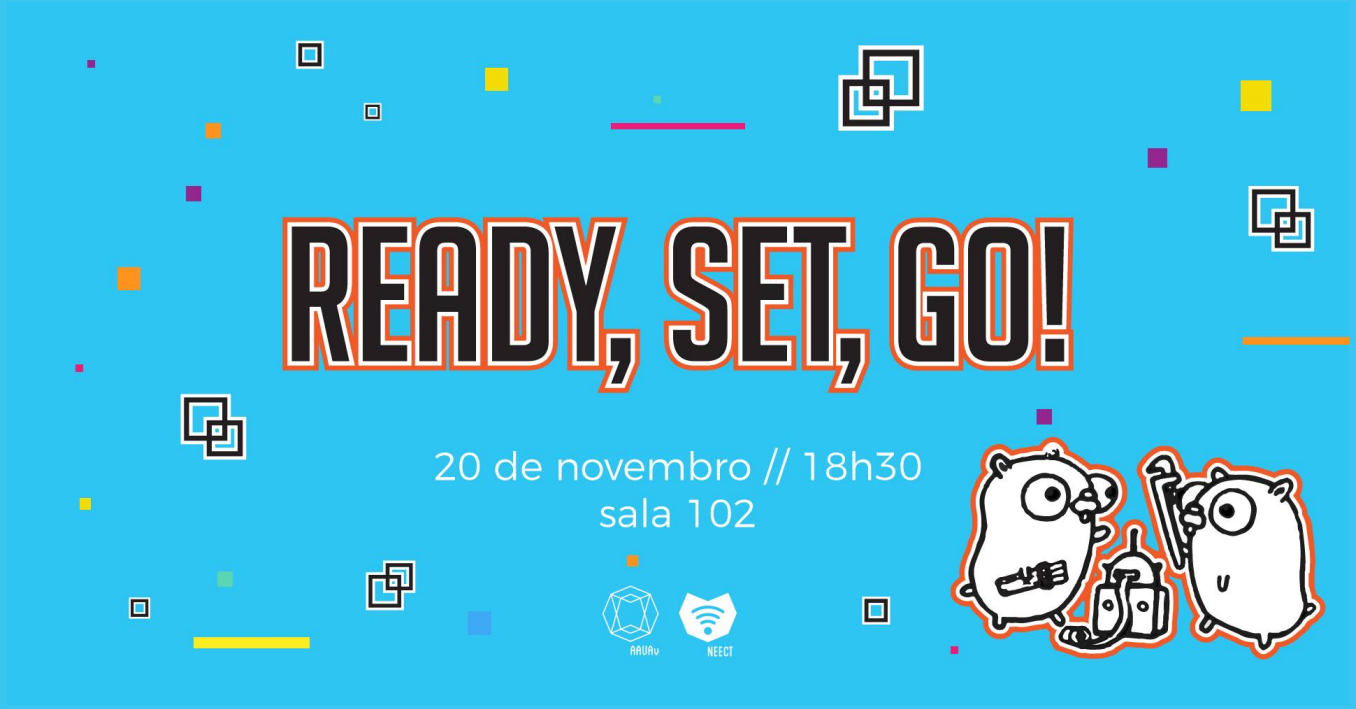

20/11/2018

Tiago Pires

Goals

- What is Go?
- Go basics (Part 1 and 2)
- Hello World
- Go command line
- Demo
- Workshop → Code and fun time!

What will be addressed

- Types
- Variables
- Control structures
- Arrays, slices and maps
- Functions
- Structs and Interfaces

What won't be addressed

- Pointers
- Concurrency
- Packages (creation and documentation)
- Testing
- Core packages
- Defer, panic & recover

Survey

- Have you already heard of Go language ?
- Have you already played with Go ?
- Have you written any code in Go ?
- Have you contributed to any open-source Go project ?

What is Go?

What is Go?*

- Open source programming language. Development began in 2007 at Google
- An attempt to bridge the gap between languages like Python and C internally in Google
- Three lead developers:
  - **Robert Griesemer** (Hacker, Computer programmer at Google)
  - **Rob Pike** (Hacker, Googler and writer; wrote "*The Practice of Programming*" and "*The Unix Programming Environment*")
  - **Ken Thompson** (Hacker, designed and implemented original Unix OS; invented B programming language)

*https://golang.org/doc/faq

What is Go?*

- Originally designed to be solely a systems language
- Go is intended to be fast: should take a few seconds to build a large execution on a single computer
- Easy to understand and learn
- Safety of statically typed language
- Compiled language (with runtime + garbage collection)
- Syntax and performance very similar to C
- First-class support for concurrency and parallelism
- Modern, with support for network and of course multicore computing

*https://golang.org/doc/faq

What is Go?*

Some of the challenges:

- Expressive but lightweight type system
- Type hierarchy
- Concurrency and garbage collection
- Rigid dependency specification
- Cross-languages builds
- Uncontrolled dependencies
- More at https://talks.golang.org/2012/splash.article

*https://golang.org/doc/faq

Go basics
Part 1

Go basics - Design Principles

- No forward declarations and no header files
- Everything is declared exactly once
- Initialization is expressive, automatic, and easy to use
- Clean syntax and light on keywords
- No generic types*
- No type hierarchy
- Easier to understand what happens by keeping concepts orthogonal:
  - Methods can be implemented for any type
  - Structures represent data
  - Interfaces represent abstraction
- Focused on scalability, readability and concurrency

*for now, topic is still open for discussion.

Go basics - Types

- Numbers
  - Integer - uint8, uint16, uint32, uint64, int8, int16, int32 and int64.
  - Floating point numbers
    - float32 and float64 (aka as single and double precision)
    - complex32 and complex64 (aka numbers with imaginary parts)
- Strings
  - “Hello”
  - “Index” starts at 0 (not 1)
- Boolean
  - true or false
  - 3 logical operators: && (and), || (or) or ! (not)

Go basics - Variables

- What is a variable?
  - A variable is storage location, with a specific type and an associated name.

```
var myString string
      name    type
```

- Attribution
  - `myString = "Hello World"`
- Short statement for creation and attribution
  - `myNewString := "Hello World"`

Go basics - Variables

- Comparison

```
myString == myNewString
true
```

- Addition (subtract not supported on strings)

```
myString += "! Go is awesome!"
```

- Subtract

```
a := 10
a -= 5
```

Go basics - Naming variables

- Variable name must start with a letter and may contain letters, numbers or _ (underscore) symbol
- Be clear and concise
- Should describe its purpose
- If it has multiple words, use lower camel case (aka mixed case)

Bad

```
a := 18
fmt.Println("Your age is", a)
```

Good

```
personAge := 18
fmt.Println("Your age is", personAge)
```

Go basics - Multiple variables

- Defining multiple variables at once

```
var (
        myString = “Hello”
        yourString = “World”
)
```

Go basics - Constants

- Basically are variables whose values cannot be changed (later, during execution)
- Instead of *var*, use *const* keyword

```
const helloConst string = “Hello World”
```

- Good way to reuse common values (e.g. *Pi* in *math* package is a constant)

Hello World

Hello World

```
package main

import "fmt"

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

Hello World

```
package main ← Go programs are organised as packages.

import "fmt" ← Allows you to use external code. fmt package provides
standard library allowing you to format and output data.

func main() { ← Main function is what gets executed - just like C.
  fmt.Println("Hello World!")
}
```

Go basics
Part 2

Scope

- Changing hello world a little bit. myString can only be accessed on main.

```
package main

import "fmt"

func main() {
  var myString = "Hello World!"
  fmt.Println(myString)
}
```

Scope

- On example below, others functions can access myString.

```
package main

import "fmt"

var myString = "Hello World!"

func main() {
  fmt.Println(myString)
}
```

Control structures

- In what other way could we implement the example below?

```
package main

import "fmt"

func main() {
  fmt.Println(1)
  fmt.Println(2)
  fmt.Println(3)
  fmt.Println(4)
}
```

Control structures - Loop

- Go only has one type of loop that can be used in different ways

```
package main

import "fmt"

func main() {
  for i := 1; i <= 10; i++ {
      fmt.Println(i)
  }
}
```

Control structures - Conditional

```
package main

import "fmt"

func main() {
  for i := 1; i <= 10; i++ {
      if i % 2 == 0 {
            fmt.Println(i, “is even”)
      } else {
            fmt.Println(i, “is odd”)
      }
  }
}
```

Control structures - Conditional

```
package main

import "fmt"

func main() {
  for i := 1; i <= 10; i++ {
      if i % 2 == 0 {
            fmt.Println(i, “divisible by 2”)
      } else if i % 3 == 0 {
            fmt.Println(i, “divisible by 3”)
      }
  }
}
```

Control structures - Conditional

```
package main

import "fmt"

func main() {
  for i := 1; i <= 10; i++ {
      switch i {
            case 1: fmt.Println(“One”)
            case 2: fmt.Println(“Two”)
            default: fmt.Println(“Unknown number”)
      }
  }
}
```

Go basics - Built-in types

- Arrays
- Slices
- Map

Go basics - Built-in types

- Arrays - numbered sequence of elements of a single type with fixed length

```
var myArray [3]int
```

```
package main

import "fmt"

func main() {
        var myArray [3]int
        myArray[1] = 50
        fmt.Println(myArray)
  }
}
```

Go basics - Built-in types

- Arrays - numbered sequence of elements of a single type with fixed length

```
var myArray [3]int
```

```
package main

import "fmt"

func main() {
        var myArray [3]int
        myArray[1] = 50
        fmt.Println(myArray) → [0 50 0]
  }
}
```

Go basics - Built-in types

- How to get total size of an array?

```
package main

import "fmt"

func main() {
      var myArray [3]int
      myArray[1] = 50
      fmt.Println(len(myArray))
  }
}
```

Go basics - Built-in types

- How to get total size of an array?

```
package main

import "fmt"

func main() {
      var myArray [3]int
      myArray[1] = 50
      fmt.Println(len(myArray)) → 3
  }
}
```

Go basics - Built-in types

- Slices - segment of an array. Very similar to arrays but slices don't have a fixed length.

```
var mySlice []int
```

```
package main

import "fmt"

func main() {
        var mySlice []int
        mySlice = append(mySlice, 50)
        fmt.Println(mySlice)
  }
}
```

Go basics - Built-in types

- Slices - segment of an array. Very similar to arrays but slices don't have a fixed length.

```
var mySlice []int
```

```
package main

import "fmt"

func main() {
        var mySlice []int
        mySlice = append(mySlice, 50)
        fmt.Println(mySlice) → [50]
  }
}
```

Go basics - Built-in types

- Slices - segment of an array. Very similar to arrays but slices don't have a fixed length.

```
mySlice := make([]int, 3)
```

```
package main

import "fmt"

func main() {
        mySlice := make([]int, 3)
        mySlice = append(mySlice, 50)
        fmt.Println(mySlice)
  }
}
```

Go basics - Built-in types

- Slices - segment of an array. Very similar to arrays but slices don't have a fixed length.

```
mySlice := make([]int, 3)
```

```
package main

import "fmt"

func main() {
        mySlice := make([]int, 3)
        mySlice = append(mySlice, 50)
        fmt.Println(mySlice) → [0, 0, 0, 50]
  }
}
```

Go basics - Built-in types

- Map - unordered collection of key-value pairs. Aka associative array, hash table or dictionary.

```
var myMap map[string]int
```

```
package main

import "fmt"

func main() {
        var myMap map[string]int
        myMap[“key”] = 50
        fmt.Println(myMap)
    }
}
```

Go basics - Built-in types

- Map - unordered collection of key-value pairs. Aka associative array, hash table or dictionary.

```
var myMap map[string]int
```

```
package main

import "fmt"

func main() {
        var myMap map[string]int
        myMap["key"] = 50 → panic: assignment to entry in nil map
        fmt.Println(myMap)
    }
}
```

Go basics - Built-in types

- Map - unordered collection of key-value pairs. Aka associative array, hash table or dictionary.

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

```
package main

import "fmt"

func main() {
        myMap := make(map[string]int)
        myMap[“key”] = 50
        fmt.Println(myMap)
    }
}
```

Go basics - Built-in types

- Map - unordered collection of key-value pairs. Aka associative array, hash table or dictionary.

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

```
package main

import "fmt"

func main() {
        myMap := make(map[string]int)
        myMap["key"] = 50
        fmt.Println(myMap) → map[key:50]
    }
}
```

Go basics - Built-in types

Map actions

- Delete one item

```
delete(myMap, "key")
```

- Get one element and check if exist at the same time

```
value, ok := myMap["key"]
if ok {
    fmt.Println(value)
}
```

Go basics - Built-in types

Map actions

- Get one element and check if exist at the same time (better version)

```
if value, ok := myMap["key"]; ok {
    fmt.Println(value)
}
```

Go basics - Functions

Until now, the only function you've seen is the main one.

- Let's write a function that given an int sums 1 and returns.

```
func add (number int) int {
    return number + 1
}
```

Go basics - Functions

Until now, the only function you've seen is the main one.

- Let's write a function that given an int sums 1 and returns.

```
func add (number int) int { → starts with keyword func followed by its name. Parameters (input) are defined by (name type, name type … ). After parameters its the return type. Parameters and return type are known as function's signature.
    return number + 1 → function body with return statement. Return statement causes the function to immediately stop and return the value.
}
```

Go basics - Functions

- Is this valid?

```
func main ()() {
    return 1
}
```

Go basics - Functions

- Returning multiple values

```
func add (number int) (int, int) {
    return number + 1, number + 2
}
```

Go basics - Functions

- Variadic functions

```
func sum (args ...int) int {
	total := 0
	for _, v := range args {
		total += v
	}
	return total
}
(...)
numbers := []int{1,2,3}
fmt.Println(sum(numbers...))
```

Go basics - Functions

- Variadic functions

```
func sum (args ...int) int {
	total := 0
	for _, v := range args {
		total += v
	}
	return total
}
(...)
numbers := []int{1,2,3}
fmt.Println(sum(numbers...)) → 6
```

Go basics - Functions

- Is this valid? What is the result?

```
numbers := []int{1,2,3}
fmt.Println(sum(numbers[1:]...))
```

Go basics - Structs

- Structs - Type with named fields

```
type Person struct {
    name string
    age int
}
```

```
fmt.Println(Person{"Alice", 20})
```

Go basics - Structs

- Methods

```
type Person struct {
    name string
    age int
}
func (p *Person) Hello() {
    fmt.Println("Hello! My name is", p.name)
}
```

Go basics - Structs

- Methods

```
type Person struct {
    name string
    age int
}
func (p *Person) Hello() { → Before method name is the
“receiver”. A receiver as a name and a type - like a parameter -
but it’s going to be linked to the struct.
    fmt.Println(“Hello! My name is”, p.name)
}
```

Go basics - Interfaces

- Interfaces - Very similar to struct but instead of defining fields, we define a “method set”. A method set is a list of methods that a type must have in order to “implement” the interface.

```
type Human interface {
    Hello()
}
```

Go basics - Interfaces

- How to check if a struct implements a specific interface?

```
var _ Human = (*Person)(nil)
```

If the struct doesn't implement the Human interface, when compiling:

```
cannot use (*Person)(nil) (type *Person) as type Human in assignment:
*Person does not implement Human (missing Hello method)
```

Go command line

Go command line

- **go build** - Compiles go source code files
- **go run** - Compiles and executes
- **go fmt** - Format source code files in current directory
- **go get** - Verify if package(s) need to be downloaded. Download if needed.
- **go install** - Compiles and *installs* package(s) (throws error if package(s) are missing
- **go test** - Executes tests associated with project

DEMO TIME !

Best place to check and read Go documentation is
https://golang.org/pkg/

Presentation content based on:

- https://golang.org/doc/faq
- https://www.manning.com/books/go-in-action
- https://www.golang-book.com/

Thank you!

Questions?

Resources

Chatroom: https://tlk.io/mindera
Presentation: https://bit.ly/2KfubQo
Workshop: https://bit.ly/2PD0Vcp