

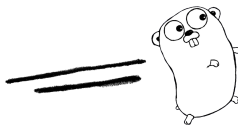
# *Die Programmiersprache Go - Eine Einführung*

## **Seminarvortrag**

Student: Adrian Helberg

Prüfer: Prof. Dr. Axel Schmolitzky

27. März 2018



# Inhalt

Geschichte

Merkmale

Error Handling

Performance

Sprachmittel

Docker

Pro & Contra

Einzelnachweise

*“Go is an open source programming language that makes it easy to build simple, reliable and efficient software.”*

(Go Website: [golang.org](https://golang.org))

*Go ist eine Open-Source-Programmiersprache, die es einfach macht, simple, zuverlässige und effiziente Software zu erstellen.*

(Eigene Übersetzung)

# Geschichte



Abbildung: Robert Griesemer, Rob Pike und Ken Thompson.

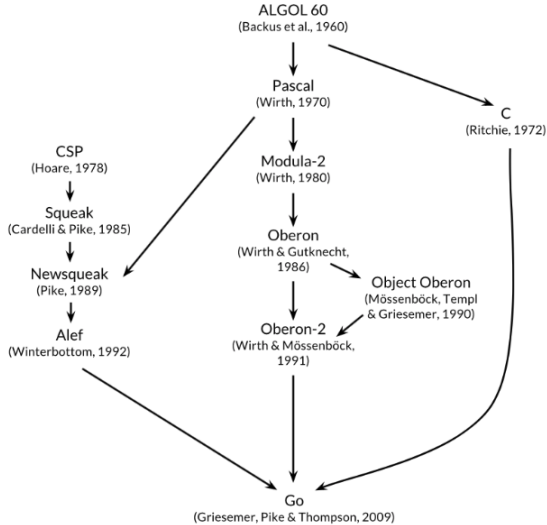
# Entwickler

- ▶ Konzipiert September 2007
- ▶ Robert Griesemer, Rob Pike und Ken Thompson
- ▶ Mitarbeiter von Google LLC ®
- ▶ Aus Frust heraus entstanden
- ▶ *“Complexity is multiplicative”* - Rob Pike

# Entwurfsphase

- ▶ Ausdrucksstarke und effiziente Kombination aus Kompilierung und Ausführung
- ▶ Ähnlichkeiten mit C
- ▶ Adaptiert gute Ideen aus einigen Programmiersprachen: Pascal, Modula-2, Oberon, Oberon-2, Alef, ...

# Entwurfsphase



# Entwurfsphase

- ▶ Vermeiden von Features, die zu komplexem, unzuverlässigem Code führen würden
- ▶ Möglichkeiten zur Nebenläufigkeit sind neu und effizient
- ▶ Datenabstraktion und Objektorientierung sind ungewohnt flexibel
- ▶ Automatische Speicherverwaltung (garbage collection)



# Veröffentlichung

- ▶ Vorgestellt November 2009
- ▶ Stable Release am 16. Februar 2018
- ▶ Berühmt als Nachfolger für nicht typisierte Scriptsprachen  
→ Verbindung aus Ausdruckskraft und Sicherheit

# Go-Community

- ▶ Open-source projekt
  - Quellcode des Compilers, Bibliotheken (libraries) und Tools sind frei verfügbar
- ▶ Aktive, weltweite Community
- ▶ Läuft auf Unix, Mac und Windows
  - Üblicherweise ohne Modifikation transportierbar

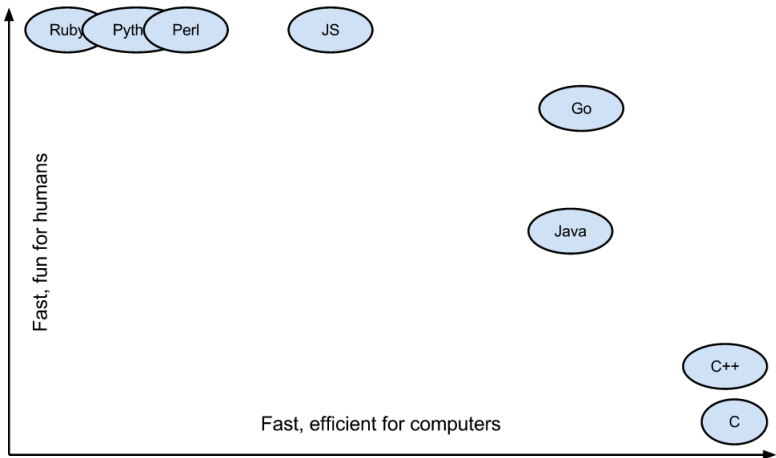


Abbildung: Go im Vergleich

# Merkmale

Abbildung: Golang "Gopher"



# Hello World!

## Go

```
package main
import "fmt"

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

## Java

```
public class HelloWorld
{
    public static void main (String[] args)
    {
        System.out.println("Hello World!");
    }
}
```

*Ausgabe:*

Hello World!

# Typsicherheit

- ▶ Statische Typisierung
- ▶ Features simulieren dynamische Typisierung
  - ▶ Keine explizite Markierung von Interface-Implementierungen (Java: *implements*)
  - ▶ Stimmt eine Methodensignatur mit der des Interfaces überein, wird diese automatisch implementiert (ähnlich: *duck-typing*)
  - ▶ Einfaches Erweitern externer Methoden (Library Funktionen)
- ▶ OOP durch *struct* und *interface* möglich

# Objektorientierung

- ▶ Typ *struct*
  - ▶ Sammlung von Variablen und Funktionen
  - ▶ Methoden nicht virtuell
  - ▶ Man spricht bei Funktionen von **Methoden** durch Zugehörigkeit des *structs*
  - ▶ Konvention: Nutzen von Zeigern bei “Settern”, *Wertkopien* bei “Gettern”

# Objektorientierung

- ▶ Typ *interface*
  - ▶ Sammlung von Funktionssignaturen
  - ▶ Objekt vom Typ *Circle* kann einer Variable vom Typ *Shape* zugeordnet werden, weil *Circle* die notwendige Funktion **Area** bietet
  - ▶ Mehrfachvererbung möglich!



# Objektorientierung

```
type Shape interface {  
    Area() float64  
}  
  
type Circle struct {  
    radius float64  
}  
  
func (c Circle) Area() float64 {  
    return math.Pi * c.radius * c.radius  
}  
  
func main() {  
    var shape Shape = Circle{2}  
    fmt.Println(shape.Area())  
}
```

*Ausgabe:*

12.566370614359172

# Objektorientierung

## ► Polymorphie

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

## ► Kein “echtes” *Duck-Typing*, da statische Typüberprüfung

# Speicherbereinigung

- ▶ Automatisch
- ▶ *Garbage Collector*
- ▶ Wird eine Variable unerreichbar, wird ihr Speicherbereich freigegeben

# Error Handling

```
func main() {  
    f, err := os.Open("filename.ext")  
  
    if err != nil {  
        log.Fatal(err)  
    }  
  
    defer f.Close()  
}
```

- ▶ Panik-System
- ▶ *defer* statt *finally*
- ▶ Benutzerdefinierte Errors: *errors.New("oh no")*

# Error Handling

```
func main() {  
    f, _ := os.Open("filename.ext")  
    defer f.Close()  
}
```

# Performance

## reverse-complement

source	secs	mem	gz	cpu	cpu load			
<u>Go</u>	<b>0.63</b>	89,120	1338	1.00	39%	43%	27%	57%
<u>Java</u>	1.02	188,428	2183	2.26	52%	53%	64%	59%

## pidigits

source	secs	mem	gz	cpu	cpu load			
<u>Go</u>	<b>2.02</b>	9,256	603	2.02	0%	0%	100%	0%
<u>Java</u>	3.13	36,984	938	3.36	4%	4%	99%	3%

Abbildung: 64-bit Ubuntu quad core

# Performance

## fannkuch-redux

source	secs	mem	gz	cpu	cpu load			
<u>Go</u>	<b>14.72</b>	1,540	900	58.65	100%	100%	99%	100%
<u>Java</u>	18.27	31,820	1282	72.06	99%	99%	98%	98%

## mandelbrot

source	secs	mem	gz	cpu	cpu load			
<u>Go</u>	<b>5.48</b>	30,912	905	21.79	99%	99%	99%	100%
<u>Java</u>	6.10	76,520	796	23.59	97%	98%	98%	96%

Abbildung: 64-bit Ubuntu quad core



Abbildung: Noch ein "Gopher"!



# Closures

## Java

```
private static Function<String, Supplier<String>> intSeq =  
x -> {  
    AtomicInteger atomicInteger = new AtomicInteger();  
    return () -> x + ": " + atomicInteger.incrementAndGet();  
}  
  
public static void main(String[] args) {  
    Supplier<String> nextInt = intSeq.apply("Test 1");  
  
    System.out.println(nextInt.get());  
    System.out.println(nextInt.get());  
}
```

*Ausgabe:*

Test 1: 1

Test 1: 2

# Closures

Go

```
func intSeq(x string) func() string {  
    i := 0  
    return func() string {  
        i++  
        return x + ": " + strconv.Itoa(i)  
    }  
}  
  
func main() {  
    nextInt := intSeq("Test 1")  
  
    fmt.Println(nextInt())  
    fmt.Println(nextInt())  
}
```

*Ausgabe:*

Test 1: 1

Test 1: 2

# Reflection

## Java

```
public static String getStringProperty(Object object ,
                                       String methodname) {
    String value = null;

    try {
        Method getter = object.getClass()
                               .getMethod(methodname, new Class[0]);

        value = (String) getter
                .invoke(object, new Object[0]);

    } catch (Exception e) {}

    return value;
}
```

# Reflection

Go

```
func getField(v *Vertex, field string) int {  
    r := reflect.ValueOf(v)  
    f := reflect.Indirect(r).FieldByName(field)  
  
    return int(f.Int())  
}
```

# Reflection

- ▶ `func ValueOf(i interface) Value`
  - ▶ Gibt ein Objekt `Value` (reflection interface) zurück, das auf den konkreten Wert initialisiert wurde, der in der Schnittstelle `i` gespeichert ist
- ▶ `func Indirect(v Value) Value`
  - ▶ Gibt den Wert zurück, auf den `v` zeigt
- ▶ `func (Value) FieldByName`
  - ▶ Gibt das *struct field* mit dem angegebenen Namen zurück
- ▶ `func (v Value) Int() int64`
  - ▶ Gibt den zugrunde liegenden Wert von `v` zurück

# Goroutines

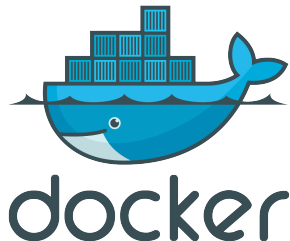
- ▶ Philosophie: *“Kommuniziere nicht, indem du Speicher teilst, sondern teile Speicher durch Kommunikation”*
- ▶ Keine Einschränkung beim Nutzen unsicherer Zugriffsmethode
- ▶ Üblich: Goroutines, Channels
  - ▶ Keine “Race Conditions”

# Goroutines

- ▶ Schlüsselwort *go*
- ▶ Kommunikation über *channels*:

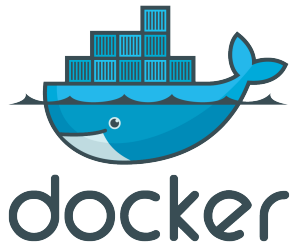
```
messages := make(chan string)
go func() { messages <- "ping" }()
```


```
msg := <- messages
fmt.Println(msg)
```



- ▶ Open-Source Apache 2.0 Lizenz
- ▶ Basiert auf *Namespaces*
- ▶ Isolierung von Anwendungen mit Containervirtualisierung
- ▶ Ressourcentrennung (Code, Laufzeitmodul, Systemwerkzeuge, Systembibliotheken, ...)
- ▶ Erstellung von Containern mit virtuellen Betriebssystemen





- ▶ Docker Hub (Online-Dienst) als Verteiler fest integriert
- ▶ Eingebaute Versionsverwaltung, angelehnt an  **git**

# Zentrale Fragestellung

„flexibel wie dynamisch getyped,  
aber mit statischer Typsicherheit?“

# Pro & Contra

## Pro

- ▶ Minimalismus
- ▶ “Eigenes” Duck-Typing
- ▶ Parallelisierung
- ▶ Aufgeräumte Syntax
- ▶ Schneller Compiler

## Contra

- ▶ Keine generische Programmierung
- ▶ nil statt Option
- ▶ Wenig grundlegende Datenstrukturen
- ▶ Keine Methodenüberladung

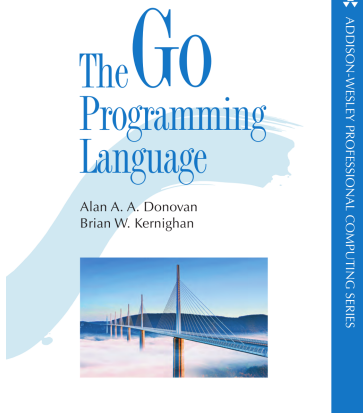
# Pro & Contra

## Pro

- ▶ Statisch gelinkte Binärdateien
- ▶ Laufzeiteigenschaften
- ▶ Integriertes Unit-Test-Framework
- ▶ Paketmanager

## Contra

- ▶ Unbefriedigende API-Dokumentation
- ▶ Teilweise umständliche APIs
- ▶ Umständliches Mocking



**Abbildung:** The Go Programming Language, von Alan A. A. Donovan und Brian W. Kernighan, 2016

# Einzelnachweise

- ▶ Performance-Grafik:  
[benchmarksgame.alioth.debian.org/u64q/go.html](http://benchmarksgame.alioth.debian.org/u64q/go.html)
- ▶ Diverse “Gopher” Grafiken: [golang.org](http://golang.org)
- ▶ Bild der Entwickler, Bild “Go im Vergleich”:  
[quora.com/Why-should-I-learn-Go-Golang-instead-of-Scala-Kotlin-Rust-Erlang-Haskell-Clojure-OCaml-etc](http://quora.com/Why-should-I-learn-Go-Golang-instead-of-Scala-Kotlin-Rust-Erlang-Haskell-Clojure-OCaml-etc)
- ▶ Bild “Docker”: [docker.com](http://docker.com)

Ende

Danke für die Aufmerksamkeit!

