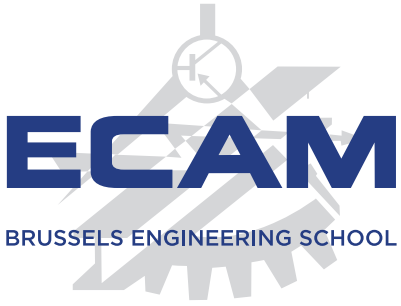


PL190μ Introduction to the
Go Programming Language

Session 2

Structure, Method, Interface and Error Handling

Sébastien Combéfis

Winter 2020

Objectives

- Basic functional programming features

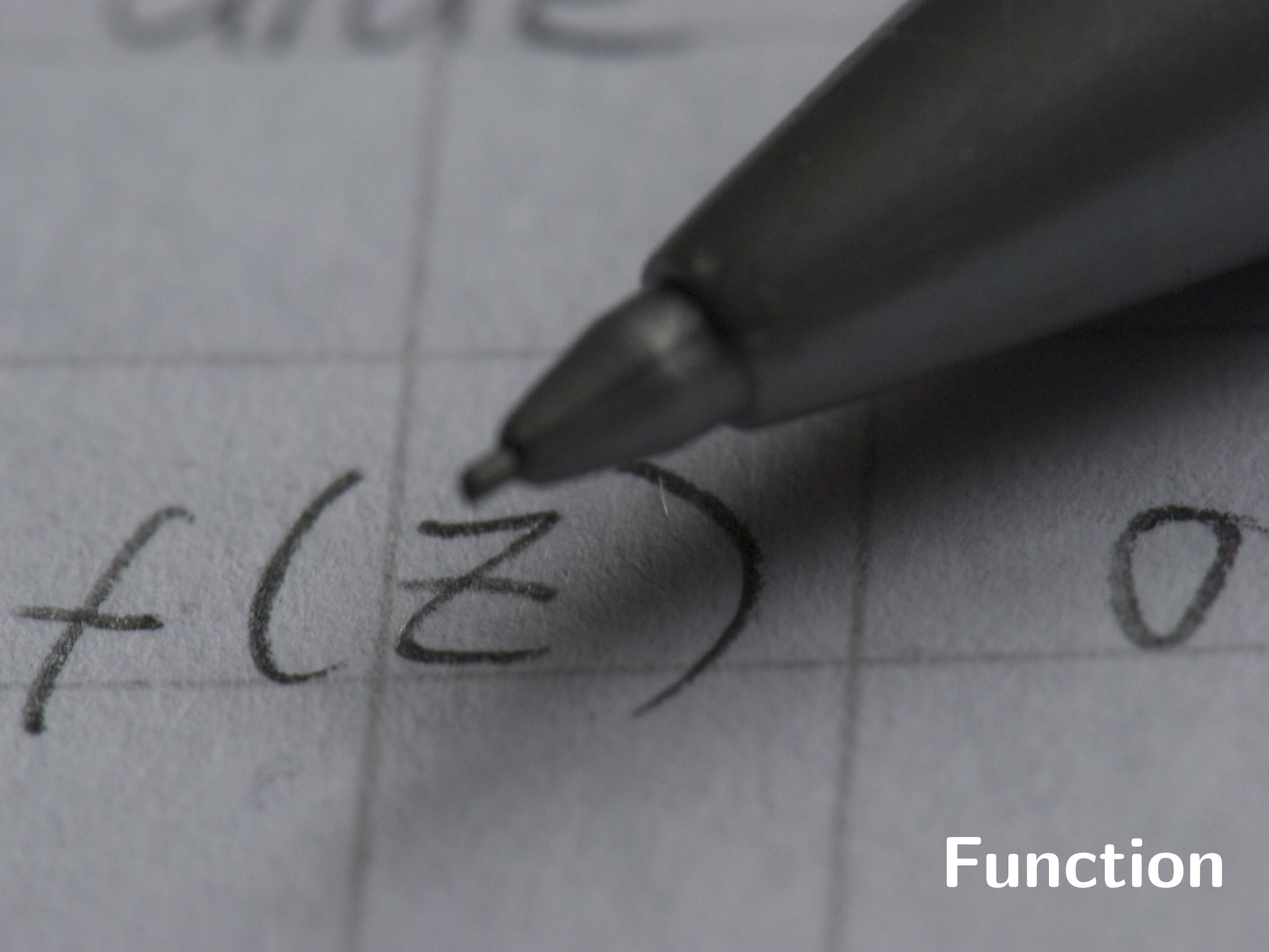
  The function type and using function as value or parameter

- Basic object oriented-like features

  Defining structure, methods acting on them and interfaces

- Basic error handling techniques and error generation

  Using the `error` type to capture and generate errors


f(z)
Function

Variadic Function

- Variadic function accepts a variable number of arguments
  - Handled inside the function as a slice with the arguments
  - Only the last argument of a function, identified with ...

```
1  func sum(data ...int) int {
2      result := 0
3      for _, elem := range data {
4          result += elem
5      }
6      return result
7  }
8
9  func main() {
10     fmt.Println(sum(1, 2, 3, 4))
11 }
```

```
10
```

Slice Unpacking

- Slice must be unpacked with ... to call variadic function

Impossible to directly call a variadic function with a slice

```
1  func sum(data ...int) int {
2      result := 0
3      for _, elem := range data {
4          result += elem
5      }
6      return result
7  }
8
9  func main() {
10     data := []int{1, 2, 3, 4}
11     fmt.Println(sum(data...))
12 }
```

```
10
```

Function Value

- Function is a type, allowing functional programming features

Function can be put in a variable and used as a argument

```
1  func main() {
2      max := func(a int, b int) int {
3          if a > b {
4              return a
5          }
6          return b
7      }
8      fmt.Println(max)
9      fmt.Println(max(7, 12))
10 }
```

```
0x10995b0
12
```

Function Parameter

- A function can take a function argument

  Declared inline, stored in a variable, defined separately

```
1  func apply(f func(int) int, data []int) {
2      for i, v := range data {
3          data[i] = f(v)
4      }
5  }
6
7  func main() {
8      data := []int{1, 2, 3}
9      apply(func(e int) int { return e * e }, data)
10     fmt.Println(data)
11 }
```

```
[1 4 9]
```

Structure


43
Hängande last
Stora kroken!
Lyftkraft 12 000 kg Radie 36-10 m
Lyftkraft 20 000 kg Radie 28-10 m
Lyftkraft 40 000 kg Radie 21-10 m
Lilla kroken!
Lyftkraft 6000 kg Radie 39-11.5 m

Defining Structure

- Several data fields can be gathered into a single structure

  Accessing/changing values with different types from one variable

- New structure defined with `struct`
  - Structure can be given a name with the `type` construct
  - Structure can be defined locally or globally (most common)

```
1 type Restaurant struct {
2     Name    string
3     Rating  float64
4     Labels  []string
5 }
```

Creating Structure

- Structure created by providing values for each field

  Either on a single line or one field per line, field can be omitted

- Fields initialised with default value if not specified

```
1  func main() {
2      var a Restaurant
3      b := Restaurant{
4          Name:   "Osteria Francescana",
5          Rating: 4.6,
6          Labels: []string{"Italian"},
7      }
8
9      fmt.Println(a)
10     fmt.Println(b)
11 }
```

```
{ 0 []}
{Osteria Francescana 4.6 [Italian]}
```

Accessing Field

- Fields of a structure accessed with the dot operator

  Field values of a structure can be read or modified

- Detailed string formatting of a structure with %+v modifier

  When using the `Printf` function of the `fmt` package

```
1  func main() {
2      a := Restaurant{Name: "Osteria Francescana"}
3      a.Rating = 4.6
4      a.Labels = append(a.Labels, "Italian")
5
6      fmt.Printf("%+v\n", a)
7  }
```

```
{Name:Osteria Francescana Rating:4.6 Labels:[Italian]}
```

Nested Structure (1)

- It is possible to nest a structure in another one

  The type of the field of a structure can be a structure

- Inner structure can be defined directly in the outer one

```
1  type Restaurant struct {
2      Name     string
3      Rating   float64
4      Labels   []string
5      Address  Address
6  }
7
8  type Address struct {
9      Street   string
10     Number   int
11     ZipCode  int
12     City     string
13     Country  string
14 }
```

Nested Structure (2)

- Nested structure can be initialised with the outer one

  Can also be defined separately and then used in the outer one

```
1  func main() {
2      a := Address{"Via Stella", 22, 41121, "Modena", "Italy"}
3      b := Restaurant{
4          Name:    "Osteria Francescana",
5          Rating:  4.6,
6          Labels:  []string{"Italian"},
7          Address: a,
8      }
9
10     fmt.Printf("%+v\n", b)
11 }
```

```
{Name:Osteria Francescana Rating:4.6 Labels:[Italian] Address:{
Street:Via Stella Number:22 ZipCode:41121 City:Modena Country:
Italy}}
```

Copying Structure

- Assigning a structure to a new variable creates a copy

  Same when nesting a structure, passing a parameter to a function

```
1  func main() {
2      a := Address{City: "Modena", Country: "Italy"}
3
4      b := a
5      b.Country = "Belgium"
6
7      fmt.Printf("%+v\n", a)
8      fmt.Printf("%+v\n", b)
9  }
```

```
{Street: Number:0 ZipCode:0 City:Modena Country:Italy}
{Street: Number:0 ZipCode:0 City:Modena Country:Belgium}
```

Pointer to Structure

- Pointer used to store a reference to a structure

  Simply obtained with reference operator &

```
1  func main() {
2      a := Address{City: "Modena", Country: "Italy"}
3
4      b := &a
5      b.Country = "Belgium"
6
7      fmt.Printf("%+v\n", a)
8      fmt.Printf("%+v\n", b)
9  }
```

```
{Street: Number:0 ZipCode:0 City:Modena Country:Belgium}
&{Street: Number:0 ZipCode:0 City:Modena Country:Belgium}
```

Method

Method

- A method associates a function to a structure

  Defined by adding a receiver structure in the function signature

- The receiver is a reference to a structure

  The receiver structure can be modified in the body of the method

```
1  func (r *Restaurant) hasLabel(label string) bool {
2      for _, v := range r.Labels {
3          if v == label {
4              return true
5          }
6      }
7      return false
8  }
```

Method Call

- A method is executed by calling it on a receiver structure

  A reference to the structure is passed to the method

```
1  func main() {
2      a := Address{"Via Stella", 22, 41121, "Modena", "Italy"}
3      b := Restaurant{
4          Name:    "Osteria Francescana",
5          Rating:  4.6,
6          Labels:  []string{"Italian"},
7          Address: a,
8      }
9
10     fmt.Println(b.hasLabel("Italian"))
11     fmt.Println(b.hasLabel("Fast-Food"))
12 }
```

```
true
false
```

Interface

- An interface is used to define a method set

  Construction used to introduce modularity in a Go program

- Definition of a set of methods with their specifications

  Provide abstraction to decouple a code from implementation

```
1  type Shape interface {
2      Perimeter() float64
3      Area() float64
4  }
```

Implementing Interface

- Implementing an interface means defining all its methods

The structure will implicitly implements the interface

```
1  type Square struct {
2      Side float64
3  }
4
5  func (s *Square) Perimeter() float64 {
6      return 4 * s.Side
7  }
8
9  func (s *Square) Area() float64 {
10     return s.Side * s.Side
11 }
```

Using Interface

- An interface can be used as a type for a variable/parameter

  Any structure implementing the interface can be assigned

- Interface variable/parameter must be assigned with a pointer

  Reference operator can be used to call function taking interface

```
1  func print(s Shape) {
2      fmt.Printf("Perimeter: %f\nArea: %f\n", s.Perimeter(), s.Area())
3  }
4
5  func main() {
6      a := Square{5}
7      print(&a)
8  }
```

```
Perimeter: 20.000000
Area: 25.000000
```

Handling Error

`error` Type

- Errors are represented by the `error` type

  Interface defining a single `Error` method returning a `string`

- Typically returned when calling functions likely to fail

  Can contain a specific message explaining the error

```
1  type error interface {
2     Error() string
3  }
```

Checking Error

- Idiomatic way to proceed is to check error and fail early

  Return or exit the executing code as early as possible

- Compared to `nil` to check whether an error occurred

  A value different of `nil` indicated an error

```
1  func main() {
2      content, err := ioutil.ReadFile("data.txt")
3      if err != nil {
4          fmt.Println("An error occurred while reading the file.")
5          return
6      }
7      fmt.Printf("%s\n", content)
8  }
```

Creating Error

- New error can be created with `New` function of package `errors`

  Specifying a message describing the error as the only parameter

- Function likely to produce an error have an `error` return value

  Can be either checked or ignored when calling the function

```
1  func fact(n int) (int, error) {
2      if n < 0 {
3          return -1, errors.New("Fact. of negative number undefined.")
4      }
5      if n == 0 {
6          return 1, nil
7      }
8
9      prev, _ := fact(n - 1)
10     return n * prev, nil
11 }
```

Panic!

- Possible to hard stop the execution of a program with `panic`

  Immediately stops the execution and prints a message

```go
func main() {
    result, err := fact(-1)
    if err != nil {
        panic(fmt.Sprintf("Error: %s", err))
    }
    fmt.Printf("Fact -1 is: %d\n", result)
}
```

```
panic: Error: Cannot compute fact of negative number

goroutine 1 [running]:
main.main()
    /tmp/work/test-go/main.go:22 +0x142
```

References

- George Ornbo (2017). *Sams Teach Yourself Go in 24 Hours: Next Generation Systems Programming with Golang*, Pearson, ISBN: 978-0-672-33803-8.
- Caleb Doxsey (2012). *An Introduction to Programming in Go*, , ISBN: 978-1-478-35582-3.
- Himanshu Singh (2018). *Let's Go ~ A Complete Guide*, August 9, 2018.
  https://medium.com/mindorks/lets-go-a-complete-guide-147aec23fd5a
- Jerry J. Muzsik (2017). *A Space Themed Intro to Golang*, December 27, 2017.
  https://hackernoon.com/golang-the-highest-paying-technology-to-know-9c6089d7081d
- Kevin Goslar (2019). *Go is on a trajectory to become the next enterprise programming language*, April 10, 2019.
  https://hackernoon.com/go-is-on-a-trajectory-to-become-the-next-enterprise-programming-language-3b75d70544e
- Uday Hiwarale (2018). *Variadic functions in Go*, October 14, 2018.
  https://medium.com/rungo/variadic-function-in-go-5d9b23f4c01a

Credits

- Vestman, August 14, 2010, https://www.flickr.com/photos/vestman/4908148942.
- Blondinrikard Fröberg, August 6, 2015, https://www.flickr.com/photos/blondinrikard/19725253793.
- Gino Zahnd, March 9, 2011, https://www.flickr.com/photos/gzahnd/5513609336.
- Eric Jusino, December 10, 2009, https://www.flickr.com/photos/mangpages/5042888638.