Unit : 8
Concurrency

- Goroutines
- Channels
- WaitGroups
- Mutexes
- Select statements

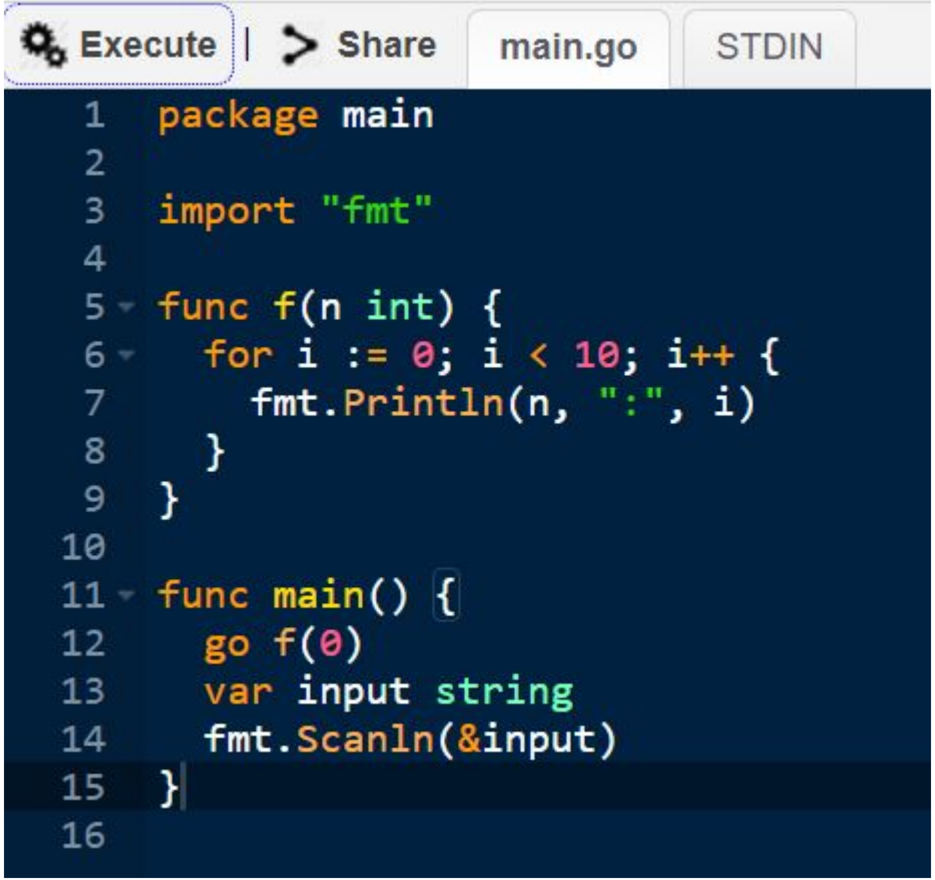
Concurrency

- Large programs are often made up of many smaller sub-programs
- For example a web server handles requests made from web browsers and serves up HTML web pages in response
- Each request is handled like a small program
- It would be ideal for programs like these to be able to run their smaller components at the same time
- Making progress on more than one task simultaneously is known as concurrency
- Go has rich support for concurrency using goroutines and channels

Goroutines

- A goroutine is a function that is capable of running concurrently with other functions
- To create a goroutine we use the keyword go followed by a function invocation:

Goroutines



```go
package main

import "fmt"

func f(n int) {
  for i := 0; i < 10; i++ {
    fmt.Println(n, ":", i)
  }
}

func main() {
  go f(0)
  var input string
  fmt.Scanln(&input)
}

```

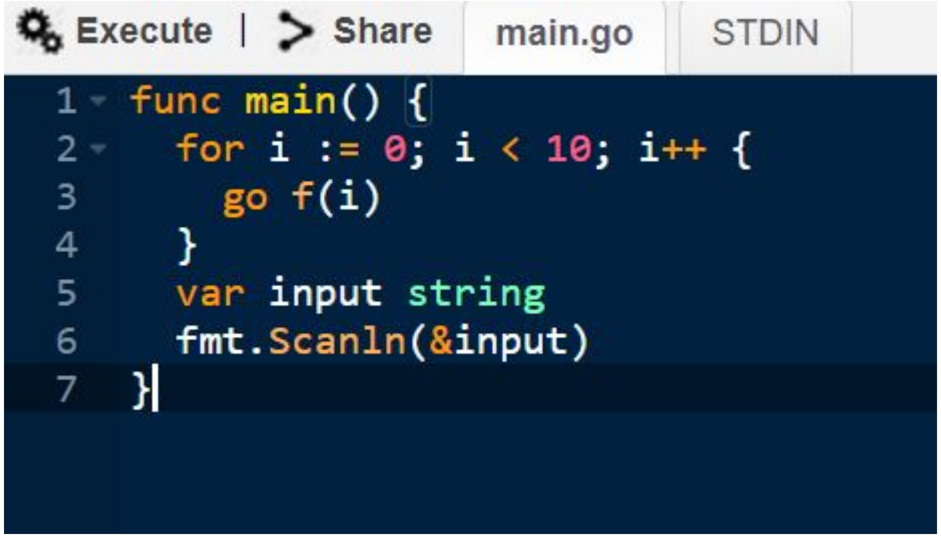
Goroutines

- This program consists of two goroutines
- The first goroutine is implicit and is the main function itself
- The second goroutine is created when we call go f(0)
- Normally when we invoke a function our program will execute all the statements in a function and then return to the next line following the invocation
- With a goroutine we return immediately to the next line and don't wait for the function to complete

Goroutines

- This is why the call to the Scanln function has been included;
  - without it the program would exit before being given the opportunity to print all the numbers
- Goroutines are lightweight and we can easily create thousands of them
- We can modify our program to run 10 goroutines by doing this:

Goroutines

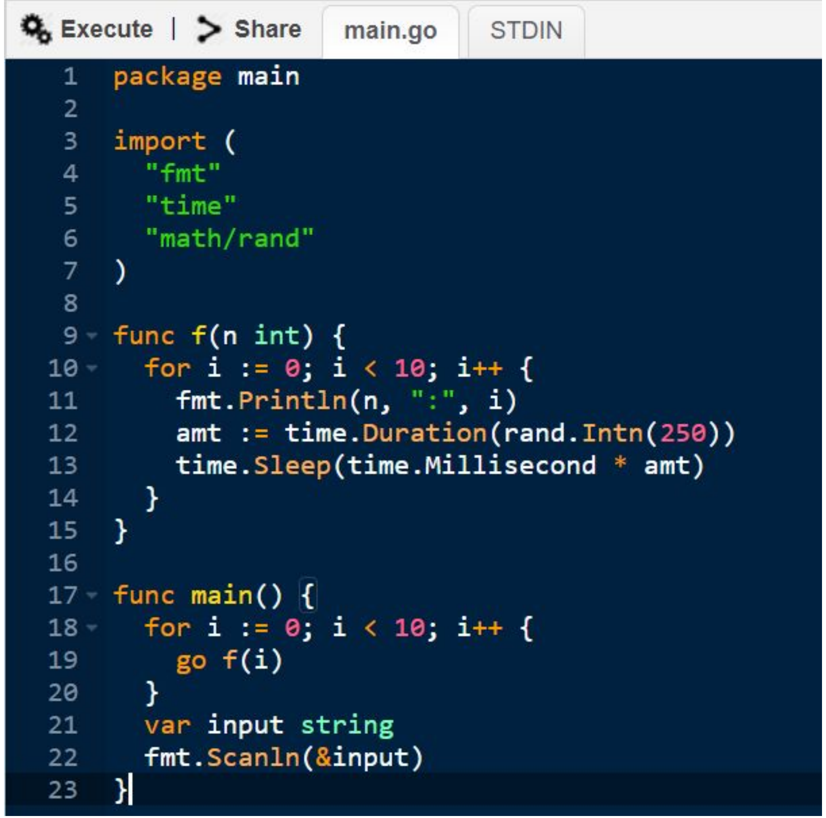


```go
func main() {
  for i := 0; i < 10; i++ {
    go f(i)
  }
  var input string
  fmt.Scanln(&input)
}
```

Goroutines

- You may have noticed that when you run this program it seems to run the goroutines in order rather than simultaneously
- Let's add some delay to the function using time.Sleep and rand.Intn:

Goroutines

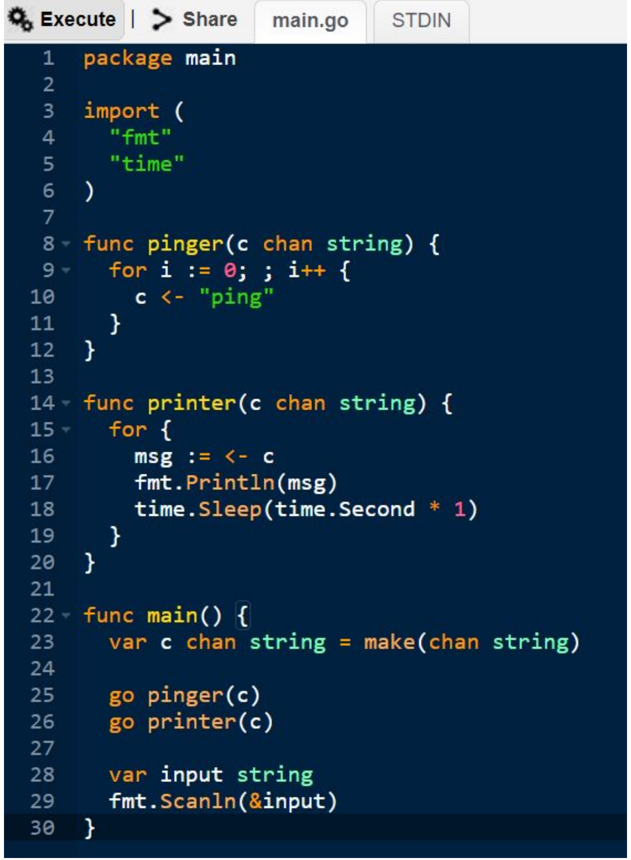


```go
package main

import (
  "fmt"
  "time"
  "math/rand"
)

func f(n int) {
  for i := 0; i < 10; i++ {
    fmt.Println(n, ":", i)
    amt := time.Duration(rand.Intn(250))
    time.Sleep(time.Millisecond * amt)
  }
}

func main() {
  for i := 0; i < 10; i++ {
    go f(i)
  }
  var input string
  fmt.Scanln(&input)
}
```

Channels

- Channels provide a way for two goroutines to communicate with one another and synchronize their execution
- Here is an example program using channels:

Channels



```go
package main

import (
  "fmt"
  "time"
)

func pinger(c chan string) {
  for i := 0; ; i++ {
    c <- "ping"
  }
}

func printer(c chan string) {
  for {
    msg := <- c
    fmt.Println(msg)
    time.Sleep(time.Second * 1)
  }
}

func main() {
  var c chan string = make(chan string)

  go pinger(c)
  go printer(c)

  var input string
  fmt.Scanln(&input)
}
```

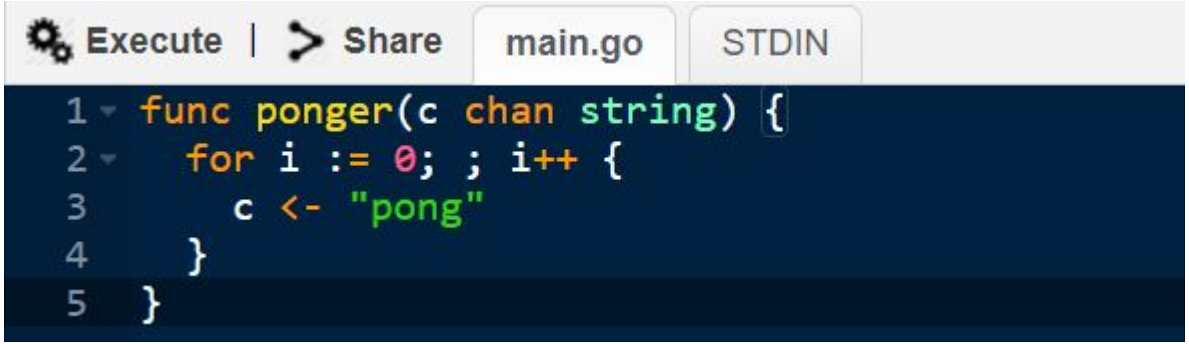
Channels

- This program will print “ping” forever (hit enter to stop it)
- A channel type is represented with the keyword chan followed by the type of the things that are passed on the channel (in this case we are passing strings)
- The <- (left arrow) operator is used to send and receive messages on the channel
  - c <- "ping" means send "ping". msg := <- c means receive a message and store it in msg

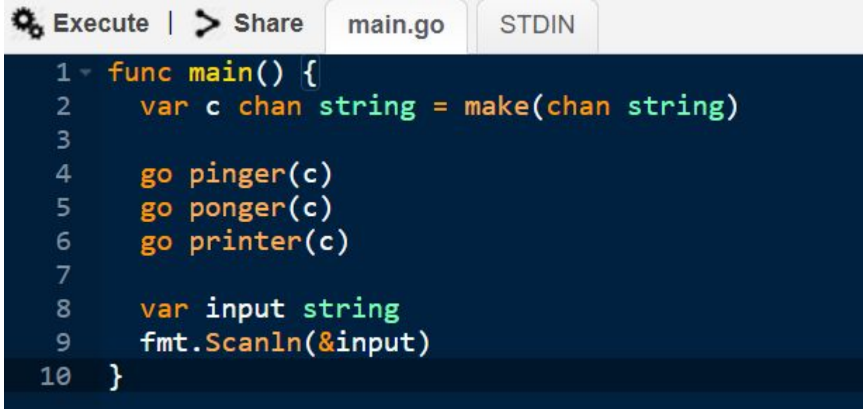
Channels

- The fmt line could also have been written like this: fmt.Println(<-c) in which case we could remove the previous line
- Using a channel like this synchronizes the two goroutines
- When pinger attempts to send a message on the channel it will wait until printer is ready to receive the message(this is known as blocking)
- Let's add another sender to the program and see what happens
- Add this function:

Channels



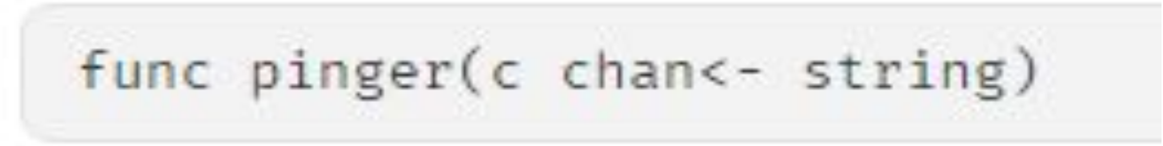
```go
func ponger(c chan string) {
  for i := 0; ; i++ {
    c <- "pong"
  }
}
```



```go
func main() {
  var c chan string = make(chan string)

  go pinger(c)
  go ponger(c)
  go printer(c)

  var input string
  fmt.Scanln(&input)
}
```

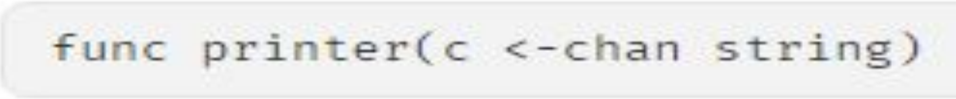
Channel Direction

- ➢ We can specify a direction on a channel type thus restricting it to either sending or receiving
- ➢ For example pinger's function signature can be changed to this:



```
func pinger(c chan<- string)
```

- ➢ Now c can only be sent to. Attempting to receive from c will result in a compiler error. Similarly we can change printer to this:



```
func printer(c <-chan string)
```

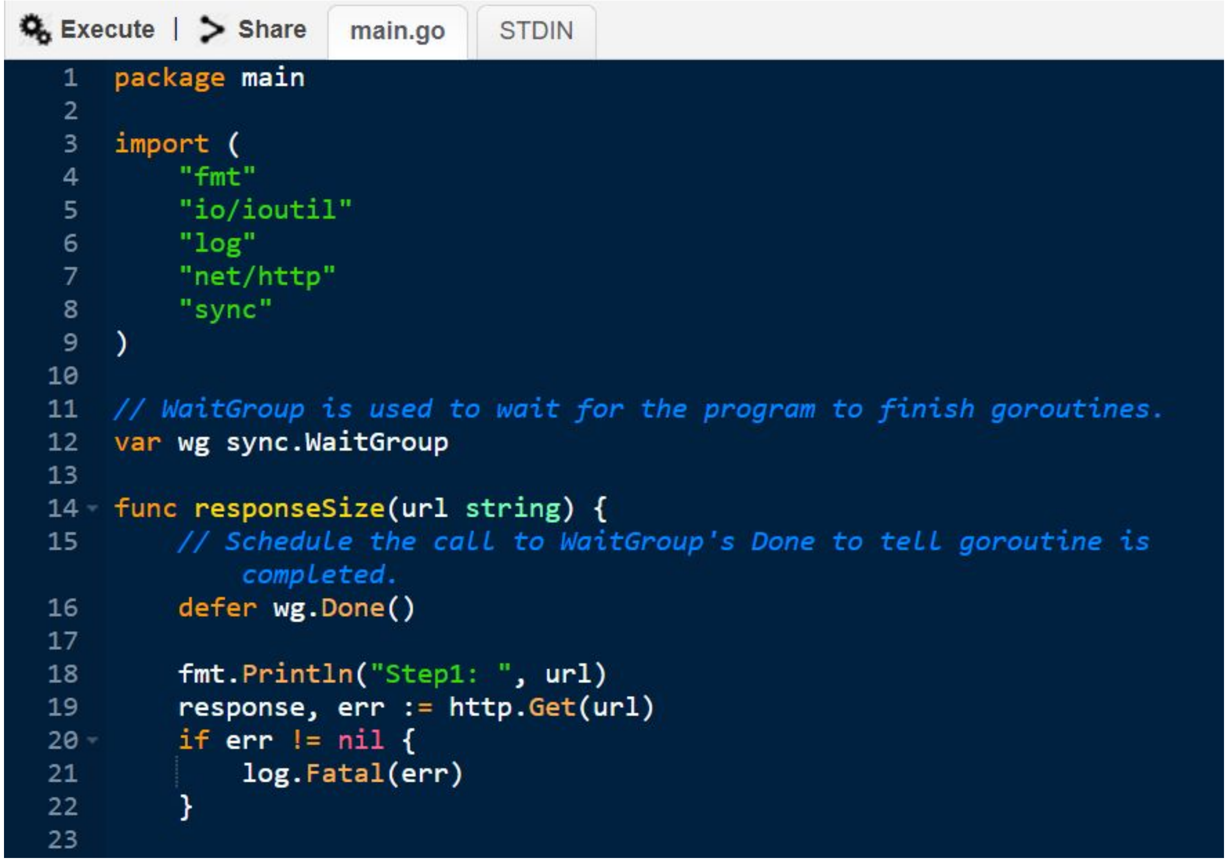
Waiting for Goroutines to Finish Execution

- The WaitGroup type of sync package, is used to wait for the program to finish all goroutines launched from the main function
- It uses a counter that specifies the number of goroutines, and
  - Wait blocks the execution of the program until the WaitGroup counter is zero
- The Add method is used to add a counter to the WaitGroup
- The Done method of WaitGroup is scheduled using a defer statement to decrement the WaitGroup counter

Waiting for Goroutines to Finish Execution

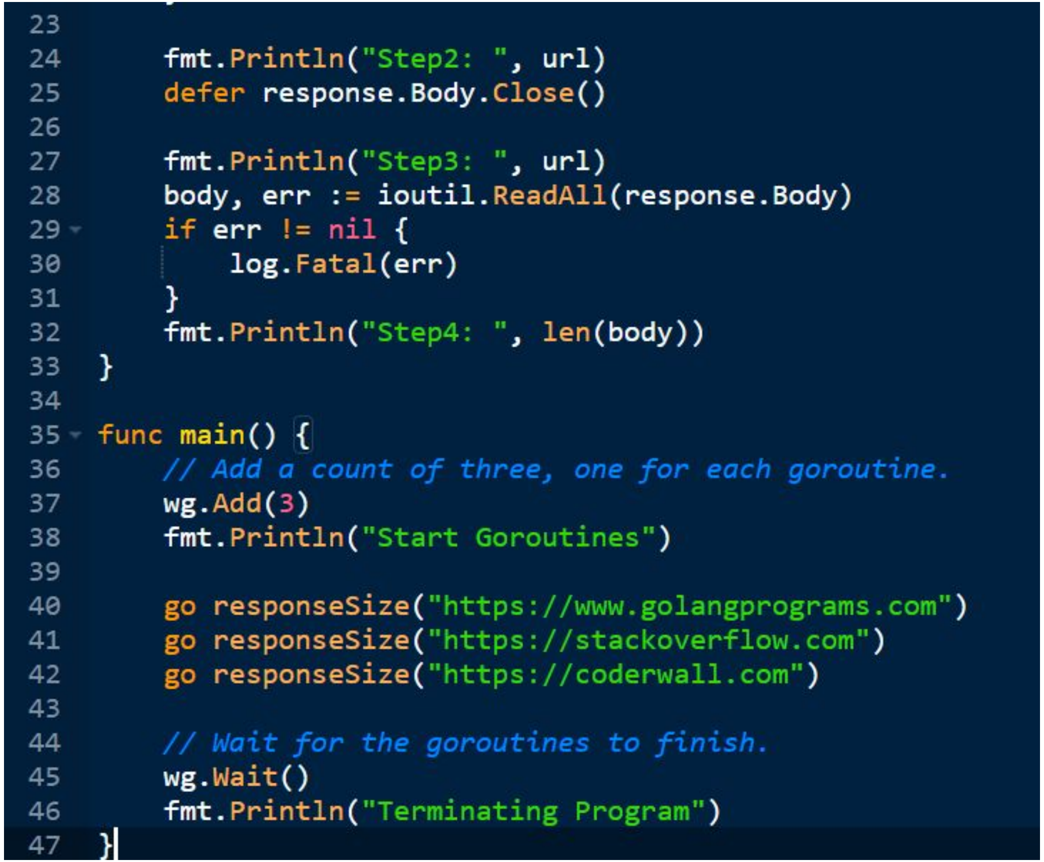
- The Wait method of the WaitGroup type waits for the program to finish all goroutines
- The Wait method is called inside the main function,
  - which blocks execution until the WaitGroup counter reaches the value of zero and ensures that all goroutines are executed

Example



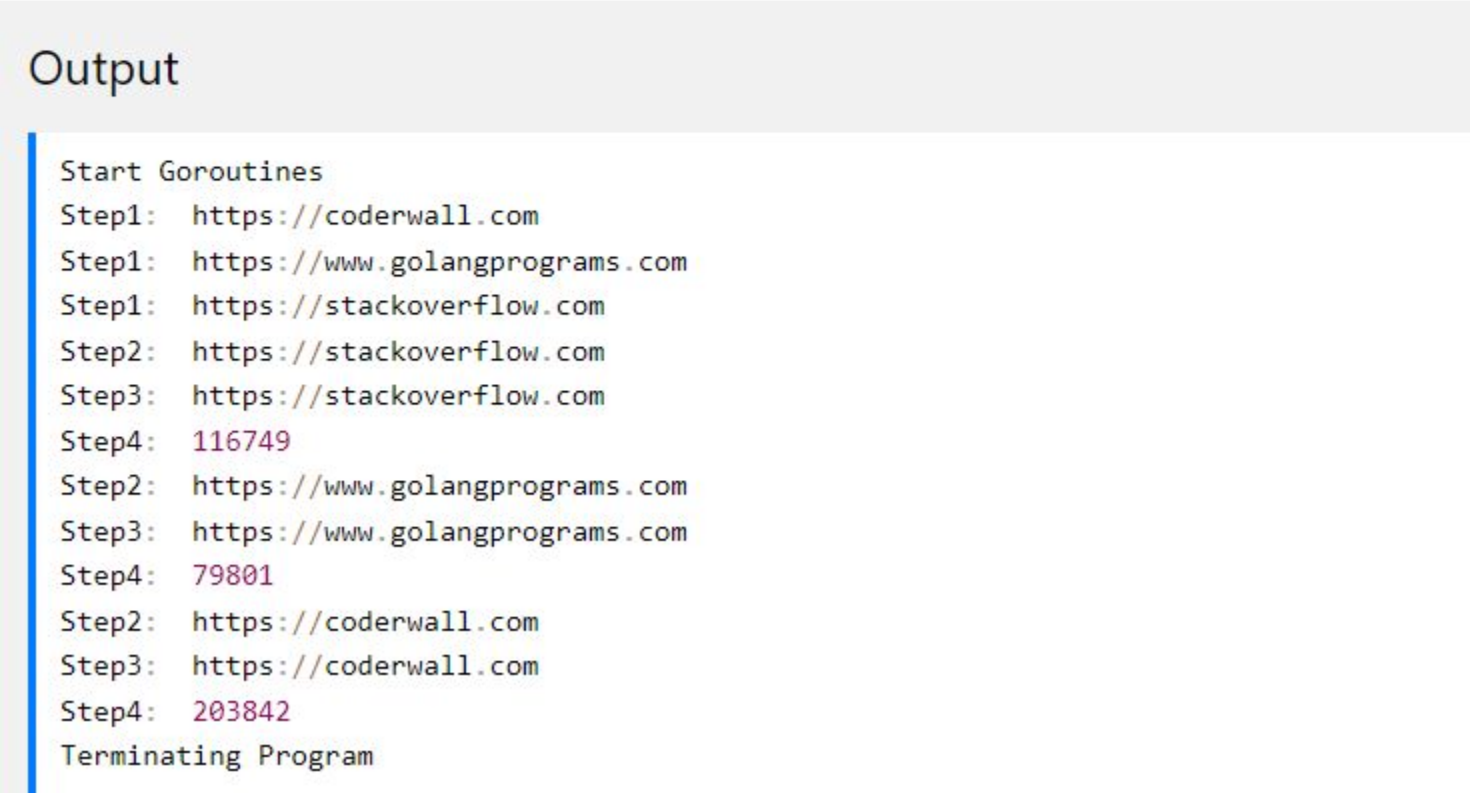
```go
package main

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

// WaitGroup is used to wait for the program to finish goroutines.
var wg sync.WaitGroup

func responseSize(url string) {
	// Schedule the call to WaitGroup's Done to tell goroutine is completed.
	defer wg.Done()

	fmt.Println("Step1: ", url)
	response, err := http.Get(url)
	if err != nil {
		log.Fatal(err)
	}

```

Example



```go

    fmt.Println("Step2: ", url)
    defer response.Body.Close()

    fmt.Println("Step3: ", url)
    body, err := ioutil.ReadAll(response.Body)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println("Step4: ", len(body))
}

func main() {
    // Add a count of three, one for each goroutine.
    wg.Add(3)
    fmt.Println("Start Goroutines")

    go responseSize("https://www.golangprograms.com")
    go responseSize("https://stackoverflow.com")
    go responseSize("https://coderwall.com")

    // Wait for the goroutines to finish.
    wg.Wait()
    fmt.Println("Terminating Program")
}
```

Example



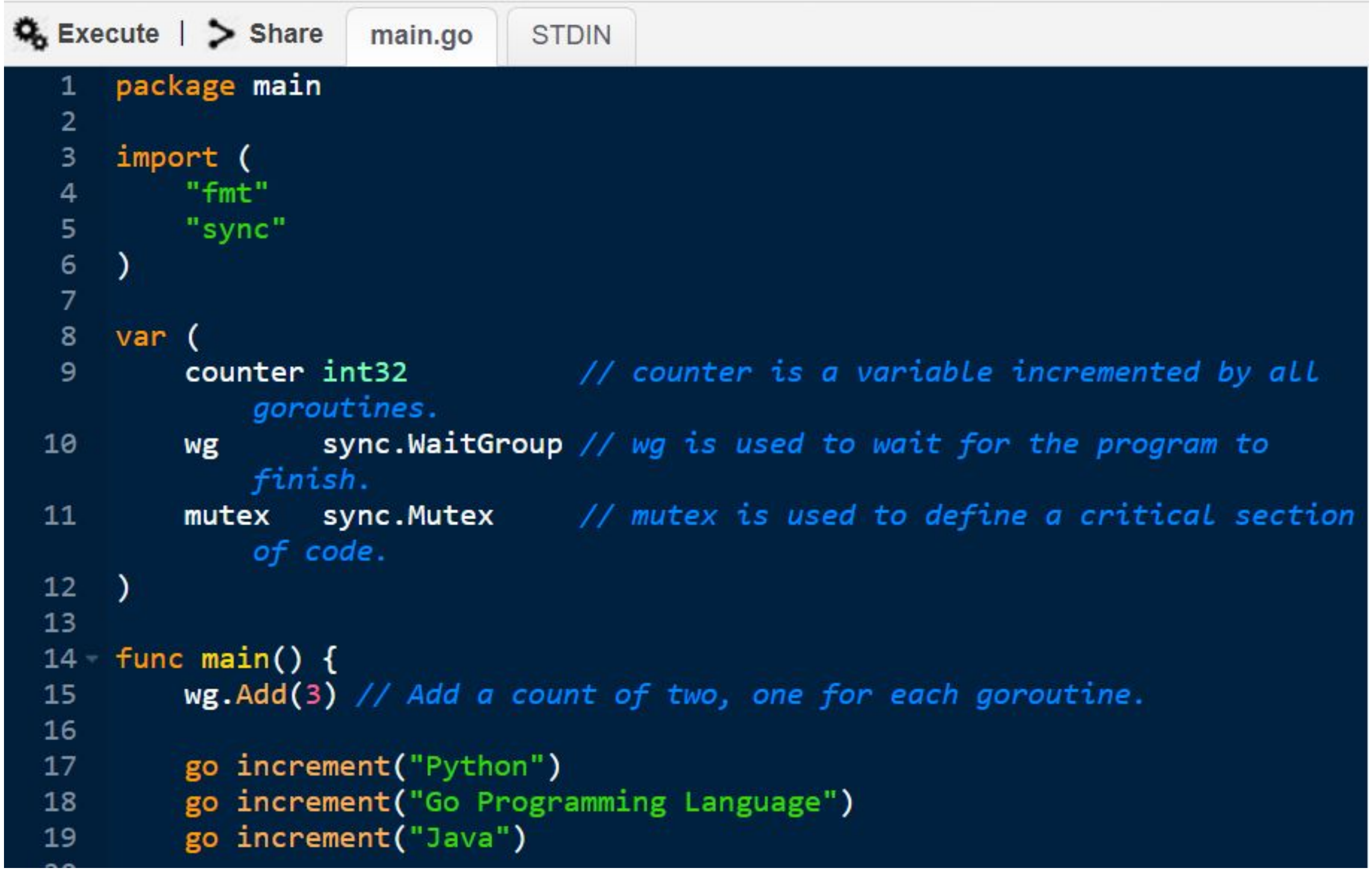
Output

```
Start Goroutines
Step1:  https://coderwall.com
Step1:  https://www.golangprograms.com
Step1:  https://stackoverflow.com
Step2:  https://stackoverflow.com
Step3:  https://stackoverflow.com
Step4:  116749
Step2:  https://www.golangprograms.com
Step3:  https://www.golangprograms.com
Step4:  79801
Step2:  https://coderwall.com
Step3:  https://coderwall.com
Step4:  203842
Terminating Program
```

Define Critical Sections using Mutex

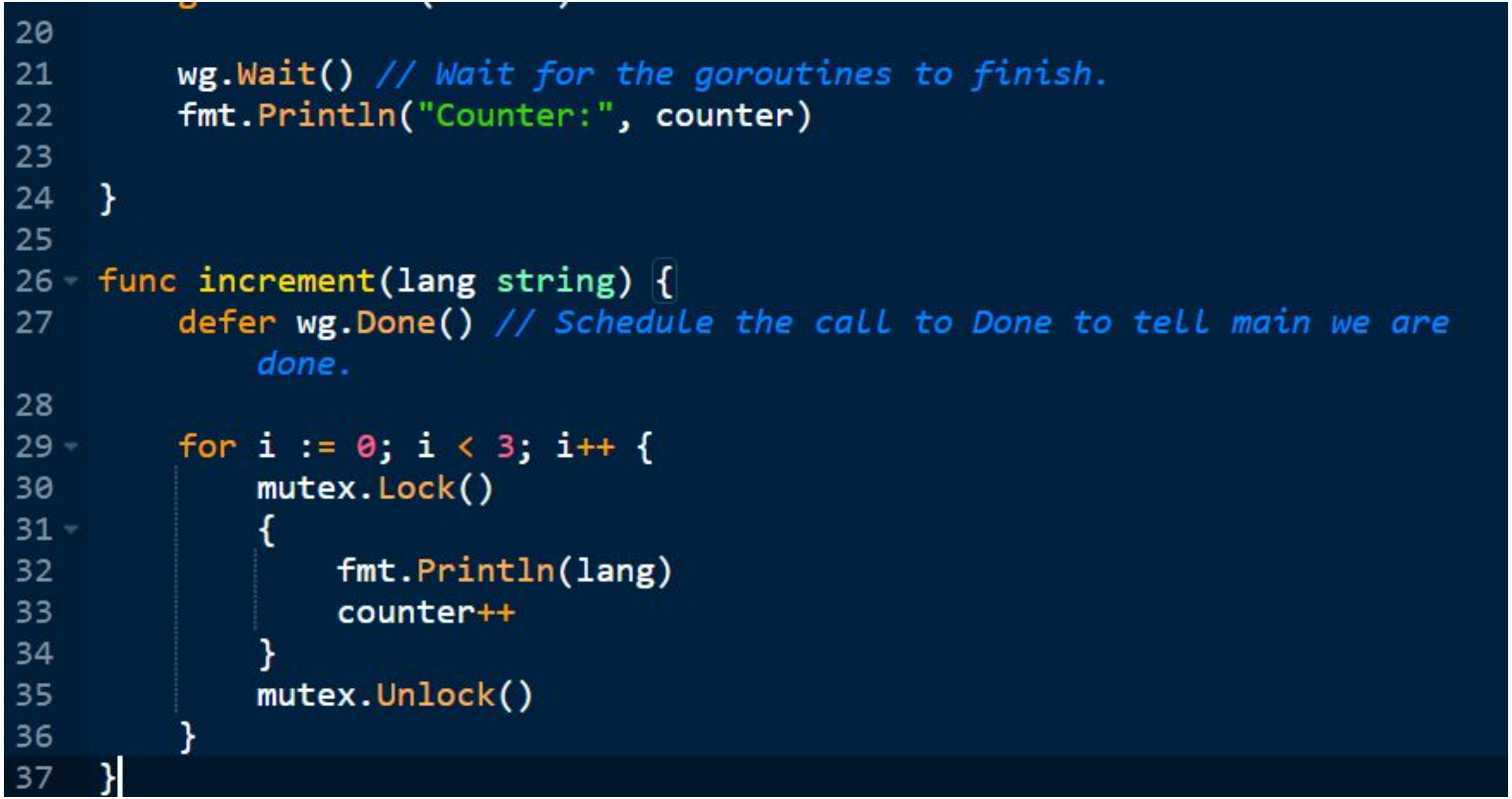
- A mutex is used to create a critical section around code that ensures only one goroutine at a time can execute that code section

Example



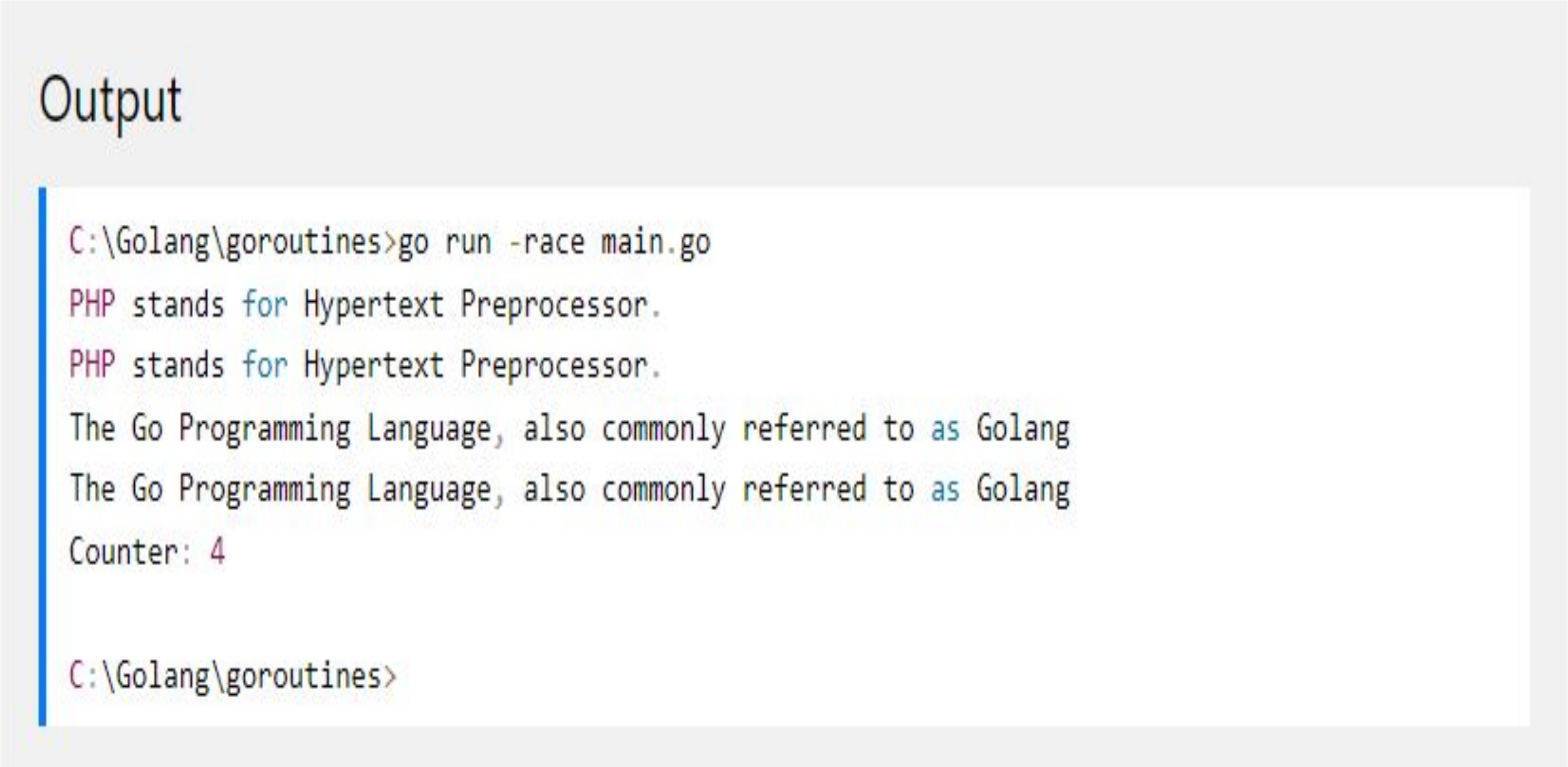
```go
package main

import (
    "fmt"
    "sync"
)

var (
    counter int32          // counter is a variable incremented by all goroutines.
    wg      sync.WaitGroup // wg is used to wait for the program to finish.
    mutex   sync.Mutex     // mutex is used to define a critical section of code.
)

func main() {
    wg.Add(3) // Add a count of two, one for each goroutine.

    go increment("Python")
    go increment("Go Programming Language")
    go increment("Java")
```

Example



```go

    wg.Wait() // Wait for the goroutines to finish.
    fmt.Println("Counter:", counter)

}

func increment(lang string) {
    defer wg.Done() // Schedule the call to Done to tell main we are done.

    for i := 0; i < 3; i++ {
        mutex.Lock()
        {
            fmt.Println(lang)
            counter++
        }
        mutex.Unlock()
    }
}
```

Example



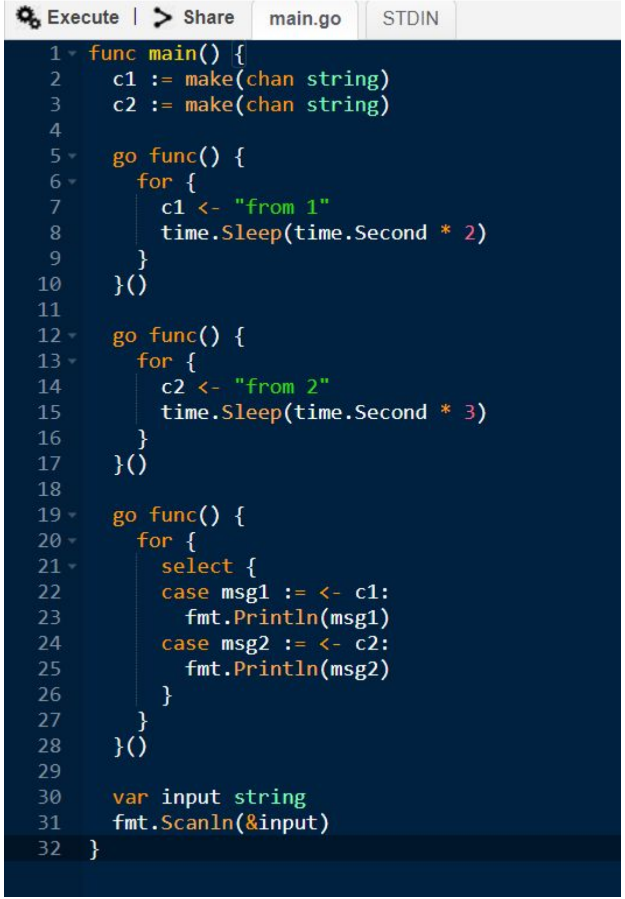
Output

```
C:\Golang\goroutines>go run -race main.go
PHP stands for Hypertext Preprocessor.
PHP stands for Hypertext Preprocessor.
The Go Programming Language, also commonly referred to as Golang
The Go Programming Language, also commonly referred to as Golang
Counter: 4


C:\Golang\goroutines>
```

Select

- ➢ Go has a special statement called select which works like a switch but for channels:

Select



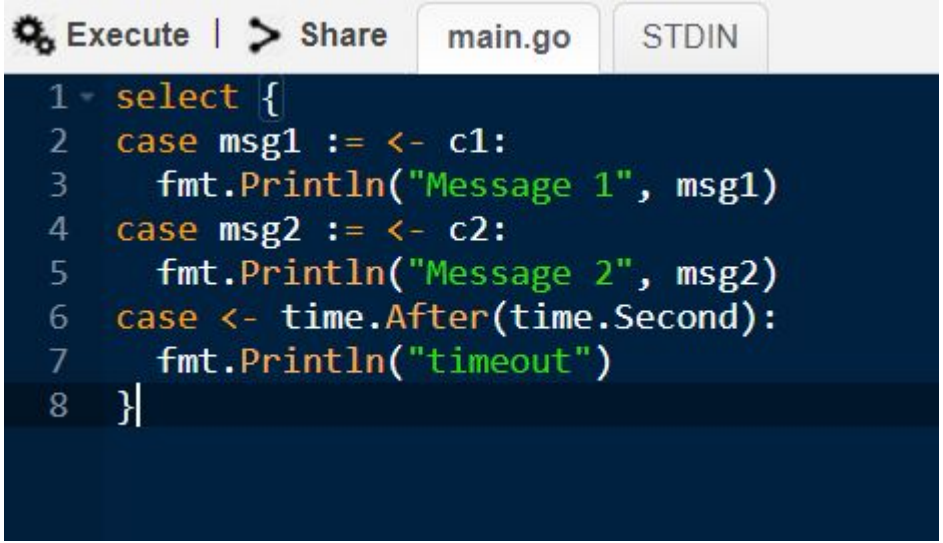
```go
func main() {
  c1 := make(chan string)
  c2 := make(chan string)

  go func() {
    for {
      c1 <- "from 1"
      time.Sleep(time.Second * 2)
    }
  }()

  go func() {
    for {
      c2 <- "from 2"
      time.Sleep(time.Second * 3)
    }
  }()

  go func() {
    for {
      select {
      case msg1 := <- c1:
        fmt.Println(msg1)
      case msg2 := <- c2:
        fmt.Println(msg2)
      }
    }
  }()

  var input string
  fmt.Scanln(&input)
}
```

Select

- This program prints “from 1” every 2 seconds and “from 2” every 3 seconds
- select picks the first channel that is ready and receives from it (or sends to it)
- If more than one of the channels are ready then it randomly picks which one to receive from
- If none of the channels are ready, the statement blocks until one becomes available

Select

- ➢ The select statement is often used to implement a timeout:

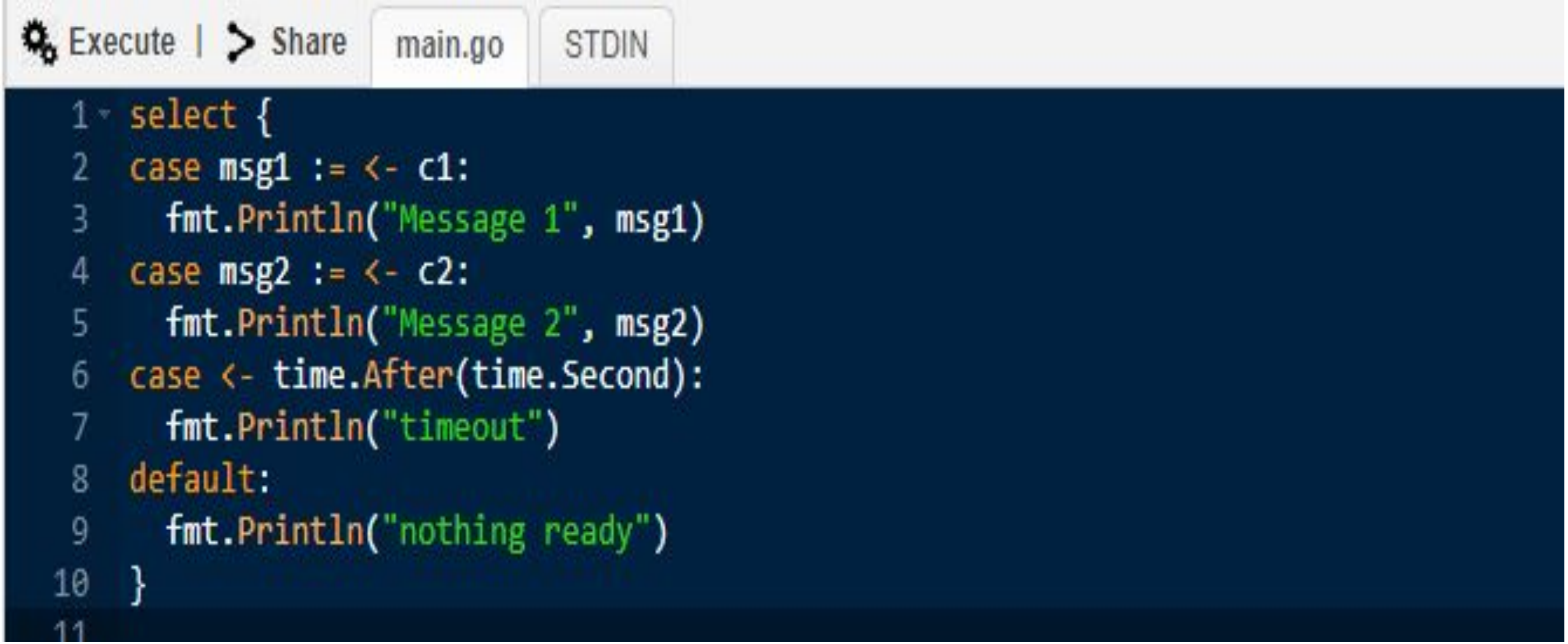


```
select {
case msg1 := <- c1:
  fmt.Println("Message 1", msg1)
case msg2 := <- c2:
  fmt.Println("Message 2", msg2)
case <- time.After(time.Second):
  fmt.Println("timeout")
}
```

Select

- ➢ time.After creates a channel and after the given duration will send the current time on it(we weren't interested in the time so we didn't store it in a variable)
- ➢ We can also specify a default case:

Select



```go
select {
case msg1 := <- c1:
  fmt.Println("Message 1", msg1)
case msg2 := <- c2:
  fmt.Println("Message 2", msg2)
case <- time.After(time.Second):
  fmt.Println("timeout")
default:
  fmt.Println("nothing ready")
}
```