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. Only interfaces and 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**
- **1.20**
- **1.19**
- **1.18 (2022 March) – Generics, any (biggest change)**

Extremely conservative changes

How to Install

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

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

Hello World

```go
package main

import "fmt"

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

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 int = 5
3     a = 10           // write
4     var a int = 100 // no redeclarations
5 }
```

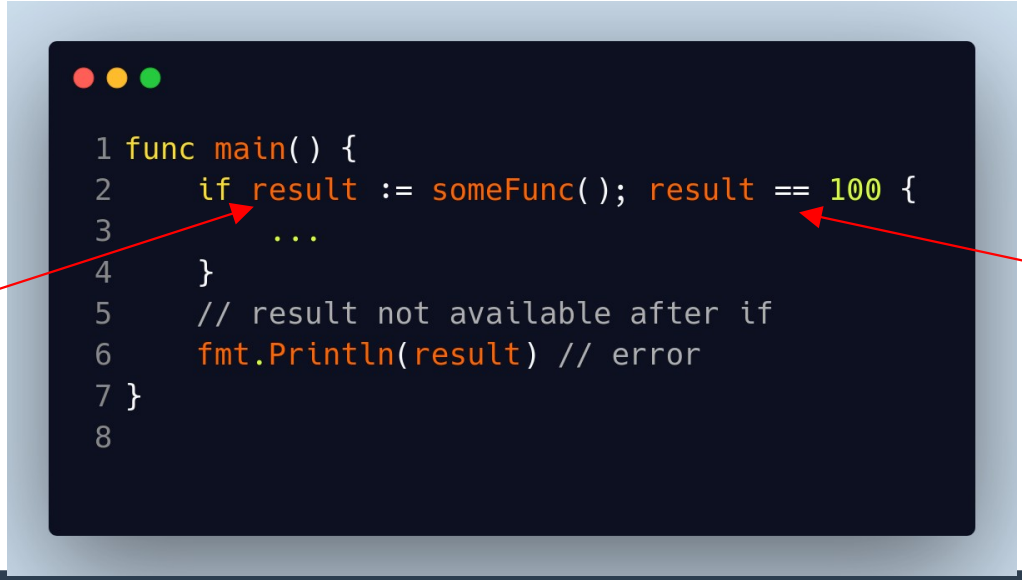
Variables (2)

- Use := for brevity, let Golang do type inferrence 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 }
```

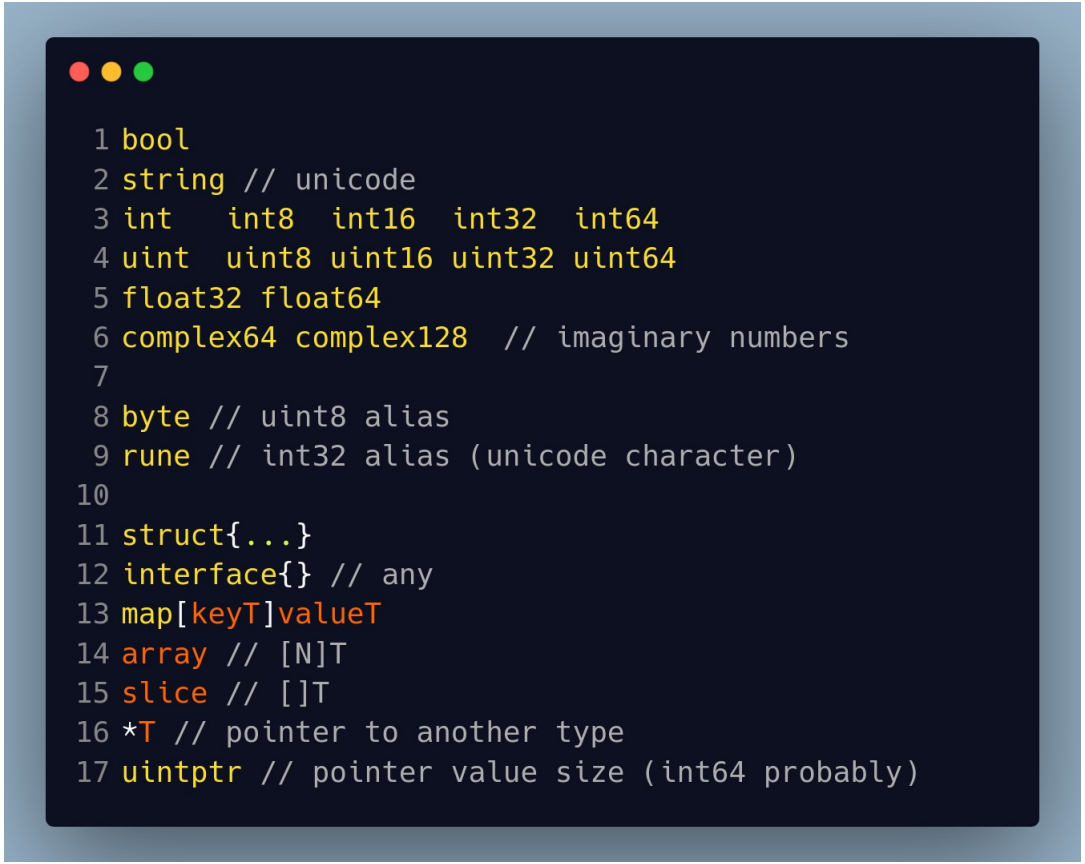
Short declarations

- “Short” declaration scopes variables to ‘if’. **Only := not =**
- Also can use with ‘for’ (init statement) and ‘switch’
- Use for variables don’t care about after the ‘if’ statement



Data Types

- Very few



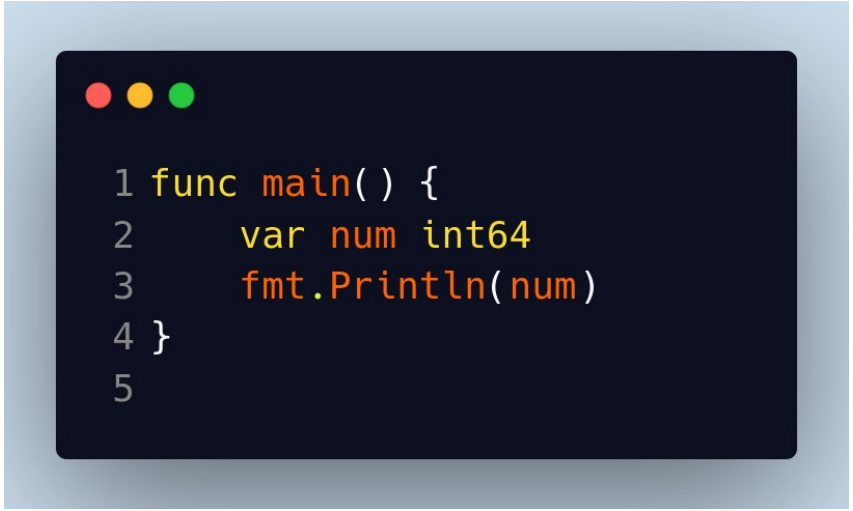
```
bool
string // unicode
int   int8  int16  int32  int64
uint  uint8 uint16 uint32 uint64
float32 float64
complex64 complex128  // imaginary numbers

byte // uint8 alias
rune // int32 alias (unicode character)

struct{...}
interface{} // any
map[keyT]valueT
array // [N]T
slice // []T
*T // pointer to another type
uintptr // pointer value size (int64 probably)
```

Zero Values

- **What will be printed?**

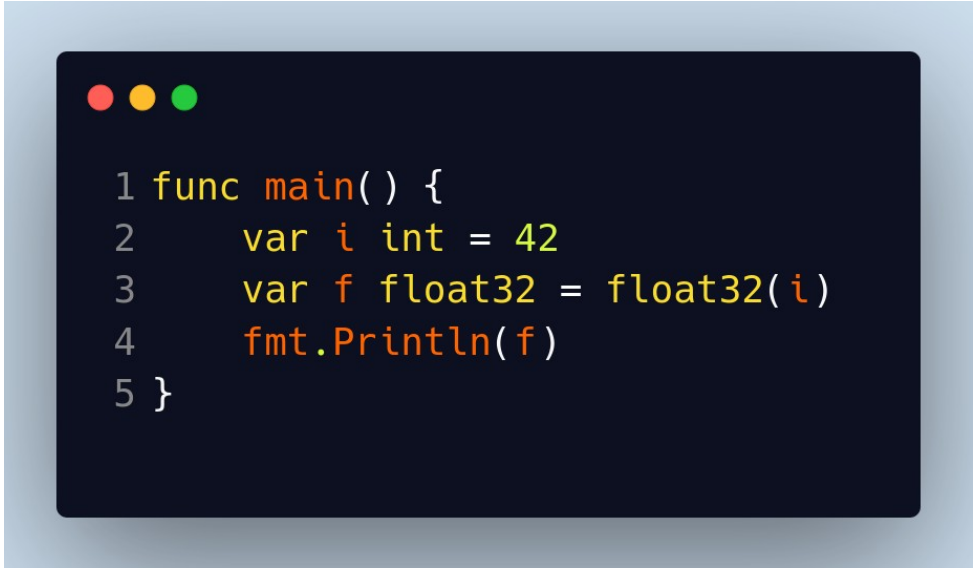


```
1 func main() {
2     var num int64
3     fmt.Println(num)
4 }
5
```

- Integers → 0
- Floats → 0.0
- Boolean → false
- Strings → “”
- Interface, slice, channel, map, pointers, functions → nil (aka null)
- Array, struct → zero value for each member

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**

1) **Numbers are arbitrary precision**
2) **Wraps**
3) **Runtime exception**

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

Wrap!

```go
func main() {
    var num uint8 = 0b1111_1110
    fmt.Printf("0x%X\n", num)
    num += 1
    fmt.Printf("0x%X\n", num)
    num += 1
    fmt.Printf("0x%X\n", num)
}
```

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 }
```

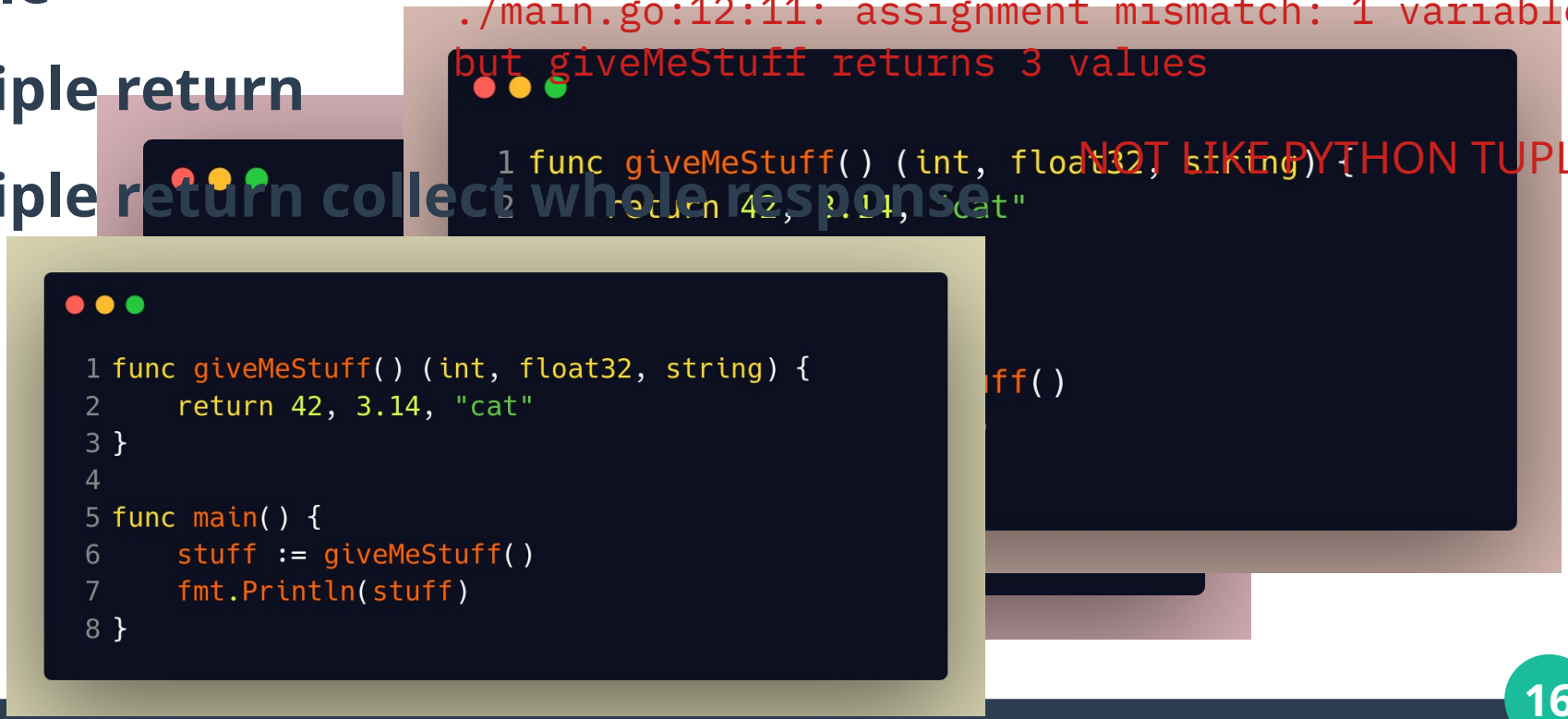
- Divide by zero
- Learn about panics later

```
$ 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



Functions more

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

```go
func giveMeStuff() (code int, radius float32, msg string) {
    radius = 3.14
    msg = "small step for man"
    return
}

func main() {
    a, b, c := giveMeStuff()
    fmt.Println(a, b, c)
}
```

Maps

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

Arbitrary scoping

```go
func main() {
    m := map[string]int{"width": 100}
    m["height"] = 200
    {
        val, exists := m["width"]
        fmt.Println(val, exists) // 100, true
    }
    {
        val, exists := m["not exist"]
        fmt.Println(val, exists) // 0, false
    }
    delete(m, "width")
    delete(m, "not exist") // OK
    clear(m)
}
```

Maps – be careful!

```go
func main() {
    m := map[int]float64{}
    test := m[400]
    fmt.Println(test)
}
```

- **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)}

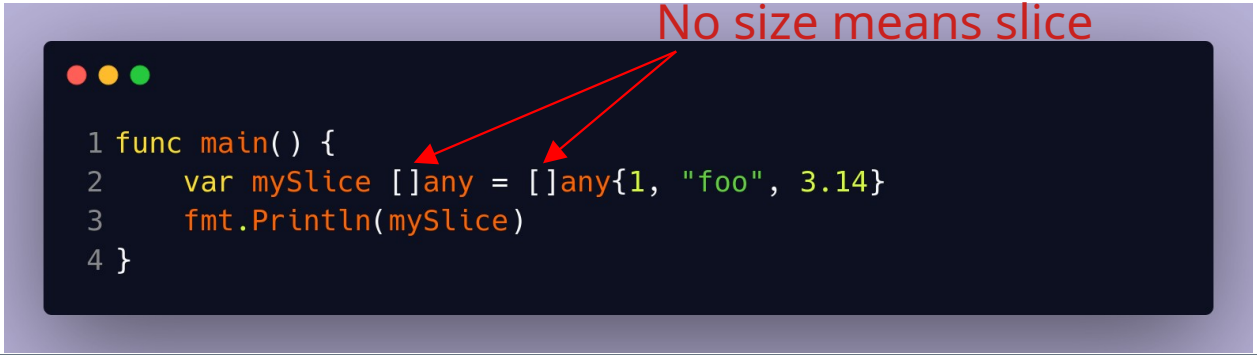
Arrays and Slices

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

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

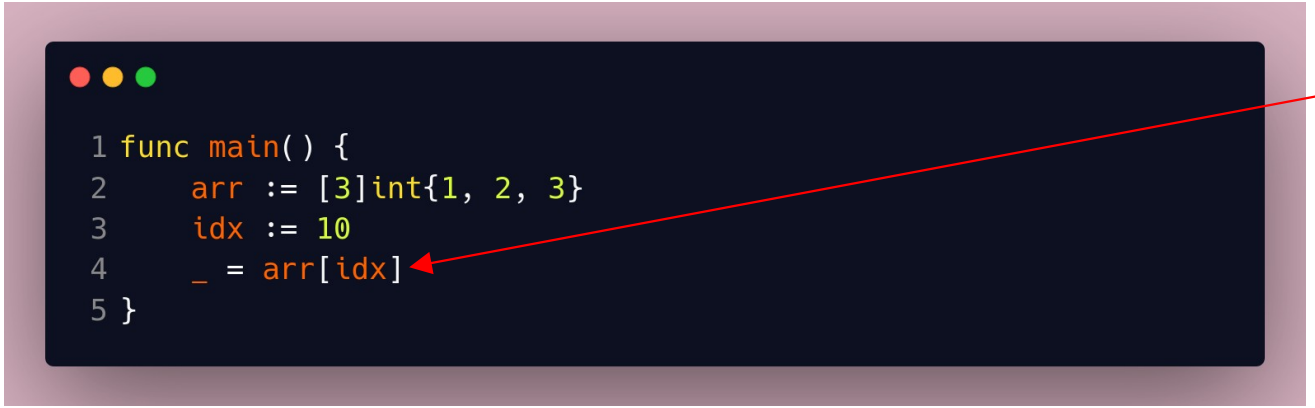
Arrays and Slices

- 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
- See ‘slices’ standard package for common operations



Let’s Cause Errors!

- **Array index out of bounds, what happens?**




```
$ 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**

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

Slice Expression

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

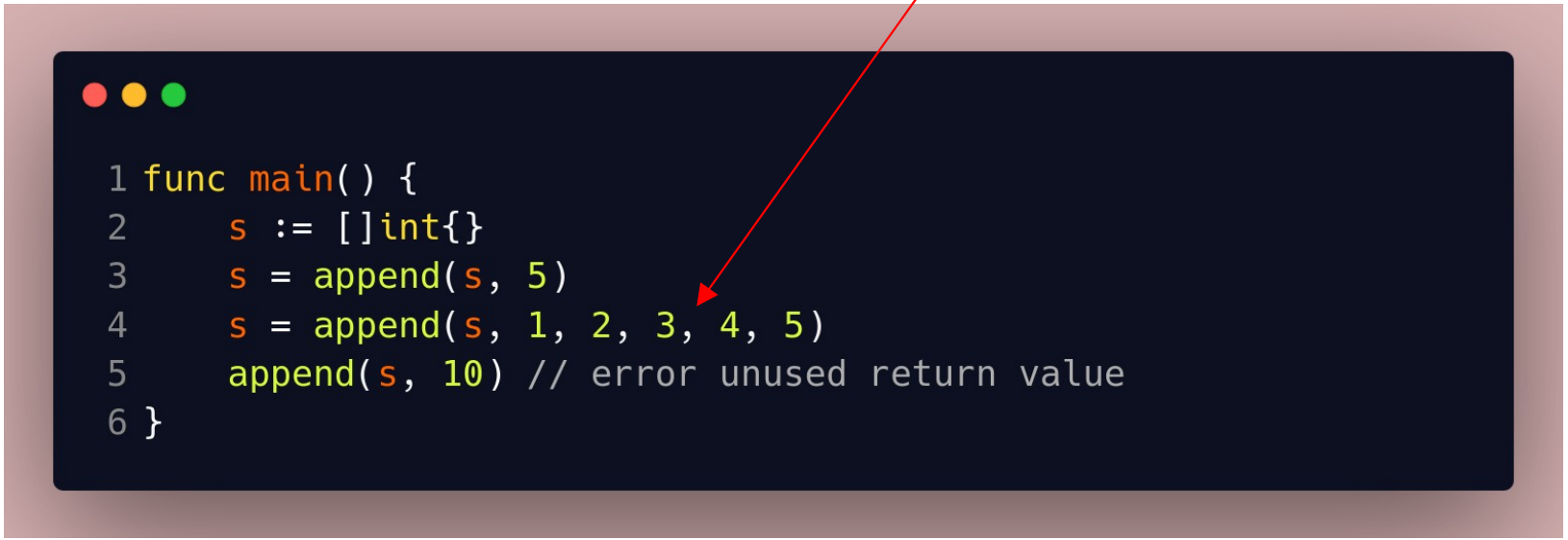
```go
func main() {
    slice := []any{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
    fmt.Println(slice[2:5])
    fmt.Println(slice[5:])
    fmt.Println(slice[:4])
    fmt.Println(slice[:])
}
```

```
$ 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

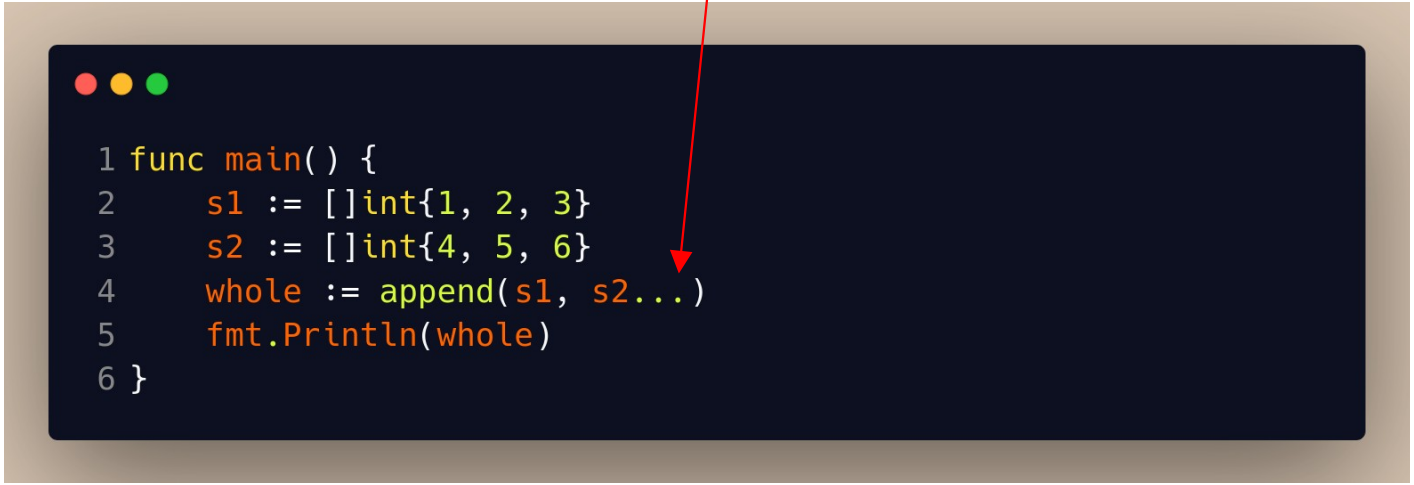


```
1 func main() {
2     s := []int{}
3     s = append(s, 5)
4     s = append(s, 1, 2, 3, 4, 5)
5     append(s, 10) // error unused return value
6 }
```

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)



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

Slice Operations

- **Remove elements**

```go
func main() {
    arr := [6]int{0, 1, 2, 3, 4, 5}
    fmt.Println("Orig arr", arr)
    s1 := arr[:]
    s2 := append(s1[:2], s1[2+1:]...)
    fmt.Println("After")
    fmt.Println("arr", arr)
    fmt.Println("s1 ", s1)
    fmt.Println("s2 ", s2)
}
```

```
$ go run .
Orig arr [0 1 2 3 4 5]
After
arr [0 1 3 4 5 5]
s1  [0 1 3 4 5 5]
s2  [0 1 3 4 5]
```

Slice Operations

- **Insert multiple elements (start, middle)**

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

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

- **Whoa! Why need multiple append()?**

Loops

```go
func main() {
    slice := make([]int, 1000)
    // Init, condition, update
    for i := 0; i < len(slice); i++ {
        slice[i] = i % 10
    }
    // While
    for len(slice) > 50 {
        slice = slice[1:]
    }
    // Forever
    for {
        if slice[0]%2 == 0 {
            break
        }
        slice = slice[1:]
    }
}
```

Loops

```go
func main() {
    slice := make([]int, 1000)
    idx := 0
    idx = complexLogic(idx, slice)
    // Condition, update
    for ; idx < len(slice); idx++ {
        slice[idx] = idx % 10
    }
    // Init, condition
    for idx2 := 0; idx2 < len(slice) ; {
        slice[idx2] = manipulateElement(slice[idx])
        // Complex idx2 update logic...
        idx2 = (slice[idx2] % 10) + 2
        idx2 = idx2 * idx2
        idx2 = max(0, min(len(slice), idx2))
    }
}
```

- Optional init and update
- Optional condition

```go
func main() {
    for i := 100; ; i++ {
        fmt.Println(i)
        if i > 500 {
            break
        }
    }
}
```

Loops Range - Iterator

- range val → slice, array, map, string, channel

```go
func main() {
    slice := []int{1, 2, 3, 4}
    arr := [...]int{1, 2, 3, 4}
    myMap := map[string]int{"a": 2, "b": 4, "c": 6, "d": 8}
    str := "abcdef"

    for idx, el := range slice { ... }
    for idx := range slice { ... }
    for idx, el := range arr { ... }
    for idx := range arr { ... }
    for key, val := range myMap { ... }
    for key := range myMap { ... }
    for idx, char := range str { ... }
    for idx := range str { ... }
}
```

- Different meaning based on datatype
- Optional second parameter
- Use '_' to ignore idx

```go
for _, el := range slice { ... }
```

Structs

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

```
type Jeff struct {
    x, y int
    u    float32
    Arr  *[]int //
    Part OtherStruc
}
```

How to instantiate structs?

```
func main() {
    jeff1 := Jeff{} // Zero init
    // Need to specify ALL fields IN ORDER:
    jeff2 := Jeff{5, 6, 3.14, &[]int{1, 2, 3}, OtherStruct{}}
    jeff3 := Jeff{y: 2, x: 1, Part: OtherStruct{num: 42}}
}
```

“jeff3” 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 // sugar for (*jeff).x
4     fmt.Println(jeff) // &{42 0 0 <nil> {0}}
5 }
```

Enums

- Enums are just constants → usually integer, string
- Enums do not actually exist as a feature of the language

```
type Command int

const (
    CommandForward   Command = 0
    CommandBackwards Command = 1
    CommandTurnLeft  Command = 2
    CommandTurnRight Command = 3
)
```

Enums

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

```go
type Command int

const (
    CommandForward   Command = 0
    CommandBackwards Command = 1
    CommandTurnLeft  Command = 2
    CommandTurnRight Command = iota // 3
)
```

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 )
```

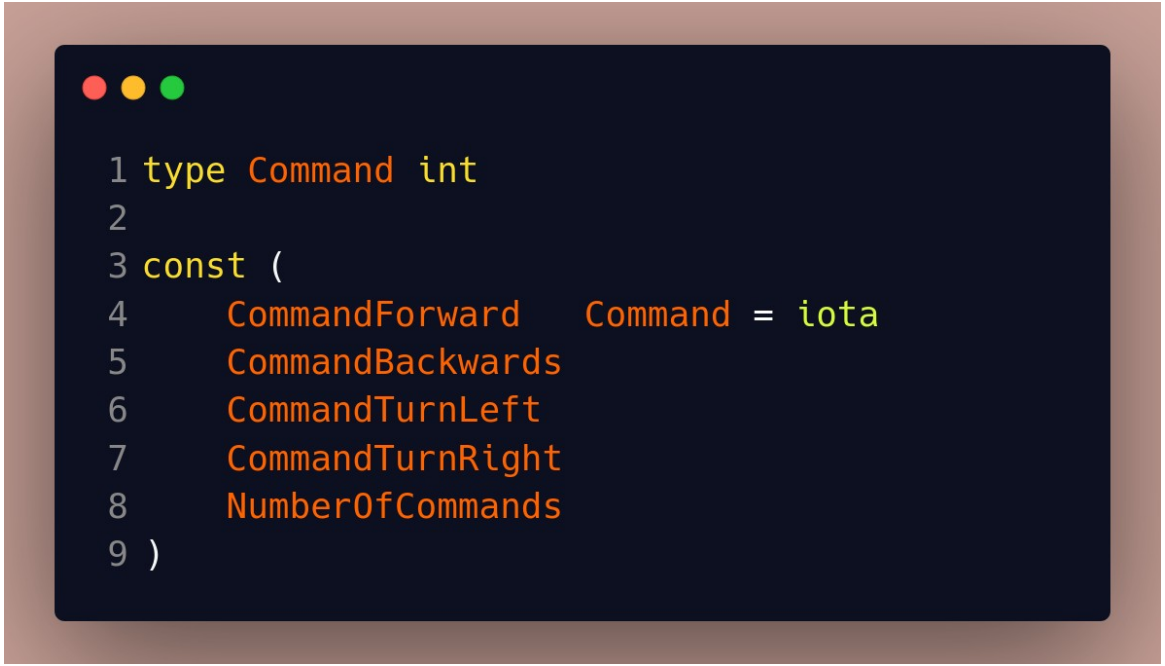
Enums

- **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)**

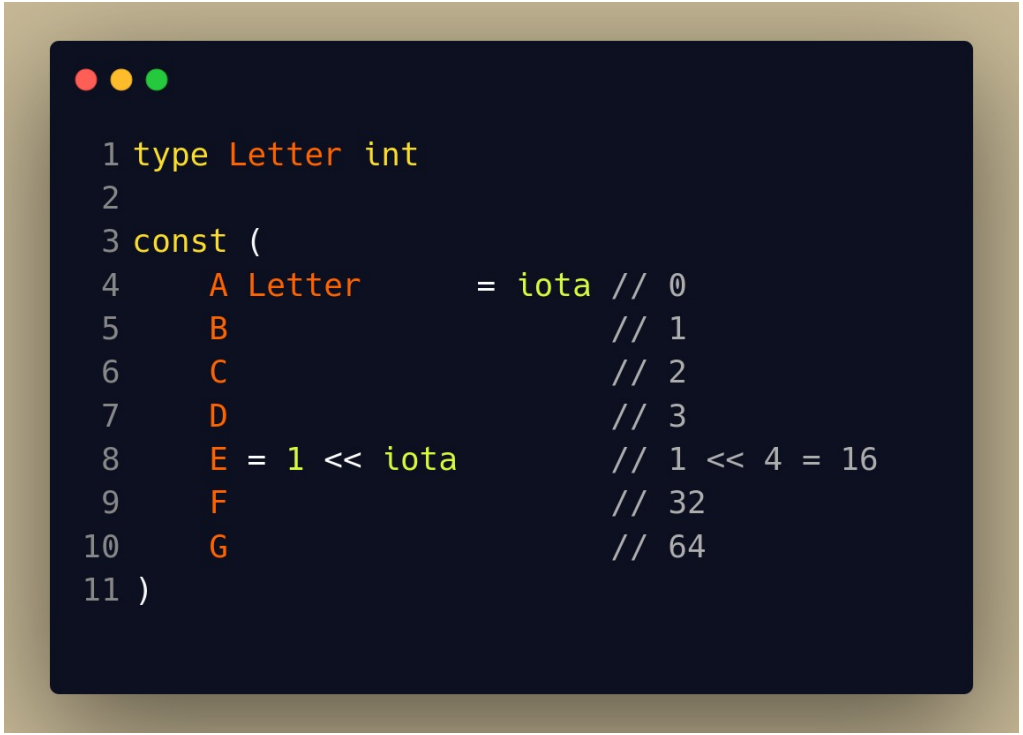


```
type Command int

const (
    CommandForward   Command = iota
    CommandBackwards
    CommandTurnLeft
    CommandTurnRight
    NumberOfCommands
)
```

Enums

- What value does G have?



```
type Letter int

const (
    A Letter      = iota // 0
    B                    // 1
    C                    // 2
    D                    // 3
    E = 1 << iota        // 1 << 4 = 16
    F                    // 32
    G                    // 64
)
```

Switch statements

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

```go
type CardinalDir int

const (
    North CardinalDir = iota
    East
    South
    West
)

func doSomething(dir CardinalDir) string {
    switch(dir) {
    case North:
        return "Success!"
    case South:
        fallthrough
    case East, West:
        panic("Only north allowed")
    default:
        panic("Unknown direction")
    }
}
```

Switch statements

- **Short assignment statement**

```go
func main() {
    switch x := calculateArea(); x {
    case 100:
        fmt.Println("Area is 100!")
    case 200:
        fmt.Println("Area is 200!")
    default:
        panic("Invalid area!")
    }
}
```

Switch statements

```go
func main() {
    x := calculateArea()
    switch { // implicit 'true' expression
    case x < 0:
        panic("Negative area bad!")
    case x > MAX_AREA:
        panic("Too big of area!")
    default:
        fmt.Println(x)
    }
}
```

- Naked switch → switch true
- Complicated case expressions
- Difference between naked switch and if elseif elseif ...

Switch statements

- **Type switch → check underlying type of variable**

```go
func foo(something any) any {
    switch val := something.(type) {
    case int:
        return val + 40 // val is int
    case float32:
        return val / 4.1134 // val is float32
    case string:
        return "hello " + val + " there!" // val is string
    default:
        panic(fmt.Sprintf("Unhandled type %T", something))
    }
}
```

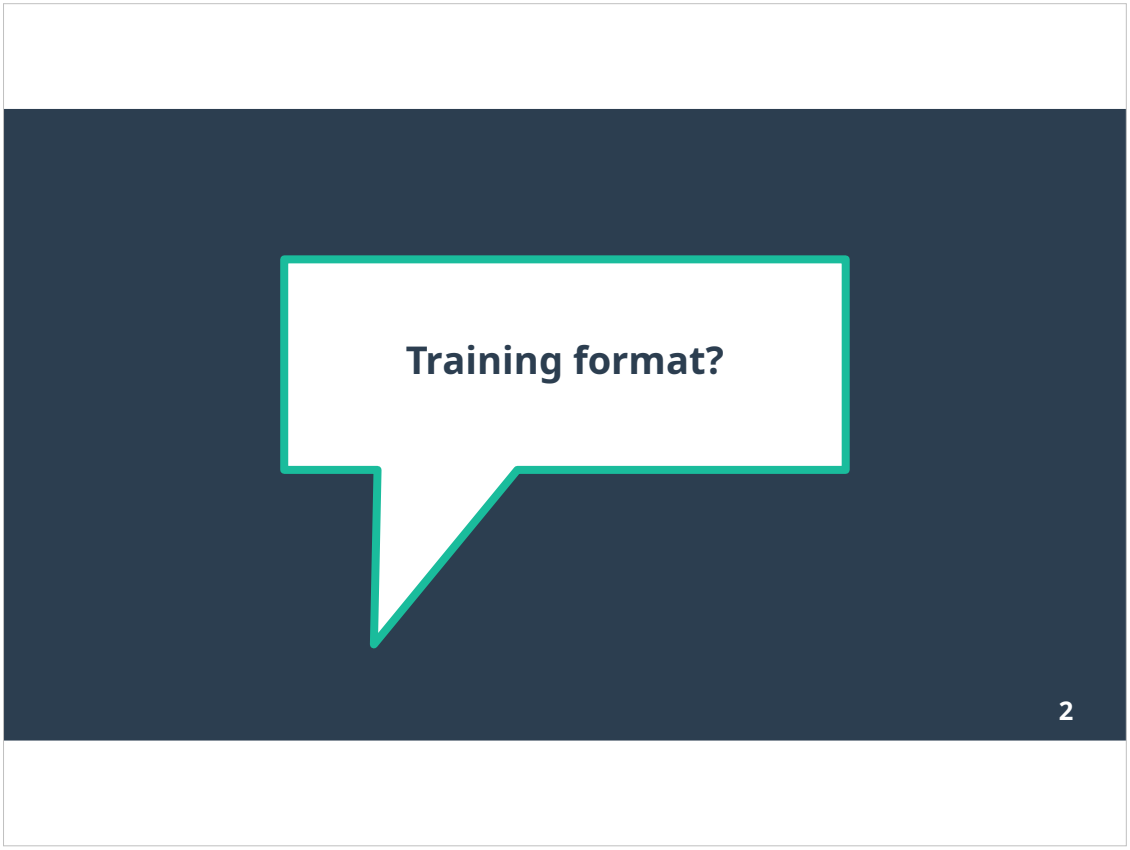
switch something.(type) { ... }

Practice!

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

Golang Training 1 – Basics, Syntax, Data Structures

Calvin Lei-Cramer
May 2025



Our training format will be four lessons.

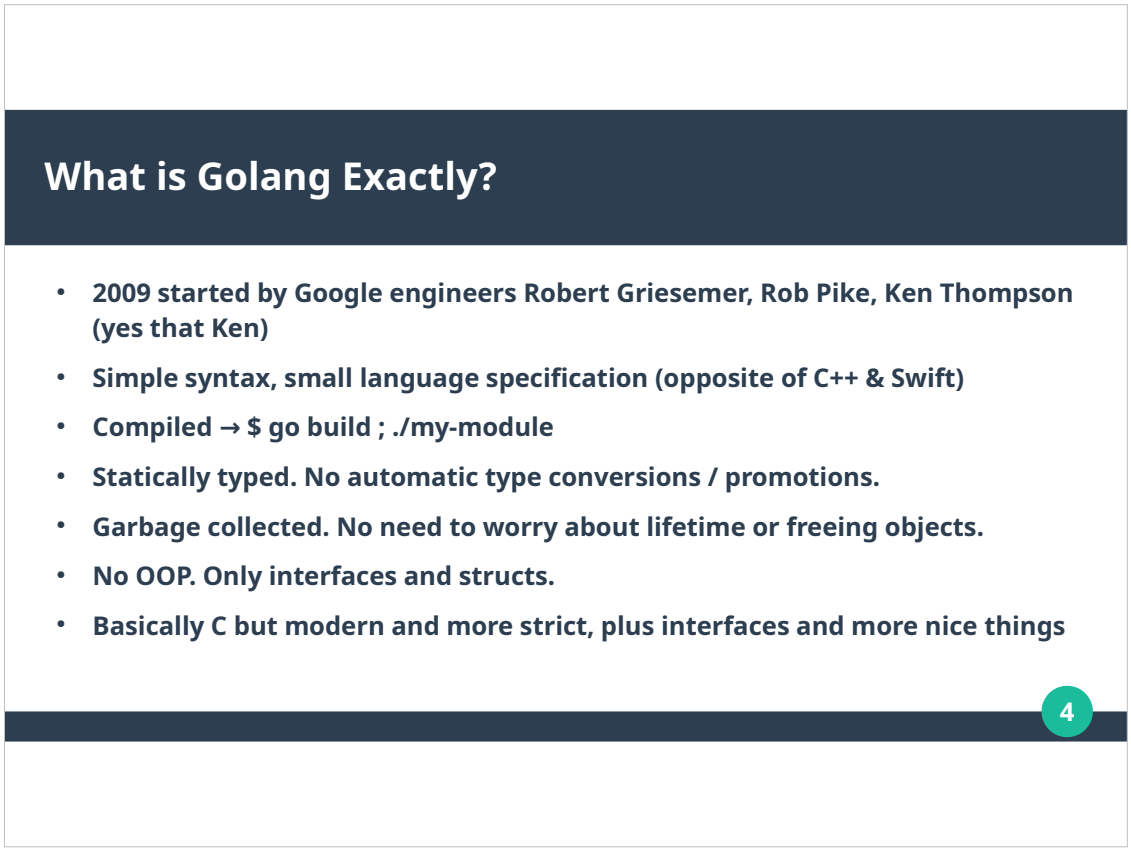
After each lesson I will release exercises you can complete to practice and reinforce the ideas.

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.

Training assumes basic knowledge of other languages. Will not go over the very basics.

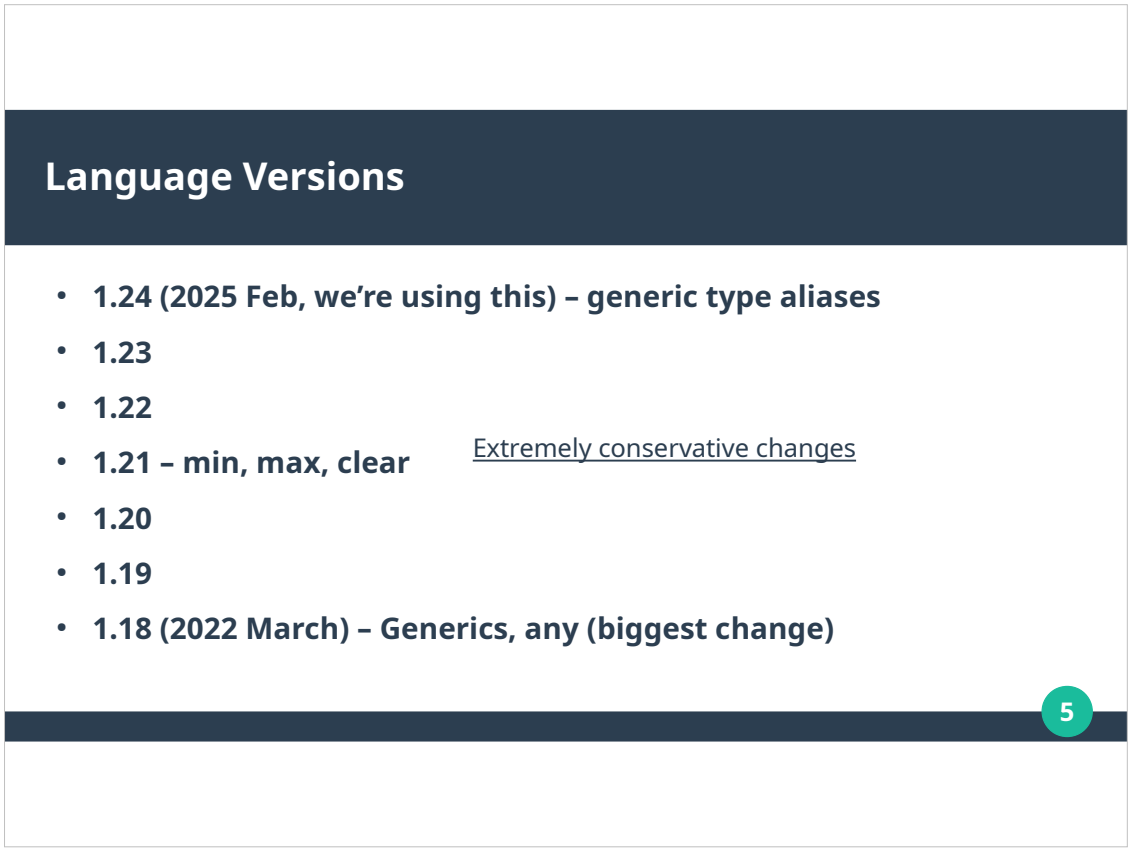

What are we covering today?
Variables, data types, functions, maps, loops, arrays, slices, structs, new(), make(), enums, switch



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.



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

Golang compatibility promise is basically a valid Go program will compile and work correctly in future releases. Promised since 1.0.

How to Install

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

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

Hello World

```go
package main

import "fmt"

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

Main.go

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

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

Variables

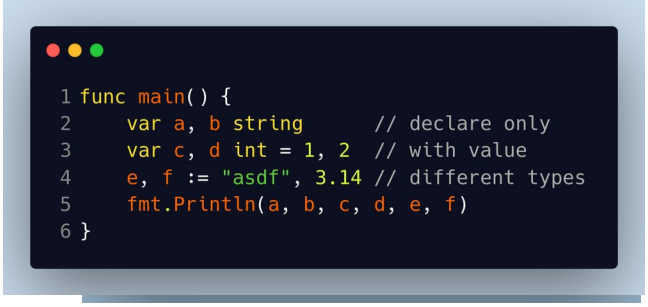
- **Use = to specify type and for globals**
- **Re-declaring variables is error**
- **Unused variables is error**

The compiler is your friend

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

Variables (2)

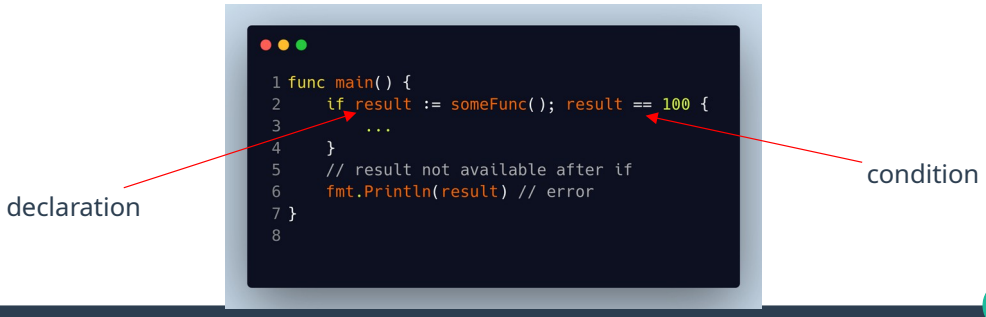
- Use := for brevity, let Golang do type inferrence like C++ auto
- Multiple declarations at once



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

Short declarations

- “Short” declaration scopes variables to ‘if’. **Only := not =**
- Also can use with ‘for’ (init statement) and ‘switch’
- Use for variables don’t care about after the ‘if’ statement



Data Types

- Very few

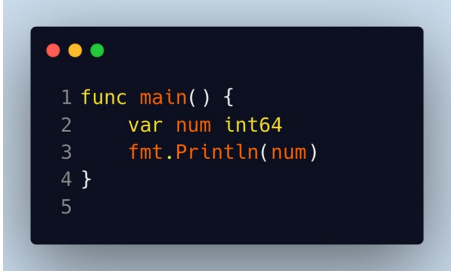
```
bool
string // unicode
int   int8  int16  int32  int64
uint  uint8 uint16 uint32 uint64
float32 float64
complex64 complex128  // imaginary numbers

byte // uint8 alias
rune // int32 alias (unicode character)

struct{...}
interface{} // any
map[keyT]valueT
array // [N]T
slice // []T
*T // pointer to another type
uintptr // pointer value size (int64 probably)
```

Zero Values

- **What will be printed?**

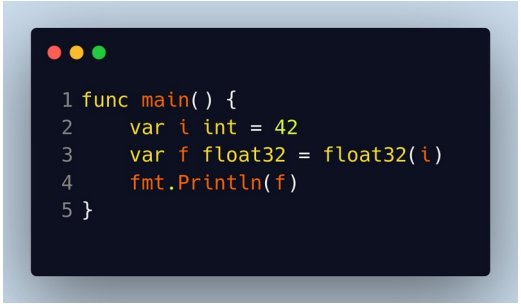


```
1 func main() {
2     var num int64
3     fmt.Println(num)
4 }
5
```

  - Integers → 0
  - Floats → 0.0
  - Boolean → false
  - Strings → “”
  - Interface, slice, channel, map, pointers, functions → nil (aka null)
  - Array, struct → zero value for each member

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**

1) **Numbers are arbitrary precision**
2) **Wraps**
3) **Runtime exception**

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

Wrap!

```go
func main() {
    var num uint8 = 0b1111_1110
    fmt.Printf("0x%X\n", num)
    num += 1
    fmt.Printf("0x%X\n", num)
    num += 1
    fmt.Printf("0x%X\n", num)
}
```

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


```
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”

```go
func giveMeStuff() (code int, radius float32, msg string) {
    radius = 3.14
    msg = "small step for man"
    return
}

func main() {
    a, b, c := giveMeStuff()
    fmt.Println(a, b, c)
}
```

Maps

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

Arbitrary scoping

```go
func main() {
    m := map[string]int{"width": 100}
    m["height"] = 200
    {
        val, exists := m["width"]
        fmt.Println(val, exists) // 100, true
    }
    {
        val, exists := m["not exist"]
        fmt.Println(val, exists) // 0, false
    }
    delete(m, "width")
    delete(m, "not exist") // OK
    clear(m)
}
```

Maps – be careful!

```go
func main() {
    m := map[int]float64{}
    test := m[400]
    fmt.Println(test)
}
```

- **Prints 0!**

What happens here?
Compiler error?
Runtime panic?

Helpful debugging

- **Print (%v, %#v)**

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

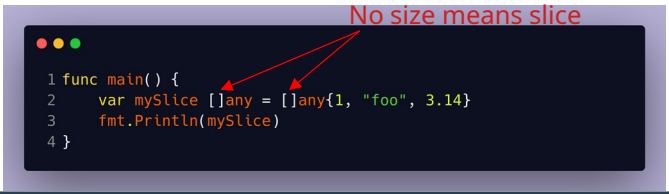
- **Array – fixed sized sequence of elements of the same type**
- **Size is included in type → [10]int != [20]int**

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


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

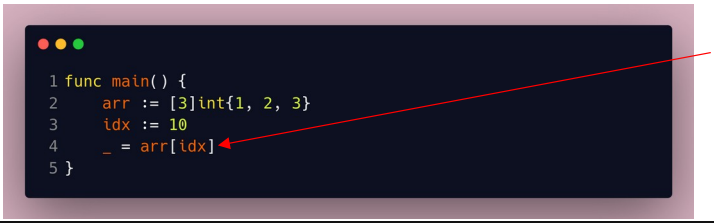
Arrays and Slices

- 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
- See ‘slices’ standard package for common operations



Let's Cause Errors!

- **Array index out of bounds, what happens?**




```
$ 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
```


Both array and slice panic

Create Slices

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

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

Slice Expression

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

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

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

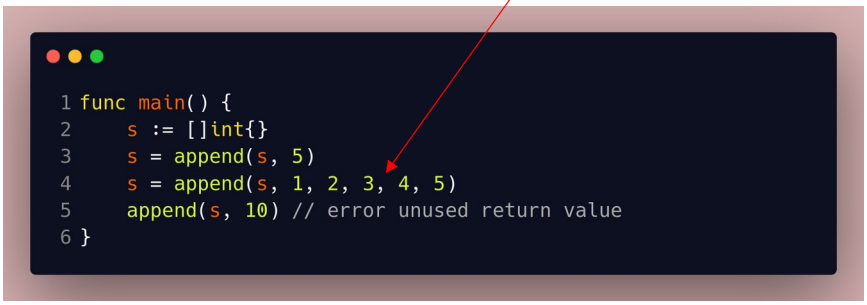
25

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

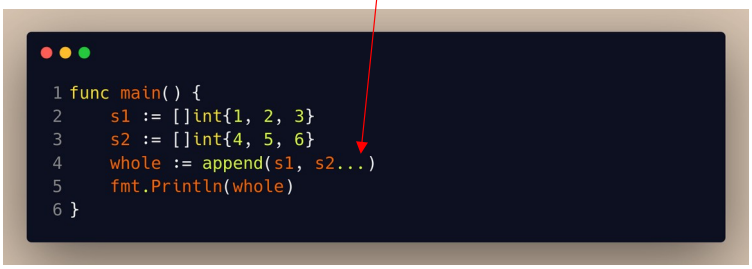


```go
func main() {
    s := []int{}
    s = append(s, 5)
    s = append(s, 1, 2, 3, 4, 5)
    append(s, 10) // error unused return value
}
```

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 }
```


Use the append() function again

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

Slice Operations

- **Remove elements**

```go
func main() {
    arr := [6]int{0, 1, 2, 3, 4, 5}
    fmt.Println("Orig arr", arr)
    s1 := arr[:]
    s2 := append(s1[:2], s1[2+1:]...)
    fmt.Println("After")
    fmt.Println("arr", arr)
    fmt.Println("s1 ", s1)
    fmt.Println("s2 ", s2)
}
```

```
$ go run .
Orig arr [0 1 2 3 4 5]
After
arr [0 1 3 4 5 5]
s1  [0 1 3 4 5 5]
s2  [0 1 3 4 5]
```

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)**

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

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

- **Whoa! Why need multiple append()?**

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

Loops

```go
func main() {
    slice := make([]int, 1000)
    // Init, condition, update
    for i := 0; i < len(slice); i++ {
        slice[i] = i % 10
    }
    // While
    for len(slice) > 50 {
        slice = slice[1:]
    }
    // Forever
    for {
        if slice[0]%2 == 0 {
            break
        }
        slice = slice[1:]
    }
}
```


Only loop in golang is FOR

Loops

```go
func main() {
    slice := make([]int, 1000)
    idx := 0
    idx = complexLogic(idx, slice)
    // Condition, update
    for ; idx < len(slice); idx++ {
        slice[idx] = idx % 10
    }
    // Init, condition
    for idx2 := 0; idx2 < len(slice) ; {
        slice[idx2] = manipulateElement(slice[idx])
        // Complex idx2 update logic...
        idx2 = (slice[idx2] % 10) + 2
        idx2 = idx2 * idx2
        idx2 = max(0, min(len(slice), idx2))
    }
}
```

- Optional init and update
- Optional condition

```go
func main() {
    for i := 100; ; i++ {
        fmt.Println(i)
        if i > 500 {
            break
        }
    }
}
```

Loops Range - Iterator

- range val → slice, array, map, string, channel

```go
func main() {
    slice := []int{1, 2, 3, 4}
    arr := [...]int{1, 2, 3, 4}
    myMap := map[string]int{"a": 2, "b": 4, "c": 6, "d": 8}
    str := "abcdef"

    for idx, el := range slice { ... }
    for idx := range slice { ... }
    for idx, el := range arr { ... }
    for idx := range arr { ... }
    for key, val := range myMap { ... }
    for key := range myMap { ... }
    for idx, char := range str { ... }
    for idx := range str { ... }
}
```

- Different meaning based on datatype
- Optional second parameter
- Use '_' to ignore idx

```go
for _, el := range slice { ... }
```

32

Range is an iterator helper

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

LOOKS LIKE IT VIOLATES THE PREVIOUS RULE
THAT FUNCTION CALLS THAT RETURN MULTIPLE VALUES NEED TO STORE ALL THE VALUES

Structs

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

```
type Jeff struct {
    x, y int
    u    float32
    Arr  *[]int //
    Part OtherStruc
}
```

How to instantiate structs?

```
func main() {
    jeff1 := Jeff{} // Zero init
    // Need to specify ALL fields IN ORDER:
    jeff2 := Jeff{5, 6, 3.14, &[]int{1, 2, 3}, OtherStruct{}}
    jeff3 := Jeff{y: 2, x: 1, Part: OtherStruct{num: 42}}
}
```

“jeff3” 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 // sugar for (*jeff).x
4     fmt.Println(jeff) // &{42 0 0 <nil> {0}}
5 }
```

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

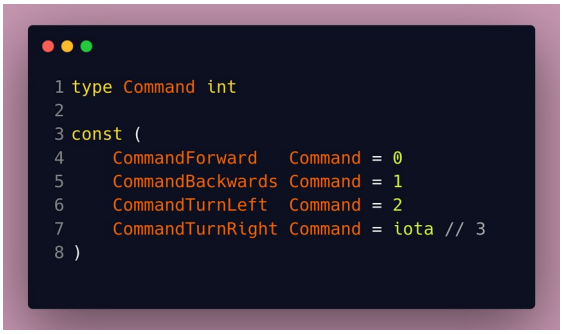
- Enums are just constants → usually integer, string
- Enums do not actually exist as a feature of the language

```
type Command int

const (
    CommandForward   Command = 0
    CommandBackwards Command = 1
    CommandTurnLeft  Command = 2
    CommandTurnRight Command = 3
)
```

Enums

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

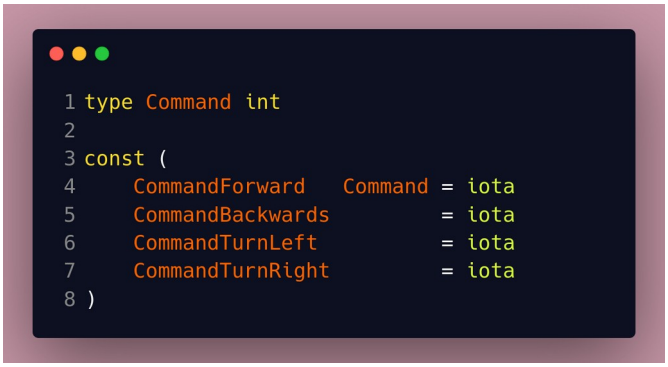


```
type Command int

const (
    CommandForward   Command = 0
    CommandBackwards Command = 1
    CommandTurnLeft  Command = 2
    CommandTurnRight Command = iota // 3
)
```

Enums

- Const declarations only need type once



```
type Command int

const (
    CommandForward   Command = iota
    CommandBackwards         = iota
    CommandTurnLeft          = iota
    CommandTurnRight         = iota
)
```

Enums

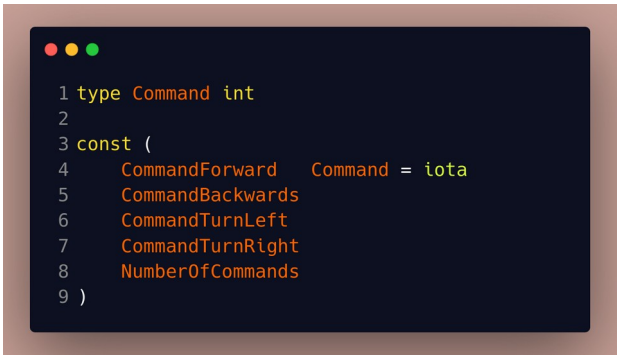
- **Implicit repetition → empty expression repeats (copy pastes) last non-empty expression**

```
type Command int

const (
    CommandForward   Command = iota // 0
    CommandBackwards                // 1
    CommandTurnLeft                 // 2
    CommandTurnRight                // 3
)
```

Enums

- **Number of enums idiom (last element)**

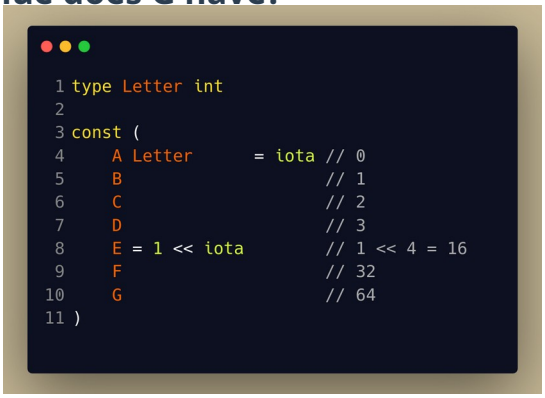


```
type Command int

const (
    CommandForward   Command = iota
    CommandBackwards
    CommandTurnLeft
    CommandTurnRight
    NumberOfCommands
)
```

Enums

- What value does G have?

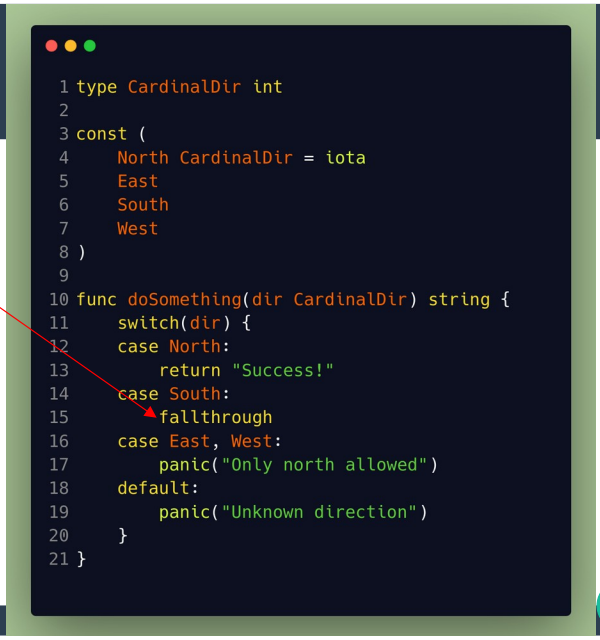


```
type Letter int

const (
    A Letter      = iota // 0
    B                    // 1
    C                    // 2
    D                    // 3
    E = 1 << iota        // 1 << 4 = 16
    F                    // 32
    G                    // 64
)
```

Switch statements

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



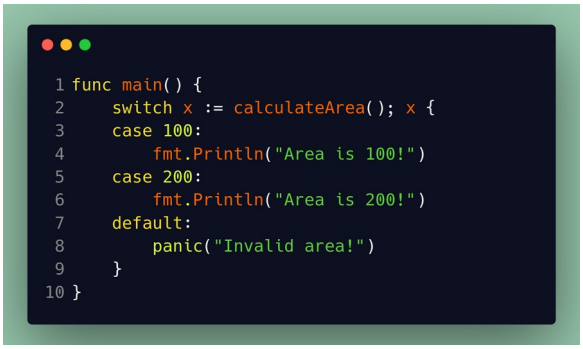
```go
type CardinalDir int

const (
    North CardinalDir = iota
    East
    South
    West
)

func doSomething(dir CardinalDir) string {
    switch(dir) {
    case North:
        return "Success!"
    case South:
        fallthrough
    case East, West:
        panic("Only north allowed")
    default:
        panic("Unknown direction")
    }
}
```

Switch statements

- **Short assignment statement**



```go
func main() {
    switch x := calculateArea(); x {
    case 100:
        fmt.Println("Area is 100!")
    case 200:
        fmt.Println("Area is 200!")
    default:
        panic("Invalid area!")
    }
}
```

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
- Difference between naked switch and if elseif elseif ...

43

Difference between switch and if else-if is that all switch cases will be executed top down, allowing for multiple cases to run. If else-if will always run at most one branch.

Switch statements

- **Type switch → check underlying type of variable**

```go
func foo(something any) any {
    switch val := something.(type) {
    case int:
        return val + 40 // val is int
    case float32:
        return val / 4.1134 // val is float32
    case string:
        return "hello " + val + " there!" // val is string
    default:
        panic(fmt.Sprintf("Unhandled type %T", something))
    }
}
```

switch something.(type) { ... }

Practice!

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