

This is your **last** free member-only story this month. [Upgrade for unlimited access.](#)

GOLANG

# Anatomy of goroutines in Go -Concurrency in Go

goroutine is a lightweight execution thread running in the background. goroutines are key ingredients to achieve concurrency in Go.



Uday Hiwarale

Nov 4, 2018 · 7 min read ★



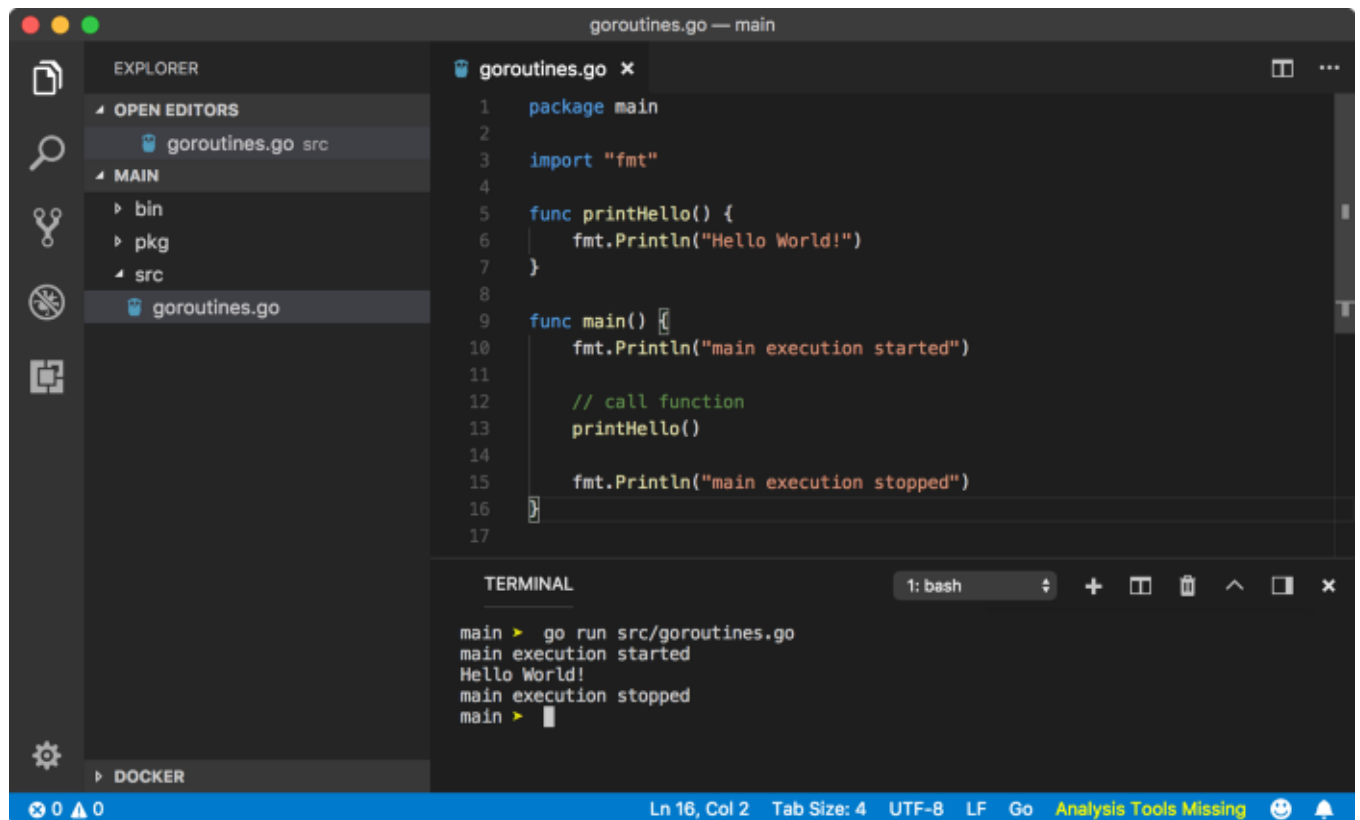
In the previous lesson, we learned about Go's concurrency model. As goroutines are lightweight compared to OS threads, it is very common for a Go application to have thousands of goroutines running concurrently. Concurrency can speed up application significantly as well as help us write code with separation of concerns (SoC).

## ▀ What is a goroutine?

We understood in theory that how goroutine works, but in code, what is it? Well, a goroutine is simply a function or method that is running in background concurrently

with other goroutines. It's not a function or method definition that determines if it is a goroutine, it is determined by how we call it.

Go provides a special keyword `go` to create a goroutine. When we call a function or a method with `go` prefix, that function or method executes in a goroutine. Let's see a simple example.



The screenshot shows the VS Code editor with a file named `goroutines.go` open. The Explorer sidebar on the left shows the project structure: `src` folder containing `goroutines.go`. The main editor window displays the following Go code:

```
1 package main
2
3 import "fmt"
4
5 func printHello() {
6     fmt.Println("Hello World!")
7 }
8
9 func main() {
10    fmt.Println("main execution started")
11
12    // call function
13    printHello()
14
15    fmt.Println("main execution stopped")
16 }
```


Below the code editor is a terminal window with the following output:

```
main > go run src/goroutines.go
main execution started
Hello World!
main execution stopped
main >
```

<https://play.golang.org/p/plGsToIA2hL>

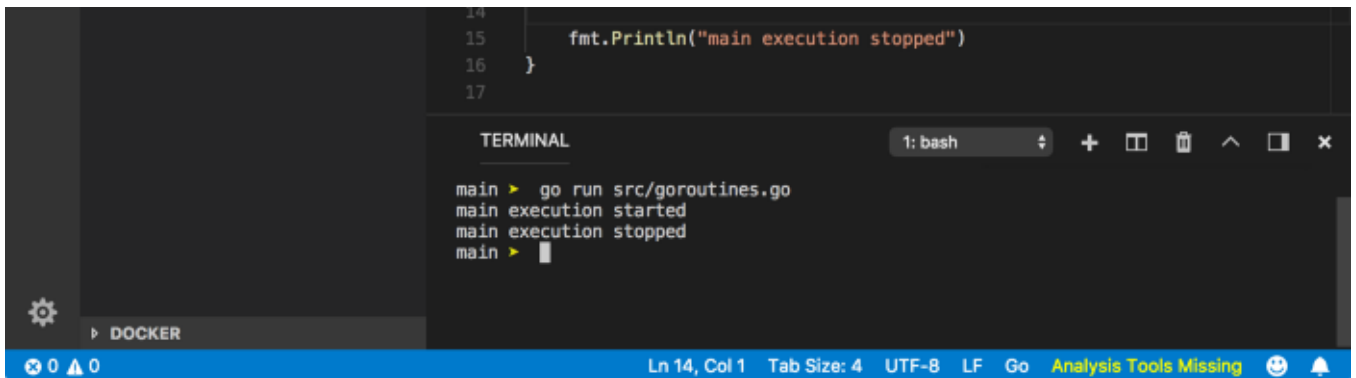
In the above program, we created a function `printHello` which prints `Hello World!` to the console. In `main` function, we called `printHello()` like a normal function call and we got the desired result.

Now let's create **goroutine** from the same `printHello` function.



The screenshot shows the VS Code editor with the same file `goroutines.go`. The code is modified to use a goroutine:

```
1 package main
2
3 import "fmt"
4
5 func printHello() {
6     fmt.Println("Hello World!")
7 }
8
9 func main() {
10    fmt.Println("main execution started")
11
12    // create goroutine
13    go printHello()
14 }
```

A screenshot of a Go Playground terminal window. The code editor at the top shows a Go program with lines 14 to 17. Line 15 contains `fmt.Println("main execution stopped")`. The terminal output shows the command `go run src/goroutines.go` being executed, with the output `main execution started` and `main execution stopped`. The terminal window has a title bar with '1: bash' and standard window controls. The bottom status bar shows 'Ln 14, Col 1', 'Tab Size: 4', 'UTF-8', 'LF', 'Go', and a yellow warning icon with the text 'Analysis Tools Missing'.

<https://play.golang.org/p/LWXAgDpTcJP>

Well, as per goroutine syntax, we prefixed function call with `go` keyword and program executed well. It yielded the following result.

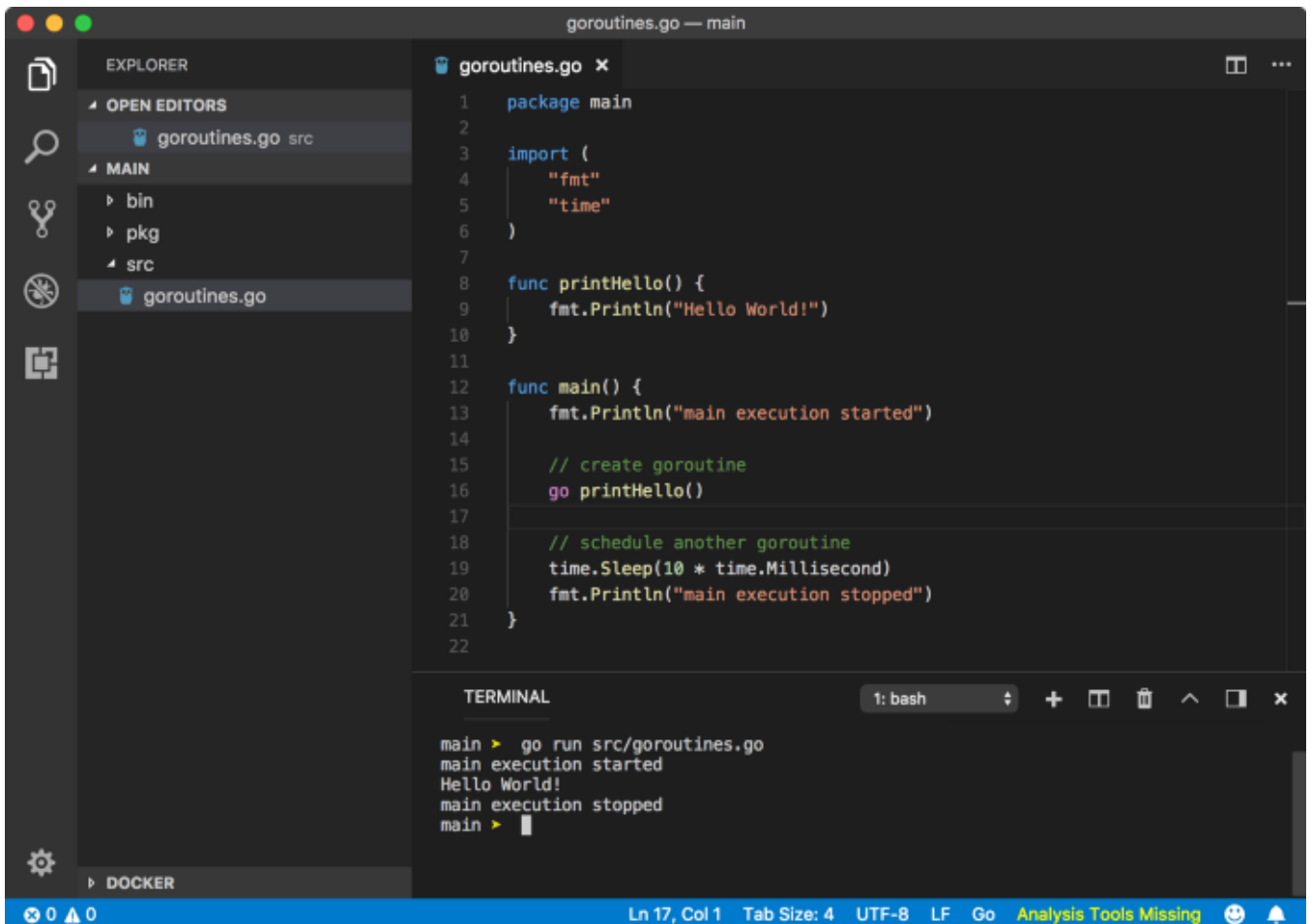
```
main execution started
main execution stopped
```

It is a bit strange that `Hello World` did not get printed. So what happened?

goroutines always run in the background. When a goroutine is executed, here, Go does not block the program execution, unlike normal function call as we have seen in the previous example. Instead, the control is returned immediately to the next line of code and any returned value from goroutine is ignored. **But even then, why we can't see the function output?**

By default, every Go standalone application creates one goroutine. This is known as the **main goroutine** that the `main` function operates on. In the above case, the main goroutine spawns another goroutine of `printHello` function, let's call it **printHello** goroutine. Hence when we execute the above program, there are two goroutines running concurrently. As we saw in the earlier program, goroutines are scheduled cooperatively.

Hence when the main goroutine starts executing, go scheduler does not pass control to the **printHello** goroutine until the main goroutine does not execute completely. Unfortunately, when the main goroutine is done with execution, the program terminates immediately and scheduler did not get time to schedule **printHello** goroutine. But as we know from other lessons, using blocking condition, we can pass control to other goroutines manually AKA telling the scheduler to schedule other available goroutines. Let's use `time.Sleep()` call to do it.



```
goroutines.go — main
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func printHello() {
9     fmt.Println("Hello World!")
10 }
11
12 func main() {
13     fmt.Println("main execution started")
14
15     // create goroutine
16     go printHello()
17
18     // schedule another goroutine
19     time.Sleep(10 * time.Millisecond)
20     fmt.Println("main execution stopped")
21 }
22
```

```
1: bash
main > go run src/goroutines.go
main execution started
Hello World!
main execution stopped
main >
```

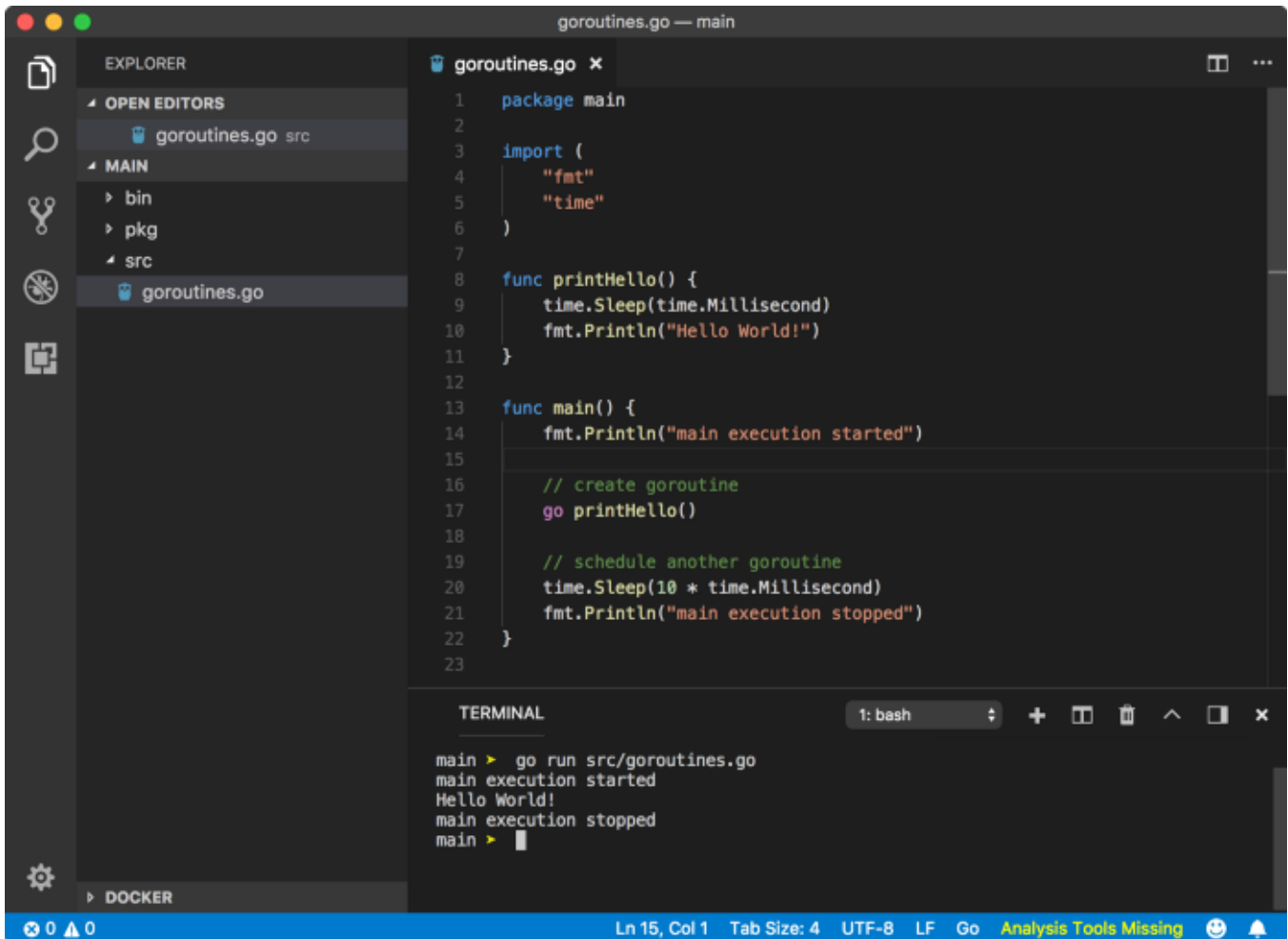
<https://play.golang.org/p/ujQKjpALIRJ>

We have modified program in such a way that before **main** goroutine pass control to the last line of code, we pass control to **printHello** goroutine using `time.Sleep(10 * time.Millisecond)` call. In this case, the main goroutine sleeps for 10 milli-seconds and won't be scheduled again for another 10 milliseconds. Once **printHello** goroutine executes, it prints '**Hello World!**' to the console and terminates, then the main goroutine is scheduled again (*after 10 milliseconds*) to execute the last line of code **where stack pointer is**. Hence the above program yields the following result.

```
main execution started
Hello World!
main execution stopped
```

If we add a sleep call inside the function which will tell goroutine to schedule another available goroutine, in this case, the main goroutine. But from the last lesson, we learned that only non-sleeping goroutines are considered for scheduling, main won't be scheduled again for 10 milli-seconds while it's sleeping.

Hence the main goroutine will print 'main execution started', spawning **printHello** goroutine but still actively running, then sleeping for 10 milli-seconds and passing control to **printHello** goroutine. **printHello** goroutine then will sleep for 1 milli-second telling the scheduler to schedule another goroutine but since there isn't any available, waking up after 1 milli-second and printing 'Hello World!' and then dying. Then the main goroutine will wake up after a few milliseconds, printing 'main execution stopped' and exiting the program.



The screenshot shows the Visual Studio Code editor with a Go file named `goroutines.go` open. The file contains the following code:

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func printHello() {
9     time.Sleep(time.Millisecond)
10    fmt.Println("Hello World!")
11 }
12
13 func main() {
14     fmt.Println("main execution started")
15
16     // create goroutine
17     go printHello()
18
19     // schedule another goroutine
20     time.Sleep(10 * time.Millisecond)
21     fmt.Println("main execution stopped")
22 }
23
```

The terminal output shows the execution of the program:

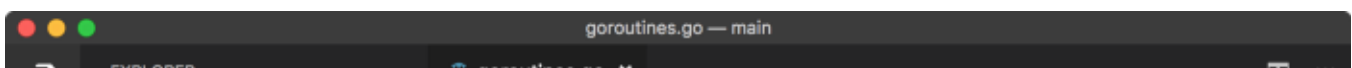
```
main > go run src/goroutines.go
main execution started
Hello World!
main execution stopped
main >
```

<https://play.golang.org/p/rWvzS8UeqD6>

Above program will still print the same result

```
main execution started
Hello World!
main execution stopped
```

What if, instead of 1 milli-second, **printHello** goroutine sleeps for 15 milliseconds.



```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func printHello() {
9     time.Sleep(15 * time.Millisecond)
10    fmt.Println("Hello World!")
11 }
12
13 func main() {
14     fmt.Println("main execution started")
15
16     // create goroutine
17     go printHello()
18
19     // schedule another goroutine
20     time.Sleep(10 * time.Millisecond)
21     fmt.Println("main execution stopped")
22 }
23
```

TERMINAL

```
1: bash
main > go run src/goroutines.go
main execution started
main execution stopped
main >
```

<https://play.golang.org/p/Pc2nP2BtRiP>

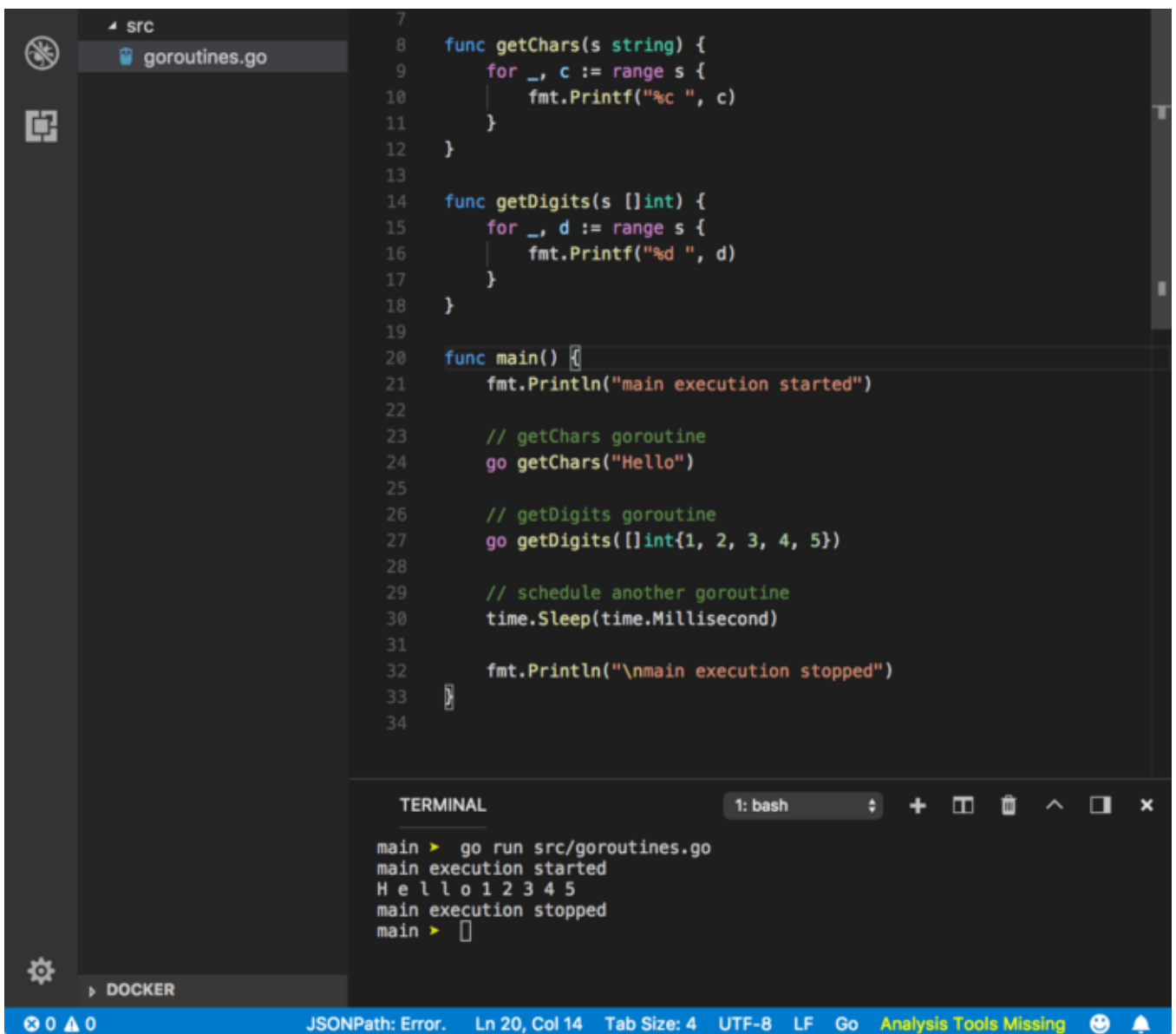
In that case, the main goroutine will be available to schedule for scheduler before **printHello** goroutine wakes up, which will also terminate the program immediately before scheduler had time to schedule **printHello** goroutine again. Hence it will yield below program

```
main execution started
main execution stopped
```

## Working with multiple goroutines

As I said earlier, you can create as many goroutines as you can. Let's define two simple functions, one prints characters of the string and another prints digit of the integer slice.

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
```



```
7
8 func getChars(s string) {
9     for _, c := range s {
10         fmt.Printf("%c ", c)
11     }
12 }
13
14 func getDigits(s []int) {
15     for _, d := range s {
16         fmt.Printf("%d ", d)
17     }
18 }
19
20 func main() {
21     fmt.Println("main execution started")
22
23     // getChars goroutine
24     go getChars("Hello")
25
26     // getDigits goroutine
27     go getDigits([]int{1, 2, 3, 4, 5})
28
29     // schedule another goroutine
30     time.Sleep(time.Millisecond)
31
32     fmt.Println("\nmain execution stopped")
33 }
34
```

TERMINAL

```
1: bash
main > go run src/goroutines.go
main execution started
H e l l o 1 2 3 4 5
main execution stopped
main >
```

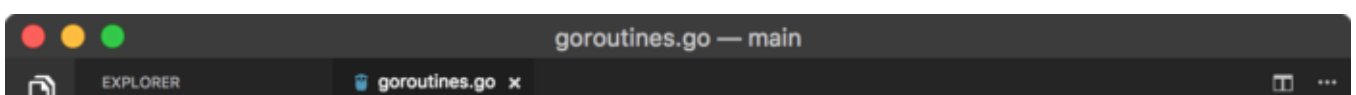
JSONPath: Error. Ln 20, Col 14 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing

[https://play.golang.org/p/SJano\\_g1wTV](https://play.golang.org/p/SJano_g1wTV)

In the above program, we are creating 2 goroutines from 2 function calls in series. Then we are scheduling any of the two goroutines and which goroutines to schedule is determined by the scheduler. This will yield the following result

```
main execution started
H e l l o 1 2 3 4 5
main execution stopped
```

Above result again proves that goroutines are cooperatively scheduled. Let's add another `time.Sleep` call in-between print operation in the function definition to tell the scheduler to schedule other available goroutines.





```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 var start time.Time
9
10 func init() {
11     start = time.Now()
12 }
13
14 func getChars(s string) {
15     for _, c := range s {
16         fmt.Printf("%c at time %v\n", c, time.Since(start))
17         time.Sleep(10 * time.Millisecond)
18     }
19 }
20
21 func getDigits(s []int) {
22     for _, d := range s {
23         fmt.Printf("%d at time %v\n", d, time.Since(start))
24         time.Sleep(30 * time.Millisecond)
25     }
26 }
27
28 func main() {
29     fmt.Println("main execution started at time", time.Since(start))
30
31     // getChars goroutine
32     go getChars("Hello")
33
34     // getDigits goroutine
35     go getDigits([]int{1, 2, 3, 4, 5})
36
37     // schedule another goroutine
38     time.Sleep(200 * time.Millisecond)
39
40     fmt.Println("\nmain execution stopped at time", time.Since(start))
41 }
42
```

TERMINAL

```
1: bash
main > go run src/goroutines.go
main execution started at time 577ns

1 at time 65.782µs
H at time 52.709µs
e at time 11.266014ms
l at time 23.853536ms
2 at time 30.274127ms
l at time 34.478401ms
o at time 44.626742ms
3 at time 60.878146ms
4 at time 91.374975ms
5 at time 122.245245ms

main execution stopped at time 201.25904ms
main >
```

<https://play.golang.org/p/lrSIEdNxSaH>

In the above program, we printed extra information to see when a print statement is executing since the time of execution of the program. In theory, the main goroutine will sleep for 200 milliseconds, hence all other goroutines must do their job in 200 milliseconds before it wakes up and kills the program. `getChars` goroutine will print 1 character and sleep for 10 milli-second, passing control to `getDigits` goroutine which will print a digit and sleeping for 3 milli-seconds passing control to `getChars` goroutine again when it wakes up. Since `getChars` goroutine can **print and sleep** multiple times,



at least 2 times while other goroutines are sleeping, we are hoping to see more characters printed in succession than digits.

Below result is taken from running above program in **Windows** machine.

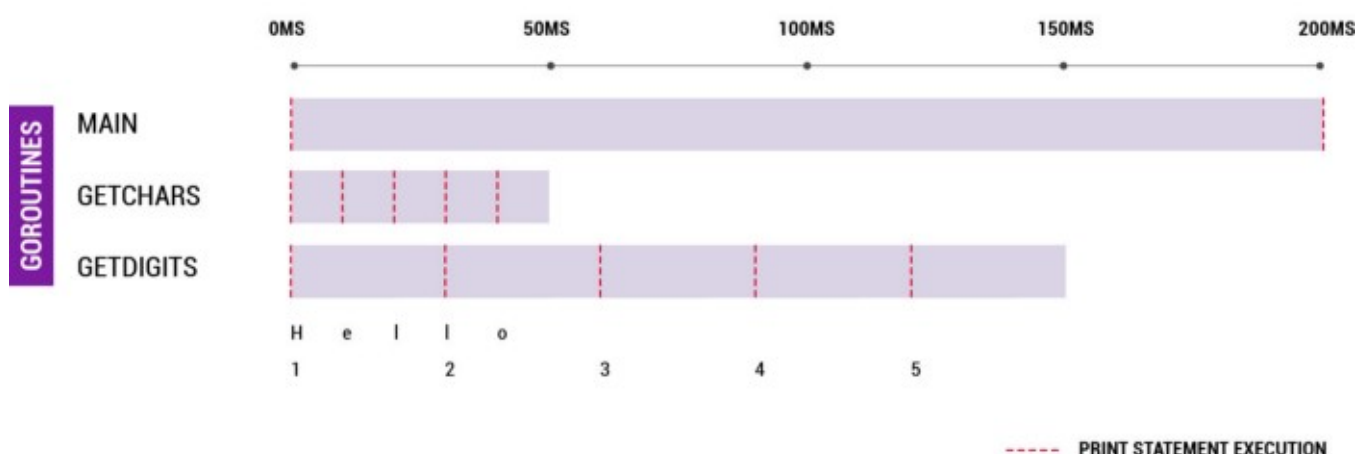
```
main execution started at time 0s
```

```
H at time 1.0012ms      <--|
1 at time 1.0012ms      | almost at the same
time
e at time 11.0283ms     <--|
l at time 21.0289ms     | ~10ms apart
l at time 31.0416ms
2 at time 31.0416ms
o at time 42.0336ms
3 at time 61.0461ms     <--|
4 at time 91.0647ms     | ~30ms apart
5 at time 121.0888ms
```

```
main execution stopped at time 200.3137ms | exiting after 200ms
```

We can see the pattern we talked about. This will be cleared to you once you see the program execution diagram. We will approximate that print command takes 1ms of CPU time, compared on the 200ms scale, that's negligible.

## MULTIPLE GOROUTINES EXECUTION

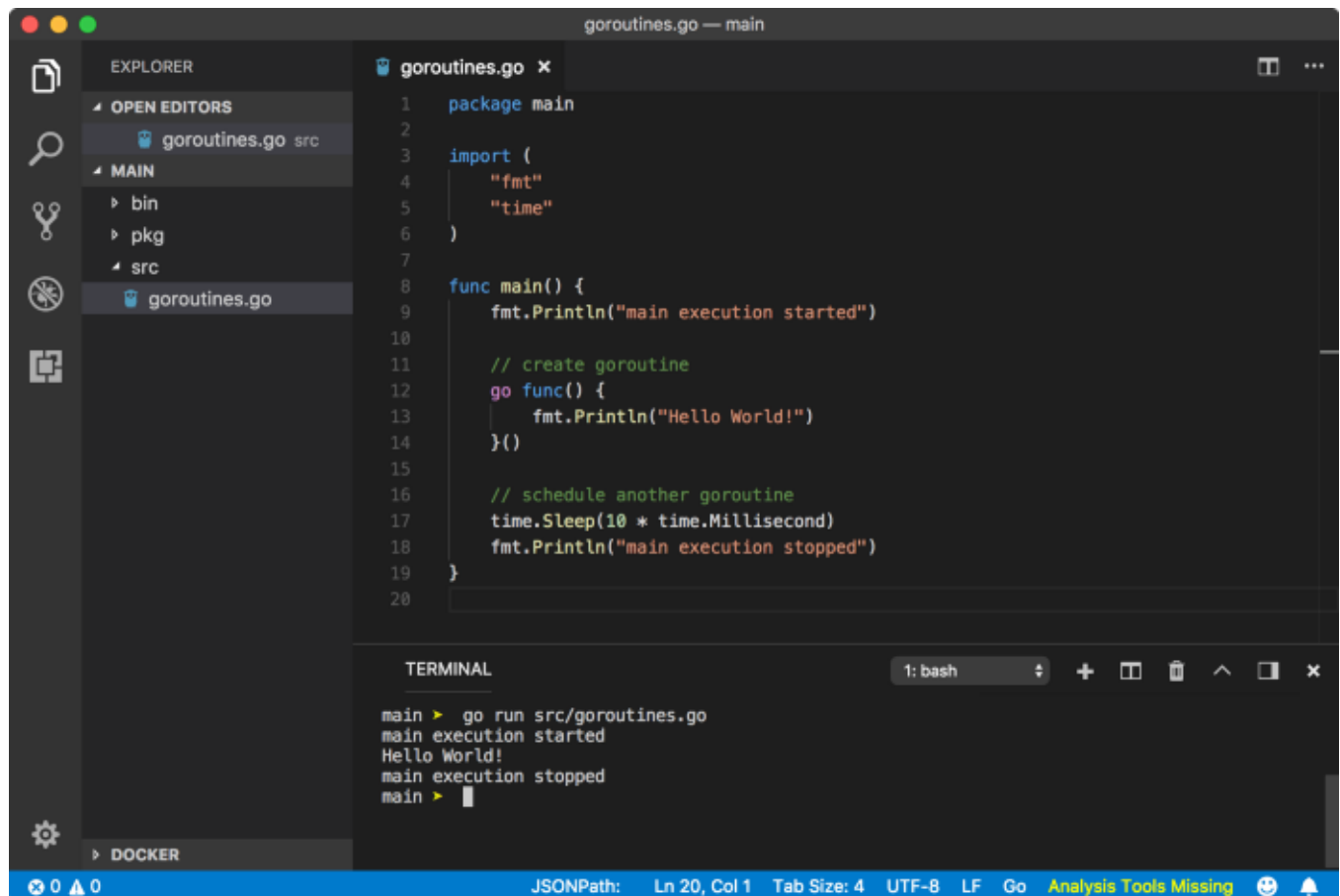


Now we understood how to create goroutine and how to work with them. But using `time.Sleep` is just a hack to see the result. In production, we don't know how much time a goroutine is going to take for the execution. Hence we can't just add random sleep call in the main function. We want our goroutines to tell when they finished the execution. Also at this point, we don't know how we can get data back from other

goroutines or pass data to them, simply, communicate with them. This is where **channels** comes in. Let's talk about them in the next lesson.

## Anonymous goroutines

If an anonymous function can exist then anonymous goroutine can also exist. Please read Immediately invoked function from functions lesson to understand this section. Let's modify our earlier example of `printHello` goroutine.



The screenshot shows a VS Code editor window titled "goroutines.go — main". The Explorer sidebar on the left shows the project structure with "goroutines.go" in the "src" directory. The main editor displays the following Go code:

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func main() {
9     fmt.Println("main execution started")
10
11     // create goroutine
12     go func() {
13         fmt.Println("Hello World!")
14     }()
15
16     // schedule another goroutine
17     time.Sleep(10 * time.Millisecond)
18     fmt.Println("main execution stopped")
19 }
20
```

The TERMINAL panel at the bottom shows the command `go run src/goroutines.go` and its output:

```
main > go run src/goroutines.go
main execution started
Hello World!
main execution stopped
main >
```

<https://play.golang.org/p/KSzsPluG-Ph>

The result is quite obvious as we defined the function and executed as goroutine in the same statement.

*All goroutines are anonymous as we learned from concurrency lesson as goroutine does not have an identity. But we are calling that in the sense that function from which it was created was anonymous.*





[thisisuday.com](https://thisisuday.com) / [GitHub](#) / [Twitter](#) / [StackOverflow](#) / [Instagram](#)

**Please clap 🖐️ if you found this article *useful*.**

Some rights reserved 

[Golang](#)   [Go Programming](#)   [Go Programming Language](#)   [Golang Tutorial](#)   [Concurrency](#)

[About](#)   [Help](#)   [Legal](#)

Get the Medium app

