




G!RLS WHO CoDE



Laurențiu Crăciun
Marius Rusu

- 
- General Facts
 - Running a Hello World
 - Environment Setup
 - Basics
 - Flow Control
 - Web Service

1. Facts

- Free and Open Source
- Created at Google by Robert Griesemer, Rob Pike and Ken Thompson
- Development started in 2007 and publicly release in November 2009
- C like syntax without semicolons
- Compiled
- Built-in concurrency

1. Facts

- Object Oriented
 - ◆ Structs are classes
 - ◆ Inheritance is composition
 - ◆ Polymorphism through interfaces
- Garbage collected
- Strongly typed. No implicit conversions: keep things explicit
- Fast build

2. Hello World

→ To play with Go programming online: play.golang.org

```
package main

import "fmt"

func main() {
    fmt.Println("Hello,world!")
}
```

2.1 Installing Go compiler on GNU/Linux

- Download Go compiler binary from golang.org/dl
- Extract into your home directory (\$HOME/go)
- Add the following lines to your \$HOME/.bashrc

```
export GOROOT=$HOME/go  
export PATH=$GOROOT/bin:$PATH
```

- Open a new console window or source \$HOME/.bashrc
- More details on <https://golang.org/doc/install>

2.2 Building and running

→ You can run the program using “go run” command

```
go run hello.go
```

→ You can also build (compile) and run the binary like this in GNU/Linux

```
go build hello.go  
./hello
```

2.3 Organizing Code

- **\$GOPATH** directory is a workspace (source, packages and binaries)
- Three sub-directories under \$GOPATH: **bin**, **pkg** and **src**
 - ◆ **bin** directory contains executable binaries
 - ◆ The **src** directory contains the source files
 - ◆ The **pkg** directory contains package objects used by go tool to create the final executable

2.3 Organizing Code

bin/

hello
outyet

#command executable
#command executable

pkg/

linux_amd64/
 github.com/golang/example/
 stringutil.a

#package object

src/

github.com/golang/example/
 .git/
 hello/
 hello.go
golang.org/x/image/

#Git repository metadata

#command source

...

3.0 Basics

→ Most used types:

- ◆ bool, int, byte, string

→ All types:

- ◆ int8, int16, int32, int64, uint, uint8, uint16, uint32, uint64, uintptr, float32, float64, complex64, complex128

3.1 Packages

- Why packages?
 - ◆ Way to modularize code
 - ◆ Similar to namespaces
 - ◆ Types and functions

3.2 Packages...

- Go has quite a lot built-in packages: <https://golang.org/pkg/>
- Every Go program is made up of packages
- To define an entrypoint for an executable, package name should be **main**
- Programs start running in package main

- All the files of a package must reside in the same directory.
- You cannot have multiple package definitions in the same directory.

```
package main
```

3.3 Imported & Exported names

- Importing gives access to exported types and functions defined in other files.
- Exported names start with an uppercase character: **Foo** and **FOO** are exported. The name **foo** is not exported.
- By convention, the package name is the same as the last element of the import path.
- Use an alias to avoid package name conflicts during imports

```
import (  
    "fmt",  
    myjwt "golang.org/x/oauth2/jwt"  
)
```

3.4 Variables

- The basic declaration of variables is done using keyword **var**
- The type is explicitly defined in a variable declaration. Every declared variable is initialized with the default value for that type.

```
var x int  
var flag bool  
var message string
```

```
fmt.Println(message, flag, x)  # "" false 0
```

3.4 Variables

→ Initialization of variables can be done on declaration

```
var x int = 10  
var flag bool = true  
var message string = "Hello Go World!"  
  
fmt.Println(message, flag, x)  
  
# "Hello Go World!" true 10
```

3.4 Variables

- Type can also be inferred from the values that are given on initialization

```
var x = 10 //int  
var flag = true //bool
```

- Short variable declaration

```
x := 10  
flag := true
```


3.7 Constants

- Constants are declared using the keyword **const**
- Constants can be char, string, bool or numeric values
- Constants cannot be declared using the **:=** syntax

```
const Pi = 3.14  
const Activated = true  
const FirstName = "John"
```

3.8 Functions

→ Declaring functions

```
func [name] ([params]) [return value]  
func [name] ([params]) ([return values])
```

3.9 Functions

→ **A function that will say Hello to the user**

```
package main
import "fmt"

func salutation(name string, greeting string) string {
    return greeting + " " + name
}

func main() {
    // "Hello John"
    fmt.Println(salutation("John", "Hello"))
}
```

3.10 Functions

- Go provides the functionality to name the return values. If named, when a function exists, it will look for locally declared variables with that name and return their values.

```
package main
import "fmt"

func salutation(name, greeting string) (message string, alternate string)
{
    message = greeting + " " + name
    alternate = "Hey, " + name

    return
}

func main() {
    helloMessage, heyMessage := salutation("John", "Hello")
    // "Hello, John" "Hey, John"
    fmt.Println(helloMessage, heyMessage)
```

4.0 Flow control

If and else

- The braces { } are mandatory
- The condition is not surrounded by parentheses

```
if 2 > 1 {  
    fmt.Println("2 is bigger than 1")  
}
```

4.1 If and else ...

If and else

- The **if** statement can start with a short statement to execute before the condition
- Variables are block-scoped. They no longer exist after the **if**'s closing bracket
- Variables declared inside an **if** short statement are also available inside any of the **else** blocks

```
if age := 43; age > 40 {  
    fmt.Println("You are old")  
} else {  
    fmt.Println("You are young")  
}
```

4.2 For

- Go has only one looping construct, the **for** loop
- There are no parentheses surrounding the three components of the for statement
- The braces { } are mandatory

```
func main() {  
    sum := 0  
    for i := 6; i < 10; i++ {  
        sum += 1  
    }  
    fmt.Println(sum) // 4  
}
```

4.3 For

- Pre and post conditions can be omitted
- Semicolons can be omitted in this case
- **while** is spelled **for** in Go

```
func main() {  
    sum := 1  
    for sum < 10 {  
        sum += sum  
    }  
  
    fmt.Println(sum) // 16  
}
```


4.4 Switch

- The cases are evaluated top to bottom until a match is found
- Break statements is not needed because is automatically added at the end of each case

```
position := 1
switch position {
    case 1:
        fmt.Println("First!")
    case 2:
        fmt.Println("Second!")
    case 3:
        fmt.Println("Third!")
}
```

4.5 Defer

- A defer statement defers the execution of a function until the surrounding function returns
- The deferred call's arguments are evaluated immediately, but the function call is not executed until the surrounding function returns
- Executed in LIFO order

```
func main() {  
    defer fmt.Println("world")  
  
    fmt.Println("hello")  
}
```

```
// "hello"  
// "world"
```

5.0 More types

Structs

- A **struct** is a collection of fields
- Struct fields are accessed using a dot
- User is a **struct** type

```
type User struct {  
    Name string  
    Age int  
}  
  
func main() {  
    p := User{"John", 22}  
    fmt.Printf("%v\n", p) // {John 22}  
    fmt.Printf("%+v\n", p) // {Name:John Age:22}  
    fmt.Printf("%#v\n", p) // main.User{Name:"John", Age:22}  
    fmt.Println("Name: ", p.Name) // Name: John  
}
```

5.0 More types

Arrays

- An ordered container type with a fixed number of values
- The number of values in an array is called **length**
- An array's length is part of its type, so arrays cannot be resized
- The type **[n]T** is an array of n values of type **T**

```
colors := [3]string{"Red", "Green", "Blue"}
```

- The expression **s[n]** accesses the nth element, starting from zero

```
c := colors[1]  
fmt.Println(c) // "Green"
```

5.0 More types

Slices

- A slice is a dynamically-sized, flexible view into the elements of an array
- Slices are much more common than arrays
- The type `[]T` is a slice with elements of type `T`
- The zero value of a slice is `nil`

```
primes := [6]int{2, 3, 5, 7, 11, 13}
```

```
var s []int  
s = primes[1:4]  
fmt.Println(s)      // [3 5 7]
```

5.0 More types

Slices

- A new element can be appended to a slice using built-in function **append**
- The resulting value of **append** is a slice containing all the elements of the original slice plus the provided values

```
var s []int
fmt.Println(s)

// append works on nil slices.
s = append(s, 0)
fmt.Println(s)

// The slice grows as needed.
s = append(s, 1)
fmt.Println(s)
```

5.0 More types

Maps

- A map creates an association between a key and a value
- The zero value of a map is nil

```
var fruitWeights = map[string]int {  
    "Apple": 45,  
    "Mango": 24,  
    "Orange": 34,  
}
```

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

```
// map[string]int{"Apple":45, "Mango":24, "Orange":34}
```

6.0 Methods

Methods

- Go does not have classes. However, you can define methods on types
- A method is a function with a special receiver argument.

```
func ([receiver]) [name] ([params]) ([return  
values])
```

- A receiver could be any type with a name
- A method can have two types of receivers: **pointer receivers** or **value receivers**

6.0 Methods

- There are two reasons to use a pointer receiver
- The first is so that the method can modify the value that its receiver points to
- The second is to avoid copying the value on each method call

```
package main
import "fmt"
type myType struct {}
func (mt *myType) Hello(){
    fmt.Println("Hello from myType")
}

func main(){
    myVar := &myType{}
    myVar.Hello()
}
```

6.1 Errors

- Go programs express error state with **error** values
- The **error** type is a built-in interface
- Functions often return an error value
- A **nil** error denotes success; a non-nil error denotes failure

```
package main
import "fmt"
func giveMeAnError() error {
    return fmt.Errorf("My Error")
}

func main(){
    err := giveMeAnError()
    if err != nil {
        fmt.Println(err.Error())
    }
}
```

7.0 Web service

- A web service is any piece of software that makes itself available over the internet and uses a standardized messaging system.
- Let's see how we can make a Web service in Go:

7.1 Hello, world!

```
package main

import (
    "fmt"
    "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hi there, I love %s!", r.URL.Path[1:])
}

func main(){
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

```
fmt.Println("question?")
```

