

# Eine Einführung in go

---

Hauke Stieler

16. Januar 2018

Fachbereich Informatik der Universität Hamburg

# Agenda

- some history
- basic features
- cool web stuff
- concurrency212121
- interfaces

# Why go?

In 2007, three guys at Google were frustrated with the existing languages for writing server software:

- Compiling C++ was too slow
- Writing Java felt too verbose
- Aversion against inheritance and design patterns
- Getting concurrency right was hard

# C++

```
1 // Within large projects, popular header files
2 // get included thousands of times and hence
3 // have to be recompiled over and over again
4 #include <iostream>
5 #include <string>
6 #include <vector>
```

```
1 // Within large projects, popular header files
2 // get included thousands of times and hence
3 // have to be recompiled over and over again
4 #include <iostream>
5 #include <string>
6 #include <vector>
```

gcc copies specified file by `#include` recursively into source file. The same header file gets recompiled over and over again.

→ Rob Pike: [Public Static Void](#) at OSCON 2010

THE #1 PROGRAMMER EXCUSE  
FOR LEGITIMATELY SLACKING OFF:

"MY CODE'S COMPILING."



# Java

Let's do some Java.

Write a `public class Person` that does the following:

- store a string name
- store an int age

Simple, right?

# Java

Let's do some Java.

Write a `public class Person` that does the following:

- store a string name
- store an int age

Simple, right?

NO :(



# Java I

```
1 public class Person {
2     private String name;
3     private int age;
4
5     public Person(String name, int age) {
6         this.name = name;
7         this.age = age;
8     }
9
10    public String getName() {
11        return name;
12    }
13
14    public void setName(String name) {
15        this.name = name;
16    }
17 }
```

# Java II

```
18     public int getAge() {
19         return age;
20     }
21
22     public void setAge(int age) {
23         this.age = age;
24     }
25
26     @Override
27     public String toString() {
28         return "Person [" + "name=" + name + ", "
29 ↪      + "age=" + age + "]\n";
30     }
31
32     @Override
33     public int hashCode() {
34         final int prime = 31;
```

# Java III

```
34         int result = 1;
35         result = prime * result + age;
36         result = prime * result + ((name == null) ? 0 :
↪     name.hashCode());
37         return result;
38     }
39
40     @Override
41     public boolean equals(Object obj) {
42         if (this == obj)
43             return true;
44         if (obj == null)
45             return false;
46         if (getClass() != obj.getClass())
47             return false;
48         Person other = (Person) obj;
49         if (age != other.age)
```

## Java IV

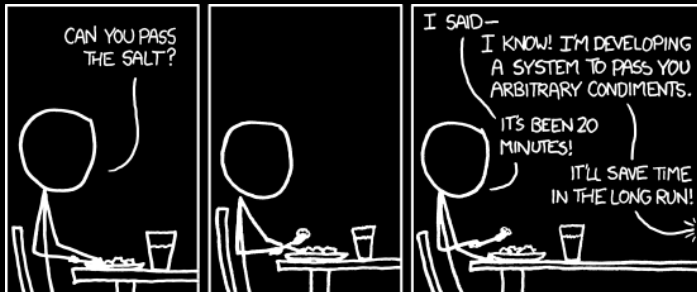
```
50         return false;
51     if (name == null) {
52         if (other.name != null)
53             return false;
54     } else if (!name.equals(other.name))
55         return false;
56     return true;
57 }
58 }
```

Initial design by 3 people with different backgrounds:

- Rob Pike (Concurrency)
- Robert Griesemer (Modules)
- Ken Thompson (Operating Systems)

All design decisions had to be agreed upon unanimously. Design team later joined by more people at Google.

- **simplicity**
- **simplicity**
- **simplicity**
- clean package model for fast compilation
- built-in concurrency based on CSP
- interfaces instead of inheritance
- no radical changes after Go 1.0



# Hello world!

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     fmt.Println("hello world")
7 }
```

- The import declaration imports entire packages
- All imported names must be qualified
- Uppercase names are visible to other packages
- Unused imports are compile-time errors!



# Hello world!

Get the go compiler:

```
$ sudo apt-get install golang-go
```

```
$ sudo pacman -S go
```

...or download from <https://golang.org/dl>

Run the code<sup>1</sup>:

```
$ go run hello.go
```

---

<sup>1</sup>The `go run` command works for single files, not always for projects

# Basics

## Keywords:

<code>break</code>	<code>default</code>	<code>func</code>	<code>interface</code>	<code>select</code>
<code>case</code>	<code>defer</code>	<code>go</code>	<code>map</code>	<code>struct</code>
<code>chan</code>	<code>else</code>	<code>goto</code>	<code>package</code>	<code>switch</code>
<code>const</code>	<code>fallthrough</code>	<code>if</code>	<code>range</code>	<code>type</code>
<code>continue</code>	<code>for</code>	<code>import</code>	<code>return</code>	<code>var</code>

## Constants:

<code>true</code>	<code>false</code>	<code>nil</code>	<code>iota</code>
-------------------	--------------------	------------------	-------------------

## Functions:

<code>new</code>	<code>len</code>	<code>complex</code>	<code>panic</code>
<code>make</code>	<code>cap</code>	<code>real</code>	<code>recover</code>
<code>close</code>	<code>append</code>	<code>imag</code>	
	<code>copy</code>		
	<code>delete</code>		

# Basics

Basic types:

```
int      int8      int16     int32     int64
uint     uint8     uint16    uint32    uint64    uintptr

float32   float64
complex64 complex128

bool      byte      rune       string    error
```

- `int` and `uint` are platform-dependent
- `byte` is the same as `uint8`
- `rune` is the same as `uint32`
- `uintptr` is large enough to hold pointers
- `error` is a special type for error handling

## Operators:

```
*    /    %    &    &^    <<    >>
+    -    ^    |
==   !=   <    <=   >    >=
&&
||
```

- only 5 precedence levels!
- `^` is both bitwise-xor (infix) and bitwise-not (prefix)
- `&^` is bitwise-andn

