Go course

Petr Shevtsov

2019-05-13

Contents

Preface **5**

1 Basic Concepts **7**
- 1.1 What is Go? . . . 7
- 1.2 Hello, world! . . . 7
- 1.3 The Go CLI . . . 7
- 1.4 Value types . . . 7
- 1.5 Variables . . . 7
- 1.6 Operators . . . 7
- 1.7 Constants . . . 7
- 1.8 Comments . . . 7
- 1.9 Packages and imports . . . 7
- Module project . . . 7

2 Conditionals and Loops **9**
- 2.1 The `if` statement . . . 9
- 2.2 The `else` statement . . . 9
- 2.3 `if`/`else` chains . . . 9
- 2.4 The `if` statement with expression . . . 9
- 2.5 The `switch` statement . . . 9
- 2.6 The `switch` without condition . . . 9
- 2.7 The `for` statement . . . 9
- 2.8 The `defer` statement . . . 9
- Module project . . . 9

3 Composite Data Types **11**
- 3.1 Arrays . . . 11
- 3.2 Loops and arrays . . . 11
- 3.3 Slices . . . 11
- 3.4 Appending items to slices . . . 11
- 3.5 Range . . . 11
- 3.6 Maps . . . 11
- 3.7 Arrays vs maps . . . 11
- 3.8 Structs . . . 11
- 3.9 Struct literals . . . 11
- 3.10 Operations with structs . . . 11
- Module project . . . 11

4 Functions and pointers **15**
- 4.1 Function declaration . . . 15
- 4.2 Functions parameters . . . 15
- 4.3 Return values . . . 15

4.4 Error handling . . . 15
4.5 Variadic functions . . . 15
4.6 Iteration and recursion . . . 15
4.7 Anonymous functions . . . 15
4.8 Panic . . . 15
4.9 Pointers . . . 15
4.10 Functions and pointers . . . 15
Module project . . . 15

5 Methods **17**
5.1 Method declarations . . . 17
5.2 Methods with a pointer receiver . . . 17
5.3 Composing types with structs . . . 17
5.4 Working with struct methods . . . 17
5.5 Method values . . . 17
5.6 Method expressions . . . 17
5.7 Encapsulation . . . 17

6 Interfaces **19**
6.1 Introduction . . . 19
6.2 Interface types . . . 19
6.3 Satisfaction . . . 19
6.4 `flag.Value` . . . 19
6.5 Interface values . . . 19
6.6 Sorting with `sort.Interface` . . . 19
6.7 The `error` interface . . . 19
6.8 Type assertions . . . 19
6.9 Type switches . . . 19

7 Goroutines and channels **21**
7.1 What is goroutine . . . 21
7.2 Introduction to concurrency . . . 21
7.3 Channels . . . 21
7.4 Types of channels . . . 21
7.5 Pipelines . . . 21
7.6 Looping in parallel . . . 21
7.7 `time.Tick` . . . 21
7.8 The `select` statement . . . 21
7.9 Cancellation . . . 21
Module project . . . 21

Preface

The course is targeted at a beginner level student new to Go but might be familiar with 1-2 other languages (e.g. Python, HTML).

Module 1

Basic Concepts

1.1 What is Go?

1.2 Hello, world!

1.3 The Go CLI

1.4 Value types

1.5 Variables

1.6 Operators

1.7 Constants

1.8 Comments

1.9 Packages and imports

Module project

```go
// This is a comment.
// Every Go file must be a part of some package.
// This file is a part of package main.
package main

// We import package "fmt" from the standard Go library.
import "fmt"

// `who` is a constant.
```

```go
const who = "world"

// Function main() is the main entry point of any application written in Go.
func main() {
    // We declare `greeting` as a variable of type string and assign the value.
    var greeting string = "Hello"
    // We declare `message` variable using the shorthand syntax. The type of
    // the variable is determined by the assigned value. In our case it is
    // string type.
    message := greeting + ", " + who
    // Let's print the value of the variable `message` using the function from
    // the package "fmt".
    fmt.Println(message)
}
```

```
## Hello, world
```

Module 2

Conditionals and Loops

2.1 The `if` statement

2.2 The `else` statement

2.3 `if`/`else` chains

2.4 The `if` statement with expression

2.5 The `switch` statement

2.6 The `switch` without condition

2.7 The `for` statement

2.8 The `defer` statement

Module project

```go
package main

import "fmt"

func main() {
    for age := 0; age < 99; age++ {
        switch age {
        case 16:
            fmt.Println("When you're", age, "you can drive a car!")
        case 18:
            fmt.Println("When you're", age, "you can buy a lottery ticket!")
        case 21:
```

```
            fmt.Println("When you're", age, "you can buy some beer!")
            break
        default:
            continue
        }
    }
}
```

```
## When you're 16 you can drive a car!
## When you're 18 you can buy a lottery ticket!
## When you're 21 you can buy some beer!
```

Module 3

Composite Data Types

3.1 Arrays

3.2 Loops and arrays

3.3 Slices

3.4 Appending items to slices

3.5 Range

3.6 Maps

3.7 Arrays vs maps

3.8 Structs

3.9 Struct literals

3.10 Operations with structs

Module project

```go
package main

import (
	"fmt"
	"strings"
)
```

```
const Shakespeare = `
From fairest creatures we desire increase,
That thereby beauty's rose might never die,
But as the riper should by time decease,
His tender heir might bear his memory:
But thou contracted to thine own bright eyes,
Feed'st thy light's flame with self-substantial fuel,
Making a famine where abundance lies,
Thy self thy foe, to thy sweet self too cruel:
Thou that art now the world's fresh ornament,
And only herald to the gaudy spring,
Within thine own bud buriest thy content,
And tender churl mak'st waste in niggarding:
  Pity the world, or else this glutton be,
  To eat the world's due, by the grave and thee.
`

func main() {
    wordCount := make(map[string]int)
    for _, word := range strings.Fields(Shakespeare) {
        word = strings.Trim(word, ",:.")
        word = strings.ToLower(word)
        count := wordCount[word]
        count++
        wordCount[word] = count
    }

    for word, count := range wordCount {
        fmt.Println(word, count)
    }
}
```

```
## but 2
## waste 1
## in 1
## self-substantial 1
## fuel 1
## die 1
## as 1
## his 2
## bear 1
## thou 2
## feed'st 1
## niggarding 1
## tender 2
## with 1
## spring 1
## or 1
## grave 1
## might 2
## should 1
## bright 1
## cruel 1
```

```
## now 1
## churl 1
## rose 1
## eat 1
## the 6
## decease 1
## thine 2
## art 1
## content 1
## memory 1
## own 2
## self 2
## gaudy 1
## mak'st 1
## lies 1
## herald 1
## too 1
## never 1
## time 1
## to 4
## thy 5
## flame 1
## where 1
## creatures 1
## a 1
## thee 1
## riper 1
## world's 2
## ornament 1
## only 1
## world 1
## this 1
## bud 1
## buriest 1
## desire 1
## by 2
## heir 1
## famine 1
## abundance 1
## and 3
## pity 1
## be 1
## fairest 1
## increase 1
## contracted 1
## fresh 1
## else 1
## we 1
## light's 1
## sweet 1
## within 1
## due 1
## foe 1
## glutton 1
```

```
## from 1
## that 2
## thereby 1
## beauty's 1
## eyes 1
## making 1
```

Module 4

Functions and pointers

4.1 Function declaration

4.2 Functions parameters

4.3 Return values

4.4 Error handling

4.5 Variadic functions

4.6 Iteration and recursion

4.7 Anonymous functions

4.8 Panic

4.9 Pointers

4.10 Functions and pointers

Module project

```go
package main

import "fmt"

// fibonacci returns the nth Fibonacci number.
func fibonacci(n int) int {
```

```
    if n < 2 {
        return n
    }
    return fibonacci(n-1) + fibonacci(n-2)
}

// fibonacciSequence changes the int slice to make it contain the Fibonacci
// numbers according to its keys. This function operates on the actual slice,
// that's why it does not return anything.
func fibonacciSequence(slice []int) {
    for n := range slice {
        slice[n] = fibonacci(n)
    }
}

func main() {
    // Create an empty int slice of length 10
    sequence := make([]int, 10)
    // Fill the slice with Fibonacci numbers secuence
    fibonacciSequence(sequence)

    fmt.Println(sequence)
}
```

```
## [0 1 1 2 3 5 8 13 21 34]
```

Module 5

Methods

5.1 Method declarations

5.2 Methods with a pointer receiver

5.3 Composing types with structs

5.4 Working with struct methods

5.5 Method values

5.6 Method expressions

5.7 Encapsulation

Module 6

Interfaces

6.1 Introduction

6.2 Interface types

6.3 Satisfaction

6.4 `flag.Value`

6.5 Interface values

6.6 Sorting with `sort.Interface`

6.7 The `error` interface

6.8 Type assertions

6.9 Type switches

Module 7

Goroutines and channels

7.1 What is goroutine

7.2 Introduction to concurrency

7.3 Channels

7.4 Types of channels

7.5 Pipelines

7.6 Looping in parallel

7.7 `time.Tick`

7.8 The `select` statement

7.9 Cancellation

Module project

```
package main

import (
    "fmt"
    "math/rand"
    "sync"
    "time"
)
```

```go
const (
    NumberOfRacers   = 10
    NumberOfLaps     = 3
    MaxSleepDuration = 3 // seconds
)

func init() {
    rand.Seed(time.Now().UnixNano())
}

func race(racer int, start chan struct{}, finish chan int, status chan []int, wg *sync.WaitGroup) {
    defer wg.Done()
    <-start
    for lap := 1; lap <= NumberOfLaps; lap++ {
        sleep := time.Duration(rand.Intn(MaxSleepDuration))
        time.Sleep(sleep * time.Second)
        go func(racer, lap int) {
            status <- []int{racer, lap}
        }(racer, lap)
    }
    finish <- racer
}

func main() {
    start := make(chan struct{})
    finish := make(chan int)
    status := make(chan []int)
    done := make(chan struct{})
    var wg sync.WaitGroup
    wg.Add(NumberOfRacers)
    for racer := 1; racer <= NumberOfRacers; racer++ {
        go race(racer, start, finish, status, &wg)
    }

    go func() {
        wg.Wait()
        close(done)
    }()

    startTime := time.Now()
    close(start)

    ticker := time.NewTicker(time.Second)
    defer ticker.Stop()

    var winners []int

OuterLoop:
    for {
        select {
        case t := <-ticker.C:
            fmt.Println("Race time:", t.Sub(startTime))
        case s := <-status:
```

```
            fmt.Printf("Racer #%d is on lap %d\n", s[0], s[1])
        case finished := <-finish:
            fmt.Printf("Racer #%d finished!\n", finished)
            if len(winners) < 3 {
                winners = append(winners, finished)
            }
        case <-done:
            break OuterLoop
        }
    }

    close(finish)
    close(status)

    fmt.Println("\nWinners:")
    place := 1
    for _, racer := range winners {
        fmt.Printf("%d place: Racer #%d\n", place, racer)
        place++
    }
}
```

```
## Racer #10 is on lap 2
## Racer #7 is on lap 1
## Racer #9 is on lap 2
## Racer #9 is on lap 1
## Racer #10 is on lap 1
## Race time: 1.000174895s
## Racer #8 is on lap 1
## Racer #1 finished!
## Racer #1 is on lap 1
## Racer #1 is on lap 2
## Racer #1 is on lap 3
## Racer #7 finished!
## Racer #7 is on lap 2
## Racer #7 is on lap 3
## Racer #6 finished!
## Racer #6 is on lap 1
## Racer #6 is on lap 2
## Racer #6 is on lap 3
## Race time: 2.000160288s
## Racer #2 is on lap 1
## Racer #10 finished!
## Racer #4 is on lap 1
## Racer #3 is on lap 2
## Racer #5 is on lap 1
## Racer #10 is on lap 3
## Racer #3 is on lap 1
## Racer #8 is on lap 2
## Racer #9 finished!
## Racer #9 is on lap 3
## Race time: 3.000149888s
## Racer #5 is on lap 2
## Racer #3 finished!
```

```
## Racer #3 is on lap 3
## Race time: 4.00017553s
## Racer #8 finished!
## Racer #2 is on lap 2
## Racer #4 finished!
## Racer #8 is on lap 3
## Racer #4 is on lap 2
## Racer #4 is on lap 3
## Race time: 5.000275273s
## Racer #5 finished!
## Racer #2 finished!
## Racer #5 is on lap 3
## Racer #2 is on lap 3
##
## Winners:
## 1 place: Racer #1
## 2 place: Racer #7
## 3 place: Racer #6
```