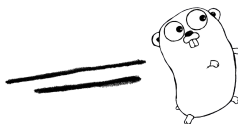


# *Die Programmiersprache Go - Eine Einführung*

## **Seminarvortrag**

Student: Adrian Helberg  
Prüfer: Prof. Dr. Axel Schmolitzky

24. März 2018



Geschichte

Merkmale und  
Sprachmittel

Docker

Zentrale  
Fragestellung

Java Vs Go

Pro & Contra

Compiler

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

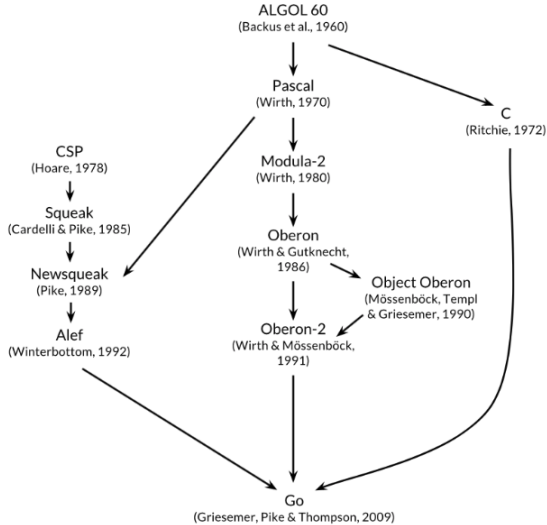


Abbildung: The Go Programming Language, Preface xii

# 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 Skriptsprachen  
→ 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 transpotrierbar

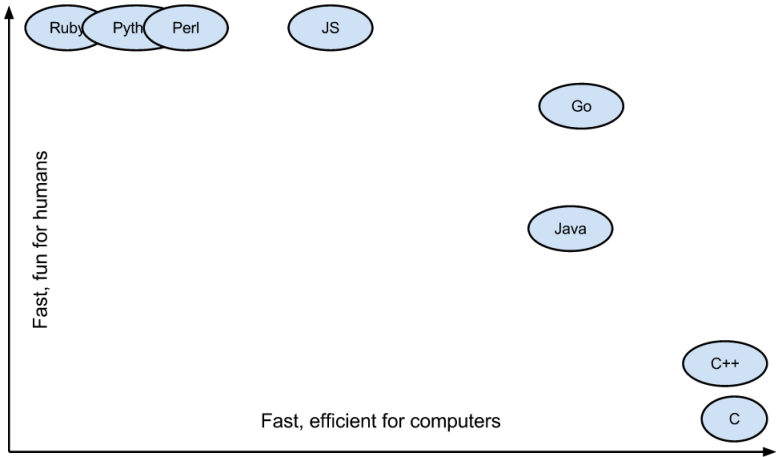


Abbildung: Go im Vergleich

# Merkmale und Sprachmittel

Abbildung: Golang “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

# Closures

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

# 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

# Typsicherheit

- ▶ Starke, 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”, *Call-by-Value* bei “Gettern”

```
func (c *Circle) Enlarge() {  
    c.radius += 1  
}
```

# Objektorientierung

- ▶ Typ *interface*
  - ▶ Virtuell
  - ▶ Objekt vom Typ *Circle* kann einer Variable vom Typ *Shape* zugeordnet werden, weil *Circle* die notwendige Funktion **Area** bietet
  - ▶ Beliebige viele Interfaces!

```
type Shape interface {  
    Area() float64  
}  
  
func main() {  
    var shape Shape = Circle{2}  
    fmt.Println(shape.Area())  
}
```

# Objektorientierung

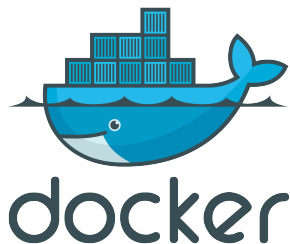
## ► Polymorphie

```
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  
}
```

# Speicherbereinigung

- ▶ Automatisch
- ▶ *Garbage Collector*
- ▶ Wird eine Variable unerreichbar, wird sie recyklet

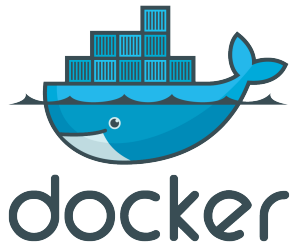
# Docker




- ▶ Open-Source Apache 2.0 Lizenz
- ▶ Basiert auf *Namespaces*
- ▶ Isolierung von Anwendungen mit Containervirtualisierung
- ▶ Ressourcentrennung (Code, Laufzeitmodul, Systemwerkzeuge, Systembibliotheken, ...)
- ▶ Offiziell von der IANA zugewiesene Portnummern 2375 für HTTP- und 2376 für HTTPS-Kommunikation
- ▶ Erstellung von Containern mit virtuellen Betriebssystemen



# Docker



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

# Zentrale Fragestellung

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

# Java Vs Go



VS

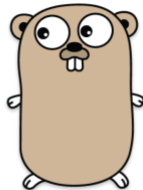


Abbildung: David gegen Goliath

# Java Vs Go

- ▶ Zeit bis zur Markteinführung
- ▶ Speicher und Geschwindigkeit
- ▶ Skalierbarkeit
- ▶ Kostenpunkt Sicherheit

# 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

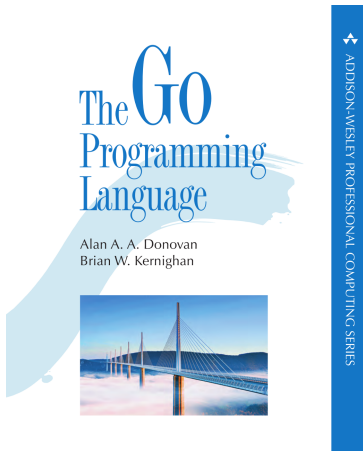
- ▶ Statisch gelinkte Binärdateien
- ▶ Laufzeiteigenschaften
- ▶ Integriertes Unit-Test-Framework
- ▶ Paketmanager

## Contra

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

# Compiler

- ▶ Gc
- ▶ Gccgo



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



Ende

Danke für die Aufmerksamkeit!

