Teil 1 Go - Programmierung

by Mohammadimohammadi

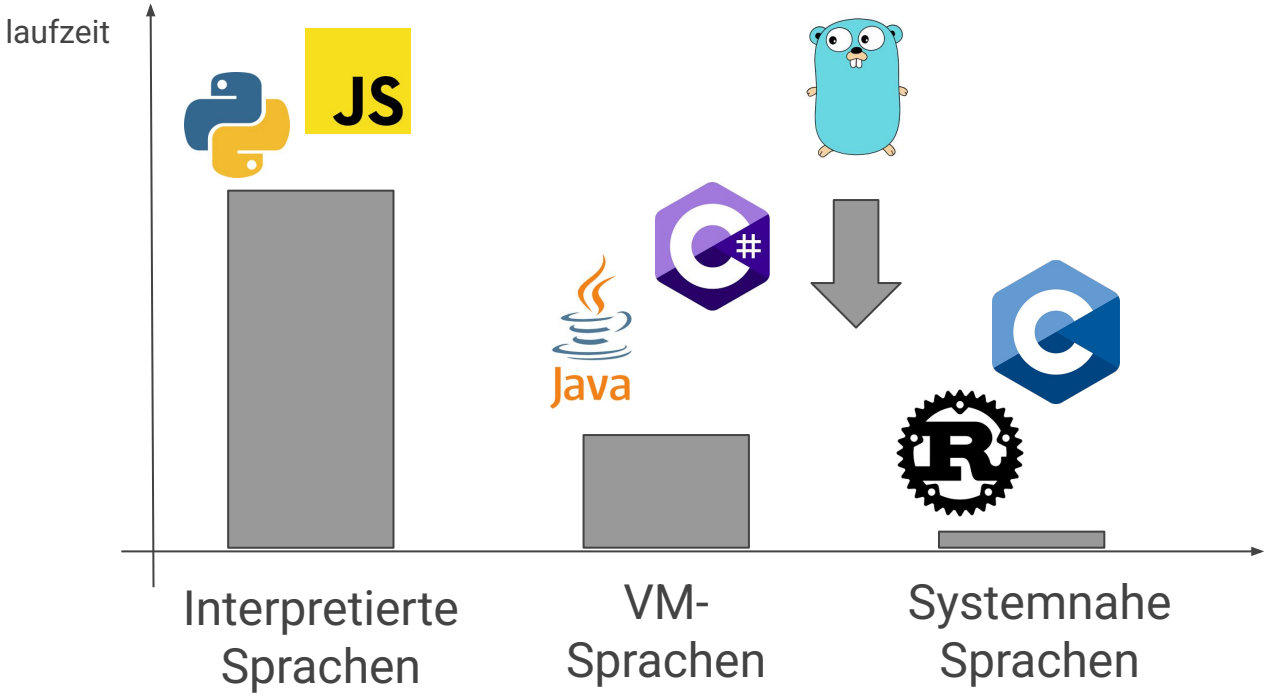
Agenda für Teil 1

1) Einleitung in Go
2) Erstes Programm (Demo)
3) Datentypen + Operatoren
4) Variablen / Konstanten
5) Typ Konvertierung
6) Bedingte Anweisung

Go

- Von Google entwickelt (2012)
- Kompilierte Programmiersprache
- Garbage Collected
- Besonders geeignet für (z. B. parallele Programmierung, Microservices)

Motivation Performance



Erstes Programm (Demo)

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, World")
}
```

Main Funktion

```
func main() {
    fmt.Println("Hello, World")
}
```

- func: keyword für Funktionen deklarationen
- main(): besondere Funktion, fungiert als Start des Programms
- { . . . }: Ausführungs Block
- fmt ist die Bibliothek und Println ein Funktion in der Bibliothek

Import

- Paket Anbindung
- "fmt" format Paket
- "fmt" ist eine **Standard** Paket
  - (ohne installation)
- Dabei existieren auch **Drittanbieter**-Paketen

```
import "fmt"
// oder mehrfach
import (
    "fmt"
    "time"
)
```

Paket / Bibliothek

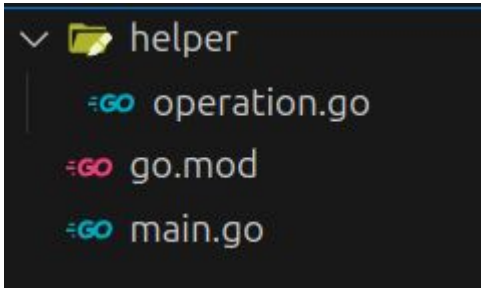
`package <name>`

- package <name> ist ganz oben in der Datei angegeben
- Jede Datei in einem Verzeichnis ist genau einem Paket zugeordnet
- Alle Elemente in dieser Datei sind dann Teil dieses Pakets
- Der <name> bestimmt, wie andere Dateien oder Pakete darauf zugreifen können

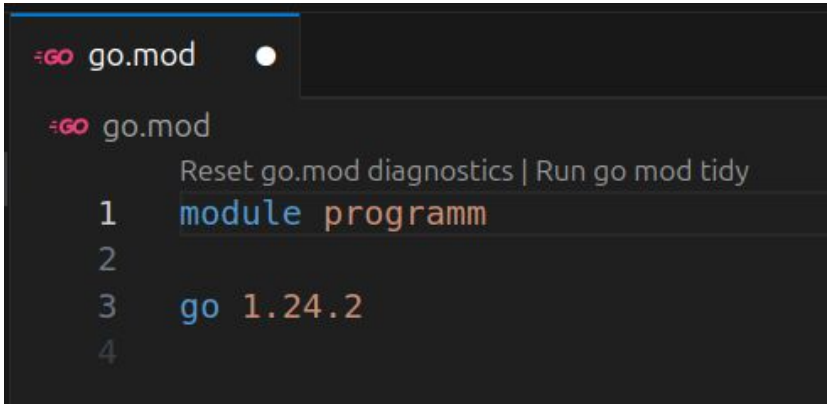
`Spezialfall: package main`

- In diesem Paket muss die `main()` Funktion existieren
- In jedem Projekt muss das main Paket existieren

Beispiel: Paket + Import



- go.mod: Konfigurationsdatei um Libraries zu verwalten (package.json)
  - `go mod init <modul>`
- <Modul>: ist ein Name für das Gesamt Projekt, hier "programm"
- Um öffentliche Pakete zu schreiben muss der Modulname eindeutig sein.



Fazit

`package main:`

- Ort des Main Funktion, existiert pro Verzeichnis nur einmal

`func main() { . . . }`

- Startpunkt jedes Programmes

Datentypen

Warum brauchen wir Datentypen?

- Sicherheit
- Performance
- (Verbessert das Verständnis von Programmiersprachen allgemein)

In Go gibt es zwei Klassen der Datentypen:

- Primitive Datentypen
  - Zahlentypen
  - Boolean
  - usw.
- Referenz Datentypen

Zahlentypen

int8	-128	+127
int16	-32768	+32767
int32	-2147483648	+2147483647
int64	-9 Trillion	+9 Trillion
uint8	0	+255
uint16	0	+65536
uint32	0	+4294967295
uint64	0	+18 Trillion
float32	10^-38	+10^38
float64	10 ^-308	+10^308
complex64	*10^-38*	*+10^38*
complex128	*10 ^-308*	*+10^308*

Plattform Typen:

int: int32 / int64

Alias Typen:

rune: int32
byte: uint8

Arithmetik

Addition: `<zahl> + <zahl> : 2 + 3 == 5`

Subtraktion: `<zahl> - <zahl> : 2 - 3 == -1`

Multiplikation: `<zahl> * <zahl> : 5 * 2 == 10`

Ganzzahldivision:

`<int> / <int> : 5 / 2 == 2`

Division:

`<float> / <float> : 5 / 2 == 2.5`

Modulo: `<int> % <int> : 5 % 2 == 1`

Boolean

bool: true / false

- `true && false`
- `!true || false`
- `false || !false`
- `true && true`

		and	or	not
x	**y**	**x && y**	**x || y**	**! x**
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Literale / Ausdrücke

- Ein Literal ist ein direkt im Quellcode notierter Wert eines bestimmten Datentyps.
- Ausdrücke bestehen aus Literale und/oder Operationen
- Operationen sind Funktionen, die aus ein Input ein Output liefern
- Input und Output können unterschiedliche Typen haben
- Operatoren haben verschiedene Ausführungsreihenfolgen:
    - Arithmetik: "punkt vor strich"

Beispiel Literale / Ausdrücke

- `42`
- `3 * 2 + 4`
- `3.14`
- `“Hallo Welt”`
- `42 - 4`
- `true && false`
- `true`

Vergleichsoperatoren

- gegeben seien zwei Ausdrücke L und R, die beide vom selben Typ T sind

Ein **Vergleich** in der Form L < R, L > R, L <= R oder L >= R ist **nur dann gültig**, wenn der Typ T einen solchen Vergleich unterstützt. Das Ergebnis eines solchen Vergleichs ist ein Ausdruck vom Typ bool.

```
Gleichheitstest: <typ>  ==  <typ> : 2 == 3, true == false, "hello" == "hallo"
                 <typ>  !=  <typ> : 3 != 4, true != false
Großer/Kleiner Test: <zahl/string> <  <zahl/string>
                     <zahl/string> >  <zahl/string>
                      <zahl/string> >= <zahl/string>
                      <zahl/string> <= <zahl/string>
```

Beispiel: Vergleichsoperatoren

```
"abc" <= "xyz"       // true
"apple" >= "banana" // false
10 < 15              // true
9 < 9                // false
32 <= 32             // true
10 != 10             // false
```

Bisherige Operatoren

Logik	`! <bool>`	`<bool> && <bool>`	`<bool> || <bool>`			
Arithmetik	`x + y`	`x - y`	`x * y`	`x / y`	`x % y`	
Vergleiche	`x == y`	`x != y`	`x > y`	`x >= y`	`x < y`	`x <= y`

* wobei x und y gleiche typen sein müssen

Ausgabe in der Konsole

- Einbindung von fmt Library
- Um Ergebnisse darzustellen / debugging

```
fmt.Println(...), fmt.Print(...)
```

```
_
```

```
1 fmt.Print("Hallo")
```

```
2 fmt.Println(1)
```

```
3 fmt.Println("Hallo", 4 + 1)
```

Variablen

- Situation: Wir wollen Werte speichern und damit arbeiten/rechnen
- Best Practices:
  - Sprechbare <name> aussuchen (camelCase, snake_case)
  - <type> so klein wie möglich wählen

Syntax:

```
var <name> <type> [= <wert>]
```

alternativen:

```
var <name> = <wert>

<name> := <wert>
```

Eingabe aus der Konsole

- Einbindung von fmt Library
- Um Benutzereingaben zu verarbeiten

```
fmt.Scan(...)
```

-

```
var x int8 = 0

fmt.Scan(&x)

fmt.Println(x)
```

Zuweisungsoperator

Syntax: `<var> = <wert>`

Bedeutet: <Wert> überschreibt den Wert von <var>

Voraussetzung: Identische Datentypen / Zuweisung kompatibel

-

```
var x int32 // x = 0
x = 42      // x = 42
x = x + 22  // x = 64
```

`x = x + 3`	`x += 3`
`x = x * 2`	`x *= 2`
…	…
`x = x + 1`	`x++`
`x = x - 1`	`x--`

Bisherige Operatoren

Logik	`! <bool>`	`<bool> && <bool>`	`<bool> || <bool>`			
Arithmetik	`x + y`	`x - y`	`x * y`	`x / y`	`x % y`	
Vergleiche	`x == y`	`x != y`	`x > y`	`x >= y`	`x < y`	`x <= y`
Zuweisung	**`x += y`**	**`x -= y`**	**`x *= y`**	**`x /= y`**	**`x %= y`**	**`x = y`**

* wobei x und y gleiche typen sein müssen

Konstanten

Situation: Verwendung von Datentypen die sich nicht **Verändern.**

Syntax: `const <name> [<type>] = <wert>`

-

```
const Pi float32 = 3.1415

const MwSt = 0.19

MwST = 0.11
```

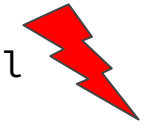
Beispiel: Konstanten

```
var result = 10 * (20 * 0.19)
```

```
const MwSt = 0.19
const articlePrice = 20
var articleCount = 10
var result = articleCount * (articlePrice * MwSt)
```

Typ Konvertierungen

- Ein Ausdruck von **`T(v)`** konvertiert den Wert von **`v`** zu den Typ **`T`**.
- Beispiel:
  - `var zahl int32 = 42`
  - `var kommazahl float32 = zahl`



  - `var kommazahl float32 = float32(zahl)`

Beispiel: Typ Konvertierungen

```
var x int32 = 0
x = 10
x = x * 10

var y int32 = 0
y = y * int32(10)
y = y * 10

var z int64 = x * 10
var z int64 = int64(x) * 10
```

```
var a int32 = 100
var b int64 = 200

var result = a + b
var result = int64(a) + b

const MwSt = 0.19
const articlePrice = 20
var articleCount = 10
var result = articleCount * (articlePrice * MwSt)
var result =
      float32(articleCount) * (articlePrice * MwSt)
```

IF Bedingungen

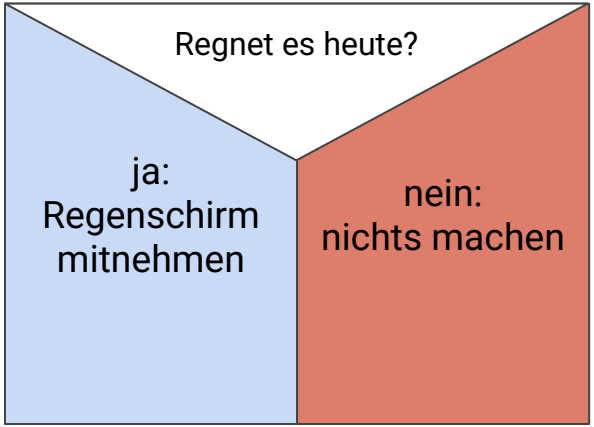
Situation: Fallunterscheidung nach bestimmte Kriterien.

Syntax:

```
if <bool> {

} [ else [ if <bool>] {

} ]
```




```
if isRaining {
    takeUmbrella();
} else {
    // do nothing
}
```

Ausführungs Block

```
{
    var x byte = 0
    // ab hier x sichtbar


}   // ab hier wird x freigegeben
```

- Deklarationen sind nach einer schließende Klammer `}` nicht mehr sichtbar.

Beispiel: Ausführungs Block

```
var alter byte = 80

if alter > 65 {
    var istRente bool = true
}

fmt.Println(istRente);
```

Beispiel: Ausführungs Block

```
var x byte = 5

if x > 0 {
    var x byte = 10;
    x++;
    x *= 5
}

fmt.Println(x);
```

Zusammenfassung

Struktur eines Go-Programms:

- `package main` + `func main() {}` = Startpunkt
- Import von Standard- & benutzerdefinierten Paketen

Datentypen & Operatoren:

- Primitive: `int`, `float64`, `bool`, `string`
- Operatoren: +, -, *, /, uvm.

Variablen & Konstanten:

- `var`, `:=` für Variablen
- `const` für feste Werte
-

Kontrollstrukturen:

- `if / else` zur Fallunterscheidung
- Sichtbarkeit (Scope) innerhalb von Blöcken beachten

Ein- & Ausgabe:

- `fmt.Println()` für Ausgabe
- `fmt.Scan()` für Benutzereingaben