HOW TO GO!

https://go.dev/doc/tutorial/getting-started

https://www.javatpoint.com/go-tutorial

https://www.geeksforgeeks.org/golang/

https://gobyexample.com/

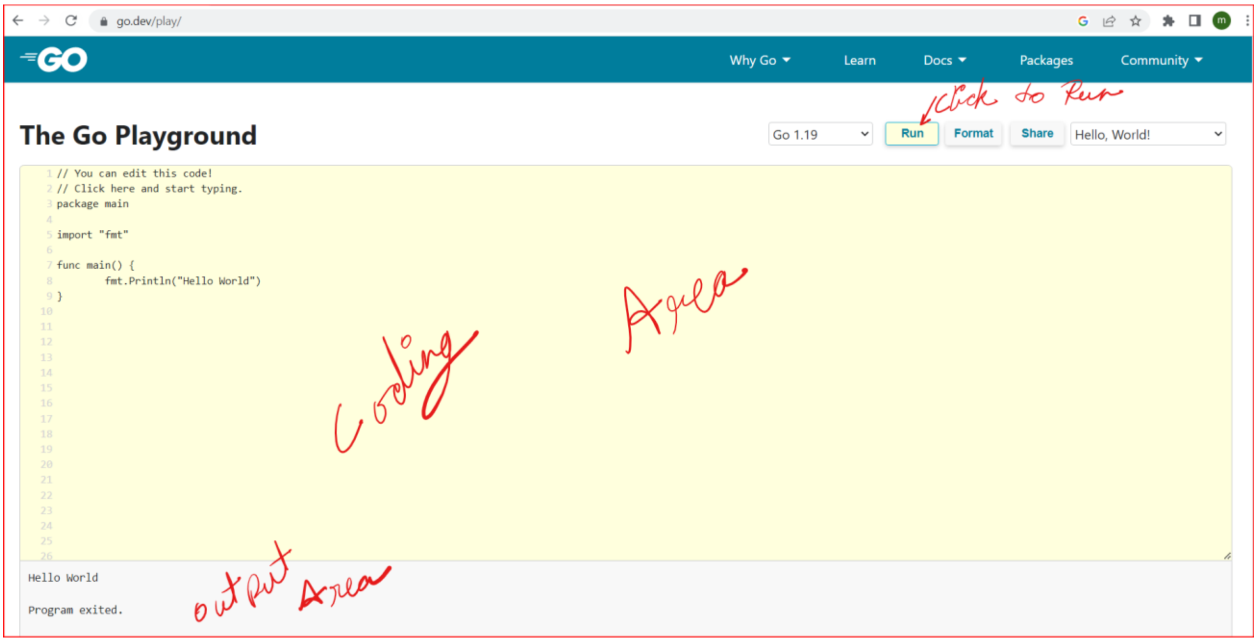
Prerequisite

* Basic knowledge of programming.

* Computer / Laptop.

GO play ground -> https://go.dev/play/

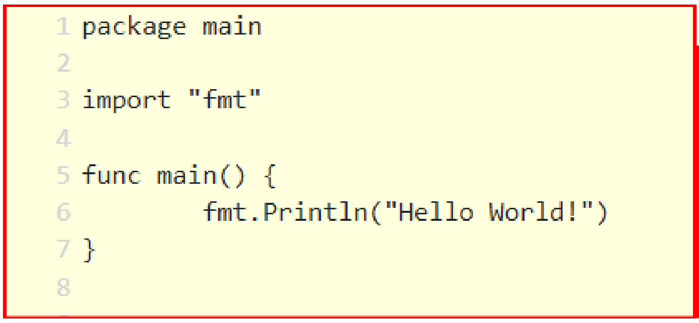
Go playground is an online space where we can code and learn GO (GOLANG).



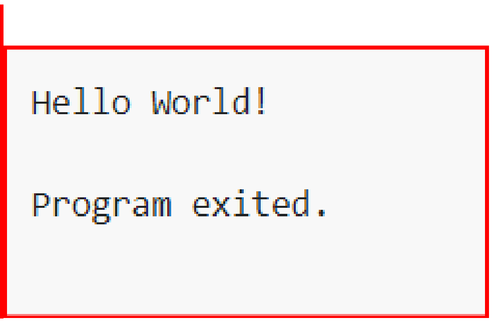
```
// You can edit this code!
// Click here and start typing.
package main

import "fmt"

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

```
Hello World

Program exited.
```

A Sample Program to “Hello World!”



```
package main

import "fmt"

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

```



```
Hello World!

Program exited.
```

Understanding the Syntax:

1. First Line must be the package name.
2. Followed by imports.
3. And then rest of the code.

What is package and how to write: -

In Golang each piece of code belongs to some package.

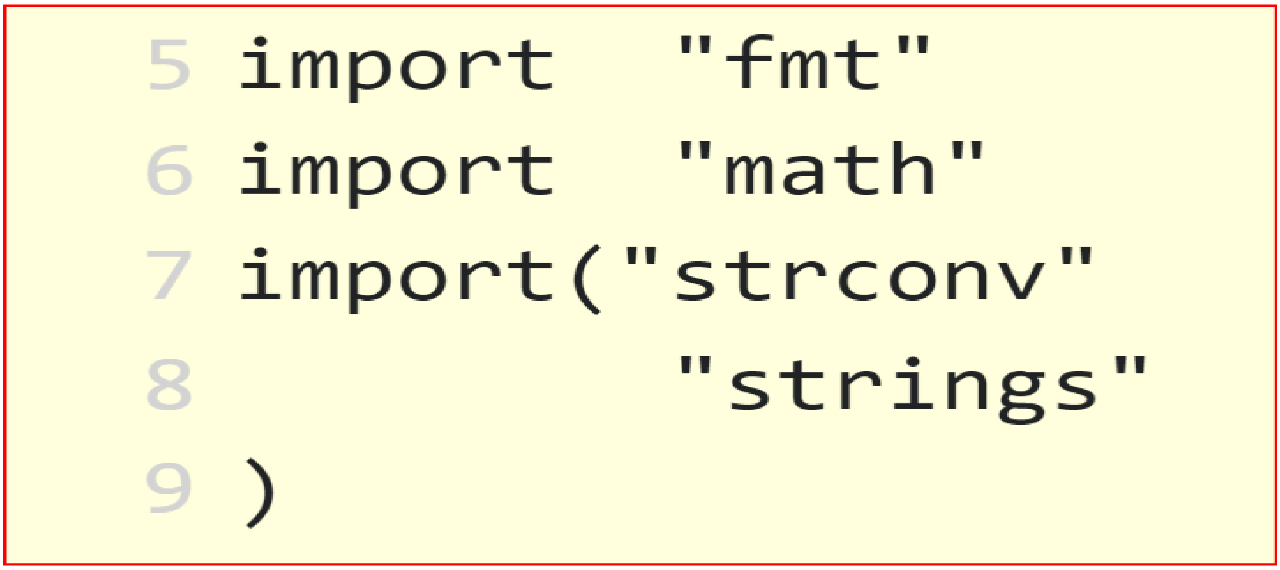
The purpose of a package is to design and maintain a large number of programs by grouping related features together into single units so that they can be easy to maintain and understand and independent of the other package programs. (link)

```
package <package_name>

<package_name> can be any thing of your choice
e.g.,
package main,
package constant
```

What is import and how to write: -

Import is used to make code in one package available in another.

```
import “<package_name>”
import “<package_name>”

import ( “<package_name>”
         “<package_name>”
         …
         )
```



```
import   "fmt"
import   "math"
import("strconv"
         "strings"
)
```

What is function and how to write: -

A function is a group of statements that together perform a task.(link)

1.

```
func <function_name>(<ipn> <ipt>){
}
```

2.

```
func <function_name>(<ipn> <ipt>) <rpt> {
}
```

3.

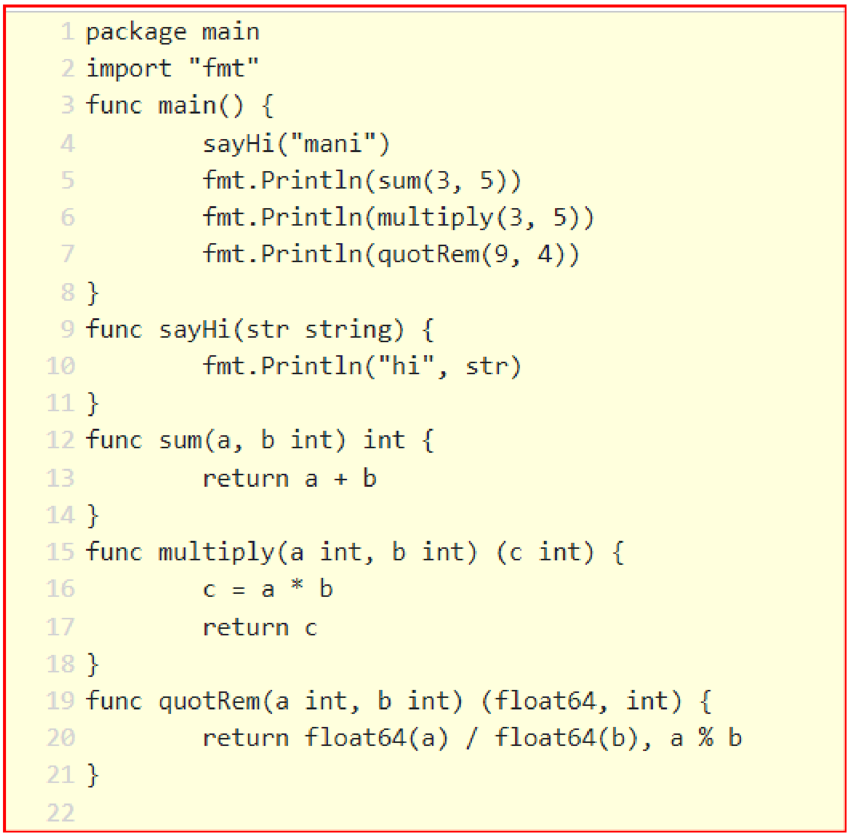
```
func <function_name>(<ipn> <ipt>, <ipn> <ipt>, <ipn> <ipt>) (<rpt>,<rpt>,<rpt>) {
}
```

4.

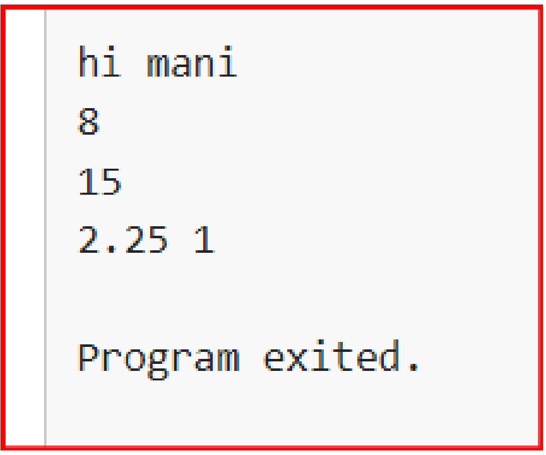
```
func <function_name>(<ipn> <ipt>, <ipn> <ipt>) (<rpn> <rpt>, <rpn> <rpt>) {
}
```

---

ipn => input_parameter_name
ipt => input _parameter_type
rpn => return_parameter_name
rpt => return_parameter_type

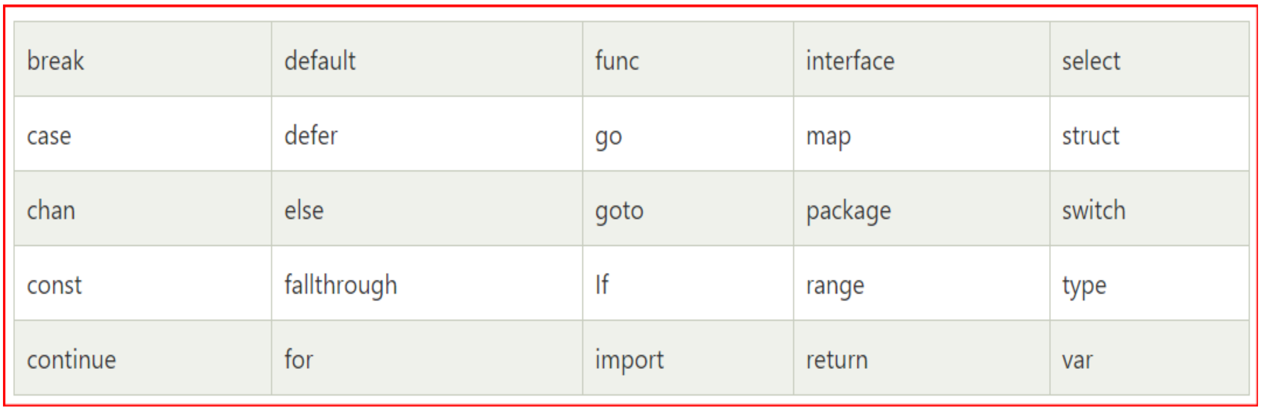


```go
package main
import "fmt"
func main() {
        sayHi("mani")
        fmt.Println(sum(3, 5))
        fmt.Println(multiply(3, 5))
        fmt.Println(quotRem(9, 4))
}
func sayHi(str string) {
        fmt.Println("hi", str)
}
func sum(a, b int) int {
        return a + b
}
func multiply(a int, b int) (c int) {
        c = a * b
        return c
}
func quotRem(a int, b int) (float64, int) {
        return float64(a) / float64(b), a % b
}

```

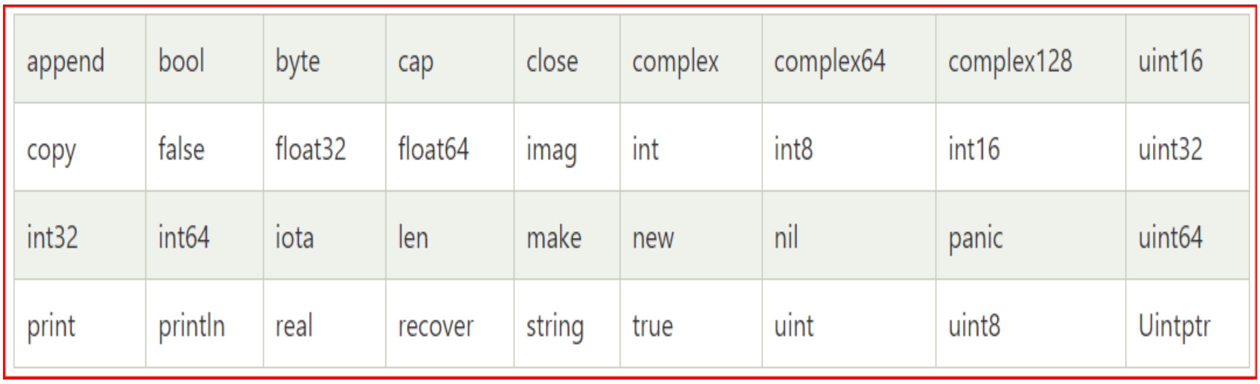


```
hi mani
8
15
2.25 1

Program exited.
```

Keywords in Golang: -



break	default	func	interface	select
case	defer	go	map	struct
chan	else	goto	package	switch
const	fallthrough	If	range	type
continue	for	import	return	var



append	bool	byte	cap	close	complex	complex64	complex128	uint16
copy	false	float32	float64	imag	int	int8	int16	uint32
int32	int64	iota	len	make	new	nil	panic	uint64
print	println	real	recover	string	true	uint	uint8	Uintptr

Variable declaration in Golang: -

1.

var <variable_name> <variable_type>

var num1 int

var num1, num2 int

2.

var <variable_name> <variable_type> = <value>

var num1,num2 int = 4,5

3.

<variable_name> := <value>

num1 := 4

num1,num2 :=4,5

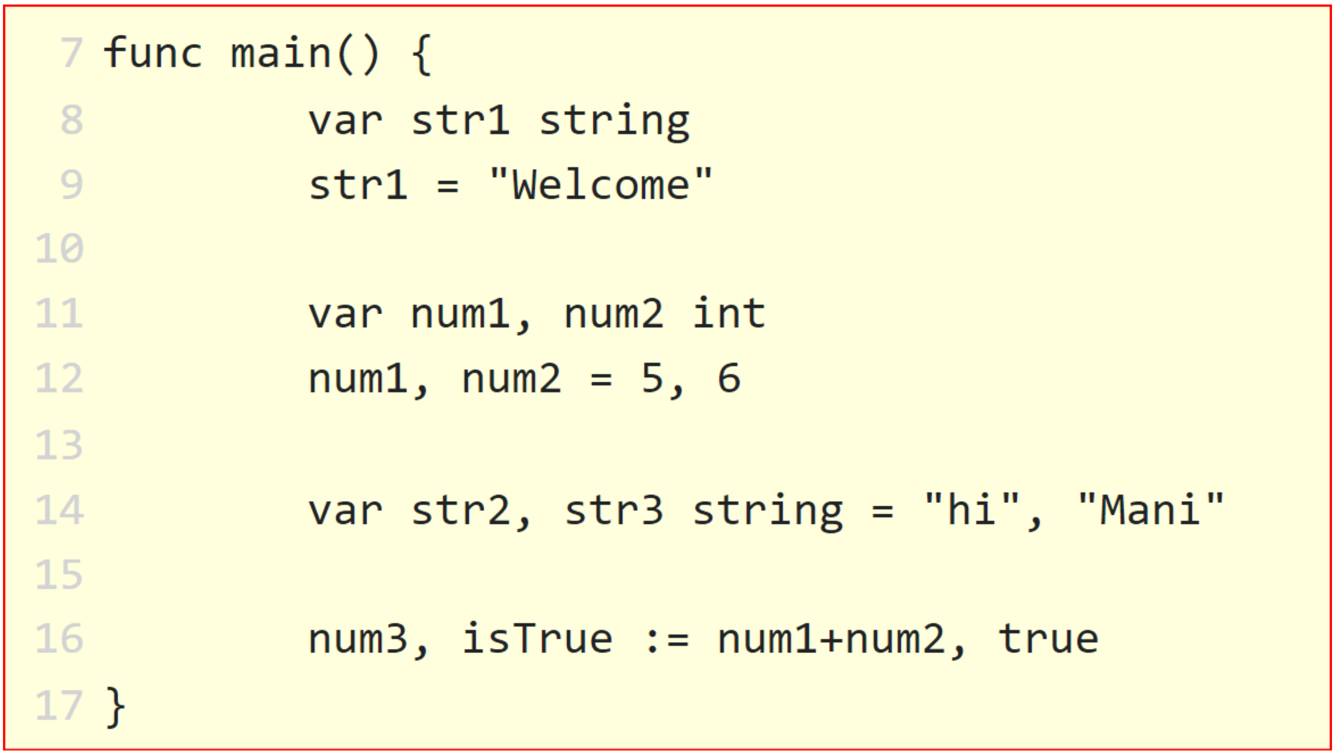
---

* We can’t declare two different kind of variable in same line using example 1 & 2 but can do with example 3.

Constants

const <variable_name> = <value>

const <variable_name> <variable_type> = <value>



```
func main() {
        var str1 string
        str1 = "Welcome"

        var num1, num2 int
        num1, num2 = 5, 6

        var str2, str3 string = "hi", "Mani"

        num3, isTrue := num1+num2, true
}
```

Controls

if-else

1. *only if*

   if <condition> {

   }

   if <condition1> <logical_operator> <condition2>{

   }

   * Logical_operator => &&, ||

---

2. *if else*

   if <condition>{

   } else{

   }

---

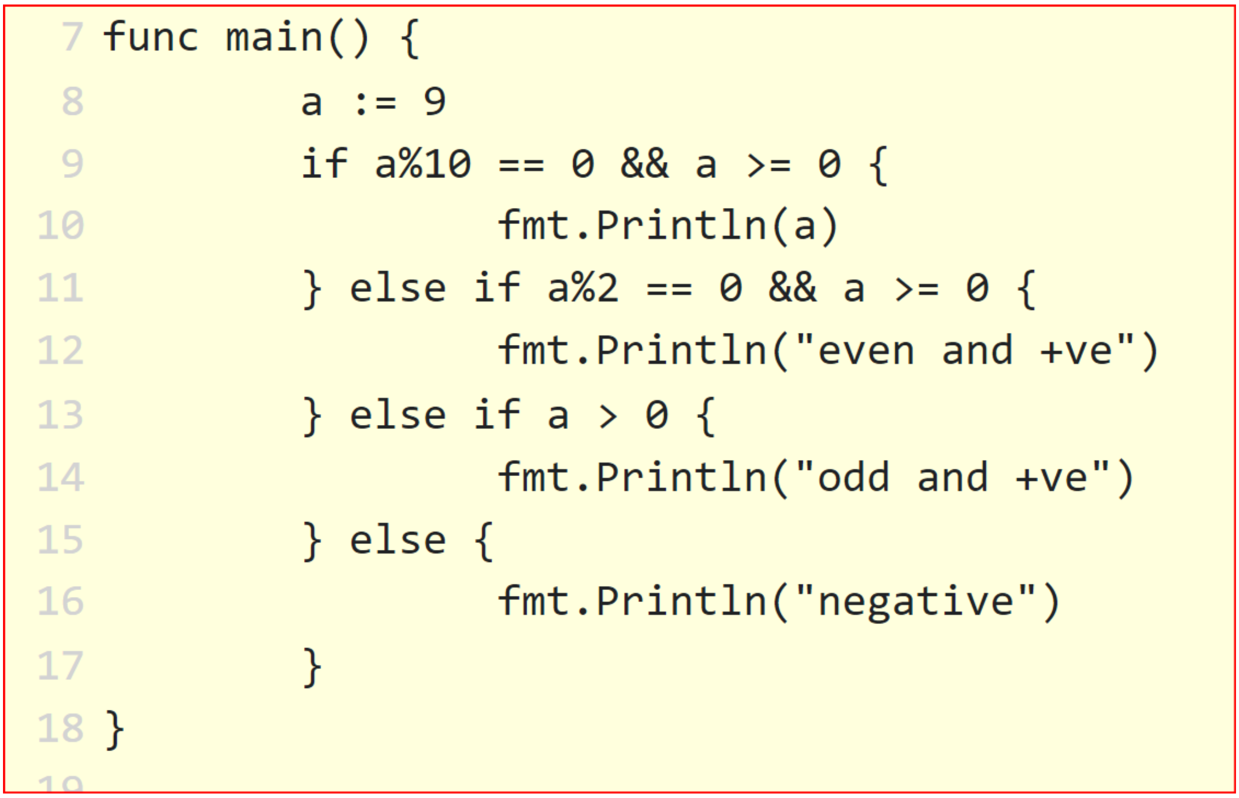
3. *if else - ladder/chain*

   if <condition>{

   } else if <condition>{

   } else{

   }



```go
func main() {
	a := 9
	if a%10 == 0 && a >= 0 {
		fmt.Println(a)
	} else if a%2 == 0 && a >= 0 {
		fmt.Println("even and +ve")
	} else if a > 0 {
		fmt.Println("odd and +ve")
	} else {
		fmt.Println("negative")
	}
}
```

Loop

for loop

```
for <variable declaration/assignment> ; <condition>; <operation> {


}


for <condition> {


}
```

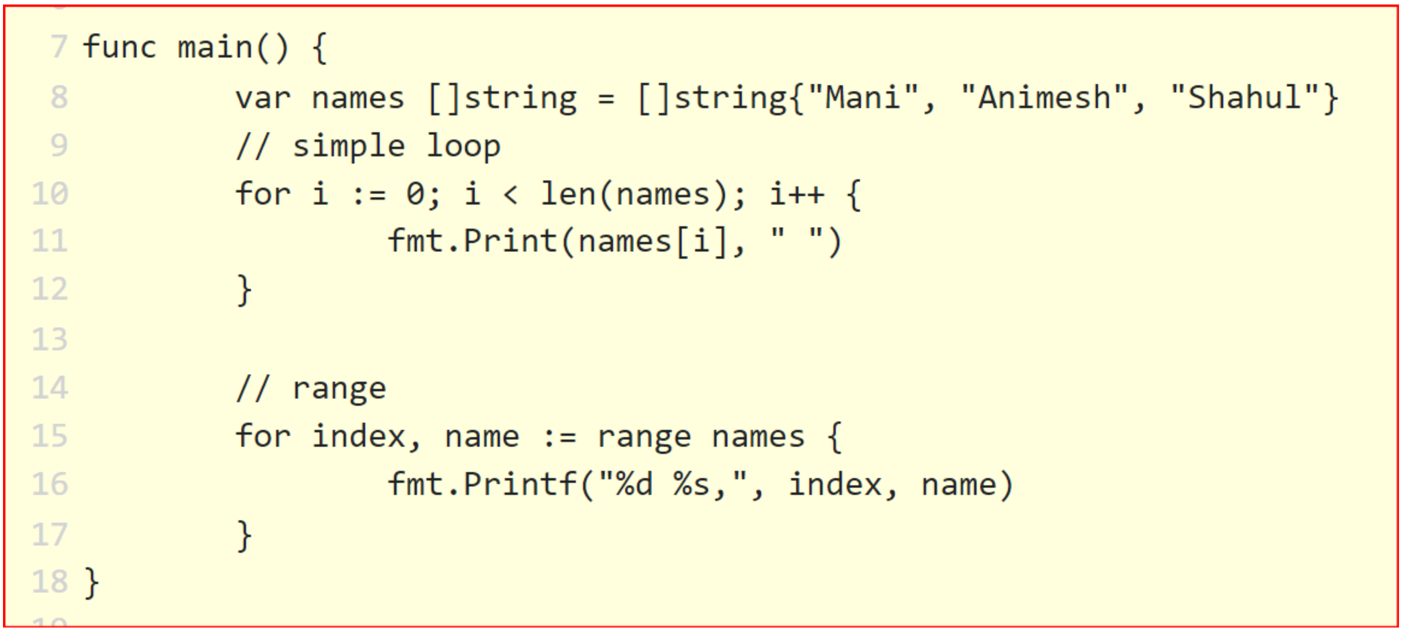
---

* There is no while and do while loop in Golang (And which make us realize that you can do any thing with for loop and some conditions).

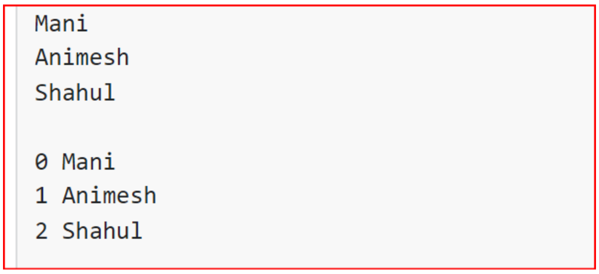
---

for range

```
var datas [5]int

for <index>, <value> := range data{


}
```



```
func main() {
        var names []string = []string{"Mani", "Animesh", "Shahul"}
        // simple loop
        for i := 0; i < len(names); i++ {
                fmt.Print(names[i], " ")
        }

        // range
        for index, name := range names {
                fmt.Printf("%d %s,", index, name)
        }
}
```

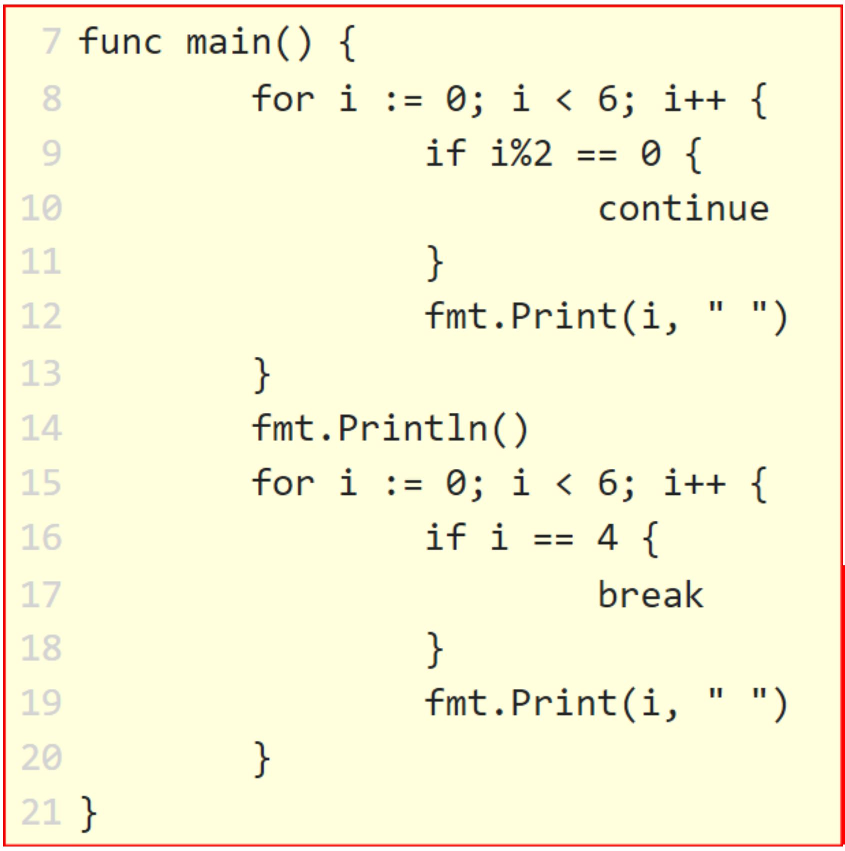


```
Mani
Animesh
Shahul

0 Mani
1 Animesh
2 Shahul
```

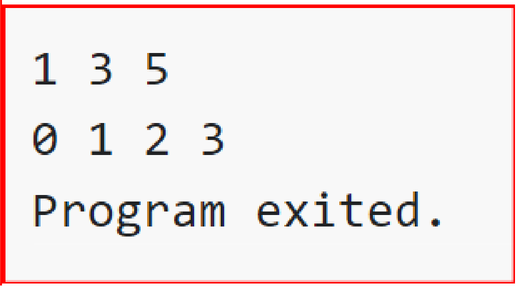
Continue/ Break

Continue is used just similar to other language to skip code.

Break is used to break out of loop.



```go
func main() {
        for i := 0; i < 6; i++ {
                if i%2 == 0 {
                        continue
                }
                fmt.Print(i, " ")
        }
        fmt.Println()
        for i := 0; i < 6; i++ {
                if i == 4 {
                        break
                }
                fmt.Print(i, " ")
        }
}
```



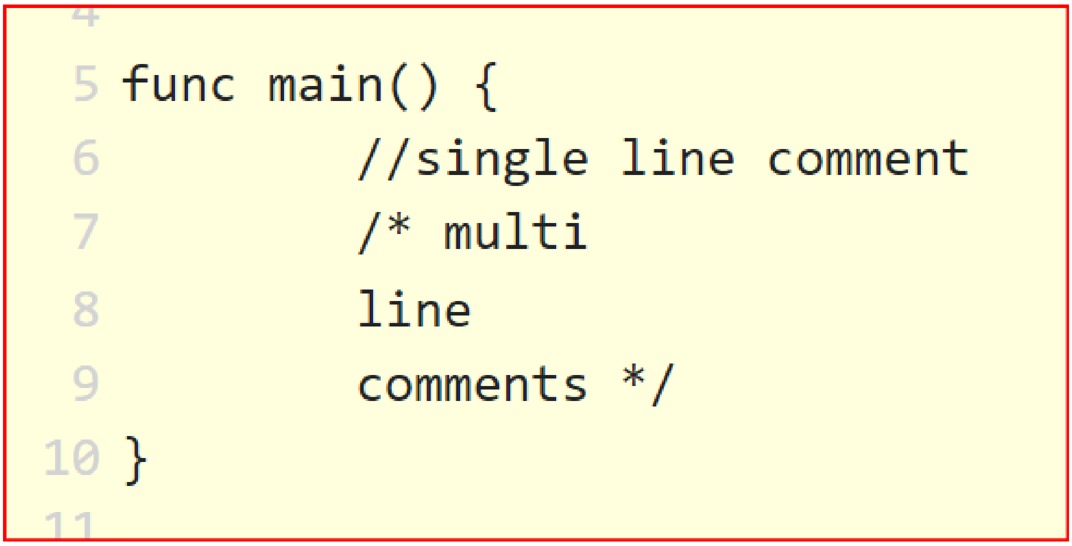
```
1 3 5
0 1 2 3
Program exited.
```

Comments

// single line comments

/* multi line

Comments*/



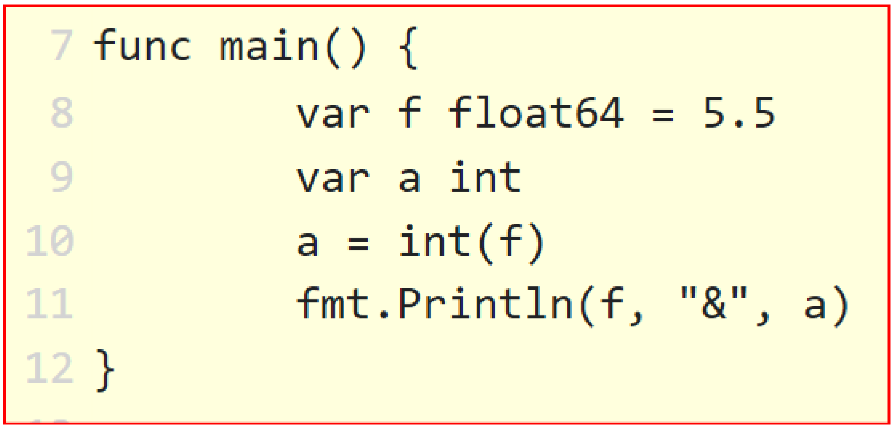
```go
func main() {
        //single line comment
        /* multi
        line
        comments */
}
```

Type Casting

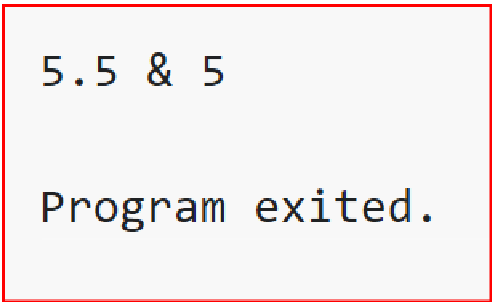
var <variable_name> <target_type>(<input_variable>)

val := 5.5

v := int(val)



```
func main() {
        var f float64 = 5.5
        var a int
        a = int(f)
        fmt.Println(f, "&", a)
}
```



```
5.5 & 5

Program exited.
```

Switch Case

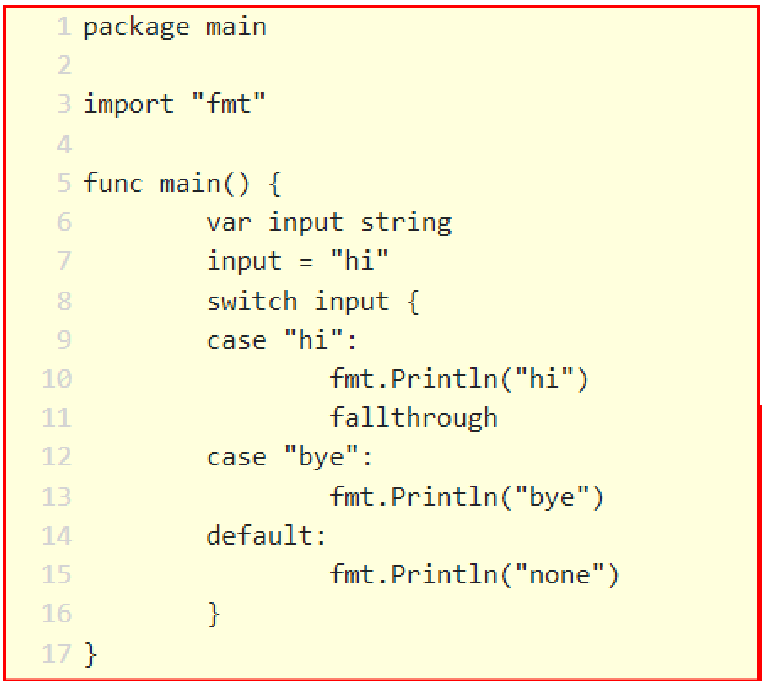
switch (<input>){

case <val1>:

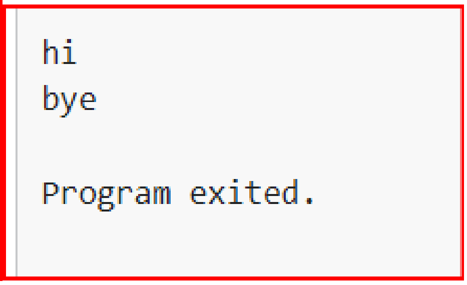
fallthrough

case <val2>:

default:



```
package main

import "fmt"

func main() {
        var input string
        input = "hi"
        switch input {
        case "hi":
                fmt.Println("hi")
                fallthrough
        case "bye":
                fmt.Println("bye")
        default:
                fmt.Println("none")
        }
}
```

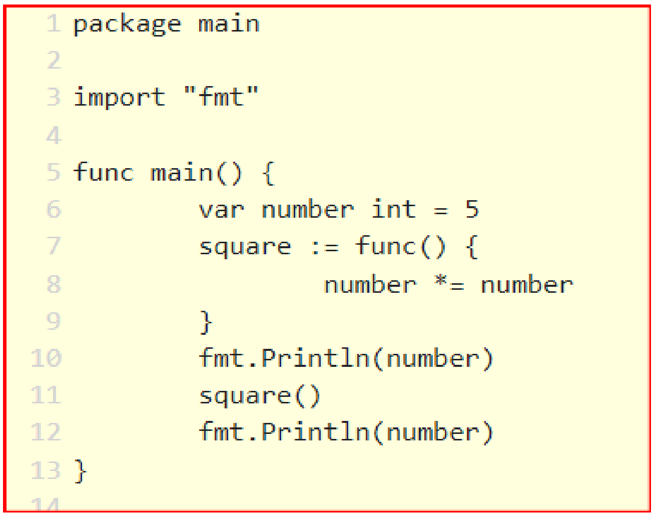


```
hi
bye

Program exited.
```

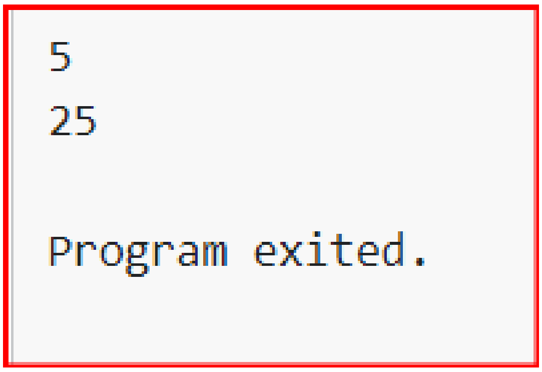
* falltrhough, is used to go in next case, as in Golang case don’t need break.

Closure

closure is a function with similar property of a function which is declared inside another function and can be used within the scope.



```
package main

import "fmt"

func main() {
	var number int = 5
	square := func() {
		number *= number
	}
	fmt.Println(number)
	square()
	fmt.Println(number)
}
```



```
5
25

Program exited.
```

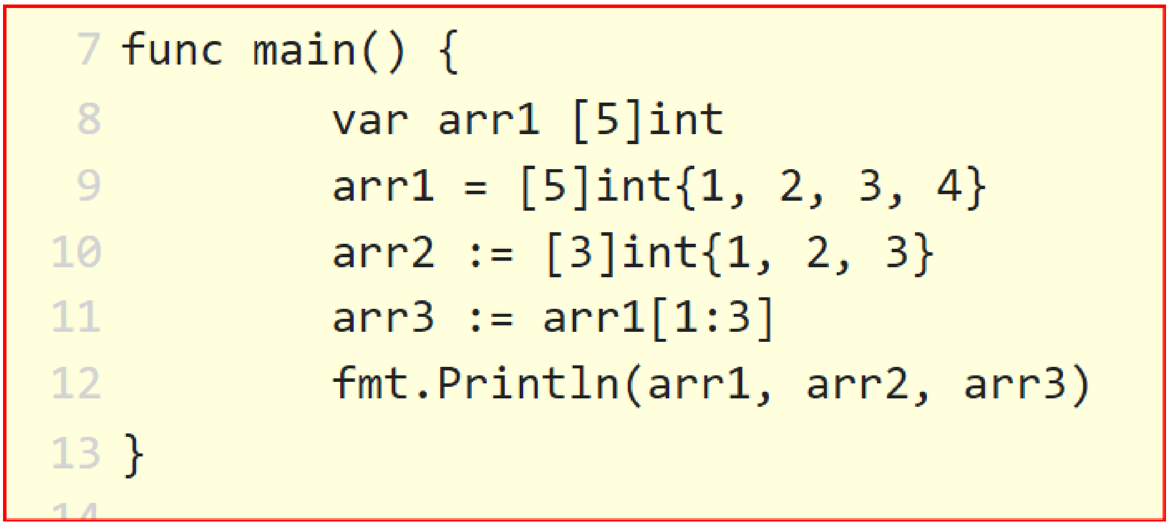
Array

var <variable_name> [<size>]<type>

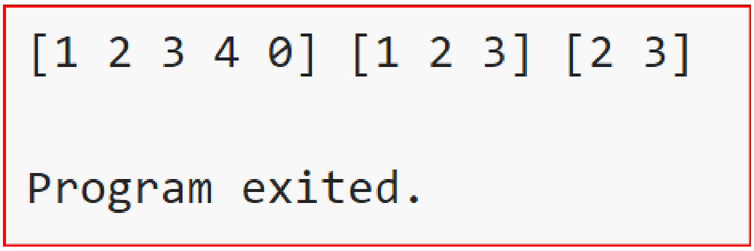
<variable_name> := make([]<type> ,<size>)

Slice

Slice is dynamically sized array.



```
func main() {
	var arr1 [5]int
	arr1 = [5]int{1, 2, 3, 4}
	arr2 := [3]int{1, 2, 3}
	arr3 := arr1[1:3]
	fmt.Println(arr1, arr2, arr3)
}
```



```
[1 2 3 4 0] [1 2 3] [2 3]

Program exited.
```

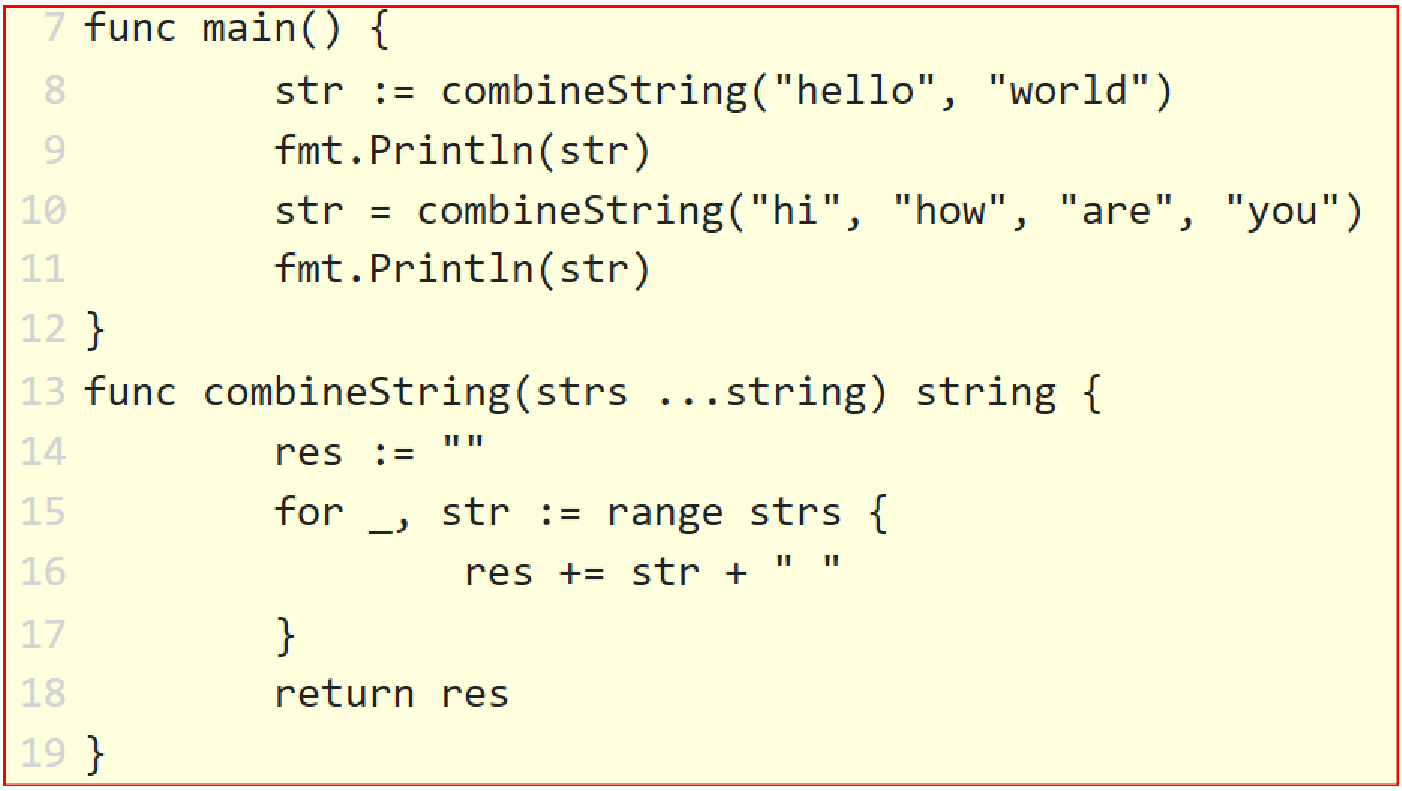
Variadic Functions

Function with variable arguments of similar type.

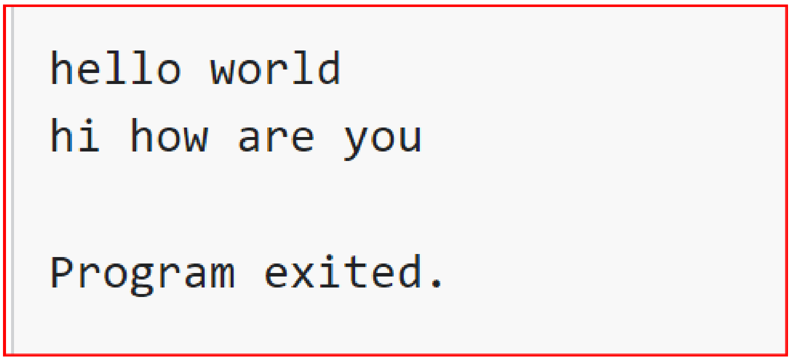
It must be the last argument of a function.

A function must have only one type of variable argument

```
func  <func_name> ( <name>  …<type>)(<type>){
}
```



```
func main() {
        str := combineString("hello", "world")
        fmt.Println(str)
        str = combineString("hi", "how", "are", "you")
        fmt.Println(str)
}
func combineString(strs ...string) string {
        res := ""
        for _, str := range strs {
                res += str + " "
        }
        return res
}
```



```
hello world
hi how are you

Program exited.
```

Structure

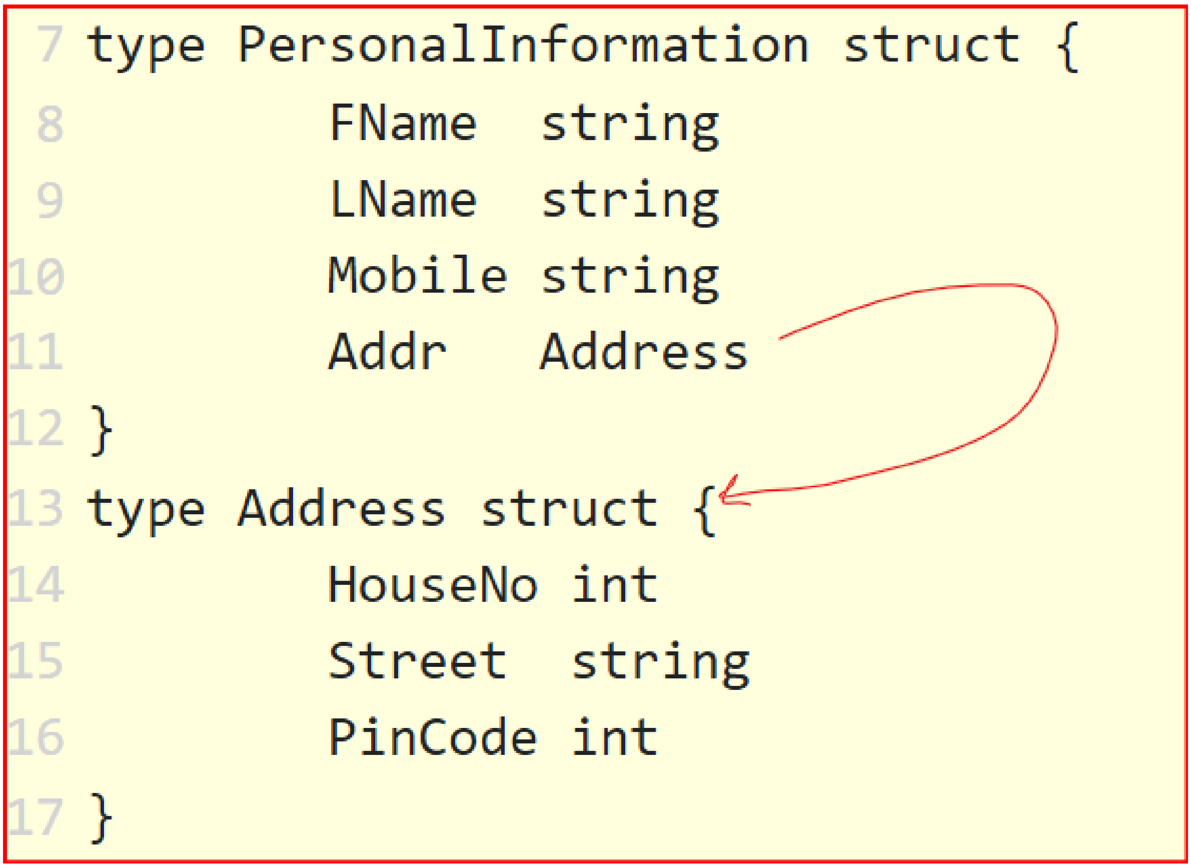
type <structure_name> struct {

<name> <type>

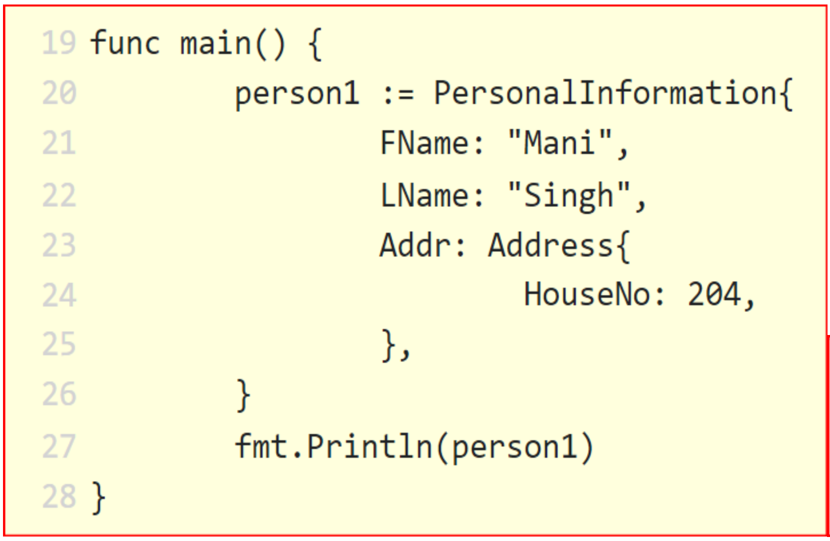
}

Embedded Structure

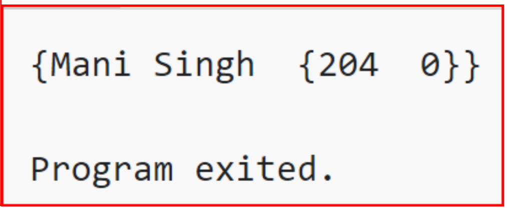
Structure inside structure is called embedded structure.



```
type PersonalInformation struct {
        FName  string
        LName  string
        Mobile string
        Addr   Address
}
type Address struct {
        HouseNo int
        Street  string
        PinCode int
}
```



```
func main() {
        person1 := PersonalInformation{
                FName: "Mani",
                LName: "Singh",
                Addr: Address{
                        HouseNo: 204,
                },
        }
        fmt.Println(person1)
}
```



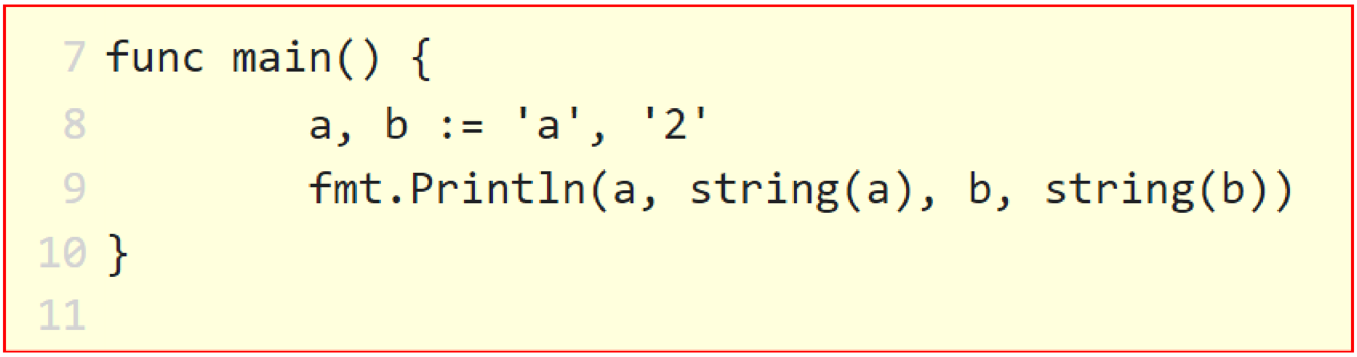
```
{Mani Singh  {204  0}}

Program exited.
```

Rune

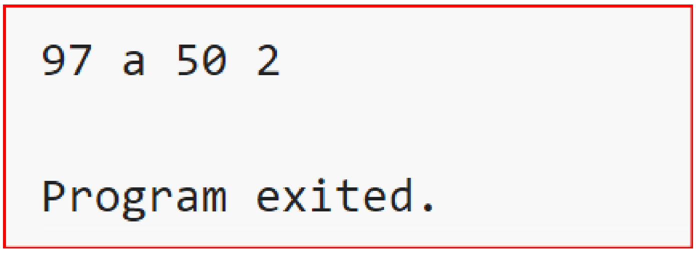
There is no char in Golang. Instead we have rune here.

var <variable_name> rune = value

<variable_name> := ‘value’



```go
func main() {
        a, b := 'a', '2'
        fmt.Println(a, string(a), b, string(b))
}

```



```
97 a 50 2

Program exited.
```

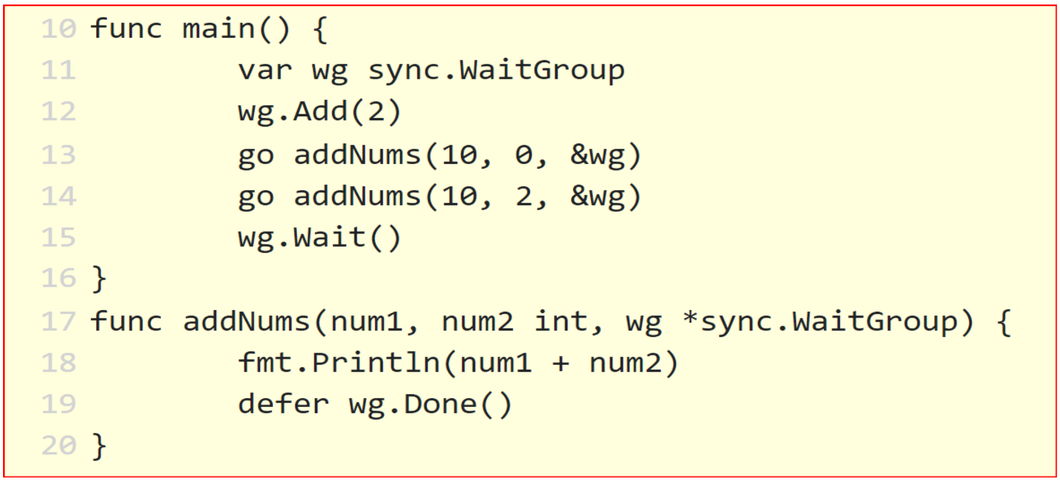
Goroutine

Goroutine is a light weighted tread of execution. Goroutines run synchronously with other threads.

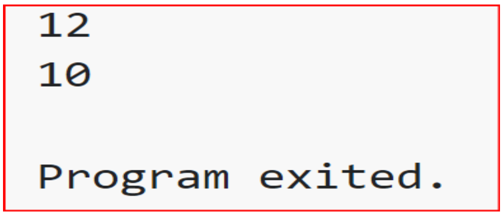
To launch a goroutine we need to add “go” keyword before calling any function.

To completely execute your goroutine you need to explicitly mention your program to wait finish execution of routines.

go <function_name>()



```go
func main() {
        var wg sync.WaitGroup
        wg.Add(2)
        go addNums(10, 0, &wg)
        go addNums(10, 2, &wg)
        wg.Wait()
}
func addNums(num1, num2 int, wg *sync.WaitGroup) {
        fmt.Println(num1 + num2)
        defer wg.Done()
}
```



```
12
10

Program exited.
```

Channels

Channels are the pipes that connect concurrent goroutines. You can send values into channels from one goroutine and receive those values into another goroutine. (link)

Unbuffered channel

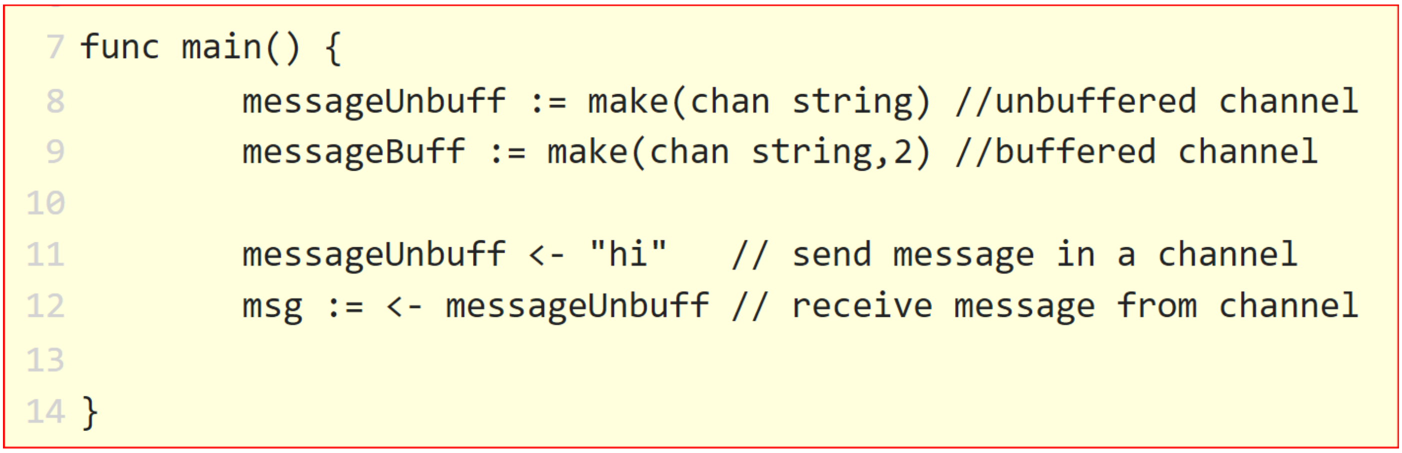
Default channel is unbuffered, which means that a buffer will only accept data if there is a receiver.

<variable_name> := make(chan <variable_type>)

Buffered channel

Buffered channel can receive message up to its size without receiver at the other end.

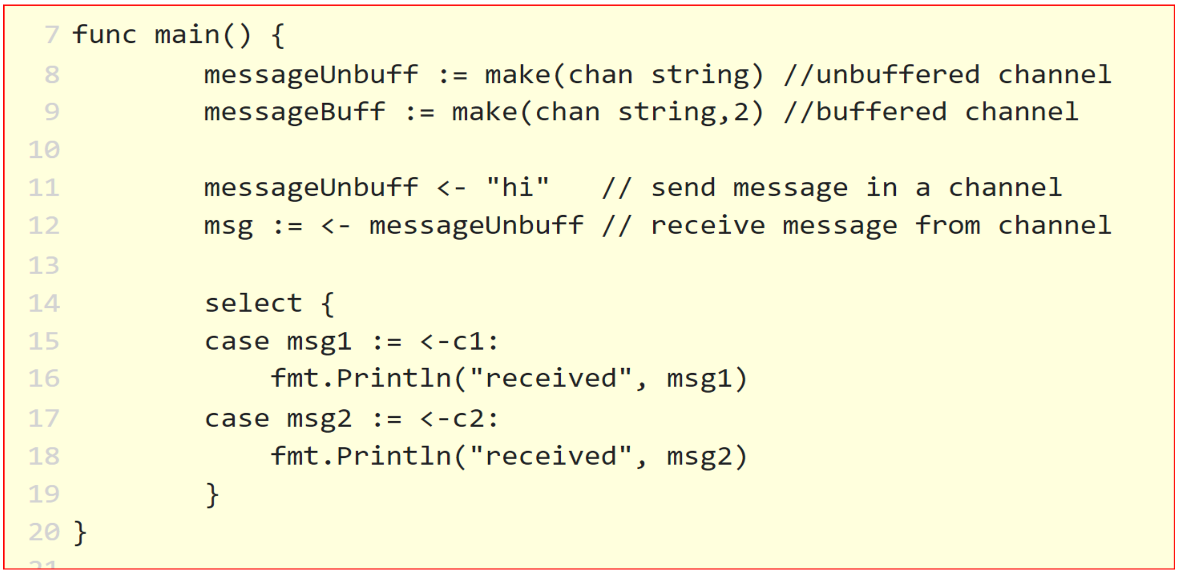
<variable_name> := make(chan <variabel_type>, <channel_size>)



```go
func main() {
        messageUnbuff := make(chan string) //unbuffered channel
        messageBuff := make(chan string,2) //buffered channel

        messageUnbuff <- "hi"   // send message in a channel
        msg := <- messageUnbuff // receive message from channel

}
```

Select

Just like switch case, select let you wait on multiple channel operation.



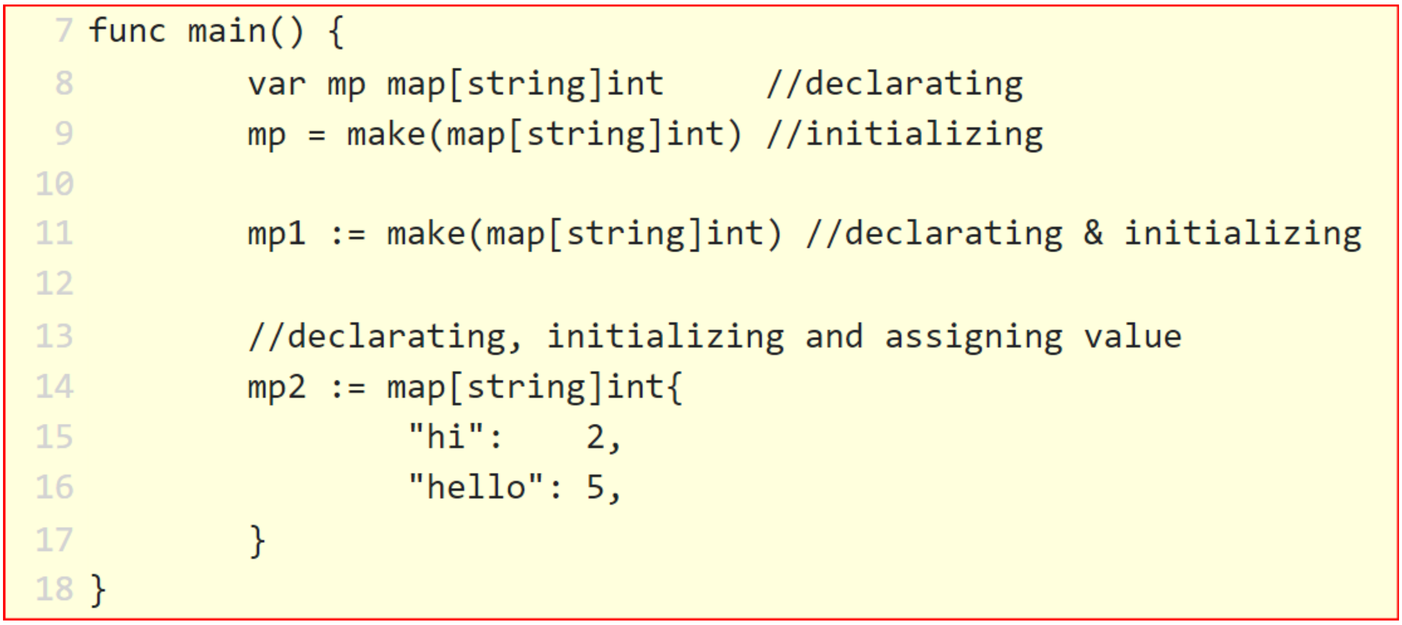
```go
func main() {
        messageUnbuff := make(chan string) //unbuffered channel
        messageBuff := make(chan string,2) //buffered channel

        messageUnbuff <- "hi"   // send message in a channel
        msg := <- messageUnbuff // receive message from channel

        select {
        case msg1 := <-c1:
            fmt.Println("received", msg1)
        case msg2 := <-c2:
            fmt.Println("received", msg2)
        }
}
```

Map

Map is an unordered collection of key and its value

var <variable_name> map[<key_type>]<value_type>

<variable_name> := make(map[<key_type>]<value_type>)



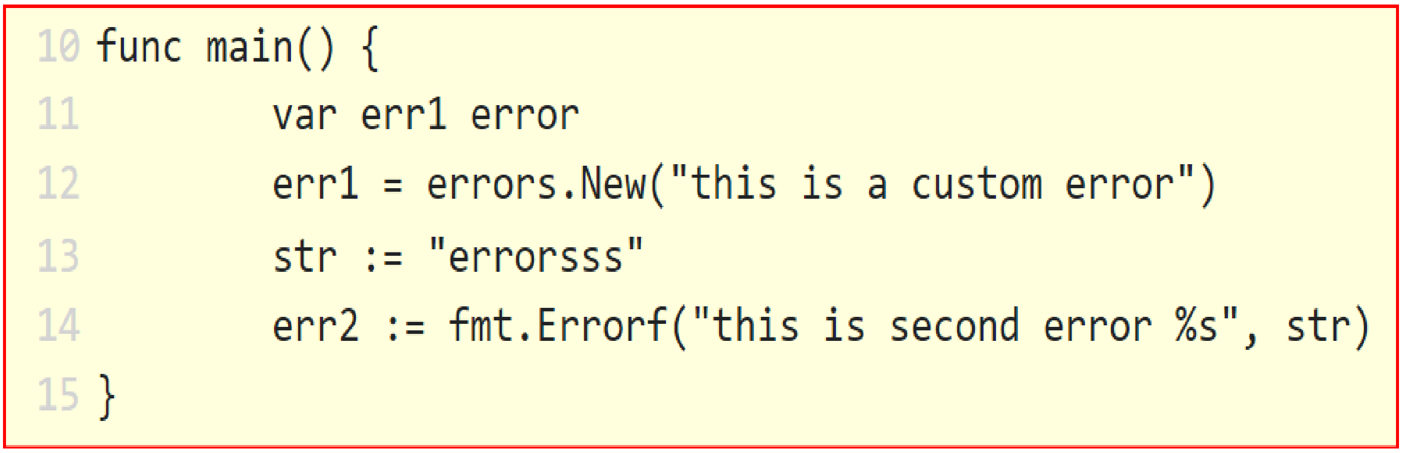
```go
func main() {
	var mp map[string]int     //declarating
	mp = make(map[string]int) //initializing

	mp1 := make(map[string]int) //declarating & initializing

	//declarating, initializing and assigning value
	mp2 := map[string]int{
		"hi":    2,
		"hello": 5,
	}
}
```

* map need declaration as well as initialization

Error

In Golang we don’t have try/catch. To handle exceptions we use errors.

In case of program crash we have different mechanism called **differ**, **panic** and **recover.**



```go
func main() {
	var err1 error
	err1 = errors.New("this is a custom error")
	str := "errorsss"
	err2 := fmt.Errorf("this is second error %s", str)
}
```

Recover

Recover is used to avoid unwanted termination of program caused due to error or panic.

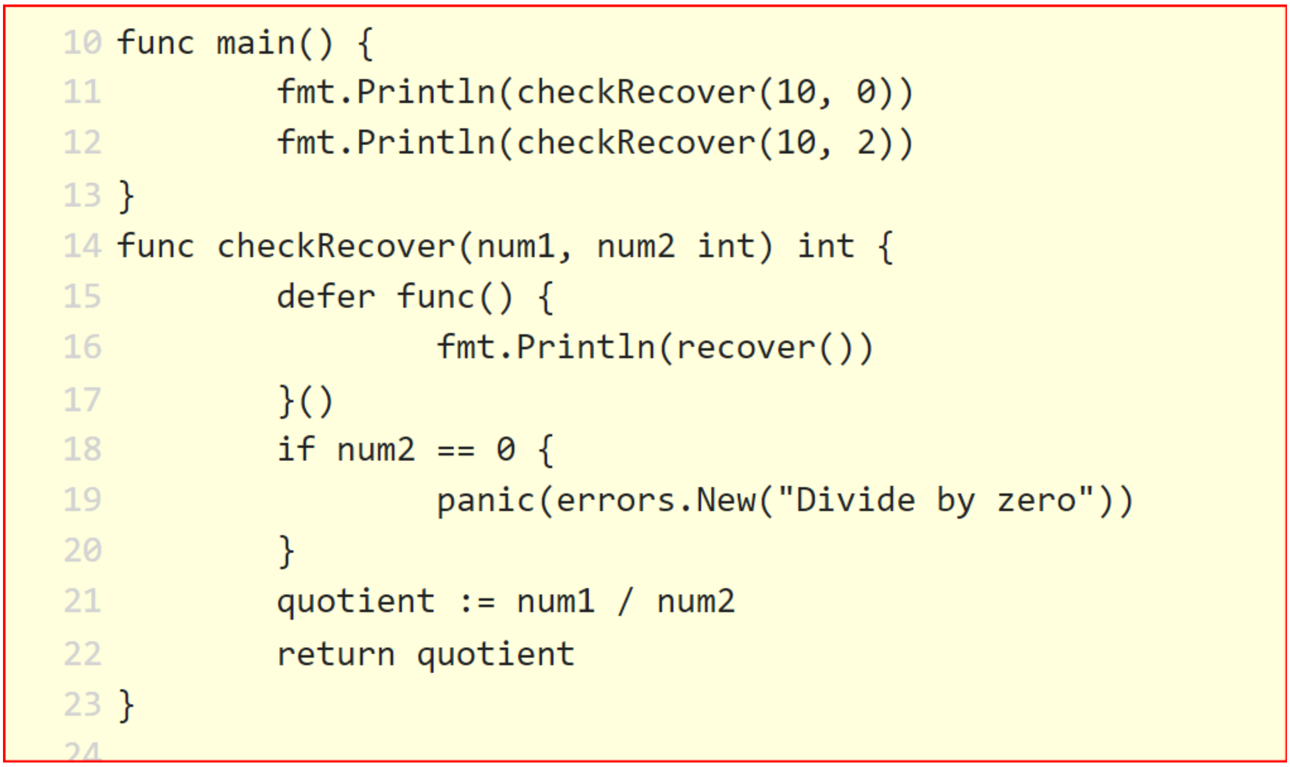
Panic

Panic is used to abort/terminate execution of program. Panic is used to handle error situation.

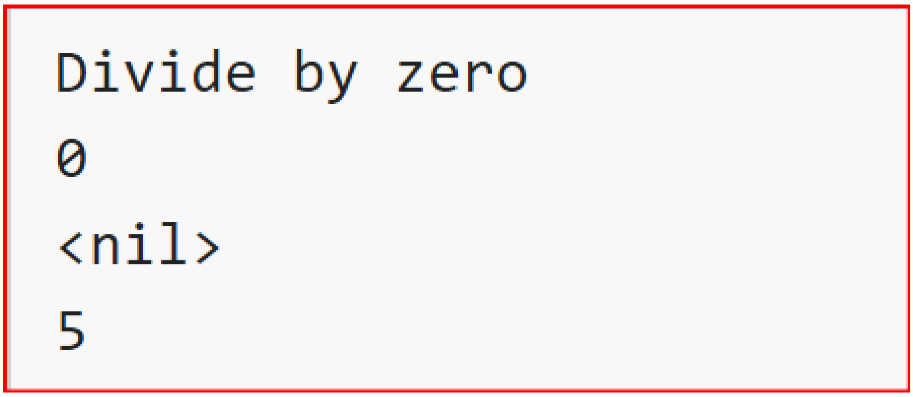
Defer

Defer keyword is used to postpone the execution of function or statement until the end of the function.

Defer can be used for cleaning purpose like close the opened files.



```
func main() {
	fmt.Println(checkRecover(10, 0))
	fmt.Println(checkRecover(10, 2))
}
func checkRecover(num1, num2 int) int {
	defer func() {
		fmt.Println(recover())
	}()
	if num2 == 0 {
		panic(errors.New("Divide by zero"))
	}
	quotient := num1 / num2
	return quotient
}
```



```
Divide by zero
0
<nil>
5
```

Contacts

Name: - *Manindra Narayan Singh*

Email Id: - *mns.manindra@gmail.com*

LinkedIn: - *https://www.linkedin.com/in/mani...*

Mobile No: - *+91 8299661294 (WhatsApp)*