeben:

- ▶ Spielsteine unterschiedlicher Größe.
- ▶ Drei Stellen A, B und C, an denen Spielsteine liegen können.

Spielregeln:

1. Die Steine werden einzeln bewegt.
2. Es wird niemals ein größerer Stein auf einen kleineren gelegt.

Türme von Hanoi

Aufgabe: Bewege einen Turm aus Spielsteinen von A nach C

Gegeben:

- ▶ Spielsteine unterschiedlicher Größe.
- ▶ Drei Stellen **A**, **B** und **C**, an denen Spielsteine liegen können.

Spielregeln:

1. Die Steine werden einzeln bewegt.
2. Es wird niemals ein größerer Stein auf einen kleineren gelegt.

Beispiel mit 3 Steinen:



Türme von Hanoi

Aufgabe: Bewege einen Turm aus Spielsteinen von A nach C

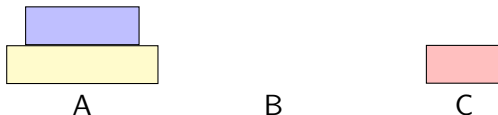
Gegeben:

- ▶ Spielsteine unterschiedlicher Größe.
- ▶ Drei Stellen **A**, **B** und **C**, an denen Spielsteine liegen können.

Spielregeln:

1. Die Steine werden einzeln bewegt.
2. Es wird niemals ein größerer Stein auf einen kleineren gelegt.

Beispiel mit 3 Steinen:



Türme von Hanoi

Aufgabe: Bewege einen Turm aus Spielsteinen von A nach C

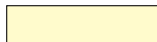
Gegeben:

- ▶ Spielsteine unterschiedlicher Größe.
- ▶ Drei Stellen A, B und C, an denen Spielsteine liegen können.

Spielregeln:

1. Die Steine werden einzeln bewegt.
2. Es wird niemals ein größerer Stein auf einen kleineren gelegt.

Beispiel mit 3 Steinen:



A



B



C

Türme von Hanoi

Aufgabe: Bewege einen Turm aus Spielsteinen von A nach C

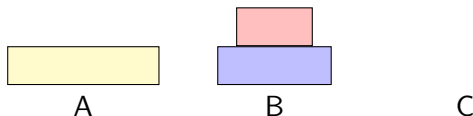
Gegeben:

- ▶ Spielsteine unterschiedlicher Größe.
- ▶ Drei Stellen **A**, **B** und **C**, an denen Spielsteine liegen können.

Spielregeln:

1. Die Steine werden einzeln bewegt.
2. Es wird niemals ein größerer Stein auf einen kleineren gelegt.

Beispiel mit 3 Steinen:



Türme von Hanoi

Aufgabe: Bewege einen Turm aus Spielsteinen von A nach C

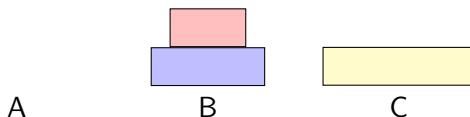
Gegeben:

- ▶ Spielsteine unterschiedlicher Größe.
- ▶ Drei Stellen **A**, **B** und **C**, an denen Spielsteine liegen können.

Spielregeln:

1. Die Steine werden einzeln bewegt.
2. Es wird niemals ein größerer Stein auf einen kleineren gelegt.

Beispiel mit 3 Steinen:



Türme von Hanoi

Aufgabe: Bewege einen Turm aus Spielsteinen von A nach C

Gegeben:

- ▶ Spielsteine unterschiedlicher Größe.
- ▶ Drei Stellen **A**, **B** und **C**, an denen Spielsteine liegen können.

Spielregeln:

1. Die Steine werden einzeln bewegt.
2. Es wird niemals ein größerer Stein auf einen kleineren gelegt.

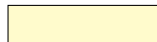
Beispiel mit 3 Steinen:



A



B



C

Türme von Hanoi

Aufgabe: Bewege einen Turm aus Spielsteinen von A nach C

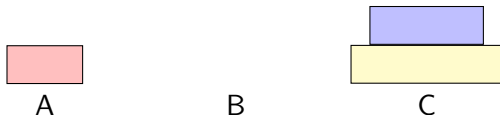
Gegeben:

- ▶ Spielsteine unterschiedlicher Größe.
- ▶ Drei Stellen **A**, **B** und **C**, an denen Spielsteine liegen können.

Spielregeln:

1. Die Steine werden einzeln bewegt.
2. Es wird niemals ein größerer Stein auf einen kleineren gelegt.

Beispiel mit 3 Steinen:



Türme von Hanoi

Aufgabe: Bewege einen Turm aus Spielsteinen von A nach C

Gegeben:

- ▶ Spielsteine unterschiedlicher Größe.
- ▶ Drei Stellen A, B und C, an denen Spielsteine liegen können.

Spielregeln:

1. Die Steine werden einzeln bewegt.
2. Es wird niemals ein größerer Stein auf einen kleineren gelegt.

Beispiel mit 3 Steinen:



Türme von Hanoi

Frage: Wie bewegt man einen Turm der Höhe h von A nach C?

Türme von Hanoi

Frage: Wie bewegt man einen Turm der Höhe h von A nach C?

Naive Antwort:

1. Bewege alle bis auf die letzte Platte von A nach B
2. Bewege die letzte Platte von A nach C
3. Bewege den Turm von B nach C

Türme von Hanoi

Frage: Wie bewegt man einen Turm der Höhe h von A nach C?

Naive Antwort:

1. Bewege alle bis auf die letzte Platte von A nach B
2. Bewege die letzte Platte von A nach C
3. Bewege den Turm von B nach C

Überraschung: So naiv ist das gar nicht!

- Wir konstruieren einen Algorithmus auf Basis dieser Vorgehensweise.

Türme von Hanoi

Wir definieren stückweise eine Funktion, die das Problem löst.

- Bewegen einer einzelnen Platte
von *Start* (s) über *Mitte* (m) nach *Ziel* (z):

```
1 func Move(s, z string) {  
2     fmt.Printf("Bewege Scheibe von %s nach %s.\n",  
3 }
```

Türme von Hanoi

Wir definieren stückweise eine Funktion, die das Problem löst.

- Bewegen eines Turms der Höhe 1:

```
1 func Hanoi1(s, m, z string) {  
2     Move(s, z)  
3 }
```

Türme von Hanoi

Konstruktion der Hanoi-Lösung (Fortsetzung)

- Bewegen eines Turms der Höhe 2:

```
1 func Hanoi2(s, m, z string) {  
2     Hanoi1(s, z, m)  
3     Move(s, z)  
4     Hanoi1(m, s, z)  
5 }
```

Türme von Hanoi

Konstruktion der Hanoi-Lösung (Fortsetzung)

- Bewegen eines Turms der Höhe 3:

```
1 func Hanoi3(s, m, z string) {  
2     Hanoi2(s, z, m)  
3     Move(s, z)  
4     Hanoi2(m, s, z)  
5 }
```


Türme von Hanoi

Konstruktion der Hanoi-Lösung (Fortsetzung)

- Bewegen eines Turms der Höhe 4:

```
1 func Hanoi4(s, m, z string) {  
2     Hanoi3(s, z, m)  
3     Move(s, z)  
4     Hanoi3(m, s, z)  
5 }
```

Laaaaaaaaa...

Türme von Hanoi

Konstruktion der Hanoi-Lösung (Fortsetzung)

- Bewegen eines Turms der Höhe 5:

```
1 func Hanoi5(s, m, z string) {  
2     Hanoi4(s, z, m)  
3     Move(s, z)  
4     Hanoi4(m, s, z)  
5 }
```

...aaaaaaang...

Türme von Hanoi

Konstruktion der Hanoi-Lösung (Fortsetzung)

- Bewegen eines Turms der Höhe 6:

```
1 func Hanoi6(s, m, z string) {  
2     Hanoi5(s, z, m)  
3     Move(s, z)  
4     Hanoi5(m, s, z)  
5 }
```

...weeeeilig

Türme von Hanoi

Beobachtungen:

- ▶ Die Funktionen `hanoi2` , `hanoi3` , `hanoi4` , ...sind alle gleich.
- ▶ Beim Aufruf wird nur die Zahl reduziert und dann wieder das Gleiche gemacht.
- ▶ Nur bei `hanoi1` wird kein `hanoi0` aufgerufen.

Türme von Hanoi

Beobachtungen:

- ▶ Die Funktionen `hanoi2` , `hanoi3` , `hanoi4` , ...sind alle gleich.
- ▶ Beim Aufruf wird nur die Zahl reduziert und dann wieder das Gleiche gemacht.
- ▶ Nur bei `hanoi1` wird kein `hanoi0` aufgerufen.

Schlussfolgerung: Wenn die Höhe als Argument übergeben wird, können wir alles in eine Funktion schreiben.

Türme von Hanoi

Rekursive Hanoi-Lösung

```
1 func Hanoi(h int, s, m, z string) {  
2     if h == 1 {  
3         Move(s, z)  
4     } else {  
5         Hanoi(h-1, s, z, m)  
6         Move(s, z)  
7         Hanoi(h-1, m, s, z)  
8     }  
9 }
```

