Basics

for functional and mixed-style programming

Quick history

- Invented by Rob Pyke, Robert Griesemer and Ken Thompson in 2007, went public in 2009

- Born out of a need for ease of programming combined with type safety and portability

- Other goals:
  - Easy to learn
  - Type and memory safety
  - Easy concurrency
  - Low latency garbage collection
  - Fast compilation

Go is used by








and thousands of other companies across the world

What Go is good for?

- "Large programs written by many developers,growing over time to support networkedservices in the cloud: in short, server software" — Rob Pike
- Mobile development (embedded code)
- System Programming
- Microcontrollers Programming

Differences from "classic" languages

- Unused variables are a compilation error
- Unused import directives are too
- No classes - Go is a procedural language
- No overloading
- No inheritance
- Visibility is controlled using capitalization, e.g: Foo() is public and foo() is not.

Syntax basics

Packages

```
package main

include "name"
include "math"
```

```
include (
 ext "github.com/user/repo"
 "math"
)
```

Types: primitives

тип	пример	нулевое значение
bool	true, false	false
string	"some-string"	""
int int8 int16 int32 int64	-123 123	0
uint uint8 uint16 uint32 uint64 uintptr	123	0
byte // uint8	123	0
rune // int32	'a' 97	0
float32 float64	123.97	0
complex64 complex128	2.96+1.58i	0+0i

Variables

```
var counter int
var iter int = 10
var ref = 10
var (
 counter int
 iter int = 10
 ref = 10
)
var x, y float32 = float32(ref), 10.33
z := "abra-cadabra" // infers type string
r := 'a' // infers type rune/uint32
```

Constants

```
const Pi = 3.14

const (
  E = 1
  Upsilon float32 = 9.46
  Phi rune = 0x03c6
  negative = false
)
```

Types: arrays

```
var [2]string
s := [2]string{"abc","def"}

s[1]
s[2] = "ghi"

var a = [2][2]int //2-dimensional array
```

Types: slices

```
var s []int // nil
s := []int{1,2,3}


s[1]
s[2] = 2


a := [6]int{1,2,3,4,5,6}
s := a[2:4] // []int{3,4} a half-open range [2,4)
```

Types: maps

```
var m map[string]string
m := map[string]string{
  "abc":"def",
  "ghi":"jkl",
}
m["abc"]
m["abc"] = "mno"
```

make

```
slice := make([]string, 2, 2)
```

```
mapping :=  make(map[string]int)
```

Functions

```
func equals(a int, b int) bool {
 return a == b
}
func paq(a, b int) (p int, q float32) {
 p = a * b
 q = float32(a) / float32(b)
 return
}
func numbers() (int, float32) {
 return 1, 3.14
}
```

Custom Types

```
type CustomInt int

type CustomFunc func (...)

type CustomStruct struct {
 customProperty int
 CustomPublicProperty string
}
```

Types: pointers

```
type T struct {}

var t *T = &T{}
t := &T{} // infers *T
```

```
var t T = T{}
p:= &t // infers *T
```

"Methods"

```
type C struct {
 name string
 value int
}
// GetName is a method
func (c *C) GetName() string {
 return c.name
}

func (c C) GetCName() string {
 return c.name
}
```

Interfaces

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

```
interface{}
```

Embedding

```
type Reader interface {
 Read(p []byte) (n int, err error)
}
type Writer interface {
 Write(p []byte) (n int, err error)
}
type ReadWriter interface {
 Reader
 Writer
}
```

Flow-control statements

if

```
if something {
} else if otherthing {
} else {
}
```

```
if v:= math.Pow(x, 2); v < 10 {
}
```

loops

```
//classic for loop
for x:=0; x<10; x++ { ... }
//while
for x < 100 { ... }
//forever
for { ... }
```

switch

```
switch value {
 case condition:
  //do smth
 default:
  //nothing matched
}
switch v:=runtime.GOOS; v {
 case "darwin":
  fmt.Println("OS X")
 default:
  fmt.Println(os)
}
```

Functional programming capabilities

Anonymous functions

```
varname := func (x,y int) { ... }
```

```
//define and invoke
result := func () bool { return !false }()
```

Functions are 1st class objects in Go

```
//get function from type, package or elsewhere
carry := math.Pow //reference without invoking
carry(2, 3) //invoke

func (strategy func(x int) string) string {
 return strategy(123)
}
```

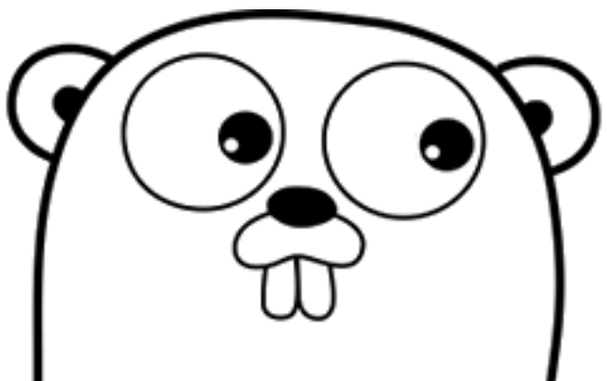
Clousures

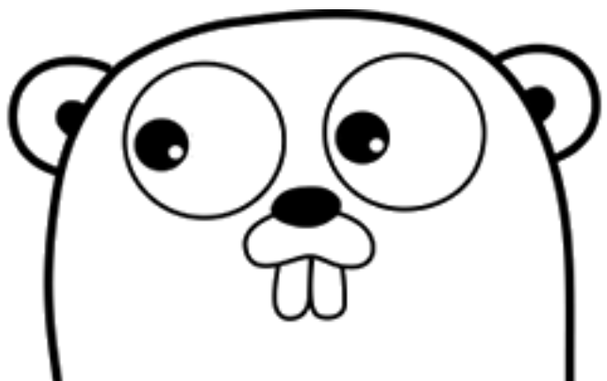
```
func Fib() func() int {
 var a,b int = 0, 1
 return func() int {
  a, b = b, a+b
  return a
 }
}

f := Fib()
for x:=0;x<10;x++ {
 fmt.Println(f())
}
```

Caveats

- Generators via channels
- No build-in implementations for Filter, Map, Reduce, etc.
- Lack of Immutable and Generic types
- Go does not have Tail Call Optimisation, so recursion comes with a tax of performance.

Parallel computations





Go main directive is

go

Goroutines

```
go func() {
  //do smth async
}()
```

Channels

**Do not communicate by sharing memory;
instead, share memory by communicating.**

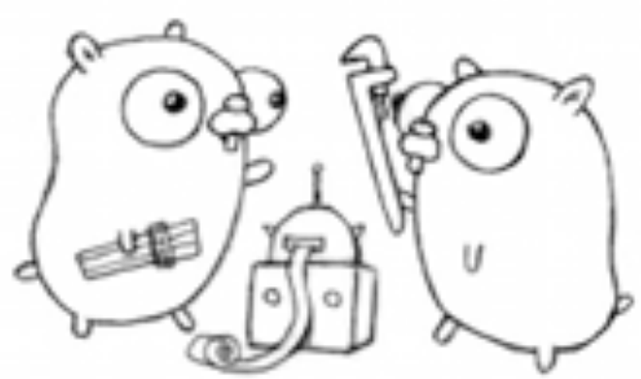
```
var c chan int = make(chan int)
```

```
go func() {
  c <- 1
}()
res := <- c
```

select

```
var c, abr chan int = make(chan int), make(chan int)
go func(){
  for { //loop forever
    select {
    case <-abr:
      return //abort execution by exiting the function
    case val:=<-c:
      //do something upon receiving value from channel
    case <-time.Tick(3 * time.Second):
      //do something every 3 seconds
    default:
      //nothing, may be absent
    }
  }
}()
```

Tools



go toolchain

- go build
- go test
- go run
- dep ensure
- go get
- go lint
- go fmt

IDEs

- JetBrains IntelliJ with Go-plugin
- JetBrains GoLand
- Sublime + plugins
- vim =)
- others...

Links

- A Tour of Go (https://tour.golang.org/)
- Go Playground (https://play.golang.org/)
- dep package manager (https://github.com/golang/dep)
- python’s itertools port to Go (https://github.com/yanatan16/itertools)
- samples of go code (https://github.com/golang-samples)
- Effective Go (https://golang.org/doc/effective_go.html)