Modules

Packages and modules in depth

A package is a way to organise and reuse code. It's a coll^n of related go source files that are grouped together under a single package name. Each go file has package declaration at start that specifies the package it belongs to. ex → utils package for utility fns.

Module is a collection of related packages. It is defined by a go.mod file, which contains the module path and dependencies it requires.

Go expects all files in a directory to belong to the same package. Go compiles all files in a directory together.

Lexer

Lexer (Indirectly) checks the code for correctness of grammar of the language. In go, it puts semicolon at the end of statements. So, we don't need to do it manually. But in for loops and so other parts where it is part of definition, we've to put it manually. Lexer converts sequence of characters into a sequence of tokens which follows a specific regular expression.

Types

→ Case sensitive; almost
→ Variable type should be known in advance
→ Everything is a Type.

- String
- Bool
- Integer → uint8 uint64 int8 int64 uintptr
- float → float32 float64
- complex

More types

Array
Slices
Maps
Structs
Pointers

Almost everything
functions · Channels

Variables

To declare a variable :-

var name-of-variable variable-type

ex:-
var name string

%T in Printf → for data types

string → string
boolean → bool
int → uint 8, uint16, uint 32, uint 64, int8, int16, int32, int64
byte = uint8, rune = int32
float → float32, float64
complex → complex32, complex64

Implicit type

We can just write var variable-name = value and lexer will figure out and fix that type for us.

No var

variable-name := value also works
↳ It is allowed only inside of functions or structures or other stuffs
Not for global variables.

Constants

const variable-name type = value

Taking input

bufio package → provide an option to take input of data from the user or a lot of other input sources. It does have a lot of other functionalities but we are focusing on Reader for now.

variable_name := bufio.NewReader(os.Stdin)

- variable_name: where we want to store the data
- bufio.NewReader: the reader fn of bufio
- os.Stdin: specifies from where we are going to provide input

input := variable_name.ReadString('\n')

↳ specifies till when we're going to read a string.

Comma ok Syntax

There is no concept of try-catch in go lang. So, we have to check for errors. Often we use comma ok syntax to mimic try catch.

Ex.

```
input, err := reader.ReadString('\n')
if input {
    fmt.Println(input)
}
fmt.Println("error")
```

↳ It returns two values → value and error. Error is always nil unless input does not end with the delimiter specified.

↳ we can also handle errors in more standard manner using errors package.

Conversions

we make use of strconv.Parse type function of strconv package. It takes two values → string value that you want to convert and no. of bits. It also returns two values → converted value and error. Error is nil unless conversion was not possible due to bad input.

Ex.

```
num, err := strconv.ParseFloat(input, 32 or 64)
if err != nil {
    fmt.Println(err)
} else {
    fmt.Println("Adding 1 to your value", num+1)
}
```

Time Package

To get current time, use time.Now() and store it somewhere to format it or for other uses.

eg.

current-time = time.Now()

To format, use,

current-time.Format("01-02-2006 15:04:05 Monday")

use this specific date and time

Create a new date using,

time.Date(year, monday, day, ... any other stuff)

Pointers

Pointers store memory addresses. These are used to refer to particular address. Use case involve changing values at exact memory addresses rather than creating a copy of it.

Declaring a pointer

var ptr *int

↳ If address/reference is not provided, it stores nil.

num := 5

var ptr = &num

ptr stores address of num.

and *ptr has value stored at num's memory address = 5

If we do, *ptr = *ptr + 2,

num will become 7

So, it is used for pointing memory address of variables to do direct changes

Arrays

Declaration:

var values [size] type or var values = [size] type {value1, value2, ...}

values[0] = value1

⋮

Ex:

var fruits [4] string

fruits[0] = "mango"

fruits[1] = "Apple"

↳ fmt.Println(fruits) → [mango Apple ____]

↓ spaces are provided to indicate that empty places are there in the array

If we do len(fruits)

↳ we expect to get 2 but golang gives 4, because of predetermined/reserved places at the time of array variable formation.

*** Slices (Dynamic Arrays with lots of inbuilt functionalities)

var values = [] int {2, 5, 6, 7}

↳ we don't need to provide a specific size to slices.

To add more content:

values = append (values, 5, 6, 7)

↳ it will add 5, 6, 7 at end of values slice

values = [2, 5, 6, 7, 5, 6, 7]

To delete/create slices of the array, we can use the append function more smartly.

values = append (values[1:]) → values from index 1 to end

values = append(values[1:4])
↳ values from 1 to 3 (1 to 3 indexed values)

sort.Ints(values)
↳ will sort the array

sort.IntsAreSorted(values) → checks if array is sorted or not.

Slice's declaration using
var values = make([]int, size)
Shows unique behaviour.
If we try adding contents using values[index] = value then it will be allowed only till the size specified. But Slices are dynamic and adding content to it should not be paused. So, append method comes into here once again. What it does is reallocation of the Slice to include more memory. So that everything can be accommodated.

How to remove a value from a Slice using index?

use append

var values = append(values[:index], values[index+1:]...) (don't forget to add ...)

Maps

values := make(map[key_type]value_type)
values[key] = value
Ex
values = make(map[int]int)
values[5] = 10
values[10] = 20
If we print values, we get the map list containing key value pairs

Deleting values

delete (values, key)

Ex delete (values, 5)

```
for key, value := range values {
    fmt.Println("For key %v, value is %v", key, value)
}
```

Structs

type literal word Struct_name struct {

```
    Name : string
    Age : int
}
```

Make first letter as capital is equivalent to making any variable or func or anything publically available (similar to export)

It has to be written outside functions for global access.

In any fn where we want to use it, write →

```
User1 := User{"User1", 18}
Println(User1) → {User1 18}
Printf("%+v", User1) → {Name:User1 Age:18}
```

Control flows

If else

```
if cond {
} else if cond {
} else {
}
```

we can also declare variables inside if too.

Ex

```
if num := 3; num < 10 {
    fmt.Println("Num is less than 10")
} else {
```

Switch Case

```
switch variable_name {
case value1:
    //do sth
case value2:
    //do sth

:

default:
    // do sth else
}
```

we don't need to put default after each case.

If we need to forcefully go one step down after getting a value. use fallthrough

Loops, Break, continue

for loop and its variations

values := []int{5,10,15}

there is no (++i) in go

```
1. for i:=0 ; i < len(values; i++ {
   Println(values[i])
   }

2. for i := range values {
     values[i]   // if we provide only one value i then it is
   }             considered as index

3. for index, value := range values {
     Printf("index is %v and value is %v", index, value)
   }

   some_value := 1
4. for some_value < 10 {
     Println(some_value)
   } some_value++
```

Break

```
for some_value < 10 {
    if some_value == 5 { break }
                breaks through the loop
}
```

continue

```
for some_value < 10 {
    if some_value == 5 {
        some_value++
        continue
    }
    // do stuff
}
```

→ skips only the part where condition matches

goto

```
a:
goto a
```

functions

```
func function_name (argument 1 type 1, argument 2 Type 2,...)
(return_value_1_type, return_value_2_type) {
    return val1, val2....
}
```

Ex

```
func add (num1 int, num2 int) int {
    return num1 + num2
}
```

Ex.

```
func adder (values ...int) (int, string) {
    total := 0
    for index, value := range values {
        total += value
    }
    return total, "Result value"
}
```

Methods

In other language, methods are functions defined inside classes. Since there is no concept of classes here, we are going to use struct.

```
type User struct {
    Name string
    Age int
}

func (u User) status() {
    if name != nil {
        Println("status OK", u.name)
    } else {
        Println("NOT OK")
    }
}
```

→ u User → would be enough to make the function status a method of User struct

If we try to update any struct value in one of its methods, make to send the pointer to address rather than direct u User, as it only makes changes by creating a new copy of it.

Defer

defer keyword is used to stop executing the code line by line whichever func has defer in front of it. It will be executed at the end.

In case, a lot of functions are deferred then LIFO order is followed to execute them.

File handling in go

Creating a file :-

```
file, err = os.Create("./file.txt")
```

→ location of file

writing to a file :-

```
content := "How are you"
length, err = io.WriteString(file, content)
```

Reading a file :-

```
data byte, err := ioutil.ReadFile(filename)
```