THE GO PROGRAMMING LANGUAGE

PART 1

THE BASICS

- What is Go?

THE BASICS

- What is Go?
- Why is Go?

THE BASICS

- What is Go?
- Why is Go?
- Where is Go?

THE BASICS

- What is Go?
- Why is Go?
- Where is Go?
- How is Go?

WHO ARE YOU TO BE TELLING ME THESE THINGS

- Joel Auterson

WHO ARE YOU TO BE TELLING ME THESE THINGS

- Joel Auterson
- 4th year undergrad

WHO ARE YOU TO BE TELLING ME THESE THINGS

- Joel Auterson
- 4th year undergrad
- @JoelOtter

WHO ARE YOU TO BE TELLING ME THESE THINGS

- Joel Auterson
- 4th year undergrad
- @JoelOtter
- #icgolang

WHO ARE YOU TO BE TELLING ME THESE THINGS

- Joel Auterson
- 4th year undergrad
- @JoelOtter
- #icgolang
- http://gobyexample.com

WHAT IS GO?

THE BEARD SCALE

THE BEARD SCALE

THE BEARD SCALE

Dennis Ritchie

THE C PROGRAMMING LANGUAGE

THE BEARD SCALE

Dennis Ritchie

THE C PROGRAMMING LANGUAGE

THE BEARD SCALE

James Gosling

Dennis Ritchie

Java

THE C PROGRAMMING LANGUAGE

THE BEARD SCALE

Guido van Rossum

James Gosling

Dennis Ritchie

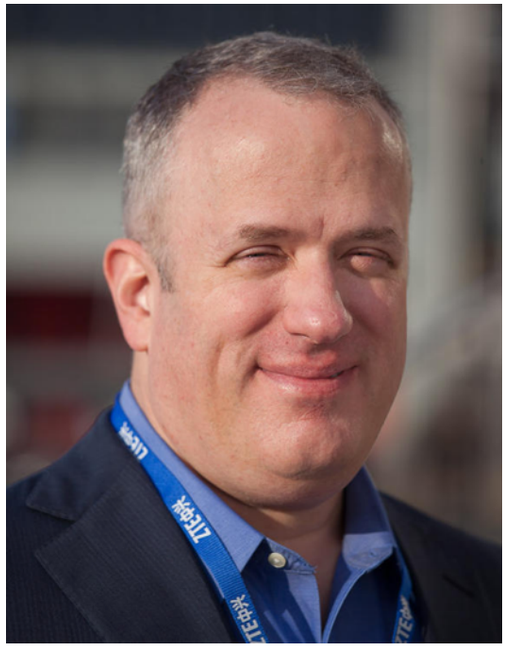
python™

Java™

THE C PROGRAMMING LANGUAGE

THE BEARD SCALE

Brendan Eich



Guido van Rossum

James Gosling

Dennis Ritchie

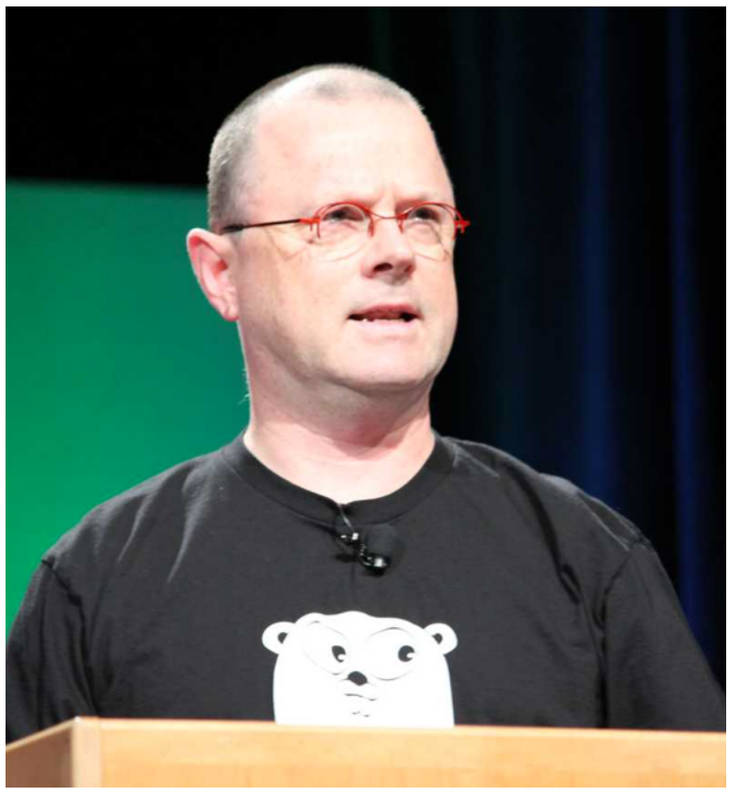
JS

python™

Java™

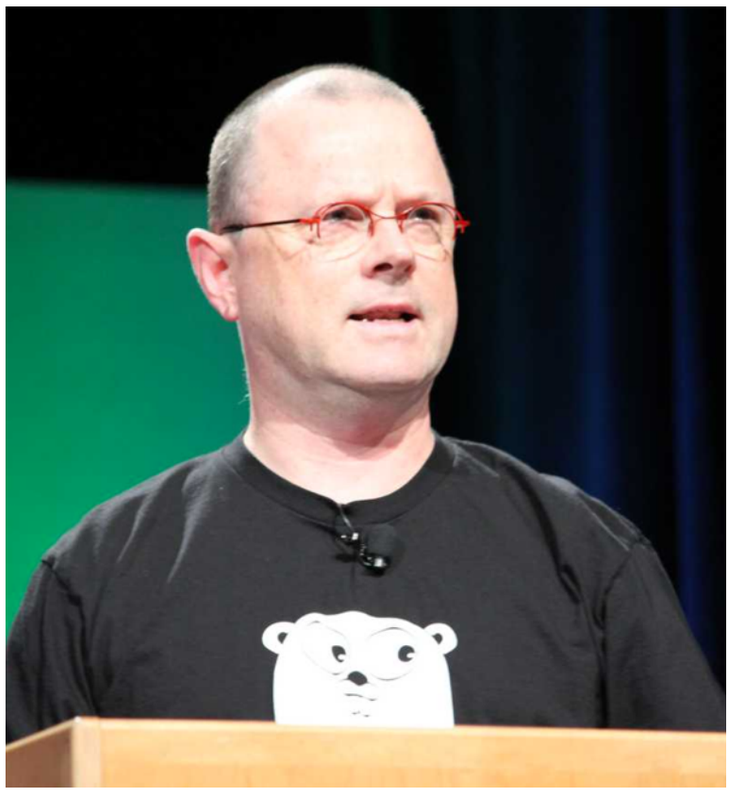
THE C PROGRAMMING LANGUAGE

GO



Rob Pike

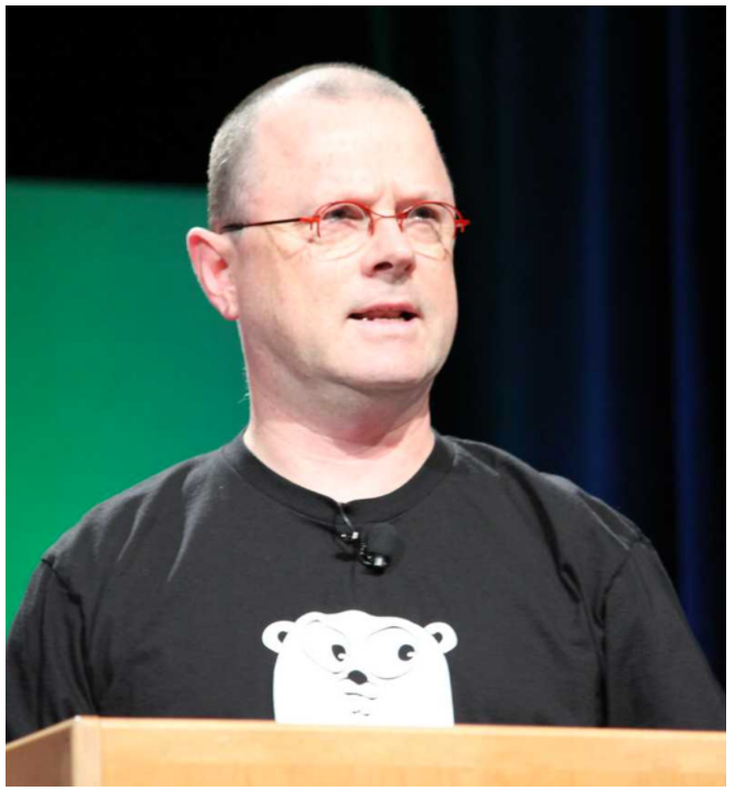
GO



Rob Pike

Robert Griesemer

GO



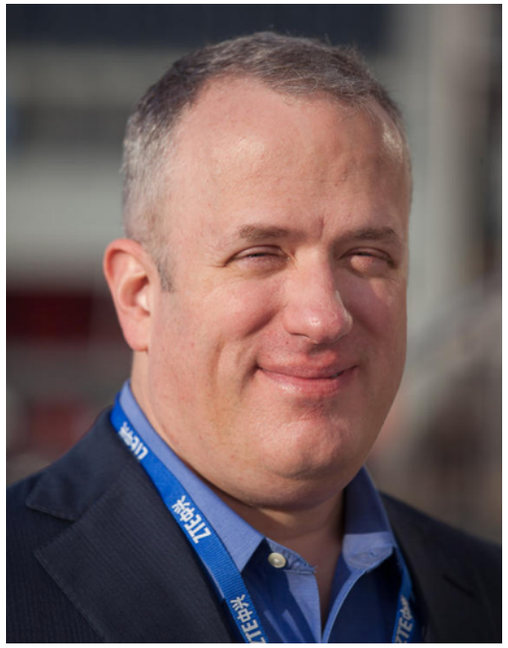
Rob Pike

Robert Griesemer

Ken Thompson

THE BEARD SCALE

Brendan Eich



Guido van Rossum

James Gosling

Dennis Ritchie

JS

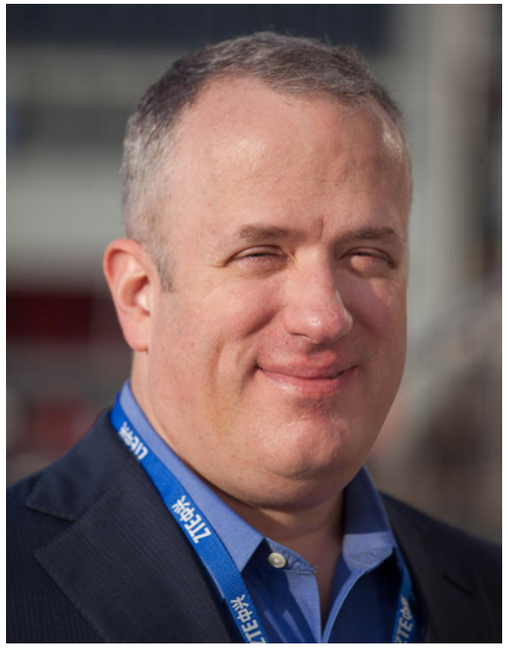
python™

Java™

THE C PROGRAMMING LANGUAGE

THE BEARD SCALE

Brendan Eich



Guido van Rossum

James Gosling

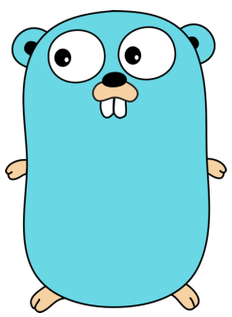
Dennis Ritchie

JS

python™

Java

THE C PROGRAMMING LANGUAGE



GO IS...

- Compiled

GO IS...

- Compiled - but garbage collected

GO IS...

- Compiled - but garbage collected
- Statically typed

GO IS...

- Compiled - but garbage collected
- Statically typed - but with type inference

GO IS...

- Compiled - but garbage collected
- Statically typed - but with type inference
- Not (really) object-oriented

GO IS...

- Compiled - but garbage collected
- Statically typed - but with type inference
- Not (really) object-oriented - but has some OO concepts

WHY IS GO?

GO IS...

- Really fast

GO IS...

- Really fast
- Cross-platform, and has cross-compilation

GO IS...

- Really fast
- Cross-platform, and has cross-compilation
- Simple

GO IS...

- Really fast
- Cross-platform, and has cross-compilation
- Simple
- Portable

GO IS...

- Really fast
- Cross-platform, and has cross-compilation
- Simple
- Portable
- Good at concurrency

WHERE IS GO?

SETTING UP

- $GOPATH environment variable

SETTING UP

- $GOPATH environment variable
  - mkdir ~/.go

SETTING UP

- $GOPATH environment variable
  - mkdir ~/.go
  - export GOPATH=/Users/joel/.go

SETTING UP

- $GOPATH environment variable
  - mkdir ~/.go
  - export GOPATH=/Users/joel/.go
  - echo “export GOPATH=/Users/joel/.go” >> ~/.bashrc

SETTING UP

- $GOPATH environment variable
  - mkdir ~/.go
  - export GOPATH=/Users/joel/.go
  - echo “export GOPATH=/Users/joel/.go” >> ~/.bashrc

```
joel@Carrot[~/.go]
-> tree -L 1 $GOPATH
/Users/joel/.go
├── bin
├── pkg
└── src
```

THE GOPATH

- src/
  - Source code for libraries
- pkg/
  - Compiled libraries
- bin/
  - Compiled commands

GO GET

- go get github.com/joelotter/termloop

GO GET

- go get github.com/joelotter/termloop

```
.gopath/
├── pkg
│   └── darwin_amd64
│       └── github.com
│           ├── joelotter
│           ├── mattn
│           └── nsf
└── src
    └── github.com
        ├── joelotter
        │   └── termloop
        ├── mattn
        │   └── go-runewidth
        └── nsf
            └── termbox-go
```

GO GET

- go get github.com/joelotter/termloop

```
.gopath/
├── pkg
│   └── darwin_amd64
│       └── github.com
│           ├── joelotter
│           ├── mattn
│           └── nsf
└── src
    └── github.com
        ├── joelotter
        │   └── termloop
        ├── mattn
        │   └── go-runewidth
        └── nsf
            └── termbox-go
```

```
.gopath/pkg/darwin_amd64/github.com
├── joelotter
│   └── termloop.a
├── mattn
│   └── go-runewidth.a
└── nsf
    └── termbox-go.a
```

HOW IS GO?

HELLO WORLD!

- Entry point is in function ‘main’.
- It needs to be in a package called ‘main’.
- “fmt” is a package providing Println and other formatting.

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

HELLO WORLD!

- Run a Go program with ‘go run’.
- Compile with ‘go build’.

VARIABLES

```
1  // Initialise a string
2  var hello string = "Hello!"
3
4  // Initialise a string, then assign after
5  var hello string
6  hello = "Hello!"
7
8  // We can assign multiple variables
9  var apple, pear int = 1, 2
10
11 // Go has type inference, which is nice
12 banana := 3
13 myName := "Joel"
14
15 // Uninitialised variables are 'zero-valued'.
16 var emptyVariable int
17 fmt.Println(emptyVariable)
```

ZERO VALUES

Type	Zero value
int	0
float	0.0
string	""
bool	FALSE
Everything else…	nil

CONDITIONALS

```
1  if x > 10 {
2      fmt.Println("Big number!")
3  } else if x > 5 {
4      fmt.Println("Medium number!")
5  } else {
6      fmt.Println("Small number!")
7  }
8
9  if y := x; y > 10 {
10     fmt.Println("Big number!")
11 }
```

- There is no ternary ‘if’ operator in Go.

LOOPING

- There is no ‘while’ in Go - only ‘for’.

```
for {
    fmt.Println("Will print once.")
    break
}

i := 0
for i < 10 {
    fmt.Println("Will print 10 times.")
    i += 1
}

for j := 0; j < 5; j++ {
    fmt.Println("Will print 5 times.")
}
```

SWITCHES

- There's no fallthrough by default - can use 'fallthrough'.
- Yes, this *is* the correct indentation...

```
1 i := 2
2 switch i {
3 case 1:
4     fmt.Println("One")
5 case 2:
6     fmt.Println("Two")
7 default:
8     fmt.Println("That's as high as I can count...")
9 }
```

ARRAYS

- Arrays are fixed-length.

```
1  // Declare array of length 5
2  var xs [5]int
3  fmt.Println(xs) // => [0 0 0 0 0]
4
5  // Addressing is pretty standard
6  xs[2] = 3
7  fmt.Println(xs[2]) // => 3
8
9  // Can declare and initialise
10 ys := [5]int{1, 2, 3, 4, 5}
11
12 // Can declare multi-dimensional
13 var zs [2][2]int
14 fmt.Println(zs) // => [[0 0] [0 0]]
```

SLICES

- Slices are variable-length.

```
1  // We use 'make' to create slices
2  words := make([]string, 3)
3  // Initially zero-valued
4  fmt.Println(words) // => [  ]
5
6  // Initialise explicitly
7  dogs := []string{"Pug", "Corgi"}
8  dogs = append(dogs, "The Hound")
9  fmt.Println(dogs) // => [Pug Corgi The Hound]
10
11 // Can copy slices too
12 alsoDogs := make([]string, len(dogs))
13 copy(alsoDogs, dogs)
14 fmt.Println(alsoDogs) // => [Pug Corgi The Hound]
15
16 fmt.Println(dogs[1:]) // => [Corgi The Hound]
17 fmt.Println(dogs[:1]) // => [Pug]
```

MAPS

- Maps are Go's key/value structure.

```go
// We use 'make' to create a new map
prices := make(map[string]float32)
prices["banana"] = 0.48
fmt.Println(prices) // => map[banana:0.48]

// Deleting from a map
prices["apple"] = 0.29
delete(prices, "banana")
fmt.Println(prices) // => map[apple:0.29]

// Explicit initialisation
ages := map[string]int{"Bob": 22, "Alice": 23}

// We can check for membership...
_, bobExists := ages["Bob"]
fmt.Println(bobExists) // => true
```

RANGE

- We can use ‘range’ to iterate over a data structure.

```
1 dogs := []string{"Pug", "Corgi"}
2 for i, dog := range dogs {
3         fmt.Printf("%v: %v\n", i, dog) // %v is default format
4 }
5
6 ages := map[string]int{"Alice": 21, "Bob": 22}
7 for person, age := range ages {
8         fmt.Printf("%v: %v\n", person, age)
9 }
```

FUNCTIONS

- We specify the input and output types.
- The function ‘sum’ takes two integers, and returns an integer.

```
1 func sum(a int, b int) int {
2     return a + b
3 }
```

FUNCTIONS

- If adjacent arguments have the same type, we can just denote it once.
- This function does exactly the same as before.

```
1 func sum(a, b int) int {
2       return a + b
3 }
```

FUNCTIONS

- We can define variadic functions, which take any number of trailing arguments.
- This function is the same, but can add any number of integers!

```
1 func sum(nums ...int) int {
2         result := 0
3         for _, num := range nums {
4                 result += num
5         }
6         return result
7 }
```

FUNCTIONS

- Functions can have multiple return values.
- (Note that fmt.Println is a variadic function! Neat.)

```
1 func divideWithRemainder(a, b int) (int, int) {
2         return a / b, a % b
3 }
4
5 func main() {
6         div, rem := divideWithRemainder(10, 3)
7         fmt.Println(div, rem) // => 3 1
8 }
```

FUNCTIONS CAN RETURN ANONYMOUS FUNCTIONS

- This lets us create **closures**.

```
1  func sequence() func() int {
2          i := 0
3          return func() int {
4                  i += 1
5                  return i
6          }
7  }
8
9  func main() {
10         seq := sequence()
11         fmt.Println(seq()) // => 1
12         fmt.Println(seq()) // => 2
13
14         seq = sequence()
15         fmt.Println(seq()) // => 1
16 }
```

STRUCTS

- Similar to structs in C.

```
1  type Person struct {
2          name string
3          age  int
4  }
5
6  func main() {
7          bob := Person{"Bob", 22}
8          alice := Person{name: "Alice", age: 21}
9          fmt.Println(bob.age)     // => 22
10         fmt.Println(alice.name) // => Alice
11 }
```

WE CAN DEFINE METHODS ON STRUCTS

```
1  type Rectangle struct {
2          width, height int
3  }
4
5  func (r *Rectangle) Area() int {
6          return r.width * r.height
7  }
8
9  func (r *Rectangle) Perimeter() int {
10         return (2 * r.width) + (2 * r.height)
11 }
12
13 func main() {
14         r := Rectangle{width: 2, height: 3}
15         fmt.Println(r.Area())       // => 6
16         fmt.Println(r.Perimeter()) // => 10
17 }
```

INTERFACES

- Collections of method signatures.
- To make sure a type implements Shape, we just need to make sure it has Area and Perimeter methods.

```
 1 type Shape interface {
 2     Area() int
 3     Perimeter() int
 4 }
 5
 6 func main() {
 7     shapes := []Shape{Rectangle{2, 3}}
 8     for _, shape := range shapes {
 9         fmt.Println(shape.Area())
10     }
11 }
```

ERRORS

- Go doesn't have 'exceptions' to catch. Instead, we return an additional value for the error.

```
1  import "errors"
2
3  func divideTenBy(n int) (int, error) {
4          if n == 0 {
5                  return -1, errors.New("Can't divide by zero")
6          }
7          return 10 / n, nil
8  }
9
10 func main() {
11         if n, err := divideTenBy(0); err != nil {
12                 fmt.Println("Got error:", err)
13         } else {
14                 fmt.Println("Got result:", n)
15         }
16 }
```

NEXT WEEK...

NEXT WEEK...

- Concurrency!

NEXT WEEK...

- Concurrency!
- Panic and defer

NEXT WEEK...

- Concurrency!
- Panic and defer
- OO in Go with object composition

NEXT WEEK...

- Concurrency!
- Panic and defer
- OO in Go with object composition
- A simple game (if there's time...)

NEXT WEEK...

- Concurrency!
- Panic and defer
- OO in Go with object composition
- A simple game (if there's time...)
- Anything else?

THANKS!