Go course

Petr Shevtsov

2019-05-16

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
Module project . . . 17

6 Interfaces 21
6.1 Introduction . . . 21
6.2 Interface types . . . 21
6.3 Satisfaction . . . 21
6.4 `flag.Value` . . . 21
6.5 Interface values . . . 21
6.6 Sorting with `sort.Interface` . . . 21
6.7 The `error` interface . . . 21
6.8 Type assertions . . . 21
6.9 Type switches . . . 21
Module project . . . 21

7 Goroutines and channels 27
7.1 What is goroutine . . . 27
7.2 Introduction to concurrency . . . 27
7.3 Channels . . . 27
7.4 Types of channels . . . 27
7.5 Pipelines . . . 27
7.6 Looping in parallel . . . 27
7.7 `time.Tick` . . . 27
7.8 The `select` statement . . . 27
7.9 Cancellation . . . 27
Module project . . . 27

Course project 31

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.
```

```
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() {
    // Here we loop from 0 to 99 and only output some sentences for particular
    // cases.
    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"
)
```

```go
// Shakespeare contains the text of one of the Shakespeare's sonets. We'll use
// this text to count words in it.
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.
`

// getWord returns a word in lower case and with trimmed punctuation.
func getWord(s string) string {
    return strings.ToLower(strings.Trim(s, ",:."))
}

func main() {
    wordCount := make(map[string]int)
    // Here we loop through the slice of words produced using strings.Fields
    // function. That function splits text into a slice of strings using
    // space-characters: whitespaces, tabs, new line symbols, etc.
    for _, word := range strings.Fields(Shakespeare) {
        // Trim punctuation and make it lower case.
        word = getWord(word)
        // Increase the count. If the there was no such word in the map it uses
        // zero as its count.
        wordCount[word]++
    }

    // Loop through the map and print its keys and values.
    for word, count := range wordCount {
        fmt.Println(word, count)
    }
}
```

```
## thou 2
## feed'st 1
## with 1
## now 1
## or 1
## by 2
## tender 2
## waste 1
## desire 1
## spring 1
## herald 1
## contracted 1
## to 4
## eyes 1
## self-substantial 1
## a 1
## glutton 1
## as 1
## the 6
## but 2
## rose 1
## buriest 1
## world 1
## fairest 1
## thereby 1
## own 2
## where 1
## abundance 1
## self 2
## sweet 1
## cruel 1
## beauty's 1
## die 1
## and 3
## making 1
## foe 1
## niggarding 1
## thine 2
## content 1
## churl 1
## world's 2
## bud 1
## bear 1
## memory 1
## fuel 1
## within 1
## mak'st 1
## creatures 1
## riper 1
## art 1
## pity 1
## else 1
## this 1
## eat 1
## never 1
## flame 1
## bright 1
## light's 1
## famine 1
## too 1
## fresh 1
## ornament 1
## should 1
## time 1
## be 1
## heir 1
## gaudy 1
## thee 1
## from 1
## his 2
## decease 1
## lies 1
## only 1
## due 1
## we 1
## might 2
## thy 5
## in 1
## grave 1
## increase 1
## that 2
```

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 project

```go
package main

import (
    "fmt"
)

// printer is a struct with no fields. It only has a method.
type printer struct{}

// receipt is a variadic function. It can be called with any number of
// arguments, just like fmt.Println()
func (p printer) receipt(a ...interface{}) {
    fmt.Println(a...)
}
```

```go
// Account represents a bank account data structure, it has one field and an
// embedded struct.
type Account struct {
    balance int
    printer
}

// NewAccount created a new Account setting the initial balance.
func NewAccount(balance int) *Account {
    return &Account{
        balance: balance,
    }
}

// Deposit increases the account balance by the specified amount.
// It prints the information about the operation using the method of the
// embedded printer struct.
func (a *Account) Deposit(amount int) {
    a.receipt("--> trying to deposit", amount)
    a.balance = a.balance + amount
}

// Withdraw checks if the account balance is not lesser than the amount to
// withdraw and decreses the balance by the specified amount.
// It prints the information about the operation using the method of the
// embedded printer struct.
func (a *Account) Withdraw(amount int) {
    a.receipt("<-- trying to withdraw", amount)
    if amount > a.balance {
        a.receipt("Withdraw error: not enough funds to withdraw", amount)
        return
    }
    a.balance = a.balance - amount
}

// Balance outputs the account balance using the Method of the embedded printer
// struct.
func (a Account) Balance() {
    a.receipt("Account balance:", a.balance)
}

func main() {
    account := NewAccount(100)
    account.Balance()

    account.Withdraw(25)
    account.Balance()

    account.Deposit(50)
    account.Balance()

    account.Withdraw(1000)
    account.Balance()
}
```

```
## Account balance: 100
## <-- trying to withdraw 25
## Account balance: 75
## --> trying to deposit 50
## Account balance: 125
## <-- trying to withdraw 1000
## Withdraw error: not enough funds to withdraw 1000
## Account balance: 125
```

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 project

```go
package main

import (
    "bufio"
    "fmt"
    "io"
    "sort"
    "strings"
)
```

```go
// Shakespeare contains the text of one of the Shakespeare's sonets. We'll use
// this text to count words in it.
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.
`

// WordCountPair is a record of word and its count.
type WordCountPair struct {
    Word  string
    Count int
}

// getWord returns a word in lower case and with trimmed punctuation.
func getWord(s string) string {
    return strings.ToLower(strings.Trim(s, ",:."))
}

// Functions can be types too!
type lessFunc func(p1, p2 *WordCountPair) bool

// multiSorter implements the Sort interface, sorting the word-count pairs.
type multiSorter struct {
    wordCountPairs []WordCountPair
    less           []lessFunc
}

// Sort sorts the argument slice according to the less functions passed to
// OrderedBy.
func (ms *multiSorter) Sort(wordCountPairs []WordCountPair) {
    ms.wordCountPairs = wordCountPairs
    sort.Sort(ms)
}

// OrderedBy returns a Sorter that sorts using the less functions, in order.
// Call its Sort method to sort the data.
func OrderedBy(less ...lessFunc) *multiSorter {
    return &multiSorter{
        less: less,
    }
}
```

```go
// Len is part of sort.Interface.
func (ms *multiSorter) Len() int {
    return len(ms.wordCountPairs)
}

// Swap is part of sort.Interface.
func (ms *multiSorter) Swap(i, j int) {
    ms.wordCountPairs[i], ms.wordCountPairs[j] =
        ms.wordCountPairs[j], ms.wordCountPairs[i]
}

// Less is part of sort.Interface. It is implemented by looping along the less
// functions until it finds a comparison the discriminates between the two items
// (one is less than the other).
func (ms *multiSorter) Less(i, j int) bool {
    p, q := &ms.wordCountPairs[i], &ms.wordCountPairs[j]
    // Try all but the last comparison.
    var k int
    for k = 0; k < len(ms.less)-1; k++ {
        less := ms.less[k]
        switch {
        case less(p, q):
            // p < q, so we have a decision.
            return true
        case less(q, p):
            // p > q, so we have a decision.
            return false
        }
        // p == q; try the next comparison.
    }
    // All comparisons to here said "equal", so just return whatever the final
    // comparison reports.
    return ms.less[k](p, q)
}

// WordCount counts words read from input (io.Reader interface) and returns the
// word-count pairs.
func WordCount(input io.Reader) []WordCountPair {
    m := make(map[string]int)
    scanner := bufio.NewScanner(input)
    scanner.Split(bufio.ScanWords)
    for scanner.Scan() {
        // Read a word using word scanner, trim punctuation and make it lower case.
        word := getWord(scanner.Text())
        // Increase the count. If the there was no such word in the map it uses
        // zero as its count.
        m[word]++
    }

    // Create a slice the same length as the word-count map.
    pairs := make([]WordCountPair, len(m))

    // Fill the silce with data from the map.
    i := 0
    for word, count := range m {
        pairs[i] = WordCountPair{Word: word, Count: count}
        i++
    }

    return pairs
}

func main() {
    input := strings.NewReader(Shakespeare)
    pairs := WordCount(input)

    // Closures that order the WordCountPair structure.
    word := func(p1, p2 *WordCountPair) bool {
        return p1.Word < p2.Word
    }
    count := func(p1, p2 *WordCountPair) bool {
        return p1.Count > p2.Count // Note: > orders downward.
    }

    OrderedBy(count, word).Sort(pairs)

    for _, pair := range pairs {
        fmt.Println(pair.Word, pair.Count)
    }
}
```

```
## the 6
## thy 5
## to 4
## and 3
## but 2
## by 2
## his 2
## might 2
## own 2
## self 2
## tender 2
## that 2
## thine 2
## thou 2
## world's 2
## a 1
## abundance 1
## art 1
## as 1
## be 1
## bear 1
## beauty's 1
## bright 1
## bud 1
## buriest 1
## churl 1
## content 1
## contracted 1
## creatures 1
## cruel 1
## decease 1
## desire 1
## die 1
## due 1
## eat 1
## else 1
## eyes 1
## fairest 1
## famine 1
## feed'st 1
## flame 1
## foe 1
## fresh 1
## from 1
## fuel 1
## gaudy 1
## glutton 1
## grave 1
## heir 1
## herald 1
## in 1
## increase 1
## lies 1
## light's 1
## mak'st 1
## making 1
## memory 1
## never 1
## niggarding 1
## now 1
## only 1
## or 1
## ornament 1
## pity 1
## riper 1
## rose 1
## self-substantial 1
## should 1
## spring 1
## sweet 1
## thee 1
## thereby 1
## this 1
## time 1
## too 1
## waste 1
## we 1
## where 1
## with 1
## within 1
## world 1
```

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

```go
package main

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

```
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 #2 is on lap 2
## Racer #3 is on lap 1
## Racer #4 is on lap 1
## Racer #5 is on lap 1
## Racer #2 is on lap 1
## Race time: 1.000111085s
## Racer #10 is on lap 1
## Racer #1 is on lap 1
## Racer #2 finished!
## Racer #3 finished!
## Racer #4 is on lap 2
## Racer #5 is on lap 2
## Racer #7 is on lap 2
## Racer #9 is on lap 1
## Racer #2 is on lap 3
## Racer #3 is on lap 2
## Racer #3 is on lap 3
## Racer #7 is on lap 1
## Race time: 2.000316665s
## Racer #7 is on lap 3
## Racer #7 finished!
## Racer #6 is on lap 1
## Racer #8 finished!
## Racer #4 finished!
## Racer #8 is on lap 1
## Racer #8 is on lap 2
## Racer #8 is on lap 3
## Racer #4 is on lap 3
## Race time: 3.000266783s
## Racer #9 is on lap 2
## Racer #10 is on lap 2
## Racer #1 finished!
## Racer #5 finished!
## Racer #1 is on lap 2
## Racer #1 is on lap 3
## Racer #5 is on lap 3
## Racer #6 finished!
## Racer #6 is on lap 2
## Racer #6 is on lap 3
## Race time: 4.000161627s
## Racer #10 finished!
## Racer #10 is on lap 3
## Race time: 5.000189757s
## Racer #9 finished!
## Racer #9 is on lap 3
##
## Winners:
## 1 place: Racer #2
## 2 place: Racer #3
## 3 place: Racer #7
```

Course project

```go
package main

import (
    "encoding/csv"
    "fmt"
    "io"
    "log"
    "os"
    "strconv"
    "strings"
)

func main() {
    var input io.Reader

    switch len(os.Args) {
    case 1:
        input = os.Stdin
    case 2:
        f, err := os.Open(os.Args[1])
        if err != nil {
            log.Fatal(err)
        }
        defer f.Close()
        input = f
    default:
        log.Fatal("This program expects either 0 or 1 arguments.")
    }

    r := csv.NewReader(input)
    r.FieldsPerRecord = -1

    records, err := r.ReadAll()
    if err != nil {
        log.Fatal(err)
    }

    // Remove the very first "record" (i.e 'Category: All categories') if exists
    if len(records[0]) == 1 {
        records = append(records[:0], records[1:]...)
    }
    // Save names to a slice
    names := records[0][1:] // Skip 'weeks' column
    commonSuffix := longestCommonSuffix(names)
    if commonSuffix != "" {
        for i, name := range names {
            names[i] = strings.TrimSuffix(name, commonSuffix)
        }
    }
    records = append(records[:0], records[1:]...)

    avg := make([]int, len(names))

    for _, record := range records {
        record = record[1:] // Skip 'weeks' column
        for i, s := range record {
            n, err := strconv.Atoi(s)
            if err != nil {
                log.Fatal(err)
            }
            avg[i] += n
        }
    }

    for i := 0; i < len(avg); i++ {
        avg[i] = avg[i] / len(records)
    }

    for i, n := range avg {
        n = n/10 + 1
        fmt.Printf("%s %s (%d)\n", strings.Repeat(" ", n), names[i], avg[i])
    }
}

func longestCommonSuffix(a []string) string {
    if len(a) == 0 {
        return ""
    }

    suffix := a[0]
    if len(a) == 1 {
        return suffix
    }

    for _, s := range a[1:] {
        suffixLength := len(suffix)
        sLength := len(s)

        if suffixLength == 0 || sLength == 0 {
            return ""
        }

        maxLength := suffixLength
        if sLength < maxLength {
            maxLength = sLength
        }

        for i := 0; i < maxLength; i++ {
            j := suffixLength - i - 1
            k := sLength - i - 1
            if suffix[j] != s[k] {
                suffix = suffix[j+1:]
                break
            }
        }
    }
    return suffix
}
```