



# Golang Programming

Program structure, data types, operators, control-flow statements, functions

# Where to Find The Code and Materials?

<https://github.com/iproduct/coursego>

# Names

- The names of **Go functions, variables, constants, types, statement labels, and packages** begin with a letter (Unicode letter) or underscore, and may have a number of additional letters, digits, or underscores.
- Names are **case-sensitive**: `findWinner` and `Findwinner` are different names.
- Scopes: **local** (declared within a function) and **global** (if declared outside of a function - visible to all files in a package)
- **Exported** (visible outside of the package) and **unexported** (package-local)
  - if the first letter is uppercased, it is exported, otherwise not. Ex: `fmt.Printf()`
- Short names are preferred.
- "Camel case" preferred when combining words. Ex: `QuoteRuneToASCII` not `quote_rune_to_ASCII`

# Reserved Keywords

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

# Keyword Categories

- `const`, `func`, `import`, `package`, `type` and `var` are used to declare all kinds of code elements in Go programs.
- `chan`, `interface`, `map` and `struct` are used as parts in some composite type denotations.
- `break`, `case`, `continue`, `default`, `else`, `fallthrough`, `for`, `goto`, `if`, `range`, `return`, `select` and `switch` are used to control flow of code.
- `defer` and `go` are also control flow keywords, but in other specific manners.

# Predefined 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`

# Built-in Types

- One boolean type: `bool`.
- 11 built-in integer numeric types:  
`int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `int`, `uint`, and `uintptr`.
- 2 floating-point numeric types: `float32` and `float64`.
- 2 built-in complex numeric types: `complex64` and `complex128`.
- One Unicode code point type (alias for `int32`): `rune`
- One built-in (immutable) string type: `string`.

# Declarations

- Declarations: `var` , `const` , `type` , and `func`
- A Go program is stored in `one or more files` whose names end in `.go` .
- The `package` declaration states that files belong to the same package
- `Packages` (e.g. `package strings`), and `commands` (`package main`)
- Followed by `import` declarations, each including `import path` – package name is by default the last segment in import path.  
Ex: `import "github.com/user/hello"` -> package name is `hello`
- Followed by sequence of package-level declarations of `constants`, `variables`, `types`, and `functions`, in any order.

# The Structure of a Go Source File

Go code is arranged in **packages**, which fill the roles of both libraries and header files in C

```
package main
```

Every program must contain a **main** package, which contains a **main()** function, which is the program entry point

```
import "fmt"
```

**fmt** package has been imported, any of its exported **types**, **variables**, **constants**, and **functions** can be used, prefixed by the package name; packages are imported when the code is linked, rather than when it is run; **access control** in Go is available only at package level.

```
func main() {
```

```
    fmt.Println("Hello, world!")
```

```
}
```

**Println()** exported (public) function prints the text on the console

# Creating Simple Library Package

```
// Package stringutil contains utility functions for working with strings.  
package stringutil  
  
// Reverse returns its argument string reversed rune-wise left to right.  
func Reverse(s string) string {  
    r := []rune(s)  
    for i, j := 0, len(r)-1; i < len(r)/2; i, j = i+1, j-1 {  
        r[i], r[j] = r[j], r[i]  
    }  
    return string(r)  
}
```

# Using It

```
package main

import "fmt"
import "github.com/iproduct/coursegopro/01-intro/stringutil"

func main() {
    s := "Hello Go World!"
    fmt.Println(s)
    fmt.Println(stringutil.Reverse(s))
}
```

# Variables

```
var global int = 50

var (
    home    = os.Getenv("HOME")
    user    = os.Getenv("USER")
    gopath = os.Getenv("GOPATH")
)

func init() {
    global = 12

    // gopath may be overridden by --gopath flag on command line.
    flag.StringVar(&gopath, "gopath", gopath, "override default GOPATH")
    if gopath == "" {
        gopath = "c:/coursego/workspace"
        log.Printf("GOPATH not set - using default: %s", gopath)
    }
}

func main() {
    l, n := 5, 12 \\ or var l, n int = 5, 12
    fmt.Printf("GOPATH=%v\nGlobal:%v\nLocal:%v, %v\n", global, gopath, l, n)
}
```

# Variables

```
var i, j int = 5, 9
```

```
i, j = j, i // swap values of i and j
```

```
f, err := os.Open(name)
if err != nil {
    return err
}
// ...use f...
f.Close()
```

# Value Literals - Integer

`0xF` // the hex form (starts with a "0x" or "0X")

`0XF`

`017` // the octal form (starts with a "0", "0o" or "00")

`0o17`

`0017`

`0b1111` // the binary form (starts with a "0b" or "0B")

`0B1111`

`15` // the decimal form (starts without a "0")

# Value Literals - Real

1.23

01.23 // == 1.23

.23

1. // A "e" or "E" starts the exponent part (10-based).

1.23e2 // == 123.0

123E2 // == 12300.0

123.E+2 // == 12300.0

1e-1 // == 0.1

.1e0 // == 0.1

0010e-2 // == 0.1

0e+5 // == 0.0

# Constants

```
type Role int

const (
    User Role = 1 << iota
    Manager
    Admin
    RoleMask = (1 << (iota)) - 1
)

func (r Role) String() string {
    switch r {
    case User:
        return "User"
    case Manager:
        return "Manager"
    case Admin:
        return "Admin"
    default:
        return "Invalid role"
    }
}
```

```
// Status type
type Status int

// User statuses enum
const (
    Registered Status = iota
    Active
    Disabled
)
// Returns string representation of the Role
func (r Status) String() string {
    switch r {
    case Registered:
        return "Registered"
    case Active:
        return "Active"
    case Disabled:
        return "Disabled"
    default:
        return "Invalid status"
    }
}
```

# Pointers

- A pointer value is the memory address of a variable. Memory addresses are often represented with hex integer literals, such as `0x1234CDEF`.
- Not every value has an address, but every variable does. With a pointer, we can access or update the value of a variable directly.
- If a variable is declared `var n int`, the expression `&n` ("address of `n`") has a type `*int`, pronounced as "pointer to int".
- The variable to which `p` points is denoted as `*p`, and can be used in the left or in the right hand side of an assignment. Ex:

```
n := 11
```

```
p := &n // p, of type *int, points to n  
fmt.Println(*p) // "11"
```

```
*p = 42 // equivalent to n = 42  
fmt.Println(n) // "42"
```

# Pointers

```
func zeroval(ival int) {
    ival = 0
}

func zeroptr(iptr *int) {
    *iptr = 0
}

func main() {
    i := 1
    fmt.Println("initial:", i)

    zeroval(i)
    fmt.Println("zeroval:", i)

    zeroptr(&i)
    fmt.Println("zeroptr:", i)

    fmt.Println("pointer:", &i)
}
```

Output:

```
initial: 1
zeroval: 1
zeroptr: 0
pointer: 0x42131100
```

# Stack or Heap?

```
var global *int
```

```
func f() {
```

```
    var x int
```

```
    x = 1
```

```
    global = &x
```

```
}
```

```
func g() {
```

```
    y := new(int)
```

```
*y = 1
```

```
}
```

# Stack or Heap?

```
var global *int
```

```
func f() {
```

```
    var x int
```

```
    x = 1
```

```
    global = &x
```

```
}
```

```
func g() {
```

```
    y := new(int)
```

```
    *y = 1
```

```
}
```

```
D:\CourseGO>go build -gcflags="-m -l"  
./main.go:8:6: moved to heap: x  
./main.go:14:10: new(int) does not escape  
./main.go:20:12: ... argument does not escape  
./main.go:20:19: *global escapes to heap
```

# Methods: Value and Pointer Receivers

```
type ByteSlice []byte

func (slice ByteSlice) Append(data []byte) []byte {
    return append([]byte(slice), data...)
}

func (slice *ByteSlice) AppendPointer(data []byte) {
    *slice = append([]byte(*slice), data...)
}

func (slice *ByteSlice) Write(data []byte) (n int, err error) {
    *slice = append([]byte(*slice), data...)
    return len(data), nil
}

func main() {
    var b ByteSlice
    fmt.Fprintf(&b, "This hour has %d days\n", 7)
    fmt.Printf("%v", b)
}
```

# Allocation with New

- The built-in function `new()` takes a type `T`, allocates storage for a variable of that type at run time, and returns a value of type `*T` pointing to it. The variable is initialized with a zero for that type. Usage: `new(T)`
- Example:

```
type S struct { a int; b float64 }  
new(S)
```

➤ allocates storage for a variable of type `S`, initializes it (`a=0, b=0.0`), and returns a value of type `*S` containing the address of the location.

# Making Slices, Maps and Channels

Call	Type T	Result
make(T, n)	slice	slice of type T with length n and capacity n
make(T, n, m)	slice	slice of type T with length n and capacity m
make(T)	map	map of type T
make(T, n)	map	map of type T with initial space for approximately n elements
make(T)	channel	unbuffered channel of type T
make(T, n)	channel	buffered channel of type T, buffer size n

# Arithmetic Operators [<https://www.geeksforgeeks.org/go-operators/>]

- + sum integers, floats, complex values, strings
- difference integers, floats, complex values
- \*
- / quotient
- % remainder integers
- & bitwise AND
- | bitwise OR
- ^ bitwise XOR
- &^ bit clear (AND NOT)
- << left shift integer << unsigned integer
- >> right shift integer >> unsigned integer
- ++ and -- are statements, not expressions (do not return anything), postfix

# Relational Operators

`--` equal comparable

`!=` not equal

`<` less integers, floats, strings

`<=` less or equal

`>` greater

`>=` greater or equal

# Logical Operators

`&&` conditional AND `p && q` means "if p then q else false"

`||` conditional OR `p || q` means "if p then true else q"

`!` NOT `!p` means "not p"

# Pointers and Channels

- & address of &x generates a pointer to x
- \* pointer indirection \*x denotes the variable pointed to by x
- <- receive <-ch is the value received from channel ch

# Type Conversion Operators

- Syntax: `Conversion = Type "(" Expression [ "," ] ")"`
- Examples:

```
uint(iota)           // iota value of type uint
float32(2.718281828) // 2.718281828 of type float32
complex128(1)        // 1.0 + 0.0i of type complex128
string(0x266c)       // "♪" of type string
string([]byte{'h', 'e', 'l', 'l', '\xc3', '\xb8'}) // "hellø"
[]byte("hellø")     // []byte{'h', 'e', 'l', 'l', '\xc3', '\xb8'}
string([]rune{0x767d, 0x9d6c}) // "\u767d\u9d6c" == "白鵬"
```

# Type Conversion Operators

```
MyString("foo" + "bar") // "foobar" of type MyString  
[]rune(MyString("白鵬翔")) // []rune{0x767d, 0x9d6c, 0x7fd4}  
(*Point)(p) // p is converted to *Point  
(<-chan int)(c) // c is converted to <-chan int  
(func())(x) // x is converted to func()  
(func() int)(x) // x is converted to func() int  
func() int(x) // x is converted to func() int (unambiguous)
```

# Type Conversion Example

```
type Sequence []int

// Copy method copies the current to a new Sequence
func (s Sequence) Copy() Sequence {
    result := make([]int, len(s))
    copy(result, s)
    return result
}

// String method sorts elements and returns them as string
func (s Sequence) String() string {
    s = s.Copy()
    sort.IntSlice(s).Sort()
    return fmt.Sprint([]int(s))
}
```

# Interface Conversions and Type Assertions

```
type Stringer interface {
    String() string
}

var value interface{} // Value provided by caller.

func String() {
    switch str := value.(type) {           // type switch
    case string:
        return str
    case Stringer:
        return str.String()
    }

    if str, ok := value.(string); ok {    // The same as above with type assertion
        return str
    } else if str, ok := value.(Stringer); ok {
        return str.String()
    }
}
```

# Loops - for

- “Classical” **for** loop:

```
sum := 0
for i := 0; i < 10; i++ {
    sum += i
}
fmt.Println(sum)
```

- Loop **for with missing init and step**:

```
sum := 1
for ; sum < 1000; {
    sum += sum
}
fmt.Println(sum)
```

- Endless loop with **break (do-while)**:

```
sum := 1
for {
    sum += sum
    if sum > 1000 {
        break
    }
}
fmt.Println(sum)
```

- Loop **while** condition is true:

```
sum := 1
for sum < 1000 {
    sum += sum
}
fmt.Println(sum)
```

# Loops – for range

```
nums := []int{2, 3, 4}
sum := 0
for _, num := range nums {
    sum += num
}
fmt.Println("sum:", sum)

for i, num := range nums {
    if num == 3 {
        fmt.Println("index:", i)
    }
}

kvs := map[string]string{"a": "apple", "b": "banana"}
for k, v := range kvs {
    fmt.Printf("%s -> %s\n", k, v)
}
for k := range kvs { fmt.Println("key:", k) }
for i, c := range "go" { fmt.Println(i, c) }
```

```
$ go run range.go
sum: 9
index: 1
a -> apple
b -> banana
key: a
key: b
0 103
1 111
```

# Decision Making - if

- **if-return (if-break, if-continue):**

```
func sqrt(x float64) string {  
    if x < 0 {  
        return sqrt(-x) + "i"  
    }  
    return fmt.Sprint(math.Sqrt(x))  
}
```

- **if with a short statement**

```
func pow(x, n, lim float64) float64 {  
    if v := math.Pow(x, n); v < lim {  
        return v  
    } // can't use v here, though  
    return lim  
}
```

- **If-else:**

```
func pow(x, n, lim float64) float64 {  
    if v := math.Pow(x, n); v < lim {  
        return v  
    } else {  
        fmt.Printf("%g >= %g\n", v, lim)  
    }  
    // can't use v here, though  
    return lim  
}
```

# Decision Making – switch-case-default

- **switch-case-default:**

```
fmt.Println("When's Saturday?")
today := time.Now().Weekday()
switch time.Saturday {
case today + 0:
    fmt.Println("Today.")
case today + 1:
    fmt.Println("Tomorrow.")
case today + 2:
    fallthrough
case today + 3:
    fmt.Println("Have to wait.")
default:
    fmt.Println("Too far away.")
}
```

- **Switch with no condition:**

```
t := time.Now()
switch {
case t.Hour() < 12:
    fmt.Println("Good morning!")
case t.Hour() < 17:
    fmt.Println("Good afternoon.")
default:
    fmt.Println("Good evening.")
}
```

# Functions – multiple return values

```
func swap(x, y string) (string, string) {
    return y, x
}

func main() {
    a, b := swap("hello", "world")
    fmt.Println(a, b)
}
```

# Functions - named return values

```
func split(sum int) (x, y int) {  
    x = sum * 4 / 9  
    y = sum - x  
    return  
}
```

```
func main() {  
    fmt.Println(split(17))  
}
```

# Value and Reference Parameters

```
func swapVal(x, y string) (string, string) {
    return y, x
}
func swapRef(x, y *string) {
    *x, *y = *y, *x
}

func main() {
    a, b := swapVal("hello", "world")
    fmt.Println(a, b)
    swapRef(&a, &b)
    fmt.Println(a, b)
}
```

Output:  
world hello  
hello world

# Variadic Parameters

```
func printf(format string, args ...interface{}) (int, error) {
    _, err := fmt.Sprintf(format, args...)
    return len(args), err
}

func main() {
    argsLen, err := printf("%v, %v\n", "abcd", 15)
    if err == nil {
        printf("Number args: %d\n", argsLen)
    } else {
        fmt.Printf("Error: %v\n", err)
    }
}
```

# Recommended Literature

- The Go Documentation - <https://golang.org/doc/>
- The Go Bible: Effective Go - [https://golang.org/doc/effective\\_go.html](https://golang.org/doc/effective_go.html)
- David Chisnall, *The Go Programming Language Phrasebook*, Addison Wesley, 2012
- Alan A. A. Donovan, Brian W. Kernighan, *The Go Programming Language*, Addison Wesley, 2016
- Nathan Youngman, Roger Peppé, *Get Programming with Go*, Manning, 2018
- Naren Yellavula, *Building RESTful Web Services with Go*, Packt, 2017

# Thank's for Your Attention!



Trayan Iliev

IPT – Intellectual Products & Technologies

<http://iproduct.org/>

<http://robolearn.org/>

<https://github.com/iproduct>

<https://twitter.com/trayaniliev>

<https://www.facebook.com/IPT.EACAD>