Alfonso De La Guardia

Task Organization

#	Task	Priority (1-5)	Description	Difficulty (1-5)	Estimation
1	Reading the project description	5	Reading the description of the project	1	30 min
2	Create Repo	3	Creating git repo for the project	1	2 min
3	Working on the theoretical section	5	Answering the questions asked	4	1 hr
4	Working on the practical section	5	Working with the .xml file for the restaurant exercise	5	5 hr
4.1	HTTP Status Code	5	Being able to return HTTP status code headers	4	
4.2	HTML Template	5	Being able to return a request from an HTML file	4	
4.3	Return XML	5	The API must return a XML object	4	
4.4	Return JSON	5	Being able to return the previous object in JSON format	4	
4.5	Return text	5	The API must return information of your choice in plain text	4	
4.6	Return image	5	The API must return an image of your choice obtained through a file	5	
5	Review	2	Review Project	2	30 min

Variables in Go

- In Go, variables are explicitly declared and used by the compiler
- Var declares 1 or more variables.
- Variables declared without initialization are zero-valued
- The := syntax is shorthand for declaring and initializing a variable

```go
package main

import "fmt"

func main() {

    var a = "initial"
    fmt.Println(a)

    var b, c int = 1, 2
    fmt.Println(b, c)

    var d = true
    fmt.Println(d)

    var e int
    fmt.Println(e)


    f := "apple"
    fmt.Println(f)
}
```

```
$ go run variables.go
initial
1 2
true
0
apple
```

Pointers in Go

- Go supports pointers, allowing you to pass references to values and records within your program.
- We'll show how pointers work in contrast to values with 2 functions:

zeroval and zeroptr. zeroval has an int parameter, so arguments will be passed to it by value.

- zeroval will get a copy of ival distinct from the one in the calling function.
- zeroptr in contrast has an *int parameter, meaning that it takes an int pointer.
- The &i syntax gives the memory address of i, i.e. a pointer to i.

```go
package main

import "fmt"

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)
}
```

```
$ go run pointers.go
initial: 1
zeroval: 1
zeroptr: 0
pointer: 0x42131100
```

Functions in Go

- This is a function that takes two ints and returns their sum as an int.
- Go requires explicit returns, i.e it won't automatically return the value of the last expression.
- Go requires explicit returns, i.e it won't automatically return the value of the last expression
- zeroptr in contrast has an *int parameter, meaning that it takes an int pointer.
- Call a function just as you'd expect, with name (args).

```go
package main

import "fmt"

func plus(a int, b int) int {


    return a + b
}

func plusPlus(a, b, c int) int {
    return a + b + c
}


func main() {

    res := plus(1, 2)
    fmt.Println("1+2 =", res)

    res = plusPlus(1, 2, 3)
    fmt.Println("1+2+3 =", res)
}
```

```
$ go run functions.go
1+2 = 3
1+2+3 = 6
```

Functions in Go

- This is a function that takes two ints and returns their sum as an int.
- Go requires explicit returns, i.e it won't automatically return the value of the last expression.
- Go requires explicit returns, i.e it won't automatically return the value of the last expression
- zeroptr in contrast has an *int parameter, meaning that it takes an int pointer.
- Call a function just as you'd expect, with name (args).

```go
package main

import "fmt"

func plus(a int, b int) int {


    return a + b
}

func plusPlus(a, b, c int) int {
    return a + b + c
}


func main() {

    res := plus(1, 2)
    fmt.Println("1+2 =", res)

    res = plusPlus(1, 2, 3)
    fmt.Println("1+2+3 =", res)
}
```

```
$ go run functions.go
1+2 = 3
1+2+3 = 6
```

Conditionals in Go

- You can have an if statement without an else.
- A statement can precede conditionals, any variables declared in this statement are available in all branches
- You don't need parentheses around conditions in Go, but that the braces are required.
- There is no ternary if in Go, so you'll need to use a full if statement even for basic conditions.

```go
package main

import "fmt"

func main() {

    if 7%2 == 0 {
        fmt.Println("7 is even")
    } else {
        fmt.Println("7 is odd")
    }

    if 8%4 == 0 {
        fmt.Println("8 is divisible by 4")
    }

    if num := 9; num < 0 {
        fmt.Println(num, "is negative")
    } else if num < 10 {
        fmt.Println(num, "has 1 digit")
    } else {
        fmt.Println(num, "has multiple digits")
    }
}
```

```
$ go run if-else.go
7 is odd
8 is divisible by 4
9 has 1 digit
```

For loops in Go

- For is Go's only looping construct. Here are three basic types of for loops.
- The most basic type with a single condition
- A classic initial/condition/after for loop
- For without a condition will loop repeatedly until you break out of the loop or return from the enclosing function.

```go
package main

import "fmt"

func main() {

    i := 1
    for i <= 3 {
        fmt.Println(i)
        i = i + 1
    }

    for j := 7; j <= 9; j++ {
        fmt.Println(j)
    }

    for {
        fmt.Println("loop")
        break
    }

    for n := 0; n <= 5; n++ {
        if n%2 == 0 {
            continue
        }
        fmt.Println(n)
    }
}
```

```
$ go run for.go
1
2
3
7
8
9
loop
1
3
5
```

Arrays/Slices in Go

- Slices are typed only by the elements they contain (not the number of elements)
- To create an empty slice with non-zero length, use the builtin make.
- We can get and set just like with arrays
- Len returns the length of the slice as expected
- Slices support a “slice” operator with the syntax slice[low:high]
- We can declare and initialize a variable for slice in a single line as well.

```go
package main

import "fmt"

func main() {

    s := make([]string, 3)
    fmt.Println("emp:", s)



    s[0] = "a"
    s[1] = "b"
    s[2] = "c"
    fmt.Println("set:", s)
    fmt.Println("get:", s[2])

    fmt.Println("len:", len(s))

    s = append(s, "d")
    s = append(s, "e", "f")
    fmt.Println("apd:", s)



    c := make([]string, len(s))
    copy(c, s)
    fmt.Println("cpy:", c)

    l := s[2:5]
    fmt.Println("sl1:", l)


    l = s[:5]
    fmt.Println("sl2:", l)

    l = s[2:]
    fmt.Println("sl3:", l)

    t := []string{"g", "h", "i"}
    fmt.Println("dcl:", t)

    twoD := make([][]int, 3)
    for i := 0; i < 3; i++ {
        innerLen := i + 1
        twoD[i] = make([]int, innerLen)
        for j := 0; j < innerLen; j++ {
            twoD[i][j] = i + j
        }
    }
    fmt.Println("2d: ", twoD)
}
```

Maps in Go

- To create an empty map, use the built-in make: make(map[key-type]val-type).
- Set key/value pairs using typical name[key] = val syntax.
- Printing a map with e.g fmt. Println will show all of its key/value pairs.
- You can also declare and initialize a new map in the same line with this syntax.
- Note that maps appear in the form `map[k:v k:v]` when printed with `fmt.Println`.

```go
package main

import "fmt"

func main() {

    m := make(map[string]int)


    m["k1"] = 7
    m["k2"] = 13

    fmt.Println("map:", m)


    v1 := m["k1"]
    fmt.Println("v1: ", v1)

    fmt.Println("len:", len(m))


    delete(m, "k2")
    fmt.Println("map:", m)

    _, prs := m["k2"]
    fmt.Println("prs:", prs)




    n := map[string]int{"foo": 1, "bar": 2}
    fmt.Println("map:", n)
}
```

Structs in Go

- Go's structs are typed collections of fields. They're useful for grouping data
- This person struct type has name and age fields
- NewPerson constructs a new person struct with the given name
- You can safely return a pointer to local variable as a local variable will survive the scope of the function
- This syntax creates a new struct. You can name the fields when initializing a struct. Access struct fields with a dot.

```go
package main

import "fmt"

type person struct {
    name string
    age  int
}

func NewPerson(name string) *person {


    p := person{name: name}
    p.age = 42
    return &p
}

func main() {

    fmt.Println(person{"Bob", 20})

    fmt.Println(person{name: "Alice", age: 30})

    fmt.Println(person{name: "Fred"})

    fmt.Println(&person{name: "Ann", age: 40})

    fmt.Println(NewPerson("Jon"))


    s := person{name: "Sean", age: 50}
    fmt.Println(s.name)

    sp := &s
    fmt.Println(sp.age)


    sp.age = 51
    fmt.Println(sp.age)
}
```

Interfaces in Go

- Interfaces are named collections of method signatures.
- For this example we'll implement this interface on rect and circle types
- To implement an interface in Go, we just need to implement all the methods in the interface. Here we implement geometry on rects.
- If a variable has an interface type, we just need to implement all the methods that are in the named interface.

```go
package main

import (
    "fmt"
    "math"
)

type geometry interface {
    area() float64
    perim() float64
}

type rect struct {
    width, height float64
}
type circle struct {
    radius float64
}

func (r rect) area() float64 {
    return r.width * r.height
}
func (r rect) perim() float64 {
    return 2*r.width + 2*r.height
}

func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}
func (c circle) perim() float64 {
    return 2 * math.Pi * c.radius
}

func measure(g geometry) {
    fmt.Println(g)
    fmt.Println(g.area())
    fmt.Println(g.perim())
}

func main() {
    r := rect{width: 3, height: 4}
    c := circle{radius: 5}

    measure(r)
    measure(c)
}
```

Errors in Go

- In Go it's idiomatic to communicate errors via an explicit, separate return value.
- By convention, errors are the last return value and have type error, a built-in interface
- A nil value in the error position indicates that there was no error.
- It's possible to use custom types as errors by implementing the Error() method on them.
- In this case we use &argError syntax to build a new struct, supplying values for the two fields arg and prob.
- The two loops below test out each of our error-returning functions.

```go
package main

import (
    "errors"
    "fmt"
)

func f1(arg int) (int, error) {
    if arg == 42 {

        return -1, errors.New("can't work with 42")


    }
    return arg + 3, nil
}

type argError struct {
    arg  int
    prob string
}

func (e *argError) Error() string {
    return fmt.Sprintf("%d - %s", e.arg, e.prob)
}

func f2(arg int) (int, error) {
    if arg == 42 {

        return -1, &argError{arg, "can't work with it"}
    }
    return arg + 3, nil
}

func main() {

    for _, i := range []int{7, 42} {
        if r, e := f1(i); e != nil {
            fmt.Println("f1 failed:", e)
        } else {
            fmt.Println("f1 worked:", r)
        }
    }
    for _, i := range []int{7, 42} {
        if r, e := f2(i); e != nil {
            fmt.Println("f2 failed:", e)
        } else {
            fmt.Println("f2 worked:", r)
        }
    }

    _, e := f2(42)
    if ae, ok := e.(*argError); ok {
        fmt.Println(ae.arg)
        fmt.Println(ae.prob)
    }
}
```

Lessons

- Go, also known as Golang, is a statically typed, compiled programming language designed at Google.
- Go is syntactically similar to the C language.
- Go is a programming language made for building large-scale, complex software.
- Go has a standard library of packages to support the development of Go programs.

Difficulties arisen

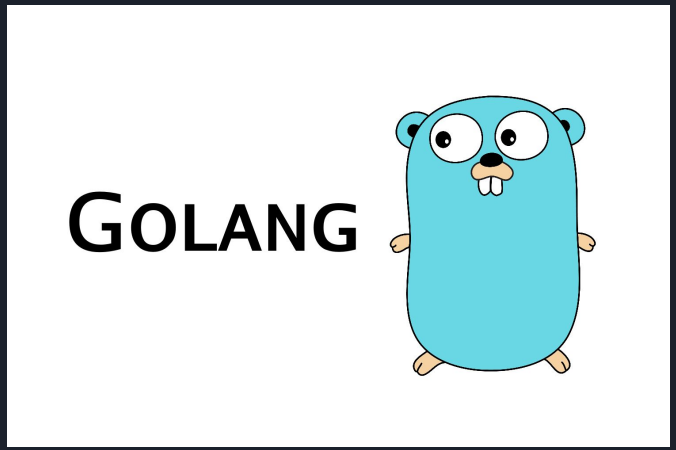
- At first some difficulty understanding how Golang works.

- Unfamiliarity of working with Go.

Conclusions about Go programming language

- Go is an interesting language. Perhaps in the future it will become very popular.

- Google is a very powerful corporation so they will probably make big efforts in popularising their language.



In what cases is Go recommended?

- Golang is a general-purpose language, it can be used for many things, such as:

-Firmware

-Desktop UI development

-Web frontend

-Mobile app development

-XML

-Games

-Banking, among other things...

THANKS!!