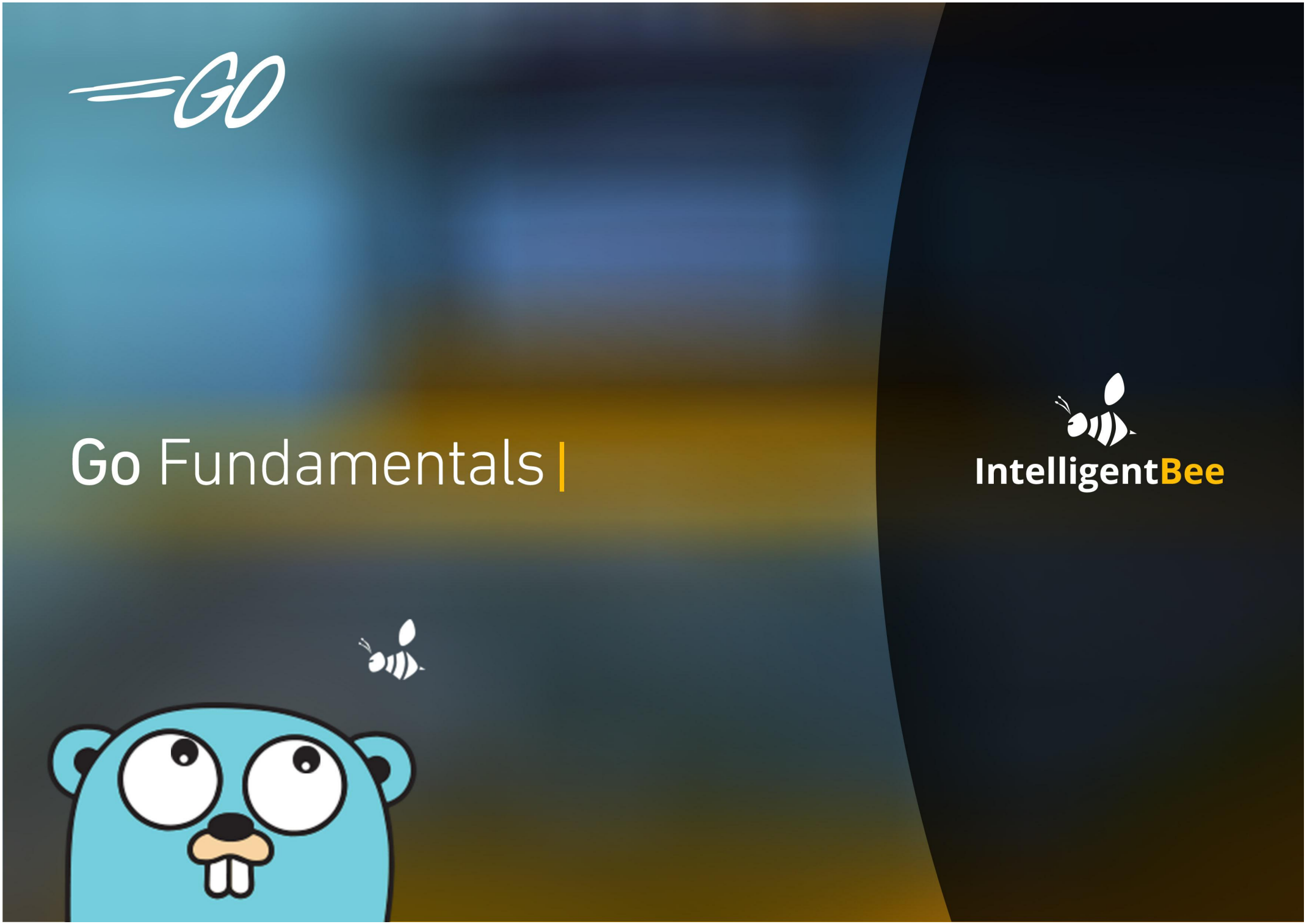
GO
Go Fundamentals |
IntelligentBee

Agenda

- General facts
- Running a Hello World
- Reasons to use Go
- Development environment setup
- Basics
- Flow control
- More types
- Methods and interfaces

Facts

- General Purpose Programming Language
- 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
- Object Oriented
  - Structs are classes
  - Inheritance is composition
  - Polymorphism through interfaces

Facts

- Compiled
- Garbage collected
- Strongly typed
- Built-in concurrency
- Fast build
- Pointers (without pointer aritmethic)

1.0 Hello, world!

```
package main

import "fmt"

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

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

2.0 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**

2.1 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
```

2.2 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

```
bin/
    hello                          # command executable
    outyet                         # command executable
pkg/
    linux_amd64/
        github.com/golang/example/
            stringutil.a           # package object
src/
    github.com/golang/example/
        .git/                      # Git repository metadata
        hello/
            hello.go               # command source

        golang.org/x/image/
        .git/                      # Git repository metadata
bmp/
   reader.go             # package source
   writer.go             # package source
... (many more repositories and packages omitted)
```

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, int) // "" flag 0
```

3.5 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, int) // "Hello Go World!" true 10
```

3.6 Variables...

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

```
var x = 10 //int
var flag, message = true, "Hello Go World!" //bool, string
```

- Short variable declaration

```
x := 10 //int
bool, message := true, "Hello Go World!" //bool, string
```

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])

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() {
  fmt.Println(salutation("Cosmin", "Hello")) // "Hello Cosmin"
}
```

3.9 Functions...

- Functions can also return multiple values

```
package main

func div(a, b int) (int, int) {
 Return a/b, a%b
}

func main(){
 v, r := div(5, 2)
 fmt.Println(v, r) // 2 1
}
```

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
}

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

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...

- 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 > 35 {
 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 := 0; i <10; i++ {
      sum += i
   }

   fmt.Println(sum)// 45
}
```

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) // 10
}
```

4.4 Switch

- The cases are evaluated top to bottom until a match is found
- Break statements can be used to end a switch early

```
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

Pointers

- A pointer holds the memory address of a value
- The type *`*T`* is a pointer to a *`T`* value. Its zero value is *`nil`*
- The **&** operator generates a pointer to its operand
- The * operator denotes the pointer's underlying value(known as *`dereferencing`* or *`indirecting`*)
- Unlike C, Go has no pointer arithmetic.

5.1 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{"Cosmin", 22}
 fmt.Println("%v\n", p) // {Cosmin 22}
 fmt.Println("%+v\n", p) // {Name:Cosmin Age:22}
 fmt.Println("%#v\n", p) // main.User{Name:"Cosmin", Age:22}
 fmt.Println("Name:", p.Name) // Name: Cosmin
}
```

5.2 Arrays

- An ordered container type with a fixed number of values
- The number of values in an array its 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"}
```

5.3 Arrays...

- The built-in *len* function returns the length of an array
- The expression *s[n]* accesses the nth element, starting from zero

```
colors := [3]string{"Red", "Green", "Blue"}
c := colors[1]
fmt.Println(c) // "Green"
```

5.4 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*

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

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

5.5 Slices...

- A slice does not store any data, it just describes a section of an underlying data
- Changing the elements of a slice modifies the corresponding elements of its underlying array
- A slice has both a `length` and a `capacity`
- The `length` of a slice is the number of elements it contains
- The `capacity` of a slice is the number of elements in the underlying array, counting from the first element of the slice
- `Length` can be obtained using *`len(s)`* and `capacity` using *`cap(s)`*

```
s := [6]int{2, 3, 5, 7, 11, 13}
fmt.Printf("len=%d cap=%d %v\n", len(s), cap(s), s) // len=6 cap=6 [2 3 5 7 11 13]
```

5.6 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
- If the backing array of *s* is too small to fit all the given values a bigger array will be allocated. The returned slice will point to the newly allocated array.

```
package main

      import "fmt"

      func main() {
            s := []int{0,1,2,3,4,5,6}
            fmt.Printf("%d %d %v\n", len(s), cap(s), s)

            s = s[:4]
            fmt.Printf("%d %d %v\n", len(s), cap(s), s)

            s = s[2:]
            fmt.Printf("%d %d %v\n", len(s), cap(s), s)
   }
```

5.7 Maps

- A map creates an association between a key and a value

```
var fruitWeigths = map[string]int {
 "Apple": 45,
 "Mango": 24,
 "Orange": 34
}
fmt.Printf("%#v\n", fruitWeights) // map[string]int{"Apple":45,"Mango":24, "Orange":34}
```

5.8 Maps...

- The zero value of a map is ***nil***
- The **make** function returns a map of the give type initialized and ready to use

```
var fruitWeights map[string]int
fmt.Printf("%#v\n", fruitWeights) // map[string]int(nil)
fruitWeights = make(map[string]int)
fmt.Printf("%#v\n", fruitWeights) // map[string]int{}
fruitWeights["Apple"] = 45
fruitWeights["Mango"] = 24
fruitWeights["Orange"] = 34
fmt.Printf("%#v\n", fruitWeights) // map[string]int{"Apple":45, "Mango":24, "Orange":34}
```

6.0 Methods and interfaces

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*

6.1 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()
}
```

6.2 Interfaces

- An `interface type` is defined as a set of method signatures
- A value of interface type can hold any value that implements those methods
- A type implements an interface by implementing its methods
- No "implements" keyword
- A type can implement multiple interfaces

```
package main

import "fmt"

type myInterface interface {
     Hello()
}

type myType struct {}

func (mt *myType) Hello() {
     fmt.Println("Hello from myType")
}

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

7.0 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())
     }
}
```

Thank you.

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