Contents

Chapter 1

Chapter one

Go code is organised into packages, which are similar to libraries or modules in other languages. A package consists of one or more `.go` source files in a single directory that define what the packages does.

Each source file begins with a `package` declaration, here `package main` that states which package the file belongs to, followed by a list of other packages that it imports, and then the declarations of the program that are stored in that file.

the `fmt` package contains functions for printing formatted output and scanning input. `Println` is one of the basic output functions in `fmt`; it prints one or more values, seperated by spaces, with a newline character at the end so that the values appear as a single line of output.

Package `main` is special. It defines a standalone executable program, not a library. Within package main the *function* main is also special – it's where execution of the program begins. Whatever `main` does is what the program does. ofcourse, `main` will normally call upon functions in other packages to do much of the work, such as function `fmt.Println`.

Command Line Arguments

The Variable `os.Args` is a *slice* of strings. Slices are a fundamental notion in Go. A slice is a dynamically sized sequence `s` of array of elements where individual elements can be accessed by `s[i]` and a contiguous subsequence as `s[m:n]`

The number of elements is given by `len(s)`.

The first element of `os.Args`, `os.Args[0]`, is the name of the command itself; The other elements are arguments that were presented to the program when it started execution

```
for _, arg := range os.Args[1: ] {
    s += sep + arg
    sep = " "
}
fmt.Println(s)
```

here range produces two values, `index` and `value` of the element at that index so arg handles `value` and `_` handles the index

each time around the loop, the string `s` gets completely new contents. The `+=` statement makes a new string by concatenating the old string, a space character, and the next argument, then assigns the new string to s. The old contents of `s` are no longer in use, so they will be garbage-collected in due course.

If the amount of data involved is large, this could be costly. A simple and more efficient solution would be to use `Join` function from the `strings` package

```
func main() {
    fmt.Println(strings.Join(os.Args[1:], " "))
}
```

Variables

The version of strings above uses short variable declaration There are several other ways to declare a variable in go

```
s := ""
var s = ""
var s string
var s string = ""
```

Chapter 2

Finding Duplicate Lines

finding duplicate lines is partly inspired from `uniq` command, which looks for adjacent duplicate lines. The structures and packages used are models that can be easily adapted.

A *map* holds a set of key / value pairs and provides constant - time operations to store, retrieve, or test for an item in the set.

`a map is like a dictionary in python ?`

The key may be of any type whose values can be compared with `==`, strings being the most common example; The value may be of any type at all. In the example, the keys are strings and the values are `ints`. The built-in function `make` creates a new empty map;

The program uses a short variable declaration to create a new variable `input` that refers to a `bufio.Scanner`

```
input := bufio.NewScanner(os.Stdin)
```

The scanner reads from the program's standard input. Each call to `input.Scan()` reads the next line and removes the newline character from the end; the result can be retrieved by calling `input.Text()` The `Scan` function returns `true` if there is a line and `false` when there is no more input.

Difference between quoted string and a string in go

In quotes "" you need to esacpe new lines, tabs and other characters that do not need to be escaped in backticks ``. If you put a line break in a backtick string, it is interpreted as a `'\n'`

String Literals

A string literal represents a `string constant` obtained from concatenating a sequence of characters. There are two forms: 1. raw string literals 2. interpreted string literals

Raw string literals are character sequences between back quotes, as in `` `foo` ``. Within the quotes, any character may appear except back quote.

The value of a raw string literal is the string composed of the uninterpreted (implicitly UTF-8 encoded.) characters between the quotes; in particular, backslashes have no special meaning and the string may contain new lines. Carriage return characters (`'\r'`) inside raw string literals are discarded from the raw string value.

Interpreted string literals are character sequences between double quotes, as in `"bar"`. Within the quotes, any character may appear except newline and unescaped double quote. The text between the quotes forms the value of the literal, with backslash escapes interpreted as they are in `rune literals`.

```
string_lit             = raw_string_lit | interpreted_string_lit .
raw_string_lit         = "`" { unicode_char | newline } "`" .
interpreted_string_lit = `"` { unicode_value | byte_value } `"` .
```

```
`abc`                // same as "abc"
`\n
\n`                  // same as "\\n\n\\n"
"\n"
"\""                 // same as `"`
"Hello, world!\n"
"  "
"\u65e5 \U00008a9e"
"\xff\u00FF"
"\uD800"             // illegal: surrogate half
"\U00110000"         // illegal: invalid Unicode code point
```

Maps in Go

Go provides a built-in map type that implements as hash table.

Declaration and initialization

A Go map type looks like this:

`map[KeyType]ValueType`

where `KeyType` may be anytype that is *comparable* and `ValueType` may be any type at all, including another map.

```
var m map[string]int
```

Map types are reference types, like pointers or slices, and so the values of `m` above is nil; it doesn't point to an initialized map. A `nil` map behaves like an empty map when reading, but attempts to write to `nil` map will cause a runtime panic; don't do that. To initialize a map, use the built in `make` function:

```
m = make(map[string]int)
```

The `make` function allocates and initializes a hash map data structure and returns a map value that points to it. The specifics of that data structure are an implementation detail of the runtime and are not specified by the language itself.

```
for key, value := range m {
    fmt.Println("Key:", key, "Value:", value)
}
```

Exploiting zero values

It can be convenient that a map retrieval yields a zero value when the key is not present.

For instance, a map of boolean values can be used as a set-like data structure (recall that the zero value for the boolean type is `false`). This example traverses a linked list of `Nodes` and prints their values. It uses a map of `Node` pointers to detect cycles in the list.

```
type Node struct {
    Next *Node
    Value interface{} // *Interfaces* are named collections of method signatures.
}
var first *Node
```

```
visited := make(map[*Node]bool)
for n := first; n != nil; n = n.Next {
    if visited[n] {
        fmt.Println("cycle detected")
        break
    }
    visited[n] = true
    fmt.Println(n.Value)
}
```

Interfaces

Interfaces are named collections of method signatures.

```
type geometry interface {
    area() float64
    perim() float64
}
```

jordan orelli's blogpost

An interface is two things: it is a set of methods, but it is also a type.

Typically, we're introduced to interfaces with some contrived example. Let's go with the contrived example of writing some application where you're defining `Animal` datatypes, because that's a totally realistic situation that happens all the time.

The `Animal` type will be an interface, and we'll define an `Animal` as being *anything that can speak*. This is a core concept in Go's type system; instead of designing our abstractions in terms of what kind of data our types can hold, we design our abstractions in terms of what actions our types can execute.

we start by defining our `Animal` interface:

```
type Animal interface {
    Speak() string
}
```

pretty simple: we define an `Animal` as being any type that has a method named `Speak`. The Speak method takes no arguments and returns a string. Any type that defines this method is said to satisfy the `Animal` interface. There is no `implements` keyword in Go; whether or not a type satisfies an interface is determined automatically.

```
type Dog struct {
}

func (d Dog) Speak() string {
    return "Woof!"
}

type Cat struct {
}

func (c Cat) Speak() string {
    return "Meow!"
}

type Llama struct {
```

```
}

func (l Llama) Speak() string {
    return "?????"
}

type JavaProgrammer struct {
}

func (j JavaProgrammer) Speak() string {
    return "Design patterns!"
}
```

We now have four different types of animals: A dog, a cat, a llama, and a Java programmer. In our main() function, we can create a slice of Animals, and put one of each type into that slice, and see what each animal says. Let's do that now:

```
func main() {
    animals := []Animal{Dog{}, Cat{}, Llama{}, JavaProgrammer{}}
    for _, animal := range animals {
        fmt.Println(animal.Speak())
    }
}
```

The `interface{}` type

The `interface{}` type, the empty interface, is the source of much confusion. The `interface{}` type is the interface that has no methods. Since there is no implements keyword, all types implement at least zero methods, and satisfying an interface is done automatically, all types satisfy the empty interface. That means that if you write a function that takes an interface{} value as a parameter, you can supply that function with any value. So, this function:

```
func DoSomething(v interface{}) {
   // ...
}
```

will accept any parameter whatsoever.

Here's where it gets confusing: inside of the `DoSomething` function, what is v's type? Beginner gophers are led to believe that "v is of any type", but that is wrong. v is not of any type; it is of `interface{}` type. Wait, what? When passing a value into the DoSomething function, the Go runtime will perform a type conversion (if necessary), and convert the value to an interface{} value. All values have exactly one type at runtime, and v's one static type is `interface{}`.

Russ Cox Blog

Chapter 3

Getting Started

- place the source files `file_name.go` in src directories

Chapter 4

GoNotes

Understanding Go Programming Language

types in go:

uint16 // min 0 max 65535 int16 // -32k –> 32k

```
func add(x float64, y float64) float64 {
    return x + y
}
```

can also be written as

```
func add(x, y float64) float64 {
    return x + y
}
```

```
var num1, num2 float64 = 5.6, 9.5
// short circuit declaration
```

inorder to return multiple elements from a function

```
func return_multiple_items(a, b string) (string, string) {
    return a, b
}
```

Chapter 5

pointers in go

```
func main() {
    x := 15
    a := &x // memory address

    fmt.Println(a)

    // print the value of x
    fmt.Println(*a)

    *a = 5
    fmt.Println(x)
    fmt.Println(a)
}
```

Chapter 6

Basics of empty web tutorial

- http.HandleFunc
- http.ListenAndServe http.ListenAndServe –> (takes_port_number)
- http.ResponseWriter
- *http.Request

Chapter 7

Structs

`structs` are the basic building blocks in go

```
type <struct_name> struct {
    // attributes
}

// usage
myStruct := <struct_name> {
    // attribute_values
}

// accessing
myStruct.attribute_value
```

methods in go

- value receivers receive values and perform calculations

```
func (<struct_attribute_name> <struct_name>) <method_name> <return_value> {
    return ...
}
```

- pointer receivers if we want to modify values in the struct we need pointer receivers

```
func (<struct_attribute_name> *<struct_name>) <method_name> <return_value> {
    return ...
}
```

Should I define methods on values or pointers ?

```
func (s *MyStruct) pointerMethod() { } // method on pointer

func (s MyStruc) valueMethod() { } // method on value
```

When defining a method on a type, the receiver (s in the above example) behaves exactly as if it were an argument to the method.

Whether to define the receiver as a value or as a pointer is the same question, then, as whether a function argument should be a value or a pointer.

First, and most important, does the method need to modify the receiver? if it does, the receiver *must* be a pointer.

(Slices and maps acts as references, so their story is a little more subtle, but for instance to change the length of a slice in a method the receiver must still be a pointer.)

In the examples above, if pointerMethod modifies the fields of s, the caller will see those changes, but valueMethod is called with a copy of the caller's argument (that's the definition of passing a value), so changes it makes will be invisible to the caller.

Second is the consideration of efficiency. if the receiver is large, a big `struct` for instance, it will be much cheaper to use a pointer receiver.

Next is consistency. If some of the methods of the type must have pointer receivers, the rest should too, so the method set is consistent regardless of how the type is used.

For types such as basic types, slices, and small structs, a value receiver is very cheap so unless the semantics of the method requires a pointer, a value receiver is efficient and clear

arrays and slices

[5 5]int == array []int == slice

Chapter 8

looping structures

```
for i:=0;i<10;i++ {
    //body of for
}
```