

Golang Training 1 – Basics, Syntax, Data Structures

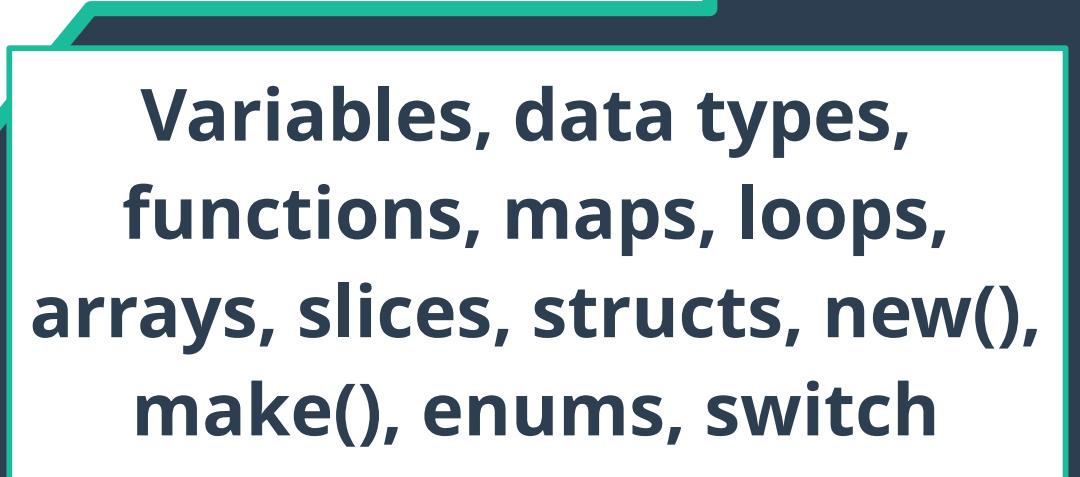
Calvin Lei-Cramer
May 2025



Training format?



What are we covering today?



**Variables, data types,
functions, maps, loops,
arrays, slices, structs, new(),
make(), enums, switch**

What is Golang Exactly?

- 2009 started by Google engineers Robert Griesemer, Rob Pike, Ken Thompson (yes that Ken)
- Simple syntax, small language specification (opposite of C++ & Swift)
- Compiled → \$ go build ; ./my-module
- Statically typed. No automatic type conversions / promotions.
- Garbage collected. No need to worry about lifetime or freeing objects.
- No OOP. Functional (interfaces, structs).
- Basically C but modern and more strict, plus interfaces and more nice things

Language Versions

- 1.24 (2025 Feb, we're using this) – generic type aliases
- 1.23
- 1.22
- 1.21 – min, max, clear Extremely conservative changes
- 1.20
- 1.19
- 1.18 (2022 March) – Generics, any (biggest change)

How to Install

- 1. Download from <https://go.dev/dl/>

```
1 sudo rm -rf /usr/local/go
2 sudo tar -C /usr/local -xzf go1.24.3.linux-amd64.tar.gz # may be different version
3 # reload terminal or shell environment
4 go version
5 # go version go1.24.3 linux/amd64
```

Hello World

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

Main.go

```
$ go run .
Hello Golang
```

Follow along locally,
or use <https://play.golang.com/>

Variables

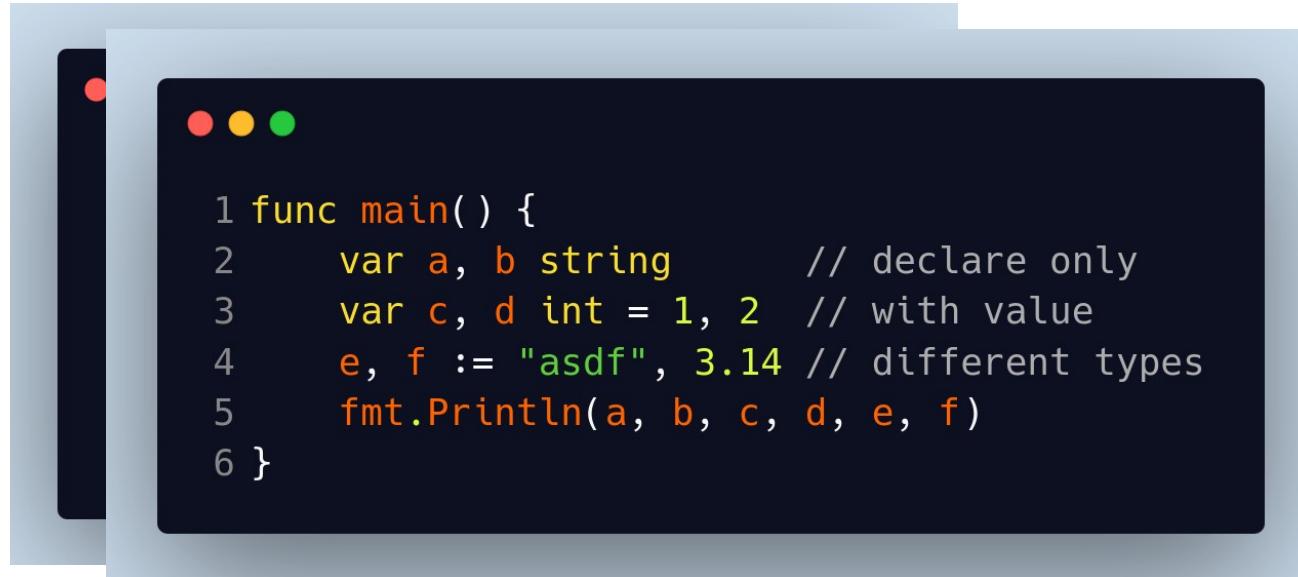
- Use = to specify type and for globals
- Re-declaring variables is error
- Unused variables is error

The compiler is your friend

```
1 func main() {  
2     var a = 5  
3     a = 10      // write  
4     var a = 100 // no redeclarations  
5 }
```

Variables (2)

- Use `:=` for brevity, let Golang do type inference like C++ `auto`
- Multiple declarations at once



A screenshot of a terminal window with a dark background and light-colored text. The window has three colored window control buttons (red, yellow, green) at the top. The terminal displays the following Go code:

```
1 func main() {
2     var a, b string      // declare only
3     var c, d int = 1, 2 // with value
4     e, f := "asdf", 3.14 // different types
5     fmt.Println(a, b, c, d, e, f)
6 }
```

Short declarations

- “Short” declaration scopes variables to if
- Also can use with for (init statement) and switch

declaration

```
1 func main() {  
2     if result := someFunc(); result == 100 {  
3         ...  
4     }  
5     // result not available after if  
6     fmt.Println(result) // error  
7 }  
8
```

condition

Data Types

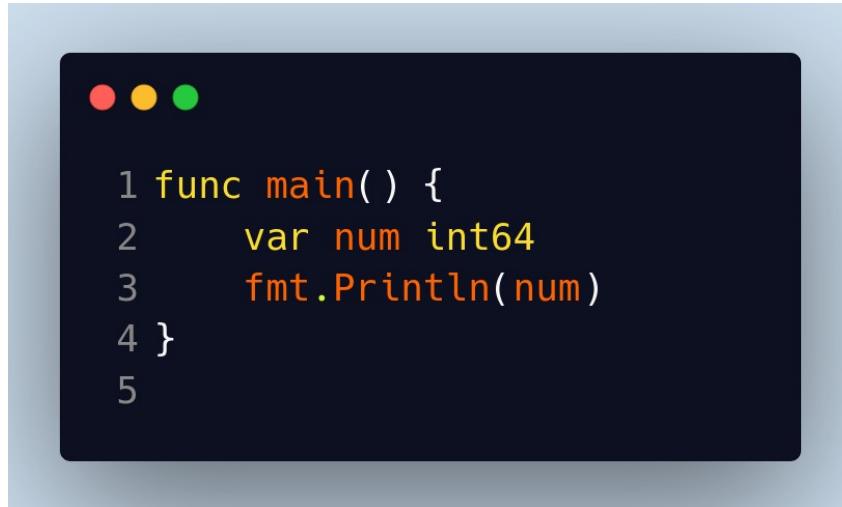
- Very few

```
● ● ●  
1 bool  
2 string  
3 int   int8  int16  int32  int64  
4 uint  uint8 uint16 uint32 uint64  
5 uintptr  
6 byte // uint8  
7 rune // int32  
8 float32 float64  
9 complex64 complex128  
10  
11 struct{...}  
12 interface{} // any  
13 array // [N]T  
14 slice // []T  
15 map[keyT]valueT  
16 // pointer types
```

Zero Values

- **What will be printed?**

- Integers → 0
- Floats → 0.0
- Boolean → false
- Strings → ""
- Interface, slice, channel, map, pointers, functions → nil
- Array, struct → zero value for each member



A screenshot of a terminal window on a Mac OS X system, indicated by the red, yellow, and green dots in the top-left corner. The terminal shows the following Go code:

```
1 func main() {
2     var num int64
3     fmt.Println(num)
4 }
5
```

Converting Between Data Types

- **T(v) converts v to T**
- **strconv package Atoi() Itoa() for strings** **integer**

```
1 func main() {  
2     var i int = 42  
3     var f float32 = float32(i)  
4     fmt.Println(f)  
5 }
```

What happens if we try to break things?

- Integer overflow

```
$ go run .
0xFE
0xFF
0x0
```

Wrap!



```
● ● ●

1 func main() {
2     var num uint8 = 0b1111_1110
3     fmt.Printf("0x%X\n", num)
4     num += 1
5     fmt.Printf("0x%X\n", num)
6     num += 1
7     fmt.Printf("0x%X\n", num)
8 }
```

What happens if we try to break things?

```
● ● ●  
5 func main() {  
6     var num uint8 = 0b1111_1110  
7     var div uint8 = 0  
8     fmt.Printf("0x%X\n", num/div)  
9 }
```

- Divide by zero
- Learn about panics later

```
$ go run .  
panic: runtime error: integer divide by zero  
  
goroutine 1 [running]:  
main.main()  
    /home/ccramer-loc/repos/golang-training-private/main.go:8 +0x9
```

Functions (not methods)

- Simple
- Multiple return
- Multiple return collect whole response

```
./main.go:12:11: assignment mismatch: 1 variable
but giveMeStuff returns 3 values
●●●
```

1 func giveMeStuff() (int, float32, string) {
2 return 42, 3.14, "cat"
3 }
NOT LIKE PYTHON TUPLE

```
●●●  
1 func giveMeStuff() (int, float32, string) {  
2     return 42, 3.14, "cat"  
3 }  
4  
5 func main() {  
6     stuff := giveMeStuff()  
7     fmt.Println(stuff)  
8 }
```

Functions more

- **Named / naked return
(important for later)**
- **Variables declared and
initialize to zero value**
- **REQUIRED to add “naked
return”**

```
1 func giveMeStuff() (code int, radius float32, msg string) {  
2     radius = 3.14  
3     msg = "small step for man"  
4     return  
5 }  
6  
7 func main() {  
8     a, b, c := giveMeStuff()  
9     fmt.Println(a, b, c)  
10 }
```

Maps

- Create, Add, Check key exists, Delete key, clear

Arbitrary scoping

```
1 func main() {
2     m := map[string]int{"width": 100}
3     m["height"] = 200
4     {
5         val, exists := m["width"]
6         fmt.Println(val, exists) // 100, true
7     }
8     {
9         val, exists := m["not exist"]
10        fmt.Println(val, exists) // 0, false
11    }
12    delete(m, "width")
13    delete(m, "not exist") // OK
14    clear(m)
15 }
```

Maps – be careful!

```
1 func main() {  
2     m := map[int]float64{}  
3     test := m[400]  
4     fmt.Println(test)  
5 }
```

- Prints 0!

What happens here?
Compiler error?
Runtime panic?

Helpful debugging

- Print (%v, %#v)



```
1 fmt.Printf("%v\n", myMap)
2 fmt.Printf("%#v\n", myMap)
```

map[inst:{500} 40:foo 3.14:0xc000098040]

map[interface {}]interface {}{"inst":main.MyStruct{member:500}, 40:"foo", 3.14:
(*int)(0xc000098040)}

Arrays and Slices

- **Array – fixed sized sequence of elements of the same type**
- **Size is included in type** → [10]int != [20]int

```
1 func main() {
2     var myArr [3]any = [3]any{1, "foo", 3.14}
3     fmt.Println(myArr)
4 }
```

Arrays and Slices

- Slice – range of contiguous elements, pointing to underlying array that actually holds the data
- Slice is “view” on an array
- Forget about arrays, just use slices.
- Slices offer dynamically-sized array features like pythons List

No size means slice

```
1 func main() {  
2     var mySlice []any = []any{1, "foo", 3.14}  
3     fmt.Println(mySlice)  
4 }
```

Let's Cause Errors!

- Index out of bounds, what happens?

```
● ● ●  
1 func main() {  
2     arr := [3]int{1, 2, 3}  
3     idx := 10  
4     _ = arr[idx]  
5 }
```

Out of bounds!

```
$ go run .  
panic: runtime error: index out of range [10] with length 3  
  
goroutine 1 [running]:  
main.main()  
    /home/ccramer-loc/repos/golang-training-private/main.go:6 +0x17  
exit status 2
```

Create Slices

- Different ways to construct slices → direct, make(), from array

```
● ● ●  
1 func main() {  
2     slice1 := []any{1, "foo", 3.14} // specific elements  
3     slice2 := make([]int, 1000) // certain size, zero value  
4     arr := [3]int{1, 2, 3}  
5     slice3 := arr[:] // slice from array  
6 }
```

Slice Expression

- Slice expression [:] syntax:
- `foo[low:high]`
- `foo[:high]`
- `foo[low:]`
- Similar to python but without negative indexing

```
 1 func main() {  
 2     slice := []any{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}  
 3     fmt.Println(slice[2:5])  
 4     fmt.Println(slice[5:])  
 5     fmt.Println(slice[:4])  
 6     fmt.Println(slice[:])  
 7 }
```

```
$ go run .  
[2 3 4]  
[5 6 7 8 9]  
[0 1 2 3]  
[0 1 2 3 4 5 6 7 8 9]
```

Slice Operations

- **Append**

Multiple items at once

```
1 func main() {  
2     s := []int{}  
3     s = append(s, 5)  
4     s = append(s, 1, 2, 3, 4, 5)  
5     append(s, 10) // error unused return value  
6 }
```



Append is a global function, not a function on slice type

Slice Operations

- Concatenate two slices
- **func append(slice []Type, elems ...Type) []Type**
- Variadic functions (later)

```
1 func main() {  
2     s1 := []int{1, 2, 3}  
3     s2 := []int{4, 5, 6}  
4     whole := append(s1, s2...)  
5     fmt.Println(whole)  
6 }
```



Slice Operations

- Remove elements

```
● ● ●  
1 func main() {  
2     arr := [6]int{0, 1, 2, 3, 4, 5}  
3     fmt.Println(arr)  
4     s := arr[:]  
5     idx := 2  
6     sNew := append(s[:idx], s[idx+1:]...)  
7     fmt.Println(arr)  
8     fmt.Println(sNew)  
9 }
```

```
$ go run .  
[0 1 2 3 4 5]  
[0 1 3 4 5 5]  
[0 1 3 4 5]
```

Underlying array

Slice Operations

- Insert multiple elements (start, middle)

```
● ● ●  
1 func main() {  
2     s := []int{}  
3     s = append([]int{1, 5, 6}, s...)  
4     fmt.Println(s)  
5     insertAtIdx := 1  
6     toInsert := []int{2, 3, 4}  
7     s = append(s[:insertAtIdx], append(toInsert, s[insertAtIdx:]...)...)  
8     fmt.Println(s)  
9 }
```

```
$ go run .  
[1 5 6]  
[1 2 3 4 5 6]
```

- Whoa! Why need multiple append()?

Loops

```
● ● ●  
1 func main() {  
2     slice := make([]int, 1000)  
3     // Init, condition, update  
4     for i := 0; i < len(slice); i++ {  
5         slice[i] = i % 10  
6     }  
7     // While  
8     for len(slice) > 50 {  
9         slice = slice[1:]  
10    }  
11    // Forever  
12    for {  
13        if slice[0]%2 == 0 {  
14            break  
15        }  
16        slice = slice[1:]  
17    }  
18 }
```

Loops

- **Optional init and update**

```
1 func main() {  
2     slice := make([]int, 1000)  
3     idx := 0  
4     idx = complexLogic(idx, slice)  
5     // Condition, update  
6     for ; idx < len(slice); idx++ {  
7         slice[idx] = idx % 10  
8     }  
9     // Init, condition  
10    for idx2 := 0; idx2 < len(slice) ; {  
11        slice[idx2] = manipulateElement(slice[idx])  
12        // Complex idx2 update logic...  
13        idx2 = (slice[idx2] % 10) + 2  
14        idx2 = idx2 * idx2  
15        idx2 = max(0, min(len(slice), idx2))  
16    }  
17 }
```

Loops Range - Iterator

- **Range(val) → slice, array, map, string, channel**

```
● ● ●  
1 func main() {  
2     slice := []int{1, 2, 3, 4}  
3     arr := [...]int{1, 2, 3, 4}  
4     myMap := map[string]int{"a": 2, "b": 4, "c": 6, "d": 8}  
5     str := "abcdef"  
6  
7     for idx, el := range slice { ... }  
8     for idx := range slice { ... }  
9     for idx, el := range arr { ... }  
10    for idx := range arr { ... }  
11    for key, val := range myMap { ... }  
12    for key := range myMap { ... }  
13    for idx, char := range str { ... }  
14    for idx := range str { ... }  
15 }
```

- **Different meaning based on datatype**
- **Optional second parameter**

Structs

- Structs are simply a collection of fields. A bunch of data.

```
● ● ●  
1 type Jeff struct {  
2     x, y int  
3     u     float32  
4     _     float32 // pad  
5     Arr  *[]int  // poi  
6     Func func()  
7     Part OtherStruct  
8 }
```

How to instantiate structs?

```
● ● ●  
1 func main() {  
2     jeff1 := Jeff{} // Zero init  
3     jeff2 := Jeff{y: 2, x: 1, Part: OtherStruct{num: 42}}  
4     // Need to specify all fields IN ORDER when not using {K: V}  
5     jeff3 := Jeff{5, 6, 3.14, &[]int{1, 2, 3}, SomeFunc, OtherStruct{}}  
6 }
```

“jeff2” method is best

`new()` vs `make()`

- `func new(Type) *Type`

-

```
 1 func main() {  
 2     var jeff *Jeff = new(Jeff) // zero init  
 3     jeff.x = 42  
 4     fmt.Println(jeff) // &{42 0 0 <nil> <nil> {0}}  
 5  
 6     var test = new([]int)  
 7     *test = append(*test, 1, 2, 3)  
 8     fmt.Println(test) // &[1 2 3]  
 9 }
```

Enums

- Enums are just constants → usually integer, string

```
1 type Command int
2
3 const (
4     CommandForward    Command = 0
5     CommandBackwards Command = 1
6     CommandTurnLeft   Command = 2
7     CommandTurnRight  Command = 3
8 )
```

Enums

- Iota – index of the expression in the const declaration, starting at 0

```
● ● ●  
1 type Command int  
2  
3 const (  
4     CommandForward   Command = 0  
5     CommandBackwards Command = 1  
6     CommandTurnLeft  Command = 2  
7     CommandTurnRight Command = iota // 3  
8 )
```

Enums

- Const declarations only need type once

```
1 type Command int
2
3 const (
4     CommandForward    Command = iota
5     CommandBackwards      = iota
6     CommandTurnLeft       = iota
7     CommandTurnRight      = iota
8 )
```

Enums

- **Implicit repetition → empty expression repeats (copy pastes) last non-empty expression**

```
1 type Command int
2
3 const (
4     CommandForward    Command = iota // 0
5     CommandBackwards           // 1
6     CommandTurnLeft            // 2
7     CommandTurnRight           // 3
8 )
```

Enums

- Number of enums idiom (last element)

```
1 type Command int
2
3 const (
4     CommandForward    Command = iota
5     CommandBackwards
6     CommandTurnLeft
7     CommandTurnRight
8     NumberOfCommands
9 )
```

Enums

- What value does G have?

```
1 type Letter int
2
3 const (
4     A Letter      = iota // 0
5     B                  // 1
6     C                  // 2
7     D                  // 3
8     E = 1 << iota    // 1 << 4 = 16
9     F                  // 32
10    G                  // 64
11 )
```

Switch statements

- Simple
- Fallthrough opt-in
- Break implied, but can be used manually
- Default optional
- Non-exhaustive (unlike Rust)

```
1 type CardinalDir int
2
3 const (
4     North CardinalDir = iota
5     East
6     South
7     West
8 )
9
10 func doSomething(dir CardinalDir) string {
11     switch(dir) {
12     case North:
13         return "Success!"
14     case South:
15         fallthrough
16     case East, West:
17         panic("Only north allowed")
18     default:
19         panic("Unknown direction")
20     }
21 }
```

Switch statements

- Short assignment statement

```
1 func main() {
2     switch x := calculateArea(); x {
3     case 100:
4         fmt.Println("Area is 100!")
5     case 200:
6         fmt.Println("Area is 200!")
7     default:
8         panic("Invalid area!")
9     }
10 }
```

Switch statements

```
1 func main() {
2     x := calculateArea()
3     switch { // implicit 'true' expression
4     case x < 0:
5         panic("Negative area bad!")
6     case x > MAX_AREA:
7         panic("Too big of area!")
8     default:
9         fmt.Println(x)
10    }
11 }
```

- Naked switch → switch true
- Complicated case expressions

Switch statements

- Type switch → check underlying type of variable

```
1 func foo(something any) any {  
2     switch val := something.(type) {  
3     case int:  
4         return val + 40 // val is int  
5     case float32:  
6         return val / 4.1134 // val is float32  
7     case string:  
8         return "hello " + val + " there!" // val is string  
9     default:  
10        panic(fmt.Sprintf("Unhandled type %T", something))  
11    }  
12 }
```

```
switch something.(type) { ... }
```

Practice!

- <https://github.com/calvincramer/golang-training-1>



Golang Training 1 – Basics, Syntax, Data Structures

Calvin Lei-Cramer
May 2025



Training format?

2

Our training format will be four lessons.

After each lesson I will release exercises you can complete to practice and reinforce the ideas.

The exercises will be released as a github repo which will also include these slides.

After a certain amount of time I will release reference solutions.

**What are we covering
today?**

**Variables, data types,
functions, maps, loops,
arrays, slices, structs, new(),
make(), enums, switch**

3

What is Golang Exactly?

- 2009 started by Google engineers Robert Griesemer, Rob Pike, Ken Thompson (yes that Ken)
- Simple syntax, small language specification (opposite of C++ & Swift)
- Compiled → \$ go build ; ./my-module
- Statically typed. No automatic type conversions / promotions.
- Garbage collected. No need to worry about lifetime or freeing objects.
- No OOP. Functional (interfaces, structs).
- Basically C but modern and more strict, plus interfaces and more nice things

4

Ken Thompson original author of UNIX

Statically typed → all types known at compile time.

Golang is strict with types. No automatic conversion of types done like in C.

Language Versions

- 1.24 (2025 Feb, we're using this) - generic type aliases
- 1.23
- 1.22
- 1.21 - min, max, clear Extremely conservative changes
- 1.20
- 1.19
- 1.18 (2022 March) - Generics, any (biggest change)

5

Golang is very hesitant to add any changes.
Things move very slowly.

How to Install

- 1. Download from <https://go.dev/dl/>

```
● ● ●  
1 sudo rm -rf /usr/local/go  
2 sudo tar -C /usr/local -xzf go1.24.3.linux-amd64.tar.gz # may be different version  
3 # reload terminal or shell environment  
4 go version  
5 # go version go1.24.3 linux/amd64
```

6

Hello World

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

Main.go

```
$ go run .  
Hello Golang
```

Follow along locally,
or use <https://play.golang.com/>

Variables

- Use = to specify type and for globals
- Re-declaring variables is error
- Unused variables is error

The compiler is your friend

```
● ● ●
1 func main() {
2     var a = 5
3     a = 10      // write
4     var a = 100 // no redeclarations
5 }
```

Variables (2)

- Use `:=` for brevity, let Golang do type inference like C++ `auto`
- Multiple declarations at once

```
 1 func main() {  
 2     var a, b string      // declare only  
 3     var c, d int = 1, 2  // with value  
 4     e, f := "asdf", 3.14 // different types  
 5     fmt.Println(a, b, c, d, e, f)  
 6 }
```

Short declarations

- “Short” declaration scopes variables to if
- Also can use with for (init statement) and switch

declaration

condition

```
● ● ●  
1 func main() {  
2     if result := someFunc(); result == 100 {  
3         ...  
4     }  
5     // result not available after if  
6     fmt.Println(result) // error  
7 }  
8
```

10

Data Types

- Very few

• • •

```
1 bool
2 string
3 int  int8  int16  int32  int64
4 uint uint8  uint16  uint32  uint64
5 uintptr
6 byte // uint8
7 rune // int32
8 float32 float64
9 complex64 complex128
10
11 struct{...}
12 interface{} // any
13 array // [N]T
14 slice // []T
15 map[keyT]valueT
16 // pointer types
```

Zero Values

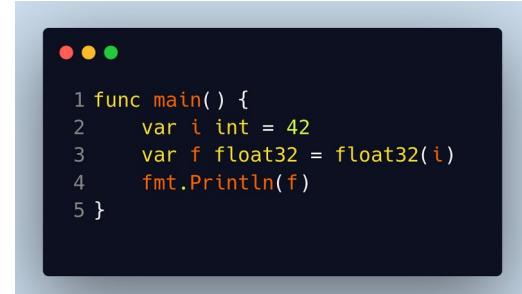
- **What will be printed?**

- Integers → 0
- Floats → 0.0
- Boolean → false
- Strings → ""
- Interface, slice, channel, map, pointers, functions → nil
- Array, struct → zero value for each member

```
● ● ●  
1 func main() {  
2     var num int64  
3     fmt.Println(num)  
4 }  
5
```

Converting Between Data Types

- **T(v)** converts v to T
- **strconv package Atoi() Itoa()** for strings integer



```
● ● ●

1 func main() {
2     var i int = 42
3     var f float32 = float32(i)
4     fmt.Println(f)
5 }
```

13

What happens if we try to break things?

- Integer overflow

```
$ go run .
0xFE
0xFF
0x0
```

Wrap!

```
● ● ●
1 func main() {
2     var num uint8 = 0b1111_1110
3     fmt.Printf("0x%X\n", num)
4     num += 1
5     fmt.Printf("0x%X\n", num)
6     num += 1
7     fmt.Printf("0x%X\n", num)
8 }
```

What happens if we try to break things?

```
● ● ●  
5 func main() {  
6     var num uint8 = 0b1111_1110  
7     var div uint8 = 0  
8     fmt.Printf("0x%X\n", num/div)  
9 }
```

- Divide by zero
- Learn about panics later

```
$ go run _  
panic: runtime error: integer divide by zero  
  
goroutine 1 [running]:  
main.main()  
/home/ccramer-loc/repos/golang-training-private/main.go:8 +0x9
```

Functions (not methods)

- Simple
- Multiple return
- Multiple return collect whole response

The terminal window shows the following Go code:

```
./main.go:12:11: assignment mismatch: 1 variable
but giveMeStuff returns 3 values
1 func giveMeStuff() (int, float32, string) {
2     return 42, 3.14, "cat"
3 }
4
5 func main() {
6     stuff := giveMeStuff()
7     fmt.Println(stuff)
8 }
```

NOT LIKE PYTHON TUPLE

A red box highlights the error message: "assignment mismatch: 1 variable but giveMeStuff returns 3 values". A green box highlights the line "NOT LIKE PYTHON TUPLE".

16

Functions more

- **Named / naked return
(important for later)**
- **Variables declared and
initialize to zero value**
- **REQUIRED to add “naked
return”**

```
●●●  
1 func giveMeStuff() (code int, radius float32, msg string) {  
2     radius = 3.14  
3     msg = "small step for man"  
4     return  
5 }  
6  
7 func main() {  
8     a, b, c := giveMeStuff()  
9     fmt.Println(a, b, c)  
10 }
```

Maps

- Create, Add, Check key exists, Delete key, clear

Arbitrary scoping

```
1 func main() {
2     m := map[string]int{"width": 100}
3     m["height"] = 200
4     {
5         val, exists := m["width"]
6         fmt.Println(val, exists) // 100, true
7     }
8     {
9         val, exists := m["not exist"]
10        fmt.Println(val, exists) // 0, false
11    }
12    delete(m, "width")
13    delete(m, "not exist") // OK
14    clear(m)
15 }
```

Maps – be careful!

```
● ● ●  
1 func main() {  
2     m := map[int]float64{}  
3     test := m[400]  
4     fmt.Println(test)  
5 }
```

- Prints 0!

What happens here?
Compiler error?
Runtime panic?

Helpful debugging

- Print (%v, %#v)



```
1 fmt.Printf("%v\n", myMap)
2 fmt.Printf("%#v\n", myMap)
```

```
map[inst:{500} 40:foo 3.14:0xc000098040]
map[interface {}]interface {}{"inst":main.MyStruct{member:500}, 40:"foo", 3.14:
(*int)(0xc000098040)}
```

Arrays and Slices

- Array - fixed sized sequence of elements of the same type
- Size is included in type → [10]int != [20]int

```
● ● ●  
1 func main() {  
2     var myArr [3]any = [3]any{1, "foo", 3.14}  
3     fmt.Println(myArr)  
4 }
```

21

Can't make functions which handles arrays of any size.

Arrays and Slices

- Slice – range of contiguous elements, pointing to underlying array that actually holds the data
- Slice is “view” on an array
- Forget about arrays, just use slices.
- Slices offer dynamically-sized array features like pythons List

```
 1 func main() {  
 2     var mySlice []any = []any{1, "foo", 3.14}  
 3     fmt.Println(mySlice)  
 4 }
```

No size means slice

Let's Cause Errors!

- Index out of bounds, what happens?

...

```
1 func main() {  
2     arr := [3]int{1, 2, 3}  
3     idx := 10  
4     _ = arr[idx] ←
```

Out of bounds!

```
$ go run _  
panic: runtime error: index out of range [10] with length 3  
  
goroutine 1 [running]:  
main.main()  
    /home/ccramer-loc/repos/golang-training-private/main.go:6 +0x17  
exit status 2
```

23

Both array and slice panic

Create Slices

- Different ways to construct slices → direct, make(), from array

```
● ● ●

1 func main() {
2     slice1 := []any{1, "foo", 3.14} // specific elements
3     slice2 := make([]int, 1000) // certain size, zero value
4     arr := [3]int{1, 2, 3}
5     slice3 := arr[:] // slice from array
6 }
```

Slice Expression

- Slice expression [:] syntax:
- `foo[low:high]`
- `foo[:high]`
- `foo[low:]`
- Similar to python but without negative indexing

```
● ● ●  
1 func main() {  
2     slice := []any{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}  
3     fmt.Println(slice[2:5])  
4     fmt.Println(slice[5:])  
5     fmt.Println(slice[:4])  
6     fmt.Println(slice[:])  
7 }
```

```
$ go run _  
[2 3 4]  
[5 6 7 8 9]  
[0 1 2 3]  
[0 1 2 3 4 5 6 7 8 9]
```

25

Low is inclusive

High is exclusive

Omitting high will default to length of slice

Omitting low will default to 0

Slice Operations

- Append

Multiple items at once

```
1 func main() {  
2     s := []int{}  
3     s = append(s, 5)  
4     s = append(s, 1, 2, 3, 4, 5)  
5     append(s, 10) // error unused return value  
6 }
```

Append is a global function, not a function on slice type

Slice Operations

- Concatenate two slices
- `func append(slice []Type, elems ...Type) []Type`
- Variadic functions (later)

```
 1 func main() {  
 2     s1 := []int{1, 2, 3}  
 3     s2 := []int{4, 5, 6}  
 4     whole := append(s1, s2...)  
 5     fmt.Println(whole)  
 6 }
```

27

Use the `append()` function again

Append takes any number of arguments, which is what the dot dot dot syntax is

Slice Operations

- Remove elements

```
● ● ●  
1 func main() {  
2     arr := [6]int{0, 1, 2, 3, 4, 5}  
3     fmt.Println(arr)  
4     s := arr[:]  
5     idx := 2  
6     sNew := append(s[:idx], s[idx+1:]...)  
7     fmt.Println(arr)  
8     fmt.Println(sNew)  
9 }
```

```
$ go run .  
[0 1 2 3 4 5]  
[0 1 3 4 5 5]  
[0 1 3 4 5]
```

Underlying array

28

USE SLICING, NOT DIRECT WAY TO REMOVE ELEMENTS LIKE POP()

See underlying array does not change size

Slice Operations

- Insert multiple elements (start, middle)

```
1 func main() {  
2     s := []int{}  
3     s = append([]int{1, 5, 6}, s...)  
4     fmt.Println(s)  
5     insertAtIdx := 1  
6     toInsert := []int{2, 3, 4}  
7     s = append(s[:insertAtIdx], append(toInsert, s[insertAtIdx:]...)...)  
8     fmt.Println(s)  
9 }
```

```
$ go run .  
[1 5 6]  
[1 2 3 4 5 6]
```

- Whoa! Why need multiple append()?

29

Append is needed multiple times because '...' is only allowed once per function call.

Loops

```
1 func main() {
2     slice := make([]int, 1000)
3     // Init, condition, update
4     for i := 0; i < len(slice); i++ {
5         slice[i] = i % 10
6     }
7     // While
8     for len(slice) > 50 {
9         slice = slice[1:]
10    }
11    // Forever
12    for {
13        if slice[0]%2 == 0 {
14            break
15        }
16        slice = slice[1:]
17    }
18 }
```

30

Only loop in golang is FOR

Loops

- Optional init and update

```
1 func main() {
2     slice := make([]int, 1000)
3     idx := 0
4     idx = complexLogic(idx, slice)
5     /* Condition, update
6     for ; idx < len(slice); idx++ {
7         slice[idx] = idx % 10
8     }
9     // Init, condition
10    for idx2 := 0; idx2 < len(slice) ;{
11        slice[idx2] = manipulateElement(slice[idx])
12        // Complex idx2 update logic...
13        idx2 = (slice[idx2] % 10) + 2
14        idx2 = idx2 * idx2
15        idx2 = max(0, min(len(slice), idx2))
16    }
17 }
```

Loops Range - Iterator

- Range(val) → slice, array, map, string, channel

```
1 func main() {  
2     slice := []int{1, 2, 3, 4}  
3     arr := [...]int{1, 2, 3, 4}  
4     myMap := map[string]int{"a": 2, "b": 4, "c": 6, "d": 8}  
5     str := "abcdef"  
6  
7     for idx, el := range slice { ... }  
8     for idx := range slice { ... }  
9     for idx, el := range arr { ... }  
10    for idx := range arr { ... }  
11    for key, val := range myMap { ... }  
12    for key := range myMap { ... }  
13    for idx, char := range str { ... }  
14    for idx := range str { ... }  
15 }
```

- Different meaning based on datatype
- Optional second parameter

32

Range is an iterator helper

Array [...] is sugar for size of array

Structs

- Structs are simply a collection of fields. A bunch of data.

The slide features two code snippets side-by-side. The left snippet shows the definition of a struct named Jeff with various fields. The right snippet shows how to instantiate this struct in three different ways: zero initialization, specifying individual fields, and specifying all fields in order. A callout box highlights the third method as the best practice.

```
1 type Jeff struct {  
2     x, y int  
3     u     float32  
4     _     float32 // pad  
5     Arr  *[]int  // pointer to array  
6     Func func()  
7     Part OtherStruct  
8 }
```

```
1 func main() {  
2     jeff1 := Jeff{} // Zero init  
3     jeff2 := Jeff{y: 2, x: 1, Part: OtherStruct{num: 42}}  
4     // Need to specify all fields IN ORDER when not using {K: V}  
5     jeff3 := Jeff{5, 6, 3.14, &[]int{1, 2, 3}, SomeFunc, OtherStruct{}}  
6 }
```

How to instantiate structs?

"jeff2" method is best

33

new() vs make()

- **func new(Type) *Type**

-

```
1 func main() {  
2     var jeff *Jeff = new(Jeff) // zero init  
3     jeff.x = 42  
4     fmt.Println(jeff) // &{42 0 0 <nil> <nil> {0}}  
5  
6     var test = new([]int)  
7     *test = append(*test, 1, 2, 3)  
8     fmt.Println(test) // &[1 2 3]  
9 }
```

34

Both allocate storage
Both zero-init storage

Difference is that new() returns pointer, make()
returns value.

And make() can ONLY be used with slice, map,
channel

Enums

- Enums are just constants → usually integer, string

```
● ● ●  
1 type Command int  
2  
3 const (  
4     CommandForward   Command = 0  
5     CommandBackwards Command = 1  
6     CommandTurnLeft   Command = 2  
7     CommandTurnRight  Command = 3  
8 )
```

Enums

- Iota – index of the expression in the const declaration, starting at 0

```
● ● ●  
1 type Command int  
2  
3 const (  
4     CommandForward   Command = 0  
5     CommandBackwards Command = 1  
6     CommandTurnLeft  Command = 2  
7     CommandTurnRight Command = iota // 3  
8 )
```

Enums

- Const declarations only need type once

```
1 type Command int
2
3 const (
4     CommandForward    Command = iota
5     CommandBackwards   = iota
6     CommandTurnLeft    = iota
7     CommandTurnRight   = iota
8 )
```

Enums

- **Implicit repetition → empty expression repeats (copy pastes) last non-empty expression**

```
● ● ●  
1 type Command int  
2  
3 const (  
4     CommandForward   Command = iota // 0  
5     CommandBackwards           // 1  
6     CommandTurnLeft            // 2  
7     CommandTurnRight           // 3  
8 )
```

Enums

- Number of enums idiom (last element)

```
● ● ●  
1 type Command int  
2  
3 const (  
4     CommandForward   Command = iota  
5     CommandBackwards  
6     CommandTurnLeft  
7     CommandTurnRight  
8     NumberOfCommands  
9 )
```

Enums

- What value does G have?

```
 1 type Letter int
 2
 3 const (
 4     A Letter      = iota // 0
 5     B              // 1
 6     C              // 2
 7     D              // 3
 8     E = 1 << iota // 1 << 4 = 16
 9     F              // 32
10     G              // 64
11 )
```

Switch statements

- Simple
- Fallthrough opt-in
- Break implied, but can be used manually
- Default optional
- Non-exhaustive (unlike Rust)

```
1 type CardinalDir int
2
3 const (
4     North CardinalDir = iota
5     East
6     South
7     West
8 )
9
10 func doSomething(dir CardinalDir) string {
11     switch(dir) {
12     case North:
13         return "Success!"
14     case South:
15         fallthrough
16     case East, West:
17         panic("Only north allowed")
18     default:
19         panic("Unknown direction")
20     }
21 }
```

Switch statements

- Short assignment statement



```
1 func main() {
2     switch x := calculateArea(); x {
3         case 100:
4             fmt.Println("Area is 100!")
5         case 200:
6             fmt.Println("Area is 200!")
7         default:
8             panic("Invalid area!")
9     }
10 }
```

42

Switch statements

```
 1 func main() {
 2     x := calculateArea()
 3     switch { // implicit 'true' expression
 4     case x < 0:
 5         panic("Negative area bad!")
 6     case x > MAX_AREA:
 7         panic("Too big of area!")
 8     default:
 9         fmt.Println(x)
10    }
11 }
```

- Naked switch → switch true
- Complicated case expressions

Switch statements

- Type switch → check underlying type of variable

```
1 func foo(something any) any {
2     switch val := something.(type) {
3         case int:
4             return val + 40 // val is int
5         case float32:
6             return val / 4.1134 // val is float32
7         case string:
8             return "hello " + val + " there!" // val is string
9         default:
10            panic(fmt.Sprintf("Unhandled type %T", something))
11    }
12 }
```

switch something.(type) { ... }

Practice!

- <https://github.com/calvincramer/golang-training-1>



45