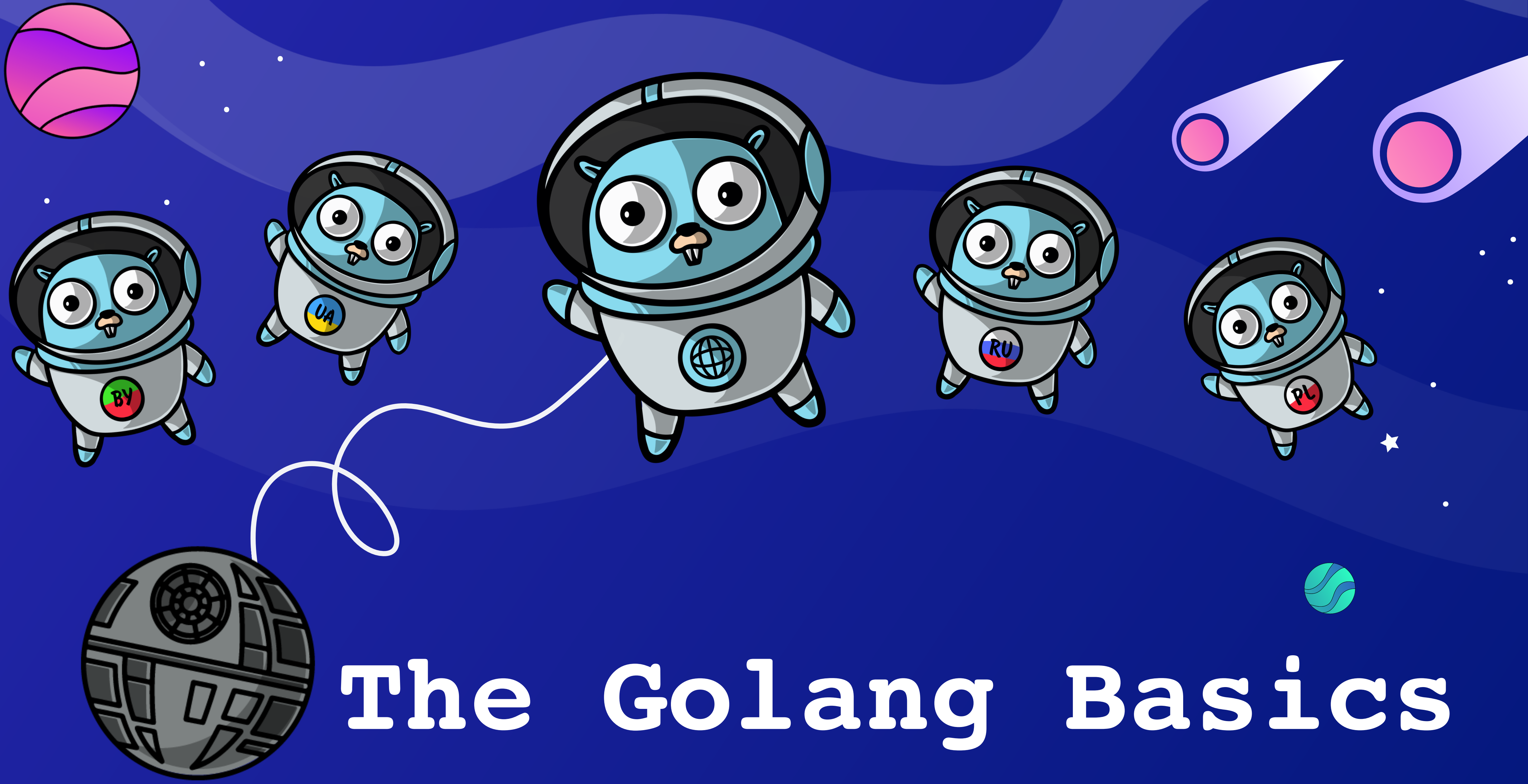


The Golang Basics

What we cover on this lecture?

- ***Variables***
- ***Basic types***
- ***Constants***
- ***functions***
- ***Flow-control***
- ***Loops***
- ***Packages***

Golang keywords

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

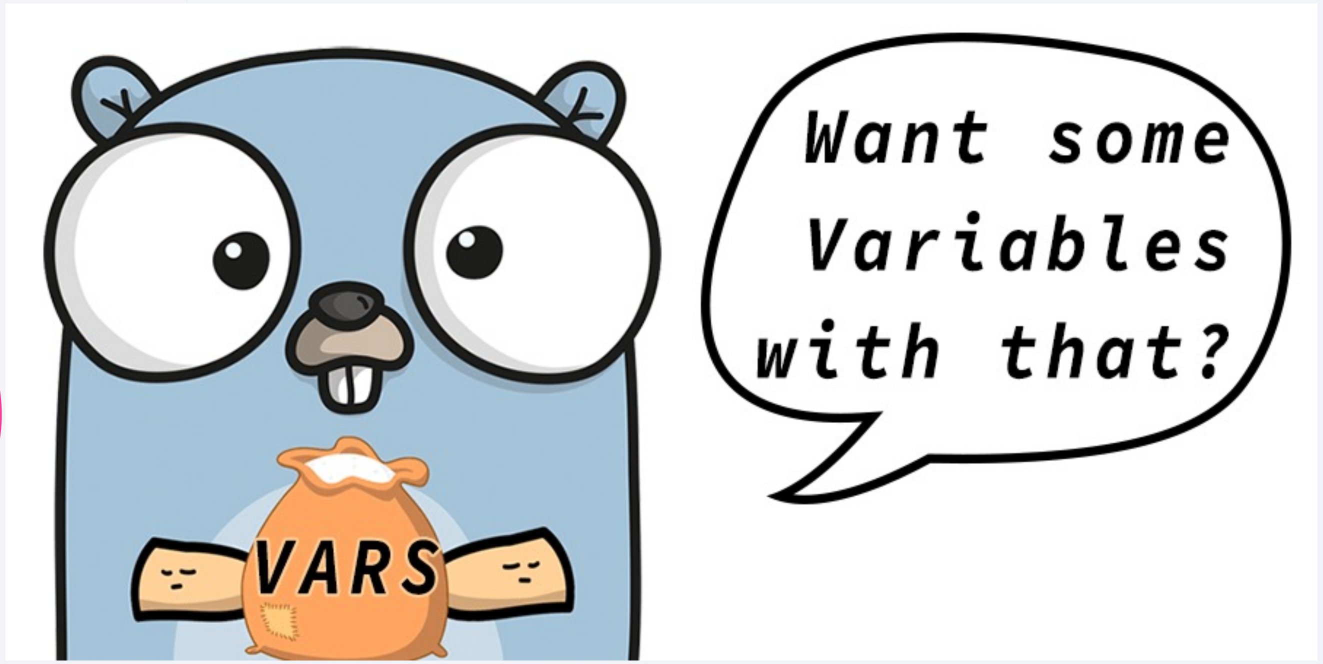
You cannot use key words as variables, types, function names in your program

Predeclared names

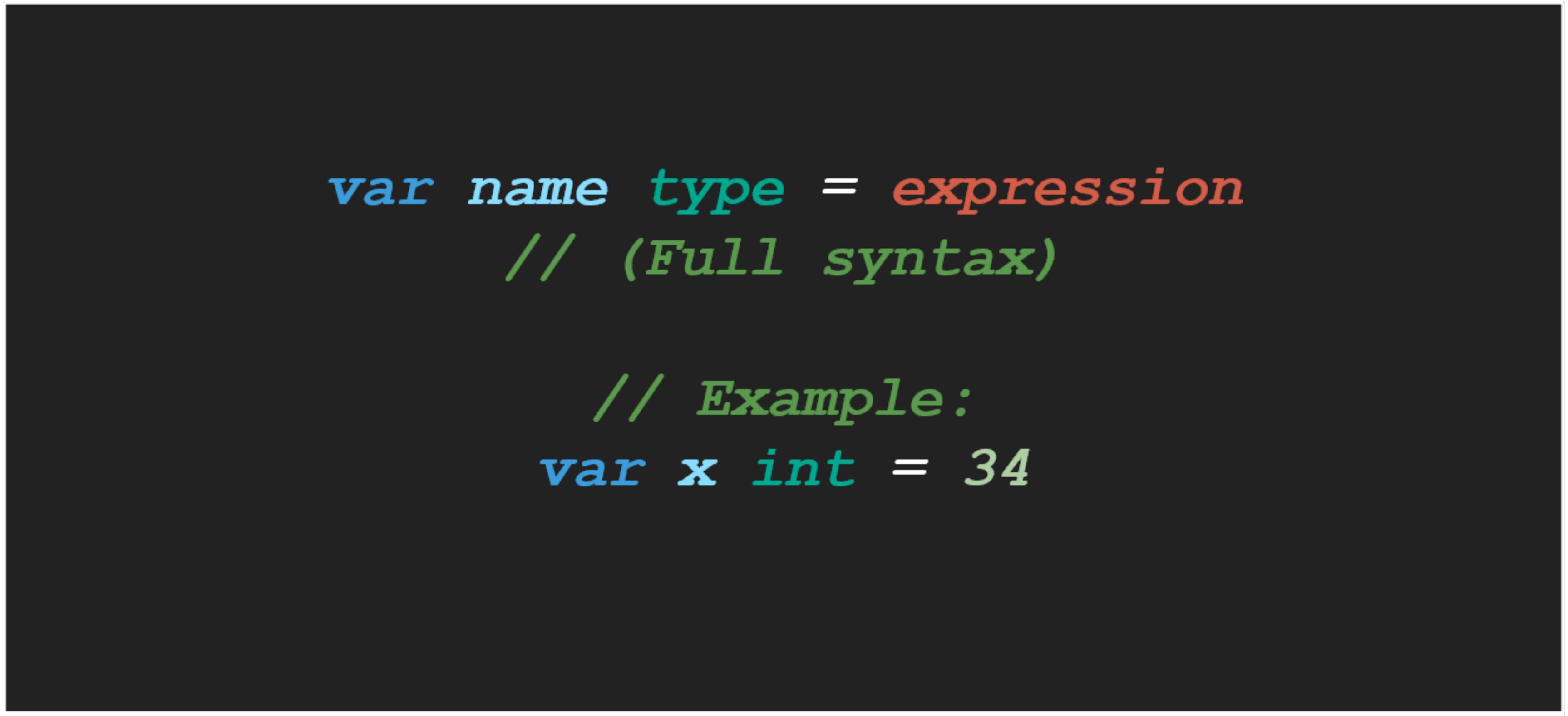
Constants	true, false, iota, nil
Types	int, int8, int16, int32, int64, uint, uint8, uint16, uint32, uint64, uintptr, float32, float64, complex128, complex64, bool, byte, rune, string, error
Functions	make, len, cap, new, append, copy, close, delete, complex, real, imag, panic, recover

Because in general standart function, types and constants is not a language keywords, you can redeclare them, BUT NEVER DO THIS! :)

Variables



Declare a variable in or out of the function



```go
var name type = expression
// (Full syntax)

// Example:
var x int = 34
```

Declare a variable in or out of the function

var name type

(Omit the value, default will be used)

Example:

var x int

(value will be 0)

Declare a variable in or out of the function

```
var name = expression
```

(Omit the type, type will be computed)

Example:

```
var x = 5
```

(type will be int)

Declare a variable in or out of the function

You can group declaration into one block

Example:

```
var (
    x1 int = 5
    x2 = 8 // type will be the same as x1
)
```

Declare a variable in the function

```
name := expression
```

(short variable declaration, type will be computed)
Can be used only inside a function/method

Example:

```
x := 5
```

(type will int)

Naming conventions

- *Begins with a letter (Unicode) or underscore (_);*
- *May have letters, digits, underscores;*
- *Case matters: “goLang” and “GoLang” are different names.*

- *Use CamelCase for word separation*
- *Long of the variable name should be connection with the scope of the variable (less scope, less characters in name preferred)*

Validname, valid_name, _validname, _v_a_l, valid_123

123, invalid-name, invalid!, 2e

Variables initialization

```
func main() {
    var (
        b1  bool
        s1  string
        i1  int
        ui1 uint
        by1 byte
        r1  rune
        f1  float32
        c1  complex64
    )

    fmt.Println("Boolean - ", b1)
    fmt.Printf("String - %q\n", s1)
    fmt.Println("Integer - ", i1)
    fmt.Println("Unsigned Integer - ", ui1)
    fmt.Println("Byte - ", by1)
    fmt.Println("Rune - ", r1)
    fmt.Println("Float number - ", f1)
    fmt.Println("Complex number - ", c1)
}
```

```
Boolean -  false
String - ""
Integer -  0
Unsigned Integer -  0
Byte -  0
Rune -  0
Float number -  0
Complex number -  (0+0i)
```

https://goplay.tools/snippet/LUmySkremfm

Every literal has it’s own default type

```go
package main

import (
    "fmt"
)

func main() {
    s := ""
    c := 'b'
    i := 0
    f := 0.0

    fmt.Printf("String literal – %T\n", s)
    fmt.Printf("Integer literal – %T\n", i)
    fmt.Printf("Character literal – %T\n", c)
    fmt.Printf("Floating number literal – %T\n", f)
}
```

```
String literal – string
Integer literal – int
Character literal - int32
Floating number literal - float64
```

https://goplay.tools/snippet/xnqb_jHuyNr

Variables in Golang key points

- *Every variable is initialized with a specified or default value.*
- *Each type has its default value.*
- *Every literal has its default type.*
- *You cannot compare numbers of different types, for example int and int8 is completely different types from Golang prospective, you cannot compare or reassign variables with different types.*

Questions

Basic types

Basic types

```
bool

string

int  int8  int16  int32  int64
uint uint8 uint16 uint32 uint64 uintptr

byte // alias for uint8

rune // alias for int32
       // represents a Unicode code point

float32 float64

complex64 complex128
```

Boolean type

- package main
-
  import (
-     "fmt"
-)
-
  func main() {
-     // var truthy = true
-     var truthy bool = 14 > 12
-     // var falsy = false
-     var falsy bool = 14 < 12
-
      fmt.Println(truthy)
-     fmt.Println(falsy)
- }

```
true
false
```

Boolean is very simple type, in Golang it is just a constant which hold result of two expressions.

```
const (
	true  = 0 == 0 // Untyped bool.
	false = 0 != 0 // Untyped bool.
)
```

https://goplay.tools/snippet/b6yluuxtOWk

Logical operators

A	B	!A	A||B	A&&B
false	false	true	false	false
false	true	true	true	false
true	false	false	true	false
true	true	false	true	true

Numeric types

Signed	***int, int8, int16, int32, int64***
Unsigned	***uint, uint8, uint16, uint32, uint64***
Special	***rune (int32), byte (uint8), uintptr***
System dependent	***int, uint, uintptr***

Signed integer

```
• func main() {
•     var (
•         a     int = -1
•         aSize     = unsafe.Sizeof(a) * 8
•
          b     int32 = 3
•         bSize       = unsafe.Sizeof(b) * 8
•
          c     int64 = 5
•         cSize       = unsafe.Sizeof(c) * 8
•     )
•
      fmt.Println(a, b, c)
•     fmt.Println(aSize, bSize, cSize)
• }
```

```
-1 3 5
64 32 64
```

https://goplay.tools/snippet/JtZ2_7tT0i-

Machine dependent types:
int – depending on architecture, take 32/64 bit

Machine un dependent types:
int8, int16, int32, int64 – take 8/16/32/64 bit independently on architecture.

Unsigned integer

```go
func main() {
    var (
        a     uint   = 1
        aSize        = unsafe.Sizeof(a) * 8

        b     uint32 = 3
        bSize        = unsafe.Sizeof(b) * 8

        c     uint64 = 5
        cSize        = unsafe.Sizeof(c) * 8
    )

    fmt.Println(a, b, c)
    fmt.Println(bSize, aSize, cSize)
}
```

```
1 3 5
32 64 64
```

https://goplay.tools/snippet/vgzxy6VNukc

Machine dependent types:

- *uint – depending on architecture, take 32/64 bit*

Machine un dependent types:

- *uint8, uint16, uint32, uint64 – take 8/16/32/64 bit independently on architecture.*

Compare integers

```go
func main() {
    var (
        a int
        b int32
        c int64
    )

    fmt.Println(a == b)
    fmt.Println(b == c)
}
```

```
./prog.go:14:16: invalid operation: a == b
  (mismatched types int and int32)

./prog.go:15:16: invalid operation: a == c
  (mismatched types int and int64)
```

https://goplay.tools/snippet/ioHcc7Jzy-H

You cannot compare different types of integers, for example int and int32.

Compare integers

```
func main() {
    var (
        a int
        b int32
        c int64
    )

    fmt.Println(a == int(b))
    fmt.Println(int64(b) == c)
}
```

```
true
true
```

https://goplay.tools/snippet/CypY4iHozG8

To compare two different types, we can use type casting.

Integer binary operators

1	*	/	%	<<	>>	&	&^
2	+	-	|	^			
3	==	!=	<	<=	>	>=	
4	&&						
5	||						

Increment/Decrement

```go
func main() {
    var a int
    a++
    fmt.Println(a)

    a--
    fmt.Println(a)

    b := int8(math.MaxInt8)
    b++
    fmt.Println(b)

    c := int16(math.MaxInt16)
    c++
    fmt.Println(c)

    d := int64(math.MaxInt64)
    d++
    fmt.Println(d)
}
```

```
1
0
-128
-32768
-9223372036854775808
```

https://goplay.tools/snippet/NBDiiSUtzii

Increment and decrement operation in Golang exists only in a suffix-based form, you can write only “i++” not “++i”, as well increment operation is mutate value and doesn’t return value of the operation, due to this “j := i++” is invalid operation, as I mentioned before Golang is explicit language, you need to write exactly what to expect, due to this some common constructions is forbidden.

Floating-point numbers

float32	*float64*

Floating point numbers

```
• func main() {
•     var f64 float64
•     f64++
•     fmt.Println(f64)
•
      var f32 float32
•     f32++
•     fmt.Println(f32)
•
      //invalid operation: f32 == f
  64 (mismatched types float32 and
  float64)
•     //fmt.Println(f32 == f64)
•
      fmt.Println(float64(f32) == f
  64)
• }
```

```
1
1
true
```

https://goplay.tools/snippet/c07pmPGQMkW

In Golang there is no double and float type, there is only float32 and float64

Float32 occupies 32 bits in memory and stores values in single-precision floating point format.

Float64 occupies 64 bits in memory and stores values in double-precision floating point format.

String type

byte	'A'	ASCI character, 1 byte size
rune	'ó'	UTF-8 character, up to 4 byte size
string	"Kraków"	String, a bunch of UTF-8 encoding charactes.

Strings

```go
package main

import (
    "fmt"
)

func main() {
    msg := "Dzień dobry"
    //msg is a string with 11 characters
    fmt.Println(len(msg)) //12
    // Why we have 12 here ?
}
```

Dzień dobry

[*https://goplay.tools/snippet/7yGc4jBf7uQ*](https://goplay.tools/snippet/7yGc4jBf7uQ)

Why len(msg) returns 12 when we have 11 characters? There answer is quite simple, len returns value in bytes, and character "ń" cost us 2 bytes in UTF-8 encoding (non-ASCI character).

Converting the string

```go
package main

import (
    "fmt"
)

func main() {
    msg := "Dzień dobry"
    //msg is a string with 11 characters
    fmt.Println(len(msg)) //12
    // Why we have 12 here ?
}
```

Dzień dobry

https://goplay.tools/snippet/7yGc4jBf7uQ

Why len(msg) returns 12 when we have 11 characters? There answer is quite simple, len returns value in bytes, and character "ń" cost us 2 bytes in UTF-8 encoding (non-ASCI character).

Converting the string

We will speak more about strings on the next lection :)

And about types on next and next after the next :-)

Default values

Type	Default value
number	0
bool	false
string	""
Interface, slice, pointer, map, channel, function	nil

Questions

Constants

Constant declarations

```
const name type = expression
```

(Full syntax, strict type)

Example:

```
const x int = 34
```

Constant declarations

```
const name = expression
```

(Omit the type, type will be computed from expression)

Example:

```
const x = 5
```

(type will be untyped integer)

(untyped integer)

Naming conventions

- *Begins with a letter (Unicode) or underscore (_);*
- *May have letters, digits, underscores;*
- *Case matters: “goLang” and “GoLang” are different names.*

- *Use CamelCase for word separation*

Validname, valid_name, _validname, _v_a_l, valid_123

123, invalid-name, invalid!, 2e

Constants in Golang key points

- *Every constant must have initialization expression, there no default value for constants.*
- *Every constant vithout specified strict type has more weak rules in terms of types system, you can compare number constant with another constant or variable of any numbered type or aliased type created from numbered, the same for strings.*
- *Every constant can have strict type, in this case you cannot compare, assign them to variables of other type.*

Constants

```go
const name string = "Go"
const (
    e  = 2.7182
    pi = 3.1415
)
const (
    a = 1.1
    b
    c = 2
    d
)

func main() {
    fmt.Println(name)
    name := "Java"
    fmt.Println(name)

    fmt.Println(a, b, c, d)
    fmt.Printf("%T %T\n", b, d
)
}
```

```
Go
Java
1.1 1.1 2 2
float64 int
```

https://goplay.tools/snippet/O-po3ANlo6u

Constants in Golang are variables that has in place initialization and that value cannot be changed after first initialization. (One exception, you can in smaller scope define variable with the same name)

*Constants have a little bit different type system, strict type of the constant computed when it used, it means that you can compare constant **e** with float32 and float64 type, as well as constant **b** can be used without type casting with any numbered type. The same applicable with type aliases(we will talk about it later.)*

Iota and constants

```
type Weekday int

const (
    Monday    Weekday = iota
    Tuesday           = iota
    Wednesday         = iota
)

const (
    a = iota * 2
    _
    b
    c
)

func main() {
    fmt.Println(Monday, Tuesda
y, Wednesday)
    fmt.Println(a, b, c)
}
```

```
0 1 2
0 4 6
```

https://goplay.tools/snippet/6DqDZZH1SCi

Iota is a constant with special behavior, it is kind of generator that return unique sequential number for constant block.

- *Value of iota not shared across different constants blocks, but*
- *You can change value of iota using standard operations as +, *, -.*
- *In case you want to skip value you can use blank identifier _*

Type system variables vs constants

VARIABLES

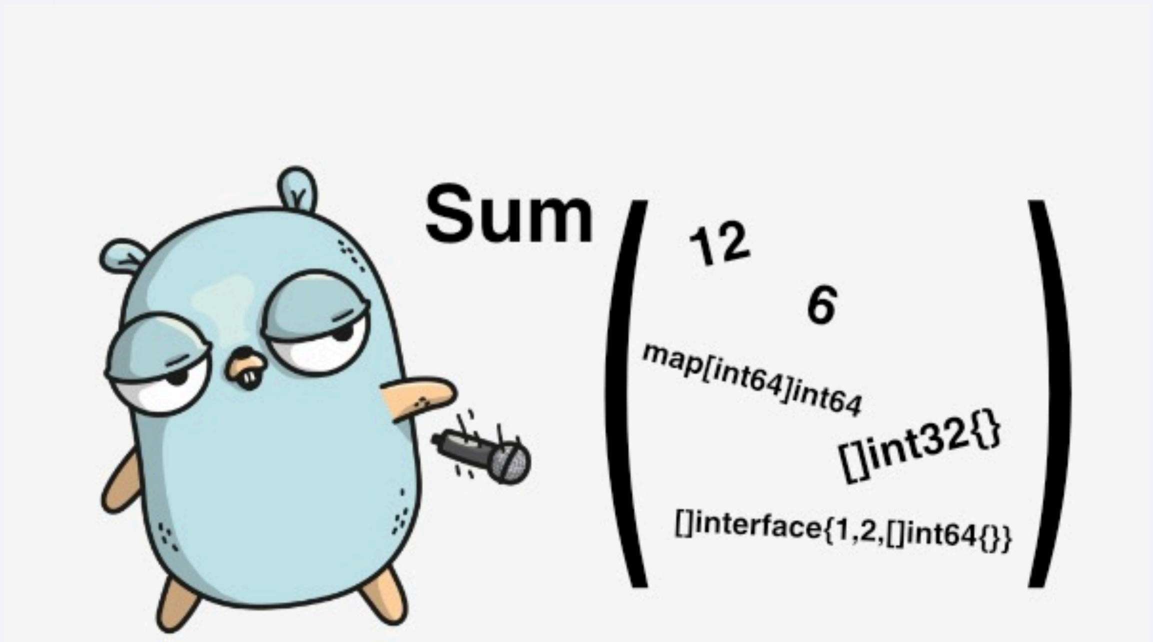
- *Variables has strict type system, int8 and int32 for example is different type and can't be used interchangeably.*

CONSTANTS

- *Constant has weak type system, for example string constant has type untyped string constant and numbers untyped integer, final type computed when constant is used, this allows to use constants with any castable type.*

Questions

Functions



Function declarations

```
func name(parameters) (results) {
    //body
}
```

Function types

```go
package main

import "fmt"

func add(x int, y int) int { return x + y }
func sub(x, y int) (z int) { z = x - y; return }
func first(x int, _ int) int { return x }
func zero(int, int) int { return 0 }

func main() {
	fmt.Printf("%T\n", add)
	fmt.Printf("%T\n", sub)
	fmt.Printf("%T\n", first)
	fmt.Printf("%T\n", zero)
}
```

```
func(int, int) int
func(int, int) int
func(int, int) int
func(int, int) int
```

https://goplay.tools/snippet/cpbj8m-40rw

Function in Golang has their own type.

Function name must be unique in the scope, you cannot override function.

Recursion

```
package main

import "fmt"

func factorial(i uint) uint {
    if i == 0 {
        return 1
    }
    return i * factorial(i-1)
}

func main() {
    fmt.Println(factorial(5))
}
```

```
120
```

https://goplay.tools/snippet/Re_kjE0PHAl

Multiple return values

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println(devide(10, 3))
}

func devide(a, b int) (int, int) {
    return a / b, a % b
}
```

```
3 1
```

https://goplay.tools/snippet/fTlb7HpuDyi

Functions in Golang can have multiple return parameters, it gives an ability to have pretty elegant solutions, but be conservative with this feature, large number or return statement usually signal that you break single responsibility principle.

Variadic parameter

```go
func main() {
    digits := []int{1, 2, 3, 4}

    //You cannot just pass an array, you have to expand it
    //fmt.Println(sum(digits)) // Will not work
    fmt.Println(sum(digits...))
}

//variadic parameter must be last parameter of the function
func sum(digits ...int) int {
    var sum int
    for _, d := range digits {
        sum += d
    }

    return sum
}
```

```
10
```

https://goplay.tools/snippet/Wobv8_U_y9d

You need to keep in mind a few things:

- Variadic parameter is a slice under the hood (slice is a dynamic array in Golan, about this type later)
- Variadic parameter must be last in a function
- You need to expand the slice to pass it as a variadic parameter with "..."

Change underlined slice in variadic parameter

```go
func main() {
    strs := []string{
        "Hello,", "My", "name",
        "is", "Alex",
    }
    mutate(strs...)

    fmt.Println(strings.Join(strs, " "))
}

func mutate(x ...string) {
    x[len(x)-1] = "Michał"
}
```

```
Hello, My name is Michal
```

https://goplay.tools/snippet/6imaJjAAB0A

You need to keep in mind a few things:

- "..." doesn't copy slice due to this you can change slice inside a variadic function

High-order functions

```go
package main

import "fmt"

func logExecution(f func()) {
    fmt.Println("Start execution")
    f()
    fmt.Println("Executed")
}

func main() {
    hello := func() { fmt.Println("Hello world!!!") }
    logExecution(hello)
}
```

```
Start execution
Hello world!!!
Executed
```

https://goplay.tools/snippet/ue1Ikd67GCp

What we just did?

- *Assign a function to a variable*
- *Pass a function to the function as a parameter*

High-order functions

```go
package main

import "fmt"

func newWriter() func(string) {
    return func(s string) {
        fmt.Println(s)
    }
}
func main() {
    var writer func(string) = ne
wWriter()
    writer("Hello Poland!")
    writer("Cześć!")
}
```

```
Hello Poland!
Cześć!
```

https://goplay.tools/snippet/4y54T8g5qst

What we just did?

- *Assign a function to a variable*
- *Return a function from the function*
- *Define anonymous function (function without name)*

Deferred functions

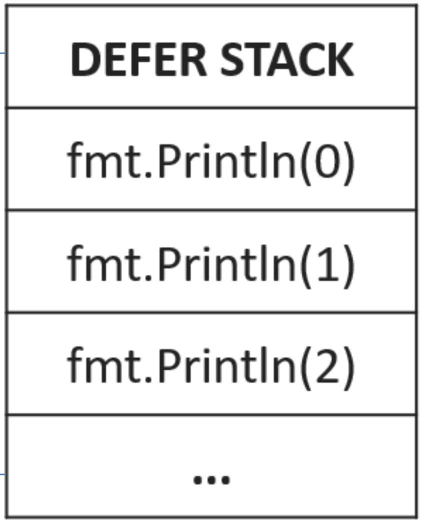
```go
package main

import (
	"fmt"
)

func main() {
	var message string = "I w
ill be executed after exit fr
om the function"

	defer func() {
		fmt.Println(message)
	}()

	fmt.Println("Exit from fu
nction")
}
```

```
Exit from function

I will be executed after exit from
the function
```

https://goplay.tools/snippet/AM29trKSYZK

Deferred functions execution order

```
package main

import "fmt"

func main() {
    for i := 0; i < 5; i++ {
        defer fmt.Println(i)
    }

    // deferred funcs run here
}
```



```
4
3
2
1
0
```

https://goplay.tools/snippet/ePn-bHg2S-X

Deferred functions

```go
package main

import (
	"fmt"
)

func main() {
	for i := 0; i < 10; i++ {
		defer func() {
			fmt.Print(i, " ")
		}()
	}
}
```

```
10 10 10 10 10 10 10 10 10 10
```

https://goplay.tools/snippet/URnuA-CzSE_V

Deferred functions use cases

```
row, err := db.Query(`SELECT ...`)
if err != nil {
        // handle error or path it to the
caller
}
defer row.Close()
```

```
defer func() {
    if err := recover(); err != nil {
        ...
    }
}()

panic("oops!")
```

https://goplay.tools/snippet/dEShiFiGuCO

Function in Golang key points

- *Function is just a bunch of operationgs grouped in a logical way.*
- *You can assign function to variable, pass it as a parameter to the function or return from the function.*
- *Functions in Golang has type as well, type of the function depending on its parameters and return statements.*

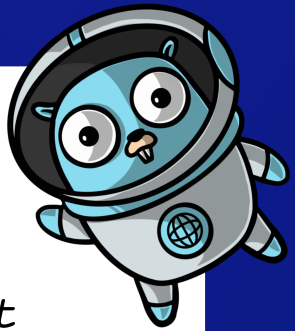
Questions

Flow control

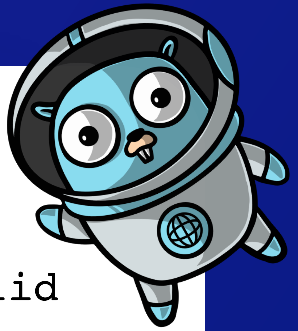
If – else statement

```
if condition {

    ...

} else if condition{

    ...

} else {

    ...
}
```

If else syntax is common for most languages, the only difference is that you don’t need parentheses around conditions, but the braces are required.



If - else



```
func main() {
    for i := 0; i <= 7; i++ {
        if weekday, err := isWeekDay(i); er
r != nil {
            fmt.Println("unexpected error –
", err)
        } else {
            fmt.Println(i, "–", weekday)
        }
    }
}

func isWeekDay(d int) (bool, error) {
    if d <= 0 || d > 7 {
        return false, fmt.Errorf("invalid v
alue, valid range is [1-7]")
    } else if d > 5 {
        return false, nil
    } else {
        return true, nil
    }
}
```

```
unexpected error - invalid value, valid
range is [1-7]
1 – true
2 – true
3 – true
4 – true
5 – true
6 – false
7 - false
```

https://goplay.tools/snippet/V27mHXgTT42

Switch statement

```go
func main() {
    fmt.Println(numberToWeekDay(1))
    fmt.Println(numberToWeekDay(3))
}
func numberToWeekDay(i int) string {
    switch i {
    case 1:
        return "Monday"
    case 2:
        return "Tuesday"
    case 3:
        return "Wednesday"
    case 4:
        return "Thursday"
    case 5:
        return "Friday"
    case 6:
        return "Saturday"
    case 7:
        return "Sunday"
    default:
        return "unknown"
    }
}
```

```
Monday
Wednesday
```

https://goplay.tools/snippet/YMHG82I9nFN

*Switch statement in Golang has pretty common syntax, one important difference is that instead of **falltrough** behavior, Golang **breaks** after each case.*

Switch statement

```
• func main() {
•     fmt.Println(isWeekDay(1))
•
      fmt.Println(isWeekDay(6))
•
      fmt.Println(isWeekDay(10))
• }
•
  func isWeekDay(i int) (bool, error) {
•     switch i {
•     case 1, 2, 3, 4, 5:
•         return true, nil
•     case 6, 7:
•         return false, nil
•     default:
•         return false, fmt.Errorf("invalid v
  alue, valid range [1-7]")
•     }
• }
```

```
true <nil>

false <nil>

false invalid value, valid range [1-7]
```

https://goplay.tools/snippet/SMVOi9WLJag

In case you have the same behaviour for multiple values, you can specify multiple values in one case.

Switch statement

```go
func main() {
    fmt.Println(isWeekend(6))
    fmt.Println(isWeekend(5))
    fmt.Println(isWeekend(10))
}
func isWeekend(i int) (bool, error) {
    switch {
    case i >= 6 && i <= 7:
        return true, nil
    case i >= 1 && i <= 5:
        return false, nil
    default:
        return false, fmt.Errorf("invalid value, valid range [1-7]")
    }
}
```

```
true <nil>
false <nil>
false invalid value, valid range [1-7]
```

https://goplay.tools/snippet/xENCYSAAXeu

You can implement all use-cases of if-else statement using switch statement, in case of absence of variable in switch you can just specify expression with boolean result.

Questions

Loops

Loops (Plain old for loop)

```
package main

import (
    "fmt"
)

func main() {
    for i := 0; i < 10; i++ {
        fmt.Print(i, " ")
    }
}
```

```
0 1 2 3 4 5 6 7 8 9
```

https://goplay.tools/snippet/KEBAFKZvwvx

Plain for loop syntax is common for most languages, the only difference is that you don't need parentheses around conditions, but the braces are required.

Loops (While loop)

```go
package main

import (
	"fmt"
)

func main() {
	var i int
	for i < 10 {
		fmt.Print(i, " ")
		i++
	}
}
```

```
0 1 2 3 4 5 6 7 8 9
```

https://goplay.tools/snippet/Fbp11kKVQw4

In Golang while loop is special type of for loop.

Loops (Infinite loop)

```go
package main

import (
    "fmt"
)

func main() {
    for {
        fmt.Println("Hello from infinite loop")
    }
}
```

```
Hello from infinite loop
Hello from infinite loop
Hello from infinite loop
Hello from infinite loop
Hello from infinite loop
...
```

https://goplay.tools/snippet/AhPisV44upd

Empty values works as infinite loop

Loops (For each)

```go
func main() {
    strs := []string{
        "Hello",
        "from",
        "for-each",
        "loop",
        "!",
     }

    for _, s := range strs {
        fmt.Print(s, " ")
    }
}
```

```
Hello from for-each loop !
```

https://goplay.tools/snippet/JvDY6ZHkrUY

For range loop is helpful do go over collections, range key word used, first parameter returned by range is index of the element, second parameter is actual value.

Loops (continue, break)

```go
func main() {
    strs := []string{
        "Hello",
        "from",
        "for-each",
        "loop",
        "!",
    }

    for _, s := range strs {
        if s == "for-each" {
            continue
        } else if s == "!" {
            break
        }
        fmt.Print(s, " ")
    }
}
```

```
Hello from loop
```

https://goplay.tools/snippet/u6f1sHcxdbM

You can use break and continue operators, to control loop behavior.

Continue stop current iteration and start next one.

Break completely stop loop.

What should you remember?

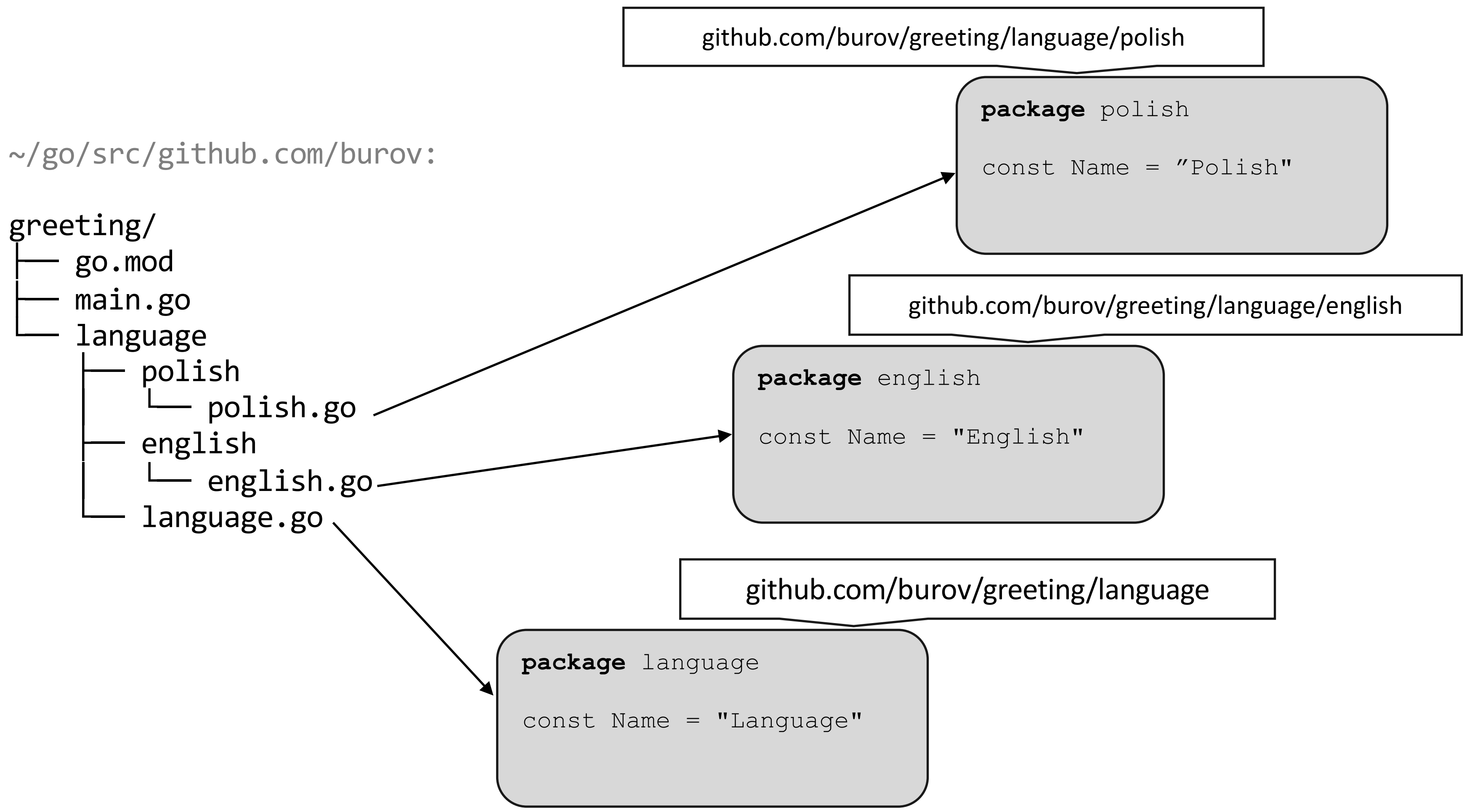
- In Golang there is only **for** loop, no **while** or **do-while.**
- Loop variable initialized only once; you need to copy it before using in closures or path to a function in case it reference type.
- For loop contains three parts, variable initialization, condition, post statement any of this part can be skipped or extended with different conditions.

Packages

Golang packages

```
~/go/src/github.com/burov:

greeting/
├── go.mod
├── main.go
└── language
    ├── polish
    │   └── polish.go
    ├── english
    │   └── english.go
    └── language.go
```

Golang packages



```
~/go/src/github.com/burov:

greeting/
├── go.mod
├── main.go
└── language
    ├── polish
    │   └── polish.go
    ├── english
    │   └── english.go
    └── language.go
```

github.com/burov/greeting/language/polish

```go
package polish

const Name = ”Polish"
```

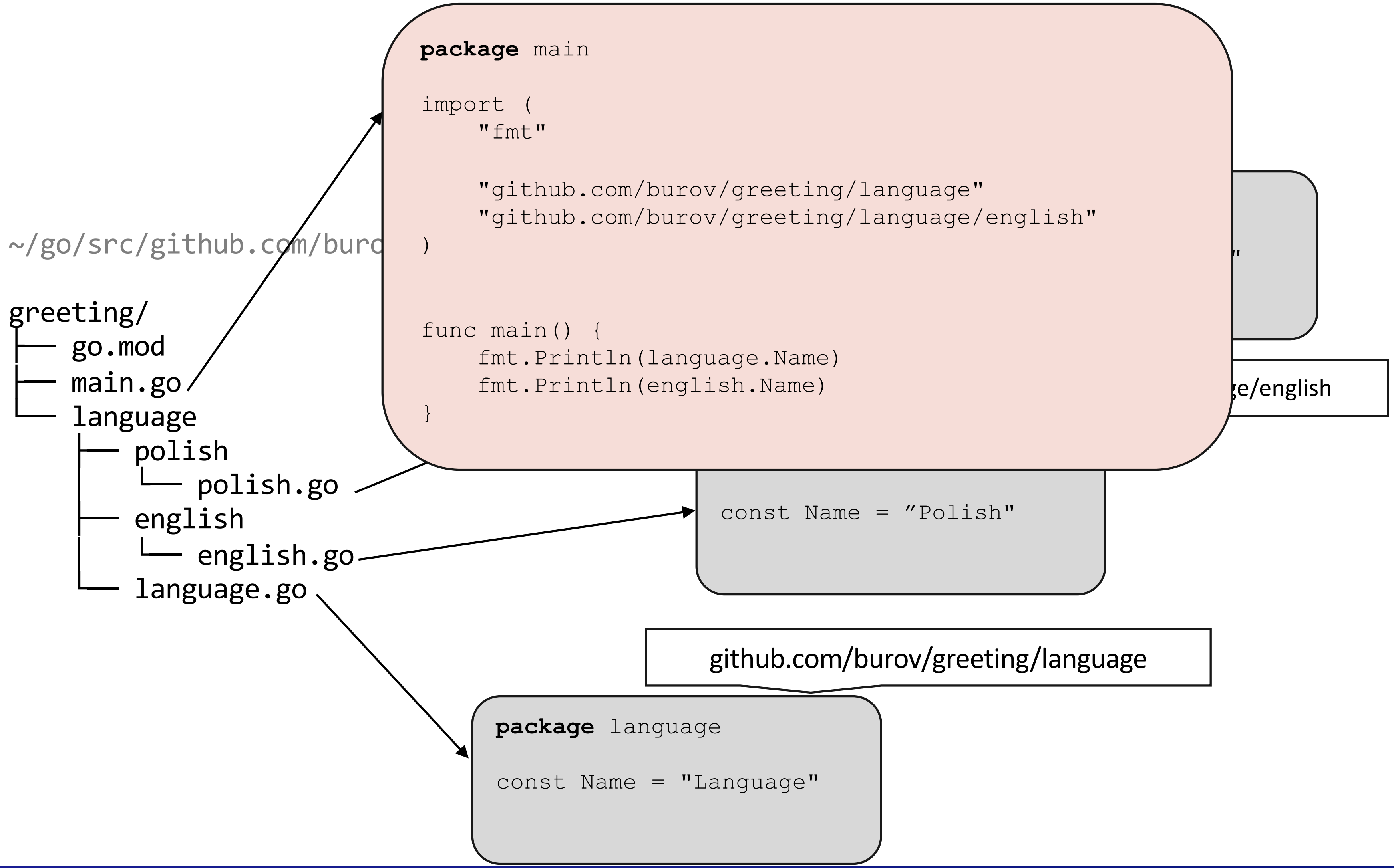
github.com/burov/greeting/language/english

```go
package english

const Name = "English"
```

github.com/burov/greeting/language

```go
package language

const Name = "Language"
```

Golang packages



```
~/go/src/github.com/buro

greeting/
├── go.mod
├── main.go
└── language
    ├── polish
    │   └── polish.go
    ├── english
    │   └── english.go
    └── language.go
```

```go
package main

import (
	"fmt"

	"github.com/burov/greeting/language"
	"github.com/burov/greeting/language/english"
)


func main() {
	fmt.Println(language.Name)
	fmt.Println(english.Name)
}
```

```go
const Name = ”Polish"
```

```go
package language

const Name = "Language"
```

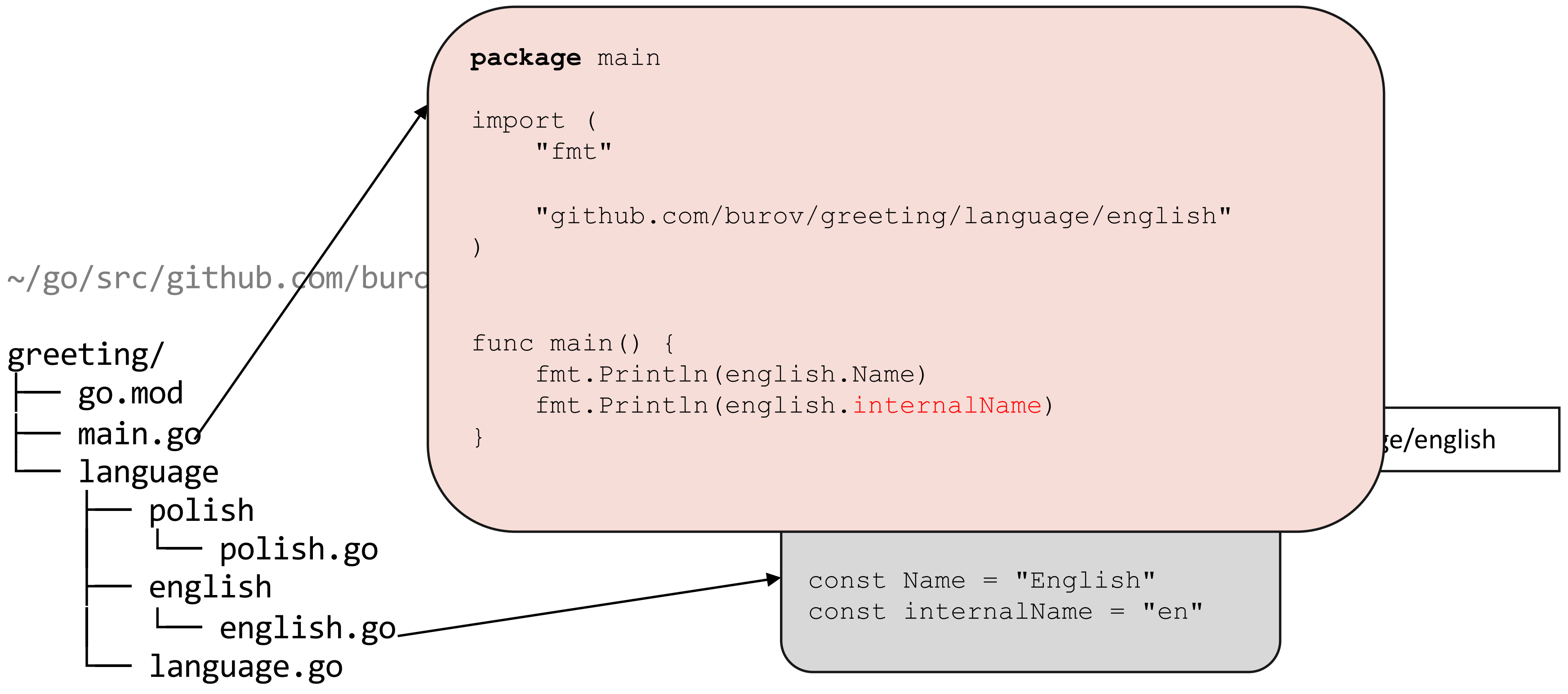
Exported/unexported objects

```
~/go/src/github.com/burov:

greeting/
├── go.mod
├── main.go
└── language
    ├── polish
    │   └── polish.go
    ├── english
    │   └── english.go
    └── language.go
```

github.com/burov/greeting/language/english

```go
package english

const Name = "English"
const internalName = "eng"
```

Exported/unexported objects



~/go/src/github.com/burc

```
greeting/
├── go.mod
├── main.go
└── language
    ├── polish
    │   └── polish.go
    ├── english
    │   └── english.go
    └── language.go
```

```go
package main

import (
    "fmt"

    "github.com/burov/greeting/language/english"
)


func main() {
    fmt.Println(english.Name)
    fmt.Println(english.internalName)
}
```

e/english

```go
const Name = "English"
const internalName = "en"
```

Exported/unexported objects

```
package main

import (
	"fmt"

	"github.com/burov/greeting/language/polish"
)


func main() {
	fmt.Println(polish.Name)
	fmt.Println(polish.internalName)
}
```

~/go/src/github.com/bur

```
greeting/
├── go.mod
├── main.go
└── language
    ├── polish
    │   └── polish.go
    ├── english
    │   └── english.go
    └── language.go
```

```
const Name = ”Polish"
const internalName = ”pl"
```

```
./main.go:10:14: cannot refer to unexported name polish.internalName
./main.go:10:14: undefined: polish.internalName
```

Packages example

```go
package main

import (
	"fmt"

	"github.com/burov/greeting/language/english"
	"github.com/burov/greeting/language/polish"
)

func main() {
	fmt.Println(english.Name)
	fmt.Println(polish.Name)
}
```

English

Polish

https://goplay.tools/snippet/5TtAEQg-ZN9

To import package, you need to use module name + package or if you use $GOPATH just a relative path from $GOPATH/src + package

Every variable/function/constant/structure named from upper-case letter is exported (you can use them from other packages), from lower-case letter is package private only.

Packages example

```
package main
import (
    "fmt"
    "github.com/burov/greeting/language/english"
    "github.com/burov/greeting/language/polish"
)
func main() {
    fmt.Println(english.Name)
    fmt.Println(polish.Name)
    //fmt.Println(english.internalName)
    //cannot refer to unexported name english.internalName
    //fmt.Println(polish.internalName)
    //cannot refer to unexported name polish.internalName
}
```

```
English
Polish
```

https://goplay.tools/snippet/GSHnaLmADtH

To import package, you need to use module name + package or if you use $GOPATH just a relative path from $GOPATH/src + package

Every variable/function/constant/structure named from upper-case letter is exported (you can use them from other packages), from lower-case letter is package private only.

Summary about packages

- *Package is just a folder with files, in one folder you can have only one package.*
- *Every file start from package definition, as a good practice name of the package is equals to folder name.*
- *Main package is a special package that is entry point of the code*
- *Every variable/function/constant/structure named from upper-case letter is exported and can be accessed outside of the project.*
- *Every variable/function/constant/structure named from lower-case letter is package private and cannot be accessed outside of the project.*

Questions

Homework

Homework “Square task”

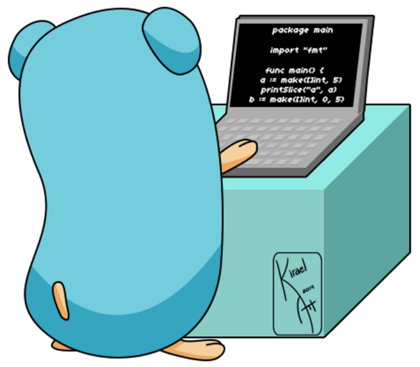
How to:

- Clone the repo
- run go mod init somename
- run go mod tidy
- Edit solution.go
  - it contains correct package name
  - follow comments placeholder

Tasks:

Implement function to calculate square of an equilateral figurine following rules:

- func CalcSquare(sideLen float64, sidesNum intCustomType) float64
- CalcSquare func must return correct square for:
  - equilateral triangle(3 sides),
  - square(4 sides)
  - circle(0 sides) (count sideLen as radius)
  - if any other sideNum param is passed, return 0
- built-in Pi constant must be used to bypass the test



```
package main

import "fmt"

func main() {
  a := make([]int, 5)
  printSlice("a", a)
b := make([]int, 0, 5)
```

Thanks