

Getting Started with Go

- [Installation](#)
- [Add it to your Path](#)
- [Configuration](#)
- [Hello World Example](#)
- [Creating a Module](#)
- [Commands](#)
- [Variables](#)
- [If Statements](#)
- [Arrays](#)
- [Maps](#)
- [Loops](#)
- [Functions](#)
- [Structs/Types](#)
- [Pointers](#)
- [Concurrency](#)
- [Organizing your Code](#)
- [References](#)

Installation

```
brew install go
```

Add it to your Path

Add the following to your .bashrc or zshrc file

```
export PATH="$PATH:$(go env GOPATH)/bin"
```

Configuration

Setup default workspace

```
mkdir ~/go
mkdir ~/go/src
```

Hello World Example

```
vi helloworld.go
```

```
package main

import (
    "fmt"
)

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

Run it

```
go run helloworld.go
```

Compile it

```
go build
```

Install it

```
go install
```

Creating a Module

Create a go.mod file

go.mod

```
module mymodule  
go 1.18
```

Add Libraries

```
go get github.com/qopher/go-torrentapi
```

Commands

Command	Description
go run <src.go>	Run some code
go build	Compile the code. Outputs to local folder. Executable has folder as name.
go install	Compiles and places the output in workspace/bin
go env GOPATH	Outputs the default workspace folder

Variables

Variables are defined using the following syntax:

```
var <variablename> <type>
```

example:

```
var x int
```

All variables have a zero value and are pre-initialized to their zero value. For example, an int would have a zero value of 0 and a string would have a zero value of empty string "".

Initializing the variable at declaration time:

```
var x int = 5
```

Quick Syntax: infers type based on value.

```
x:=5
```

Example:

```
package main

import (
    "fmt"
)

func main(){
    var x int
    x = 5
    fmt.Println(x)
}
```

If Statements

```
package main

import (
    "fmt"
)

func main(){
    x := 7

    if x > 6 {
        fmt.Println("More than 6")
    } else if x < 2 {

    } else {

    }
}
```

Arrays

```
package main

import (
    "fmt"
)

func main(){
    //fix size
    var a [5]int
    a[2] = 7
    fmt.Println(a)

    a:= [5]int{5,4,3,2,1}
    fmt.Println(a)

    //slices
    a:= [] int{5,4,3,2,1}
    a = append(a,13)
    fmt.Println(a)
}
```

Maps

```
package main

import (
    "fmt"
)

func main(){

    vertices := make(map[string]int)

    vertices["triangle"] = 2
    vertices["square"] = 3
    vertices["tree"] = 12

    delete(vertices, "square")

    fmt.Println(vertices["triangle"])
    fmt.Println(vertices)

}
```

Loops

The only type of loop in go is the for loop.

```

package main

import (
    "fmt"
)

func main(){

    //for loop
    for i :=0; i<5; i++){
        fmt.Println(i)
    }

    //while loop
    j:=0
    for j<5 {
        fmt.Println(j)
        j++
    }

    //loop over array or slice
    arr := []string{"a","b","c"}
    for index, value := range arr {
        fmt.Println("index:", index, "value:", value)
    }

    //loop over a map
    m := make(map[string]int)
    m["triangle"] = 2
    m["square"] = 3
    for key, value := range m {
        fmt.Println("key:", key, "value:", value)
    }
}

```

Functions

Example showing error return

```

package main

import (
    "fmt"
    "error"
    "math"
)

func main() {
    result, err := sqrt(16)

    if err != nil {
        fmt.Println(err)
    } else {
        fmt.Println(result)
    }
}

func sqrt(x float64) (float64, error) {
    if x < 0 {
        return 0, errors.New("Undefined for negative numbers")
    }
    return math.Sqrt(x), nil
}

```

Method Set

```
func (h handler) ServeHTTP(w http.ResponseWriter, r *http.Request) {  
    ...  
}  
  
func (s *GracefulServer) BlockingClose() bool {  
    ...  
}
```

This is called the 'receiver'. In the first case (`h handler`) it is a value type, in the second (`s *GracefulServer`) it is a pointer. The way this works in Go may vary a bit from some other languages. The receiving type however, works more or less like a class in most object-oriented programming. It is the thing you call the method from, much like if I put some method `A` inside some class `Person` then I would need an instance of type `Person` in order to call `A` (assuming it's an instance method and not static!).

One gotcha here is that the receiver gets pushed onto the call stack like other arguments so if the receiver is a value type, like in the case of `handler` then you will be working on a copy of the thing you called the method from meaning something like `h.Name = "Evan"` would not persist after you return to the calling scope. For this reason anything that expects to change the state of the receiver, needs to use a pointer or return the modified value (gives more of an immutable type paradigm if you're looking for that).

Here's the relevant section from the spec; https://golang.org/ref/spec#Method_sets

Structs/Types

Example of Person Type

```
package main  
  
import (  
    "fmt"  
)  
  
type person struct {  
    name string  
    age int  
}  
  
func main(){  
    p:= person{name: "Jake", age: 23 }  
    fmt.Println(p)  
  
    fmt.Println(p.age)  
}
```

Pointers

Pass by Reference Example

```
package main

import (
    "fmt"
)

func main(){
    i := 7
    inc(&i)
    fmt.Println(i)
}

func inc(x *int) {
    *x++
}
```

Concurrency

Go supports concurrency by the use of the go command. All go functions in the background will terminate when the main terminates.

```
package main

import (
    "fmt"
)

func main(){

    //run in background
    go count("sheep")

    go count("fish")

    //wait for keyboard press before exiting
    fmt.Scanln()

}

func count(thing string) {
    for i:=0; true; i++ {
        fmt.Println(i, thing)
        time.Sleep(time.Millisecond * 500)
    }
}
```

Using Channels

Channels are blocking

```
package main

import (
    "fmt"
    "time"
)

func main(){

    //make channel of type string
    c := make(chan string)

    //run in background
    go count("sheep", c)

    //block until message is pushed to the channel
    msg:= <- c

    fmt.Println(msg)
}

func count(thing string, c chan string) {
    for i:=0; i <= 5; i++ {
        //send message on channel
        c <- thing
        time.Sleep(time.Millisecond * 500)
    }
}
```

In the above code, the program exits once the count routine returns a message on the channel.

```

package main

import (
    "fmt"
    "time"
)

func main(){

    //make channel of type string
    c := make(chan string)

    //run in background
    go count("sheep", c)

    //loop until count function closes the channel
    for {
        msg, open := <- c
        if !open {
            //channel closed, break out of for loop
            break
        }
    }

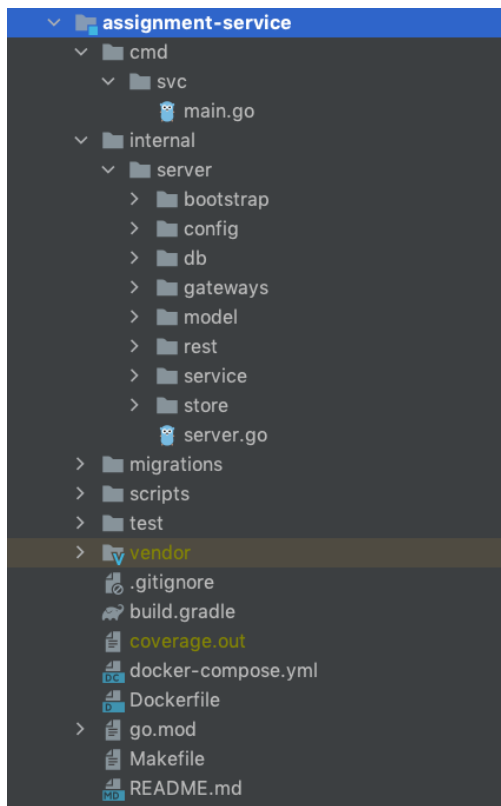
    fmt.Println(msg)
}

func count(thing string, c chan string) {
    for i:=0; i <= 5; i++ {
        //send message on channel
        c <- thing
        time.Sleep(time.Millisecond * 500)
    }

    //close channel
    close(c)
}

```

Organizing your Code



References

Reference	URL
Learn Go in 12 Minutes	https://www.youtube.com/watch?v=C8LgvuEBral
Concurrency in Go	https://www.youtube.com/watch?v=LvgVSSpwND8
Go Packages	https://golang.org/pkg/
Go By Examples	https://gobyexample.com/