# Herbstcampus 2022

## Go für Ein- und Umsteiger

# Agenda

- Go Hype & Hate
- Go: Die Sprache
- Concurrent Go
- Enntwicklung einer komplexeren Beispielapplikation

# Go Hype & Hate

- Go ignoriert 20 Jahre der Forschung in Programmiersprachen
- Go hat ein primitives Typensystem
- jede zweite Zeile Go-Code ist eine If-Anweisung
- Go ist einfach gehalten, damit Juniorprogrammierer möglichst schnell produktiv werden können
- Go soll auf große Teams skalieren

# Go Hype & Hate

- Gos Dependency-Management **war** primitiv
- Go ist immer schnell
- Go ist super einfach zu lernen
- Go ist von alten Unix-Hasen programmiert

# Mission

*Go is an attempt to combine the ease of programming of an interpreted, dynamically typed language with the efficiency and safety of a statically typed, compiled language. It also aims to be modern, with support for networked and multicore computing. Finally, it is intended to be fast: it should take at most a few seconds to build a large executable on a single computer. **

# Über Go

- gestartet 2007 / 2008 von Robert Griesemer, Rob Pike and Ken Thompson
- ein Fokus: schnelles kompilieren
- C-Syntax mit Anleihen von Pascal/Modula
- keine Typ-Hierarchien
- lange keine Generics

# Über Go

- keine Exceptions
- implizite Interfaces
- keine Überladung von Funktions-/ Methodennamen
- keine Default-Parameter
- kein automatisches Casting

# Über Go

- kaum Helper-Funktionen für Arrays (Container-Typen)
- keine Option / Result / Maybe-Typ
- kein Pattern-Matching
- Garbage-Collected
- Kompiliert zu statischen Binaries (inkls. Runtime)
- kurze und verständliche Spezifikation

# Über Go

- Go-Routinen: sehr einfaches Concurrency-Konzept
- Channel: basierend auf *Don't communicate by sharing memory; share memory by communicating.*
- Go-Routinen & Channel: Communicating sequential processes
- Funktionen sind *first-level-citizens*

# Batteries included

- **`http/net`** bietet http server & client package
- **`encode/json`** bietet JSON (Un-)Marshaller
- **`testing`** bietet Test-Framework
- **`benchmark`** bietet Test-Framework
- **`flags`** Option-Parser

# Batteries included

- **`net/rpc`** simples rpc package
- **`compress`** bzip2, zlib, gzip, lzw, flate
- **`gofmt`** Code-formatierung
- **`godoc`** Dokumentation + Example code
- **`present`** diese Dokumentation mit *Play*

# Batteries included

- `crypto`
- `text/template` & `html/template`
- `...`
- Cross-Compilation quasi von Haus aus

# Go: Die Sprache

- Datentypen / Embedding (Composition)
- Funktionen / Methoden
- Kontrollstrukturen / Operatoren
- Error Handling
- Interfaces
- Pointer

# Struktur eines Programms

- besteht aus einem (main) oder mehreren **packages**
- mehrere **Dateien** pro **package** möglich
- ein package enthält: **Typdefinitionen**, **Funktionsdefinitionen**, **Konstanten-** und **Variablendefinitionen** und den **Import** von **Abhängigkeiten**
- packages bestimmen die **Sichtbarkeit** (*Groß-/Kleinschreibung*)

# Struktur eines Programms

- packages können **Initialisierungscode** enthalten
- die **main Funktion** im main package ist der **Einstiegspunkt** in das Programm
- aus dem main package können **keine Datenypen/Funktionen importiert** werden
- wird mit den transitiven Abhängigkeiten (**import**) zu **einem statischen Binary** gelinkt

# Struktur eines Programms

```
package main

import (
    "fmt"

    m "math"

    "golang.org/x/tools/present"
)

const Π = m.Pi

type myString string

var hello myString = "日本語"
var (
    code = new(present.Code)
)

// func init() {
//    hello = "Hello Herbstcampus 2022!"
// }

func main() {
    fmt.Println(hello)
}
```

RUN

# Übung 1: Go Code richtig formatieren

```
instructions/01_instructions.md
```

# Übung 2: Hello Herbstcampus

```
instructions/02_instructions.md
```

# Datentypen

## Built-in

- primitive types
- container types

## Benutzerdefiniert

- type alias
- struct
- interface

# Numerische Typen

```
uint8       // the set of all unsigned  8-bit integers (0 to 255)
uint16      // the set of all unsigned 16-bit integers (0 to 65535)
uint32      // the set of all unsigned 32-bit integers (0 to 4294967295)
uint64      // the set of all unsigned 64-bit integers (0 to 18446744073709551615)

int8        // the set of all signed  8-bit integers (-128 to 127)
int16       // the set of all signed 16-bit integers (-32768 to 32767)
int32       // the set of all signed 32-bit integers (-2147483648 to 2147483647)
int64       // the set of all signed 64-bit integers (-9223372036854775808 to 9223372036854775807)

float32     // the set of all IEEE-754 32-bit floating-point numbers
float64     // the set of all IEEE-754 64-bit floating-point numbers

complex64   // the set of all complex numbers with float32 real and imaginary parts
complex128  // the set of all complex numbers with float64 real and imaginary parts

byte        // alias for uint8
rune        // alias for int32
```

# Numerische Typen

## implementation dependent

```
uint     // either 32 or 64 bits
int      // same size as uint
uintptr  // an unsigned integer large enough to store the uninterpreted bits of a pointer value
```

## integer & floating-point literals

```
42
0600 // octal
0xBadFace
170141183460469231731687303715884105727

0.
72.40
072.40  // == 72.40
2.71828
1.e+0
6.67428e-11
1E6
.25
.12345E+5
```

# Booleans & Runes & Nil

## boolean types

```
bool       //  true or false
```

## rune literal

```
'a'
'ä'
'本'
'\t'            // horizontal tab
'\''            // single '
'\007'          // octal (0-255)
'\377'          // octal (0-255)
'\x0f'          // hex
'\u12e4'        // Unicode code point
'\U0010123f'    // Unicode code point
```

## The billion-dollar mistake

```
nil
```

# Strings

```
string    // var helloWorld string = "Hello World!"
```

## raw string literal

```
`abc`                // same as "abc"
`\n
\n`                  // same as "\\n\n\\n"
```

## interpreted string literals

```
"\n"
"\""                 // same as `"`
"Hello, world!\n"
"日本語"
"\u65e5本\U00008a9e"
"\xff\u00FF"
```

# Array

- Sequenz fester Länge eines Typ T: **`[len]T`**

```
[5]int
[2*n]float64
```

- können geschachtelt werden: **array of arrays**

```
[10][20]byte
```

- Initialisierung per `{value, value, ...}`

```
[2]string{"Hello", "World"}  // ["Hello", "World"]
[...]int{1,2,3, 4}              // [1 2 3 4]
[5]int{1,2,3}                // [1 2 3 0 0]
[4]int{1: 1, 3: 2}           // [0 1 0 2]
[2][3]int{{1,2,3}, {4,5,6}}  // [[1 2 3] [4 5 6]]
```

# Array

- die built-in Funktion **`len(array)`** gibt die Länge eines Array zurück
- Indiziert von **`0`** bis **`len(a)-1`**
- Zugriff über Indexnotation **`arr[index]`**

```
a := [6]int{}
a[5] = 10      // schreibender Zugriff
b := a[10]  // lesender Zugriff
```

- Arrays haben **call-by-value** Semantik

# Slice

- Sequenz variabler Länge eines Typ T: **`[]T`**

```
[]int
[]string
```

- besteht aus Länge, Kapazität und Pointer auf ein Array
- können geschachtelt werden: **slice of slices** [][]byte

# Slice

- Initialisierung per **`{value, value, ...}`**

```
[]int{1,2}
[][]float64{{1.2, 2.4}, {3.0, 4.1, 2.3}} // [[1.2 2.4] [3 4.1 2.3]]
```

- Initialisierung mit der built-in Funktion **`make`**

```
// make([]T, length, capacity)
make([]string, 5, 10) // Länge = 5, Kapazität = 10
```

# Slice

- **slice expressions**: machen ein slice aus einem slice/array/pointer auf ein array/string
- simple slice expression: **`s[low:high]`**

```
arr := [...]int{1, 2, 3, 4, 5} // [1 2 3 4 5]
// low ist inklusiv, high ist exklusiv
s := arr[1:4] // [2, 3, 4]
// low und high sind optional
arr[2:]  // == arr[2 : len(a)]
arr[:3]  // == arr[0 : 3]
arr[:]   // == arr[0 : len(a)]
```

# Slice

- full slice expression **`s[low:high:max]`**

```
arr := [...]int{1, 2, 3, 4, 5, 6, 7, 8}
// Kapazität ist max - low
s := arr[1:3:6] // [2, 3] mit Länge 2 und Kapazität 5
// low ist optional
arr[:3:5] // == arr[0:3:5]
```

- die built-in Funktion **`len(slice)`** gibt die Länge einer Slice zurück
- die built-in Funktion **`cap(slice)`** gibt die Kapazität einer Slice zurück

# Slice

- Indiziert von **`0`** bis **`len(s)-1`**
- Zugriff über Indexnotation **`s[index]`**

```
s := make([]int, 10, 15) //
s[5] = 10   // schreibender Zugriff
b := s[10]  // lesender Zugriff
```

# Slice

- die built-in Funktion **`copy(d dst, s src) int`** kopiert die Elemente von **`src`** nach **`dst`** und liefert die Anzahl der kopierten Elemente zurück. Es werden **`min(len(src), len(dst))`** Elemente kopiert. **`src`** und **`dst`** müssen Elemente vom selbem Typ haben.

```
s1 := [...]int{0, 1, 2, 3, 4, 5, 6, 7}
s2 := make([]int, 5)
copy(s2, s1[1:3])
// s2 == [1 2 0 0 0]
```

# Slice

- die built-in Funktion **`append(d dst, v ...T)`** hängt **`0`** bis **`n`** Elemente an ein Slice an. Wenn die Kapazität des Slice zu gering ist, alloziert **`append`** automatisch ein Array mit passender Größe.

```
s1 := []int{0, 1, 2, 3, 4, 5, 6, 7}
s2 := append(s1, 8, 9)  // s2 == [0 1 2 3 4 5 6 7 8 9]
s3 := append(s1, s2...) // s3 == [0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9]
```

- **Vorsicht:** slices haben zwar **`call-by-value`** Semantik, aber das unterliegende Array wird duch einen Pointer referenziert und wird nicht kopiert

# Slice

- Beispiel: shared array

```
arr := [5]int{1, 2, 3, 4, 5}
s1 := arr[:3]
fmt.Printf("initial len(s1) == 3, cap(s1) == len(arr):\narr == %v s1 == %v\n\n", arr, s1)

arr[2] = 10 // change the underlying array
fmt.Printf("after change to arr:\narr == %v s1 == %v\n\n", arr, s1)

s1 = append(s1, 8) // append to the slice
fmt.Printf("after append to s1:\narr == %v s1 == %v\n\n", arr, s1)
```

RUN

# Slice

- Beispiel: Kapazität überschritten

```
arr := [5]int{1, 2, 3, 4, 5}
s1 := arr[:3]
fmt.Printf("initial len(s1) == 3, cap(s1) == len(arr):\narr == %v s1 == %v\n\n", arr, s1)

s1 = append(s1, 11, 12, 13) // capacity exceeded, append creates a new array
fmt.Printf("after append to s1 (capacity s1 exceeded):\narr == %v s1 == %v\n\n", arr, s1)

arr[2] = 10 // change the underlying array
fmt.Printf("after change to arr:\n == %v s1 == %v\n\n", arr, s1)
```

RUN

# Map

- ungeordnete Menge von Schlüssel/Wert Paaren: **`map[T1]T2`**
- alle **Schlüssel** müssen den **gleichen Typ** haben **`T1`**
- alle **Werte** müssen den **gleichen Typ** haben **`T2`**
- für die Schlüssel müssen Operatoren **`==`** und **`!=`** definiert sein: Funktionen, Maps, und Slices können keine Schlüssel sein

# Map

- Initialisierung per **`{key: value, key: value, ...}`**

```
map[int]float64{1: 1.2, 4: 2.4} // [1:1.2 4:2.4]
```

- Initialisierung mit der built-in Funktion **`make`**

```
// make(map[T1]T2 oder make(map[T1]T2, initialCapacity)
make(map[string]string)
make(map[string]string, 100)
```

- Zugriff über Indexnotation **`m[key]`**

```
m := make([string]string)
m["foo"] = "bar"   // schreibender Zugriff
s := m["foo"]      // lesender Zugriff
```

# Map

- Werte können aus einer map mit der built-in Funktion **`delete(map, key)`** entfernt werden

```
delete(s, "foo")
```

- Prüfen ob ein Wert für einen Schlüssel existiert

```
m := map[string]string{"foo": "bar"}
value = m["bar"]
fmt.Printf("%#v, \n", value)
value, ok = m["foo"]
fmt.Printf("%#v, %v\n", value, ok)
value, ok = m["bar"]
fmt.Printf("%#v, %v\n", value, ok)
```

RUN

- Maps haben **`call-by-reference`** Semantik

# Weitere built-in Typen

- function types
- pointer types
- channel types
- error type

# Benutzerdefinierte Typen

- werden mit dem **`type`** keyword definiert
- können ein alias für built-in Typen sein

```
type GermanZip int
type MyIntSlice []int
```

- zusammengesetzte Typen werden über das **`struct`** keyword definiert

```
type Person struct {
  Name                      string  // 1. Name, 2. Typ
  Age                       int
  CamelCaseNotationIsCommon bool
}
```

# Structs

- Structs können mit Metadaten annotiert werden

```
type Document struct {
	CurrentPage int `json:"pageNumber"`
	TotalPages  int `json:"TotalPages"`
}
```

- Initialisierung per **`{value, value, ...}`**

```
d := Document{1, 2}
```

- Initialisierung per **`{name: value, name: value, ...}`**

```
d := Document{CurrentPage: 1, TotalPages: 2}
```

# Structs

- Zugriff über **dot** Notation

```
d := Document{CurrentPage: 1, TotalPages: 2}
d.TotalPages // == 2
```

- Sichtbarkeit von Feldern

```
type Person struct {    // public type Person
  id int                // package private property
  Name string           // public property
  Age int               // public property
}
```

- Sichtbarkeit bezieht sich immer auf Package-Basis, *nicht* auf Struct-Basis

# Variablen

## Initialisierung / Zuweisung

- built-in typen

```
// Variante 1
var i int
i = 5

// Variante 2
k := int(5)

// Variante 3
j := 5 // <- gebräuchliste Variante
```

# Variablen

## Initialisierung / Zuweisung

- eigene primitive Typen

```
type GermanZip int

// Variante 1
var z GermanZip
z = 5

// Variante 2
z2 := GermanZip(5) // <- gebräuchliste Variante
```

# Variablen

## Initialisierung / Zuweisung

- structs

```
type Person struct {
  Name string  // 1. Name, 2. Typ
  Age int
}

// Variante 1
var p1 Person
p1.Name = "Franz"
p1.Age = 23

// Variante 2 & 3
p2 := Person{"Franz", 23} // <- häufige Variante
p3 := Person{Age: 23, Name: "Franz"}  // <- gebräuchliste Variante

// Variante 4
var p4 Person
p4 = Person{"Franz", 23}
```

# Variablen

## Initialisierung / Zuweisung

- Zero-Values

```
package main

import "fmt"

func main() {
    var b bool
    var s string
    var i int
    var f float64

    var a [10]int
    var sl []int
    var m map[int]int
    var pi *int

    fmt.Printf("bool: '%#v'\nstring: '%#v'\nint: '%#v'\nfloat64: '%#v'\narray: '%#v'\nslice: '%#v'\nm
}
```

RUN

# Initialisierung / Zuweisung

- Achtung:

```
package main

import "fmt"

func main() {
    i := 1

    // the loop body creates a new scope
    for _ = range [3]int{} {
        fmt.Printf("%d\n", i)
        i := i + 2
    }
}
```

RUN

# Erzeugen mit new & make

- **`new`**: alloziert / initialisiert einen Typ **`T`**, setzt Zero-Werte und liefert einen Pointer auf **`T`** zurück: **`new(T)`**. Heute nicht mehr gebräuchlich, stattdessen **`&`**
- **`make`**: alloziiert / initialisiert ein **`slice`**, **`map`** oder **`channel`**, setzt Zero-Werte und liefert ein Value zurück

```
type S struct { i int }

s := new(S)

ap := new([10]int)
a := make([]int, 10, 100)

mp := new(map[string]int)
m := make(map[string]int, 100)

fmt.Printf("%#v\n%#v\n%#v\n%#v\n%#v", s, ap, a, mp, m)
```

RUN

# Type conversion

- Konvertieren von primitiven Typen:

```
i1 := 10
j1 := float64(i1)

i2 := 10.5
j2 := int(i2)
// j2 := string(i2)

fmt.Printf("i1 == %v (%#T), j1 == %v (%#T)\n", i1, i1, j1, j1)
fmt.Printf("i2 == %v (%#T), j2 == %v (%#T)\n", i2, i2, j2, j2)
```

RUN

# Type conversion

- Konvertieren von Structs:

```
type S1 struct {
    a string
    b string
    //c string
}
type S2 struct {
    a string
    b string
}

s1 := S1{"abc", "def"}
s2 := S2{"uvw", "xyz"}

s3 := S1(s2)

fmt.Printf("type of s1: %T\ntype of s2: %T\ntype of s3: %T", s1, s2, s3)
```

RUN

# Übung 3 / 4: Eigene Datenypen

```
instructions/03_instructions.md
instructions/04_instructions.md
```

# Operatoren

## Arithmetische-Operatoren

```
+    sum                    integers, floats, complex values, strings (concat)
-    difference             integers, floats, complex values
*    product                integers, floats, complex values
/    quotient               integers, floats, complex values
%    remainder              integers

&    bitwise AND            integers
|    bitwise OR             integers
^    bitwise XOR            integers
&^   bit clear (AND NOT)    integers

<<   left shift             integer << unsigned integer
>>   right shift            integer >> unsigned integer
```

# Operatoren

## Vergleichs Operatoren

```
==    equal
!=    not equal
<     less
<=    less or equal
>     greater
>=    greater or equal
```

## Logische Operatoren

```
&&    conditional AND    p && q  is  "if p then q else false"
||    conditional OR     p || q  is  "if p then true else q"
!     NOT                !p      is  "not p"
```

# Funktionen

- werden mit dem **`func`** keyword definiert

```
func myFunc() {
     // do something useful here
}
```

- Funktion mit Parametern und Rückgabewert

```
func awesomeRandom(min int64, max int64) int64 {
     return max - min
}
```

# Funktionen

- Funktion können mehr als einen Rückgabewert haben

```
func funcWithMultipleReturnValues(n int, s string) (int, bool) {
     // do something useful here
     return 100, true
}
```

- Rückgabewerte können Bezeichner haben

```
// named return
func funcWithNamedReturnValues(n int, s string) (n int, ok bool) {
     n = someOtherFunc()
     ok = isValid(n)
     return
}
```

# Funktionen

## variadic

- ... Parameter

```
package main

import "fmt"

func foo(b string, a ...string) {
    fmt.Printf("%s %#v", b, a)
}

func main() {
    foo("wem", "gehören", "meine", "Socken?")
}
```

RUN

# Funktionen

- Funktionen sind *first class citizens* und können Variablen zugeordnet werden

```
package main

func main() {
    f := func() {
        println("hallo")
    }
    f()
}
```

RUN

```
package main

func main() {
    func() {
        println("hallo")
    }()
}
```

RUN

# Funktionen

- Funktionen können übergeben und / oder zurückgegeben werden

```
package main

func greet(f func() string) func() {
    return func() {
        println(f())
    }
}

func main() {
    greet(func() string { return "hallo" })()
}
```

RUN

# Fehlerbehandlung

- Go hat keine Exceptions
- Funktionen zeigen Fehler über einen zweiten Rückgabewert an: **`return val, err`**
- **`err`** ist überlicherweise vom Typ **`error`** (interface)

```
import "errors"
import "fmt"

// neuen Fehler erzeugen:
err1 := errors.New("emit macho dwarf: elf header corrupted")

// oder noch praktischer:
err2 := fmt.Errorf("Fehler: %s", err1)
```

godoc.org/builtin#error

# Fehlerbehandlung

- Go Code sieht in der Regel so aus:

```
val, err := foo()
if err != nil {
  return fmt.Errorf("error while calling foo: %s", err)
}

val2, err := bar()
if err != nil {
  fmt.Fprintf(os.Stderr, "error while calling bar: %s", err)
  os.Exit(1)
}

val3, err := baz()
if err != nil {
  fmt.Fprintf(os.Stderr, "error while calling baz: %s", err)
  os.Exit(1)
}
```

# Fehlerbehandlung

- Wie räume ich ohne **finally** Ressourcen auf?

```
func MergeFile(dstFile, srcFile string) (int, error) {
    src, err := os.Open(srcFile)
    if err != nil {
        return 0, err
    }

    dst, err := os.Create(dstFile)
    if err != nil {
        // leaks src
        return 0, err
    }
    mergedLines := merge(dst, src)

    dst.Close()
    src.Close()
    return mergedLines, nil
 }
```

# Fehlerbehandlung

- **defer** wird nach dem **return** der Funktion ausgeführt

```
func MergeFile(dstFile, srcFile string) (int, error) {
    src, err := os.Open(srcFile)
    if err != nil {
        return 0, err
    }
    defer src.close()

    dst, err := os.Create(dstFile)
    if err != nil {
        return 0, err
    }
    defer dst.close()

    mergedLines := merge(dst, src)

    return mergedLines, nil
 }
```

# Fehlerbehandlung

- Argumente werden evaluiert, wenn das **`defer`** Statement evaluiert wird

```
package main

import "fmt"

func deferArgs() {
    i := 0
    defer fmt.Println(i)
    i++
    return
}

func main() {
    deferArgs()
}
```

RUN

# Fehlerbehandlung

- **`defer`** Statements werden in *Last In First Out* Reihenfolge ausgeführt

```
package main

import "fmt"

func deferLifo() {
    for i := 0; i < 5; i++ {
        defer fmt.Println(i)
    }
}

func main() {
    deferLifo()
}
```

RUN

# Fehlerbehandlung

- **`defer`** kann die Rückgabewerte eines **named return** ändern

```
package main

import "fmt"

func namedReturn() (s string) {
    s = "return from function"
    //defer func() {
    //    s = "return from defer"
    //}()
    return
}

func main() {
    fmt.Println(namedReturn())
}
```

RUN

# Fehlerbehandlung

## panic() und recover()

- ein Aufruf von **`panic()`** beendet sofort die Ausführung der aktuellen Funktion. Alle **`defer`** werden ausgeführt, danach kehrt die aktuelle Funktion zu ihrem Aufrufer zurück. Die Rückkehr bewirkt beim Aufrufer **`panic()`**. Die geht solange bis das Ende des Stacks erreicht ist und das Programm abstürzt oder vorher **`recover()`** aufgrerufen wurde.
- **`recover`** fängt ein **`panic()`** ab. Rückgabewert ist das Argument mit dem **`panic()`** aufgerufen wurde. Außerhalb von **`defer`** ist **`recover`** sinnlos und gibt immer **`nil`** zurück.

# Fehlerbehandlung

```
func main() {
    foo()
    fmt.Println("Returned normally from foo.")
}

func foo() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered in foo with value", r)
        }
    }()
    value, _ := rand.Int(rand.Reader, big.NewInt(2))
    random := value.Int64()
    bar(random)
    fmt.Println("Returned normally from bar.")
}

func bar(i int64) {
    if i > 0 {
        fmt.Println("Panicking in bar!")
        panic(fmt.Sprintf("%v", i))
    }
    fmt.Println("About to exiting bar.")
}
```

RUN

# Kontrollstrukturen

Go hat drei Kontrollstrukturen:

- `if/else` `if/else`
- `switch/case/default`
- `for` `loops`

# Kontrollstrukturen

Go hat **`if/else`** **`if/else`** in zwei Ausprägungen

- das klassische oder mit extra statement
- die geschweiften Klammern **`{}`** um den Codeblock sind obligatorisch. Runde Klammern **`( )`** um die Bedingungen entfallen

# Kontrollstrukturen

- **`if/else if/else`** klassisch

```
m := map[string]string{
    "bar": "baz",
    "foo": "",
}

k := "bar"
v := m[k]

if v == "" {
    fmt.Printf("m[%#v] value is the empty string\n", k)
} else {
    fmt.Printf("m[%#v] == %#v\n", k, v)
}
```

RUN

# Kontrollstrukturen

## `if/else if/else` mit extra statement

```
m := map[string]string{
    "bar": "baz",
    "foo": "",
}

k := "bar"

if v, ok := m[k]; !ok {
    fmt.Printf("m[%#v] does not exists\n", k)
} else if v, ok = m[k]; ok && v == "" {
    fmt.Printf("m[%#v] value is the empty string: %v\n", k, v)
} else {
    fmt.Printf("m[%#v] == %#v\n", k, v)
}
```

RUN

# Kontrollstrukturen

- **`switch (exp|stmt exp|stmt|) {}`**
- **`case`** Klauseln können **`1...n value|exp`** enthalten
- **`case`** Klauseln werden von oben nach unten evaluiert
- explizites **`fallthrough`** statt **`break`**
- kann statt **values** auch **types** vergleichen

# Kontrollstrukturen

## **switch** mit **values**

```
    value, _ := rand.Int(rand.Reader, big.NewInt(6))

    v := value.Int64() // v is random int64 between 0 and 5

    fmt.Printf("v == %v\n", v)

    switch v {
    case 0:
        fmt.Println("Value is 0")
    case 1:
        fallthrough
    case 2:
        fmt.Println("Value is 1 or 2")
    default:
        fmt.Println("Value is 3, 4 or 5")
    }

    fmt.Printf("v == %v\n", v)
```

RUN

# Kontrollstrukturen

## `switch` mit `types`

```
types := [...]interface{}{nil, new(error), errors.New("err"), new(io.Reader)}

value, _ := rand.Int(rand.Reader, big.NewInt(4))
random := value.Int64()

t := types[random]

switch t.(type) {
case nil:
    fmt.Printf("t == %v is nil", t)
case *error:
    fmt.Printf("t == %v is of type *error", t)
case error:
    fmt.Printf("t == %v is of type error", t)
default:
    fmt.Printf("don't know the type of t == %#v", t)
}
```

RUN

# Kontrollstrukturen

Es gibt in Go nur ein Schleifenkonstrukt, die **for-loop**:

- als **`for`** und **`while`** nutzbar
- zur **Iteration** über Arrays, Slices, Maps, Strings und Channel
- unterstützt **`break`**/**`continue`** (optional mit **Label**)

# Kontrollstrukturen

## for-loop

- klassich: **`for <init>; <condition>; <post> {...}`**

```
for i := 5; i > 0; i-- {
    fmt.Printf("%v\n", i)
}
```

RUN

# Kontrollstrukturen

## for-loop

- wie eine **while(<exp>)**: **for <exp> {...}**

```
j := 5
for j > 0 {
    fmt.Printf("%v\n", j)
    j--
}
```

RUN

# Kontrollstrukturen

## for-loop

- wie eine **while(true)** bzw. **for(;;)**: **for {...}**

```
k := 5
for {
    fmt.Printf("%v\n", k)
    k--
    if k <= 0 {
        break
    }
}
```

RUN

# Kontrollstrukturen

## for-loop

- Iteration mit **`range`**

```
for idx, value := range [...]int{5, 4, 3, 2, 1, 0} {
    fmt.Printf("%v (idx: %v)\n", value, idx)
}

for key, value := range map[string]string{"5": "five", "4": "four", "3": "three"} {
    fmt.Printf("key: %v, value: %v\n", key, value)
}

for idx, rune := range "aö日本語" {
    fmt.Printf("idx: %v, rune: %v, as str: %#v, type: %T\n", idx, rune, string(rune), rune)
}
```

RUN

# Methoden

- Funktionen die auf eine Typ **`T`** definiert sind:

**`func (t T) myMethod() {}`**

- Aufruf per **dot** Notation: **`t.method()`**

```
type Person struct {
    Name string
    Age  int
}

func (p Person) String() string {
    return fmt.Sprintf("name: %s, age: %d", p.Name, p.Age)
}

func main() {
    person := Person{Name: "Franz", Age: 23}
    println(person.String())
}
```

RUN

# Methoden

- der **Typ** auf dem die Methode definiert ist heißt: **receiver**

```
package main

import "fmt"

type GermanZip int

func (z GermanZip) String() string {
    return fmt.Sprintf("Plz %d", z)
}

func main() {
    plz := GermanZip(10245)
    println(plz.String())
}
```

RUN

# Übung 5: String() Methode

```
instructions/05_instructions.md
```

## Hinweise

- die Methode **`Sprintf`** aus package **`fmt`** nutzen / oder **`+`** Operator
- Iteration über ein Array mit **`for`** und **`range`**:

```
for i, element := range array {
  // ...
}


// Funktionssignatur
func (s Stats) String() string {

}
```

# Übung 6: error handling

```
instructions/06_instructions.md
```

## Hinweise

- Konvertieren eines Byte Arrays in einen String:

```
// []byte -> string
string(bytevar)
```

- error handling idiom

```
value, err := something()
if err != nil {
        return nil, fmt.Errof("error message: %s", err)
}
```

- Ein Datei vollständig einlesen: **`ioutil.ReadFile`**
- Erstellen eines Templats mit: **`template.New`**

# Embedding

- Typen können in andere Typen eingebettet werden

```
type MyType struct {
     x int
}
func (myType MyType) foo() string {
     return "bar"
}
type MyOtherType struct {
     MyType // MyOtherType is anonymous
}
```

- alle Felder und Methoden von **`MyType`** sind in **`MyOtherType`** verfügbar

```
m := MyOtherType{MyType{1}}
m.x
m.foo()
```

# Embedding

```
package main
import "fmt"

type Person struct{ Name string }

func (p Person) String() string {
    return "Name: " + p.Name
}

type Employee struct {
    Person
    id int
}

func main() {
    e := Employee{Person: Person{Name: "brunhilde"}, id: 1}

    fmt.Printf("String method: %s\nProperty: %s", e.String(), e.Name)
}
```

RUN

# Embedding

- Was passiert bei Nameskonflikten?

```
type Person        struct{ Name string }
type SomethingElse struct{ Name string }

func (p Person)        String() string { return p.Name }
func (s SomethingElse) String() string { return s.Name }

type Employee struct {
    Person
    SomethingElse
    // Name string
}

func main() {
    e := Employee{Person{Name: "brunhilde"}, SomethingElse{Name: "friedburg"}}

    fmt.Printf("String method: %s\nField: %s", e.String(), e.Name)
}
```

RUN

# Übung 7: die init() Funktion

```
instructions/06_instructions.md
```

- Auf Package Ebene deklarieren:

```
var indexTemplate *template.Template
```

- Signatur:

```
func init() { /* init code here */ }
```

- aus Übung 6 aufrufen:

```
NewTemplateFromFile(fileName string) (*template.Template, error)
```

# Interfaces

- Definiert eine Menge von Methoden

```
type MyInterface interface {
     MyMethod() int
     MyOtherMethod() string, err
}
```

- Typen auf denen alle Methoden eines Interface definiert sind, implementieren das Interface
- keine explizite Deklaration nötig
- das leere Interface **`interface{}`** wird vom jedem Typ implementiert, seit Go 1.18 (Generics) auch **`any`**

# Interfaces

```
type Shape interface {
    Area() float64
}

type Rectangle struct{ length, width int }
type Circle struct{ radius int }

func (r Rectangle) Area() float64 { return float64(r.length * r.width) }
func (c Circle) Area() float64    { return float64(c.radius) * float64(c.radius) * math.Pi }

func printArea(x Shape) {
    fmt.Printf("Fläche von %#v: %.2f\n", x, x.Area())
}

func main() {
    rectangle := Rectangle{5, 3}
    printArea(rectangle)
    circle := Circle{radius: 4}
    printArea(circle)
}
```

RUN

# Interfaces

- Beispiel Interface: Stringer

```
type ComplexNumber struct {
    Real      int
    Imaginary int
}

// func (c ComplexNumber) String() string {
//   return fmt.Sprintf("%d+%di", c.Real, c.Imaginary)
// }

func main() {
    complex := ComplexNumber{4, 2}
    fmt.Print(complex)
}
```

RUN

golang.org/pkg/fmt/#Stringer

# Interfaces

## type assertion

- Interfaces lassen sich auf ihren konkreten Typ casten: **`x.(T)`**

```
var i interface{} = 5

j := i
//j := int(i)
//j := i.(int)          // type assertion succeed
//j := i.(float64)      // type assertion fails
//j, ok := i.(float64) // check if type assertion succeed; if !ok j will be the zero value
j++
//if ok { j++ }
fmt.Println(j)
```

RUN

# Generics

```
func Map[I any, O any](in []I, f func(I) O) []O {
    var out []O
    for _, v := range in {
        out = append(out, f(v))

    }
    return out
}

func main() {
    floats := []float32{1.2, 1.5, -3.5}

    s2i := func(f float32) int {
        f = f * f
        return int(f)
    }

    intSquares := Map[float32, int](floats, s2i)

    fmt.Println(intSquares)
}
```

RUN

# Generics

```
type Unsigned interface {
    uint | uint8 | uint16 | uint32 | uint64
}

func Sum[U Unsigned](u []U) U {
    var s U
    for _, v := range u {
        s = s + v
    }
    return s
}

func main() {
    fmt.Println(Sum([]uint32{3214, 46456, 534535}))
    fmt.Println(Sum([]uint8{3, 4, 5}))
}
```

RUN

# Pointer

- ja: go hat pointer! Aber: keine Pointer-Arithmetik
- Deklaration eines Pointers auf den Typ **T**: ***T**
- mit dem * Operator kann der Wert eines Pointers dereferenziert werden: ***x** ist der Wert auf den der Pointer **x** zeigt
- mit dem **&** Operator können Pointer erzeugt werden: **&x** erzeugt einen Pointer auf **x**.

# Pointer

```
i := 5
j := &i     // j is pointer to i

fmt.Printf("i == %v, *j == %v\n\n", i, *j)

// increase i
i++
fmt.Printf("after increasing i:\ni == %v, *j == %v\n\n", i, *j)

// increase the value j points to
*j++
fmt.Printf("after increasing *j:\n i == %v, *j == %v\n\n", i, *j)

fmt.Printf("the actual value of j is an address: j == %v\n", j)

// j++ // no pointer arithmetic
```

RUN

# Pointer

```
type cnt int

func (c cnt) inc() {
    c++
}

func main() {
    c := cnt(1)

    fmt.Printf("before inc(): %d\n", c)

    c.inc()

    fmt.Printf("after inc(): %d\n", c)
}
```

RUN

# Pointer

## value / pointer receiver

- Automatische Konvertierung nach **`(&T).method()`** und **`(*T).method()`**

```
type cnt int

func (c cnt) inc() {
    c++
}

func main() {
    c := cnt(1)
    //d := &c
    fmt.Printf("before inc(): %d\n", c)
    c.inc()       // automatic conversion to (&c).inc() for pointer receiver
    //d.inc()     // automatic conversion to (*d).inc() for value receiver
    fmt.Printf("after inc(): %d\n", c)
}
```

RUN

# Übung 8: Pointer

```
instructions/08_instructions.md
```

```
package main
import "fmt"

type array []int

func (a array) push(i int) {
    for _, e := range a {
        if e == i {
            return
        }
    }
    a = append(a, i)
}

func main() {
    a := array{1, 2}
    fmt.Printf("%#v\n", a)

    a.push(3)
    a.push(1)
    fmt.Printf("%#v\n", a)
}
```

RUN

# Concurrent Go

- Basiert auf Tony Hoares CSP
- Message-Passing mit *Channels*
- *Processes* == Funktionen / Methoden
- Parallelität mit *Go - Routinen*

# Go-Routinen

- sind normale Funktionen oder Methoden, die mit dem Schlüsselwort **`go`** aufgerufen werden
- Go-Routinen blockieren nicht
- Leightweight-Threads / Green-Threads: werden auf OS-Threads verteilt
- Synchronisation & Kommunikation über Channels

# Go-Routinen

- Der Aufruf einer Go-Routine ist ein Statement, keine Expression

```
package main

func helloAutraliaString() string {
    return "¡ɐıl ɐɹɐʇsnA ollǝɥ"
}

func main() {
    s := helloAutraliaString()
    println(s)
}
```

RUN

- Funktionen die mit **go** aufgerufen werden dürfen aber Rückgabewerte haben, diese werden dann ignoriert

# Go-Routinen

## Nebenläufigkeit

```
func print(name string) {
    for i := 1; i <= 100; i++ {
        fmt.Printf("%s ", name)
    }
}

func main() {
    print("*")
    print("-")
    print("|")

    time.Sleep(0 * time.Second)
}
```

RUN

# Go-Routinen

## Synchronisation mit Mutex

```
func print(name string) {
    for i := 1; i <= 100; i++ {
        fmt.Printf("%s ", name)
    }
}

func main() {
    wg := sync.WaitGroup{}
    ccPrint := func(name string) {
        print(name)
        wg.Done()
    }

    wg.Add(3)
    go ccPrint("*")
    go ccPrint("-")
    go ccPrint("|")
    wg.Wait()
}
```

RUN

# Channel

- sind zum Nachrichten schicken und empfangen
- sind typisiert
- können Puffer besitzen
- blockieren wenn sie voll sind und jemand möchte schreiben oder wenn leer sind und jemand möchte lesen
- wie eine FIFO-Pipe

# Channel

- Channel müssen mit **`make`** erstellt werden:

```
c    := make(chan bool)

cBuf := make(chan bool, 100) // Puffer von 100
```

# Channel

- um in den Channel zu schreiben, muss der Channel Operator **<-** auf den Channel zeigen

```
myChannel := make(chan bool)

myChannel <- true
```

# Channel

- um von Channel zu lesen, muss der Channel Operator **<-** vom Channel weg zeigen

```
myChannel := make(chan bool)

...

var x bool

x <- myChannel

// variable initialisieren und setzen durch Channel Werte
value := <- myChannel
```

# Channel

- Schreiboperationen blockieren wenn der Puffer voll ist
- Leseoperationen blockieren wenn der Puffer kein Element enthält

```
package main

import "fmt"

func main() {
    c := make(chan bool)

    c <- true    // in den Channel schreiben

    value := <-c // aus den Channel lesen

    fmt.Printf("%v\n", value)
}
```

RUN

# Channel

- Channel können von verschiedenen Consumern/Producern geteilt werden
- Lese-/Schreiboperationen sind synchronisiert

```
func main() {
    c := make(chan string, 0)
    produce := func(name string) {
        for i := 1; i <= 100; i++ {
            c <- name
        }
    }

    go produce("*")
    go produce("-")
    go produce("|")
    for i := 0; i < 300; i++ {
        fmt.Print(<-c + " ")
    }
}
```

RUN

# Channel

## Leseoperation multiplexen mit select

```
func produce(c chan string, name string, finished chan bool) {
    for i := 1; i <= 100; i++ {
        c <- name
    }
    finished <- true
}

func main() {
    f := make(chan bool) // finished channel
    a := make(chan string); b := make(chan string); c := make(chan string)
    go produce(a, "*", f);  go produce(b, "-", f);  go produce(c, "|", f)

    cnt := 0
    for {
        select {
        case value := <-a: fmt.Print(value + " ")
        case value := <-b: fmt.Print(value + " ")
        case value := <-c: fmt.Print(value + " ")
        case <-f         : if cnt++; cnt == 3 { return }
        }
    }
}
```

RUN

# Übung 9: Concurrency Issues

```
instructions/09_instructions.md
```

# Go Best Practice

- avoid (deep) nesting (use early returns)
- accept interfaces, return structs
- useful, reasonable zero values
- use struct literal initialization (avoids invalid intermediate state)
- avoid concurrency in your API
- avoid stateful packages
- no panic() should cross a package boundary

# Übung 10: HTTP Server

```
instructions/10_instructions.md
```

- Hinweise

go.dev/pkg/net/http/

# Go Pitfalls

- re-slicing
- leaking go routines
- `:=` - multiple assignments, variable shadowing
- **`nil`** - two nil values may be not equal
- pointer / value receivers

# Übung 11: (Paralleles) Scraping

```
instructions/11_instructions.md
```

# Dependency Management

- **`$GOPATH`**: Dependencies sind global und immer **`HEAD`**
- **`$GOPATH`** == /home/go (seit Go 1.8)
- **Vendoring** (seit Go 1.5): Dependencies werden in den Quelltextbaum kopiert
- 3rd-Party tools: glide, gom, goop, gopm, gb, ...
- Modules: **`go mod`** (Go 1.11): Module mit festen Version und klaren Abhängigkeiten

# Übung 12: CLI-parameter

```
instructions/12_instructions.md
```

# Tooling

- testing: **go test**
- coverage: **go test -cover**, **go tool cover**
- profiling and benchmarking: **go tool pprof**, **go test -bench**
- fuzzing: **go test -fuzz**
- documentation: **godoc**, **go doc**
- linting: **gofmt**, **go fmt**, **golint**
- static analysis: **go vet**
- race detector: **go build -race**
- debugging: **delve**

# Fragen? Antworten!

Christoph Iserlohn
christoph.iserlohn@innoq.com

innoQ Deutschland GmbH