Go Basics Training

Labs

Welcome to the Go basics training! We will begin by installing Go and setting up an editor. Then we will look at the basic language syntax. After an introduction to testing, we will discover the extensive Go standard library.

1. Getting Started

Install Go

To install Go follow the official installation instructions at https://go.dev/doc/install . After the installation you can verify if everything works by running `go version` in a terminal.

Install Editor

Of course you need an editor to edit the Go source files. There are no special requirements for the editor so you can use whatever editor you are familiar with. However, among Go developers certain editors are more popular than others and hence there is more tooling and plugins available for them. According to the Go Developer Survey 2020 the two most popular editors are:

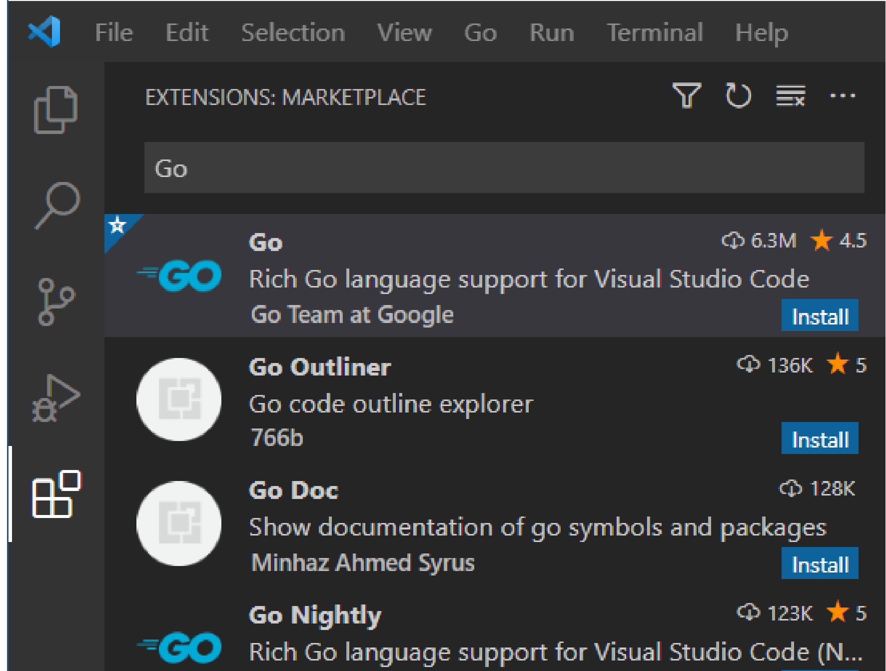
- VS Code (41%)
- GoLand / IntelliJ (35%)

For this course it doesn't matter which editor you use. If you are not sure which editor to choose, we recommend you use VS Code. It is available for free and there is an official Go extension.

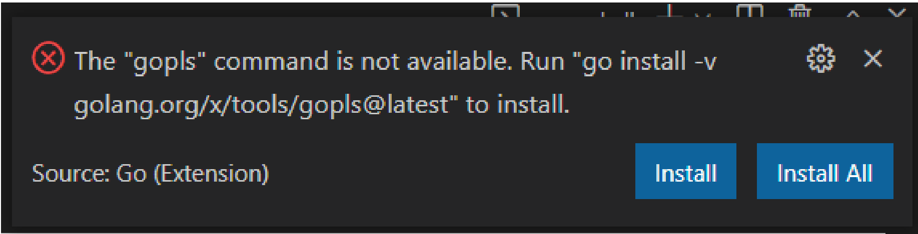
Install VS Code (optional)

If you don't use VS Code you can skip this section. This section will briefly describe how to install VS Code and the recommended extension for Go development:

- Download and Install VS Code from https://code.visualstudio.com/
- Start VS Code and open the extensions menu on the left side
- Search for the Go extension and install it



The first time you open a Go file the Go plugin will show a warning to you that certain tools are missing. Click on “Install All” to install all missing tools. This will take some time.



> **Auto Save**
>
> We recommend setting the “Auto Save” setting to `onFocusChanged`. This allows you to modify code and then run it, without manually saving.

Hello World

Now we want to build our first Go program which prints `Hello, World!`. To create a new Go project we create a new directory and initialize a Go module in it

```
mkdir hello
cd hello
go mod init hello
```

Then we create a new file `main.go` and put the following code in it:

```go
package main

import "fmt"

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

This file contains the main function which is the entrypoint for our application. We will learn about packages and the project structure in general later in this course.

Open a terminal and make sure you are in the `hello/` directory. Execute the following commands to build and run your program:

```bash
go build

# Linux / Mac
./hello

# Windows
.\hello.exe
```

2. Basics

This chapter will give you an overview of the most important Go language features. We tried to deliberately leave out more advanced features. Based on our experience, the covered topics are sufficient to understand most Go programs.

The official documentation is an excellent resource. It provides more details than this training and is especially helpful if you encounter a syntax construct that we did not cover.

- The Tour of Go is similar to the overview in this chapter but it covers more details. It goes over most of the language features by looking at small examples.
- Go Language Specification
- Effective Go offers some best practices and tips for writing Go.

Learning a new language is hard. You will be bombarded by new information and concepts. Remember that the primary goal is to get to know which features are available and not to memorize every little detail. You will probably have to come back to these earlier exercises later in this training.

Ask questions if something is unclear, use Google and make your own notes.

2.1. Variables

Basics

Go is a statically typed language. This means that each variable gets a type on declaration which can't be changed later.

Commonly used data types are:

- `int` and `uint`
- `float32` and `float64`
- `bool`
- `string`
- `byte` (alias for `uint8`)
- `error` to return errors from functions

All Go's predeclared identifiers are defined in the builtin package.

Declaration

The short assignment statement `:=` declares a variable and assigns a value to it. The type of variable is inferred from the value (type inference).

```go
package main

import "fmt"

func main() {
	// type string inferred from "foo.txt"
	name := "foo.txt"

	// type int inferred from 42
	size := 42

	// type bool inferred from true
	isFile := true

	fmt.Println(name, size, isFile)
}
```

```
Output:
foo.txt 42 true
```

Once a variable is declared you can assign values to it with a regular assignment `=`.

```go
package main

import "fmt"

func main() {
	name := "foo.txt"

	name = "bar.csv"

	fmt.Println(name)
}
```

```
Output:
bar.csv
```

With the `var` keyword you can declare a variable without assigning a value to it.

```go
package main

import "fmt"

func main() {
	// single variable
	var name string

	// multiple variables
	var (
		size   int
		isFile bool
	)

	fmt.Println(name, size, isFile)
}
```

```
Output:
 0 false
```

Zero values

If you declare a variable and do not assign a value it is initialized with the zero value of its type.

The zero values are:

- `0` for numeric types
- `false` for booleans
- `""` (empty string) for strings

2.2. Flow control

If Else

Conditionals in Go are similiar to other programming languages. Notice that there are no round brackets surrounding the condition.

```go
package main

import "fmt"

func main() {
 x := 10
 if x >= 5 {
  fmt.Println("X is greater or equal to 5")
 } else {
  fmt.Println("X is smaller than 5")
 }
}
```

```
Output:
X is greater or equal to 5
```

Multiple logical conditions can be combined with `&&` (AND) and `||` (OR).

Switch

If you are testing one variable for multiple conditions with `else if` the code can quickly become confusing. In these cases a switch statement can be used. The last `default` case is equivalent to `else`.

```go
package main

import (
	"fmt"
)

func main() {
	dayOfWeek := 3

	switch dayOfWeek {
	case 1:
		fmt.Println("Sunday")
	case 2:
		fmt.Println("Monday")
	case 3:
		fmt.Println("Tuesday")
	case 4:
		fmt.Println("Wednesday")
	case 5:
		fmt.Println("Thursday")
	case 6:
		fmt.Println("Friday")
	case 7:
		fmt.Println("Saturday")
	default:
		fmt.Println("Invalid day")
	}
}
```

```
Output:
Tuesday
```

Loops

The examples below show the basic loop constructs. We will look at an additional variant in the chapter *2.6. Slices*.

For - Classical

```go
package main

import "fmt"

func main() {
 for i := 0; i < 5; i++ {
  fmt.Println(i)
 }
}
```

```
Output:
0
1
2
3
4
```

While equivalent

```go
package main

import "fmt"

func main() {
 i := 0
 for i < 5 {
  fmt.Println(i)
  i++
 }
}
```

```
Output:
0
1
2
3
4
```

2.3. Functions

Basics

The `func` keyword declares a function.

In the following example we declare the function `add`. It takes two parameters `a` and `b` of type `int` and returns a value of type `int` which is the sum of `a` and `b`.

```
1	package main
2	
3	import "fmt"
4	
5	func add(a int, b int) int {
6		return a + b
7	}
8	
9	func main() {
10		result := add(2, 3)
11		fmt.Println(result)
12	}
```

```
Output:
5
```

No Return Value

If a function does not return a result the return type can be left out.

```
1	package main
2	
3	import "fmt"
4	
5	func sayHello(name string) {
6		fmt.Println("Hello", name)
7	}
8	
9	func main() {
10		name := "Bob"
11		sayHello(name)
12	}
```

```
Output:
Hello Bob
```

Multiple Return Values

A function can return multiple values. The following example contains a function `sub`, which returns two values:

1. The result of the subtraction of `a` and `b`
2. A boolean which indicates if the result is negative

```
package main

import "fmt"

func sub(a int, b int) (int, bool) {
 result := a - b
 isNegative := result < 0
 return result, isNegative
}

func main() {
 result, negative := sub(2, 3)
 fmt.Println(result, negative)
}
```

```
Output:
-1 true
```

In Go it is an error to declare variables without using them. If we only need a single value we can discard variables by assigning them to `_`:

```
result, _ := sub(2, 3)
```

Returning Errors

Functions that can fail return error values. In many other languages exceptions are thrown to indicate error conditions. Go does not have exceptions.

Errors are returned from functions like every other return value. From the function signature we can see if a function can fail or not.

Let's take a look at the function os.ReadFile from the Go standard library. It's signature looks like this:

```
func ReadFile(name string) ([]byte, error)
```

In the function signature we see that the last return value is of type `error`. So we know that the function can fail. In the case of `ReadFile` possible errors could be that the file does not exist or that we do not have enough

permission to read the file.

If the returned error value is not empty (`nil`) an error has occurred.

```
content, err := os.ReadFile("test.txt")
if err != nil {
 // handle error
}
```

In many cases handling the error means:

- Passing the error up to the caller: `return err`
- Logging or printing an error (e.g. print to standard error with fmt.Fprintln)

```
1   package main
2
3   import (
4    "fmt"
5    "os"
6   )
7
8   func main() {
9    fileContent, err := os.ReadFile("test.txt")
10   if err != nil {
11    fmt.Fprintln(os.Stderr, err)
12    os.Exit(1)
13   }
14   fmt.Println(fileContent)
15  }
```

```
Output:
open test.txt: no such file or directory
```

Function As Values

In Go functions are first-class values. This means that you can pass functions around like ordinary values.

```go
package main

import "fmt"

// run takes a function myfunc and runs it with a and b
func run(a int, b int, myfunc func(int, int) int) int {
	return myfunc(a, b)
}

func main() {
	add := func(a int, b int) int { return a + b }
	sub := func(a int, b int) int { return a - b }

	result1 := run(2, 3, add)
	result2 := run(2, 3, sub)

	fmt.Println(result1, result2)
}
```

```
Output:
5 -1
```

2.4. Pointers

Basics

In addition to the basic data types Go also have pointers. A pointer contains a memory address of an actual value or `nil` if they do not point to anything. The zero value of a pointer is `nil`.

Pointer types are indicated with a star. For example:

- The type `*int` is a pointer to an `int`
- The type `*bool` is a pointer to a `bool`

Concerning pointers you mainly have to remember two operations. How to create a pointer to a value and how go obtain a value from a pointer:

Operation	Example	Code	Description
Value To Pointer	`int` to `*int`	`myPointer = &myInt`	With the `&` operator you can create a pointer to a value
Value From Pointer	`*int` to `int`	`myInt = *myPointer`	With the `*` operator we can obtain the actual value from a pointer (dereference)

```go
package main

import "fmt"

func main() {
 // declare myPointer as *int. Initially it is nil (zero value)
 var myPointer *int

 // myInt is of type int
 myInt := 42

 // Value To Pointer
 // myPointer becomes pointer to myInt
 myPointer = &myInt

 // Value From Pointer
 // Change myInt through myPointer. We can not assign an int to a *int hence we
 // have to dereference myPointer with * to assign a new number
 *myPointer = 12

 fmt.Println(myInt)
 fmt.Println(*myPointer)
}
```

```
Output:
12
12
```

If you dereference a pointer you should always be sure that it is not `nil`. Otherwise your program crashes with an error `invalid memory address or nil pointer dereference` (same as NullPointerException in Java).

Why Use Pointers

If we work with basic data types like `int`, `bool`, `float64`, etc. we rarely use pointers. We introduce the topic at this time because in the next chapter we will get to know about structs. Pointers are often used together with structs.

In Go function arguments are passed by value. This means they are copied in memory and a new variable is created. For large variables like structs this can be a performance problem. To overcome this we can use pointers. With pointers only the memory address to the variable value is passed and the variable itself is not copied. This is called passing values by reference.

2.5. Structs

Basics

Structs are used to group related variables called fields. In that respect structs in Go are similar to classes or objects in other languages.

Declare Struct Type

This statement declares a new struct type called `User`. This struct contains three fields. The name of the user (`string`), the number of failed login attempts (`int`) and if the user is locked (`bool`):

```
type User struct {
 Name         string
 FailedLogins int
 Locked       bool
}
```

Create Instance

A new struct instance can be created using a struct literal:

```
myUser := User{
 Name:         "admin",
 FailedLogins: 0,
 Locked:       false,
}
```

When createing a new instance you don't have to specify all struct fields. If a field is not set it is initialized to the zero value of its type.

The following example creates the same struct as the previous example. The field `FailedLogins` is set to `0` and `Locked` is set to `false` since `0` and `false` are the zero values of `int` and `bool`.

```
myUser := User{
 Name: "admin",
}
```

Access Struct Fields

The individual fields of a struct are accessed using a dot:

```
// say hello to the user
fmt.Printf("hello %s", myUser.Name)

// increase the number of failed login attempts
myUser.FailedLogins += 1

// lock to user
myUser.Locked = true
```

Here you see a full example:

```
1   package main
2
3   import "fmt"
4
5   type User struct {
6    Name         string
7    FailedLogins int
8    Locked       bool
9   }
10
11  func main() {
12   myUser := User{
13    Name: "admin",
14   }
15
16   fmt.Println(myUser)
17
18   myUser.FailedLogins += 1
19
20   if myUser.FailedLogins > 0 {
21    myUser.Locked = true
22   }
23
24   fmt.Println(myUser)
25  }
```

```
Output:
{admin 0 false}
{admin 1 true}
```

Structs As Function Arguments

Structs can be passed to functions like any other value. If we pass a struct to a function it gets copied.

```go
func Print(user User) {
 fmt.Println(user.Name)
}
```

If we want to change a struct within a function we have to pass a pointer to a struct to the function. Otherwise within the function we would only change the copy. Even if we do not change a struct within a function we often just pass a pointer

Consider a function `Reset` which should reset the failed login attempts and unlock a user. For this function instead of expecting a user `User` which would only be copy we expect a pointer to a user (`*User`) as argument:

```go
func Reset(user *User) {
 user.FailedLogins = 0
 user.Locked = false
}
```

As we learnt in chapter *2.4. Pointers* we can obtain a pointer to a value by preceding it with `&` :

```go
Reset(&myUser)
```

We can also directly obtain a pointer to a user `*User` on creation as follows:

```go
myUser := &User{
 Name: "admin",
}

Reset(myUser)
```

Full example:

```go
package main

import "fmt"

type User struct {
 Name         string
 FailedLogins int
 Locked       bool
}

func Print(user User) {
 fmt.Println(user.Name, user.Locked)
}

func Reset(user *User) {
 user.FailedLogins = 0
 user.Locked = false
}

func main() {
 // myUser gets type *User because we use &User{} for initialization
 myUser := &User{
  Name: "admin",
 }

 // with *myUser we dereference the *User to obtain a User
 Print(*myUser)

 myUser.Locked = true

 Print(*myUser)

 Reset(myUser)

 Print(*myUser)
}
```

```
Output:
admin false
admin true
admin false
```

Constructors

Go does not have constructors. It is common to use a function `NewXxx` where `Xxx` is the struct name.

```go
package main

import "fmt"

type person struct {
	Name string
	Age  int
}

func NewPerson(name string, age int) person {
	return person{
		Name: name,
		Age:  age,
	}
}

func main() {
	pers := NewPerson("Ursli", 45)
	fmt.Println(pers)
}
```

```
Output:
{Ursli 45}
```

Methods

Instead of passing structs to functions we can attach methods to structs. In the previous example we passed a `User` or a `*User` to a function. Sometimes it is more convient to call such a function as method directly on the variable:

```go
myUser.Reset()

// instead of
Reset(myUser)
```

A method declaration looks like a function declaration. The only difference is that you specify to which struct you want to attach the method between the `func` keyword and the argument list:

```
func (user *User) Reset() {
 user.FailedLogins = 0
 user.Locked = false
}
```

Full example:

```
1    package main
2
3    import "fmt"
4
5    type User struct {
6     Name         string
7     FailedLogins int
8     Locked       bool
9    }
10
11   func (user User) Print() {
12    fmt.Println(user.Name, user.Locked)
13   }
14
15   func (user *User) Reset() {
16    user.FailedLogins = 0
17    user.Locked = false
18   }
19
20   func main() {
21    myUser := &User{
22     Name:    "admin",
23     Locked: true,
24    }
25
26    myUser.Print()
27
28    myUser.Reset()
29
30    myUser.Print()
31   }
```

```
Output:
admin true
admin false
```

Nested Structs

In comparision with other languages Go does not support inheritance. However something similiar can be achieved by using composition:

```
package main

import "fmt"

type person struct {
	firstName string
	lastName  string
	age       int
}

type blogPost struct {
	title   string
	content string
	author  person
}

func (p person) fullName() {
	// Printf allows us to use placeholders for variables.
	// %v can be used for all variable types.
	// You may also restrict this to %s and %d for strings and digits.
	fmt.Printf("%v %v\n", p.firstName, p.lastName)
}

func main() {
	chrigu := person{
		firstName: "Christian",
		lastName:  "Müller",
		age:       28,
	}
	blogPost1 := blogPost{
		// we do not need to specify the struct field names
		"Inheritance in Go",
		"Go supports composition instead of inheritance",
		chrigu,
	}
	blogPost1.author.fullName()
}
```

```
Output:
Christian Müller
```

Embedding

Multiple structs can also be combined. This is called embedding. Notice that we are using the `person` struct directly in `blogPost` without giving it a struct field name:

```go
package main

import "fmt"

type person struct {
	firstName string
	lastName  string
	age       int
}

type blogPost struct {
	title   string
	content string
	person
}

func (p person) fullName() {
	fmt.Println(p.firstName, p.lastName)
}

func main() {
	chrigu := person{
		firstName: "Christian",
		lastName:  "Müller",
		age:       28,
	}
	blogPost1 := blogPost{
		// we do not need to specify the struct field names
		"Inheritance in Go",
		"Go supports composition instead of inheritance",
		chrigu,
	}
	// Even though we did not specify a struct field name, we can use the type name
	blogPost1.person.fullName()
	// Since the struct is embedded, we can also directly access its fields!
	blogPost1.fullName()
	fmt.Println(blogPost1.firstName)
}
```

```
Output:
Christian Müller
Christian Müller
Christian
```

2.6. Slices

Basics

With a slice literal (e.g. `[]int{1, 2, 3}`) and the short assignment we we can initialize a new slice. Slices store multiple items of the same type.

```go
package main

import "fmt"

func main() {
	q := []int{2, 3, 5, 7, 11, 13}
	fmt.Println(q)
	fmt.Println("Length of slice:", len(q))
}
```

```
Output:
[2 3 5 7 11 13]
Length of slice: 6
```

Slice of structs

We can also store multiple instances of a struct.

```go
package main

import "fmt"

type Person struct {
	name string
	age  int
}

func main() {
	people := []Person{
		{name: "Christoph", age: 48},
		{name: "Susanne", age: 35},
		{name: "Peter", age: 29},
	}
	fmt.Println(people)
	fmt.Println(people[1])
}
```

```
Output:
[{Christoph 48} {Susanne 35} {Peter 29}]
{Susanne 35}
```

Appending items

To add items to an existing slice you can use `append`. Notice that `append` does not modify the original slice, but returns a new one.

```go
package main

import "fmt"

func main() {
 list := []int{1, 2, 3}
 fmt.Println("Initial slice:", list)

 // Add an item to a slice.
 list = append(list, 4)
 fmt.Println("Add one item:", list)

 // Add multiple items
 list = append(list, 5, 6, 7)
 fmt.Println("Add multiple items:", list)
}
```

```
Output:
Initial slice: [1 2 3]
Add one item: [1 2 3 4]
Add multiple items: [1 2 3 4 5 6 7]
```

Loops

In *2.2. Flow control* we learned about the basic `for` loop. However if we want to loop over all the slice elements, we prefer to use `range`. With `range` we can iterate over all items of a slice.

```go
package main

import "fmt"

func main() {
	slice := []int{2, 4, 8}
	for index, item := range slice {
		fmt.Println(index, item)
	}
}
```

```
Output:
0 2
1 4
2 8
```

In many cases we discard the index with `_` because we only need the actual item:

```go
package main

import "fmt"

func main() {
	slice := []int{2, 4, 8}
	for _, item := range slice {
		fmt.Println(item)
	}
}
```

```
Output:
2
4
8
```

Links

- SliceTricks contains various examples (delete an item from slice, push to a slice, pop from a slice, etc.)

2.7. Maps

Basics

A map maps keys to values. In other languages it is also called hash map or dictionary. The following example shows how to:

- initialize a map with an empty map literal
- set a value by key
- get a value by key
- delete a key
- get the length of the map

```go
package main

import "fmt"

func main() {
	m := map[string]int{}
	m["john"] = 66
	fmt.Println("john", m["john"])
	fmt.Println("len", len(m))
	delete(m, "john")
	fmt.Println("john", m["john"]) // the zero value is returned, if the item does not exist
	fmt.Println("len", len(m))
}
```

```
Output:
john 66
len 1
john 0
len 0
```

Check if key exists

We can check if a key exists by using the second return value when accessing a key in a map.

```go
package main

import "fmt"

func main() {
 m := map[string]int{
  "john": 66,
 }
 i, ok := m["john"] // Try changing the key
 if !ok {
  fmt.Println("The key does not exist in the map")
 }
 fmt.Println("i", i)
 fmt.Println("ok", ok)
}
```

```
Output:
i 66
ok true
```

Looping over elements

And range over all the elements:

```go
package main

import "fmt"

func main() {
 m := map[string]int{
  "john":    66,
  "another": 50,
 }
 for key, value := range m {
  fmt.Println("key", key)
  fmt.Println("value", value)
 }
}
```

```
Output:
key another
value 50
key john
value 66
```

2.8. Interfaces

Basics

Interfaces are used to express general behaviour across multiple types. Interface types are defined as a set of method signatures. All types which implement this set of methods implement the defined interface.

As an example we define the interface `Stringer`:

```
type Stringer interface {
 String() string
}
```

All types that implement the method `String()` and return a `string` implement the `Stringer` interface. Types implicitly implement an interface if they implement the required methods. This is different to many other language where we have to explicitly specify that a certain type implements an inteface (e.g. in Java with `implements`).

In the following example `User` and `Group` implement the `Stringer` interface. The `PrintAll` function expects a slice of `Stringer`s. This way we can pass users, groups and every other type which implements the `Stringer` interface to the `PrintAll` function. In the main function we then initialize a user and a group and put them both into a slice of `Stringer`s which we then pass to the `PrintAll` function.

```go
package main

import "fmt"

// interface type Stringer
type Stringer interface {
 String() string
}

// concrete type User
type User struct {
 Username string
}

func (u *User) String() string {
 return u.Username
}

// concrete type Group
type Group struct {
 Groupname string
}

func (g *Group) String() string {
 return g.Groupname
}

// a function which takes a list of Stringers
func PrintAll(items []Stringer) {
 for _, item := range items {
  // the only thing we know about the items is that they have a method String()
  text := item.String()
  fmt.Println(text)
 }
}

func main() {
 user1 := &User{
  Username: "andrea",
 }

 group1 := &Group{
  Groupname: "admins",
 }

 myItems := []Stringer{user1, group1}

 PrintAll(myItems)
}
```

```
Output:
andrea
admins
```

Any (Empty Interface)

The interface `any` is an alias for the empty interface `interface{}`. This alias got introduced in Go 1.18. Before we just wrote `interface{}` instead.

As the name says the empty interface does not contain any method signatures. Hence all types implement the empty interface.

> **Note**
>
> Usually you should avoid using `any`, because then you have to check the types yourself in the code during runtime and the compiler no longer helps you during the compile time. We still mention it here because in rare cases it can be useful and it is used in a couple of places in the standard library.

Any is used by functions which can handle any or at least multiple types. An example for this is the Marshal function from the `encoding/json` package which serializes a type into its JSON representation.

Its function signature looks as follows:

```go
func Marshal(v any) ([]byte, error)
```

It takes any type and returns the serialized data (`[]byte`) or an error (`error`) if something went wrong during the serialization.

Type Assertion

With a type assertion we can obtain the underlying concrete type of an interface type during runtime. In the following example we have a variable of type `Stringer`. `Stringer` is an interface type. With type assertion we can check if the variable is a `User`. If the variable is a `User` then `ok` is `true` and the `User` is returned.

```go
package main

import "fmt"

type Stringer interface {
 String() string
}

type User struct {
 Username string
}

func (u *User) String() string {
 return u.Username
}

func main() {
 var myStringer Stringer = &User{
  Username: "simone",
 }

 // type assertion
 user, ok := myStringer.(*User)

 if ok {
  fmt.Println(user.Username)
 } else {
  fmt.Println("myStringer is not of type User")
 }
}
```

```
Output:
simone
```

Type Switch

With a type switch we can check of which type a certain interface type is:

```go
package main

import "fmt"

func printType(item any) {
	switch val := item.(type) {
	case int:
		fmt.Println("type is int")
	case []int:
		fmt.Printf("int list with length %d\n", len(val))
	default:
		fmt.Println("unexpected type")
	}
}

func main() {
	printType(1)
	printType([]int{1, 2, 3, 4})
	printType(true)
}
```

```
Output:
type is int
int list with length 4
unexpected type
```

3. Project Structure

Basics

A Go project usually consists of exactly one module. Every directory within the project is a package. Hence a module is a collection of packages.

So usually we can say:

- Module = Project = Git Repository
- Package = Directory

Module path

To create a new Go project in the current directory we initialize a new module by running `go mod init <module-path>`. If we want to create a new project named `myproject` we would do the following steps:

```
mkdir myproject
cd myproject/
go mod init myprojct
```

The module path which is passed to `go mod init` is used to identify the module itself. It is the root for all the packages in your module. Usually a name like `myproject` is fine. If you plan to create a library which will be used by other Go projects, the module path should point to the location where the code is located.

Typically this would be something like this, which points to Github or another remote location:

```
go mod init github.com/myuser/myproject
```

For the following exmaples we assume that we choose `github.com/myuser/myproject` as our module path. On running `go mod init` a file `go.mod` gets created. In it we can see the module path of the current package.

`go.mod`

```
module github.com/myuser/myproject

go 1.17
```

Packages

Every directory within your project is a package. All files within a directory belong to the same package and hence must have the same package clause at the top. A package is a compilation unit. Identifiers in one file of

a package can be seen in every other file of the same package.

In the following example we have two files in a directory. They both belong the the package `main`. We can use the function `otherFunc()` in the file `./main.go` which is defined in the file `other.go` because the files are in the same package.

`./main.go`

```go
package main

import "fmt"

func main() {
 fmt.Println(otherFunc())
}
```

`./other.go`

```go
package main

func otherFunc() string {
 return "hello from otherFunc"
}
```

Package Name

In most cases the name of the directory and the package name are the same.

Assume we have a directory `./app` in our `github.com/myuser/myproject` module. In this case the package name would be `app`. Files in the directory `./app` would all have the same package clause at the top:

`./app/run.go`

```go
package app

// ...
```

`./app/app.go`

```go
package app

// ...
```

Main Package

A common exception where the package name is not the same as the directory name is the package `main`. If you want to build an executable binary from the package, we have to call the package `main`.

The `main` package must contain a function `main()` which is the entrypoint for the executable.

```
package main

func main() {
 // program entrypoint
}
```

You can not import a package main from other packages.

Imports

To use functions, variables, types, etc. (identifiers) from other packages we can import them by their import path using the import keyword:

```
package main

import (
 // imports package fmt from the Go standard library
 "fmt"

 // import package app from our own module
 "github.com/myuser/myproject/app"
)

func main() {
 // use exported function Calc() from our own app package
 number := app.Calc(23)

 // use exported function Println from the fmt package
 fmt.Println(number)
}
```

We can only use exported identifiers from the imported packages. Identifiers of a package are exported if the first character of an identifier is a Unicode upper case letter.

Assume we have a directory `./app` in your `github.com/myuser/myproject` module. In this case the package name would be `github.com/myuser/myproject/app`. In the directory we have file `calc.go` with some functions and variables:

`./app/calc.go`

```go
package app

// exported variable
var MagicNumber = 12

// exported function
func Calc(i int) int {
 return addOne(i)
}

// private function
func addOne(i int) int {
 return i + 1
}
```

In this case only the function `Calc()` and the variable `MagicNumber` can be used from other packages. The function `addOne()` can not be used becuase it is not exported.

We can divide the imports into three categories:

- Standard Library
- Imports from package in same project
- Imports from external projects

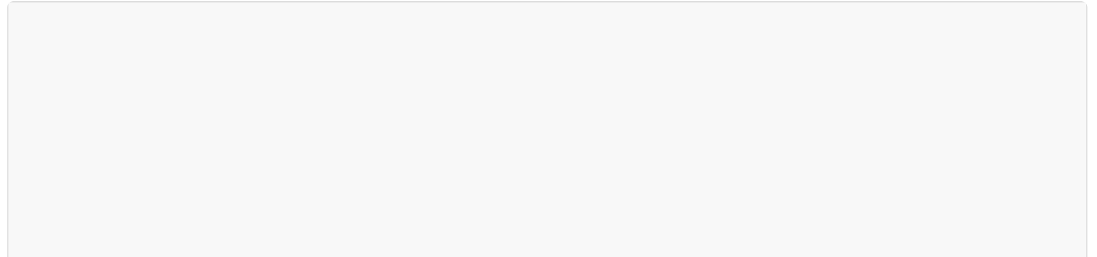
Standard Library

The Go standard library provides many useful packages. For more information see the documentation at: https://pkg.go.dev/std

Common packages in the standard library are:

- fmt : Formatted I/O with functions analogous to C's printf and scanf.
- os : Operating system functionality (opening files, ...)
- net/http : HTTP client and server
- encoding/json : Encoding and decoding of JSON

You can import packages from the standard library in every file without depending on external dependencies.



```go
package main

import (
 "fmt"
 "os"
)

func main() {
 content, err := os.ReadFile("test.txt")
 if err != nil {
  fmt.Fprintln(os.Stderr, err)
  os.Exit(1)
 }
 fmt.Printf("%s", content)
}
```

```
Output:
open test.txt: no such file or directory
```

Packages From Same Project

Assume the module path of your project is `github.com/myuser/myproject`. In your project you have a directory `./user` which contains a file `def.go`

`./user/def.go`

```go
package user

type User struct {
 Name string
}

func NewUser(name string) *User {
 return &User{
  Name: name,
 }
}
```

Then you can use the `user` package in another package of your project as follows:

`cmd/show/main.go`

```go
package main

import (
	"fmt"

	"github.com/myuser/myproject/user"
)

func createAdminUser() *user.User {
	return user.NewUser("admin")
}

func main() {
	admin := createAdminUser()
	fmt.Println(admin)
}
```

External Libraries

External packages are imported like internal packages. The only difference is that the root does not point to our own module `github.com/myuser/myproject`.

Further, before we can use an external package we have to download it using `go get`.

In the following example we create a function which generates UUIDs. For this we are using a UUID library from Google github.com/google/uuid .

```bash
# run this in the root of your project
go get github.com/google/uuid
```

```go
package main

import (
	"fmt"

	"github.com/google/uuid"
)

func generateUUID() string {
	return uuid.NewString()
}

func main() {
	fmt.Println(generateUUID())
}
```

```
Output:
3e87369c-76fd-4e73-b86c-cc46f46996e5
```

Instead of getting the dependency beforehand with `go get` you can also just add the code and then run `go mod tidy`.

4. Testing

Writing Tests

Tests are defined as functions in the following form:

```
func TestXxx(t *testing.T)
```

The tests reside in the same directory as the source code. The test files end with `_test.go`. The code in these files is not compiled into the binary when building the project. For demonstration purposes we put the function and test in the same code block. These are usually in a different file (e.g. `calculator.go` and `calculator_test.go`).

```
1   package main
2
3   import "testing"
4
5   func Add(a, b int) int {
6    return a + b
7   }
8
9   func TestAdd(t *testing.T) {
10   expected := 5
11
12   got := Add(2, 3)
13
14   if got != expected {
15    t.Fatalf("expected %d, got %d", expected, got)
16   }
17  }
```

```
Output:
=== RUN   TestAdd
--- PASS: TestAdd (0.00s)
PASS
```

Test failures

There are two methods to output test failures:

- `t.Error` prints the error without stopping the current test
- `t.Fatal` aborts the current test

Both variants also support template strings by appending `f`:

```
if got != expected {
 t.Fatalf("Wrong output returned. Got: %v Expected: %v", got, expected)
}
```

Try and be explicit with your error messages. It should be clear:

- What is being tested
- What was received
- What was expected

Running tests

```
# Run all tests in the currenty directory
go test

# Run all tests in the current directory and all subdirectories
go test ./...

# Run a single test
go test -run TestAdd

# Clean the cache, so that all tests are rerun
go clean -testcache
```

Compare structs and slices

It should be noted that comparing structs with `==` only works for simple cases. As soon as the struct contains pointers or slices a different method has to be used. There are two variants:

- reflect.DeepEqual is older and in the standard library. However it has less features (e.g. no `Diff` function) and cannot compare things like `time` in different time zones.
- go-cmp is a new library that makes comparing and printing the output easy. It should only be used for tests and never in production code. We recommend using this library.

```go
package main

import (
 "fmt"
 "reflect"
 "testing"

 "github.com/google/go-cmp/cmp"
)

func TestEqual(t *testing.T) {
 type Role struct {
  Name string
 }
 type User struct {
  Name string
  Role *Role
 }
 user1 := User{Name: "Andrea"}
 user2 := User{Name: "Andrea"}
 fmt.Println("Compare simple struct:", user1 == user2)

 user1.Role = &Role{"admin"}
 user2.Role = &Role{"admin"}
 fmt.Println("Compare struct with pointer", user1 == user2)
 fmt.Println("Compare complex structs with reflect", reflect.DeepEqual(user1, user2))
 fmt.Println("Compare complex structs with go-cmp", cmp.Equal(user1, user2))

 user2.Role = &Role{"user"}
 fmt.Print("Diff of two structs with go-cmp", cmp.Diff(user1, user2))
}
```

```
Output:
=== RUN   TestEqual
Compare simple struct: true
Compare struct with pointer false
Compare complex structs with reflect true
Compare complex structs with go-cmp true
Diff of two structs with go-cmp  main.User{
   Name: "Andrea",
-  Role: &main.Role{Name: "admin"},
+  Role: &main.Role{Name: "user"},
  }
--- PASS: TestEqual (0.00s)
PASS
```

Table driven tests

Instead of writing many small tests, we prefer to group tests that test a single function. This is achieved by putting all the test cases in a single “table”. The table contains input and expected output. We then loop over the table and check if the function returns the expected output.

```go
package main

import "testing"

func Add(a, b int) int {
 return a + b
}

func TestAdd(t *testing.T) {
 // Define a struct
 type TestCase struct {
  arg1     int
  arg2     int
  expected int
 }
 // Define the different inputs and expected outputs
 cases := []TestCase{
  TestCase{
   arg1:     2,
   arg2:     3,
   expected: 5,
  },
  TestCase{
   arg1:     20,
   arg2:     30,
   expected: 50,
  },
  TestCase{
   arg1:     2,
   arg2:     -3,
   expected: -1,
  },
 }
 // Loop through the test cases
 for _, tc := range cases {
  got := Add(tc.arg1, tc.arg2)
  if got != tc.expected {
   t.Errorf("Add(%d, %d): expected %d, got %d", tc.arg1, tc.arg2, tc.expected, got)
  }
 }
}
```

```
Output:
=== RUN   TestAdd
--- PASS: TestAdd (0.00s)
PASS
```

Subtests

A single test can be run with `go test -run TestAdd`. However with table driven tests, we cannot run a single test case from the table. The `testing` package allows us to `Run` another test within our test. The format is:

```
t.Run("test name", func(t *testing.T) {
 // The test comes here
})
```

```
1       package main
2
3       import "testing"
4
5       func Add(a, b int) int {
6        return a + b
7       }
8
9       func TestAdd(t *testing.T) {
10       t.Run("Add(2,3)", func(t *testing.T) {
11        expected := 5
12
13        got := Add(2, 3)
14
15        if got != expected {
16         t.Fatalf("expected %d, got %d", expected, got)
17        }
18       })
19       t.Run("Add(2,-3)", func(t *testing.T) {
20        expected := -1
21
22        got := Add(2, -3)
23
24        if got != expected {
25         t.Fatalf("expected %d, got %d", expected, got)
26        }
27       })
28      }
```

```
Output:
=== RUN   TestAdd
=== RUN   TestAdd/Add(2,3)
=== RUN   TestAdd/Add(2,-3)
--- PASS: TestAdd (0.00s)
    --- PASS: TestAdd/Add(2,3) (0.00s)
    --- PASS: TestAdd/Add(2,-3) (0.00s)
PASS
```

Now a single test can be run:

```
go test -v -run 'TestAdd/Add\(2,3\)'
```

We need to escape the parentheses, because the value is interpreted as a regex. The `-v` flag makes the command output more verbose.

Coverage

The test coverage tells us how much of our code is tested.

```
go test -cover
```

To see exactly which lines are not tested, run the following:

```
# Generates coverage report
go test -coverprofile=coverage.out
# Opens browser with detailed report
go tool cover -html=coverage.out
```

Benchmarks

To test the performance of function we can write benchmarks. These are also placed in the `_test.go` files and following format:

```
func BenchmarkXxx(b *testing.B) {
 for n := 0; n < b.N; n++ {
  // Call a function here
 }
}
```

Run the benchmarks with `go test -bench .`

5. Standard Library

The standard library consists of many useful packages. When possible try to use the standard library. Only use a 3rd party package from Github when necessary.

In this chapter we will look at some of the most important packages.

5.1. Input/Output

Basics

```go
package main

import (
 "fmt"
 "log"
 "os"
)

func main() {
 const filename = "/tmp/file.txt"

 err := os.WriteFile(filename, []byte("Hello, file system\n"), 0644)
 if err != nil {
  log.Fatal(err)
 }

 content, err := os.ReadFile(filename)
 if err != nil {
  log.Fatal(err)
 }

 fmt.Printf("%s", content)
}
```

```
Output:
Hello, file system
```

Reader/Writer

The above example of reading a file loads the whole file into memory. This can cause problems when we are dealing with large files. To solve this problem, we can load chunks of data and place them into a buffer.

The Go standard library provides the Reader and Writer interface for reading and writing data in a streaming fashion. The various reader and writer implementations are then often used together with the primitives from

the io package (e.g. io.Copy).

The io.Writer interface expects a slice of bytes and returns the amount of bytes that were written and an error.

```
type Writer interface {
 Write(p []byte) (n int, err error)
}
```

The io.Reader interface expects an byte slice. This slice is used as a buffer. It returns the amount of bytes that were read and an error.

```
type Reader interface {
 Read(p []byte) (n int, err error)
}
```

Here is an example were we copy data from a bytes.Buffer to a file (os.File). We create the buffer from a string using bytes.NewBufferString . Then we open the file with os.Create . With io.Copy we copy all bytes from the buffer to the file.

```
1   package main
2
3   import (
4    "bytes"
5    "fmt"
6    "io"
7    "os"
8   )
9
10  func main() {
11   // buffer implements io.Reader and is our source
12   buffer := bytes.NewBufferString("Some file content")
13
14   // file implements io.Writer and is our destination
15   file, err := os.Create("./io.txt")
16   if err != nil {
17    fmt.Println(err)
18    os.Exit(1)
19   }
20   defer file.Close()
21
22   // reades bytes from buffer and writes them to file
23   n, err := io.Copy(file, buffer)
24   fmt.Printf("%d bytes written\n", n)
25  }
```

```
Output:
17 bytes written
```

5.2. JSON

Encoding

Go allows us to encode structs to json by using json.Marshal .

```
package main

import (
 "encoding/json"
 "fmt"
 "os"
)

type User struct {
 Name      string
 FullName  string
 Followers int
}

func main() {
 user := User{
  Name:      "Alice",
  FullName:  "Alice Nyffenegger",
  Followers: 44,
 }

 output, err := json.Marshal(user)
 if err != nil {
  fmt.Println(err)
  os.Exit(1)
 }
 fmt.Printf("%s\n", output)
}
```

```
Output:
{"Name":"Alice","FullName":"Alice Nyffenegger","Followers":44}
```

> **Note**
>
> Other packages (e.g. json) cannot access the struct fields if they are lower case. Be sure to make them public by upper casing the first letter.

Tags

Often the names in your struct and the names you want in the serialized JSON are not the same. With tags you can map certain struct fields to other field names in the JSON representation:

```
package main

import (
 "encoding/json"
 "fmt"
 "os"
)

type User struct {
 Name      string `json:"short_name"`
 FullName  string `json:"name"`
 Followers int    `json:"followers"`
}

func main() {
 user := User{
  Name:      "Alice",
  FullName:  "Alice Nyffenegger",
  Followers: 44,
 }

 output, err := json.Marshal(user)
 if err != nil {
  fmt.Println(err)
  os.Exit(1)
 }
 fmt.Printf("%s\n", output)
}
```

```
Output:
{"short_name":"Alice","name":"Alice Nyffenegger","followers":44}
```

Decoding

Decoding works similiarly with the json.Unmarshal function. In addition to the JSON data (as a byte slice `[]byte`) we also pass a pointer to the struct itself. Since we use a pointer the struct can be modified directly and is not returned by `Unmarshal`.

With backticks we can create multiline strings or strings which contain a lot of quotes (`"`) like JSON data.

```go
package main

import (
	"encoding/json"
	"fmt"
	"os"
)

type User struct {
	Name      string `json:"short_name"`
	FullName  string `json:"name"`
	Followers int    `json:"followers"`
}

func main() {
	input := `{"short_name":"Alice","name":"Alice Nyffenegger","followers":44}`

	user := User{}

	err := json.Unmarshal([]byte(input), &user)
	if err != nil {
		fmt.Println(err)
		os.Exit(1)
	}
	fmt.Printf("%s %s %d\n", user.Name, user.FullName, user.Followers)
}
```

```
Output:
Alice Alice Nyffenegger 44
```

5.3. HTTP Client

The Go standard library offers a HTTP package which provides a server and a client. In the following sections we learn how we can use the HTTP client.

Quick start

The following example shows how we can perform a GET request and print the body to the standard output.

```go
package main

import (
 "fmt"
 "io"
 "net/http"
 "os"
)

func main() {
 resp, err := http.Get("https://google.com")
 if err != nil {
  fmt.Fprintln(os.Stderr, err)
  os.Exit(1)
 }
 _, err = io.Copy(os.Stdout, resp.Body)
 if err != nil {
  fmt.Fprintln(os.Stderr, err)
  os.Exit(1)
 }
}
```

The function http.Get returns a http.Response . The response contains the field `Body` which is an `io.Reader` from which we can read the response.

http.Client

The shortcut functions like http.Get and http.Post are convenient but in most of the cases we want more control over our requests (e.g. set a request timeout).

Usually we create our own http.Client with appropriate settings. You should always set a `Timeout` otherwise, if a request is blocked it will hang forever. We can use a single HTTP client instance throughout our whole application as it is safe for concurrent use. The client then manages a connection pool for us and we benefit from reusing connections if we do multilple requests to the same host.

We can then use the `Get` method on our http client::

```
client = &http.Client{
 Timeout: time.Second * 30,
}

resp, err := client.Get("https://google.com")
```

Set Headers

To set headers on a request we create a new http.Request with http.NewRequest .

```
req, err := http.NewRequest("GET", "http://localhost:8080", nil)

req.Header.Set("Authorization", "Bearer: my-super-secret-token")

resp, err := client.Do(req)
```

Send Content

To send content in the body of a POST request we pass a `io.Reader` to the `NewRequest` function. The reader could for example be an open file or a buffer. In the following example we create a bytes.Buffer from a string.

```
body := bytes.NewBufferString("body content")

req, err := http.NewRequest("POST", "http://localhost:8080", body)
if err != nil {
 return err
}

req.Header.Set("Content-Type", "text/plain")

resp, err := client.Do(req)
```

Send JSON

Often we want to send a struct serialized as JSON to an endpoint. The following example sends `{"name":"execute"}` as payload.

```go
type action struct {
	Name string `json:"name"`
}

executeAction := &action{
	Name: "execute",
}

payload, err := json.Marshal(executeAction)
if err != nil {
	return err
}

req, err := http.NewRequest("POST", "http://localhost:8080/run", bytes.NewBuffer(payload))
if err != nil {
	return err
}

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

resp, err := client.Do(req)
```

Receive JSON

```go
client := &http.Client{
 Timeout: time.Second * 20,
}

type action struct {
 Name string `json:"name"`
}

req, err := http.NewRequest("POST", "http://localhost:8080/run", nil)
if err != nil {
 return err
}

resp, err := client.Do(req)
if err != nil {
 return err
}

if resp.StatusCode > 399 {
 return fmt.Errorf("http status code %d", resp.StatusCode)
}

// we wrap the resp.Body reader in a io.LimitReader to avoid a crash if the server sends to
o much content
data, err := io.ReadAll(io.LimitReader(resp.Body, 1000))

// initialize an action
a := action{}

// deserialize data into a
err = json.Unmarshal(data, &a)

fmt.Println("action", a.Name)
```

5.4. HTTP Server

Quick start

Go allows us to start a simple HTTP server with minimal code:

```
1   package main
2
3   import (
4    "fmt"
5    "net/http"
6    "os"
7   )
8
9   func helloHandler(w http.ResponseWriter, r *http.Request) {
10   fmt.Fprintln(w, "Hello World")
11  }
12
13  func main() {
14   http.HandleFunc("/hello", helloHandler)
15
16   err := http.ListenAndServe(":8080", nil)
17   if err != nil {
18    fmt.Println(err)
19    os.Exit(1)
20   }
21  }
```

Now you can access http://localhost:8080/hello .

http.Handler

The http.Handler interface is the basic building block of all HTTP servers.

```
type Handler interface {
 ServeHTTP(ResponseWriter, *Request)
}
```

The ServeHTTP method has two parameters:

- http.ResponseWriter is an interface with the following methods:
  - `Header()` returns all headers and allows setting new ones.
  - `WriteHeader(statusCode int)` allows setting a status code for the response. It must be called before `Write()` and defaults to `http.StatusOK` if not set. For a full list of status codes see the

documentation .
    - `Write()` allows us to use all functions that expect an io.Writer interface (e.g. fmt.Fprint or io.WriteString)
- http.Request defines the request parameters. Example:
    - Method: GET, POST, PUT etc.
    - Header: request headers
    - Body: request body (typically used with POST requests)
    - Form: form values

So a minimal handler would look like this:

```
type myHandler struct{}

func (h *myHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
 fmt.Fprintln(w, "hello")
}
```

With the function http.HandlerFunc we can transform a function with the following signature into a http.Handler.

```
func myHandlerFunc(w http.ResponseWriter, r *http.Request) {
 fmt.Fprintln(w, "hello")
}

handler := http.HandlerFunc(myHandlerFunc)
```

`http.ResponseWriter`

The following is an example how to use `http.ResponseWriter` :

```go
package main

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

func helloHandler(w http.ResponseWriter, r *http.Request) {
 // set content type header
 w.Header().Set("Content-Type", "text/plain")

 // set status header (400)
 w.WriteHeader(http.StatusBadRequest)

 // respond with an error message
 // `fmt.Fprint` expects an `io.Writer` interface
 fmt.Fprint(w, "invalid request")
}

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

 // ListenAndServe always returns a non-nil error.
 err := http.ListenAndServe(":8080", nil)
 if err != nil {
  fmt.Println(err)
  os.Exit(1)
 }
}
```

`http.Request`

```go
package main

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

func helloHandler(w http.ResponseWriter, r *http.Request) {
 fmt.Fprintln(w, "Method: ", r.Method)
 fmt.Fprintln(w, "URL: ", r.URL)
 fmt.Fprintln(w, "Proto: ", r.Proto)
 fmt.Fprintln(w, "Header: ", r.Header)
 fmt.Fprintln(w, "Body: ", r.Body)
 fmt.Fprintln(w, "ContentLength: ", r.ContentLength)
 fmt.Fprintln(w, "Form: ", r.Form)
 fmt.Fprintln(w, "RemoteAddr: ", r.RemoteAddr)
}

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

 // ListenAndServe always returns a non-nil error.
 err := http.ListenAndServe(":8080", nil)
 if err != nil {
  fmt.Println(err)
  os.Exit(1)
 }
}
```

Receive Data

The field `Body` in the http.Request implements io.Reader. We can use functions that expect an `io.Reader` like io.ReadAll to read the body.

```go
package main

import (
 "io"
 "log"
 "net/http"
)

func echoHandler(w http.ResponseWriter, r *http.Request) {
 body, err := io.ReadAll(r.Body)
 if err != nil {
  w.WriteHeader(http.StatusBadRequest)
  return
 }
 w.Write(body)
}

func main() {
 http.HandleFunc("/echo", echoHandler)

 // ListenAndServe always returns a non-nil error.
 log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Try sending a request with `curl`:

```
curl -d 'body' http://localhost:8080/hello
```

Middeware

A common pattern is to wrap one handler within another handler. This could for example be used for logging or authentication. For this we pass our actual handler (inner handler) to the middleware.

```go
package main

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

func secretHandler() http.HandlerFunc {
 return func(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintln(w, "top secret")
 }
}

func authMiddleware(next http.Handler) http.HandlerFunc {
 return func(w http.ResponseWriter, r *http.Request) {
  user, password, _ := r.BasicAuth()
  if !(user == "admin" && password == "secret") {
   code := http.StatusUnauthorized
   http.Error(w, http.StatusText(code), code)
   return
  }
  next.ServeHTTP(w, r)
 }
}

func main() {
 http.Handle("/secret", authMiddleware(secretHandler()))

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

5.5. Date and Time

Basics

Time is represented with the time.Time struct.

The output in the examples below is generated by the Go playground . The Go playground uses a static time to achieve optimal caching performance.

```go
package main

import (
 "fmt"
 "time"
)

func main() {
 time := time.Now()
 fmt.Println(time)
}
```

```
Output:
2009-11-10 23:00:00 +0000 UTC m=+0.000000001
```

Parsing

To parse a string to `time.Time` we use time.Parse . Other programming languages often use “format strings” that look something like: `yyyy-MM-dd HH:mm:ss` . Go does things differently and uses a reference time.

```
Jan 2 15:04:05 2006 MST
```

An easy way to remember this value is that it holds, when presented in this order, the values (lined up with the elements above):

```
 1  2  3  4  5   6  -7
```

Every layout string is a representation of this time stamp. If you want to format a date differently, you must rewrite the above date.

Example:

```
Mon Jan 2 15:04:05 MST 2006
```

```go
package main

import (
 "fmt"
 "log"
 "time"
)

func main() {
 t, err := time.Parse("Mon Jan 2 15:04:05 MST 2006", "Thu Feb 25 11:06:39 PST 2021")
 if err != nil {
  log.Fatal(err)
 }
 fmt.Println(t)
 fmt.Println(t.Weekday())
}
```

```
Output:
2021-02-25 11:06:39 +0000 PST
Thursday
```

Comparing dates

Dates can be compared with time.Before and time.After

```go
package main

import (
 "fmt"
 "time"
)

func main() {
 year2000 := time.Date(2000, 1, 1, 0, 0, 0, 0, time.UTC)
 year3000 := time.Date(3000, 1, 1, 0, 0, 0, 0, time.UTC)

 isYear3000AfterYear2000 := year3000.After(year2000) // True
 isYear2000AfterYear3000 := year2000.After(year3000) // False

 fmt.Printf("year3000.After(year2000) = %v\n", isYear3000AfterYear2000)
 fmt.Printf("year2000.After(year3000) = %v\n", isYear2000AfterYear3000)

}
```

```
Output:
year3000.After(year2000) = true
year2000.After(year3000) = false
```

Duration

A time.Duration represents the time between two instants.

To get a duration we can subtract two dates:

```go
package main

import (
 "fmt"
 "time"
)

func expensiveCall() {
 time.Sleep(10 * time.Second)
}

func main() {
 t0 := time.Now()
 expensiveCall()
 t1 := time.Now()
 duration := t1.Sub(t0)
 fmt.Printf("The call took %v to run.\n", duration)
}
```

```
Output:
The call took 10s to run.
```

6. Concurrency

Goroutines

To run multiple functions concurrently we can start Goroutines. You can think of a Goroutine as a lightweight thread managed by the Go runtime.

```go
package main

import (
 "fmt"
 "time"
)

func print(word string) {
 for i := 0; i < 5; i++ {
  fmt.Println(word)
  time.Sleep(100 * time.Millisecond)
 }
}

func main() {
 go print("hello")
 print("world")
}
```

```
Output:
world
hello
world
hello
hello
world
world
hello
hello
world
```

When the main function returns, the program exits. It does not wait for other (non-main) goroutines to complete. If we would run both invocations of print in a goroutine the program would most likely not output anything at all, because it would be terminated before the goroutines are started.

```go
go print("hello")
go print("world")
```

> **Note**
>
> Be aware that all goroutines run in the same address space. So if two goroutines want to modify the same variable the access must be synchronized.

To synchronize goroutines we can use channels or the primitives from the sync package.

Package sync

sync.WaitGroup

With sync.WaitGroup we can wait for a collection of goroutines to finish. The following example downloads multiple URLs concurrently:

```go
package main

import (
	"fmt"
	"io"
	"net/http"
	"sync"
)

func main() {
	urls := []string{
		"https://google.com",
		"https://golang.org",
		"https://pkg.go.dev",
	}

	results := make([][]byte, len(urls))

	wg := &sync.WaitGroup{}

	for i, url := range urls {
		// we add one for each go routine
		wg.Add(1)
		go func(i int, url string) {
			// wg.Done() will decrease the couner of the WaitGroup by one
			defer wg.Done()

			resp, err := http.Get(url)
			if err != nil {
				return
			}

			results[i], _ = io.ReadAll(resp.Body)
		}(i, url)
	}

	// wait until the WaitGroup counter becomes zero
	wg.Wait()

	for i, result := range results {
		fmt.Printf("%s, size=%d", urls[i], len(result))
	}
}
```

```
Output:
https://google.com, size=0https://golang.org, size=0https://pkg.go.dev, size=0
```

sync.Mutex

In the following example we increase a counter from from ten goroutines 100 times. Since all goroutines modify

the same variable (`counter`) we have to synchronize the access. For this we can use sync.Mutex which provides us a mutual exclusion lock.

```
package main

import (
 "fmt"
 "sync"
)

func main() {
 counter := 0

 wg := &sync.WaitGroup{}
 mu := &sync.Mutex{}

 for i := 0; i < 100; i++ {
  wg.Add(1)
  go func() {
   for j := 0; j < 100; j++ {
    mu.Lock()
    counter++
    mu.Unlock()
   }
   wg.Done()
  }()
 }
 wg.Wait()
 fmt.Println(counter)
}
```

If you would remove the lock in the example above, in most of the cases the programm would not print `10000` becuase the access to `counter` is no synchronized.

Channels

A unique language feature of Go are channels. Channels allow you the pass data between goroutines in a concurrent-safe way. They are often used as a concurrency synchronization technique. You can think of a channel as a typed message queue. A producer can write data of a certain type into it (e.g. `int`) and a consumer can read from the channel.

We use the builtin function `make` to create a new channel. The following example creates a channel for numbers (`int`):

```
ch := make(chan int)
```

Then we can send values to and read values from the channel: the channel:

Operation	Code	Description
Send to channel	`ch <- 4`	Send the value `4` to the channel.
Read from channel	`myInt := <-ch`	Read a value from the channel and assign it to `myInt` .

Send and read operations on a channel are blocking. So we always need a goroutine on the other end which will send or read.

```go
package main

import "fmt"

func main() {
	ch := make(chan int)

	// we send the channel within a goroutine. otherwise the program would block here.
	go func() { ch <- 4 }()

	// read from the channel
	myInt := <-ch
	fmt.Println(myInt)
}
```

```
Output:
4
```

Buffered Channels

We can initialize a buffered channel if we pass an additional parameter to make:

```go
ch := make(chan int, 10)
```

This way a send will only block if the buffer is full. In the following example we don't need a goroutine to send the value, because the first two values we write go into the buffer.

```go
package main

import "fmt"

func main() {
	// initialize an int channel with a buffer size of 2
	ch := make(chan int, 2)

	// send does not block and values go into buffer
	ch <- 1
	ch <- 2

	// read values from channel
	fmt.Println(<-ch)
	fmt.Println(<-ch)
}
```

```
Output:
1
2
```

close and for-range

With the builtin function `close` a producer can close a channel to signal the consumers that no more data will be sent. Consumers can use `range` to read values from a channel until it is closed.

```go
package main

import "fmt"

func main() {
 ch := make(chan int)

 // start producer
 go func() {
  for i := 0; i < 10; i++ {
   // send to channel
   ch <- i
  }
  // close the channel. this will break the for loop below
  close(ch)
 }()

 // read from channel
 for item := range ch {
  fmt.Println(item)
 }
}
```

```
Output:
0
1
2
3
4
5
6
7
8
9
```

select

With `select` we can wait on multiple read or write operations on channels. A select blocks until one of the operations could be performed.

In the following example we have a calculation function which takes a long time. In the select block we wait on the result of the calculation function. Another select arm runs time.After with a timeout. The `time.After` function returns a channel on which a value is sent after the time has elapsed. This way we either get back the result from the calculation or we run into the timeout. You can change the timeout value to see how the behavoiur changes:

```go
package main

import (
 "fmt"
 "time"
)

func calculation(input int, result chan int) {
 // expensive calculation
 time.Sleep(time.Second * 2)
 output := input + 42

 result <- output
}

func main() {
 result := make(chan int)
 timeout := time.Second * 1

 go calculation(13, result)

 // wait on result or timeout
 select {
 case v := <-result:
  fmt.Printf("result %d\n", v)
 case <-time.After(timeout):
  fmt.Println("calculation timed out")
 }
}
```

```
Output:
calculation timed out
```

Further we can define a default action which is executed if all channels in the select are blocked. This allows us to implement a non blocking read or write on a channel.

```go
package main

import "fmt"

func nonBlockingRead(ch chan int) {
 select {
 case v := <-ch:
  fmt.Println(v)
 default:
  fmt.Println("blocked")
 }
}

func main() {
 ch := make(chan int, 1)

 // prints blocked becuase there is nothing to read
 nonBlockingRead(ch)

 ch <- 42

 // prints 42
 nonBlockingRead(ch)
}
```

```
Output:
blocked
42
```

Channel as Semaphore

In the following example we use a buffered channel as a sempahore to limit the number of functions which run concurrently:

```go
package main

import (
 "fmt"
 "sync"
 "time"
)

func calc(input int) {
 fmt.Printf("start calculation for %d\n", input)
 time.Sleep(time.Second * 2)
 fmt.Printf("calculation for %d finished\n", input)
}

func main() {
 wg := &sync.WaitGroup{}
 semaphore := make(chan struct{}, 3)
 workItems := []int{1, 2, 3, 4, 5}

 for _, workItem := range workItems {
  wg.Add(1)
  semaphore <- struct{}{}
  go func(workItem int) {
   defer wg.Done()
   calc(workItem)
   <-semaphore
  }(workItem)
 }
 wg.Wait()
}
```

```
Output:
start calculation for 3
start calculation for 2
start calculation for 1
calculation for 1 finished
start calculation for 4
calculation for 2 finished
start calculation for 5
calculation for 3 finished
calculation for 5 finished
calculation for 4 finished
```

7. Generics

The Go 1.18 release (February 2022) adds support for generics. In the following section, we will take a look at the new language feature but we will not cover every detail.

Why do we need Generics

Imagine you implement a function `contains` to check wheter a certain item is in a list:

```
func contains(list []int, item int) bool {
 for _, current := range list {
  if current == item {
   return true
  }
 }
 return false
}
```

The function above only works for slices of the type `int`. If you need the same logic for strings you have to implement a new function which would look exactly the same apart from the types in the function signature. If the function is in the same package you also have to choose a different name.

```
func containsString(list []string, item string) bool {
 for _, current := range list {
  if current == item {
   return true
  }
 }
 return false
}
```

Since Go 1.18 we can write the `contains` function in a generic way:

```go
package main

import "fmt"

func contains[T comparable](items []T, item T) bool {
 for _, current := range items {
  if item == current {
   return true
  }
 }
 return false
}

func main() {
 numbers := []int{1, 2, 4, 5}
 fmt.Println(contains(numbers, 1))
 fmt.Println(contains(numbers, 3))

 names := []string{"bob", "alice"}
 fmt.Println(contains(names, "alice"))
 fmt.Println(contains(names, "eve"))
}
```

```
Output:
true
false
true
false
```

Other examples

Max Function

In this example we create a `max` function which returns the bigger of two values. To specify the constraint we

use the `Ordered` interface from the constraints package:

```
1   package main
2
3   import (
4    "fmt"
5
6    "golang.org/x/exp/constraints"
7   )
8
9   func max[T constraints.Ordered](a, b T) T {
10   if a > b {
11    return a
12   }
13   return b
14  }
15
16  func main() {
17   fmt.Println(max(13, 42))
18   fmt.Println(max(3.14, 2.71))
19  }
```

```
Output:
42
3.14
```

Generic struct

We can also use generics to parametrize structs:

```
type LinkedList[T any] struct {
 head *Node[T]
}

type Node[T any] struct {
 item T
 next *Node[T]
}
```

Links

Official Go resources about the topic:

- An Introduction To Generics
- Tutorial: Getting started with generics
- When To Use Generics

* Type Parameters Proposal

Other blog posts about the topic:

* Generics can make your Go code slower
* Faster sorting with Go generics

8. Packaging

Basics

Go binaries are statically linked. That means all dependencies are included within the binary.

```
go build main.go
```

The binary will only work on the same platform that built it (e.g. 64bit Linux). Go allows us to easily generate binaries compatible with other architectures:

```
GOOS=linux GOARCH=arm64 go build main.go
GOOS=windows GOARCH=amd64 go build main.go
```

Docker

The `Dockerfile` uses multi-stage builds so that the resulting image is small and secure. The image is built with the full Golang Docker image and the resulting binary is copied into the Distroless image .

```
FROM golang:1.17 AS build
WORKDIR /src
# pre-copy/cache go.mod for pre-downloading dependencies and only redownloading them in sub
sequent builds if they change
COPY go.mod go.sum ./
RUN go mod download && go mod verify
COPY . .
# Build the executable
RUN CGO_ENABLED=0 go build -o /app ./main.go

FROM gcr.io/distroless/static
USER nonroot:nonroot
# Copy compiled app
COPY --from=build --chown=nonroot:nonroot /app /app
ENTRYPOINT ["/app"]
```

You might also see images online that are built with scratch . Scratch is a Docker image that contains absolutely nothing by default. We recommend using Distroless, because it already includes things like timezone data, user support and root certificates. The image is used by popular projects like Kubernetes .

Quality Control

Go Vet

`go vet` starts where the compiler ends by identifying subtle issues in your code. It's good at catching things where your code is technically valid but probably not working as intended.

```
go vet main.go
# test all files
go vet ./...
```

Unreachable code is an example of something `go vet` would find. The program compiles and run without errors.

```
1    package main
2
3    import "fmt"
4
5    func main() {
6     fmt.Println("Before return")
7     return
8     fmt.Println("After return")
9    }
```

```
Output:
./main.go:8:2: unreachable code
```

Linting

There are a few community tools that go even further and add many more checks:

- https://golangci-lint.run/
- https://staticcheck.io/

Testing

We already covered testing in chapter *4. Testing*. Tests are important to ensure everything works.

Race conditions can occur when two pieces of code run concurrently. They are usually hard to debug and cause mysterious failures. To find race conditions Go offers a handy `-race` flag, which can be added to any Go command:

```
go test -race mypkg     # test the package
go run -race mysrc.go  # compile and run the program
go build -race mycmd   # build the command
go install -race mypkg # install the package
```

Versioning

Go packages follow Semantic Versioning .

Given a version number MAJOR.MINOR.PATCH, increment the:

1. MAJOR version when you make incompatible API changes,
2. MINOR version when you add functionality in a backwards compatible manner, and
3. PATCH version when you make backwards compatible bug fixes.

Public Go packages should be backwards compatible .

9. Exercises

9.1. ASCII Pyramid

Task

Create a function which prints an ASCII Pyramid. The function must take the height of the pyramid as parameter.

If you call the function with `5` as parameter we should get the following pyramid:

```
    *
   ***
  *****
 *******
*********
```

Tips

The package fmt contains various print functions.

With fmt.Print you can print strings:

```
// print space
fmt.Print(" ")

// print newline
fmt.Print("\n")
```

On each line you have to print the appropriate number of spaces and stars:

- spaces: `height - lineNumber`
- stars: `lineNumber * 2 - 1`

Solution

https://github.com/acend/go-basics-training-examples/blob/master/ascii-pyramid/main.go

9.2. Number Guessing Game

Task

Write a number guessing game. At the start create a random number. Then ask the user to enter a number on the command line (standard input) until the user guesses the correct number.

If the user enters the string `exit` the program should exit.

The output could look like in the following example. In the example the user enters the numbers 5, 7 and 6.

```
guess number between 0 and 9
guess number: 5
wrong number. try again.
guess number: 7
wrong number. try again.
guess number: 6
correct
```

Tips

With the bufio package we can read from the standard input (os.Stdin) until a certain character occurs:

```
line, err := bufio.NewReader(os.Stdin).ReadString('\n')
```

The strings package contains various functions to work with strings. For example strings.TrimSpace to remove whithspace characters (spaces, newlines, etc.) from strings.

```
line = strings.TrimSpace(line)
```

The function rand.Intn returns a random integer. Before we use one of the random functions we have to seed the random number generator, otherwise we would always get the same number. As seed we can use the current time as nano seconds provided by the time package.

```
rand.Seed(time.Now().UnixNano())

// 0 <= randomNumber < 10
randomNumber := rand.Intn(10)
```

The package strconv implements conversions to and from string representations of basic data types like `int` or `float64`.

With strconv.Atoi you can convert a string into a `int`.

```
number, err := strconv.Atoi("8")
if err != nil {
 // handle failed conversion
}
```

Solution

https://github.com/acend/go-basics-training-examples/blob/master/number-guessing/main.go

9.3. Robot

Overview

Imagine we have a robot which moves around on a coordinate system. The robot starts at the position `0,0`.

For the robot we have a list of instructions in the following form:

```
right 3
up 1
left 5
right 4
down 2
up 2
```

Each line is one instruction. An instruction consists of a direction and a number which describes how far we move into this direction. After each instruction the robot is in a new position.

With the instructions from the example above we would do the following steps:

- The first instruction `right 3` moves the robot to the position `3,0`
- `up 1` moves the robot to the position `3,1`
- `left 5` -> `-2,1`
- `right 4` -> `2,1`
- `down 2` -> `2,-1`
- `up 2` -> `2,1`

In the example above we visited 6 positions (`3,0` , `3,1` , `-2,1` , `2,1` , `2,-1` , `2,1`). The furthest distance to the left we visited was `-2` . The furthest distance to the right we visited was `3` .

Tasks

Read all instructions from the file input.txt and perform the appropriate actions with the robot.

Answer the following questions:

1. What is the end position of the robot?
2. Which is the distance furthest to the left, which the robot visited?
3. Which is the distance furthest to the right, which the robot visited?
4. Which position did we visit most often?

Tips

- Try to solve the exercise only with the 6 example instructions first. Do not solve all tasks at once. Try to find the end position first and then try to extend your solution for the other tasks.
- To read the file you can use os.ReadFile which gives you the content of a whole file as a `[]byte`.

```
rawData, err := os.ReadFile(fileName)
if err != nil {
 return err
}
```

- You can iterate over each line by splitting the whole content at newlines:

```
for _, line := range strings.Split(string(rawData), "\n") {
 // parse line into an instruction
}
```

- Another option to read a file line by line would be to use bufio.Scanner
- The strings package contains a lot of other useful functions to work with strings (eg. strings.Cut or strings.Fields).
- You can convert a string into an integer using strconv.Atoi from the strconv package.
- You can represent directions (`up` , `right` , etc.) as integers:

```
const (
 UP    = 0
 RIGHT = 1
 DOWN  = 2
 LEFT  = 3
)
```

- Keep related state together in a struct. For example an instruction could look like this:

```
type Instruction struct {
 Direction int
 Distance  int
}
```

And a postion could look like this:

```
type Position struct {
 X int
 Y int
}

func (p *Position) Move(direction int, distance int) {
 // update coordinates accordingly
}
```

There are many ways on how to structure your code. If you have no idea how to start you can use the following skeleton:

```
const (
 UP    = 0
 RIGHT = 1
 DOWN  = 2
 LEFT  = 3
)

type Instruction struct {
 Direction int
 Distance  int
}

type Position struct {
 X int
 Y int
}

func (p *Position) Move(i Instruction) {
 // update position
 // e.g. p.X += i.Distance
}

func main() {
 // read/parse instructions from file
 instructions, err := readInstructions("input.txt")
 if err != nil {
  // handle error
 }

 // create your initial position
 pos := Postion{
  X: 0,
  Y: 0,
 }

 // loop over the instructions
 for _, i := range instructions {
  // adjust your position accordingly
  pos.Move(i)
```

```
 }

 // print final position
 fmt.Println(pos)
}

func readInstructions(fileName string) ([]Instruction, error) {
 instructions := []Instruction{}

 // iterate over lines in file and fill instructions slice
 for _, line := range lines {
  instruction, err := parseInstruction(line)
  if err != nil {
   return nil, err
  }
  instructions = append(instructions, *instruction)
 }
 return instructions, nil
}

func parseInstruction(line string) (*Instruction, error) {
 // split line

 // read direction and distance

 // return instruction
 return &Instruction{
  Direction: direction,
  Distance:  distance,
 }, nil
}
```

Solution

- end position: `19,20`
- most left position: `-3`
- most right position: `25`
- most visited position: `18,7` (5 times)

https://github.com/acend/go-basics-training-examples/tree/master/robot

9.4. Joke API

Task

Try and solve the following. Every task has links to relevant documentation.

API Client

1. Begin with the following main.go which requests a joke from https://official-joke-api.appspot.com/random_joke and decodes it into a Go struct.
2. Refactor the HTTP request into another function that returns `joke, error`
3. Update the code to also decode and print the punchline
4. Delay the punchline by a few seconds
5. Make the delay configurable with a `--delay 10` flag

We recommend using flag.IntVar , instead of flag.Int . Both variants work, but with flag.IntVar you end up with a normal variable, instead of a pointer . You may also try using flag.DurationVar .

To learn more about the `*int` type, see *2.4. Pointers*.

Do not forget to run flag.Parse() after defining the flags.

When solving the last task, you might receive the error: `mismatched types int and time.Duration`. This is because you cannot multiply different types. You must first convert the value to the same type.

```
i := 42
f := float64(i)
u := uint(f)
d := time.Duration(u)
```

For more information why `5 * time.Second` works see: https://stackoverflow.com/a/49498375

API Server

6. Write another program that reads a json file and picks a random joke. You will need to use slices
7. Implement a HTTP Server that serves the picked joke
8. Update your program from part one so that you can configure the URL (maybe as another flag) and query your own API

Add jokes

9. Extend your HTTP Server that you can send a joke to it which gets persisted (POST request)
10. Extend your program from part one that it supports adding new jokes. For this ask the user to enter a joke

on the command line (standard input) and send it to your own API.

With the bufio package we can read from the standard input (os.Stdin) until a certain character occurs:

```
line, err := bufio.NewReader(os.Stdin).ReadString('\n')
```

The strings package contains various functions to work with strings. For example strings.TrimSpace to remove whithspace characters (spaces, newlines, etc.) from strings.

```
line = strings.TrimSpace(line)
```

Solution

Client

Server

Full Combined

9.5. HTTP Client

Overview

We will write a small CLI tool that gets the follower count and the full name of a specific Github user.

Information about a Github user we can obtain from the Github REST API under `https://api.github.com/users/<user>`. Besides many other information a call to https://api.github.com/users/mitchellh returns the following information:

```
{
  "login": "mitchellh",
  ...
  "name": "Mitchell Hashimoto",
  ...
  "followers": 9973,
  ...
}
```

The Github API limits the number of unauthenticated requests per IP to 60 per hour. So it is likely that you get a HTTP response code of 429 (Too Many Requests) and no answer.

To send more requests from one IP you have to generate a personal access token (you don't have to select any permission/scope) and send it as part of the `Authorization` header:

```
Authorization: Bearer <your token>
```

Tasks

1. Create CLI tool `github-info` which takes a username as argument and prints the users full name and the number of followers.
2. Read the personal access token from the environment variable `GITHUB_TOKEN` and send it with the API request.
3. Github returns info about the ratelimiting in the HTTP headers of the response. Add an option `-debug` which shows the number of remaining requests (header `x-ratelimit-remaining`).

The result should look similar to this:

```
$ ./github-info mitchellh
name: Mitchell Hashimoto
followers: 9973
```

After task 3 with debug flag:

```
$ ./github-info -debug mitchellh
remaining requests: 48
name: Mitchell Hashimoto
followers: 9973
```

Tips

Check out the examples in *5.3. HTTP Client* and *5.2. JSON*.

- To get arguments from the command line see os or flag
- net/http
  - Especially take a look at http.Response because we have to check the HTTP status code and read information from the response headers.
- encoding/json

```
debug := false

flag.BoolVar(&debug, "debug", debug, "show additional information about rate limiting")

flag.Parse()

if flag.NArg() < 1 {
 // missing arugments
}

userName := flag.Arg(0)
```

Solution

https://github.com/acend/go-basics-training-examples/tree/master/github-info

9.6. HTTP Server

Part 1: Ping and Logger

Tasks

1. Run a server and implement a handler which returns `pong` on the endpoint `/ping`.
2. Create a middleware to log every request. Log the path, method, duration and the IP of the client of the request.

Your log should look similar to this:

```
2022/04/11 10:03:08 remote=192.168.1.143 path=/foo method=GET duration=13.765874ms
```

Tips

See *5.4. HTTP Server*.

- Consider using the log package from the standard library
- You can measure time using the time package:

```go
start := time.Now()

// perform action

duration := time.Now().Sub(start)
```

Part 2: User API

Tasks

Create a JSON REST API where you can create, list and delete users.

The API should provide the following endpoints:

- `POST /user` : create a user
- `GET /user` : returns all users
- `GET /user/<ID>` : returns a single user by ID
- `DELETE /user/<ID>` : delete user by ID

The user should look like this:

```
{
    "name": "john",
    "full_name": "John Doe",
    "followers": 13
}
```

You can decide how you would like to store the users. One option would be to store them only in-memory for example in a map `map[int]User`. Another option would be to store serialize the users into a file (e.g. with JSON).

Tips

See *5.4. HTTP Server*.

Part 3: User API Client

Tasks

Create CLI tool to list, create and delete users:

```
# create user
./usercli create bob
./usercli create bob "Bob Meier"
./usercli create bob "Bob Meier" 34

# list all users
./usercli get

# show specific user
./usercli get bob

# delete user
./usercli delete bob
```

Tips

Take a look at the github-info-client example to get an idea on how to implement this.

Solutions

- Part 1: Ping and Log Middleware: https://github.com/acend/go-basics-training-examples/tree/master/http-server
- Part 2 + 3: User API: https://github.com/acend/go-basics-training-examples/tree/master/user-api