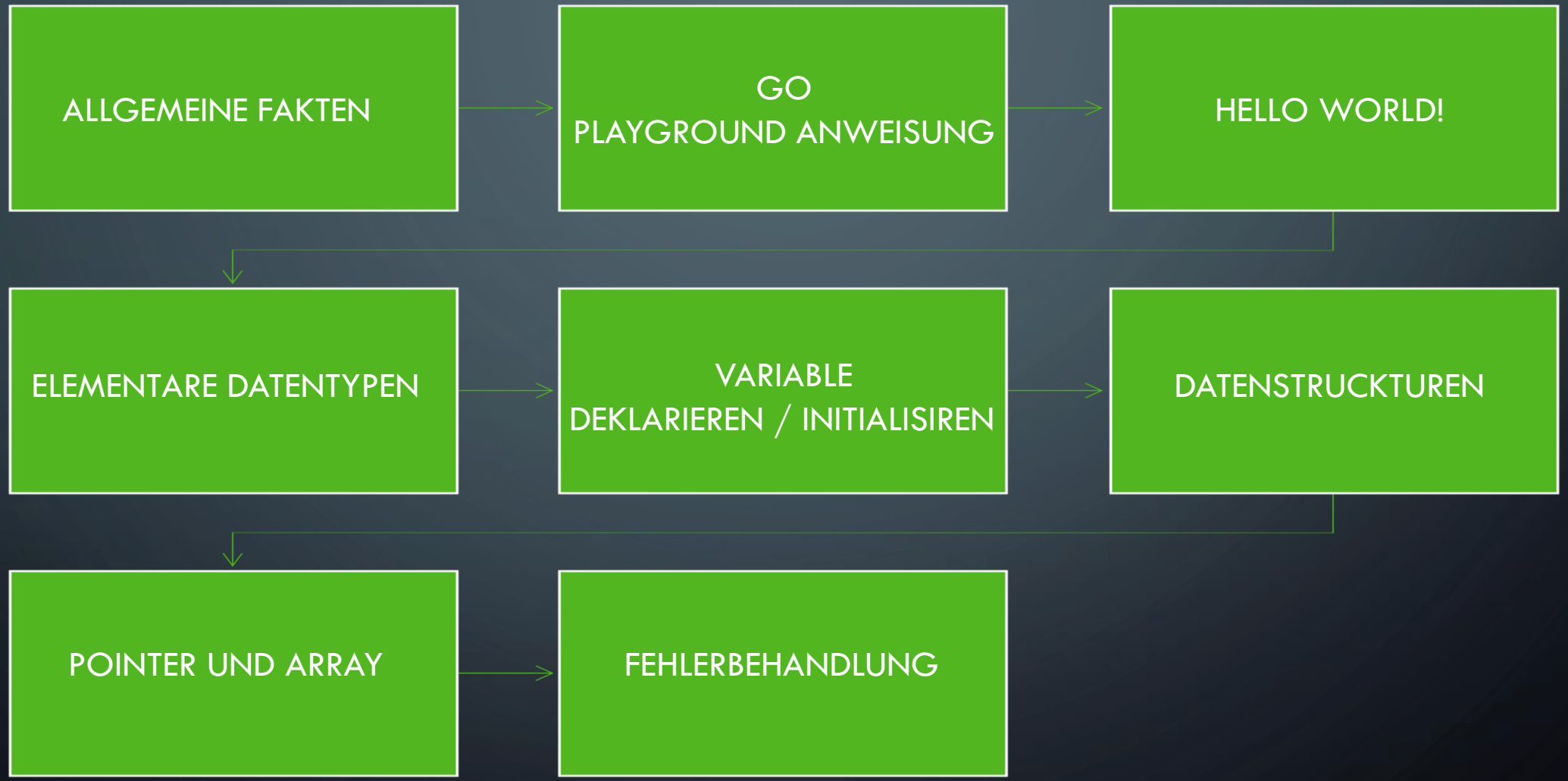
Eine Präsentation von Team C-Killer

Gebastelt von: Mohammad, Caner, Aya, Muzamil, Yamen, Oliver, Laila und Odin

INHALTSVERZEICHNIS

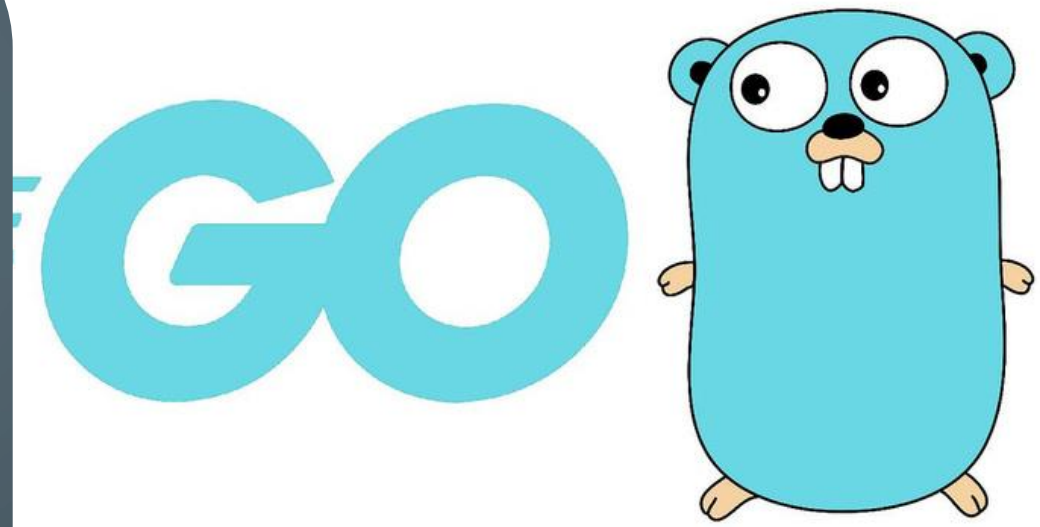


GESCHICHTE

- Golang (Go) wurde als der wahre C-Killer von einige Ingenieruen aus dem Hause Google entwickelt
- Die Entwürfe stammen von Robert Griesemer, Rob Pike und Ken Thompson
- 2012 erschien die erste Stabile Version der Sprache
- Mittlerweile ist die aktuelle Version -> go 1.22.2 (Stand Mai 2024)

NAMENSURSPRUNG

- Das ist ein Gopher
- Auf deutsch -> Taschenratte
- Grund für die Maskottchenwahl -> ??? Süß



EINFÜHRUNG

- Einige sehr bekannte Unternehmen aus dem IT-Bereich arbeiten mittlerweile mit Go im Backend
  - Google, Uber, Twitch Soundcloud, Dropbox, Netflix und PayPal

- Go ist anscheinen ein infinite Money-Glitch, aber was ist das den jetzt?

NUTZEN

- Kurz -> Eine typsichere, vorkompillierbare, prozedurale Programmiersprache
- Typsicher kennen wir schon von unserem alten Freund Java
- Vorkompilierbar und prozedural kennen wir von unserem Freund C

NACHTEILE

- Benötigt doppelt so viel Speicherplatz als C für eine Excecutable-Datei
- Besitzt nicht die allgemeine Zuverlässigkeit die C für Micro-Controller und Anlagen hat (Stichwort Speicherplatzbelegung und garbage-collection)

VORTEILE (TEIL 1)

- Go ist einfacher zu lernen und hat weniger Keywords als sein Cousinen Java und C++
- Liefert standardmäßig viele nützliche Funktionen für die Parallelität
- Sowie nützliche Tools für die Erstellung von Netzwerkanwendungen

VORTEILE (TEIL 2)

- Vollautomatischer Garbage-Collection (Kennen wir schon aus Java)
- Einfache Lesbarkeit und die Nichterlaubte Pointer-Arithmetik (DANKE)
- Spielend leichte Ausführung von Go-Code
- Durch [go build _dateiName.go] -> innerhalb von Sek. ausführbare Datei
  - Noch einfacherer Befehl -> [go run _dateiName.go]

GRUNDGERÜST & FUNKTIONSWEISE

- Code wird vom Compiler in Bytecode übersetzt (wie C/C++) und es ensteht eine ausführbare Datei
- Go hat viele Eigenschaften von C übernommen und manche sogar besser dargestellt
- Datentypen und Kontrollstrukturen haben ähnlichen Aufbau wie in C/C++/Java (Familie)
- Go verzichtet auf die Objektorientierung
- Stucts, Pointer, Switch-Statements und Functions sind elementar syntaktische Elemente

HELLO WORLD!

ELEMENTARE DATENTYPEN

GANZZAHLEN

- int, int8, int16, int32, int64
  - Sind vorzeichenbehaftet
  - Standard-bit = 32 Bit
  - Default-Value = 0
- uint, uint8, uint16, uint32, uint64
  - Ist nicht vorzeichenbehaftet
  - Soll nicht mit einem negativen Wert initialisiert werden
  - Default-Value = 0

FLIEßKOMMAZAHLEN

- Float32 und Float64
  - Float64 ist das double in Java
  - Standardmäßig verwendet man float64
  - Default-Value = 0

KOMPLEXE ZAHLEN

- Complex64 und Complex128
  - Als Standard wird complex128 verwendet
  - Verwendung für Signalverarbeitungen, Elektrotechnik, Regelungstechnik, etc.
  - Aufbau von Komplex zahlen:
    - Erste Hälfte -> Realteil
    - Zweite Hälfte -> Imaginärteil

```
//var name complex64 = realteil + imaginärteil
var c complex64 = 5 + 5i
fmt.Println(c) // -> (5+5i)
```

BOOLESCHE WERTE

- Wird in Go als bool geschrieben
- Repräsentiert -> false und true
- Konvertierung von Zahl zu Boolean und Boolean zu Zahl möglich
- Default-Value = false

STRINGS

- UTF8 kodiert Zeichen und sind unveränderlich
- Können nicht an dem Index verändert werden, da sonst eine Fehlermeldung kommt
- Initialisierung durch:
  - Anführungszeichen (“ “), kann durch Format-Spezifizierung formatiert werden
  - Backticks (` `) -> kann ohne Format-Spezifizierung formatiert werden

VARIABLE DEKLARIEREN/INITIALISIEREN

- Wenn Variable nicht verwendet wird = Warnung

- Deklarierung ist auf 3 arten möglich:
  - 1.Art -> var <name> datentyp = value
  - 2.Art -> var <name> = value
  - 3.Art (kurzschreibweise) -> <name> := value

```
//1. Art var <name> datentyp = value
var intNum int = 3

//2. Art var <name> = value
var intNum2 = 4

//3. Art <name> := value
intNum3 := 5
```

DATENSTRUKTUREN

BLOCK

- Block wird wie in Java durch -> { } gekennzeichnet

```
// Block
func main() {
    fmt.Println( a...: "Hallo Welt!") //Ausgabe: Hallo Welt!
}
```

IF-ANWEISUNG

- If-Anweisungen haben gleiche Syntax wie in Java, nur ohne Klammern

- Auch möglich:
  - Zuerst eine Funktion aufrufen oder neue Variable erstellen, bevor if-Abfrage gemacht wird

```
// if-Anweisung
x := 5
if x < 13 {
    fmt.Println( a...: "x ist kleiner als 13")
}
// Ausgabe: x ist kleiner als 13
```

```
if x := twoTimes( i: 5); x < 9{
    fmt.Println( a...: "x ist kleiner als 9")
}
// Keine Ausgabe
```

```
if x := 5; x < 13 {
    fmt.Println( a...: "x ist kleiner als 13")
}
// Ausgabe: x ist kleiner als 13
```

IF-ELSE / IF-ELSE-IF ANWEISUNG

```go
// if-else Anweisung
if x := 5; x < 13 {
    fmt.Println( a...: "x ist kleiner als 13")
} else {
    fmt.Println( a...: "x ist größer als 13")
}
//Ausgabe: x ist kleiner als 13
```

```go
//if-else-if-anweisung
if x := 31; x < 10 {
    fmt.Println( a...: "x ist kleiner als 10")
} else if x < 20 {
    fmt.Println( a...: "x ist kleiner als 20, aber größer 10")
} else if x < 30 {
    fmt.Println( a...: "x ist kleiner als 30, aber größer als 10, 20 und 30")
} else {
    fmt.Println( a...: "x ist größer als 10, 20, 30 und 40")
}
// Ausgabe: x ist kleiner als 40, aber größer als 10, 20 und 30
```

- If-else-Anweisung:
  - Syntax ist ähnlich zu der wie in Java
- If-else-if-Anweisung:
  - Wie in Java, beliebig viele else-if möglich und nur ein else

SWITCH-CASE

```go
// switch-case
switch note := "gut"; note {
case "sehr gut":
    fmt.Println( a...: "Deine Note ist zwischen 1.0 - 1.4")
case "gut":
    fmt.Println( a...: "Deine Note ist zwischen 1.5 - 2.4")
case "befriedigend":
    fmt.Println( a...: "Deine Note ist zwischen 2.5 - 3.4")
case "ausreichend":
    fmt.Println( a...: "Deine Note ist zwischen 3.5 - 4.0")
case "ungenügend":
    fmt.Println( a...: "Deine Note ist zwischen 4.1 - 5.0")
}
// Ausgabe: Deine Note ist zwischen 1.5 - 2.4
```

```go
note := 4
switch note {
case 1, 2, 3:
    fmt.Println( a...: "bestanden")
case 4:
    fmt.Println( a...: "nicht bestanden")
}
// Ausgabe: nicht bestanden
```

- Syntax ist ähnlich zu Java
- Auch wie beim if, man kann eine Variable zuweisen oder eine Funktion aufrufen

```go
// switch-case
switch note := "Fremde"; note {
case "Freund": fmt.Println( a...: "Hallo Freund!")
case "Fremde":
    fmt.Println( a...: "Hallo Fremde!")
default:
    fmt.Println( a...: "Hallo!")
}
// Ausgabe: Hallo Fremde!
```

```go
switch {
case 1 > 0:
    fmt.Println( a...: "Alles funktioniert einwandfrei!")
case 0 > 1:
    fmt.Println( a...: "Mit deinem PC stimmt etwas nicht...")
}
// Ausgabe (hoffentlich): Alles funktioniert einwandfrei!
```

- Go hat auch Default-Werte wie in Ruby
- Switch-Case kann auch ohne vorherige Anweisung verwendet werden
- Nach jedem Case muss aber eine Bedingung angegeben werden

SCHLEIFEN

```go
x := 0

for i := x; i < 10; i++ {
    fmt.Println( a...: "%v ", i)
}
// Ausgabe: 0 1 2 3 4 5 6 7 8 9
```

```go
for i := 0; i < 10; i++{
    fmt.Println( a...: "%v ", i)
}
// Ausgabe: 0 1 2 3 4 5 6 7 8 9
```

- Go hat nur eine Schleifenart -> for-schleife
- For-schleife kann alles, was auch eine while und do-while kann
- For-schleife -> sieht genau so aus wie in Java
- Man kann auch eine externe Variable i zuweisen, welches auch in Java möglich ist

WHILE / DO-WHILE SCHLEIFE

```
z := 0
for z < 10 {
    fmt.Println( a...: "%v ", z)
    // x++ fehlt
}
// Endlosschleife
```

```
x := 0
for x < 10 {
    fmt.Println( a...: "%v ", i)
    x++
}
// Ausgabe: 0 1 2 3 4 5 6 7 8 9
```

- **While-schleife:**
  - Hierfür wird die for-schleife verwendet
  - Man muss zusätzlich aufpassen das es kein Endlosschleife wird
- **Do-While-Schleife:**
  - Man kann auch hierfür die for-schleife verwenden

KONSTANTEN

```
const (
    HELLO = "Hello "
    WORLD = "World"
)

fmt.Print(HELLO, WORLD)
// Ausgabe: Hello World
```

```
const name = "constName"
// oder
const name = "constName"
```

- Enums gibt es nicht, jedoch ist es möglich dies durch Konstanten zu konstruieren
- Unterschied zu Java ist, Enums müssen sofort initialisiert werden
- Initialisierung gleich wäre wie in Java
- Initialisierung von mehreren Konstanten möglich

```go
func main() {
    const (
       HELLO = "Hello "
       WORLD = "World"
    )

    fmt.Print(HELLO, WORLD)
    printHello()
}

func printHello() {
    fmt.Print(HELLO) // HELLO hier nicht sichtbar
}

// Fehler
```

```go
const (
       HELLO = "Hello "
       WORLD = "World"
)
func main() {
    fmt.Print(HELLO, WORLD, " ")
    printHello()
}

func printHello() {
    fmt.Print(HELLO) // HELLO hier auch sichtbar
}

// Ausgabe: Hello World Hello
```

- Initialisierung innerhalb und außerhalb von Funktion / Methoden möglich
- Beeinflusst jedoch die Sichtbarkeit der Konstanten
- Davon abhängig ob diese in mehreren Funktionen verwendet werden oder nicht
- Wäre von Vorteil wenn man Konstanten großschreibt

SCHLÜSSELWORT IOTA

```go
const (
    EINS = iota + 1
    ZWEI
    DREI
    VIER
)

func main() {

    fmt.Print(EINS, ZWEI, DREI, VIER, " ")
    printOne()
}

func printOne() {
    fmt.Print(EINS)
}

//Ausgabe: 1 2 3 4 1
```

```go
const (
    NULL = iota
    EINS
    ZWEI
    DREI
    VIER
)

func main() {

    fmt.Print(NULL, EINS, ZWEI, DREI, VIER, " ")
    printOne()
}

func printOne() {
    fmt.Print(EINS)
}

//Ausgabe: 0 1 2 3 4 1
```

- Erste Variable muss noch initialisiert werden
- Andere Variablen wird dann der Wert zugewiesen
- Man kann auch mit einem beliebigen wert anfangen -> iota + x
- Wird dann trotzdem um eins Inkrementiert

METHODEN

```go
type Rectangle struct {  3 usages
    width, height int
}

// Area Methode mit Wert-Receiver
func (r Rectangle) Area() int {  3 usages
    return r.width * r.height
}

// Methode mit Pointer-Rectangle
func (r *Rectangle) Scale(factor int) {  1 usage
    r.width *= factor
    r.height *= factor
}

func main() {
    rect := Rectangle{width: 10, height: 5}
```

- Methode ist eine Funktion, die an einem bestimmten Datentypen gebunden ist

- Syntax:
  - Func (receiver ReceiverType) MethodenName(parameter) returnType {…}
- Receiver-Typen:
  - Zwei Arten von Receiver:
  - Wert-Receiver: eine Kopie des Wertes übergeben
  - Pointer-Receiver: tatsächlicher Speicherort des Wertes wird übergeben

VORTEILE VON METHODEN

- Hält zusammengehörige Daten und Funktion zusammen
- Verbessert Lesbarkeit und Struktur des Codes
- Ermöglicht benutzerdefinierte Typen mit Verhalten auszustatten

FUNKTIONEN

```
func functionName(parameter1 type, parameter2 type) returnType {
    // code to be executed
    return value
}
```

```
func teile(a, b int) (int, int) {
    return a / b, a % b
}

func main() {
    // Aufruf der Funktion 'teile'
    division, rest := teile(11, 2)
    fmt.Printf("division: %d, rest: %d\n", division, rest)
}
```

```
// Funktion mit variadischen Parametern
func Sum(x ...int) (n int) {
    for i := range x {//Iteriere über alle Indizes von x
        n += x[i]
    }
    return
}
func main() {
fmt.Println(Sum(1, 2, 3))  // Ausgabe: 6
}
```

- Syntax einer Funktion:
  - func -> Schlüsselwort zu Definition einer Funktion
  - functionName -> Name der Funktion
  - ParameterName -> type Parameter der Funktion
  - returnType -> Rückgabewert der Funktion
- Funktionen können mehrere Rückgabewerte besitzen
- Variadische Funktionen

STRUKTUREN

```
// Definition der Struktur
type Person struct {
    Name string
    Age  int
}
```

- Strukturen sind wie Container, die verschiedene Arten von Informationen zusammenfassen können

- Deklarieren einer Struktur:
  - Schlüsselwort -> type
  - Name der Struktur
  - Schlüsselwort -> struct
  - Definiert die Felder innerhalb der geschweiften Klammern

STRUKTUREN

```go
// Direkte Initialisierung
var p1 Person
p1.Name = "Paul"
p1.Age = 30

// Initialisierung mit einem Struct-Literal
p2 := Person{Name: "Müller", Age: 25}

// Verwendung von new()
p3 := new(Person)
p3.Name = "Charlie"
p3.Age = 40
```

- Instanziierung und Initialisierung
  - Direkter Initialisierung
  - Initialisierung mit einem Struct-Literale
  - Verwendung des new()-Schlüsselwort
- Zugriff auf Strukturmitglieder
  - Der Zugriff auf die Felder einer Struktur in Go erfolgt über den Punktoperator (*) gefolgt von dem Namen des Feldes

STRUKTUREN

```go
// Definition einer Struktur für eine Person
type Person struct {
    Vorname  string
    Nachname string
    Alter    int
}

// Methode, die auf der Struktur Person definiert ist
func (p Person) anzeigen() {
    fmt.Printf("Name: %s %s, Alter: %d\n", p.Vorname, p.Nachname, p.Alter)
}
```

```go
// Definition eines eingebetteten Typs
type ContactInfo struct {
    Email         string
    Telefonnummer string
}

// Definition eines Typs Person mit einem eingebetteten Typ
type Person struct {
    Name        string
    Alter       int
    ContactInfo // Eingebetteter Typ
}
```

- **Methoden für Strukturen:**
  - In Go können Methoden auf Strukturen definiert werden
  - Ähnlich wie bei Klassen in anderen Sprachen
- **Einbettung von Strukturen**
  - Die Einbettung von Strukturen in Go ermöglicht es
  - Eine Struktur innerhalb einer anderen Strukturen zu integrieren

POINTER

```c
int main(int argc, char** argv){
  int x=10;
  int *ptr=&x;
  printf("%d\n",x);  // Der Wert von x ist 10
  printf("%d\n",&x); // Adresse: 1201666180

  *ptr=20;
  printf("%d\n",x); // Der Wert von x ist 20
  printf("%d\n",&x); // Adresse: 1201666180
    return 0;
}
```

```go
func main() {
    x := 10
    ptr := &x
    fmt.Println(x)  //Der Wert von x ist 10
    fmt.Println(&x) //Adresse: 0xc000096068

    *ptr = 20
    fmt.Println(x)  //Der Wert von x ist 20
    fmt.Println(&x) //Adresse: 0xc000096068
}
```

- Ein Pointer ist eine Variable, die die Speicheradresse einer anderen Variable enthält

ARRAYS

```go
func main() {
    array := []int{1, 2, 3}

    fmt.Println(array) // Ausgabe: [1 2 3]

    array = append(array, 5)

    fmt.Println(array) // Ausgabe: [1 2 3 5]
}
```

```c
int main(int argc, char** argv){
  int i;
  int x[] = {1, 2, 3};
  for(int i = 0; i<3; i++){
    printf("%d ", x[i]); // Ausgabe 1 2 3
  }
    return 0;
}
```

- Ein Arrays ist eine Sammlung von Elementen eines bestimmten Typs

MAPS

- Werden verwendet, um Datenwerte in Schlüssel: Wert-Paaren zu speichern
- Jedes Element in einer Map ist ein Schüssel: Wert-Paar
- Eine Map ist eine ungeordnete und veränderbare Sammlung, die keine Duplikate zulässt
- Die Länge einer Map ist die Anzahl ihrer Elemente, sie lässt sich mit der Funktion len() ermitteln
- Der Standardwert einer Map ist nil
- Maps enthalten Verweise auf eine zugrundeliegende Hashtabelle, Go hat mehrere Möglichkeiten, Maps zu erstellen

MAPS
DEKLARATION UND INITIALISIERUNG

```go
func main() {
    var a = make(map[string]string)
    a["Marke"] = "VW"
    a["Model"] = "Golf"
    a["Jahr"] = "2024"

    fmt.Println( a...: "\t%v\n", a)
}
//Ausgabe: map[Jahr: 2024 Marke:VW model:Golf]
```

```go
func main() {
    var a = make(map[string]string)
    a["Marke"] = "VW"
    a["Model"] = "Golf"
    a["Jahr"] = "2024"

    fmt.Println(a)

    a["Jahr"] = "2024"
    a["color"] = "red"
```

- 1. Verwendung der make-funktion
  - meineMap := make(map[string] int)
- 2. Verwendung eines Map-Literals:
  - meineMap := map[string] int {...}
- 3. Hinzufügen und Aktualisieren von Elementen
  - Um ein Element in einer Mac hinzufügen oder aktualisieren, verwende die folgende Syntax

```java
import java.util.HashMap;
import java.util.Map;

public class Main {
    Run | Debug | tabnine: test | explain | document | ask
    public static void main(String[] args) {
        Map<String, Double> map = new HashMap<>();
        map.put(key:"Milch", value:0.99);
        map.put(key:"Donat", value:0.60);
        map.put(key:"Indomie", value:2.50);
        map.put(key:"Fladenbrot", value:0.75);
        System.out.println(map.keySet());
        //Ausgabe: [Milch, Donat, Indomie, Fladenbrot]
        System.out.println(map.values());
        //Ausgabe: [0.99, 0.6, 2.5, 0.75]
    }
}
```

```go
package main

import (
    "fmt"
)
tabnine: test | explain | document | ask
func main() {
    mymap := map[string]float64{
        "Milich":     0.99,
        "Donat":      0.60,
        "Indomie":    2.50,
        "Fladenbrot": 0.75,
    }
    fmt.Println(mymap)
    //map[Donat:0.6 Fladenbrot:0.75 Indomie:2.5 Milich:0.99]
}
```

- Die Unterschiede zwischen Maps in Java und Go:
  - Deklaration:
    - Java: verwendet man Klassen aus dem java.util-Paket, um Maps zu erstellen, wie HashMap, TreeMap, etc.
    - Go: ist eine eingebaute Datenstruktur und wird mit der make-funktion oder einem Map-literal erstellt
  - Typensicherheit und Generics:
    - Java: Maps sind typsicher durch Generics, die zur Compile-Zeit festgelegt werden
    - Go: Maps sind ebenfalss typsicher, jedoch ohne Generics, Typ wird bei Erstellung festgelegt und zur Compile-Zeit überprüft

FEHLERBEHANDLUNG

```go
func divide(a, b float64) (float64, error) {  1 usage
    if b == 0 {
        return 0, errors.New( text: "Division durch Null")
    }
    return a / b, nil
}

func main() {
    result, err := divide( a: 4, b: 0)
    if err != nil {
        fmt.Println( a...: "Error", err)
        return
    }
    fmt.Println( a...: "Ergebnis: ", result)
}
```

```go
type DivideByZeroError struct {
    message string
}

func (e DivideByZeroError) Error() strir
    return e.message
}
```

- Go benutzt Error Values anstatt Exceptions
- Methoden werfen keine Exceptions, sondern geben error value zurück
- Eigene Error Typen durch implementierung von Error interface und Methode error()
- `return 0, DivideByZeroError{"Division durch Null"}`

ARBEITEN MIT ERRORS

```go
func openFile(filename string) error {
    return fmt.Errorf("Datei nicht im Verzeichnis gefunden %s: %w", filename, errors.New("IOError"))
}
```

```go
if err != nil {
    unwrappedErr := errors.Unwrap(err)
    fmt.Println(unwrappedErr)
}
```

```go
if errors.Is(err, ErrFileNotFound) { … }

if errors.As(err, ErrFileNotFound) { … }
```

- Durch Wrapping mit Errorf() kann einem Fehler mehr Kontext hinzugefügt werden
- Mit der Methode Unwrap( ) kommt man wieder zum ursprünglichen Fehler
- Mit der Is( ) Methode kann man Errors vergleichen
- Die As( ) Methode ermöglicht Type Assertions

LIVE-ÜBUNG

DANKE FÜR EURE AUFMERKSAMKEIT