Golang Programming Workshop
Basics

Wojciech Barczynski

Contents

1 Agreement

Notice: No copy&paste.

2 Prerequiments

2.1 Audience

Expected background knowledge and skills for the workshop:

- Have 1-year hands-on experience in other programming language;
- Know how to work with the Command Line Interface in Linux or OS.

2.2 Your workstation

In the github repository, you will find the instruction on how to set up your workstation.

- Linux or OSX recommended;
- Basic:
  - Golang;
  - IDE or code editor, vscode is a good start;
  - Git;
  - Docker.
- SQL and noSQL exercise (recommended with docker):
  - Postgres,
  - MongoDB.

Check Go Wiki to see how to configure your favorite editor to write golang programs.

2.3 Verify the setup

Let's run a hello world to check whether you can run go applications on your workstation.

Notice: No copy&paste, please.

Let's create a simple program to verify our setup:

```
# here we will create
# all projects
mkdir workspace
cd workspace

#
mkdir first_project
cd first_project

go mod init
go mod init first_project

# to see
# in the explorer
xdg-open .
```

Create `main.go` in your code editor in the catalog `first_app`, it should have the following text:

```
package main

import "fmt"

func main() {
        // 1 tabulator
        fmt.Println("Hello! YOUR_NAME")
}
```

Now, let's run it:

```
# 1. enforce formatting
# (with IDEs it is automatic):
gofmt -w .

# 2. run with go:
go run main.go

# 3. build and run:
go build .

# notice new binary executable:
ls

  main.go  first_app

# 3. Run the built binary:
./first_app
```

3 Golang Package Manager

Notice: Golang has a rich standard library, you should first see whether a given functionality is not already provided, before looking for an external library.
Installing new packages, e.g., github.com/logrusorgru/aurora:

```
# you should be
# in ~/workspace/first_project
pwd

go get -u github.com/logrusorgru/aurora/v4

# open also in vscode
# notice the *warnings*
cat go.mod

#
cat go.sum
```

```
# download all (used) deps to vendor/
go mod vendor

ls

  vendor/

# good to know:
go mod tidy
```

Let's use the library in our application:

```
package main

import "fmt"
import "github.com/logrusorgru/aurora/v4"

func main() {
        fmt.Println("Hello, ", aurora.Magenta("Natalia"))
}
```

3. Installing golang apps/tools and adding them to PATH:

```
# let's install our *first_app*
go install

# where is it installed?
#
# Let's find it out.
```

```
go env GOPATH

# do we see the *first_app* binary?
ls $(go env GOPATH)

# do we see the *first_app* binary?
ls $(go env GOPATH)/bin
```

What do you see when you run `go env`?

```
# to have available your golang tooling
# in your terminal:
export PATH=$PATH:$GOPATH/bin

# run it
# now whereever you are in the terminal
first_app
```

Note down:

- What is GOPATH: . . .
- What is GOBIN: . . .

4 Golang Playground

Open the browser and run our program on golang playground: https://play.golang.org/. Notice: you can generate a link to your code sample.

5 Variable definition

0. Create new program `hello-world` (directory in `workspace/`). Copy the `main.go` from our `first_project` project. You should start with:

```
package main

import "fmt"

func main() {
  // 1 tabulator
  fmt.Println("Hello! YOUR_NAME")
}
```

1. Please extract your name as a variable, use the following definitions and mark the incorrect ones:

```
// 1
var myName string = "Natalia"
var myName = "Natalia"

// 2
myName := "Natalia"

// 3
var myName
myName = "Natalia"

// 4
var myName string
myName = "Natalia"

// 5
var myName string
myName := "Natalia"

// see which combo works:
fmt.Println("Hello!", myName)
```

Notice: in Golang, we use **camelCase** for variable names (and function names).

2. Mark the myName as `const`.

3. Declare a variable for your home country and city, use the following construct:

```
var (
  x = 10
  y = 20
)
```

and print it on the screen.

6 Integers and Floats

No big surprise here, numbers:

- `int`, `int8, ..., int64`, byte → int8, rune → int32
- `unint`, `uint8, ..., uint64`
- `float64`, `float32, float64`
- `complex`

Golang does not support automatic conversion between types. Let's experience it.

1. Declare a variable `devExpDays` and `msg`:

```
package main

import (
  "fmt"
  "reflect"
  "strconv"
)

func main() {
  name := "Natalia"
  devExpDays := 365
  msg := name + " has " + devExpDays + " exp as developer"
  fmt.Println(msg)
}
```

Run it. What error message did you see?

2. To make the code running, we should use `strconv.Itoa`. Add the following import and call the function:

```
import (
  "strconv"
  "fmt"
)
```

3. Now let's go back from string to integer:

```
// imagine, we got it from the user:
devExpYears := "2"

// does it work?
devExpDays := 365 * devExpYears
```

to convert `devExpYears` use the following code:

```
// the famous error-return
days, err := strconv.Atoi("12020")
if err != nil {
  fmt.Printf("Cannot convert %v", err)
  return
}
```

You have more functions to convert from basic types to string and back, check Package strconv documentation.
Notice: `import (_ "strconv")`.

7 Boolean

Just to note: `true` and `false`, standard logical operators: `&&`, `!`, and `||` works.

8 Math

Nothing dramatic here. For more advance mathematical functions, you should check the golang.org/pkg/math:

```
import (
  "math"
)
```

9 Slices and hidden arrays

In Golang, we use `slices`, seldom we use arrays.

1. If you want to defined an array, you specify the length explicitly:

```
arr1 := [...]string{"pa", "rr", "ot"}
arr2 := [3]string{"pa", "rr", "ot"}

fmt.Print(arr1)
fmt.Printf("%v", arr1)
```

Slice, an interface of the array, on the other hand we create with:

```
arr1 := [...]string{"pa", "rr", "ot"}

slice1 := []string{"pa", "na", "ma"}
slice2 := arr1[:]
```

Notice: `relect.TypeOf(arr1)` vs `relect.TypeOf(slice1)`. Good to know when reading compilation errors or runtime panics.

2. What is the output?

```
  var three [3]int
  two := [2]int{10, 20}

  three = two

  fmt.Println(three)
  fmt.Println(two)
```

3. Let's define our hello world messages and add one more:

```
helloWorld := []string{"dzień dobry", "Hallo", "guten Tag"}
fmt.Printf("A: len: %d cap: %d \n", len(helloWorld),
  cap(helloWorld))

czechia := "Ahoj"
helloWorld = append(helloWorld, czechia)

fmt.Printf("B: len: %d cap: %d \n", len(helloWorld),
  cap(helloWorld))

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

Note down:

- the `len` and `cap` in `A` and `B`: . . .
- What has happen when we appended one slice to another?

4. Slices of slices. How would you write a one liner to print out:

- `["dzień dobry"]`:
- middle hallo messages:
- all except the last one:
- just 1st element:
- just 15th element:

Hint: use `slice[x]`, `slice[x:]`, `slice[:x]`, and `slice[x:y]`.

5. Watch out, the slices might bite your head off. Note, slice has *capacity*, *length*, and **pointer to the underlaying array** (see https://golang.org/pkg/reflect/#SliceHeader):

```
package main

import "fmt"

func main() {
  helloWorld := []string{"dzień dobry", "Ahoj", "Goodmorning"}
  centralEuHello := helloWorld[0:2]
  fmt.Printf("len: %d cap: %d \n", len(centralEuHello),
    cap(centralEuHello))

  centralEuHello[0] = "Dobry Wieczor"
  fmt.Printf("%v\n", helloWorld)
  fmt.Printf("%v\n", centralEuHello)
}
```

What is the result?

Replace `centralEuHello[0] = "Dobry Wieczor"` by:

```
centralEuHello = append(centralEuHello, "Dobry Wieczor")
centralEuHello = append(centralEuHello, "Dobry Wieczor")
```

What is the result? What has happend?

It should be not a suprise that:

```
a := []int{1,3,5,7}
b := []int{2,4,6}

a = b

a[0] = 99

// prints the same
fmt.Printf("%+v\n", a)
fmt.Printf("%+v\n", b)
```

6. Let's fix that with the following code snipped:

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

7. We can also use make to create a slice with desired length and capacity:

```
msgs := make([]string, 2, 20)
msgs[0] = "Ahoj"
msgs[1] = "Goodmorning"

for idx := range msgs {
  fmt.Printf("%s\n", msgs[idx])
```

```
  }

  for idx, v := range msgs {
    fmt.Printf("%d, %s\n", idx, v)
  }

  for _, v := range msgs {
    fmt.Printf("%s\n", v)
  }
```

8. Note, when a function returns no result, use a *nil slice*:
`var nilSlice []string`.

10 nil gotchas

What does `fmt.Println(nilSlice==nil)` prints?

```
func main() {
        var nilSlice []string

        fmt.Println(nilSlice==nil)
}
```

9. Let's add a nil value for a pointer to integer:

```
//var nilSlice []string

var i *int = nil
fmt.Println(i==nil)
```

ok, so both variable are `nil`, let's compare them to each other:

```
fmt.Println(nilSlice==i)
```

What did happen?

[Homework] read golang.org FAQ entry on nill errors and Dave Cheney blog post.

11 Control structure: Loops

The Go for loop is similar to—but not the same as—C's. It unifies for and while and there is no do-while[1]. In Golang there is only one loop keyword:

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

  for i < 10 {
    fmt.Println(i)
    i++
  }
  // also map
  for index, value := range someSlice {
    fmt.Println(index, value)
  }
```

```
// Like a C for
for init; condition; post { }

// Like a C while
for condition { }

// Like a C for(;;)
for { }
```

12 String

String is a read-only **slice** of bytes. Go source code is always in UTF-8.

1. Run the following code, note down the results:

```
const adress := "ul. Przeskok 2"

fmt.Printf("len: %d\n", len(adress))
```

[1] go.dev/doc/effective_go#for.

```
fmt.Printf("1: %s\n", adress[0:3])
fmt.Printf("2: %c\n", adress[2])
fmt.Printf("3: %s\n", adress[2:])
fmt.Printf("4: %s\n", adress[5:])

fmt.Printf("5: %s\n", adress[16:])
fmt.Printf("6: %s\n", adress[:16])
```

2. Use the following example to build your own program printing out your 3 favorite emoicons:

```
var milk = "우유"

for index, runeValue := range milk {
        fmt.Printf("%c (%U) starts at byte position %d\n",
                runeValue, runeValue, index)
}
```

3. What does now happen?:

```
const milk = "우유"

  for i := 0; i < len(milk); i++ {
    fmt.Printf("byte: %x at the index %d\n",
      milk[i], i)
  }
```

Now let's mix things up, write a short program that prints runes for: `const mixed = "wöjtk유wx"`.

Notice: Important packages are `Package strings` (pkg.go.dev/strings) and `Package unicode/utf8` (pkg.go.dev/unicode/utf8).

13 Maps

Build a program that displays a hello-world message for different languages.
1. Define the map:

```
  helloMsgs := map[string]string{}
  helloMsgs["pl"] = "Dzień Dobry"
  helloMsgs["en"] = "Good morning"
```

2. Read input from the user:

```
  helloMsgs := map[string]string{}
  helloMsgs["pl"] = "Dzień Dobry"
  helloMsgs["en"] = "Good morning"

  var lang string
  // read a single world in ASCII
  // skip error handling
  fmt.Scan(&lang)
```

2. Print the hello message:

```
if val, ok := helloMsgs[lang]; ok {
  // found
} else {
  // not found
}
```

3. Notice, Golang has a cool feature:

```
  helloMsgs := map[string]string{
    "pl": "Dzień Dobry",
    "en": "Good morning",
  }
```

We want to have different greetings depending on the time of day:

```
  helloMsgs := map[string]map[string]string{
    "pl": {"morning": "Dzień Dobry"},
    "en": {"morning": "Good morning"},
  }
```

Now, users got bored with the same greeting:

```
helloMsgs := map[string]map[string][]string{
  "pl": {"morning": []string{
    "Dzień Dobry",
    "Piękny poranek",
  }},
  "en": {"morning": []string{
    "Good morning",
    "Morning",
  }},
}
```

Let's make it more readable:

```
type daytimeGreetings map[string][]string
type g map[string]daytimeGreetings

helloMsgs := map[string]daytimeGreetings{
  "pl": {"morning": []string{
    "Dzień Dobry",
  }},
  "en": {"morning": []string{
    "Good morning",
  }},
}
```

You can use this declarative style to define even the most complex JSON structures.

4. Write a program that randomize the messages, use *math/rand* and *time* packages to initialize random seed.

5. [Homework] Use time information to find out which part of the day we have.

14 User defined type

You can define types over the basic and composite types:

14.1 Type Definitions

```
package main

import "fmt"

type myInt int

func display(i int) {
  fmt.Printf("%d", i)
}

func main() {
  var i myInt = 12
  i = i + 12
  // how to fix it?
  display(i)
}
```

14.2 Type Alias Declarations

Since Go 1.9, we can declare custom type aliases by using the following syntax:

```
package main

import "fmt"

type myInt = int

func display(i int) {
  fmt.Printf("%d", i)
}

func main() {
  var i myInt = 12
  i = i + 12
  display(i)
}
```

15 fmt.Printf

Check the https://gobyexample.com/string-formatting.

16 Functions

1. Let's move the logic for displaying the hello message to a function. We will return `false` if we do not support a given language:

```
func displayHello(lang string) (bool) {
    helloMsgs := map[string]string{
    "pl": "Dzień Dobry",
    "en": "Good morning",
    }
    // here code
    return false
}
```

2. Let's follow the golang way and return an error:

```
func displayHello(lang string) (err error) {
  // put code here
  err = fmt.Errorf("Unsupported language")
  // .. and here
  return err
}

func main() {
  err := displayHello()
  if err != nil {
    fmt.Printf("Not found!!! %v", err)
    return
  }
}
```

3. Functions are the first class citizens in Golang and we often use them as arguments. Let's write new program:

```
func printThings(msg []string, decorator func(string) string) {
  for _, l := range msg {
    d := decorator(l)
    fmt.Println(d)
  }
}

func main() {
  things := []string{"mleko", "cars", "programming"}

  likeThat := func(s string) string {
    return "Ania likes " + s
  }
  printThings(alphabet, likeThat)
}
```

Now something more complicated:

```
type letterDecorator func(string) string

//func printLetters(msg []string, decorator func(string) string)
func printLetters(msg []string, decorator letterDecorator) {
  for _, l := range msg {
    d := decorator(l)
    fmt.Println(d)
  }
}

func main() {
  alphabet := []string{"a", "b", "c", "d", "d"}

  printLetters(alphabet, func(s string) string {
    return strings.ToUpper(s)
  })
  printLetters(alphabet, func(s string) string {
    return "::" + strings.ToLower(s) + "::"
  })
}
```

4. Notice: we have support for variadic parameters in functions:
`func printSymbols(msg ...string):`

```
printSymbols()
printSymbols("a", "z")
printSymbols(alphabet...)
```

5. We can move the execution of a function to the end of the scope with `defer`:

```
package main

import "fmt"

func main() {
  defer fmt.Println("booom!")

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

We will come back to `defer` later in this workshop.

17 Control structure: if and switch

17.1 switch

The *switch* can work on any data type, you do not need to switch on a value, see: https://github.com/golang/go/wiki/Switch.
The very common case for *switch* is to check types:

```
func do(v interface{}) string {
  switch u := v.(type) {
  case int:
    return strconv.Itoa(u*2) // u has type int
  case string:
    mid := len(u) / 2 // split - u has type string
    return u[mid:] + u[:mid] // join
```

```
  }
  return "unknown"
}
```

17.2 if

The *If* and *else* works as in other programming languages, except that you can put a language expression:

```
if err := dec.Decode(&val); err != nil {
  // handling error
}
// happy path
```

Example from the Package net:

```
if nerr, ok := err.(net.Error); ok && nerr.Temporary() {
  // here nerr is an instance of net.Error
  // and the error is Temporary
}
```

Let's build an app for writing and reading a file:

```
package main

import (
  "io/ioutil"
  "fmt"
)

func main() {
  // change to /dat1
  fPath := "/tmp/dat1"
  inD := []byte("hello\nWorld\n")
  if err := ioutil.WriteFile(fPath, inD, 0644); err != nil {
    panic(err) // failfast
  }
  fmt.Println("Write was successful!")
```

```
  outData, err := ioutil.ReadFile(fPath)
  if err != nil {
     panic(err) // failfast
  }
  fmt.Print(string(outData))
}
```

If it works, change `ioutil.ReadFile(fPath)` to `ioutil.ReadFile(fPath + "x")`. What does happen?

18 Pointers

Remember:

- Because of the Golang design, your code will work usually faster if you pass small data types by value;
- Do not be overzealous with the pointers;
- Maps, slices, and pointers are reference types.

Write the following program:

```
package main

import "fmt"

func tryAnswerEverything(i int) {
  i = 45
}

func answerEverything(i *int) {
  *i = 42
}

func main() {
  i := 33

  fmt.Println(i)
  tryAnswerEverything(i)
```

```
  fmt.Println(i)

  answerEverything(&i)
  fmt.Println(i)
}
```

Notice, we can return in functions pointers to local variables: `func Answer() *int`.

19 Structures

In Golang, we do not have *classes*, instead *struct* with *methods*. The language does not support inheritance, favors composition instead.

19.1 Structures and Methods

1. Write a program to manage employees:

```
package main

type Employee struct {
    FirstName   string
    LastName    string
    leavesTotal int
    LeavesTaken int
}

func (empl *Employee) TakeHolidays(days int) error {
  // write an implementation
}

func (empl *Employee) limitExceeded(days int) bool {
  // write an implementation
}

func main() {
  empl := new(Employee)
  empl.FirstName = "Laste"
  empl.LastName = "BB"
  empl.leavesTotal = 26
```

```
  fmt.Println(empl.FirstName)
  // simulate here taking day offs
  // and printing the new value.

  // try to call limitExceeded on empl
}
```

2. Refactor our application and move implementation of the `employee` to a separate package. To do it, create a directory `employee` and create inside `employee.go`:

```
package employee

type Employee struct {
  FirstName   string
  LastName    string
  leavesTotal int
}

// ... the functions
// ...
```

Create an instance of `Employee` in `main.go`.

3. What if, we do not want to expose the implementation details to the code in the main. Change the name of the type from `Employee` to `employee`:

```
package employee

type employee struct {
  FirstName   string
  LastName    string
  leavesTotal int
}

// ... rest of the code
```

Does your app works? How we should solve this problem.

3. Implement the solution.

20 Pointer receiver vs value receiver

What is the difference?

```
func (empl *Employee) TakeHollidays(taken int) {
    empl.leavesTaken = empl.leavesTaken + taken
}
```

and:

```
func (empl Employee) TakeHollidays(taken int) {
    empl.leavesTaken = empl.leavesTaken + taken
}
```

Write a new app to find out.

21 Structures and Interfaces

In Golang, we have duck typing interfaces, it means that the structure has to implement the functions from the interface.
An example, we will build an application that supports two storage types - postgres and mongo:

```
package main

import "db-example-app/postgres"

type DataStore interface {
  GetEmployee(id int) string
}

type App struct {
  ds DataStore
}
```

```
func main() {
  app1 := App{ds: &postgres.PsqlStore{}}
  fmt.Print(app1.ds.GetEmployee(12))
}
```

We might have two configurable stores, one psql:

```
package postgres

type PsqlStore struct {
}

func (ps *PsqlStore) GetEmployee(id int) string {
  return "psql"
}
```

and one, mongodb:

```
package mongo

type MongoStore struct {
}

func (ps *MongoStore) GetEmployee(id int) string {
  return "mongo"
}
```

Run the app and verify that the mongoStore works as well. When it works, break it, for e.g., change the type of args in `MongoStore`.

22 Composition instead of Inheritance

```
type person struct {
  firstName string
}

func (p person) name() string {
  return p.firstName
```

```
}

type employee struct {
  EmployeeID string
  person
}

func main() {
  empl := employee{}
  empl.firstName = "Natalia"
  empl.person.firstName = "Christof"
  fmt.Println(empl.firstName)
}
```

Notice: For polymorphism, you need to use *interface*.

23 Interfaces vs Functions

If you get high on interface, do not forget that you can use functions instead. See an example from net/http:

```
type HandlerFunc func(ResponseWriter, *Request)

// ServeHTTP calls f(w, r).
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)
}
```

Let's see it on an example:

```
package main

// employee package
type PolicyHandler func(employee *Employee) bool

func (f PolicyHandler) canTakeHolidays(empl *Employee) bool {
  return f(empl)
}
```

```
func (f PolicyHandler) HolidayFreeze() bool {
  return true
}

type Employee struct {
  totalLeaves int
  leavesTaken int
  PolicyHandler PolicyHandler
}

func (empl *Employee) TakeHolidays() bool  {
  if empl.PolicyHandler.canTakeHolidays(empl) != true {
    return false
  }
  return true
}
```

```
import "fmt"

func main() {
  var newPolicy PolicyHandler = func(empl *Employee) bool {
      return empl.totalLeaves > empl.leavesTaken
  }
  freeze := newPolicy.HolidayFreeze()
  fmt.Println(freeze)

  employee := Employee{12, 20, newPolicy}
  employee.TakeHolidays()
}
```

24 First Web App

Now, we will build our first web app before going to the 2nd part of the basics. We will use github.com/go-chi/chi.
1. Create an app: *first_web_app*, install the `chi` package.

2. Start with the example from the github:

```
package main
```

```go
import (
  "net/http"

  "github.com/go-chi/chi/v5"
  "github.com/go-chi/chi/v5/middleware"
)

func main() {
  r := chi.NewRouter()
  r.Use(middleware.Logger)
  r.Get("/", func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("welcome"))
  })
  http.ListenAndServe(":3000", r)
}
```

3. Tasks:

   1. add endpoint `/hello`, that returns `world`;
   2. add support for adding name: `/hello?name=natalia`, should return: `Hello Natalia!`;
   3. add support for language as a GET parameter;
   4. move all the logic to a separate package `greetings`;
   5. Use one of the middlewares, e.g., `SetHeader` or `BasicAuth`.

25 Linters

Linters are a part of the language:

- gofmt
- goimport - gofmt + sorting imports
- govet - now executed with tests

To apply the fixes, use `-w` flag:

```
$ gofmt -w .
$ goimports -w *.go && goimports -w */*.go
```

There are many linters out there, check, for e.g., awesome-go-linter page. You should call the linters as part of your CI/CD pipeline:

- linter
- test
- integration-test

26 References

- golang.com/doc/effective_go.html;
- devs.cloudimmunity.com/gotchas-and-common-mistakes-in-go-golang.