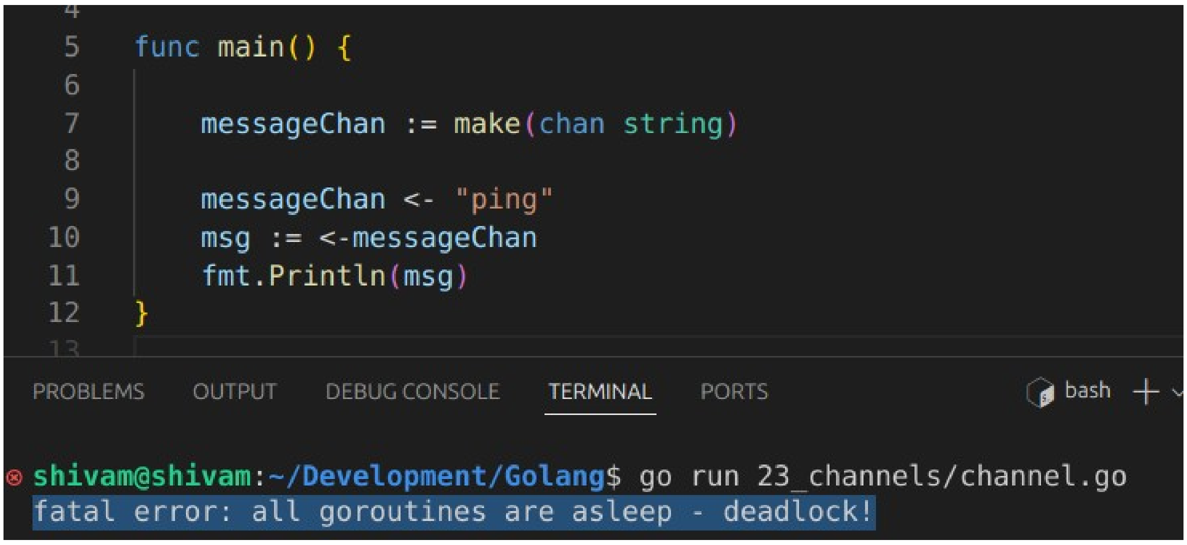
*** CHANNELS ***

- like pipe : to send data from one side and receive at other end
- so when more than one goroutines are running then to transfer the data from on goroutine to another we use goroutine

* a way of communication between the goroutines



```go
func main() {

    messageChan := make(chan string)

    messageChan <- "ping"
    msg := <-messageChan
    fmt.Println(msg)
}
```

```
shivam@shivam:~/Development/Golang$ go run 23_channels/channel.go
fatal error: all goroutines are asleep - deadlock!
```

* first we create the channel using make and define what type of data we will be passing

* to send the data in the channel we use <- incoming symbol and for receiving the data we use outgoing <- symbol

Problem : but here we face the deadlock condition because ??

@1 : here we are using the unbuffered channel of type string and unbuffered channels require both sender and receiver to be ready simultaneously for operation to complete.

Channel Creation:

messageChan := make(chan string)

Here, a unbuffered channel of type string is created. In Go, unbuffered channels require both the sender and receiver to be ready simultaneously for the operation to complete.

Sending to the Channel:

messageChan <- "ping"

The program tries to send the string "ping" to the channel. Since the channel is unbuffered, this operation will block until another goroutine is ready to receive the data.

Receiving from the Channel:

```
      msg := <- messageChan
```

The program tries to read from the channel. However, this read is never reached because the program is already stuck at the previous messageChan <- "ping" operation.

** so it never reaches at this line so messageChan receiver part is also waiting to receive the data

Deadlock Situation:

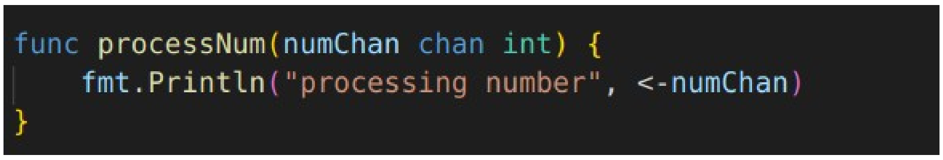
Single Goroutine: The main function runs in a single goroutine. There are no other goroutines to perform the corresponding read operation (msg := <-messageChan) while the send operation (messageChan <- "ping") is waiting.

Unbuffered Channel: The unbuffered channel blocks until both send and receive operations occur simultaneously. Since there's only one goroutine (the main goroutine), neither operation can proceed, leading to a deadlock.

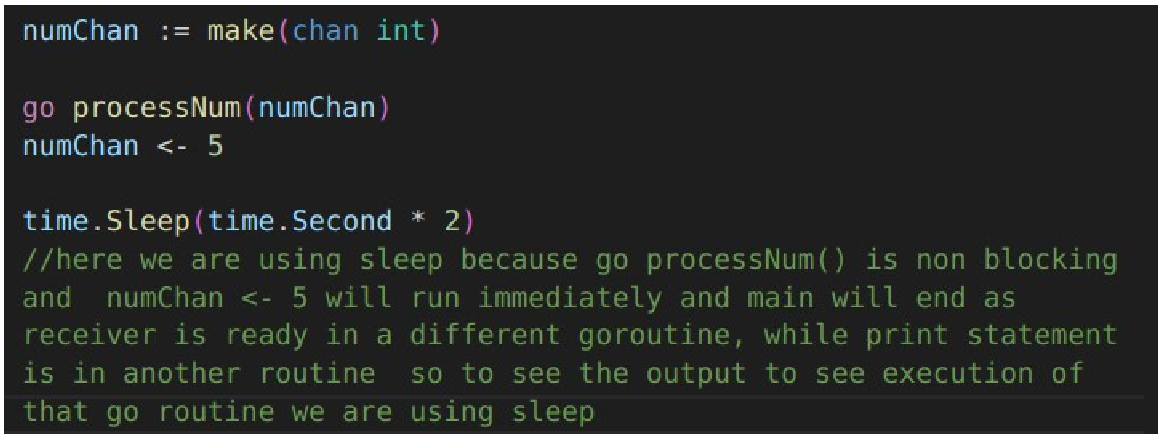
** keep in mind that main is also a goroutine.

----

* Let’s make one more go routine and try to send the data over the channel
- we will create a function which will run in a different goroutine



```go
func processNum(numChan chan int) {
    fmt.Println("processing number", <-numChan)
}
```

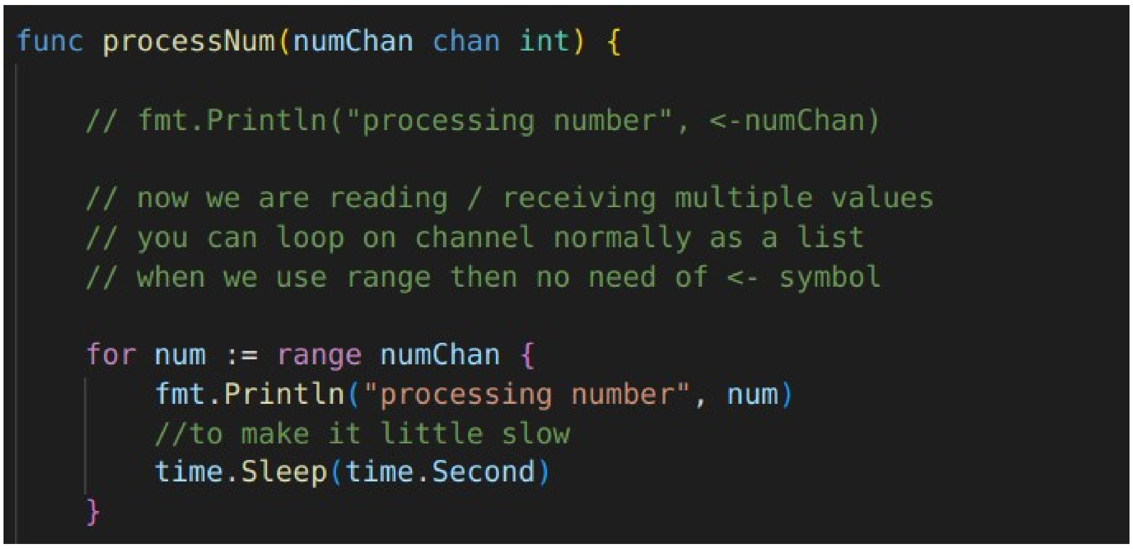


```go
numChan := make(chan int)

go processNum(numChan)
numChan <- 5

time.Sleep(time.Second * 2)
//here we are using sleep because go processNum() is non blocking
and  numChan <- 5 will run immediately and main will end as
receiver is ready in a different goroutine, while print statement
is in another routine  so to see the output to see execution of
that go routine we are using sleep
```

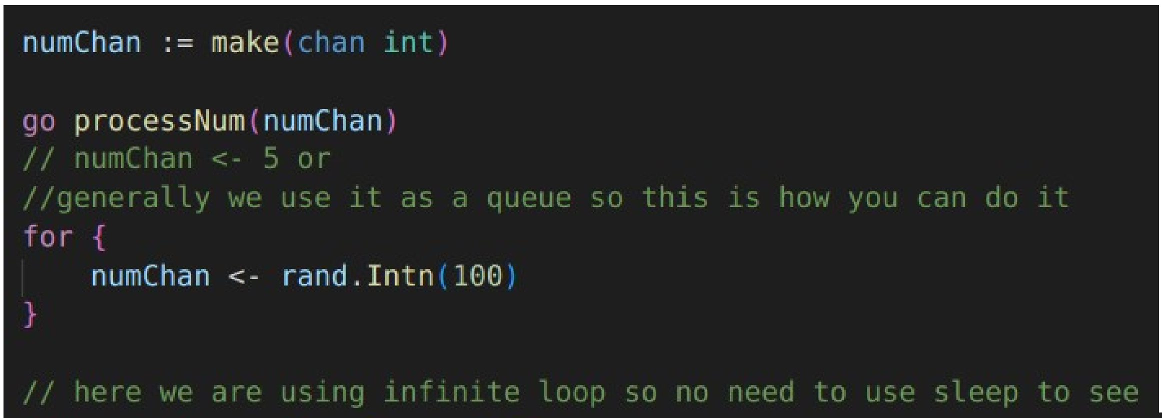
- here processNum runs in a different goroutine
- also we need to pass the channel to the function where we will be receiving the data

note : first we are passing the channel so that receiver is also ready and also if you send the data first then it will block the function calling and receiver part will never be ready and you will get deadlock

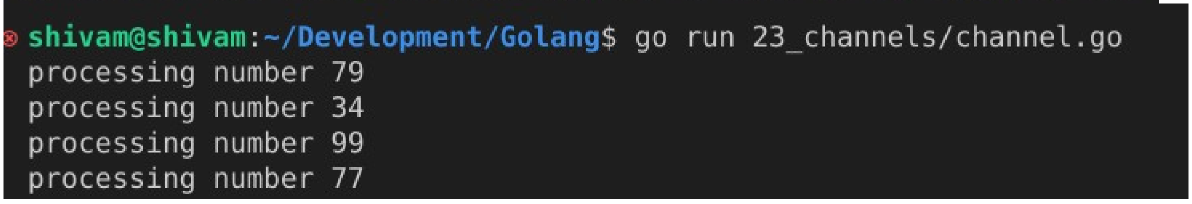
- -- -
* and generally we don't pass a single data, we use it as a queue ..



```go
func processNum(numChan chan int) {

    // fmt.Println("processing number", <-numChan)

    // now we are reading / receiving multiple values
    // you can loop on channel normally as a list
    // when we use range then no need of <- symbol

    for num := range numChan {
        fmt.Println("processing number", num)
        //to make it little slow
        time.Sleep(time.Second)
    }
```



```go
numChan := make(chan int)

go processNum(numChan)
// numChan <- 5 or
//generally we use it as a queue so this is how you can do it
for {
    numChan <- rand.Intn(100)
}

// here we are using infinite loop so no need to use sleep to see
```



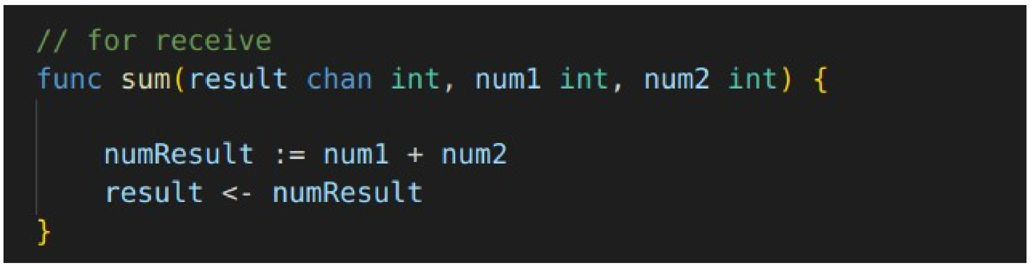
```
shivam@shivam:~/Development/Golang$ go run 23_channels/channel.go
processing number 79
processing number 34
processing number 99
processing number 77
```

- now you are getting the values and receiving through the channel

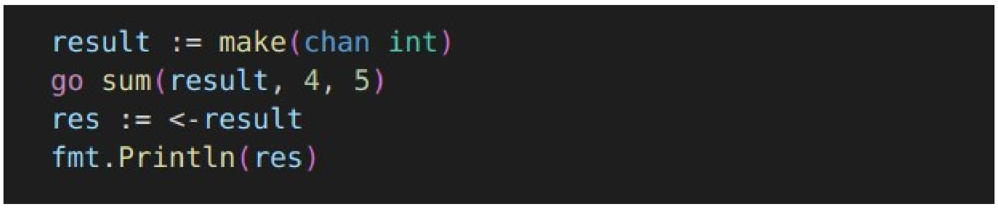
* now you can see we can use the queue system using channels
----

* here we are sending the data to the goroutine where we have send the channel similarly we can receive the data from the routine

- to receive the data this is how you can do that ...



```go
// for receive
func sum(result chan int, num1 int, num2 int) {

    numResult := num1 + num2
    result <- numResult
}
```

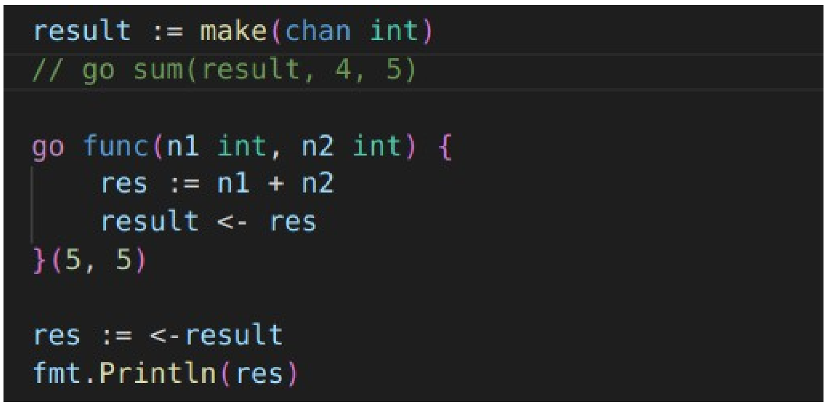


```go
result := make(chan int)
go sum(result, 4, 5)
res := <-result
fmt.Println(res)
```

- you send the channel with arguments and through that channel you
  receive the result

* [I discovered this point yayyyy] note : here you are making the function outside of main so you need to pass the channel what if you make the anonymous function in the main then will it get the channel from the closure ?? YES it will get it and we will run this anonymous function also in a routine .

- below you can see this :

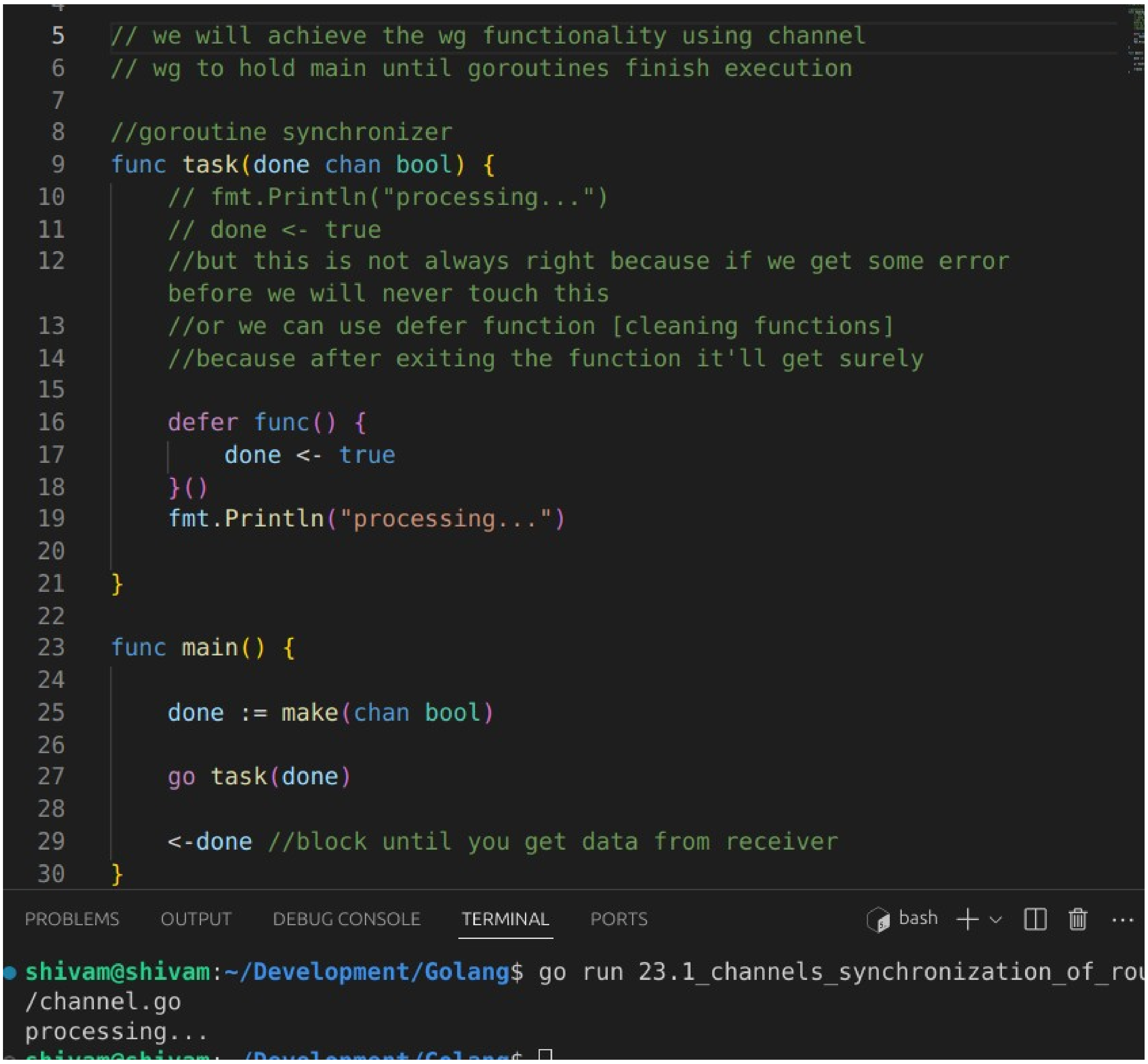


```go
result := make(chan int)
// go sum(result, 4, 5)

go func(n1 int, n2 int) {
    res := n1 + n2
    result <- res
}(5, 5)

res := <-result
fmt.Println(res)
```


=> channels for synchronization :

- previously we were using waitgroups to let finish the goroutines and then our main function should end

- we can achieve the same thing using channel also



```go
// we will achieve the wg functionality using channel
// wg to hold main until goroutines finish execution

//goroutine synchronizer
func task(done chan bool) {
	// fmt.Println("processing...")
	// done <- true
	//but this is not always right because if we get some error before we will never touch this
	//or we can use defer function [cleaning functions]
	//because after exiting the function it'll get surely

	defer func() {
		done <- true
	}()
	fmt.Println("processing...")

}

func main() {

	done := make(chan bool)

	go task(done)

	<-done //block until you get data from receiver
}
```

```
shivam@shivam:~/Development/Golang$ go run 23.1_channels_synchronization_of_rou
/channel.go
processing...
```

- but now there is a confusion when to use the channel or waitgroups
** if you have single goroutine then use channels otherwise in case of more goroutines use waitgroups for synchronization …

BUFFERED CHANNELS

- previously we were using unbuffered channel and in unbuffered channel sending and receiving is blocking
- if you want to do one by one then use unbuffered channel

- if you want to use queue kind of thing then it will work but it is blocking so it will be very slow because

Unbuffered channels require that **every send operation (`chan <- value`) is matched with a corresponding receive operation (`value := <-chan`) at the exact moment.** This introduces significant synchronization overhead compared to buffered channels.

- **Blocking Behavior:**
  - A `send` blocks until another goroutine performs a `receive`.
  - A `receive` blocks until another goroutine performs a `send`.

This tight synchronization makes queue operations inherently slower because **senders and receivers must wait for each other to proceed**, leading to a lot of idle time.

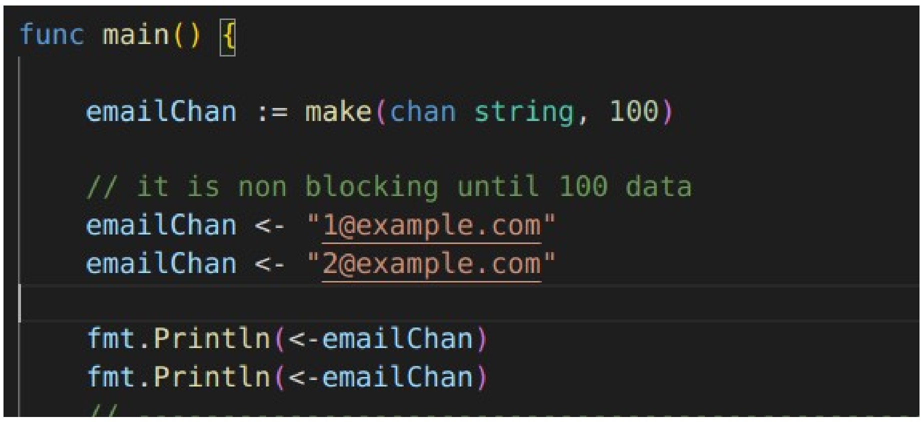
```
-
* so basically in queue there is enqueue and dequeue so for each
enqueue there must be a dequeue then again enqueue
```

If you implement a queue using an unbuffered channel, each operation requires:

- **Synchronization between producer (enqueue) and consumer (dequeue).**
- No overlap or parallelism between enqueue and dequeue operations.

Since every enqueue must wait for a corresponding dequeue (or vice versa), the throughput of the queue depends entirely on the timing and responsiveness of both sender and receiver goroutines.

```
-----------------------------------------------------------------------

* we will use buffered channels for these kind of implementation :

[in unbuffered channel at one time you can send only one data until it
don’t get receive at other end]

-------------------------------------------------------------------
Ex:1 simplest buffer channel
```



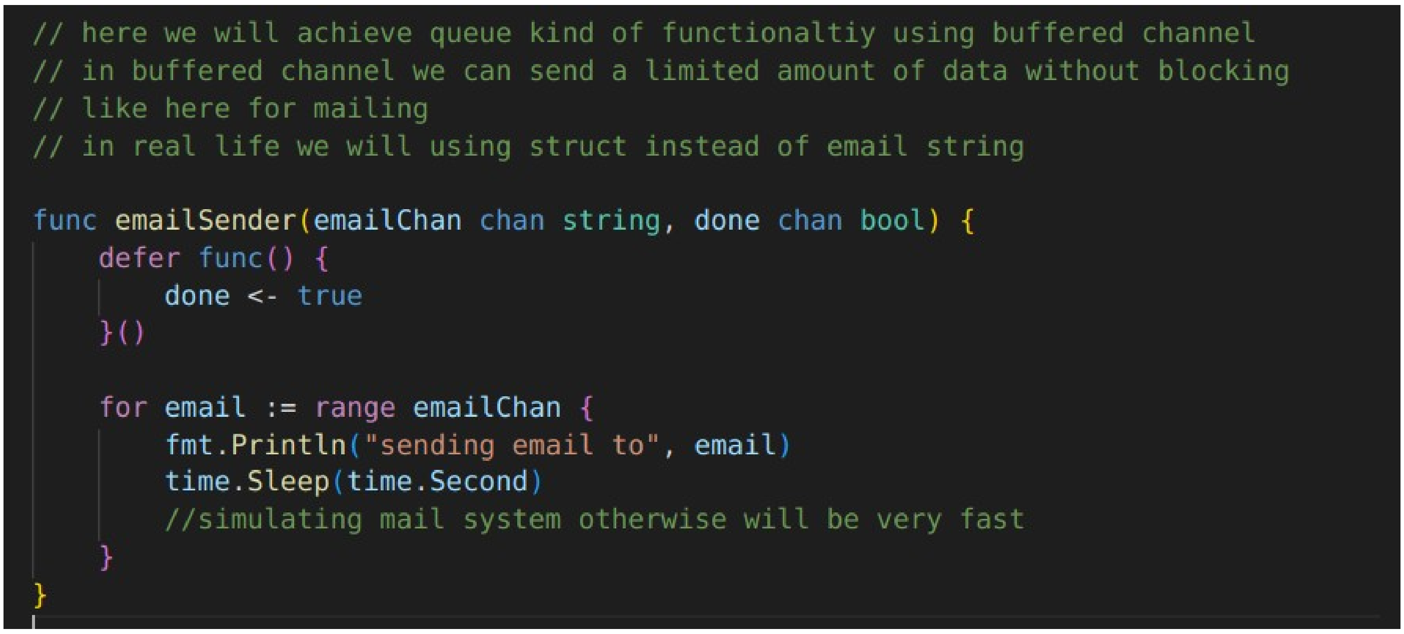
```go
func main() {

    emailChan := make(chan string, 100)

    // it is non blocking until 100 data
    emailChan <- "1@example.com"
    emailChan <- "2@example.com"

    fmt.Println(<-emailChan)
    fmt.Println(<-emailChan)
    // -----------------------------------------
```

NOTE :
[here notice that : we created channel and we are sending the data and receiving the data in the same routine and it’s happening because buffered channels are non blocking till their defined size

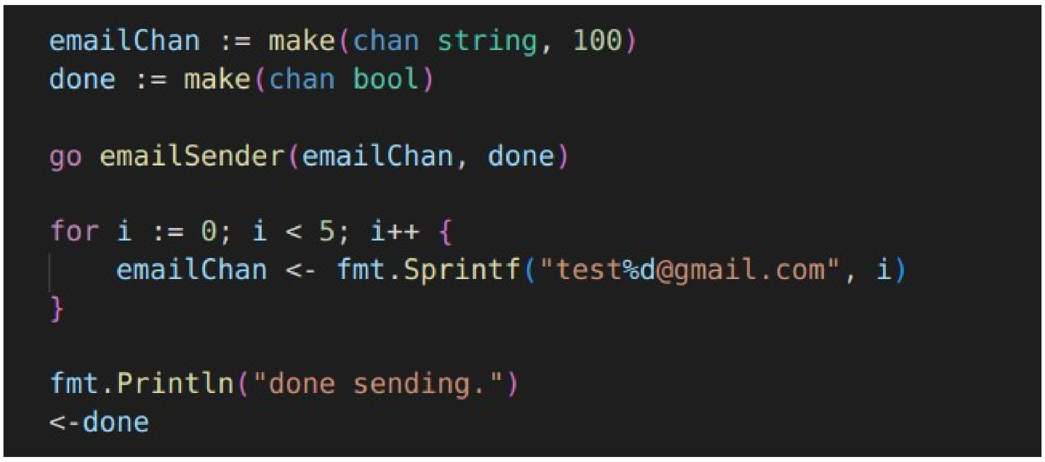
* here we are defining the size of the buffer
* so it already knows the size that’s why it doesn’t block

---

* now we will simulate the reading the emails from the db and sending them emails :



```go
// here we will achieve queue kind of functionaltiy using buffered channel
// in buffered channel we can send a limited amount of data without blocking
// like here for mailing
// in real life we will using struct instead of email string

func emailSender(emailChan chan string, done chan bool) {
	defer func() {
		done <- true
	}()

	for email := range emailChan {
		fmt.Println("sending email to", email)
		time.Sleep(time.Second)
		//simulating mail system otherwise will be very fast
	}
}
```



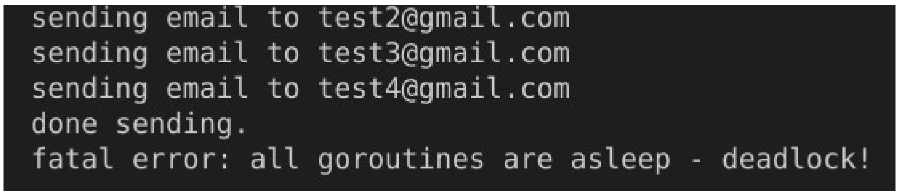
```go
emailChan := make(chan string, 100)
done := make(chan bool)

go emailSender(emailChan, done)

for i := 0; i < 5; i++ {
	emailChan <- fmt.Sprintf("test%d@gmail.com", i)
}

fmt.Println("done sending.")
<-done
```

- here we are also using done chan so that we can simulate the sleep thing otherwise main will be finish execution so fast and we will not be able to see the output of other routine

- and since there is single go routine [2 go routines including the main routine] that’s why we are using the done channel otherwise we will be using the wait groups

PROBLEM : but there is a problem :

- the problem is with range of emailSender function because
- range will try to loop infinite times over the emailChain so after 5 times here it will keep waiting which will cause deadlock

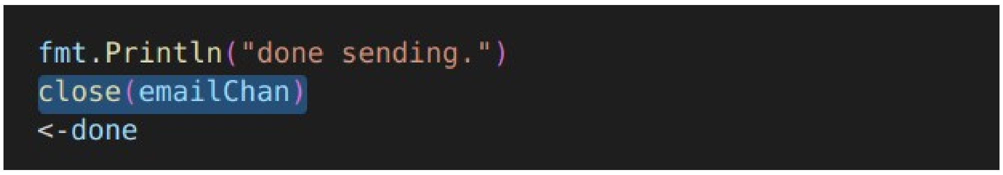


```
sending email to test2@gmail.com
sending email to test3@gmail.com
sending email to test4@gmail.com
done sending.
fatal error: all goroutines are asleep - deadlock!
```

because due to that it will keep waiting to receive and due to that done will never send true because defer get executed once the function get finish and **←done** in the main() will keep waiting till infinite

----

* so in case of buffered channel where we are using the range we must use close method



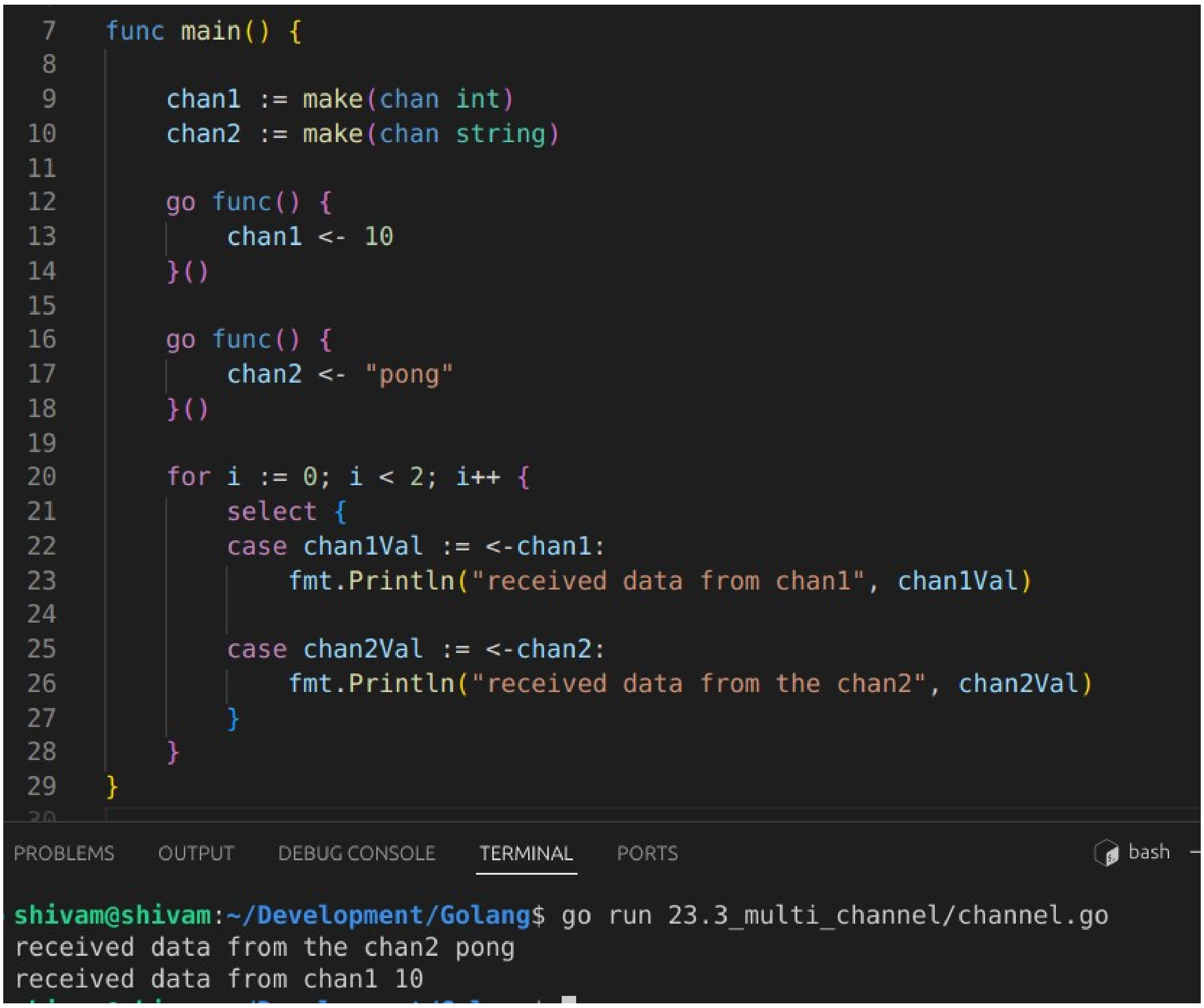
```
fmt.Println("done sending.")
close(emailChan)
<-done
```

* so after finishing our work we need to close the channel .

* and this is how we can built queue kind of things

-
- here we were working with one channel

- if we have to receive the data from multiple channels then we can use select with loop



```go
func main() {

	chan1 := make(chan int)
	chan2 := make(chan string)

	go func() {
		chan1 <- 10
	}()

	go func() {
		chan2 <- "pong"
	}()

	for i := 0; i < 2; i++ {
		select {
		case chan1Val := <-chan1:
			fmt.Println("received data from chan1", chan1Val)

		case chan2Val := <-chan2:
			fmt.Println("received data from the chan2", chan2Val)
		}
	}
}
```

```
shivam@shivam:~/Development/Golang$ go run 23.3_multi_channel/channel.go
received data from the chan2 pong
received data from chan1 10
```

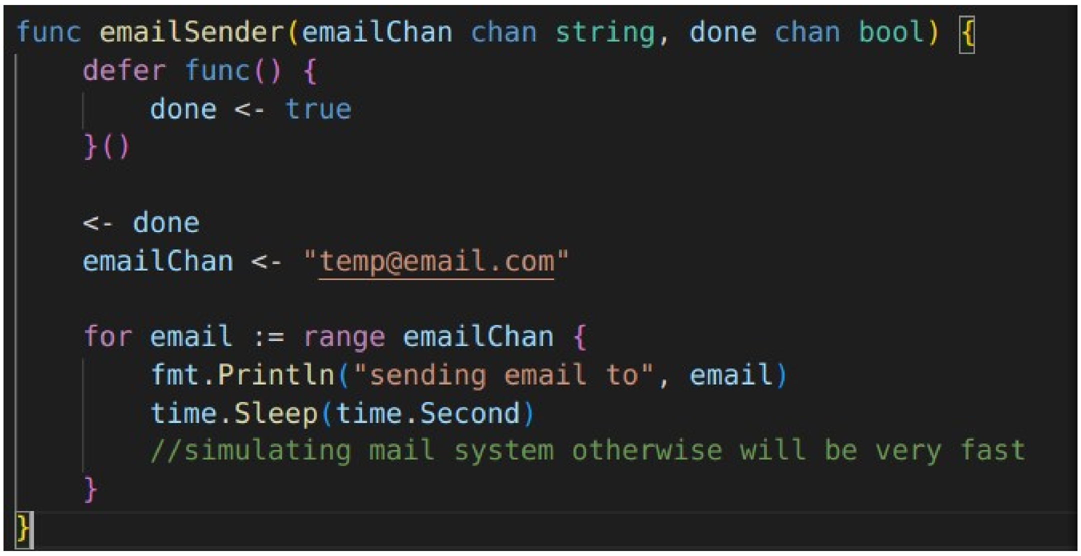
NOTE : here also we are using channels from the closure so we don’t need to pass the channel

==

- there is one more thing left in the channels :
- type safety

** like previously we see that we were passing the channel but there was no rule like they will receive or send data only

** there they can be use in anyway
- below you can see that

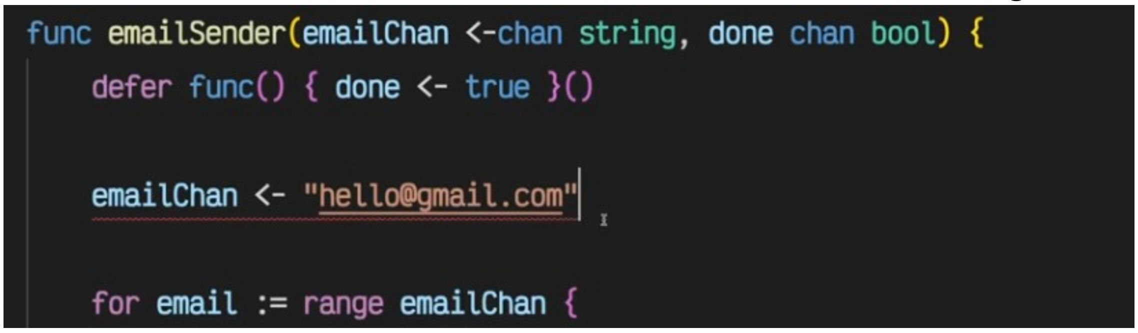


```go
func emailSender(emailChan chan string, done chan bool) {
    defer func() {
        done <- true
    }()

    <- done
    emailChan <- "temp@email.com"

    for email := range emailChan {
        fmt.Println("sending email to", email)
        time.Sleep(time.Second)
        //simulating mail system otherwise will be very fast
    }
}
```

- here you can see that we were using the channels
→ emailChain to receive the data
→ done to send the data

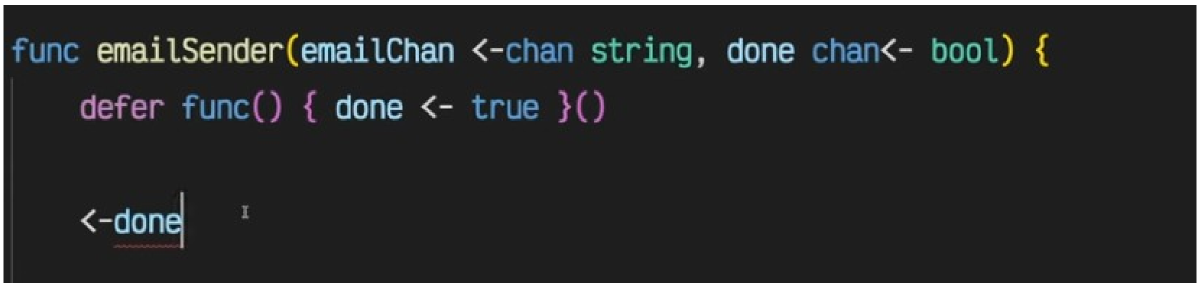
- but below you can see that we can pass the data and receive the data but there should be a proper way of channels if they are made to receive then they can’t send same for sender channels

=> to achieve this we can use the **emailChain ←chan** string



```go
func emailSender(emailChan <-chan string, done chan bool) {
    defer func() { done <- true }()

    emailChan <- "hello@gmail.com"

    for email := range emailChan {
```

- here you can see now we are getting error
- similarly for send also we can do this done **chan← bool**



```go
func emailSender(emailChan <-chan string, done chan<- bool) {
    defer func() { done <- true }()

    <-done
```