Go, from the beginning

from o to hero

Chris Noring

Go, from the beginning

from o to hero

Chris Noring

This book is for sale at http://leanpub.com/go-from-the-beginning

This version was published on 2022-05-06

To the brave people of Ukraine for which I'm writing this book

Contents

Your first program

This lesson covers some history of Go and also teaches you how to build your first Go app.

Introduction

In this lesson we'll cover:

- The history of Go
- Why use Go for your apps
- The anatomy of a Go app
- Authoring and running your first app

A history of Go

The language is called Go but is sometimes known as Golang as the first website for it was golang.org.

Go was created in 2009 by Robert Griesemer, Rob Pike and Ken Thompson. It's hard to estimate the number of Go developers but it's somewhere between 1.1 and 2.7 million, quite a sizeable amount. More than 2500 companies are using Go including, Google, Pinterest and Uber. So, you see, used by a lot of folks by big companies.

> Why was Go created?

As is often the case, a programming language is created to deal with the shortcomings of other languages. In this case, the creators wanted this new language to have the following capabilities:

- **Static typing** and run-time efficiency from C.
- **Readability** from JavaScript and Python.
- **High-performance** networking and multi-processing.

It seems the creators agreed on disliking C++ :)

What is it used for though?

Here's some areas where you are likely to find a Go being used:

- Cloud based and server-side apps.
- DevOps, automation.
- Command-line tools.
- AI and data science.

References

There are many great resources out there for learning the Go programming language like:

- https://go.dev/
- https://www.tutorialspoint.com/go/index.htm
- https://gobyexample.com/
- https://www.w3schools.com/go/
- https://docs.microsoft.com/en-us/learn/modules/go-get-started/
- https://docs.microsoft.com/en-us/learn/modules/serverless-go/

Features

So, what features makes Go compelling? Well, there are some features worth mentioning:

- **Static typing**, I like my types :)
- **Package system**. You can consume and create your own packages. Go to pkg.go.dev[1] to read more on what packages there are.
- **Command-line tools**, there's a set of executables that are installed when you install Go. With these executables, you can run, build, install packages, run tests and much more.
- **Standard library**. Go has a powerful standard library that will help you with most things you might need. You can read more about what's in the standard library[2] here.
- **Built-in testing**. Having a testing library that just works out of the box is something you shouldn't take for granted.
- Concurrency. Go is great at handling concurrency. It uses concepts like goroutines and channels.
- **Garbage collection**. You can read more about that here[3]. I like when I don't have to deal with that myself and just focusing on solving problems.

[1] https://pkg.go.dev/
[2] https://pkg.go.dev/std
[3] https://medium.com/safetycultureengineering/an-overview-of-memory-management-in-go-9a72ec7c76a8#:~:text=Go%20has%20all%20goroutines%20reach,the%20collector%20to%20run%20simultaneously

Install Go

Ok then, hope you are intrigued at this point and just want to see some code? Of course, you are :)

Make sure you've followed the instructions for installing Go on your machine.

> https://go.dev/doc/install

A Go program

Here's what a first program can look like:

```
1  package main
2
3  import "fmt"
4
5  func main() {
6   fmt.Println("hello")
7  }
```

The program in detail

- `package main`, the entry point module needs to have this instruction.
- `import "fmt"`, fmt is standard package for input and output.
- `func main`, entry point function, where your program starts.

Commands

Now that you have a program, there's two things you might want to do:

- **Run it**, to see if it compiles and runs.
- **Create executable**, an executable is no longer Go code but like any executable program on your machine.

Run your app

To run your app, type `go run <file>.go`, for example:

```
go run main.go
```

Build your app

To produce an executable, run `go build <file>.go`, for example:

```
go build main.go
```

It produces an executable, on MacOS and Linux that's a file with -X as permission, on Windows, it's a .exe file.

Congrats, you've created your first Go application.

Summary

In this article, you learned about the programming language Go, some features it has and how to write your first program.

Challenge

Compare Go to other programming languages, can you list some differences between them?

Review & Self Study

Select one of the resources below and try do a tutorial.

- https://go.dev/
- https://www.tutorialspoint.com/go/index.htm
- https://gobyexample.com/
- https://www.w3schools.com/go/
- https://docs.microsoft.com/en-us/learn/modules/go-get-started/
- https://docs.microsoft.com/en-us/learn/modules/serverless-go/

Assignment

Create a file *main.go*. Use the `fmt` library to print out to the console. Remember that your run programs with `go run <my program>.go`.

Solution

Create a file *main.go*

```
1  package main
2
3  import "fmt"
4
5  func main() {
6    fmt.Println("printing to the console")
7  }
```

Using variables

With variables, we can remember values and later refer to them via named references. using variables will make our code easier to read.

Introduction

In this lesson we'll cover:

- The usage of variables in Go.
- How to create them.
- Assign different types and values.

Declare variables

In Go, there are many ways to declare variables:

- **Define a name and type**. Here, you declare a variable with the keyword `var`, give it a name and lastly a type `string`. Below is an example:

```
  var firstName string
```

- **Define a group** of variables. It's possible to define a group of variables. Using this way of declaring means you only type the `var` keyword once. The group is defined using parenthesis `()`:

```
var (
  firstName = "Chris"
  age = 20
)
```

  Note how each variable is on a new row.
- **Define and assign a value**. Within functions, you can use the `:=` operator, it declares and assigns at the same time. The below code shows the creation of the `firstName` variable. The data type is inferred to be a string:

```
firstName := "Chris"
```

Assign variables

To assign a new value to a variable, it needs to exist first. You use the assignment operator, =. Here's an example:

```
firstName = "Mike"
```

Data types

There are many data types you can use with Go. They are divided into different categories:

- **Basic types**. In this category, we find types like integers, floats (numbers with decimals) and other types like Booleans (for true/false), strings (for text) and more.
- **Composite types**. We will talk about composite types in a separate article, but they are more complex, and examples of composite types are arrays, structs and interfaces.

Declare a variable with a type

There are two ways you can declare a variable and give it a type:

- **explicitly**, by specifying its type, for example:

```
var name string
```

- **implicitly**, by assigning it a value and having it been inferred:

```
name := "chris"
```

  In the preceding code, the data type is inferred by the value you give it. In this case, the data type becomes `string` based on the value "chris".

String interpolation

Sometimes, you want to be able to write things to the screen and mix different data types doing so. For example, you might want to write, "Customer: Adam has 20$ in his bank account".

Let's say then that this information is represented by these two variables:

```
var (
  customerName = "Adam"
  accountBalance = 20
)
```

How can you print out the text above? For this purpose, you can use the `Printf()` function that takes formatters. The idea is that a formatter is an instruction to what a certain type is. By providing this information to `Printf()`, it's able to print the type correctly.

Here's how you can print the example string from before:

```
fmt.Printf("Customer %s has %d$ on their bank account", customerName, accountBalance)
```

Above, the `%s` represents a string and `%d` represents a number. By using these formatters as placeholders, the variables are correctly implemented, and the output becomes:

```
Customer Adam has 20$ on their bank account
```

Assignment - define some variables and print them out

Define some variables you might need for the card game Texas Holdem and print them out.

Create a file *main.go* and give it the following content:

```
package main

import "fmt"

func main () {
}
```

1. Add the following variables after the import section:

```
var (
  players = 3
  replay = false
  namePlayerOne = "chris"
)
```

Now you have:

- `players`, to represent the number of players in the game.

- `replay`, a boolean stating whether to start a new game session when the old one has ended.
- `namePlayerOne`, a string representing the name of the first player.

All of these variables help describe essential information in a Texas Holdem game.

Next, let's run our app to make sure it works.

2. Add the following code to the `main()` function to print out the variables:

```
fmt.Println(players)
fmt.Println(replay)
fmt.Println(namePlayerOne)
```

3. Run `go run main.go` in the terminal:

```
go run main.go
```

You should see the following output:

```
 3
 false
 chris
```

Great, you now have a starting point for an app you can keep building on.

Challenge

See if you can come up with more variables to represent the state in a Texas Holdem card game, like for example, other players, the card deck etc. What data type would you give those variables?

Review & Self Study

Have a look at this official tutorial on variables[4] using a Go sandbox

Solution

[4] https://go.dev/tour/basics/8

```
package main

import "fmt"

var (
  players = 3
  replay = false
  namePlayerOne = "chris"
)

func main () {
  fmt.Println(players)
  fmt.Println(replay)
  fmt.Println(namePlayerOne)
}
```

Flow control

In this chapter, we're looking to learn about constructs `if` and `else` to control the flow of your application.

Introduction

This chapter will cover:

- Working with Boolean logic.
- Create Boolean data.
- Use constructs like `if`, `else if` and `else`.

Flow control

Using Boolean logic in your program is about creating different execution paths through your code?

> What does that mean?

It means there's more than one way that your program can run depending on what data you feed it.

> Ok, can you show me?

Sure, consider this code:

```
printMessage := true

if printMessage {
  fmt.Println("Message")
}
```

If `printMessage` is `true`, the string "Message" will print. If the value is `false`, nothing will print.

> Ok, I think I get it.

The `if` construct

You've seen an example already about code that runs or doesn't run depending on a value. The `if` construct is what makes that possible. An `if` take a Boolean expression like so:

```
if true {
  // statements here will always run
}
```

Using a Boolean variable

When you use a Boolean value as part of your Boolean expression, it needs to be evaluated. Here's code showing just that:

```
accountBalance = 100
accountCredit = 200
if accountBalance + accountCredit > 0 {
  fmt.Println("You have money to spend")
}
```

The program above does the job, meaning it correctly evaluates whether you have money to spend. However, you might want to print something out if the condition is not met, for that you have `else`.

Introducing `else`

You would like to improve the preceding code. The `else` clause is run when `if` is evaluated to false. Here's how you can add it to the program:

```
accountBalance = 100
accountCredit = 200
if accountBalance + accountCredit > 0 {
  fmt.Println("You have money to spend")
} else {
  fmt.Println("No money left, please add more funds")
}
```

Using `else if`

`if` and `else` take you far. Sometimes, it's not enough. You might need to grade a course at different levels depending on the points achieved on the exam. For this situation, you need an `else if` construct, a construct that will be evaluated if the `if` construct evaluates to false. It differs from `else` in that it also takes an expression. Here's an example where it's used:

```
if testScore >= testScoreGrade5 {
  fmt.Println("Top mark")
 } else if testScore >= testScoreGrade4 {
  fmt.Println("Pass with distinction")
 } else if testScore >= testScoreGrade3 {
  fmt.Println("Pass with distinction")
 } else {
  fmt.Println("Failed")
 }
```

Multiple expressions

Your expression can examine more than one variable or condition. There are Boolean operators you can use to help you. Here are some operators you are likely to encounter:

- `&&`, evaluates to true if values on the left and right side are both true. Here's an example of this operator in use:

```
  hasGas := true
  hasKeyInIgnition := true
  if hasGas && hasKeyInIgnition {
    fmt.Println("Can drive car")
   }
```

  In the preceding code, the expression will evaluate to true as both `hasGas` and `hasKeyInIgnition` is true.
- `||` , evaluates to true if either left or right value is true. Here's an example of this operator in use:

```
hasBurger := true
hasSandwich := false

if hasBurger || hasSandwich {
  fmt.Println("Can eat")
}
```

  In the preceding code, `hasBurger` is true and that's enough for this expression to become true.
- `!`, also known as NOT, it will negate the expression. Here's an example:

```
hasSandwich := false

if !hasSandwich {
 mt.Println("No sandwiches, then I will starve, I only eat sandwiches")
}
```

Above, the expression will evaluate to true, thanks to the negation with `!`.

Assignment - create a program that tests your Boolean logic

In this assignment, you are creating a program that tests out various Boolean logic.

1. Create a file *main.go* and give it the following content:

```
package main

import "fmt"

func main() {
 testScoreGrade5 := 80
 testScoreGrade4 := 60
 testScoreGrade3 := 50
 testScore := 49

 hasGas := true
 hasKeyInIgnition := true

 hasBurger := true
 hasSandwich := false

 printMessage := true
 if printMessage {
  fmt.Println("Message")
 }

 if testScore >= testScoreGrade5 {
  fmt.Println("Top mark")
 } else if testScore >= testScoreGrade4 {
  fmt.Println("Pass with distinction")
 } else if testScore >= testScoreGrade3 {
  fmt.Println("Pass with distinction")
```

```
28    } else {
29     fmt.Println("Failed")
30    }
31
32    if hasGas && hasKeyInIgnition {
33     fmt.Println("Can drive car")
34    }
35
36    if hasBurger || hasSandwich {
37     fmt.Println("Can eat")
38    }
39
40    if !hasSandwich {
41     fmt.Println("No sandwiches, then I will starve, I only eat sandwiches")
42    }
43   }
```

2. Run the command `go run main.go`, to run the program

```
goo run main.go
```

   You should see the following output:

```
 Message
 Failed
 Can drive car
 Can eat
 No sandwiches, then I will starve, I only eat sandwiches
```

3. Try playing around with the code, how does the output change if you change `testScore` value to 51, 62, 3 or 90?

Challenge

A test score shouldn't be negative, how can you add a check for that?

Solution

```go
package main

import "fmt"

func main() {
	testScoreGrade5 := 80
	testScoreGrade4 := 60
	testScoreGrade3 := 50
	testScore := 49

	hasGas := true
	hasKeyInIgnition := true

	hasBurger := true
	hasSandwich := false

	printMessage := true
	if printMessage {
		fmt.Println("Message")
	}

	if testScore >= testScoreGrade5 {
		fmt.Println("Top mark")
	} else if testScore >= testScoreGrade4 {
		fmt.Println("Pass with distinction")
	} else if testScore >= testScoreGrade3 {
		fmt.Println("Pass with distinction")
	} else {
		fmt.Println("Failed")
	}

	if hasGas && hasKeyInIgnition {
		fmt.Println("Can drive car")
	}

	if hasBurger || hasSandwich {
		fmt.Println("Can eat")
	}

	if !hasSandwich {
		fmt.Println("No sandwiches, then I will starve, I only eat sandwiches")
	}
}
```

Review & Self Study

Have a look at this official tutorial on flow control[5] using a Go sandbox.

[5] https://go.dev/tour/flowcontrol/6

Converting between types

This chapter covers how to convert between strings and numbers.

Introduction

This chapter will:

- Introduce uses cases where data conversion makes sense.
- Showcase how to use `strconv` library.

Why convert between types

There are different data types and a need to convert between them. For example, we often need to convert between text and numbers for presentational and other reasons. We also need to convert between numbers and decimals without losing information in the process.

The main package for dealing with conversions in Go is `strconv`.

Use case - command-line arguments

Let's show a common case where you start off with strings and you need to make it into numbers, command-line arguments. To use command-line arguments in a program, you need the `os` package.

`os.Args` points to an array representing your command line arguments. To access a specific argument, you would use the index operator `[]` like so:

```
arg := os.Args[1]
```

You can then start your program like so:

```
go run main.go 1
```

The 1 would then be stored in `arg`.

Finding the type

What type is `arg` in our code above? There are some ways to find out:

- **IDE**, if you use for example Visual Studio Code and the Go plugin, hovering over the code, it will tell you that `os.Args` is a string array, `string[]`.
- **PrintF() and %T**. One of the easiest ways to find the type is typing like so:

```
Printf("%T", os.Args[1])
Printf("%T", 1)
```

You would get an output like so:

```
string
int
```

- **Type coercion**, you could try to modify that code and coerce it to be an integer like so, now what?

```
var no int = os.Args[1]
```

You get an error:

```
cannot use os.Args[1] (type string) as type int in assignment
```

- **Use reflection**. Another way to find the above is by using the `reflect` package like so:

```
package main

import (
  "reflect"
  "fmt"
  "os"
)

func main () {
  arg := os.Args[1]
  fmt.Println(reflect.TypeOf(arg))
}
```

Now, the program will print "string" as the type.

Addressing the problem with `strconv`

Ok, so we know what type something is, what if we need to use these command-line arguments, which are of type `string`, and feed them into let's say a calculator program?

Consider the below code, that at present WOULDN'T compile:

```
package main

import (
  "fmt"
  "os"
)

func add(first int second) int {
  return first + second
}

func main() {
  add(os.Args[1], os.Args[2]) // this would NOT compile
}
```

The reason is that the values on `os.Args[1]` and `os.Args[2]` are `string` not `int`. To fix this issue, we need to use the conversion package `strconv`.

Convert from string to int with `strconv`

To convert strings to integers, we need to use `strconv` and call the `Atoi()` (stands for Ascii to integer) function like so:

```
package main

import (
 "fmt"
 "os"
 "strconv"
)

func add(first int, second int) int {
 return first + second
}

func main() {
 no1, _ := strconv.Atoi(os.Args[1])
 no2, _ := strconv.Atoi(os.Args[2])
 var sum = add(no1, no2)
 fmt.Println(sum)
}
```

Note _, this is a *don't care* symbol. What happens when you call `Atoi()` is that it returns two things, the number and an error if it fails.

Handling conversion error

To handle an error, we need to store it in a variable, `err` and inspect it. If it's not `nil`, then we have an error.

Here's how we could encode that behaviour below:

```
1  package main
2
3  import (
4   "fmt"
5   "os"
6   "strconv"
7  )
8
9  func main() {
10  no, err := strconv.Atoi(os.Args[1])
11  fmt.Println(no)
12  if err != nil {
13   fmt.Println(err)
14   fmt.Println("Couldn't convert: " + os.Args[1])
15
16  } else {
17   fmt.Println(no)
18  }
19 }
```

Try to compile the above program and run it like so:

```
1 main 1 # 1
2 main hi # trconv.Atoi: parsing "hi": invalid syntax, Couldn't convert: hi
```

Parse string to int

There's another way to convert a string to an int. That's by using the `ParseInt()` method. It does more than converting though, it does two things in fact:

- **base**, you can select according to what base to interpret the number as.
- **size**, bit size, from 0 to 64.

The syntax for the method looks like so:

```
ParseInt(<s string>, <base int>, <bit int>) (int64, error)
```

Here's some examples:

```
package main

import (
 "fmt"
 "reflect"
 "strconv"
)

func main() {
 var no int = 100
 fmt.Println(reflect.TypeOf(no))

 var intStr string = "100"
 fourBaseEightBitInt, _ := strconv.ParseInt(intStr, 4, 8)    // becomes no 16 and in\
t64
 tenBaseSixteenBitInt, _ := strconv.ParseInt(intStr, 10, 16) // no 100,  and int64
 fmt.Println(reflect.TypeOf(fourBaseEightBitInt))
 fmt.Println(reflect.TypeOf(tenBaseSixteenBitInt))
}
```

Integer to string

You might be dealing with the opposite; you have an integer, and you want it to be a string. In this case, you can use the `Itoa()` function, integer to ascii. Here's an example:

```
var noOfPlayers = 8
str, _ := strconv.Itoa(noOfPlayers)
```

Additional parsing

The `strconv` library is what you want if you start with a string, and you want to convert to and from another format. Learn more about strconv library here[6]

Assignment

Create an app that adds two numbers together. The values should come from the command line. Here's how the program should run:

[6] https://pkg.go.dev/strconv

```
go run main.go 2 4
6
```

Solution

```
package main

import (
 "fmt"
 "os"
 "strconv"
)

func add(no int, secondNumber int) int {
 return no + secondNumber
}

func main() {
 no1, _ := strconv.Atoi(os.Args[1])
 no2, _ := strconv.Atoi(os.Args[2])
 var sum = add(no1, no2)
 fmt.Println(sum)
}
```

Challenge

What happens if run the program like so?

```
go run main.go one two
```

Handle any conversion error and call `panic()` if there's a conversion error.

Working with loops

This chapter covers working with loops in Go. Loops are used to repeat statements in your code.

Introduction

This chapter will cover:

- Loop statements with `for`.
- Device conditions on when to break a loop.
- Apply `range` to iterate over an array.
- Control the loop with `continue` and `break`.

The case for looping statements

You are likely to want to repeat a set of instructions. For example, you might have a list of orders where you need to process each order. Or you have a file that you need to read line by line or there might be some other calculation. Regardless of your situation, you are likely to need a looping construct, so what are your options in Go?

You are using the `for` loop. There are three major ways you can use it:

- **increment steps**. With the help of a variable, you define a start point, a condition when to stop and an incrementation step. This is your "classic" for-loop. Here's what it looks like:

```
1  for i := 0; i < 10; i++ {
2    // run these statements
3  }
```

- **while**. In many programming languages you have a `while` keyword. Go doesn't have that, but what you can do is use the for-loop similarly. You omit the initialization step and increment step and get this code:

```
1  for <condition> {
2    // run these statements
3  }
```

- **for each**. lastly, you have the `for-each` like construct that operates on an array-like sequence. It uses the `range` keyword to function:

```
for i,s := range array {
  // run these statements
}
```

The `for` loop

The conventional for-loop has three different parts:

- **initialization**, here you want to create a variable and assign it a starter value like so:

```
for i:= 0;
```

  Note the use of `;`, you usually don's use semicolons, but for this construct, you need it.
- **condition**. The next step is evaluating whether you should continue incrementing or not. You define a Boolean expression here, that if `true`, continues to loop:

```
for i := 0; i< 10
```

  `i < 10`, as long as a value is between 0 and 10 (becomes 10, then loop breaks), then it returns true, and the loop continues.
- **increment**, in this step, the loop variable `i` is updated, updating it by 1 is common but you can add any increment size, negative or positive.

```
for i := 0; i< 10; i++ {

}
```

  Here, `i` is updated by 1. This loop will run ten times.

Repeat until the condition is met with `while`

A simplified version of this loop can omit the initialization and increment steps. You are then left with the condition step only. This step tests whether a variable is `true` or `false` and the loop exits on false. Here's an example:

```
i := 1
for i < 10 {
  i++
  // do something
}
```

In this case, we are declaring `i` outside of the loop. Within the loop, we need to change the value to something that will make the loop expression evaluate to false or it will loop forever.

Here's another code, using the same idea, but this time we ask for input from the user:

```
var keepGoing = true
answer := ""
for keepGoing {
  fmt.Println("Type command: ")
  fmt.Scan(&answer)
  if answer == "quit" {
    keepGoing = false
  }
}
fmt.Println("program exit")
```

An example run of the program could look like so:

```
Type command: test
Type command: something
Type command: quit
program exit
```

Using for-each over a range

For this next loop construct, the idea is to operate over an array or known sequence. For each iteration you can get the index as well as the next item in the loop. Here's some example code:

```
arr := []string{"arg1", "arg2", "arg3"}
 for i, s := range arr {
  fmt.Printf("index: %d, item: %s \n", i, s)
 }
```

`arr` is defined as an array and then the `range` construct is used to loop over the array. For each iteration, the current index is assigned to `i` and the array item is assigned to `s`. An output of the above code will look like so:

```
index: 0, item: arg1
index: 1, item: arg2
index: 2, item: arg3
```

Controlling the loop with `continue` and `break`

So far, you've seen three ways you can use the `for` construct. There are also ways to control the loop. You can control the loop with the following keywords:

- `break`, this will exit the loop construct

```
arr = []int{-1,2}
for i := 0; i< 2; i++ {
  fmt.Println(arr[i])
  if arr[i] < 0 {
    break;
  }
}
```

The output will be:

```
-1
```

it won't output `2` as the loop exits after the first iteration.

- `continue`, this will move on to the next iteration. If `break` exits the loop, `continue` skips the current iteration:

```
arr = []int{-1,2,-1, 3}
for i := 0; i< 4; i++ {
  if arr[i] < 0 {
    break;
  }
  fmt.Println(arr[i])
}
```

The output will be:

```
2
3
```

Assignment - create a command line loop

When creating console apps, you often want to read user input. The user input could be data used in the program or it can be the user typing a command to do something like "save", "print", "backup" etc. We will build a program for the latter case.

1. Create a file *main.go* and give it the following content:

```
package main

import "fmt"

func main() {

}
```

2. Add the following code to the `main()` method:

```
var keepGoing = true
answer := ""
for keepGoing {
  fmt.Println("Type command: ")
  fmt.Scan(&answer)
  if answer == "quit" {
    keepGoing = false
  }
}
fmt.Println("program exit")
```

3. Run the code by typing `go run main.go`:

```
go run main.go
```

You should see an output like so:

```
Type command: command
Type command: quit
program exit
```

Challenge

- Add a command “print” that ends up outputting “printing file”.
- See if you can use a `break` instead of the variable `keepGoing` to break the loop when the user types”quit”.

Solution

```
1  package main
2
3  import "fmt"
4
5  func main() {
6    var keepGoing = true
7     answer := ""
8     for keepGoing {
9       fmt.Println("Type command: ")
10      fmt.Scan(&answer)
11      if answer == "quit" {
12        keepGoing = false
13      }
14    }
15    fmt.Println("program exit")
16 }
```

Reading user input

You will learn how to read user input, both a simpler technique and a more advanced one using formatters.

Introduction

This chapter will cover:

- The `Scan()` method to read user input.
- Reading one input.
- Reading multiple inputs.
- Working with formattters.

User input

It's an important thing to be able to read user input from the console. It gives the user a chance to interact with the program. Things to consider are how you are asking the user to input, is it one word, several inputs. Will the user separate input by space or newline? Regardless of what approach you go with, try to communicate the chosen approach to the user.

Managing user input with `fmt`

So far, you've seen how the `fmt` package can be used to print to the console. It can also be used to read user input.

The `fmt` library has a built-in `Scan()` method that we will use to capture the user input.

Reading one input

You might start out wanting to read one input from the user. That's what the `Scan()` method does by default.

Here's some code showing how to collect user input:

```
package main

import "fmt"

func main() {
    var response string
    fmt.Scan(&response)
    fmt.Println("User typed: ", response)
}
```

Note how you send in the string `response` as a reference, using the `&` operator. It's done this way as the `Scan()` method will modify the variable you send in.

When you run this code, you will see the below output:

```
hello
User typed: hello
```

Reading multiple inputs

You can provide several arguments to the `Scan()` method. By providing several arguments, you can collect more than one user input and separate each input, in the `Scan()` function, with a comma. Here's how to apply this technique:

```
var a1, a2 string
// multiple input
fmt.Scan(&a1, &a2)

// formatted string to print out the user input
str := fmt.Sprintf("a1: %s a2: %s", a1, a2)
```

Note how `a1` and `a2` is sent into `Scan()` and they are comma-separated. So how will those codes run?

When you run this code, there are two ways for the user to input. The user can either separate the values by space like so:

```
hi you
a1: hi a2: you
```

or by newline:

```
hi
you
a1: hi a2: you
```

`Scanf()`, scan the input with formatters

So far, you've seen how you can collect input with spaces and endlines as separators. You can also be specific in how you collect input. Imagine that the user wants to type in an invoice number like so "INV200" or "INV114" and what you are interested in is the number part, or you are interested in the prefix as well?

The answer to solving this is in the `Scanf()` function. It takes formatters. With formatters, you're able to pick the part of the user input you are interested in and place that into the variable you want.

In the above-mentioned case, you can construct code looking like so:

```
var prefix string
var no int
// in110
fmt.Scanf("%3s%d", &prefix, &no)
fmt.Printf("prefix: %s, invoice no: %d", prefix, no)
```

The interesting part lies in the first argument of `Scanf()`, namely `3s%d`. What this means is, take the first 3 characters, `%3s` and interpret them as a string. Then interpret and place the remaining as a number, `%d`.

Running this program, you will get an output like so:

```
inv200
prefix: inv, invoice no: 200
```

"inv" is placed in `prefix` and 200 in `no` variable.

Learn more

To learn more about this area, check out this link https://pkg.go.dev/fmt#Scanf

Assignment - read user input

In this assignment, you will read user input for a card game. The objective is to capture the names of the players.

1. Create a file *main.go* with the following content:

```
package main

import "fmt"

func main() {

}
```

2. Add the following code to the `main()` method:

```
fmt.Println("Enter player names (separated by space or newline):")
var player1, player2 string
fmt.Scan(&player1, &player2)
fmt.Println("Players are: ", player1, player2)
```

3. Run the program using `go run main.go`:

```
go run main.go
```

You should see the following output:

```
Enter player names (separated by space or newline):
Alice Bob
Players are:  Alice Bob
```

Challenge

Try modifying the program so it takes several players first and then ensures all players get a name. Here's an example output of such a program:

```
Enter number of players:
3
Enter Player 1 name:
Alice
Enter Player 2 name:
Bob
Enter Player 3 name:
Jean
Players are: Alice Bob Jean
```

Solution

```go
package main

import "fmt"

func main() {
 fmt.Println("Enter player names (separated by space):")
 var player1, player2 string
 fmt.Scan(&player1, &player2)
 fmt.Println("Players are: ", player1, player2)
}
```

Using functions

In this chapter, we will discuss how you can define and use functions. Functions are great when you have the same type of code used in many places. By using functions, you thereby reduce repetition.

Introduction

This chapter will cover:

- What a function is and why use it.
- How to define and call a function.
- Returning multiple values.

Why functions

As soon as you have a set of statements you repeat in many places, it's a good use case for creating a function. Typical things you put in functions are logging to a file, performing a calculation or talking to a data source.

Your first function `main()`

So far, you've seen the function `main()`, define like so:

```
1  func main(){
2  }
```

There's only one such function, it's called an entry point and represents the start of the program. You can however define other functions.

The anatomy of a function

A function consists of various parts. By incorporating all these parts, you ensure you have a reusable piece of code you can use in many places.

Here are the parts you need to care about:

- `func`, the keyword `func`.
- **parameters**, 0 to many parameters
- **a function body**, i.e. statements that say what the function does.
- **a return construct**, if the function returns something.

Here's an example:

```
func add(first int, second int) int {
  return first + second
}
```

In the preceding code, the function is named `add()`. It has the parameters `first` and `second`. The function body, what the function does, consists of this code:

```
return first + second
```

Adding a return type

To add a return type, we add that after the function parenthesis in form of a type. Here's an example:

```
add(firstNumber int, secondNumber int) int {
  ...
}
```

Because we've added a return type of `int`, our function must return something. A way to return a value is by using the keyword `return`, like so:

```
add(firstNumber int, secondNumber int) int {
  return firstNumber + secondNumber
}
```

Named return

We can also name the return parameter like so:

```
add(firstNumber int, secondNumber int) (sum int) {
  sum = firstNumber + secondNumber
  return
}
```

- Note how `sum` is part of function prototype declaration `(sum int)` and then assigned a value `sum = firstNumber + secondNumber`.
- There's also a `return` on its own row, this code will compile as there's a notion of a return variable.

Multiple returns

It's possible to return more than one value.

Just like you returned a named parameter via `(sum int)`, you can comma separate like so `(sum int, product int)`. When returning multiple values, you can type like so:

```
sum = first + second
product = first * second
return
```

Both `sum` and `product` are assigned values and you have a closing `return`.

Putting it altogether you get a function that looks like so:

```
func calc(first int, second int) (sum int, product int) {
  sum = first + second
  product = first * second
  return
}
```

To call the function, you type like so:

```
sum, product := calc(1, 2)
fmt.Println(sum)
fmt.Println(product)
```

Note how you assign the two returned values to variables `sum` and `product`.

Assignment - adding a function to a program

1. Create a file *main.go* and give it the following content:

   ```
   package main

   import "fmt"

   func main() {

   }
   ```

2. Add a function `log()`, that we can use to print messages.
   Added to the program, your code should now look like so:

```
package main

import "fmt"

func log() {
  fmt.Println("message")
}

func main() {
  log()
}
```

At this point, the `log()` function isn't very flexible, it prints "message" every time it's invoked.

To make the `log()` function more flexible, lets add a parameter.

Adding a parameter

A parameter needs a data type, in this case, we will make it of type `string`.

1. Add the parameter within the parenthesis `()`, like so:

```
func log(message string) {
  fmt.Println(message)
}

// to use
log("hi")
log("there")
```

Note how the `log()` function takes the parameter `message` that is of type string. Our code is more flexible.

Solution

```
package main

import "fmt"

func log(message string) {
  fmt.Println(message)
}

func main() {
  log("hi")
  log("there")
}
```

Handling errors

In this chapter, we will cover error handling.

Introduction

This chapter will cover:

- Causing and handling errors with `panic` and `recover`.
- Use the error pattern to use a more idiomatic approach to managing errors.

Errors

Our apps aren't perfect, they will throw errors. Either they throw errors because of code we wrote or because of code written in a package we are using. Sometimes, that's even what the code is supposed to do when we feed it bad input.

Regardless, our users expect us to handle these errors. Also, for our own sake, we need to log these errors in such a way that we can fix those errors if they are unexpected.

There are two major ways to handle errors in Go:

- **panic/recover**. `panic` and `recover` are language constructs. If you know another programming language, they function like throw and catch. What does it mean though? It means there's an error, with an error message and stack trace that we can catch and recover from or let it crash the program.
- **error pattern**. This is referred to as the idiomatic Go way of doing things. The idea is to return a value and an error object from a function call. If an error occurred, it contains an error object or `nil`, if no error.

Crash the program with `panic()`

Let's take this function `Divide()`:

```
func Divide(nominator int, divider int) float32 {
  if divider == 0 {
    panic("can't divide by 0")
  }
  return float32(nominator) / float32(divider)
}
```

It has an `if` check. If `Divider()` is `0` then it calls `panic()`. So what happens then? You see something like:

```
panic: can't divide by 0

goroutine 1 [running]:
main.Divide(...)
        /<path>/panic.go:20
main.main()
        /<path>/panic.go:33 +0x96
exit status 2
```

At the top is the error string you sent to `panic()`, the string "can't divide by 0". Then you have the stack trace, entries that indicate where the error started. Read it from the bottom, the error started on line 33, then you have line 20, which is this row:

```
panic("can't divide by 0")
```

Ok, so we have a way to protect ourselves from input values that we don't want. But crashing the program, is that necessary? In some cases, it is, in some cases, it isn't. For the latter cases, we have `recover()`.

Capture the error with `recover()`

Using `recover()` is about capturing an error so our program can continue. We need to learn about a concept before proceeding. That concept is `defer`. `defer` is a language construct that delays the execution of a function until the nearby function returns. Here's an example:

```
defer fmt.Println("second")
fmt.Println("first")
```

Running this program, you should see:

```
first
second
```

> See `defer` as the last thing that happens.

How is this useful in the context of capturing an error? Capturing errors, if you want to capture it, is something you want to do as the last thing that happens. Take our `Divide()` function again:

```
func Divide(nominator int, divider int) float32 {
  defer errorHandler()
  if divider == 0 {
    panic("can't divide by 0")
  }
  return float32(nominator) / float32(divider)
}
```

Note how it now has a line in it that says `defer errorHandler()`. It will be run the last thing that happens. Depending on the value of `divider`, it will either call `panic()` or call the `return` statement as the last thing.

Ok, so what does `errorHandler()` look like?

```
func errorHandler() {
  if r := recover(); r != nil {
    fmt.Println("Recovered ", r)
  }
}
```

In `errorHandler()`, we invoke `recover()` and assign it to variable `r` and then we test it for `nil`. If it's NOT `nil` then we have an error and we print it out. If it is `nil` then the user notices nothing.

Improve the error handling

There are steps we can take to improve the error handling. So far, we're printing back an error message. Imagine if someone read this error from a log file, would they be able to understand where things went wrong and fix the input data or the code itself?

There are a couple of things we can do:

- **Inspect the error**. Our error didn't only come with an error message, there's a stack trace as well.
- **Logging**. If we want someone to work with these errors, we should look at logging them to a file.

Inspect the error with `runtime/debug`

There's a library `runtime/debug`. With this library, you can find out more about an error when it's thrown, information like stack trace, where the error originated. There's a function `Stack()` that produces the stack trace. Here's how to use it:

```
debug.Stack()
```

Log the error with `log`

While `runtime/debug` can produce a stack trace, what would be useful is logging all this error information to a file. To log to a file, use the `os` and `log` package like so:

```
f, err := os.OpenFile("logs", os.O_WRONLY|os.O_CREATE|os.O_APPEND, 0644)
 if err != nil {
  log.Println(err)
 }
 log.SetOutput(f)
```

Use the error pattern with the `errors` package

Other languages tend to use Exceptions to signal that something is wrong.

Go has a different and idiomatic approach. It wants you to create errors as return values to a function, next to the actual value being returned. You are then expected to inspect a function and see if it returns an error.

There's an `errors` package that can help us with the above approach.

Define an error

To define an error, we call the `New()` function with a string describing the error, here's an example:

```
var NoTooSmall = errors.New("the number is too small")
```

Next, let's look at how to add the error to a function.

Return an error

Let's start with function that uses a `panic()` as error handling:

```
func ReturnPositive(no int) int {
  if no > 0 {
    return no
  } else {
    panic("No too small")
  }
}
```

We can improve this function, by ensuring it always returns the result and an error, like so:

```
func ReturnPositive(no int) (int, error) {
  if no > 0 {
    return no, nil
  } else {
    return 0, NoTooSmall
  }
}
```

Note in the `if` clause that it returns `no` and `nil` when everything is fine. For the `else`, it returns a bogus value and our error `NoTooSmall`.

Inspect the result

Let's see how we would call the `ReturnPositive()` function and use this new pattern we established:

```
no, err := ReturnPositive(-2)
if err != nil {
  fmt.Println("error: ", err)
} else {
  fmt.Println("value:", no)
}
```

What you are seeing above is how we use an `if` clause to check for errors, if so, print out. On the `else`, we have our actual value.

Assignment I - add error handling

In this exercise, we'll add error handling to our program.

1. Create a file *panic.go* and give it the following content:

```
package main

import (
  "fmt"
)

 // func errorHandler() {
 // if r := recover(); r != nil {
 //  fmt.Println("Recovered ", r)
 // }
 // }

 func Divide(nominator int, divider int) float32 {
  // defer errorHandler()
  if divider == 0 {
   panic("can't divide by 0")
  }
  return float32(nominator) / float32(divider)
 }

 func main() {
  no := Divide(10, 0)
  fmt.Println(no)
  no = Divide(10, 1)
  fmt.Println(no)
 }
```

2. Run the program `go run panic.go`

```
go run panic.go
```

You should see output similar to:

```
panic: can't divide by 0

 goroutine 1 [running]:
 main.Divide(...)
        /<path>/panic.go:20
 main.main()
        /path/panic.go:33 +0x96
 exit status 2
```

Note how these two statements was never run as the program crashed:

```
no = Divide(10, 1)
fmt.Println(no)
```

3. Uncomment the commented out part and run the app again.
   You should now see the following output:

```
Recovered  can't divide by 0
0
10
```

Congrats, you've managed to implement error handling with `panic()` and `recover()`.

Solution I

```
package main

import (
   "fmt"
)

func errorHandler() {
   if r := recover(); r != nil {
      fmt.Println("Recovered ", r)
   }
}

func Divide(nominator int, divider int) float32 {
  // defer errorHandler()
  if divider == 0 {
    panic("can't divide by 0")
  }
  return float32(nominator) / float32(divider)
}

func main() {
  no := Divide(10, 0)
  fmt.Println(no)
  no = Divide(10, 1)
  fmt.Println(no)
}
```

Assignment II - improve error logging

Let's improve our *panic.go* file by adding error logging:

1. Locate the `errorHandler()` and change it to the following:

```
func errorHandler() {
  if r := recover(); r != nil {
    log.Println(r, string(debug.Stack()))
  }
}
```

2. Ensure that *panic.go* looks like so:

```
 package main

 import (
  "fmt"
  "log"
  "os"
  "runtime/debug"
 )

 func errorHandler() {
  if r := recover(); r != nil {
   log.Println(r, string(debug.Stack()))
  }
 }

 func Divide(nominator int, divider int) float32 {
  defer errorHandler()
  if divider == 0 {
   panic("can't divide by 0")
  }
  return float32(nominator) / float32(divider)
 }

 func main() {
  f, err := os.OpenFile("logs", os.O_WRONLY|os.O_CREATE|os.O_APPEND, 0644)
  if err != nil {
   log.Println(err)
  }
  log.SetOutput(f)
```

```
30
31    log.Println("starting program")
32    no := Divide(10, 0)
33    fmt.Println(no)
34
35    no = Divide(10, 1)
36    fmt.Println(no)
37    f.Close()
38   }
```

3. Run this program:

```
go run panic.go
```

You should see this output:

```
0
10
```

It was able to run all statements without being affected by the error that was thrown.

4. Inspect the *logs* file that was just created, it should have content like the below:

```
  2022/03/11 15:03:59 starting program

 2022/03/11 15:03:59 can't divide by 0 goroutine 1 [running]:
 runtime/debug.Stack(0xc000111d30, 0x10b1b40, 0x10eae78)
  /usr/local/Cellar/go/1.16/libexec/src/runtime/debug/stack.go:24 +0x9f
 main.errorHandler()
  /<path>/panic.go:14 +0x5b
 panic(0x10b1b40, 0x10eae78)
  /usr/local/Cellar/go/1.16/libexec/src/runtime/panic.go:965 +0x1b9
 main.Divide(0xa, 0x0, 0x0)
  /<path>/panic.go:21 +0xa5
 main.main()
  /<path>/panic.go:34 +0x115
```

Solution II

```go
package main

import (
  "fmt"
  "log"
  "os"
  "runtime/debug"
)

func errorHandler() {
  if r := recover(); r != nil {
    log.Println(r, string(debug.Stack()))
  }
}

func Divide(nominator int, divider int) float32 {
  defer errorHandler()
  if divider == 0 {
    panic("can't divide by 0")
  }
  return float32(nominator) / float32(divider)
}

func main() {
  f, err := os.OpenFile("logs", os.O_WRONLY|os.O_CREATE|os.O_APPEND, 0644)
  if err != nil {
    log.Println(err)
  }
  log.SetOutput(f)

  log.Println("starting program")
  no := Divide(10, 0)
  fmt.Println(no)

  no = Divide(10, 1)
  fmt.Println(no)
  f.Close()
}
```

Arrays and slices

In this chapter, we will cover arrays and slices.

Introduction

This chapter will cover:

- Declaring and inspecting an array.
- Accessing elements in the array.
- Working with slices.

Arrays

An array is a group of elements that are connected. You want to use an array when you have a group of something, like many orders, cars, and rows in a file.

The idea with an Array is to collect all that data in one structure. You also want to be able to iterate over it and carry out an operation on it as a group.

Declare an array

To declare an array, you need to specify the following properties:

- **capacity**, how many elements it holds.
- **type**, what type of elements it holds.
- **array content**, you can assign it elements at creation or do so later.

Here's the syntax:

```
1 [<capacity>]<type>{...element}
```

It starts with the square brackets, `[]`. Within the square brackets, you set the capacity, and how many elements it can hold.

and here's a more real example:

```
cities := [5]string{"NY", "LA"}
```

In the preceding code, an array of strings is declared. It has a capacity for 5 elements and two of the places are filled with “NY” and “LA”. Note also because we set the capacity to 5 and the number of elements it’s assigned is 2, there are 3 spaces free.

Capacity by inference

You don’t have to set capacity to an explicit number, you can set it to . . . , in which case the capacity will be set to the number of elements you assign to it, like so:

```
ids := [...]int{1, 2, 3, 4}
```

The preceding code has 4 elements, and that’s also its capacity.

Accessing elements

The way to access an element is by using its index. The index is 0-based, meaning the first index is 0 and its last is the length -1.

```
ids := [...]int{1, 2, 3, 4}
ids[0] // 1
ids[3] // 4
```

Length and capacity

Imagine we have the following array declared:

```
cities := [5]string{"NY", "LA"}
```

- **length**. The length is defined as the number of elements in the array. You can use the `len()` method to find this out:

```
 len(cities) // 2
```

- **capacity**. The capacity is how many elements the array can hold. `cap()` is the method you use to find the capacity:

```
cap(cities) // 5
```

Slices

A slice is a part of an array. A slice is created when the slice operator is being used. Here's the syntax for the slice operator:

```
s[i:p]
```

- `s`, the array
- `i`, the first index of the array to take elements from
- `p`, The variable p corresponds to the last element in the underlying array that can be used in the new slice. I.e. cut right before this index.

```
items := [5]int{1,2,3,4,5}
part = items[1:3] // 2,3
```

Adding elements

A slice differs from an array, you can add items to it. The `append()` method lets you add elements to it. The syntax for `append()` is as follows:

```
append(slice, element)
```

Here's how you can append to a slice:

```
var numbers []int
numbers = append(numbers, 1)
numbers = append(numbers, 2) // 1,2
```

Removing elements

Remove an element by constructing a new slice.

```
letters := []string{"A", "B", "C", "D", "E"}
remove := 2 // remove index
// 0 - remove index, remove +1 to end
letters = append(letters[:remove], letters[remove+1:]...)
```

Create a slice with `make()`

You can use the `make()` method to create a slice. Here's how:

```
slice := make([]int, 5) // creates a slice with length 5 and capacity 5
```

You can set these to different values:

```
slice2 := make([]int, 2, 5)
fmt.Println(slice2)
fmt.Println(len(slice2))
fmt.Println(cap(slice2))
```

Here, the slice has a length of 2, and a capacity of 5.

Copy elements

```
arr := [3]int{1, 2, 3}
dest := make([]int, 5)
copy(dest, arr[0:2]) // copies slice {1,2} into dest
fmt.Println(dest) // [1 2 0 0 0]
```

Assignment - store log entries

Create an array meant for log entries. It can be used in the following way:

```
command> new
here's a new entry
command> new
here's another entry
command> list
here's a new entry
here's another entry
command> quit
bye
```

So, you need to have a way to store multiple strings and list them when asked for.

Here's some starter code to deal with console input:

```
package main

import (
 "fmt"
)

func main() {
 // create array

 var response string
 for {
  fmt.Print("command> ")
  fmt.Scan(&response)
  if response == "quit" {
   break
  } else if response == "new" {
   fmt.Print("Entry:")
   fmt.Scan(&response)
   // save entry to list
   fmt.Println("Saving entry")
  } else if response == "list" {
   // list entries
   fmt.Println("Listing entries")
  } else {
   fmt.Println("Unknown command", response)
  }

 }
 fmt.Println("bye")
}
```

Add your own code where the comments are

Solution

```go
package main

import (
 "fmt"
)

func main() {
 // create array
 arr := make([]string, 0)

 var response string
 for {
  fmt.Print("command> ")
  fmt.Scan(&response)
  if response == "quit" {
   break
  } else if response == "new" {
   fmt.Print("Entry:")
   fmt.Scan(&response)

   // save entry to list
   arr = append(arr, response)
   fmt.Println("Saving entry")
  } else if response == "list" {
   // list entries
   fmt.Println("Listing entries")
   for i := 0; i < len(arr); i++ {
    fmt.Println(arr[i])
   }
  } else {
   fmt.Println("Unknown command", response)
  }

 }
 fmt.Println("bye")
}
```

Structs

In this chapter, we will learn about structs. A struct is a complex data type capable of holding many fields. It can also be extended to hold behaviour.

Introduction

This chapter will cover:

- Declaring and inspecting a struct.
- Embedding a struct within another struct.
- Adding implementations to structs.

Why structs

Let's start with a simple scenario, you have an account balance. You might store it in a variable like so:

```
1  accountBalance int32
```

Now that's great, but if you want to describe something more complex, like a bank account? A bank account consists of a variety of information like an ID, balance, account owner and so on. You could try representing each one of those properties as integers like so:

```
1  var accountBalance int32
2  var owner string
3  var id int
```

However, what happens if you need to operate on more than one bank account, I mean you could try to store it like so:

```
var accountBalance int32
var owner string
var id int

var accountBalance2 int32
var owner2 string
var id2 int
```

It doesn't scale though, what you need is a more complex type, like a `struct` that's able to group all this information like so:

```
type Account struct {
  accountBalance int32
  owner string
  id int
}
```

Defining a struct

Ok, so we understand why we need a struct, to gather related information, and we've seen one example so far `Account`. But let's try breaking the parts down and see how we go about defining a struct. Here's what the syntax looks like:

```
type <a name for the struct> struct {
  ... fields
}
```

Let's show another example but this time we create a struct for an address:

```
type Address struct {
 city   string
 street string
 postal string
}
```

Create a struct instance

To create an instance from a struct, we can use one of two approaches:

- **define a variable**, and set the fields after the variable declaration:

```
var address Address
address.city = "London"
address.street = "Buckingham palace"
address.postal = "SW1"
```

- **define all at once**, we can set all the values in one go as well:

```
address2 := Address{"New York", "Central park", "111"}
```

Embedding a struct

We can also embed a struct in another struct. Let's see we have our `Address` struct, an address is something that a higher level struct like `Person` can use. Here's how that can look:

```
type Person struct {
 name    string
 address Address
}
```

In this code, the `Person` struct has a field `address` of type `Address`.

To instantiate a struct, we can type like so:

```
person := Person{
  name: "chris",
  address: Address{
   city: "Stockholm",
  },
 }
```

Relying on default naming

Note how we created a field `address`, we can skip typing a few characters by defining it like so instead:

```
type Employee struct {
 Address
 company string
}
```

Note how we omit the name for the field and just type `Address`, this means the field name and field type will be the same name. Creating an instance from it is similar:

```
employee := Employee{
  Address: Address{
   city: "LA",
  },
  company: "Microsoft",
 }
```

Adding implementation to structs

Structs are by their very nature just data fields that describe something complex. You can add behaviour to it though by creating functions that operate on a struct. Here's an example:

```
func (a Address) string() string {
 return fmt.Sprintf("City: %s, Street: %s, Postal address: %s", a.city, a.street, a.\
postal)
}
```

We've added a `string()` method. The method *belongs* to `Address` and we can see that with `(...)` right after the `func` keyword that takes a `Address`. The rest of the implementation returns a formatted string via `Sprintf()`. Given the following code:

```
var address Address
address.city = "London"
address.street = "Buckingham palace"
address.postal = "SW1"
fmt.Println(address.string())
```

We would get the following output when calling `string()`:

```
City: London, Street: Buckingham palace, Postal address: SW1
```

Assignment - defining a struct

Define a struct representing a row in a shopping basket for an e-commerce store.

Here's example data:

```
Title, Description, Quantity, Price per unit, Total
LEGO set, 4000 pieces, 1, 600GBP, 600GBP
```

Write a program representing the shopping basket

Write a program that iterates over the shopping basket and calculates the total:

```
Title, Description, Quantity, Price per unit, Total
LEGO set, 4000 pieces, 1, 600GBP, 600GBP
Plushy, plush toy, 3, 5 GBP, 15GBP

Total: 615 GBP
```

Solution

Part I

```go
package main

import (
 "fmt"
)

type Row struct {
 Title       string
 Description string
 Quantity    int
 UnitPrice   float32
}

func main() {
 row := Row{
  Title:       "LEGO set",
  Description: "4000 pieces",
  Quantity:    1,
  UnitPrice:   600,
 }
 fmt.Println(row)
}
```

Part II

```go
package main

import (
 "fmt"
)

type Row struct {
 Title       string
 Description string
 Quantity    int
 UnitPrice   float32
}

func main() {
 row := Row{
  Title:       "LEGO set",
  Description: "4000 pieces",
  Quantity:    1,
  UnitPrice:   600,
 }
 row2 := Row{
  Title:       "Plushy",
  Description: "plush toy",
  Quantity:    3,
  UnitPrice:   5,
 }

 basket := make([]Row, 0)
 basket = append(basket, row)
 basket = append(basket, row2)

 var sum int = 0
 for i := 0; i < len(basket); i++ {
  current := basket[i]
  fmt.Println(current)
  sum += current.Quantity * int(current.UnitPrice)
 }
 fmt.Println("Total", sum)
}
```

Using maps

A map is a complex data structure that enables you to store things in a key-value fashion. This lets you implement scenarios like phone books, translation dictionaries and more.

Introduction

This chapter will cover:

- Defining a map.
- Reading the values of map by key but also iterating over it.
- Change the content of a map.

The use case for a map

You will have scenarios when you code that there are things you need to look up. If you use a dictionary for example you might look up how you can translate a word from English to Spanish or vice versa. In programming, you have similar situations, maybe you want to know what service is run on a certain port for example. There are also databases that are based on the concept of having a unique key that points to a certain value.

How all this is implemented is via map structure. The idea is that you define a key and value and collect all those in a group, a map.

Creating a map

To create a map in Go, we need to use the following syntax:

```
map[<key type>]<value type>{ ... entries }
```

Here's an example of creating a map structure that could hold a phone book:

```
phonebook := map[int]string{ 555123: "Robin Hood", 555404: "Sheriff of Nottingham"}
```

We define a map structure with key type `int` and value type string. Then we assign it a value with `{}`. Each entry is defined according to `<key>: <value>` and separated by a comma. So how do we read a value?

Create a map with `make()`

Another way to create a map us by using the `make()` function. `make()` returns a initialized map if you give it a type like so:

```
dictionaryEnSv = make(map[string]string)
```

Adding entries

To add entries to the map, you need to provide it with a key and value entry like so:

```
dictionaryEnSv["hello"] = "hej"
```

Read a value by key

Imagine now that we have these two entries, and you want the value given that you have then entry 555404, how would we do that? We use the square brackets like so `[]`:

```
phonebook[555404] // "Sheriff of Nottingham"
```

Check for existing entry

So, you learned that `phonebook[555404]` gives you a value back. What if it doesn't exist? What happens if you give it a key that's not stored in the map is that you get nothing back as a result:

```
phonebook[888] // this prints as empty in the console
```

There's a better way to check this because accessing an entry with a key returns two values, the value, and a Boolean. The Boolean indicates if this key exists in the map. See this code:

```
_, exist phonebook[888]
fmt.Println(exist) // false
```

Here you get the value back, but you choose to ignore for this one-time occasion to only focus on `exist`, a Boolean that tells you if the entry exists.

You can even use this construct in an if statement:

```
if _, exist := phonebook[888] {
  // number exist, call person
}
```

Iterate over a map

We can iterate over a map with a `for` construct and a `range`. Here's how you can iterate:

```
for key, value := range phonebook {
  fmt.Println(key, value)
}
```

Delete an entry

To remove an entry from a map, you can use the `delete()` method. The `delete()` method takes the map and the key to delete as parameters, like so:

```
delete(phonebook, 555404)
```

Assignment - build a phone book

Here are your contacts:

```
Alice 555-123
Bob 555-124
Jean 555-125
```

```
Welcome to your phonebook.
Command> store
Enter contact: Rob 555-126
Contact saved
Command> list
Alice 555-123
Bob 555-124
Jean 555-125
Rob 555-126
Command> lookup
Enter name: Alice
Alice has number: 555-123
```

HINT: you might need to use both a map and a slice.

Solution

```
package main

import "fmt"

func main() {
 var command string
 contacts := make(map[string]string)
 fmt.Println("Welcome to your phonebook")

 for {
  fmt.Print("Command> ")
  fmt.Scan(&command)
  if command == "store" {
   fmt.Print("Enter contact: ")
   var contact string
   var no string
   fmt.Scan(&contact, &no)
   contacts[contact] = no
   fmt.Println("Contact saved")
  } else if command == "list" {
   for key, value := range contacts {
    fmt.Println(key, value)
   }
  } else if command == "lookup" {
   fmt.Print("Enter name: ")
   var contact string
   fmt.Scan(&contact)
   fmt.Println(contacts[contact])
  } else if command == "quit" {
   break
  } else {
   fmt.Println("Unknown command: ", command)
  }
 }
 fmt.Println("Bye")
}
```

Challenge

Right now, there's no error checking. Add a check so that if you look up a contact that doesn't exist, you should get an error message. Here's how it could work:

```
command> lookup
Enter name: Jane
Contact doesn't exist, do you want to add it? y/n: y
Enter contact: Jane 123
Contact saved
```

Interfaces

This chapter covers what an interface is and what to use it for.

Introduction

This chapter will cover:

- What is an interface and how does it differ from a struct.
- How to add behaviour.
- Implement an interface.
- Type assertions.
- Changing a value.

Interface

To describe what an interface is, let's start by talking about structs and how they are different from an interface.

With structs, we can define properties we want a concept to have, like for example a car:

```
type Car struct {
  make string
  model string
}
```

An interface is meant to communicate something different, a behaviour. Instead of describing the car itself, as a struct does, it describes what a car can do.

Interface - describing a behaviour

Now that we've described how an interface differs from a struct, let's talk about the motivation for using an interface. There are a couple of good reasons for when to use an interface:

- **Adding behaviour**. When you want your types to have a behaviour, that's when you want an interface

- **Communicate via contract**. Often, when you call other code, you want to reveal as little of your concrete implementation as possible. Instead of saying, here's a car, you might want to say, here's something that can run. It enables your code to be flexible and you don't have to implement specific code for each type but can instead write code that deals with a certain behaviour.

Define an interface

To define an interface, you need the keywords `type` and `interface` and you need a set of methods, one or many that a type should implement. Here's an example interface:

```
type Describable interface {
  describe() string
}
```

Here's another example:

```
type Point struct {
  x int
  y int
}

type Shape interface {
  area() int
  location() Point
}
```

Implement an interface

Everything that's a type can implement an interface. More than one type can implement the same interface. Let's look at how a type `Rectangle` can implement the `Shape` interface:

```
type Rectangle struct {
  x int
  y int
}

func (r Rectangle) area() int {
  return r.x * r.y
}

func (r Rectangle) location() Point {
  return P{ x: r.x, y: r.y }
}
```

So, what's going on here? Let's look at the first method `area()`:

```
func (r Rectangle) area() int {
  return r.x * r.y
}
```

It looks like a regular function but there's this `(r Rectangle)` right before the function name. That's a signal to Go that you are implementing a certain function on the type `Rectangle`. There's also a second implementation for `location()`.

By implementing both these methods, `Rectangle` has now fully implemented the `Shape` interface.

Pass an interface

Ok, so we've fully implemented an interface, what does it allow me to do? Well, there are two things you can do:

- **Call properties and behaviour**. At this point, you are ready to create an instance and call both properties and methods (its new behaviour):

```
  var rectangle Rectangle = Rectangle{x: 5, y: 2}
  fmt.Println(rectangle.area()) // prints 10
```

  Great, our `Rectangle` type has both the properties `x` and `y` as well as the behaviour from `Shape`.
- **Pass an interface**. Imagine you wanted to pass the behaviour to a function to make it flexible:

```
func printArea(shape Shape) {
  fmt.Println(shape.area())
}
```

To make that happen, lets change slightly how we construct our `Rectangle instance`:

```
var shape Shape = Rectangle{x: 5, y: 2}
printArea(rectangle) // prints 10
```

Implement `Square`

To see the power in what we just created, let's create another struct `Square` and have it implement `Shape`:

```
type Square struct {
  side int
}

func (s Square) area() int {
  return s.square * s.square
}
func (s Square) location() Point {
  return Point{x: s.side, y: s.side}
}

func main() {
  var shape Shape = Rectangle{x: 5, y: 2}
  var shape2 Shape = Square{side: 5}
  printArea(shape) // prints 10
  printArea(shape2) // prints 25
}
```

The power lies in the fact that `printArea()` doesn't have to deal with the internals of `Rectangle` or `Shape`, it just needs the parameter to implement `Shape`, a behaviour.

Full code

Here's the full code:

```
package main

import "fmt"

type Rectangle struct {
 x int
 y int
}

type Point struct {
 x int
 y int
}

type Square struct {
 side int
}

type Shape interface {
 area() int
 location() Point
}

func printArea(shape Shape) {
 fmt.Println(shape.area())
}

func (r Rectangle) area() int {
 return r.x * r.y
}

func (r Rectangle) location() Point {
 return Point{x: r.x, y: r.y}
}

func (s Square) area() int {
 return s.side * s.side
}

func (s Square) location() Point {
 return Point{x: s.side, y: s.side}
}

```

```
44 func main() {
45  var shape Shape = Rectangle{x: 5, y: 2}
46  var shape2 Shape = Square{side: 5}
47  printArea(shape)  // prints 10
48  printArea(shape2) // prints 25
49 }
```

Type assertions

So far, a `Rectangle` or `Square` implements the `Shape` interface

Let's have a closer look at this code:

```
1 var shape Shape = Rectangle{x: 5, y: 2}
2 var shape2 Shape = Square{side: 5}
3 printArea(shape)  // prints 10
4 printArea(shape2) // prints 25
```

We've said for `shape` and `shape2` to be of type `Shape`. That's great for being sent to the `printArea()` method. What if we need to access a `Rectangle` property on `shape`, can we? Let's try:

```
1 var shape Shape = Rectangle{x: 5, y: 2}
2 fmt.Println(shape.x) // shape.x undefined (type Shape has no field or method x)
```

Ok, not working, we need to find a way to reach the underlying fields. We can use something called *type assertion* like so:

```
1 var shape Shape = Rectangle{x: 5, y: 2}
2 fmt.Println(shape.(Rectangle).x) // 5
```

Ok, that works, so `.(<type>)` works, if the underlying type is the correct type.

Change a value

So, one thing about our approach so far is that we have implemented interfaces with methods that read data from the underlying struct instances. What if we want to change data, can we do that?

Let's look at an example:

```
package main
import "fmt"

type Car struct {
 speed int
 model string
 make string
}

type Runnable interface {
 run()
}

func (c Car) run() {
 c.speed = 10
}

func main() {
  c := Car{make: "Ferrari", model: "F40", speed: 0}
  c.run()
  fmt.Println(c.speed) // ?
}
```

Running this code, it returns `0`. So, looking at our `run()` method:

```
func (c Car) run() {
 c.speed = 10
}
```

shouldn't this work? Well, no, because you are not changing the instance. For that, you need to send a reference.

A slight alteration to the `run()` method, with `*`:

```
func (c *Car) run() {
 c.speed = 10
}
```

and your code now does what it's supposed to.

Assignment

Start with the following code:

```
package main

type Point struct {
 x float32
 y float32
}

type Vehicle struct {
 velocity float32
 Point
}

func main() {
 v := Vehicle{
  velocity: 0,
  Point: Point{
   x: 0,
   y: 0,
  },
 }
 v.fly()
 fmt.Println(v.velocity)
 v.land()
 fmt.Println(v.velocity)
}
```

Implement the following interface:

```
type Spaceship interface {
 fly()
 land()
 position() Point
}
```

The output from running the program should be:

```
10
0
```

Solution

```go
package main

import "fmt"

type Point struct {
 x float32
 y float32
}

type Vehicle struct {
 velocity float32
 Point
}

type Spaceship interface {
 fly()
 land()
 position() Point
}

func (v *Vehicle) fly() {
 v.velocity = 10
}

func (v *Vehicle) land() {
 v.velocity = 0
}

func (v Vehicle) position() Point {
 return v.Point
}

func main() {
 v := Vehicle{
  velocity: 0,
  Point: Point{
   x: 0,
   y: 0,
  },
 }
 v.fly()
 fmt.Println(v.velocity)
 v.land()
```

```
44   fmt.Println(v.velocity)
45  }
```

Your first project

In this chapter, we will cover how to create your first project in Go.

Introduction

This chapter will cover:

- Creating a project.
- Organize your files.

Module use cases

There are two interesting use cases with modules:

- **Consuming a module**, you will use a combination of core modules and external 3rd party modules
- **Creating a module**, in some cases you will create code that you or someone else will be able to use. For this scenario, you can create a module and upload it to GitHub.

Consume internal files

You want to split up your app in many different files. Let's say you have the following files:

```
/app
  main.go
  /helper
    helper.go
```

What you are saying above is that your program consists of many files and that you want code in the fiile *main.go* to use code from *helper.go* for example.

To handle such a case, you need the following:

- **a project**. By creating a project, you create a top-level reference that you can use in the `import` directive.
- **an import** that points to the project root name as well as the path to the module you want to import.

You can use `go mod init`, this will initialize your project.

Creating a project

To create a project, you run `go mod init` and a name for a project, for example, “my-project”:

```
go mod init my-project
```

You end up with a *go.mod* file looking something like so:

```
module my-project

go 1.16
```

The *go.mod* file tells you the name of your project and the currently used version of Go. It can contain other things as well like libraries you are dependent on.

The import statement

Imagine now we have this file structure in our project:

```
/app
  main.go
  /helper
    helper.go
```

with *helper.go* looking like so:

```
package helper

import "fmt"

func Help() {
  fmt.Println("This is a helper function")
}
```

to use the public `Helper()` function from *main.go*, we need to import it.

In *main.go* we need an import statement like so:

```
import (
  "my-project/helper"
)
```

We are now able to invoke the `Help()` function from *main.go* like so:

```
helper.Help()
```

Assignment - create a project

In this assignment, you will create a project.

1. Create a project like so:

```
 go mod init
```

2. create the **helper** directory and *helper.go* file and give it the following content:

```
 // helper.go

 package helper

 import "fmt"

 func Help() {
  fmt.Println("This is a helper function")
 }
```

3. Create the *main.go* file and give it the following content:

```
package main

import (
  "log-tester/helper"
)

func main() {
  helper.Help()
}
```

   Note this import `"log-tester/helper"`, it ensures the `helper` package is in scope.
4. Compile and run

```
go run main.go
```

Solution

helper/helper.go

```
package helper

import "fmt"

func Help() {
 fmt.Println("help")
}
```

main.go

```
package main

import "my-project/helper"

func main() {
 helper.Help()
}
```

Challenge

See if you can create another function in *helper.go*, this time, make the function name lowercase, what happens if you try to import it?

Consume an external module

In this chapter, we are looking at downloading and using external modules.

Introduction

This chapter will cover:

- Creating a project.
- Adding an external module to your project.
- Use the external library in your app.

External module

To consume an external module, you need to:

- **Import it**, involves using the `import` instruction and fully qualifying the address to the module's location.
- **Use it in code**, call the code from the module that you mean to use
- **Ensure it's downloaded**, so your code can be run.

Import module

To import a module, you can do one of two things:

- `go get <path to module>`, this will fetch the module and download it and make it available for your project to use.
- `go mod tidy`, this command checks the imports used in your program and fetches the module if not fetched already.

Use it in code

To use your module in code, you need to add it to the `import` section and then invoke it where you need it in the application code.

```
import (
  "github.com/<user>/<repo name>"
)

func main() {
  <repo name>.<Function>()
}
```

Here's an example:

```
package main

import (
"fmt"
"github.com/softchris/math"
)

func main() {
  math.Add(1,1) // 2
}
```

Assignment - consume an external module

Let's create a new project

1. Run `go mod init`:

```
    go mod init hello
```

Note how *go.mod* was created with the following content:

```
    module hello

    go 1.16
```

Add reference to an external lib

Next, lets create some code that will use the external library:

1. Create the file *main.go*

```
package main

import (
  "fmt"
  "github.com/softchris/math"
)
```

2. To the same file, add `main()` function and use external function, from package:

```
func main() {
  sum += math.Add(1,2)
  fmt.Println(sum)
}
```

Fetch the lib

Now, we need to resolve the external library.

1. Run `go mod tidy`:

```
go mod tidy
```

Your *go.mod* is updated:

```
require github.com/softchris/math v0.2.0
```

There's also *go.sum* file with the following content:

```
github.com/softchris/math v0.2.0 h1:88L6PLRBGygS3LY5KGIJhyw9IZturmd2ksU+p13OPa4=

github.com/softchris/math v0.2.0/go.mod h1:v8WzhjKC+ipuH+i9IZ0Ta2IArniTP53gc5TgCINC\
qAo=
```

This is Go's way of keeping track of how to build the app by referencing to the go module in question.

2. Run `go run`:

```
go run main.go
```

Running the program gives you the following output:

```
3
```

Solution

go.sum

```
github.com/softchris/math v0.2.0 h1:88L6PLRBGygS3LY5KGIJhyw9IZturmd2ksU+p13OPa4=
github.com/softchris/math v0.2.0/go.mod h1:v8WzhjKC+ipuH+i9IZ0Ta2IArniTP53gc5TgCINCq\
Ao=
```

go.mod

```
module hello

go 1.16

require github.com/softchris/math v0.2.0
```

main.go

```
package main

import (
  "fmt"
  "github.com/softchris/math"
)

func main() {
  sum += math.Add(1,2)
  fmt.Println(sum)
}
```

Challenge

See if you can find another module you want to use in your project. Add it to the project and use it in your code.

Create a module meant for sharing

In this chapter, we will cover how you can create a module you can share with others.

Introduction

This chapter will cover:

- Creating a project.
- Testing the module locally.
- Tag the module with different versions.
- Try consuming the module as an external library.

Create a module

When you build a module meant for sharing, there's some gotchas:

- You need to create a package.
- Your package won't be called main.
- There's the concept of public and private parts of your code.
- You can test it locally.
- Upload your package to GitHub for wide distribution.

Assignment - create a module meant for sharing and consume it

To create a module meant for wider use you need to first initialize a module.

1. Create a directory *logger* for your new package:

```
mkdir logger
cd logger
```

2. Run `go mod init <address at github>`, for example:

```
go mod init github.com/softchris/logger
```

This will create a *go.mod* file in your directory.

```
logger/
  go.mod
```

The file looks like so:

```
module github.com/softchris/logger

go 1.16
```

It contains the package name and the version of Go it means to use.

3. Create a file to host your package code, for example *log.go* and give it the following content:

```
 package logger

 import (
  "fmt"
 )

 var Version string = "1.0"

 func Log(mess string) {
  fmt.Println("[LOG] " + mess)
 }
```

- Note `package logger` instead of `main`.
- The uppercase variables and methods makes the publicly available. Anything named with lowercase will be private for the package.

Test it locally

You can test your package locally. To do so you need a separate package that you can import your package from.

1. Move up a directory:

```
cd ..
```

2. Create a new directory **logger-test**:

```
mkdir logger-test
cd logger-test
```

3. Create a package, it will be used for testing only:

```
go mod init logger-test
```

4. Create a file *main.go* and add the following code:

```
 package main

 import "github.com/softchris/logger"

 func main() {
  logger.Log("hey there")
 }
```

   At this point, you are consuming the "logger" package but it's pointing to GitHub and your package doesn't live there yet. However, you can repoint to a local address on your machine, let's do that next.

5. Open *go.mod* and add the following:

```
require github.com/softchris/logger v0.0.0

replace github.com/softchris/logger => ../logger
```

   Two things are happening here:

   - you are asking for the "logger" package:

   ```
    require github.com/softchris/logger v0.0.0
   ```

   - you are making it point to your local system instead of GitHub

   ```
    replace github.com/softchris/logger => ../logger
   ```

6. Run the package with `go run`:

```
 go run main.go
```

   You should see:

```
 [LOG] hey there
```

Publish a package

To publish your package, you can put it on GitHub.

1. Create a git repo with `git init`:

```
git init
```

2. Create the repo on GitHub.
3. Make you do at least one commit:

```
 git add .
 git commit -m "first commit"
```

4. Do the following to upload your package to GitHub:

```
git remote add origin https://github.com/softchris/logger.git

git branch -M main
git push -u origin main
```

5. Tag your package with `git tag`:

```
git tag v0.1.0
git push origin v0.1.0
```

   Now your package has the tag 0.1.0

Test it out

1. Go to your project "logger-test":

```
cd ..
cd logger-test
```

2. Open up *go.mod* and remove these lines:

```
require github.com/softchris/logger v0.1.0
replace github.com/softchris/logger => ../logger
```

3. Run `go mod tidy`, this will force Go to go look for the package:
   Your *go.mod* should now contain:

```
require github.com/softchris/logger v0.1.0
```

   Also, your *go.sum* should contain:

```
github.com/softchris/logger v0.1.0 h1:Kqw7t9C3Y7BtHDLTx/KXEqHy5x8EJxrLian742S0di0=

github.com/softchris/logger v0.1.0/go.mod h1:rrzWjMsM3tqjetDBDyezI8mFCjGucF/b5RSAqpt\
KF/M=
```

4. Run the program with `go run`:

```
go run main.go
```

You should see:

```
[LOG] hey there
```

Challenge

See if you can add a feature to your new package. Give it a new tag via Git. Then ensure your app is using this new version.

Test your code in Go

In this chapter, we will look at creating and running unit tests.

Introduction

This chapter will cover:

- Why you should test your code.
- The testing library in Go.
- Authoring and running a test.
- Controlling how to run your tests.
- Produce coverage reports.

Why we test

It's good to test your code to ensure it works as intended. In this chapter, we're looking at unit tests specifically.

What Go provides

Go has a package `testing` that gives us two things to start out with:

- a parameter for the test. The `testing` library exposes a `t *testing.T` parameter. By putting it as a parameter to a function, said function becomes a test.

```
1   func TestAdd(t *testing.T) {}
```

- a way to assert the result. `testing` also exposes `t.Errorf()`. By invoking it with a string, the test counts as failed. To pass a test you do nothing:

```
1   t.Errorf("Sum was incorrect, Actual: %d, Expected: %d", total, 4)
```

Here's an example test function:

```
func TestAdd(t *testing.T) {
 total := Add(2, 2)
 if total != 4 {
  t.Errorf("Sum was incorrect, Actual: %d, Expected: %d", total, 4)
 }
 t.Log("running TestAdd")
}
```

- First, the code to test is called:

```
  total := Add(2, 2)
```

- Secondly, the assertion is made, to see if it succeeded or failed:

```
  if total != 4 {
    t.Errorf("Sum was incorrect, Actual: %d, Expected: %d", total, 4)
  }
```

  if the result is not the expected, then `t.Errorf()` is called to state what's gone wrong.

Your first test

Make sure you've created a project with `go mod init`. Then create a file structure like so:

```
main.go
math/
  math.go
```

What you want to do next is to create a test file. You want to keep the test file as close to the code you want to test as possible. Because you want to test *math.go* you create `math_test.go` file in the math/ directory like so:

```
main.go
math/
  math.go
  math_test.go
```

Authoring and running your first test

Now that you have the file structure above, ensure the *math_test.go* file has the following content:

```
package math

import (
 "testing"
)

func TestAdd(t *testing.T) {
  total := Add(2, 2)
  if total != 4 {
    t.Errorf("Sum was incorrect, Actual: %d, Expected: %d", total, 4)
  }
  t.Log("running TestAdd")
}
```

To run a test, you invoke the `go test` command. Here are different ways to run your tests:

- `go test`, runs the test in the current working directory. Here's what it looks like:

```
  ok      test-example/math       0.258s
```

- `go test -v`, runs a verbose version. Here's what it can look like:

```
  === RUN   TestAdd
   math_test.go:12: running TestAdd

  --- PASS: TestAdd (0.00s)
  PASS
  ok      test-example/math       0.422s
```

  In the verbose version, you see the name of the test and if it failed.
- `go test ./..`, recursive run. If you run the command like so it will run all the tests in the subfolders as well.

Control the test run

There are ways to control how many tests are run. Here are some ways:

- **Run test by pattern**. . You can provide a pattern to have Go run some of the tests, which matches it by a substring or even at a certain depth and more. Here's how:

```
    go test -run <pattern>
```

- **Skip a test**. `t.Skip()` by calling this inside the test, the test is skipped.

- **Run a single test**. You can run a single test by running a pattern that specifies the package and the name of the test, here's how:

```
go test -run TestAdd ./math
```

Here's the package is called math and the name of the test is `TestAdd()`.

Coverage

There's a built-in tool for dealing with coverage. To learn more about the tool, you can type:

```
go tool cover -help
```

it will list a set of commands.

The tool is centred on the concept of having an out file. The out file contains instructions on where your code is covered by tests and where it isn't. An out file can look something like this:

```
mode: set
test-example/math/math.go:3.32,5.2 1 1
test-example/math/math.go:7.37,9.2 1 1
test-example/math/math.go:11.47,13.2 1 0
```

This is a format readable by the tool.

It's a prerequisite to generate said "out file" before you can view your code's coverage. Place yourself in the directory you mean to measure coverage on and run this command:

```
go test -coverprofile=c.out
```

Now you are ready to run a command that shows the result in a browser:

```
go tool cover -html=c.out
```

The above command spins up a browser and the output look something like so:

coverage

The coverage report tells us that the the green portions are covered by tests whereas the red portions should have tests covering it.

Learn more

There's a lot more to learn on testing with Go, have a look at package documentation, docs[7]

[7] https://pkg.go.dev/testing

Challenge

Create a test for a piece of code you wrote. Run the test. See if you can produce a coverage report and implement any gaps pointed out by the report.

Working with JSON

In this chapter, you will work with the JSON data format.

Introduction

In this chapter, you will learn the following:

- The JSON data format.
- How to read JSON data and map it to existing structures in Go.
- How to create JSON and persist it in files.

JSON

JSON is a lightweight data format for storing and transporting data. The acronym stands for **J**ava**S**cript **O**bject **n**otation.

The format is commonly used in Web services. Usually, data is encoded as JSON and sent via HTTP. A client, for example, a web browser, consumes the JSON data and uses it to render a frontend with the help of HTML and CSS. It's also common for JSON to be used to communicate between services.

Here's an example of what the format looks like:

```
{
  "products": [{
    "id": 1,
    "name": "a product"
  },
  {
    "id": 2,
    "name": "another product"
  }]
}
```

The above depicts a list of products. Each key needs to be encased with quotes and values can be everything from primitives like numbers, strings, Booleans etc to more complex types like an array or an object.

what is this format, and what contexts is it used in.

Reading JSON

To work with JSON, you need to use the `encoding/json` library. It allows you to both read and write JSON data.

To read JSON data you need to first have a structure in your Go code read to map to the JSON data. Imagine having the following JSON data:

```
{
  "products" : [
    {
      "id": 1,
      "name": "some product"
    }
  ]
}
```

To map this in Go, you need a struct that matches its structure like so:

```
type Product struct {
  Id int
  Name string
}

type Response struct {
  Products []Product
}
```

Notation

The first step is to create the structures needed to match your JSON data. But you also need to do one more thing, add notations. The problem we are looking to solve is how the `Product` property is called `Id` and not `id` as it's called in the JSON data.

> Won't that just work?

Unfortunately, not

> Why don't you just name the struct property `id`

The library `encoding/json`, is a separate package from the main package, meaning we need to make a field name, defined in the main package, uppercase for the `encoding/json` package to find it.

So that means, we need to add the following annotations to our above created structs:

```
type Product struct {
  Id int `json: "id"`
  Name string `json: "name"`
}

type Response struct {
  Products []Product `json: "products"`
}
```

What these annotations do is to say, in the JSON data, look for properties with these names and map them to the following property. Like in this example from above:

```
Id int `json: "id"`
```

Reading the data

Ok, so we've defined the structures in Go that we will map our JSON data to. So how do we read from a JSON source? Well, JSON is usually stored in one of two ways:

- a string literal, like so `{ "name": "my product", "id": 1 }`
- in a JSON file:

```
  {
    "id": 1,
    "name": "my product"
  }
```

Let's show how to work with both approaches:

Reading from a string literal

We will use the `Unmarshal()` function and provide it with the string literal as the first parameter and the data to write the result to as the second parameter.

```
package main

imports (
  "fmt"
  "encoding/json"
)

func main() {
  str := `{ "name": "my product", "id": 1 }`
  product := Product{}
  json.Unmarshal([]byte(str), &person)
  fmt.Println(person) // prints the object
}
```

Note how we also convert the response to a byte array `[]byte(str)` and how the data is written in the second parameter to the `person` instance as a reference, `&person`.

read from a file

To read from a file, we will use the `io/ioutil` library and its `ReadFile()` function. Like with the string literal, the `Unmarshal()` function will be used to write the data to a struct instance.

```
package main

import (
 "encoding/json"
 "fmt"
 "io/ioutil"
 "iohelper/dir"
 "iohelper/file"
 "log"
)

type Products struct {
 Products []Product `json: products`
}

type Product struct {
 Id   int    `json: "id"`
 Name string `json: "name"`
}

func main(){
  file, _ := ioutil.ReadFile("products.json")
```

```
23
24    data := Products{}
25
26    _ = json.Unmarshal([]byte(file), &data)
27
28    for i := 0; i < len(data.Products); i++ {
29      fmt.Println("Product Id: ", data.Products[i].Id)
30      fmt.Println("Name: ", data.Products[i].Name)
31    }
32  }
```

Writing JSON

We've seen so far how we can read JSON data either from a string literal or from a file, but what about writing data?

Two cases are important for us:

- Writing data to a structure. We are working with structs so any changes we do on the structs need to be converted back to JSON.
- Generate JSON from data. In the second case, we might be working with a raw string literal or even with pure primitives, how would we do that?

Let's address both these cases. What these cases have in common is the `Marshal()` function. The `Marshal()` method can take a primitive or a struct and taking that into JSON:

```
1  package main
2
3  import (
4   "fmt"
5   "encoding/json"
6  )
7
8  type Person struct {
9    Id int `json: "id"`
10   Name string`json: "name"`
11 }
12
13 func main() {
14   aBoolean, _ := json.Marshal(true)
15   aString, _ := json.Marshal("a string")
16   person := Person{
```

```
    Id: 1
    Name: "a person"
  }
  aPerson, _ := json.Marshal(&person)
  fmt.Println(string(aBoolean)) // true
  fmt.Println(string(aString))  // a string
  fmt.Println(string(person))  // { "id": 1, "name": "a person" }
}
```

Assignment

Given the following file *response.json*, find a way to read the data and display it on the screen:

```
{
  "orders": [
   {
    "id": 1,
    "items": [
      { "id": 1, "quantity": 3, "total": 34.3 },
     { "id": 2, "quantity": 2, "total": 17.8 }
    ]
   },
   {
    "id": 2,
    "items": [
      { "id": 3, "quantity": 3, "total": 10.0 },
      { "id": 4, "quantity": 2, "total": 100.5 }
    ]
   }
  ]
}
```

Solution

```go
package main

import (
 "encoding/json"
 "fmt"
 "io/ioutil"
)

type OrderItem struct {
 Id       int     `json: "id"`
 Quantity int     `json: "quantity"`
 Total    float32 `json: "total"`
}

type Order struct {
 Id    int         `json: "id"`
 Items []OrderItem `json: items`
}

type Response struct {
 Orders []Order `json: orders`
}

func main() {
 file, _ := ioutil.ReadFile("orders.json")

 data := Response{}

 _ = json.Unmarshal([]byte(file), &data)
 fmt.Println(data)
 for i := 0; i < len(data.Orders); i++ {
  fmt.Println("Order Id: ", data.Orders[i].Id)

  for j := 0; j < len(data.Orders[i].Items); j++ {
   item := data.Orders[i].Items[j]
   fmt.Println("Item id", item.Id)
   fmt.Println("Item quantity", item.Quantity)
   fmt.Println("Item total", item.Total)
  }
 }
}
```

Challenge

See if you can add `products` to your JSON file. Here's the JSON for it:

```
"products" :[{
  "id": 1
  "name" "product"
}]
```

What structs do you need and how would you iterate over them?

Learn more

https://gobyexample.com/json

Build a Web API

A Web API is usually what we interact with to serve data to our services or a client. Said client may either be web page or tool like curl. In this chapter, we will learn to build a Web API that will process requests via HTTP.

Introduction

In this chapter, you will learn the following:

- What's a Web API.
- The `net/http` library, and its capabilities at high-level.
- Responding to a request.
- Working with request data like router and query parameters but also the body.
- Using ServeMux, and why it may the preferred choice.

Web API

Common responsibilities for web services are to respond to requests:

- **asking for data** and serve data like JSON, XML images, CSS, HTML
- **asking to modify a resource** either by creating, updating, or deleting it.

The `net/http` library

There's a library `net/http` that will help us build a web server. Building a web server with this library involves the following:

- Create a server instance.
- Define route requests and how to respond to them.
- Start the server instance, making sure it's accessible on a certain address and port.

Create a server instance

In `net/http`, `http` represents your service instance.

```
import (
 "fmt"
 "net/http"
)

func main() {
  // do something with `http`
}
```

Define routes

A route is you defining logical separations in your app like `products`, `orders` or some other area it makes sense to divide your app in.

To define a route, you define a route pattern and function that is invoked when the route is hit:

```
func hello(w http.ResponseWriter, req *http.Request) {
  fmt.Fprintf(w, "hello\n")
}

func main(){
  http.HandleFunc("/hello", hello)
}
```

In the code above, the string "/hello" is a route pattern that states that all web requests to "/hello" should be handled by the `hello()` function.

Response and Request

A close inspection of the `hello()` function reveals that it takes a `ResponseWriter` and `Request`:

```
func hello(w http.ResponseWriter, req *http.Request) {
  fmt.c(w, "hello\n")
}
```

The expectation is that you inspect the `req` object, your request for any data that decides what to return. Then you are to use `w` to produce a response. In this case, you are returning a string by passing `w` to `Fprintf()`. `FPrintf()` takes a writer. The writer is anything IO, so it could be, be writing to a file for example as well, or as in this case an HTTP response stream.

Start the server

Ok, we've gone through routes, producing a response, how would we get this server activated so it starts responding to requests?

You use the `ListenAndServe()` function that takes a port like so:

```
http.ListenAndServe(":8090", nil)
```

Responding to a request

An incoming request could be asking for a specific route like /products or /orders for example, or it could be asking for a specific static file like an image, a text file or maybe CSS. The request itself gives us a hint, about not only the logical domain it wants data from, like orders or products but what data type it wants, or it might even present credentials for authentication. The hint is known as headers.

Header

There's a concept called headers. A header is giving off a piece of information that could say what piece of content it is, how big the content is, or it could be a token helping you authenticate for example.

Headers can exist both on the incoming request as well as the response.

Serving different types of content

Serving different types of content means that we are working on the response. To serve various content type, we need to instruct the response on what type of content it is so that a consuming client knows how to interpret it, (in some cases, clients like a web browser can figure that out anyway through a process called content sniffing).

To serve a specific type of content, there are two things you need to do:

- **set the content type**, you set the content type by calling:

```
 w.Header().Set("Content-Type", "image/jpeg")
```

  Here the content type is an image of subtype jpeg. There are many content types you could be setting like plain text, CSS, JSON, XML and more.
- **produce the response**. Producing a response means writing to the response stream. That can be done by calling the `Write()` method on the `ResponseWriter` instance we are passed when we handle a route. There are other methods capable of writing to said stream as well.

Serving image data

To serve an image, you need to load it into memory, set the content type and write it to the response stream like below code:

```
func GetImage(w http.ResponseWriter, r *http.Request) {
    f, _ := os.Open("/image.jpg")

    // Read the entire JPG file into memory.
    reader := bufio.NewReader(f)
    content, _ := ioutil.ReadAll(reader)

    // Set the Content Type header.
    w.Header().Set("Content-Type", "image/jpeg")

    // Write image to the response.
    w.Write(content)
}
```

- First, we open the image:

```
f, _ := os.Open("/image.jpg")
```

- Secondly, we read the file into memory:

```
reader := bufio.NewReader(f)
content, _ := ioutil.ReadAll(reader)
```

- Thirdly, set the `Content-Type` header and tell it it's a JPEG image, with the value "image/jpeg":

```
w.Header().Set("Content-Type", "image/jpeg")
```

- Finally, we write the content to the response:

```
w.Write(content)
```

Serving JSON data

Just like with serving images, we need to follow a similar approach of configuring the correct content-type header and then constructing the response. Here's the code:

```
package main

import (
  "encoding/json"
)

type Person struct {
  Id int
  Name string
}

func ReturnJson(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "application/json")

  p := Person {
    Id: 1
    Name: "a person"
  }

  json.NewEncoder(w).Encode(p)
}

func main() {
  http.HandleFunc("/json", ReturnJson)
}
```

- First, we set the content type, by setting the value “application/json”:

```
w.Header().Set("Content-Type", "application/json")
```

- Secondly, we construct the data we are about to send out:

```
p := Person {
  Id: 1
  Name: "a person"
}
```

- Finally, we encode the data as JSON and write it to the response stream:

```
json.NewEncoder(w).Encode(p)
```

It’s also possible to use the `Marshal()` function like so, instead of `json.NewEncoder()`:

```
data, err := json.Marshal(p)
w.Write(data)
```

Working with the request

There are various ways, additionally to headers, to instruct the server program what to do:

- **HTTP verb**, the HTTP verb expresses intention. The POST verb means to create a resource and the GET verb says to only read the data for example. There are many HTTP verbs that we will cover later in this chapter. These two below requests mean different things:

```
    GET /products # fetching a list of products
    POST /products # creating a new product resource
```

- **body**, The body can contain a payload, data we can use to create or update a resource usually. Here's an example:

```
    {
      "name" : "a new product"
    }
```

- **router parameters**. As part of a route request, you can have parameters that carry meaning. If the client asks for the route `/products/5` then the `5` can mean the calling client is after a specific product whose unique identifier is `5`.
- **query parameter**. At the end of the route, there can be a query section. That section can give further instruction to the request to for example reduce the size of the returning data. Does the query part start with a question mark? and is followed by key-value pairs separated by ampersands, &. It can look like so: `/products?page=1&pageSize=20`

Parsing a body

The request has a `Body` property. Depending on what's in the body, you might need to decode it. Below code is decoding a piece of JSON and writing it to the response stream:

```
package main

import (
  "fmt"
  "encoding/json"
)

type Person struct {
  Id int
  Name string
}

func handleRequest(w http.ResponseWriter, r *http.Request) {
  var p Person

  json.NewDecoder(r.Body).Decode(&p)
  // save person to storage

  fmt.Fprintf(w)
}

func main() {
  mux := http.NewServeMux()
  mux.HandleFunc("/person/", handleRequest)

  err := http.ListenAndServe(":4000", mux)
}
```

Read a route parameter

There's no built-in way to access a route parameter so you would have to parse it like so:

```
tokens := strings.Split(r.URL.Path, "/")
// check each part
```

or use for example a regular expression to parse out the parts.

Another choice is using a library like the following:

- httprouter[8]
- Gorilla Mux[9]

Here's an example using `httprouter`:

[8] https://github.com/julienschmidt/httprouter
[9] http://www.gorillatoolkit.org/pkg/mux

```
func Hello(w http.ResponseWriter, r *http.Request, ps httprouter.Params) {
    fmt.Fprintf(w, "The id us, %s!\n", ps.ByName("id"))
}

func main() {
    router := httprouter.New()
    router.GET("/products/:id", Hello)

    log.Fatal(http.ListenAndServe(":8080", router))
}
```

Read a query parameter

The query part of the route is accessible via the `Query()` function on the `URL` property of the request instance:

```
r.URL // /products?page=3&pageSize=20
r.URL.Query() // ?page=3&pageSize=20
```

To access a specific parameter know that the `Query()` function returns a `Values` map.

```
r.URL.Query()["page"] // 3
```

It's possible to call the `Get()` function as well, but only if there is only one parameter:

```
r.URL.Query().Get("page")
```

HTTP method

The method means different things and should be handled differently. To access the request method, there's a `Method` property on the request, `r`.

```
r.Method
```

There's also defined constant like `MethodGet`, `MethodPost` on `http`, so you could write code like so:

```
func handleRoute(w http.ResponseWriter, r *http.Request) {
  if r.Method == http.MethodGet {
    fmt.Println("It's a GET request")
  }
}
```

ServeMux, a better way

So far, you've created an HTTP server by calling `ListenAndServe()` with a port argument and nil. But there's another way to do it. You could be using something called servemux. A servemux is also known as a router. Much like using the `http` directly to add routes, you instead add those routes on the servemux. Let's show some code:

```
mux := http.NewServeMux()
mux.Handle("/hello", handleHello)
http.ListenAndServe(":8090", mux)
```

In the preceding code, you instantiate the servemux by calling `NewServeMux()`. Then, you set up a route and its handler by calling `Handle()`. Finally, you call `ListenAndServe()` but this time around you pass the `mux` instance instead of `nil`.

So how is this better than the other way we've used so far? The first way we learned about, uses a `DefaultServeMux` and risks exposing profiling endpoints, which is bad. Another reason is connecting the routes directly to `http` changes the global state, which is looked down upon in Go generally.

Assignment - build a first web app

Your web app should have at least one route. The said route should write to the response stream. The web app should start at a specific port, for example, 5500.

Solution

```
package main

import (
  "fmt"
  "net/http"
)

func handleRequest(w http.ResponseWrite, r *http.Request) {
  fmt.Fprintf(w, "Hi there")
}

func main() {
  http.HandleFunc("/hello", handleRequest)
  http.ListenAndServe(":8090", nil)
}
```

Challenge

- list details on the request such as the route, the verb used and the query parameters.
- See if you can serve up different types of data like JSON or images.

Better logging with a logging library

Once you start writing code, you realize quite early that you need to print things to the screen as well as sometimes to a file or even a log service. What you want to say is usually what type of logging you want to do.

Introduction

In this chapter, you will learn the following:

- Why and what to log.
- Using the `log` library.

Reasons to log

There are many reasons to log, here are some reasons:

- **Information**, there might be a case where you want to provide some type of information that could be of use to the one using the program.
- **Success**. A success message is a little more than just information, it indicates that you succeeded with something.
- **Warning**. When you have a warning, something happened that you should be aware of. It's usually not serious enough to shut down the app but it should make you vigilant, it could be that memory is running low for example.
- **Error**. When you get an error, you tend to end up in a state where it's no longer a wise choice to continue.
- **Performance**. It's common to measure how long something takes, for the sake of improving things this information can be useful.
- **Other**. There are also other reasons why you would log something, usually, that's connected to your business.

What to log

The general rule is the more you can log the better. Especially if it's an error you want to fix you might want to log things like:

- When it happened
- What happened
- Specific error info

For every case, you want to log, have a log at how the log message will be used, will a team be logging through these logs, and what would help them. See if you can interview someone on that team.

Using `log`

In general, you want to log in places where things might go wrong such as when you make web requests, work with I/O and so on.

In general, use these as guidelines for when to log:

- **Faulty input**. If the program risks producing a faulty response, there was a problem converting/casting a number or it received an unexpected input for example.
- **Error state**. If the program ends up in a state from which it can't recover, for example, unable to fetch a batch of data from a data source.

You don't want logs on every single line of code.

Standard log `Println()`

To produce a standard log message, you can use the `Println()` function in the `log` package. It takes a string and will produce a log message that combines a date, time, and your error message.

Here's some code using `Println()`:

```
1  package main
2
3  import (
4   "fmt"
5   "log"
6  )
7
8  func main() {
9    log.Println("log message")
10 }
```

It will produce an output like so:

```
2022/03/24 12:42:13 log message
```

Use `Fatal()` for errors

`Println()` produces a normal looking log message with a date, time and message. `Fatal()` is used when you want to end the program. What `Fatal()` does is to print out the message you give it and call `os.Exit(1)`.

Here's how it can be used:

```
log.Fatal("quit program due to <specify reason>")
```

Log to a file

If you develop an app, you are likely to run it and keep an eye on the console for what the app prints out.

However, as your app becomes ready for production, you want to make sure that all logs that can be useful to analyse is kept somewhere, either sent to a log service or stored in a file.

That way, you ensure that you can analyze these logs later to understand where things went wrong or if you want to analyze the performance of your program.

To log into a file, you can use the `SetOutput()` function. It takes a file handler as input. Thereby, you can use these three lines of code to log:

```
f, err := os.OpenFile("testlogfile", os.O_RDWR|os.O_CREATE|os.O_APPEND, 0666)

defer f.Close()

log.SetOutput(f)
```

- In the first line, you open up a file "testlogfile" and ensure you can append it to it.

```
f, err := os.OpenFile("testlogfile", os.O_RDWR|os.O_CREATE|os.O_APPEND, 0666)
```

- In the second line, you ensure the file is closed the last thing that happens in the program.

```
defer f.close()
```

- Finally you call `SetOutput()` which ensures all log message are sent to file "testlogfile", and not shown in the console.

```
log.SetOutput(f)
```

Assignment

In this assignment, you will add the `log` library to your code.

1. Create a file *records.csv* with the following content:

```
item,quantity

112, 2
94, 3
```

2. Create a file *app.go* and give it the following content:

```
package main

import (
 "fmt"
 "io/ioutil"
 "os"
)

func ProcessFile(path string) {
 filebuffer, err := ioutil.ReadFile(path)
 if err != nil {
  fmt.Println("Error: ", err)
  os.Exit(1)
 }
 inputdata := string(filebuffer)
 fmt.Println("Do something with input: \n", inputdata)
}

func main() {
 fileName := "records.csv"

 fmt.Printf("processing file '%s' \n", fileName)
 ProcessFile(fileName)
}
```

3. Run the file with `go run`:

```
go run app.go
```

You should see the following output:

```
processing file 'records.csv'

Do something with input:
item,quantity
112, 2
94, 3
```

4. Add the `log` package to the import and replace all calls to `fmt` with `log`, like so:

```
 package main

 import (
  "io/ioutil"
  "log"
 )

 func ProcessFile(path string) {
  filebuffer, err := ioutil.ReadFile(path)
  if err != nil {
   log.Fatal("Error: ", err)
  }
  inputdata := string(filebuffer)
  log.Print("Do something with input: \n", inputdata)
 }

 func main() {
  fileName := "records.csv"

  log.Printf("processing file '%s' \n", fileName)
  ProcessFile(fileName)
 }
```

Let's see how the output differs.

5. Run the program with `go run`:

```
go run main.go
```

your output should be similar to:

```
2022/03/28 13:57:57 processing file 'records.csv'

2022/03/28 13:57:57 Do something with input:
item,quantity
112, 2
94, 3
```

6. Next, lets change the name of `fileName` to "record.csv", to trigger an error (there's no such file).

```
fileName := "record.csv"
```

7. Now, run the app `go run`:

```
go run app.go
```

You should see a similar output:

```
2022/03/28 14:04:52 processing file 'record.csv'

2022/03/28 14:04:52 Error: open record.csv: no such file or directory
exit status 1
```

This time around though you see the program exciting with exit status 1.

The conclusion is that it's better to rely on the `log` library because you get dates and times, and you type less. But there's more, we can log to a file, let's see how we do that next.

Log to a file

Someone examining the output of the program is likely to inspect a log file overlooking the terminal. Let's instruct `log` to log to a file instead.

1. At the start of the `main()` function, add the following code:

```
  logFile := "logfile"

  f, err := os.OpenFile(logFile, os.O_RDWR|os.O_CREATE|os.O_APPEND, 0666)

  if err != nil {
   log.Fatal("Could not log to file: ", logFile)
  }
  defer f.Close()
```

Now you have instructed `log` to write all entries to the file *logfile*.

2. Run the program again, `go run`:

```
go run main.go
```

You should now see the following output:

```
exit status 1
```

All your log entries have moved to *logfile*, let's see what it looks like:

```
2022/03/28 14:11:24 processing file 'record.csv'

2022/03/28 14:11:24 Error: open record.csv: no such file or directory
```

Great, now we have all entries in a central place, which should make it easier for us to analyze.

Solution

```
package main

import (
 "io/ioutil"
 "log"
 "os"
)

func ProcessFile(path string) {
 filebuffer, err := ioutil.ReadFile(path)
 if err != nil {
  log.Fatal("Error: ", err)
 }
 inputdata := string(filebuffer)
 log.Print("Do something with input: \n", inputdata)
}

func main() {
 fileName := "record.csv"
 logFile := "logfile"

 f, err := os.OpenFile(logFile, os.O_RDWR|os.O_CREATE|os.O_APPEND, 0666)

 if err != nil {
  log.Fatal("Could not log to file: ", logFile)
 }
 defer f.Close()
```

```
28
29  log.SetOutput(f)
30
31  log.Printf("processing file '%s' \n", fileName)
32  ProcessFile(fileName)
33 }
```

Working with strings

There are many reasons why we need to work with strings in different ways. Below are some situations that you are likely to encounter and that the `strings` library has a solution for:

- **user input**, or other types of storage may contain special characters that you need to cater for.
- **inspection**, does the string contain what we need for our business logic?
- **parsing**, splitting the file until we get what we need, could be things like a number or date for example.
- **presentation**, sometimes we need to present the text in certain way in a UI for example, to highlight said information.

Handling special characters

Lets say we read user input and we want to interpret what we get as a number. To make our program robust, we're ok with the user typing spaces or newline characters. The following input should be allowed:

```
114
   114
114\n
```

There's three methods of interest to handle such a case for us, namely `Trim()`, `TrimLeft()` and `TrimRight()`.

- `Trim()`, what it does is remove whitespace characters from both left and right side.
- `TrimLeft()`. It removes whitespace from the left side only, and if you specify special characters as well.
- `TrimRight()`. It removes whitespace from the right side only, and if you specify special characters as well.

All these functions above have as their second parameter a so called *cutset* parameter, where you specify what character you want to get rid of. You can for example specify to remove space, newline and tab characters like so:

```
" \n\t"
```

Here's some code that shows all three methods in use:

```
fmt.Printf("%s , string length %d \n", s, len(s))
 res := strings.Trim(s, " ")
 fmt.Printf("%s , string length %d \n", res, len(res))

 s2 := "   114  "
 fmt.Printf("%s , string length %d \n", s2, len(s2))
 res = strings.TrimLeft(s2, " ")
 fmt.Printf("%s , string length %d \n", res, len(res))

 s3 := "   114  "
 fmt.Printf("%s , string length %d \n", s3, len(s3))
 res = strings.TrimRight(s3, " ")
 fmt.Printf("%s , string length %d \n", res, len(res))
```

The above string has three whitespaces to the left and two to the right, giving it a total length of 8.

The output of the above code is:

```
  114   , string length 8
114 , string length 3
   114   , string length 8
114   , string length 5
   114   , string length 8
   114 , string length 6
```

Lets break down the output per row.

For this output, using `Trim()`, the spaces are removed on both sides and we end up with something looking left aligned:

```
"114"
```

The next output, using `TrimLeft()`, shows how the right spaces are still there:

```
"114   "
```

Our final row, using `TrimRight()`, shows a right alignment and how the spaces on the left side still remains:

```
"   114"
```

Inspect with `Contains()`

Imagine you want to inspect a string to verify whether it contains a certain substring.

For that, you can use the `Contains()` function. Its syntax looks like so:

```
strings.Contains(stringSource, pattern)
```

You can then for example use it to process a list from a point of sale system, and for each item check if it contains a certain prefix:

```
rows := []string{"order: 5", "order: 10", "order: 5", "separator"}


for item :=  range rows {
  if strings.Contains("order") {
    // process order
  }
  // ignore
}
```

Parsing with `Split()`

Lets continue with processing rows from our point of sale system. This time, we will be looking at a specific item and extract the information we need. For this, we will use the `Split()` function:

```
rows := []string{"order: 5", "order: 10", "order: 5", "separator"}

for item :=  range rows {
  if strings.Contains("order") {
     tokens := strings.Split(item, ":") // [ "order", " 5"]
     value := strings.Trim(tokens[1])
     fmt.Println(value)
  }
  // ignore
}
```

By using this code:

```
strings.Split(item, ":")
```

on this string “order: 5”, we end up with an array `["order", "5"]` and `strings.Trim(tokens[1])` would then refer to 5.

> TIP: If we need to treat the 5 above as a number, as part of calculation, we would need to convert it to a number first

Presentation

Say you have customer management system and there’s a lot of data to present. To give importance to certain data, we can use functions to highlight their visual appearance.

Take the following multiline customer string:

```
Jean Normand
123 Way
Washington
```

If you use `ToUpper()` on city you get a result like so:

```
Jean Normand
123 Way
WASHINGTON
```

With `ToLower()` you ensure all characters are formatted as lowercase.

Assignment

Write a program that given a struct containing, name, address and city ensures that the name is lowercase and the address is uppercase.

Solution

```
package main

import (
 "fmt"
 "strings"
)

type Person struct {
 Name    string
 Address string
 City    string
}

func main() {
 person := Person{Name: "jean Normand", Address: "123 Way", City: "Washington"}

 fmt.Println(strings.ToUpper(person.Name))
 fmt.Println(person.Address)
 fmt.Println(strings.ToUpper(person.City))
}
```

Use regular expressions to parse text

In this chapter, you'll learn how to use Regular Expressions to search and replace text.

What is RegEx and what to use it for

Regular Expressions or RegEx for short, is used for searching and replacing text. Technically a RegEx is a sequence of characters that specifies a search pattern.

Why use RegEx

RegEx primary usage area is for searching text, replacing it and also extracting text. There do exist string libraries that can do some of the functionality RegEx is capable of. Sometimes using those string libraries might even be the best thing to do. However, sometimes a RegEx pattern is better.

> Fair word of warning though, RegEx is hard to get right. You are encouraged to learn more of how they work cause they are quite powerful.

My hope is by you reading this chapter, that you will find RegEx less intimidating and see it as a valuable tool in your toolbox.

Where is it used ?

RegEx shows up in many different contexts:

- **Text editors, any programs with a search**. In most text editors, for example Visual Studio Code, you can search for files and inside of files with a RegEx search pattern.
- **Code**, many programming languages and runtimes have libraries that helps you use RegEx.

Your first RegEx

Let's construct a simple RegEx to get a feel for it. Here it is:

```
an
```

if you apply this search pattern `an` to the following text:

```
highlands is a part of Scotland
```

It will match like so:

highl**an**ds is a part of Scotl**an**d

For simpler cases, where you are looking to see if a specific word matches, in one or more places in a sentence, a pattern like the above is enough.

RegEx in Go

To start using RegEx in Go, there's the regexp library. There are two approaches:

- `regexp` directly, here's an example:

```
matched, err := regexp.FindString("an", "highlands is a part of Scotland")
```

Here, you get a boolean back that returns true if there's a match.

- **compiled**, in this way, you compile a regular expression and then calls a method on it, like so:

```
r, _ := regexp.Compile("an")
matches := r.FindAllString("highlands is a part of Scotland", -1)
```

The above returns a string array with all the matches, in this case `["an", "an"]`.

In this version, you have more functions available.

Character classes

Character classes are able to distinguish between different types of characters. Different types can be newlines, digits, letters and so on.

Let's have a look at some common types you are likely to encounter:

Type	Description
.	This type matches any character except for a carriage return
\	This type escapes what's coming next
\w	matches any character from the latin alphabet including underscore _
\d	matches any digit
D	this is the inverse of \d and matches any character that's not a digit
\s	matches a white space character like space tab, line feed etc.

Lets show an example:

```
matched, err := regexp.FindString("\d", "abc123")
```

There would be a match above due to 123. However, there would be no match against "abc" as there's no digits in it.

Repetition

If you want to express repetition, there's two characters of interest:

- +, matches 1 to many characters.

```
\w+
```

Given the string "aaaa bab" it would match:

aaaaab bab as the above describes matching characters but not the white space.

```
r, _ := regexp.Compile("\\w+")
matches := r.FindAllString("aaaa bab", -1)
```

Note the extra \, we need that because of the way we construct our Regex.

- *, matches 0 to many characters. Lets say you want to match a postal address that starts with "PA" and may contain 0 or many numbers. It should then match strings:

```
PA
PA111
```

We can use a * to construct this looking like so:

```
regexp.MatchString("PA*", "PA")
regexp.MatchString("PA*", "PA111")
```

- ?, also known as a greedy or optional quantifier. It looks backwards and makes it optional and takes it, if it can. Consider this case:

```
http
https
```

If you want to match them both, you can type:

```
https?
```

Another example is:

```
r, _ := regexp.Compile("an.")
matches := r.FindAllString("and ant an", -1)
```

The above will only match **and** and **ant** but not *an*. If we modify the regex to `an.?` it will match **and ant an**.

Anchors and boundaries

There are different anchors you can use like for example:

- `^`, beginning of the string. The following states that the string needs to begin with the following string "INV" to signify the start of an invoice row:

```
^INV
```

- `$`, end of the string. An example could be matching a string ends with a certain domain ".com":

```
\.com$
```

Groups

Groups are way to capture part of a string and have that returned. It's very useful for parsing out the info you need. Consider this example parsing out the info from a CSV row:

```
Name: myarticle, Price: 114, Quantity: 3
```

To get the data you need, you want everything after the colon, :. You can construct a RegEx like so:

```
\w+:\s?(\w+)
```

what we are doing is defining we want to capture a group using parenthesis `()` but that group should happen after:

- a number of letters, `\w+`
- followed by a colon, `:`
- followed by 0 or 1 space `\s?`
- then our group `(\w+)`, one ore more letters

All this ends up capturing myarticle, 114 and 3.

Named groups

A named group is a group you want to capture where the groups have names. Why would you want that? Well, say that you want to break down a URL in pieces and wants to know what's what. Given a URL "http://myapi.com/products?page=1", you have:

- `http`, the protocol.
- `myapi.com`, the domain.
- `/products`, is the route.
- `?page=1`, is the query parameters.

So how can we break it apart and give it a name?

Well, to break it apart, we will use something called named groups, it will allow us to look at our matches and know what's what. So instead of getting:

```
1 http
```

We will get a key and value that says:

```
1 protocol: http
```

Syntax wise, we need to use `?<name of our group>` within our parenthesis ().

You use the following syntax:

```
1 (?<mygroup>\w+)
```

In Go, we need a `P` right after the question mark, so the code for this would be:

```
1 r, err := regexp.Compile(`(?P<mygroup>\w+):`)
```

Extract the data from a URL

Let's approach this problem then given the string "http://myapi.com/products?page=1":

- matching the protocol:

```
1     ^(?<protocol>\w+):
```

- domain, to match the domain as well, we're looking to capture everything after http:// and until the next /:

```
^(?<protocol>\w+):\/\/(?<domain>\w+\.\w+)\/?
```

- route, ok so we've matched up "http://mydomain.com" so far, now lets match the route, i.e what happens after the / but before any questions marks, ?
- query params

Here's what our Go code would look like:

```
r, err := regexp.Compile(`^(?P<protocol>\w+):\/\/(?P<domain>\w+\.\w+)\/(?P<route>\w+\
)\/?`)
```

Ok, so we have the pattern, what about printing the parsed parts?

To pair the named groups with their values, we need to combine values from both the Regex and the response. First, we call `FindStringSubmatch()`, that will give us the values.

```
m := r.FindStringSubmatch("http://myapi.com/products")
```

Then, we need to match the names with these values. We will need to call `r.SubexpNames()` and iterate over the response.

```
result := make(map[string]string)
 for i, name := range r.SubexpNames() {
  if i != 0 && name != "" {
   result[name] = m[i]
  }
 }
```

Note this line where each name is assigned a value:

```
result[name] = m[i]
```

Finally, to get the values, we can print them out as they are now in a map structure:

```
fmt.Println(result["protocol"]) // http
fmt.Println(result["domain"]) // myapi.com
fmt.Println(result["route"]) // products
```

Assignment - create a Go program that parses a URL

From the above use case on named groups, write a Go program that takes a URL and analyzes it. It should work like so:

```
Type URL: http://myapi.com/products
The URL consist of:
protocol: http
domain: myapi.com
route: products
```

Solution

```
package main

import (
 "fmt"
 "log"
 "regexp"
)

func main() {
 var url string
 fmt.Println("Type URL: ")
 fmt.Scan(&url)

 r, err := regexp.Compile(`^(?P<protocol>\w+):\/\/(?P<domain>\w+\.\w+)\/(?P<route>\w\
+)\/?`)
 if err != nil {
  log.Fatal("Error compiling: ", err)
 }
 m := r.FindStringSubmatch(url)
 if m == nil {
  panic("mo match")
 }
 result := make(map[string]string)
 for i, name := range r.SubexpNames() {
  if i != 0 && name != "" {
   result[name] = m[i]
  }
 }
 fmt.Println("The URL consist of:")
 fmt.Println(result["protocol"])
 fmt.Println(result["domain"])
 fmt.Println(result["route"])
}
```

Replacing

A common use case for Regex is when it's used to replace something with something else.

There's more than one method in Go you could be using but one you could use is `ReplaceAllString()` that sits on the compiled RegEx object:

```
r := regexp.MustCompile(`aa`)
 s := r.ReplaceAllString("aabbcc", "cc") // s = ccbbcc
```

The above replaces all occurrences of `aa` with `cc` on the string `aabbcc`.

You can also use capture groups and replace a captured group with a string. Here's an example:

```
r := regexp.MustCompile(`(\d)`)
 s := r.ReplaceAllString("productid:114", "0${1}") // s = productid:0114
```

in the above case, we replace 114 with itself but we also prepend it with a 0.

Use case, replace XML Nodes

Imagine you are working with XML for example and want to rename all nodes with a certain name.

Here's your XML

```
<books>
    <book>
      <author>Shakespeare</author>
      <title>Romeo and Juliet</title>
      <pages>400</pages>
      <type>paperback</type>
      <cost>17</cost>
    </book>
    <book>
      <author>Shakespeare</author>
      <title>Hamlet</title>
      <pages>270</pages>
      <type>paperback</type>
      <cost>15</cost>
    </book>
</books>
```

Imagine `title` should be replaced by `name`, how do we do that?

Well, it would be straight forward to replace title by name. Let's say we have this file content though:

```
<books>
    <book>
      <author>Shakespeare</author>
      <title>The title is Romeo and Juliet</title>
      <pages>400</pages>
      <type>paperback</type>
      <cost>17</cost>
    </book>

</books>
```

Then we would not only rename the element `title` to `name` but also the content would be replaced o "The title is Romeo and Juliet", that's NOT what we want.

We need to restrict the replace operation to only target element, like so:

```
\<\/?(title)\>
```

The above would match for example `<title>` and `</title>`. If we try this however on this XML, we almost get what we want:

```
<author>Shakespeare</author>
```

becomes

```
nameShakespearename
```

What happened, why did we loose `<>` ? We need a way to express keeping what was there before AND replace the name. A way to do that is to express capture groups on `<>` and the element name, like so:

```
(\<\/?)(title)(\>)
```

Now we have three groups, we need to fit the result together, and this is something we can express like so:

```
${1}name${3}
```

- `${1}` corresponds to capture group matching `<` or `</`
- `name` is the string we replace `title` with.
- `{3}` corresponds to capture group matching `>`.

Assignment - replace content

Take the file *books.xml* containing:

```
<books>
    <book>
      <author>Shakespeare</author>
      <title>Romeo and Juliet</title>
      <pages>400</pages>
      <type>paperback</type>
      <cost>17</cost>
    </book>
    <book>
      <author>Shakespeare</author>
      <title>Hamlet</title>
      <pages>270</pages>
      <type>paperback</type>
      <cost>15</cost>
    </book>
</books>
```

and replace:

- author with name
- cost with price

TIP: you might need to apply the replace twice.

Solution II

```
package main

import (
 "fmt"
 "regexp"
)

func main() {
 file := `<books>
    <book>
      <author>Shakespeare</author>
      <title>Romeo and Juliet</title>
      <pages>400</pages>
      <type>paperback</type>
      <cost>17</cost>
```

```
    </book>
    <book>
      <author>Shakespeare</author>
      <title>Hamlet</title>
      <pages>270</pages>
      <type>paperback</type>
      <cost>15</cost>
    </book>
</books>`

 r := regexp.MustCompile(`(\<\/?)(title)(\>)`)
 s := r.ReplaceAllString(file, "${1}name${3}")
 fmt.Println(s)

 r = regexp.MustCompile(`(\<\/?)(cost)(\>)`)
 s = r.ReplaceAllString(s, "${1}price${3}")
 fmt.Println(s)
}
```

Goroutines and channels

A goroutine is a lightweight thread managed by the Go runtime.

Channels is how you communicate between routines.

Introduction

In this chapter you will:

- Understand the difference between concurrency and parallelism
- Use Goroutines to run your functions
- Create and use channels to communicate between your Goroutines
- Apply Goroutines to an app that searches files for faster execution.

Concurrency, what's the benefit

Concurrency is the task of running and managing the multiple computations at the same time. While *parallelism* is the task of running multiple computations simultaneously.

So what are some benefits:

- **Faster processing**. The benefit is getting tasks done faster. Imagine that you are searching a computer for files, or processing data, if it's possible to work on these workloads in parallel, you end up getting the response back faster.
- **Responsive apps** Another benefit is getting more responsive apps. If you have an app with a UI, imagine it would be great if you can perform some background work without interrupting the responsiveness of the UI.

Goroutines

A goroutine is a lightweight thread managed by the Go runtime. What you do is to add the keyword `go` in front of a function. Here's an example:

```
1  go myFunction()
```

Imagine the following code running, what would happen?

```
package main

import "fmt"

func myFunction() {
 for i := 0; i < 3; i++ {
  fmt.Println("my function: ", i)
 }
}
func anotherFunction() {
 for i := 4; i <7; i++ {
  fmt.Println("another function: ", i)
 }
}

func main() {
 go myFunction()
 anotherFunction()
}
```

It would only print the result from `anotherFunction()` as it takes a short while for the go routine to start up. You can have the go routine execute as well by adding a little delay, like so:

```
func main() {
 go myFunction()
 anotherFunction()
 time.Sleep(1 * time.Second)
}
```

The result is now the following:

```
another function:  4
another function:  5
another function:  6
my function:  0
my function:  1
my function:  2
```

The function with the go routine finishes last. Lets modify the code slightly and have the two functions use a delay, so we simulate workloads taking different time to finish:

```
func myFunction() {
 time.Sleep(1500 * time.Millisecond)
 for i := 0; i < 3; i++ {
  fmt.Println("my function: ", i)
 }
}
func anotherFunction() {
 time.Sleep(500 * time.Millisecond)
 for i := 4; i < 7; i++ {
  fmt.Println("another function: ", i)
 }
}

func main() {
 go myFunction()
 go anotherFunction()
 time.Sleep(2 * time.Second)
}
```

at this point, `anotherFunction()` finishes first as it has the shortest delay, which is to be expected. Here's what the output looks like now:

```
another function:  4
another function:  5
another function:  6
my function:  0
my function:  1
my function:  2
```

Use case - a file search

Imagine you have case where you need to find a file on disk. If you write a function like so, it will search a directory and report back the result if the file is found:

```
func SearchFiles(dir string, lookFor string) string {
 log.Println("[SEARCHING] ", dir)
 files, err := ioutil.ReadDir(dir)
 if err != nil {
  log.Fatal(err)
 }

 for _, file := range files {
  log.Println(dir+file.Name(), file.IsDir())
  if file.Name() == lookFor {
   return "[FOUND] " + filepath.Join(dir, file.Name())
  }
 }
 return "[NOT FOUND] " + dir
}
```

Imagine you now run this code like so, to search many directories:

```
result := make([]string, 0)
append(result, SearchFile("./tmp", "myfile.txt"))
append(result, SearchFile("./tmp2", "myfile.txt"))
append(result, SearchFile("./tmp3", "myfile.txt"))
append(result, SearchFile("./tmp4", "myfile.txt"))

for i := 0 i< len(result); i++ {
  fmt.Println(result[i])
}
```

If found, you will get an output similar to the below, depending on whether *myfile.txt* is found in any of the searched directories:

```
[FOUND] ./tmp/myfile.txt
[NOT FOUND] ./tmp2/myfile.txt
[NOT FOUND] ./tmp3/myfile.txt
[NOT FOUND] ./tmp4/myfile.txt
```

Now to speed up this process, it would be great if you are able to search many directories at once, so you could type something like so:

```
go SearchFile("./tmp", "myfile.txt")
go SearchFile("./tmp2", "myfile.txt")
go SearchFile("./tmp3", "myfile.txt")
go SearchFile("./tmp4", "myfile.txt")
```

This works, it now searches all directories, in parallel. However, now we don't have a way to get the response back as we can't write like so:

```
result := make([]string, 0)
go append(result,SearchFile("./tmp", "myfile.txt")) // won't compile, says "go disca\
rds results"
```

So how can we get the result from a go routine, the answer is by using channels, so lets discuss those next.

Channels

A channel is how we can communicate cross go routines but also between go routines and the part of our code not using a go routine.

The idea is to send a value to a channel, and have part of our code listen to values from a channel.

Creating a channel

To create a channel, you need the keyword `chan` and the data type of the messages you are about to send into it. Here's an example:

```
ch := make(chan int)
```

In the above example, a channel `ch` will be created that accepts messages of type `int`.

Sending a value to a channel

To send to a channel, you need to use this operator `<-`, it look like a left pointing arrow and is meant to be read as the direction something is sent. Here's an example of sending a message to a channel:

```
ch <- 2
```

In the above code, the number 2 is sent into the channel `ch`.

Listening to a channel

To listen to a channel, you again use the arrow `<-`, but this time you need a receiving variable on the left side and the channel on the right side, like so:

```
value := <- ch
```

Matching sending and receiving

Let's say you have the following code:

```
package main

import "fmt"

func produceResults(ch chan int) {
 ch <- 1
 ch <- 2
}

func main() {
 ch := make(chan int)
 go produceResults(ch)

 var result int
 result = <-ch
 fmt.Println(result)
 result = <-ch
 fmt.Println(result)
}
```

You are invoking `produceResults()` and it sends messages to the channel twice:

```
ch <- 1
ch <- 2
```

in `main()`, you receive the results:

```
var result int
result = <-ch
fmt.Println(result)
result = <-ch
fmt.Println(result)
```

So what happens if you produce more values than you receive like so?

```
ch <- 1
ch <- 2
ch <- 3
```

answer: you will miss out on the extra value.

What if it's the opposite, you try to receive one more value than you actually get?

```
var result int
result = <-ch
fmt.Println(result)
result = <-ch
fmt.Println(result)
result = <-ch
fmt.Println(result)
```

At this point, your code will deadlock, like so: **fatal error: all goroutines are asleep - deadlock!**. Your code will never finish as that value will never arrive.

The lesson here is that you need to keep track of how many results you might get and only try to receive that many.

There's another way to receive values, and that's by using a `select` like so:

```
for i := 0; i < 2; i++ {
  select {
  case x, ok := <-ch:
   if ok {
    fmt.Println(x)
   }
  }
 }
```

The idea is to *match* the receiving of a value like so:

```
case x, ok := <-ch:
```

What you are getting is two things, the value itself `x` and `bool` we name `ok`. If we managed to get a value ok, then `ok` holds the value `true`. What happens if it's not ok then? It would be `false` if the channel is closed and can no longer produce any more values, so lets discuss that next.

Closing a channel

A channel is open until you close it. You can actively close it by calling `close()` with the channel as an input parameter:

```
close(ch)
```

However, when we close a channel, we need to test for it. If we attempt to receive a value from a closed channel, it will cause a crash. To test whether the channel is open or not, we can use the `select` we just wrote:

```
  select {
  case x, ok := <-ch:
   if ok {
    fmt.Println(x)
   } else {
     break // channel is closed
   }
  }
```

The value of `ok` is now false.

To apply the concept of closing a channel, we add `close()` to `produceResults()` and we have our for loop run one more time than there's values, like so:

```
package main

import (
 "fmt"
)

func produceResults(ch chan int) {
 ch <- 1
 ch <- 2
 // ch <- 3
 close(ch)
}

func main() {
 ch := make(chan int)
 go produceResults(ch)
 // time.Sleep(1 * time.Second)

 for i := 0; i < 3; i++ {
  select {
  case x, ok := <-ch:
   if ok {
    fmt.Println(x)
```

```
    } else {
     fmt.Println("channel closed")
    }
   }
  }
}
```

The output of running said code is:

```
1
2
channel closed
```

We can see how the `else` clause is matched on the third iteration.

Now, we might have more long running tasks, at which point we need to sit and wait until the channel tells us it closed. Here's code to handle that:

```
label:
 for {
  select {
  case x, ok := <-ch:
   if ok {
    fmt.Println(x)
   } else {
    fmt.Println("channel closed")
    break label
   }
  }
 }
```

What's happening here is that we set up a for loop that runs forever, until closed. To ensure we break out of the for loop and not just the `select`, we add `label:`

TODO, you can use range over the channel as well.

Assignment - `SearchFiles()` with channels

Let's take all our learning and add channels to the program we wrote containing a file searcher.

Challenge

Solution

```
package main

import (
 "io/ioutil"
 "log"
 "path/filepath"
)

func SearchFiles(dir string, lookFor string, ch chan string) {
 log.Println("[SEARCHING] ", dir)
 files, err := ioutil.ReadDir(dir)
 if err != nil {
  log.Fatal(err)
 }

 for _, file := range files {
  log.Println(dir+file.Name(), file.IsDir())
  if file.Name() == lookFor {
   ch <- "[FOUND] " + filepath.Join(dir, file.Name())
   return
  }
 }
 ch <- "[NOT FOUND] " + dir
}

func main() {
 ch := make(chan string)

 go SearchFiles("./test/", "test2.txt", ch)
 go SearchFiles("./other/", "test2.txt", ch)

 var res = ""
 for i := 0; i < 2; i++ {
  res = <-ch
  log.Println(res)
 }
}
```

Working with a database

You will be working with a sqlite database and read and write values to it.

Introduction

In this chapter you will:

- Create a database and its structure.
- Write to the database.
- Read from the database.

Select a sqlite driver

To connect with a sqlite database we've got a few libraries to choose from. These libraries will provide you with a sqlite driver that you need to successfully connect with your database. Here are some common ones:

- SQLite (uses cgo): https://github.com/mattn/go-sqlite3 [*]
- SQLite (uses cgo): https://github.com/gwenn/gosqlite - Supports SQLite dynamic data typing
- SQLite (uses cgo): https://github.com/mxk/go-sqlite
- SQLite: (uses cgo): https://github.com/rsc/sqlite
- SQLite: (pure go): https://modernc.org/sqlite

Refer to this link to see libraries for other databases:

- https://github.com/golang/go/wiki/SQLDrivers

.

Use `sqlite3` from the console

To work with your database, it's beneficial to use sqlite from the command line. Consult the official downloads page[10] for sqlite and ensure you pick the executable for your operating system. Installing sqlite will give you an executable.

With the executable, you can:

- Create a database.
- Run SQL commands.
- Run other commands supported by the executable.

Create a database

To create a database, you give it the name of the database like so:

```
sqlite3 activities.db
```

The preceding command will give you a database in a file called "activities.db"

> It will also start up a shell where you can execute SQL commands as well as commands supported by sqlite3.

Run SQL commands

Run a SQL command in the shell by typing SQL and end it by a semicolon, `;`.

```
CREATE TABLE `person` (
        `uid` INTEGER PRIMARY KEY AUTOINCREMENT,
        `name` VARCHAR(64) NULL,
        `lastname` VARCHAR(64) NULL,
        `created` DATE NULL
    );
```

Exit the shell

After you're done, you can exit the shell by typing `.exit`:

[10] https://www.sqlite.org/download.html

```
.exit
```

Talking to your database via Go.

To talk to your database via Go, there's some steps you need to take in order:

1. **Create a project**. You need to create a project so you can import a Go package containing your sqlite driver. Create a project by running `go mod init`. Below is an example:

```
go mod init "example-project"
```

2. **Add imports**. Once you have the needed packages you need to refer to them in the import section:

```
import (
  "database/sql"
  _ "github.com/mattn/go-sqlite3"
 )
```

   Above, are the two packages we will use, "database/sql" that provides an interface for us to run queries and statements. "github.com/mattn/go-sqlite3" contains the driver that will enable us connecting to the database.
3. **Establish connection**. To connect with the database, you call the `Open()` function on the `sql` instance like so:

```
db, err := sql.Open("sqlite3", "./mydb.db")
```

   In the preceding command, we specify first the type of database and in the second parameter the name of the database and where it's located. We get a database instance back of type `*sql.DB`.
4. **Run queries**. At this point, we are free to run queries. You use `Query()` function and give it a SQL statement like in this example:

```
rows, err := db.Query("SELECT * FROM person")
```

   To iterate over the results, you can use a for-loop like so:

```
for rows.Next() {
   var uid int
   var name string
   var lastname string
   var created time.Time
   err = rows.Scan(&uid, &name, &lastname, &created)
}
```

5. **Run prepared statements**. Prepared statements are SQL statements where we can provide parameter values at a later point. You call the `Prepare()` function with `?` as placeholders where there will be data inserted:

```
stmt, err := db.Prepare("UPDATE person set lastname=? where uid=?")
```

   To run the statement against the database, you can call `Exec()`:

```
res, err := stmt.Exec("smith", 1)
```

   The `res` instance coming back has a function `RowsAffected()` that returns the number of affected rows:

```
affected, _ := res.RowsAffected()
```

   Getting affected rows is a good indicator that you actually changed something.
6. Close the database connection. You should close the database when you're done with it like so:

```
db.Close()
```

Assignment

In this assignment, we will create a Go program that's able to write and write to the database. We will go all the way from creating the database with the console to writing the Go code needed.

Create the database and populate it

We will create our database using the sqlite executable in the console.

1. Run `sqlite` to create the database and initialize the sqlite shell:

```
 sqlite3 mydb.db
sqlite3SQLite version 3.32.3 2020-06-18 14:16:19
Enter ".help" for usage hints.
```

   At this point, you have a database created. Next, we need some tables in there.
2. Run the following SQL command in the sqlite shell:

```
CREATE TABLE `person` (
          `uid` INTEGER PRIMARY KEY AUTOINCREMENT,
          `name` VARCHAR(64) NULL,
          `lastname` VARCHAR(64) NULL,
          `created` DATE NULL
      );
```

You know have the table "person" created. Next, we need some data in the table that we will interact with later in our Go code.

3. Run this SQL command to insert data into "person" table:

```
 insert into person(name,lastname, created) values ("joe", "schmoo", '2021-01-01');
```

Great, we now have data in our table. Time to focus on the Go code next.

4. Run `.exit` to exit the database.

Create a project

Now we will create a Go project and some code able to access our database.

1. Create *db.go* and give it this content:

```
package main

import (
 "database/sql"
 "fmt"
 "log"
 _ "github.com/mattn/go-sqlite3"
)

func main() {
 db, err := sql.Open("sqlite3", "./mydb.db")
 if err != nil {
  log.Fatal(err)
 }
 fmt.Println("database open")

 fmt.Println("bye")

 fmt.Println("closing db")
 db.Close()

}
```

Next, lets initialize our Go project.

2. Run the following commands to create our project:

```
go mod init sql-demo
```

3. Run `go mo tidy`, to install the needed packages you specified in the import section of your program (this will download and add "github.com/mattn/go-sqlite3" to your project):

```
go mod tidy
```

Read data

Next, we will add a function capable of reading data.

1. Add a function `Read()` like so:

```
func Read(db *sql.DB) {
  rows, err := db.Query("SELECT * FROM person")

 }
```

At this point, we have read the response into `rows`. Next, we need iterate on the response.

2. Add the following code, in `Read()` to iterate over the response:

```
for rows.Next() {
   var uid int
   var name string
   var lastname string
   var created time.Time
   err = rows.Scan(&uid, &name, &lastname, &created)
   if err != nil {
    log.Fatal(err)
   }
   fmt.Println(uid)
   fmt.Println(name)
   fmt.Println(lastname)
   fmt.Println(created)
}
```

Not the usage of `Scan()` and variables being sent in as references so the response is written to them.

Create data

Now we will create code that will allow us to create data in our database.

1. Add a function `Read()`:

```
func Create(db *sql.DB) {
  stmt, err := db.Prepare("INSERT INTO person(name, lastname, created) values(?,?,?)\
")

}
```

At this point, you have created a statement that when executed will attempt to insert row. Note the use of `?`, these are placeholders that you will need to provide values to at the moment of execution.

2. Add the following code to `Create()`:

```
if err != nil {
   log.Fatal(err)
  }
  res, err := stmt.Exec("Mrs", "Smith", "2022-01-01")
  if err != nil {
   log.Fatal(err)
  }
  affected, _ := res.RowsAffected()
  log.Printf("Affected rows %d", affected)
```

Note the call to `Exec()`, here you are providing data and `?` is being replaced by the values you send in. Also note the last two rows:

```
affected, _ := res.RowsAffected()
log.Printf("Affected rows %d", affected)
```

Here for result `res` we are invoking `RowsAffected()` that returns the number of affected rows then we go on to print said value.

Update and delete data

Updating and deleting data takes on very similar approach to how to create data. You will use a statement that you prepare and then send in the real values. Below is the code for performing both these actions:

Update

```
func Update(db *sql.DB) {
 stmt, err := db.Prepare("UPDATE person set lastname=? where uid=?")
 if err != nil {
  log.Fatal(err)
 }
 res, err := stmt.Exec("smith", 1)
 if err != nil {
  log.Fatal(err)
 }
 affected, _ := res.RowsAffected()
 log.Printf("Affected rows %d", affected)
}
```

Delete

```
func Delete(db *sql.DB) {
 stmt, err := db.Prepare("delete from person where uid=?")
 if err != nil {
  log.Fatal(err)
 }
 res, err := stmt.Exec(1)
 if err != nil {
  log.Fatal(err)
 }
 affected, _ := res.RowsAffected()
 log.Printf("Affected rows %d", affected)
}
```

Solution

```
package main

import (
 "database/sql"
 "fmt"
 "log"
 "time"

 _ "github.com/mattn/go-sqlite3"
)

func Read(db *sql.DB) {
```

```
 rows, err := db.Query("SELECT * FROM person")
 if err != nil {
  log.Fatal(err)
 }
 for rows.Next() {
  var uid int
  var name string
  var lastname string
  var created time.Time
  err = rows.Scan(&uid, &name, &lastname, &created)
  if err != nil {
   log.Fatal(err)
  }
  fmt.Println(uid)
  fmt.Println(name)
  fmt.Println(lastname)
  fmt.Println(created)
 }
}

func Update(db *sql.DB) {
 stmt, err := db.Prepare("UPDATE person set lastname=? where uid=?")
 if err != nil {
  log.Fatal(err)
 }
 res, err := stmt.Exec("smith", 1)
 if err != nil {
  log.Fatal(err)
 }
 affected, _ := res.RowsAffected()
 log.Printf("Affected rows %d", affected)
}

func Create(db *sql.DB) {
 stmt, err := db.Prepare("INSERT INTO person(name, lastname, created) values(?,?,?)")
 if err != nil {
  log.Fatal(err)
 }
 res, err := stmt.Exec("Mrs", "Smith", "2022-01-01")
 if err != nil {
  log.Fatal(err)
 }
 affected, _ := res.RowsAffected()
```

```
56   log.Printf("Affected rows %d", affected)
57  }
58
59  func Delete(db *sql.DB) {
60   stmt, err := db.Prepare("delete from person where uid=?")
61   if err != nil {
62    log.Fatal(err)
63   }
64   res, err := stmt.Exec(1)
65   if err != nil {
66    log.Fatal(err)
67   }
68   affected, _ := res.RowsAffected()
69   log.Printf("Affected rows %d", affected)
70  }
71
72  func main() {
73   db, err := sql.Open("sqlite3", "./mydb.db")
74   if err != nil {
75    log.Fatal(err)
76   }
77   fmt.Println("database open")
78   Create(db)
79   Read(db)
80   // Update(db)
81
82   fmt.Println("bye")
83
84   fmt.Println("closing db")
85   db.Close()
86
87  }
```

Read and write to files

There are different types of files, text files, files containing images, videos etc. For that reason you might want to read the content differently, either byte by byte or maybe even the entire file in one go, as text.

Introduction

In this chapter you will learn to:

- Read content from text files.
- Write to text files.
- Append text at the end of a text file.

Read a text file

One approach could be using the `ioutil` library and its `ReadFile()` method like so:

```
1   import (
2     "io/ioutil"
3     "log"
4   )
5   func main() {
6     filebuffer, err := ioutil.ReadFile(path)
7
8     if err != nil {
9       log.Fatal(err)
10    }
11    var inputdata string = string(filebuffer)
12  }
```

Note how the result of reading the file ends up in `filebuffer`. To interpret it as a string, you need to convert it via `string(filebuffer)`. Now, you're ready to process the file content, read it line by line or whatever you want to do.

Write text to a file

In this scenario, we are looking to do two things:

- Create a file.
- Write text to our newly created file.

For this scenario, we can use the `os` library and a combination of the `Create()` method, to create a file and the `WriteString()` method to write a string to the file.

```
import (
 "os"
 "log"
)

f, err := os.Create(path)
if err != nil {
 log.Fatal(err)
}

n, err := f.WriteString(content + "\n")
if err != nil {
 log.Fatal(err)
}
fmt.Printf("wrote %d bytes\n", n)
f.Sync()
```

First the file is created calling `Create()`. From that, we get a file handle `f`. With `f`, we can call `WriteString()` with a string. Lastly, we call `Sync()` to ensure the string is persisted in the file.

Append to a file

Appending text, implies you already have an existing file. When you append, you information to the end of the file. Appending is something you are likely to do when you add new entries to a log file or adding a new purchase to a Point of Sale, POS system in a grocery store for example.

To append to a file you use the `OpenFile()` method in the `os` lib. What you need to do is to pass it some flags that states that you want to append content. You should also have a behavior that says, create if it doesn't already exist. You end up with code looking like so:

```
f, err := os.OpenFile("text.log",
 os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
if err != nil {
 log.Println(err)
}
defer f.Close()
```

The 0644, is a 3x3 bit flag. It sets permissing for User (6, Read/Write), Group (4, Read), and Other (4, Read).

To append a string, you can call `WriteString()` like so:

```
f.WriteString("my text \n")
```

Assignment

Imagine you have a file *invoices.csv* looking like so:

```
customer, amount, date
Wood LTD, 100, 2020-01-01
Metal, 345, 2020-01-29
Steel, 700, 2020-07-29
```

Open up and read the file content, line by line.

Solution

```
package main

import (
 "fmt"
 "io/ioutil"
 "log"
 "strings"
)

func main() {
 var path = "invoices.csv"
 filebuffer, err := ioutil.ReadFile(path)

 if err != nil {
```

```
  log.Fatal(err)
 }
 var inputdata string = string(filebuffer)

 rows := strings.Split(inputdata, "\n")
 for _, row := range rows {
  fmt.Println("row:", row)
 }
}
```

Challenge

See if you can split up each row further in columns and summarize how much the orders are worth in total.

Perform operations on files and directories

Things you likely want to do to files and directories are moving them about, removing them, rename them etc. In short, high-level operations that is less about the content of the file but doing something with it.

Introduction

In this chapter you will:

- Analyze a file for its metadata like size, modified date and more.
- Copy files from one location to the next.
- Rename files.
- Remove files.
- Create and read directories.

File information

You might want to look at a specific file and find out various details about it. Things that can be interesting are:

- **Name**, maybe youu started from a path looking at a list of file. Getting the filename can be valuable.
- **Size**. Getting the size of disc can be good to know.
- **Permission**. To know what permissions you have tells you want you are able to do with the file, like running it, writing to it and so on.
- **Last modified**. You might have a query looking for newly updated files only. Inspecting the modified date is what you want.
- **Is a directory**. Files and directories are ultimately just files. There's a flag distinguishing a file from a directory.

To get file information, use the `Stat()` function like so:

```
fileStat, err := os.Stat(path)
fmt.Println("File Name:", fileStat.Name())        // Base name of the file
fmt.Println("Size:", fileStat.Size())             // Length in bytes for regular fil\
es
fmt.Println("Permissions:", fileStat.Mode())      // File mode bits
fmt.Println("Last Modified:", fileStat.ModTime()) // Last modification time
fmt.Println("Is Directory: ", fileStat.IsDir())   // Abbreviation for Mode().IsDir()
```

Also when you call `ReadDir()` you get back an array of `FileInfo` objects:

```
files, err := ioutil.ReadDir(path)
```

Copy file

Copying a file is really three operations:

- **open** the file at the destination.
- **create** a new file at a given destination.
- **copy**, then transfer the content to that location.

Here's how you can implement a *copy* operation:

```
// copies 'test.txt' and its content to 'copy.txt'
src := "test.txt"
dest := "copy.txt"

srcFile, err := os.Open(src)
if err != nil {
  log.Fatal(err)
}
defer srcFile.Close()

newFile, err := os.Create(dest)
if err != nil {
  log.Fatal(err)
}
defer newFile.Close()

_, err = io.Copy(newFile, srcFile)
if err != nil {
  log.Fatal(err)
}
```

Rename

Rename is a bit easier to achieve. The `os` package has a `Rename()` function. Here's how to use it:

```
err := os.Rename(src, dest)
```

Remove file

To remove file, there's a `Remove()` function you can use. Here's how to use it:

```
err := os.Remove(path)
```

Create dir

To create a directory, you can use the `MkdirAll()` function in the `os` library. However, you should check whether the directory exist first. The way to do that is to use the `IsNotExist()` function like so:

```
_, err := os.Stat(dirName)

 if os.IsNotExist(err) {
  errDir := os.MkdirAll(dirName, 0755)
  if errDir != nil {
   log.Fatal(err)
  }
 } else if err != nil {
  log.Fatal("error creating dir")
 } else {
  log.Fatal("directory exist")
 }
```

As you can see on the above code:

1. you first use the `Stat()` function. The `Stat()` returns a `FileInfo` object or an error of type `PathError` if path doesn't exist.
2. `os.IsNotExist(err)`. This returns `true` if `err` is a `PathError`, i.e the path don't exist, which is good, we want to create it.
3. Finally, we call `os.MkdirAll(dirName, 0755)`. The `755` instruction is about permissions on the created directory, which gives the permissions, Read/Write/Execute, Read/Execute, Read/Execute. 755 is a common permission set on web servers. You essentially want to avoid anyone but you to modify the file.

Read dir

Reading a directory is quite straight forward. You can use the `ReadDir()` function on the `io/ioutil` library. Here's how you would read a directory:

```
files, err := ioutil.ReadDir(path)
```

`files` is an array of type `FileInfo`.

TODO, copy, rename, remove, check for existence

Assignment

Create the following files and directories like so:

```
tmp/
  a.txt
  b.xt
  subdir/
```

Your program should read the diretory *tmp* and for each entry list, if it's a dir or a file, the size and when last modified.

The programs output should look something like:

```
Reading directory tmp:
Name, Type, Size, Modified
a.txt, file, 1kb, 2022-01-01
b.txt, file, 1kb, 2022-01-01
subdir, directory, 1kb, 2022-01-01
```

Solution

```
package main

import (
 "fmt"
 "io/ioutil"
 "log"
)

func GetType(isDir bool) string {
 if isDir {
  return "directory"
 }
 return "file"
}

func main() {
 var path string = "tmp"
 files, err := ioutil.ReadDir(path)
 if err != nil {
  log.Fatal(err)
 }
 fmt.Println("Reading directory ", path)
 fmt.Println("Name, Type, Size, Modified")
 for _, file := range files {
  if err != nil {
   log.Fatal(err)
  }
  fmt.Printf("File Name: %s, ", file.Name())
  fmt.Printf("Type: %s, ", GetType(file.IsDir()))
  fmt.Printf("Size: %d, ", file.Size())
  fmt.Printf("Last Modified: %s, ", file.ModTime())
  fmt.Print("\n")
 }
}
```