Anatomy of Channels in Go - Concurrency in Go

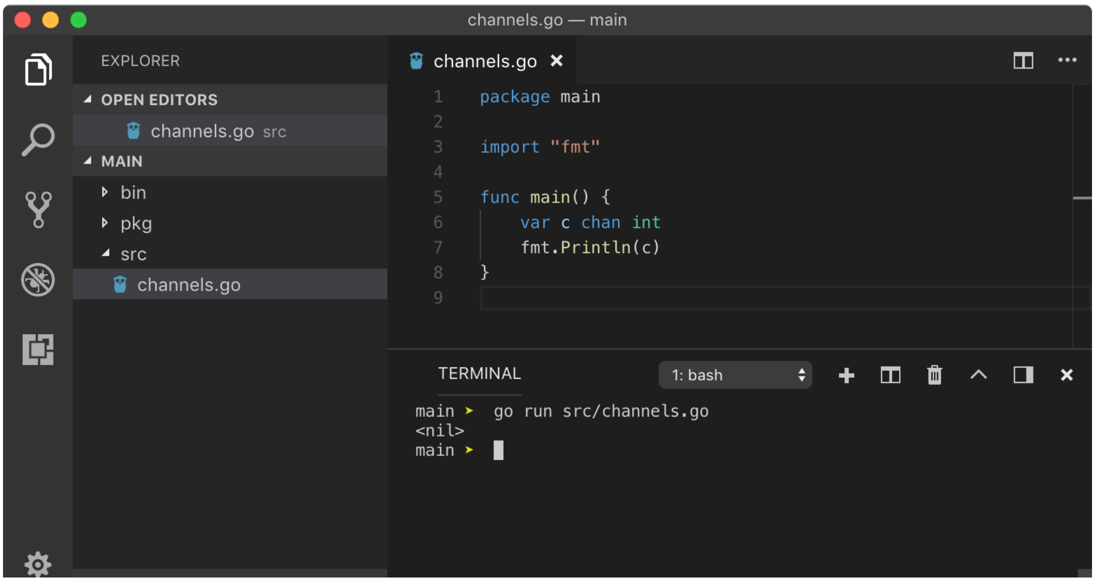
Uday Hiwarale Follow
Nov 19, 2018 · 26 min read

What are the channels?

A **channel** is a communication object using which goroutines can communicate with each other. Technically, a channel is a data transfer pipe where data can be **passed into** or **read from**. Hence one goroutine can send data into a channel, while other goroutines can read that data from the same channel.

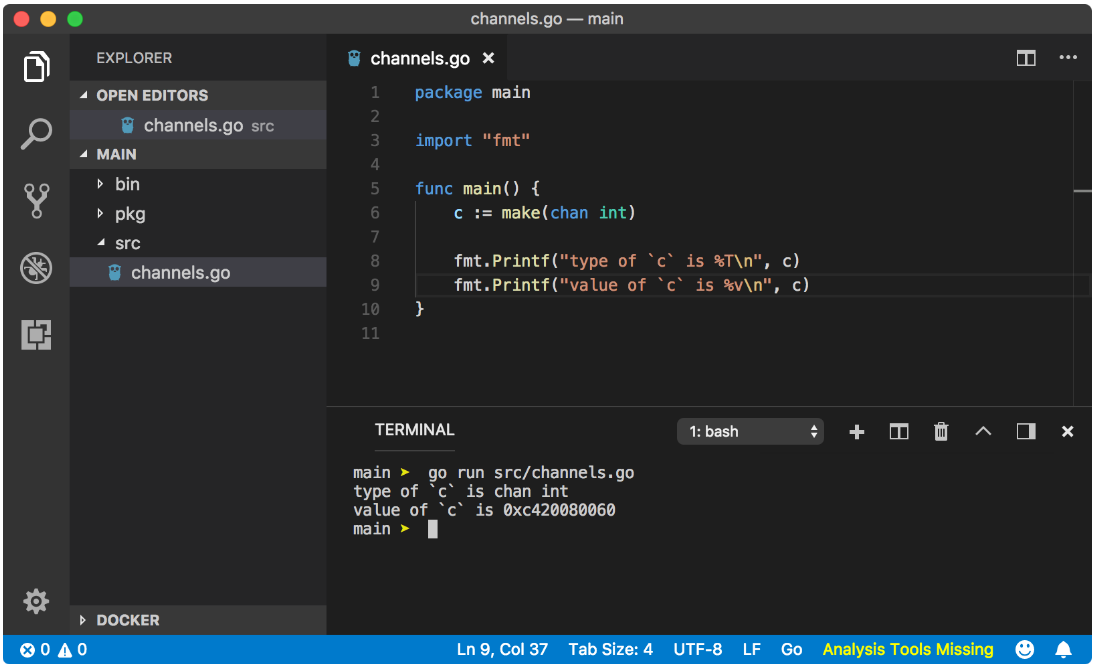
Declaring a channel

Go provides `chan` keyword to create a channel. A channel can transport data of **only one data type**. No other data types are allowed to be transported from that channel.



https://play.golang.org/p/iWOFLfcgfF-

Above program declares a channel `c` which can transport data type of `int` . Above program prints `<nil>` because **zero-value** of a channel is `nil` . But a `nil` channel is not useful. You can not pass data to or read data from a channel which is `nil` . Hence, we have to use `make` function to create a ready-to-use channel.



https://play.golang.org/p/N4dU7Ql9bK7

We have used short-hand syntax `:=` to make a channel using `make` function. The above program yields the following result.

```
type of `c` is chan int
value of `c` is 0xc0420160c0
```

Notice value of the channel `c` . Looks like it is a memory address. Channels by default are **pointers**. Mostly, when you want to communicate with a goroutine, you pass the channel as an argument to the function or method. Hence when goroutine receives that channel as an argument, you don't need to dereference it to push or pull data from that channel.

Data read and write

Go provide very easy to remember **left arrow syntax** `<-` to read and write data from a channel.

```
c <- data
```

Above syntax means, we want to push or write `data` to the channel `c` . Look at the direction of the arrow. It points from `data` to channel `c` . Hence we can imagine that we are trying to push `data` to `c` .

```
<- c
```

Above syntax means, we want to read some data from channel `c` . Look at the direction of the arrow, it starts from the channel `c` . This statement does not push data into anything, but still, it's a valid statement. If you have a variable that can hold the data coming from the channel, you can use below syntax

```
var data int
data = <- c
```

Now data coming from the channel `c` which is of type `int` can be stored into the variable `data` of type `int` .

Above syntax can be re-written using shorthand syntax as below

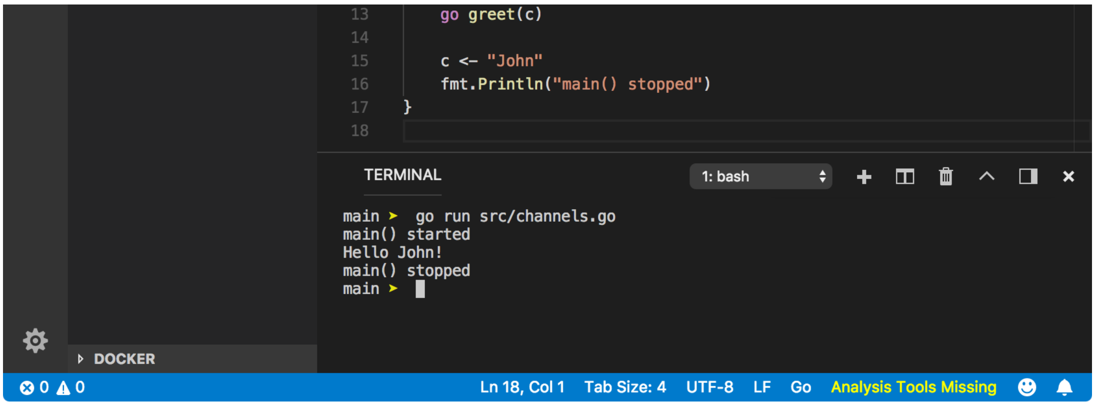
```
data := <- c
```

Go will figure out the data type of data being transported in channel `c` and gives `data` a valid data type.

All the above channel operations are blocking by default. In the previous lesson, we saw `time.Sleep` blocking a goroutine. Channel operations are also blocking in nature. When some data is written to the channel, goroutine is blocked until some other goroutine reads it from that channel. At the same time, as we seen in `concurrency chapter` , channel operations tell the scheduler to schedule another goroutine, that's why a program doesn't block forever on the same goroutine. These features of a channel are very useful in goroutines communication as it prevents us from writing manual locks and hacks to make them work with each other.

Channels in practice

Enough talking among us, let's talk to a goroutine.



https://play.golang.org/p/OeYLKEz7qKi

Let's talk about the execution of the above program step by step.

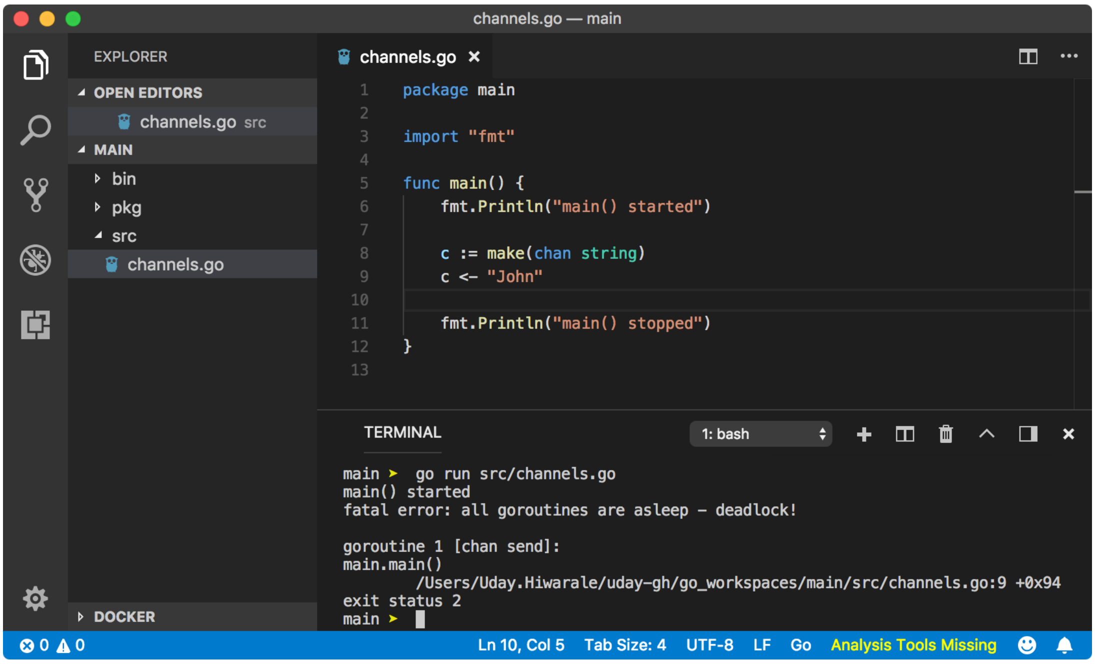
- We declared `greet` function which accepts a channel `c` of transport data type `string`. In that function, we are reading data from the channel `c` and printing that data to the console.
- In the main function, program prints `main started` to the console as it is the first statement.
- Then we made the channel `c` of type `string` using `make` function.
- We passed channel `c` to the `greet` function but executed it as a goroutine using `go` keyword.
- At this point, the process has 2 goroutines while active goroutine is `main goroutine` (*check the previous lesson to know what it is*). Then control goes to the next line.
- We pushed a string value `John` to channel `c`. At this point, goroutine is blocked until some goroutine reads it. Go scheduler schedule `greet` goroutine and it's execution starts as per mentioned in the first point.
- Then `main goroutine` becomes active and execute the final statement, printing `main stopped`.

Deadlock

As discussed, when we write or read data from a channel, that goroutine is blocked and control is passed to available goroutines. What if there are no other goroutines available, imagine all of them are sleeping. That's where `deadlock` error occurs crashing the whole program.

> *If you are trying to read data from a channel but channel does not have a value available with it, it blocks the current goroutine and unblocks other in a hope that some goroutine will push a value to the channel. Hence, **this read operation will be blocking**. Similarly, if you are to send data to a channel, it will block current goroutine and unblock others until some goroutine reads the data from it. Hence, **this send operation will be blocking**.*

A simple example of deadlock would be only main goroutine doing some channel operation.



https://play.golang.org/p/2KTEoljdci_f

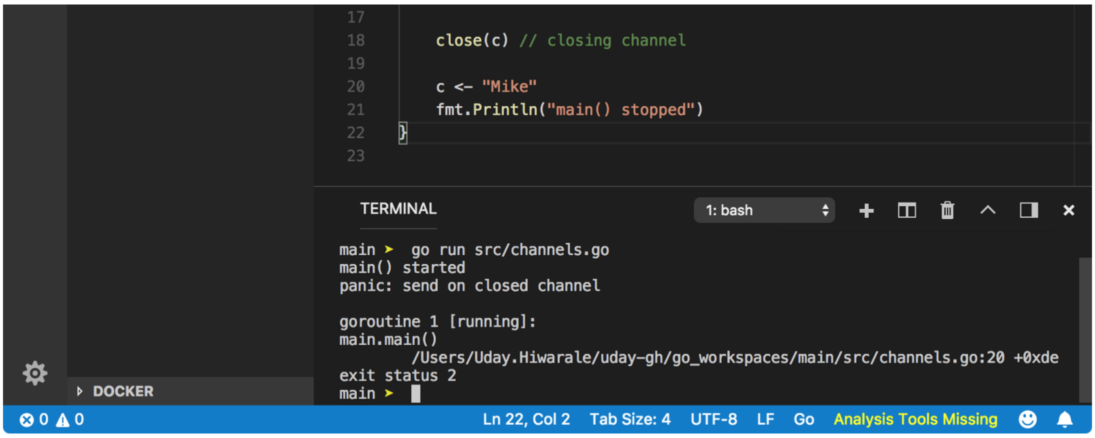
Above program will throw below error in runtime.

```
main() started
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:
main.main()
        program.Go:10 +0xfd
exit status 2
```

fatal error: all goroutines are asleep — deadlock!. Seems like all goroutines are asleep or simply no other goroutines are available to schedule.

☛ Closing a channel

A channel can be closed so that no more data can be sent through it. Receiver goroutine can find out the state of the channel using `val, ok := <- channel` syntax where `ok` is `true` if the channel is **open** or **read operations can be performed** and `false` if the channel is **closed** and **no more read operations can be performed**. A channel can be closed using `close` built-in function with syntax `close(channel)`. Let's see a simple example.



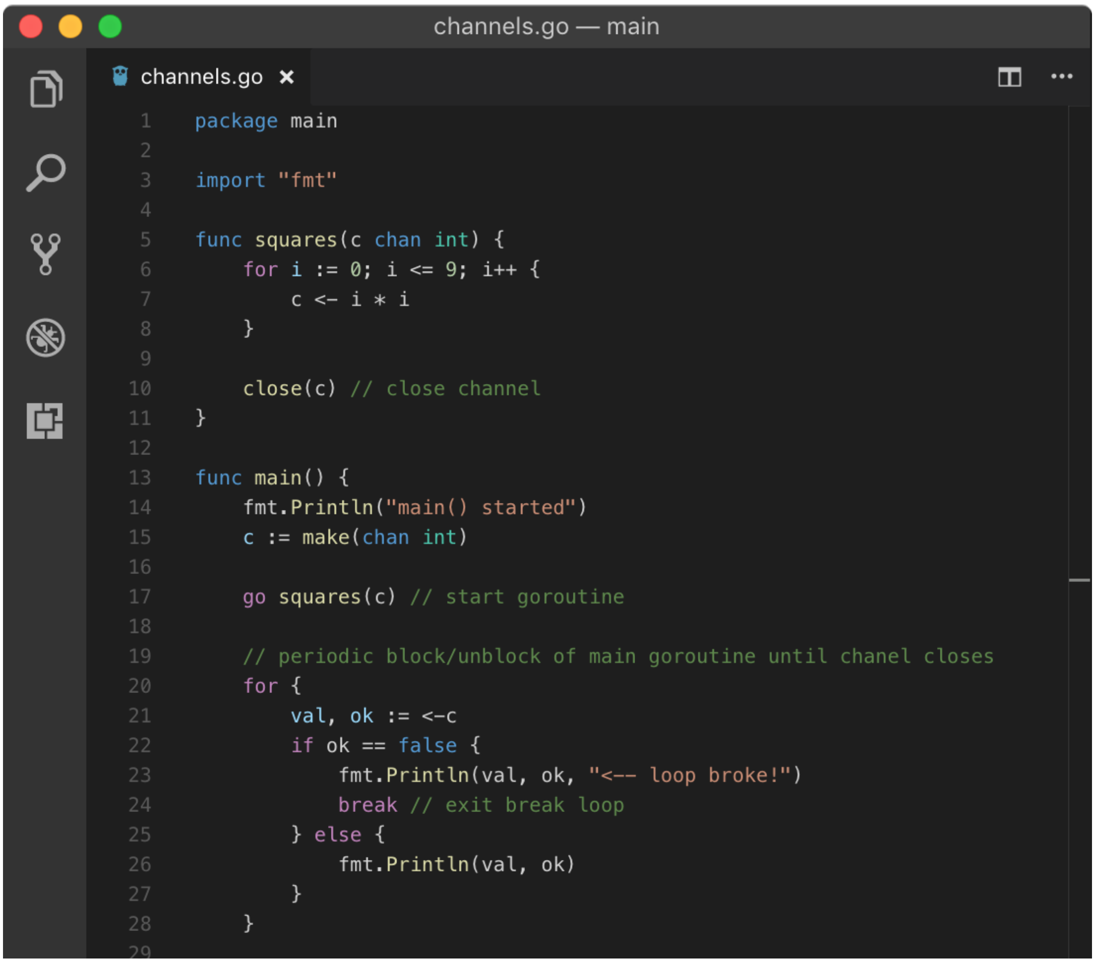
https://play.golang.org/p/LMmAq4sgm02

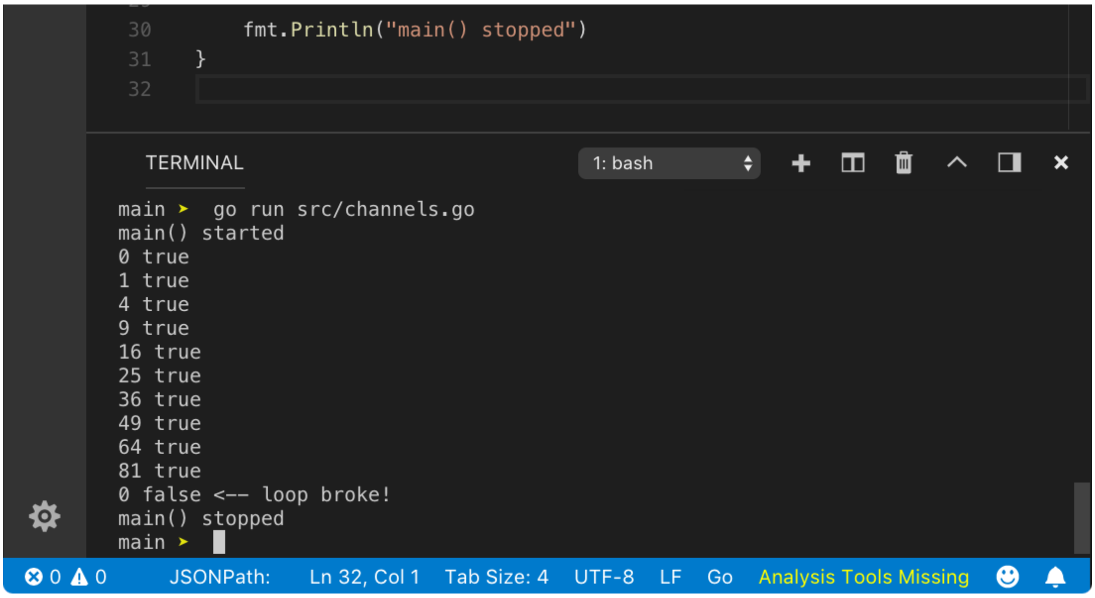
> *Just to help you understand blocking concept, first send operation* `c <- "John"` *is blocking and some goroutine has to read data from the channel, hence* `greet` *goroutine is scheduled by the Go Scheduler. Then first read operation* `<-c` *is non-blocking because data is present in channel* `c` *to be read from. Second read operation* `<-c` *will be blocking because channel* `c` *does not have any data to be read from, hence Go Scheduler activates* `main` *goroutine and program starts execution from* `close(c)` *function.*

From the above error, we can see that we are trying to send data on a closed channel. To better understand the usability of **closed channels**, let's see `for` loop.

☛ For loop

An **infinite** syntax `for` loop `for{}` can be used to read multiple values sent through a channel.





https://play.golang.org/p/X58FTgSHhXi

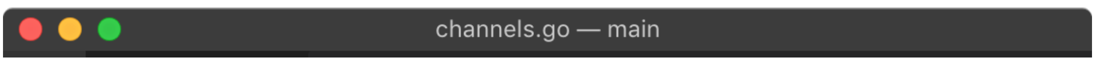
In the above example, we are creating goroutine `squares` which returns squares of numbers from `0` to `9` one by one. In `main` goroutine, we are reading those numbers inside infinite for loop.

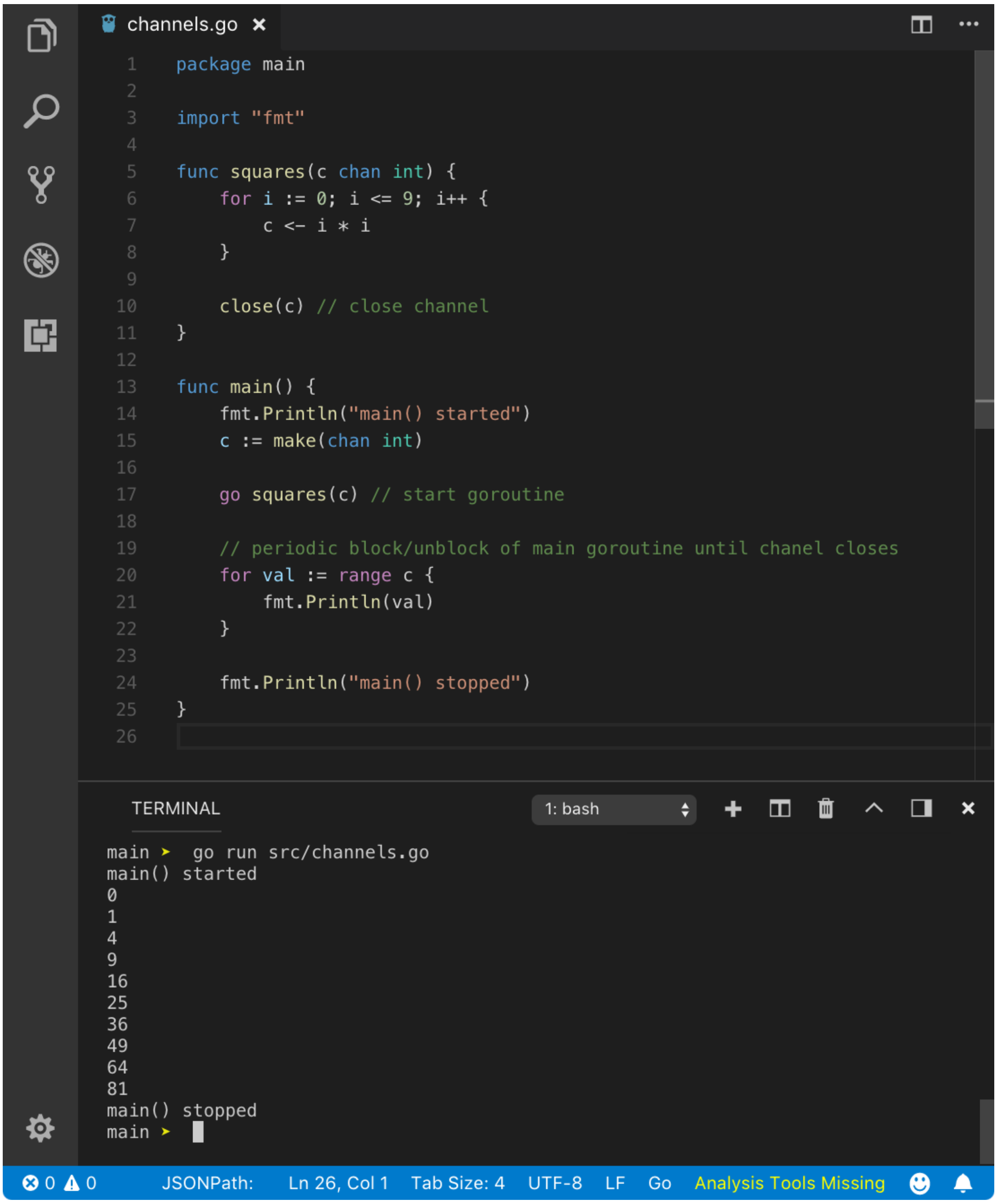
In infinite `for` loop, since we need a condition to break the loop at some point, we are reading the value from the channel with syntax `val, ok := <-c`. Here, `ok` will give us additional information when the channel is closed. Hence, in `squares` goroutine, after done writing all data, we close the channel using the syntax `close(c)`. When `ok` is `true`, program prints value in `val` and channel status `ok`. When it is `false`, we break out of the loop using `break` keyword. Hence, the above program yields the following result.

```
main() started
0 true
1 true
4 true
9 true
16 true
25 true
36 true
49 true
64 true
81 true
0 false <-- loop broke!
main() stopped
```

> *When channel is closed, value read by the goroutine is zero value of the data type of the channel. In this case, since channel is transporting `int` data type, it will be `0` as we can see from the result.*

To avoid the pain of manually checking for channel closed condition, Go gives easier `for range` loop which will automatically close when the channel is closed. Let's modify our previous above program.





https://play.golang.org/p/ICCYbWO7ZvD

In the above program, we used `for val := range c` instead of `for{}`. `range` will read the value from the channel one at a time until it is closed. Hence, the above program yields below result

```
main() started
0
1
4
9
16
25
36
49
64
81
main() stopped
```

> *If you don't close the channel in `for range` loop, program will throw `deadlock` fetal error in runtime.*

☛ Buffer size or channel capacity

As we saw, every send operation on channel blocks the current goroutine. But so far we used `make` function without the second parameter. This second parameter is the capacity of a channel or the buffer size. By default, a channel buffer size is 0 also called as `unbuffered` channel. Whatever written to the channel is immediately available to read.

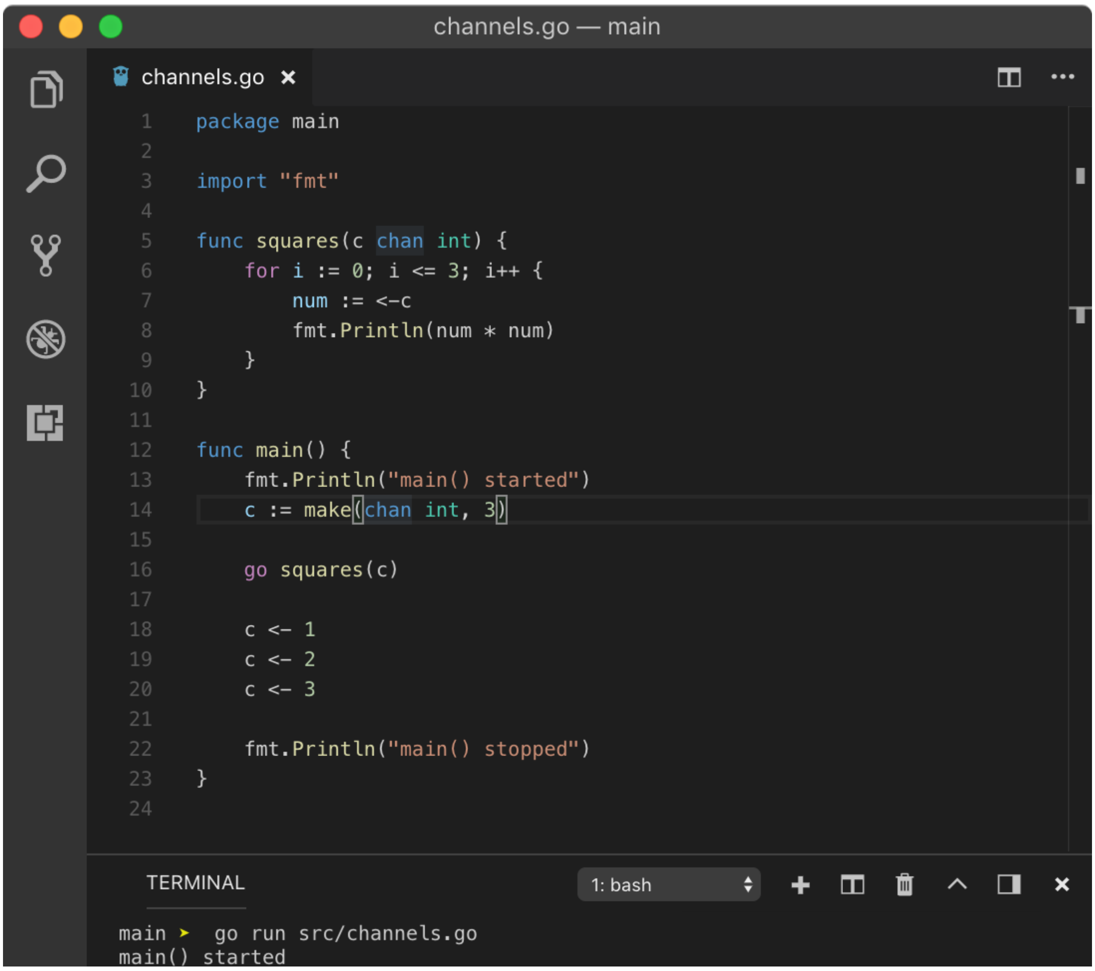
When the buffer size is non-zero `n`, **goroutine is not blocked until after buffer is full**. When the buffer is full, any value sent to the channel is added to the buffer by throwing out last value in the buffer which is available to read (*where the goroutine will be blocked*). But there is a catch, **read operation on buffered is thirsty**. That means, once read operation starts, it will continue until the buffer is empty. Technically, that means **goroutine that reads buffer channel will not block until the buffer is empty**.

We can define a buffered channel with the following syntax.

```
c := make(chan Type, n)
```

This will create a channel of a data type `Type` with buffer size `n`. Until the channel receives `n+1` send operations, it won't block the current goroutine.

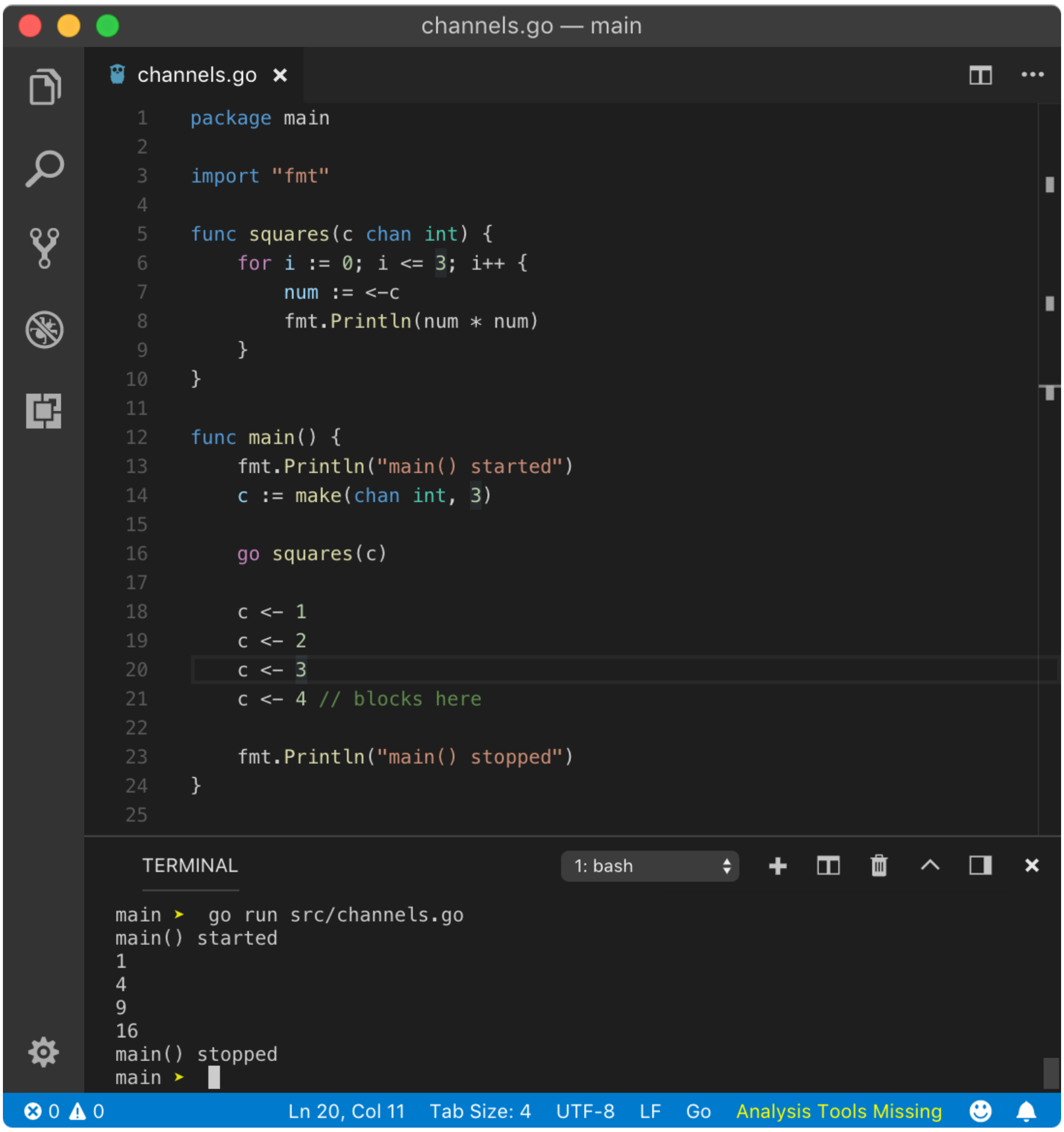
Let's prove that goroutine doesn't block until the buffer is full and overflows.



https://play.golang.org/p/k0usdYZfp3D

In the above program, channel `c` has buffer capacity of `3`. That means it can hold `3` values, which is does at line no. 20 but since the buffer is not **overflowing** (*as we didn't* ***push*** *any new value*), the main goroutine will not block and program exists.

Let's push send extra value.

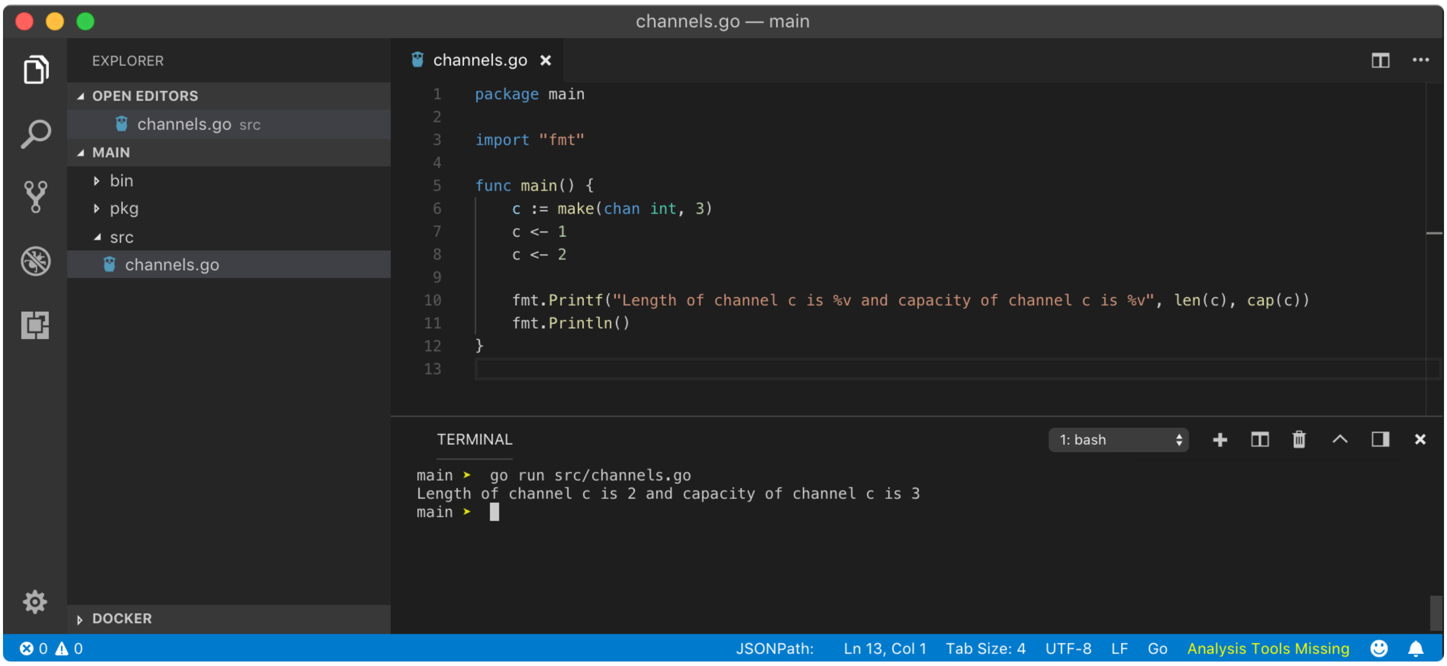


https://play.golang.org/p/KGyiskRj1Wi

As stated earlier, as now a filled buffer gets the push by `c <- 4` send operation, main goroutine blocks and `squares` goroutine **drain out** all the values.

`length` and `capacity` of a channel

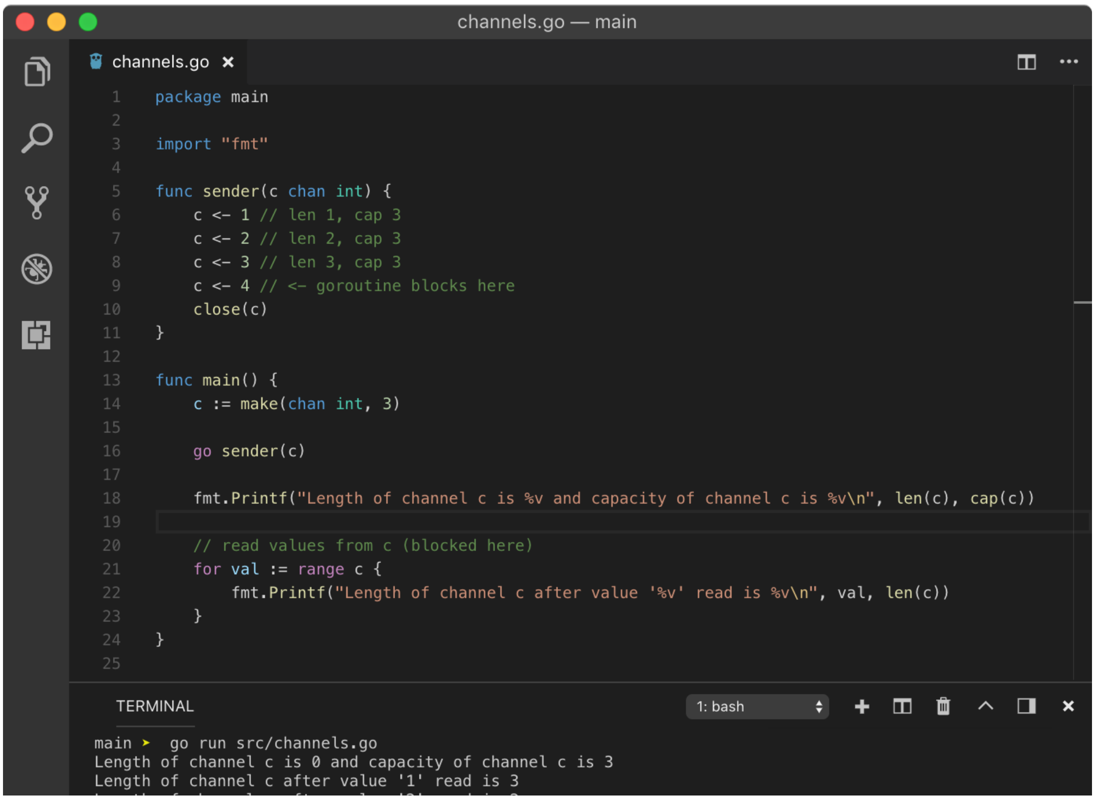
Similar to a slice, a buffered channel has length and capacity. Length of a channel is the number of values queued (*unread*) in channel buffer while the capacity of a channel is the buffer size. To calculate length, we use `len` function while to find out capacity, we use `cap` function, just like a slice.



https://play.golang.org/p/qsDZu6pXLT7

If you are wondering, why the above program runs well and deadlock error was not thrown. This is because, as channel capacity is 3 and only 2 values are available in the buffer, Go did not try to schedule another goroutine by blocking main goroutine execution. You can simply read these value in the main goroutine if you want because **even if the buffer is not full, that doesn't prevent you to read values from the channel**.

Here is another cool example



```
Length of channel c after value '2' read is 2
Length of channel c after value '3' read is 1
Length of channel c after value '4' read is 0
main ➤ █
```

https://play.golang.org/p/-gGpm08-wzz

I have a brain teaser for you.

channels.go — main

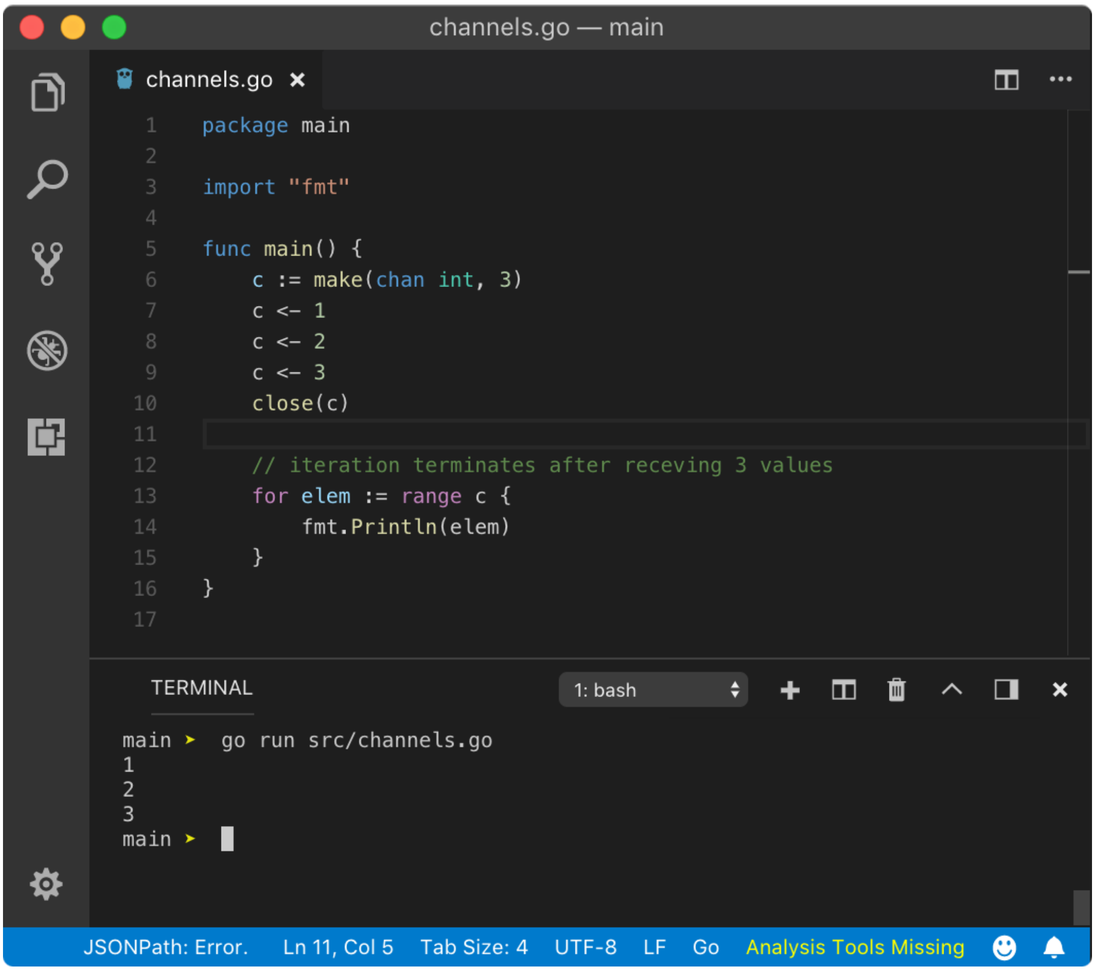
```go
package main

import (
    "fmt"
    "runtime"
)

func squares(c chan int) {
    for i := 0; i < 4; i++ {
        num := <-c
        fmt.Println(num * num)
    }
}

func main() {
    fmt.Println("main() started")
    c := make(chan int, 3)
    go squares(c)

    fmt.Println("active goroutines", runtime.NumGoroutine())
    c <- 1
    c <- 2
    c <- 3
    c <- 4 // blocks here

    fmt.Println("active goroutines", runtime.NumGoroutine())

    go squares(c)

    fmt.Println("active goroutines", runtime.NumGoroutine())

    c <- 5
    c <- 6
    c <- 7
    c <- 8 // blocks here

    fmt.Println("active goroutines", runtime.NumGoroutine())
    fmt.Println("main() stopped")
}
```

TERMINAL

```
main ➤ go run src/channels.go
main() started
active goroutines 2
1
4
9
16
active goroutines 1
active goroutines 2
25
36
49
64
```

https://play.golang.org/p/sdHPDx64aor

Using buffered channel and `for range`, we can read from closed channels. **Since for closed channels, data lives in the buffer, we can still extract that data.**



https://play.golang.org/p/vULFyWnpUoj

> *Buffered channels are like Pythagoras Cup, watch this interesting video on Pythagoras Cup.*

Working with multiple goroutines

Let's write 2 goroutines, one for calculating the **square** of integers and other for the **cube** of integers.

```go
        c <- num * num * num
    }

func main() {
    fmt.Println("[main] main() started")

    squareChan := make(chan int)
    cubeChan := make(chan int)

    go square(squareChan)
    go cube(cubeChan)

    testNum := 3
    fmt.Println("[main] sent testNum to squareChan")

    squareChan <- testNum

    fmt.Println("[main] resuming")
    fmt.Println("[main] sent testNum to cubeChan")

    cubeChan <- testNum

    fmt.Println("[main] resuming")
    fmt.Println("[main] reading from channels")

    squareVal, cubeVal := <-squareChan, <-cubeChan
    sum := squareVal + cubeVal

    fmt.Println("[main] sum of square and cube of", testNum, " is", sum)
    fmt.Println("[main] main() stopped")
}
```

TERMINAL

```
main ➤  go run src/channels.go
[main] main() started
[main] sent testNum to squareChan
[cube] reading
[square] reading
[main] resuming
[main] sent testNum to cubeChan
[main] resuming
[main] reading from channels
[main] sum of square and cube of 3  is 36
[main] main() stopped
main ➤
```

https://play.golang.org/p/6wdhWYpRfrX

Let's talk about the execution of the above program step by step.

- We created 2 functions `square` and `cube` which we will run as goroutines. Both receive the channel of type `int` as an argument in argument `c` and we read data from it in variable `num`. Then we write data to the channel `c` in the next line.
- In the main goroutine, we create 2 channels `squareChan` and `cubeChan` of type `int` using `make` function.
- Then we run `square` and `cube` goroutine.
- Since control is still inside the main goroutine, `testNumb` variable gets the value of `3`.
- Then we push data to `squareChan` and `cubeChan`. The main goroutine will be blocked until these channels read it from. Once they read it, the main goroutine will continue execution.
- When in the main goroutine, we try to read data from given channels, control will be blocked until these channels write some data from their goroutines. Here, we have

used shorthand syntax `:=` to receive data from multiple channels.

- Once these goroutines write some data to the channel, the main goroutine will be blocked.
- When the channel write operation is done, the main goroutine starts executing. Then we calculate the sum and print it on the console.

Hence the above program will yield the following result

```
[main] main() started
[main] sent testNum to squareChan
[square] reading
[main] resuming
[main] sent testNum to cubeChan
[cube] reading
[main] resuming
[main] reading from channels
[main] sum of square and cube of 3  is 36
[main] main() stopped
```

☛ Unidirectional channels

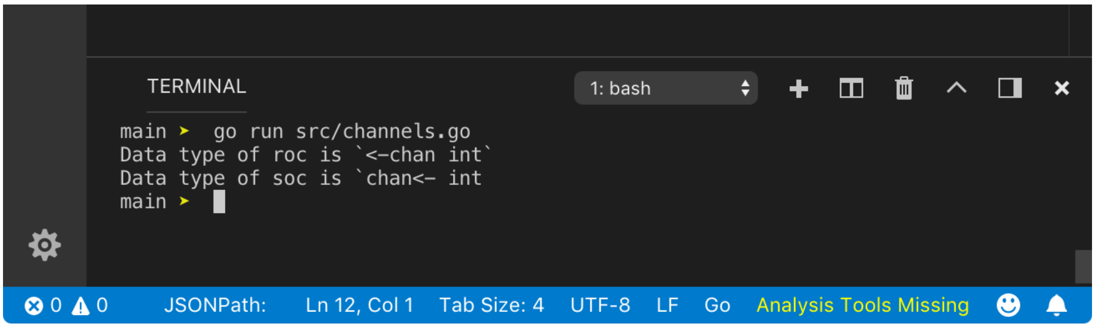
So far, we have seen channels which can transmit data from both sides or in simple words, channels on which we can do **read** and **write** operations. But we can also create channels which are **unidirectional** in nature. For example, **receive-only** channels which allow only **read operation** on them and **send-only** channels which allow only to **write operation** on them.

The unidirectional channel is also created using `make` function but with additional arrow syntax.

```
roc := make(<-chan int)
soc := make(chan<- int)
```

In the above program, `roc` is **receive-only channel** as arrow direction in `make` function points away from `chan` keyword. While `soc` is **send-only channel** where arrow direction in `make` function points towards `chan` keyword. They also have a different type.


```go
package main

import "fmt"

func main() {
	roc := make(<-chan int)
	soc := make(chan<- int)

	fmt.Printf("Data type of roc is `%T`\n", roc)
	fmt.Printf("Data type of soc is `%T\n", soc)
}
```

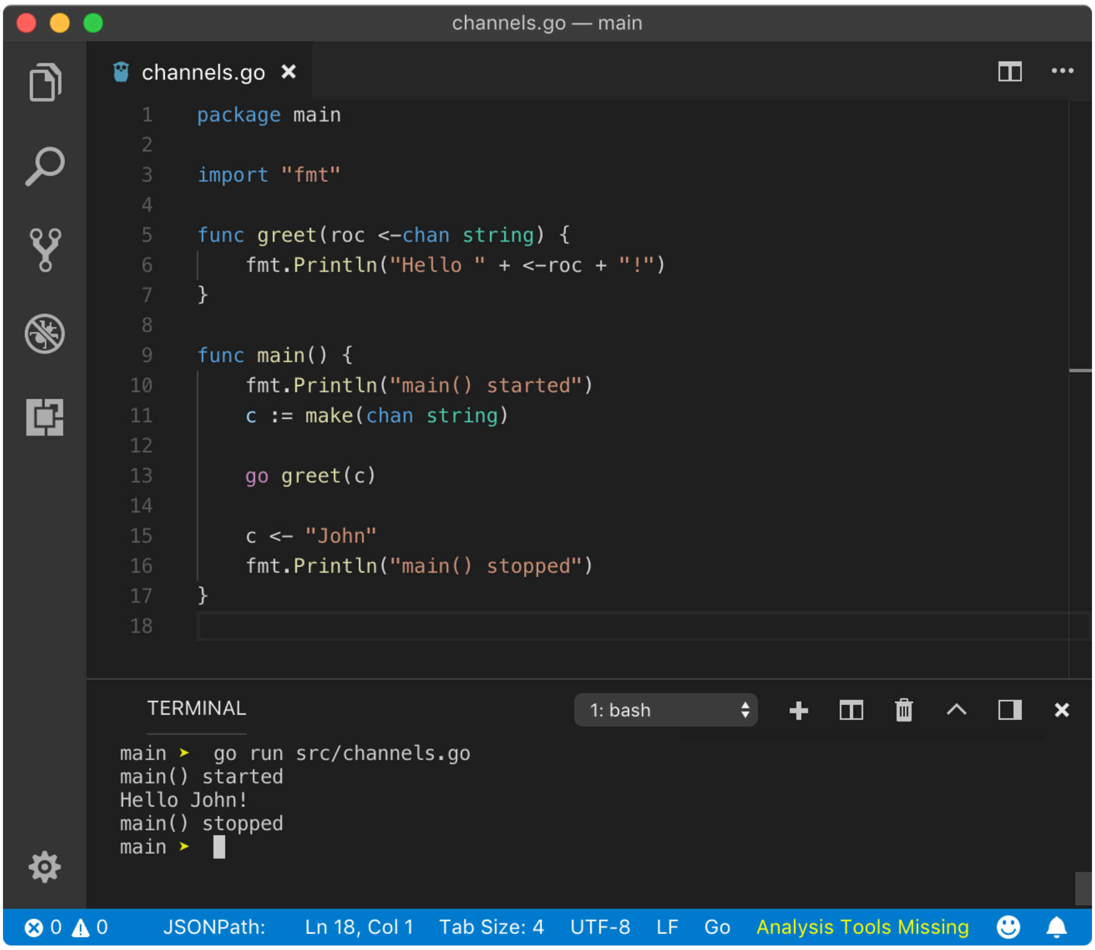


https://play.golang.org/p/JZO51IoaMg8

But what is the use of unidirectional channel? Using unidirectional channels **increases the type-safety** of a program. Hence the program is less prone to error.

But let's say that you have a goroutine where you need to **only read** data from a channel but `main` goroutine needs to **read and write** data from/to the same channel. How that will work?

Luckily, Go provide easier syntax to **convert bi-directional channel to unidirectional channel**.



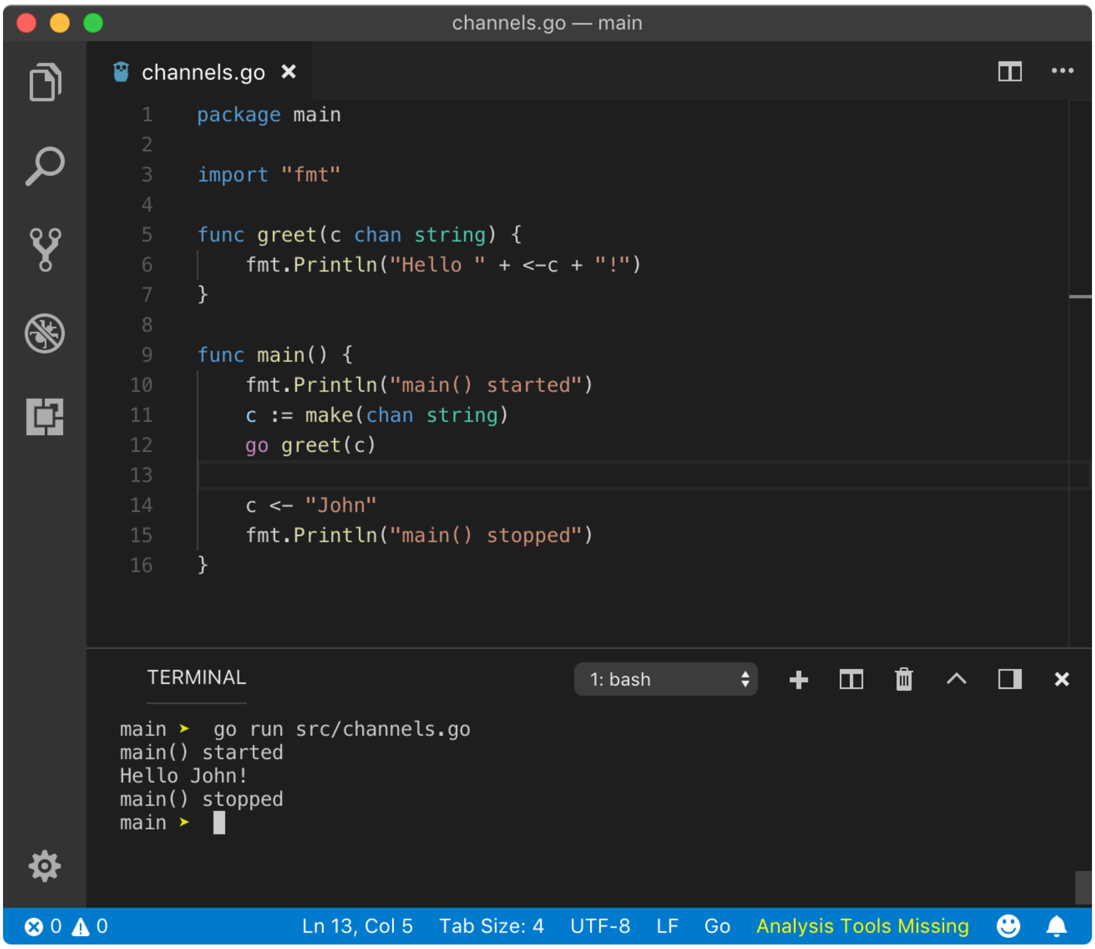
https://play.golang.org/p/k3B3gCelrGv

We modified `greet` goroutine example to convert bi-directional channel `c` to receive-only channel `roc` in `greet` function. Now we can only read from that channel. Any write operation on it will result in a fetal error `"invalid operation: roc <- "some text" (send to receive-only type <-chan string)"`.

☛ Anonymous goroutine

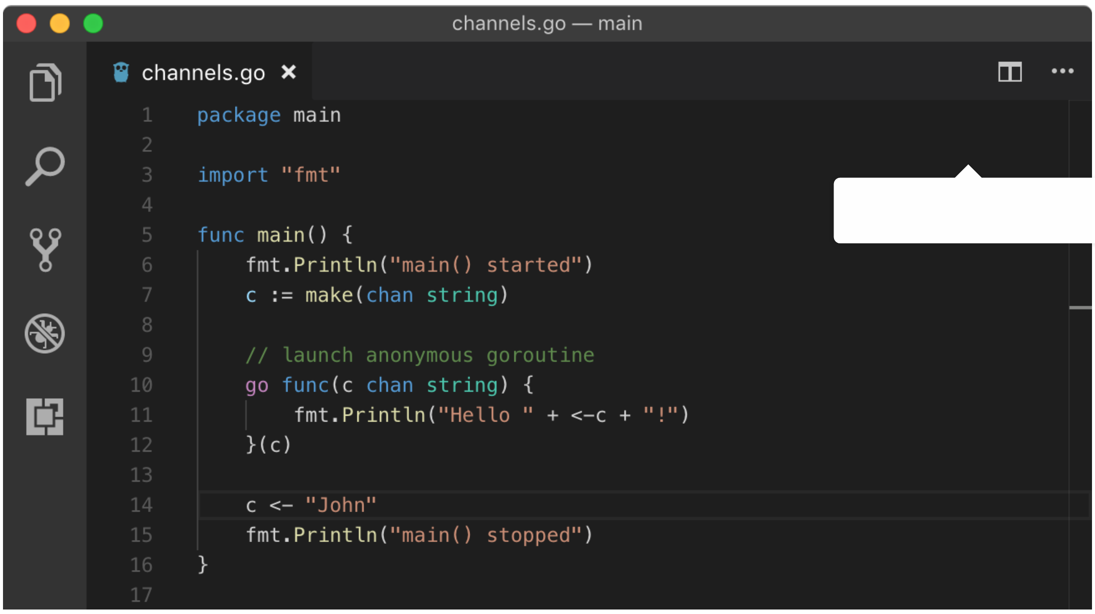
In `goroutines` chapter, we learned about ***anonymous goroutines***. We can also implement channels with them. Let's modify the previous simple example to implement channel in an anonymous goroutine.

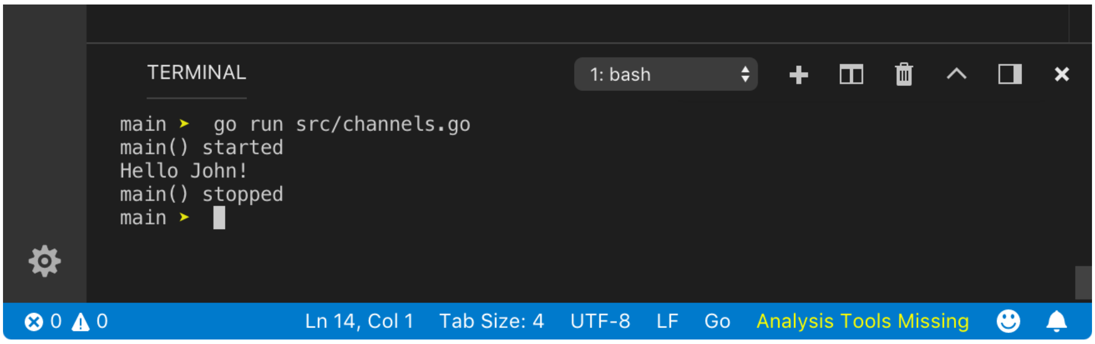
This was our earlier example



https://play.golang.org/p/c5erdHX1gwR

Below one is the modified example where we made `greet` goroutine an anonymous goroutine.

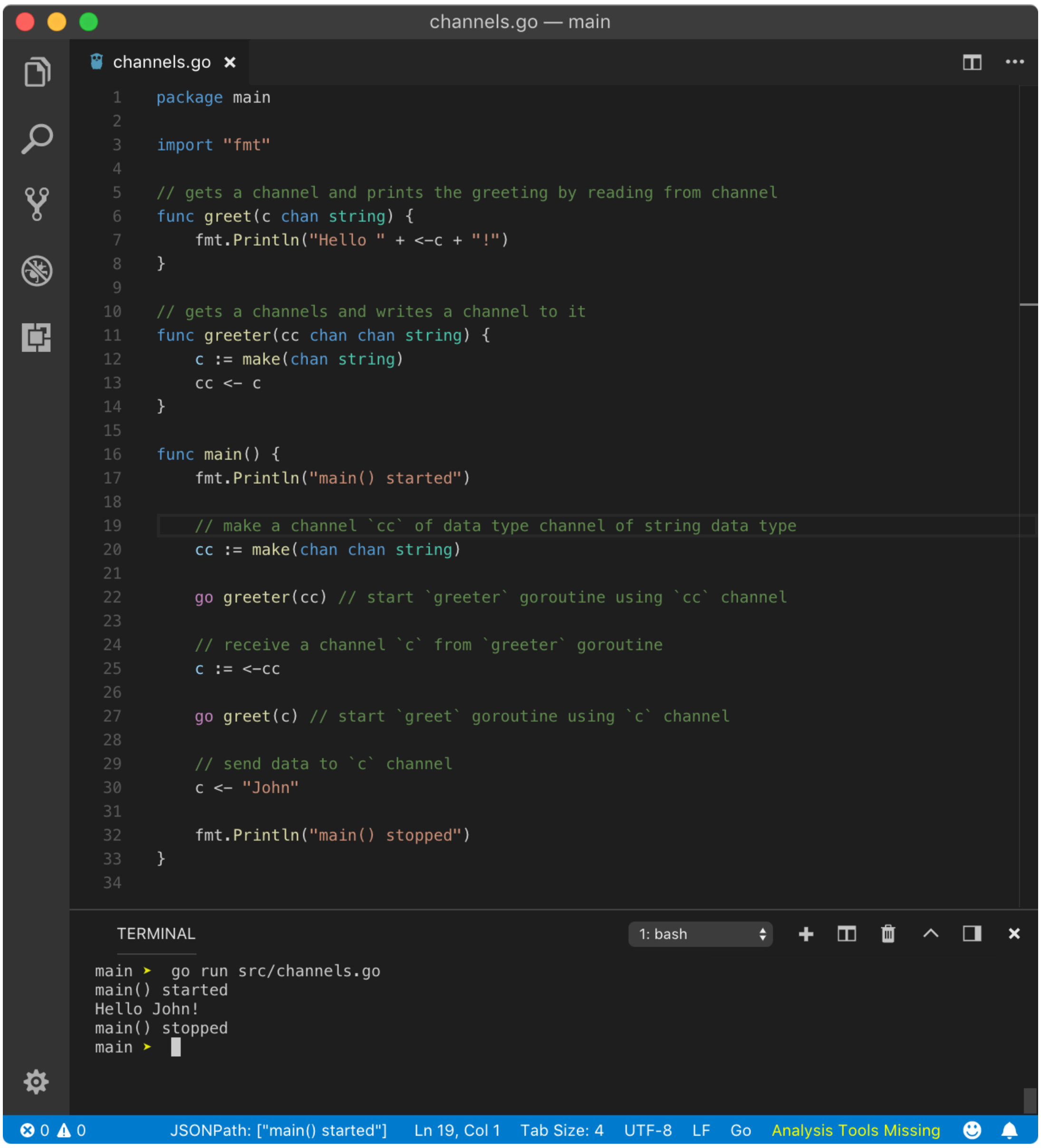




https://play.golang.org/p/cM5nFgRha7c

☛ channel as the data type of channel

Yes, channels are first-class values and can be used anywhere like other values: as struct elements, function arguments, returning values and even like a type for another channel. Here, we are interested in using a channel as the data type of another channel.



https://play.golang.org/p/xVQvvb8O4De

☛ Select

`select` is just like `switch` without any input argument but it only used for channel operations. The **`select`** statement is used to perform an operation on only one channel out of many, conditionally selected by `case` block.

Let's first see an example, then discuss how it works.

```go
package main

import (
    "fmt"
    "time"
)

var start time.Time
func init() {
    start = time.Now()
}

func service1(c chan string) {
    time.Sleep(3 * time.Second)
    c <- "Hello from service 1"
}

func service2(c chan string) {
    time.Sleep(5 * time.Second)
    c <- "Hello from service 2"
}

func main() {
    fmt.Println("main() started", time.Since(start))

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

    go service1(chan1)
    go service2(chan2)

    select {
    case res := <-chan1:
        fmt.Println("Response from service 1", res, time.Since(start))
    case res := <-chan2:
        fmt.Println("Response from service 2", res, time.Since(start))
    }

    fmt.Println("main() stopped", time.Since(start))
}
```

```
main ➤ go run src/channels.go
main() started 612ns
Response from service 1 Hello from service 1 3.000156611s
main() stopped 3.00030791s
main ➤
```

https://play.golang.org/p/ar5dZUQ2ArH

From the above program, we can see that `select` statement is just like `switch` but instead of boolean operations, we add channel operations like `read` or `write` or mixed of `read` and `write` . **The select statement is blocking except when it has a default case (*we will see that later*).** Once, one of the case conditions fulfill, it will unblock. **So when a case condition fulfills**?

If all case statements (*channel operations*) are blocking then select statement will wait until one of the case statement (*its channel operation*) unblocks and that case will be executed. If some or all of the channel operations are non-blocking, then one of the non-blocking cases will be chosen randomly and executed immediately.

To explain the above program, we started 2 goroutines with independent channels. Then we introduced `select` statement with 2 cases. One case reads a value from `chan1` and other from `chan2`. Since these channels are **unbuffered**, `read` operation will be blocking (*so the write operations*). So both the cases of select are blocking. Hence `select` will wait until one of the `case` becomes unblocking.

When control is at `select` statement, the main goroutine will block and it will schedule all goroutines present in the select statement (*one at a time*), which are `service1` and `service2`. `service1` waits for `3 second` and then unblocks by writing to the `chan1`. Similarly, `service2` waits for `5 second` and then unblocks by writing to the `chan2`. Since `service1` unblocks earlier than `service2`, case 1 will be unblocked first and hence that case will be executed and other cases (*here case 2*) will be ignored. Once done with case execution, `main` function execution will proceed further.

> *Above program simulates real world web service where a load balancer gets millions of requests and it has to return a response from one of the services available. Using goroutines, channels and select, we can ask multiple services for a response, and one which responds quickly can be used.*

To simulate when all the cases are blocking and response is available nearly at the same time, we can simply remove `Sleep` call.

```go
package main

import (
    "fmt"
    "time"
)

var start time.Time
func init() {
    start = time.Now()
}

func service1(c chan string) {
    c <- "Hello from service 1"
}

func service2(c chan string) {
    c <- "Hello from service 2"
}

func main() {
    fmt.Println("main() started", time.Since(start))

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

    go service1(chan1)
    go service2(chan2)

    select {
    case res := <-chan1:
        fmt.Println("Response from service 1", res, time.Since(start))
    case res := <-chan2:
        fmt.Println("Response from service 2", res, time.Since(start))
    }

    fmt.Println("main() stopped", time.Since(start))
```

https://play.golang.org/p/giSkkqt8XHb

The above program yields the following result (*you may get different result*)

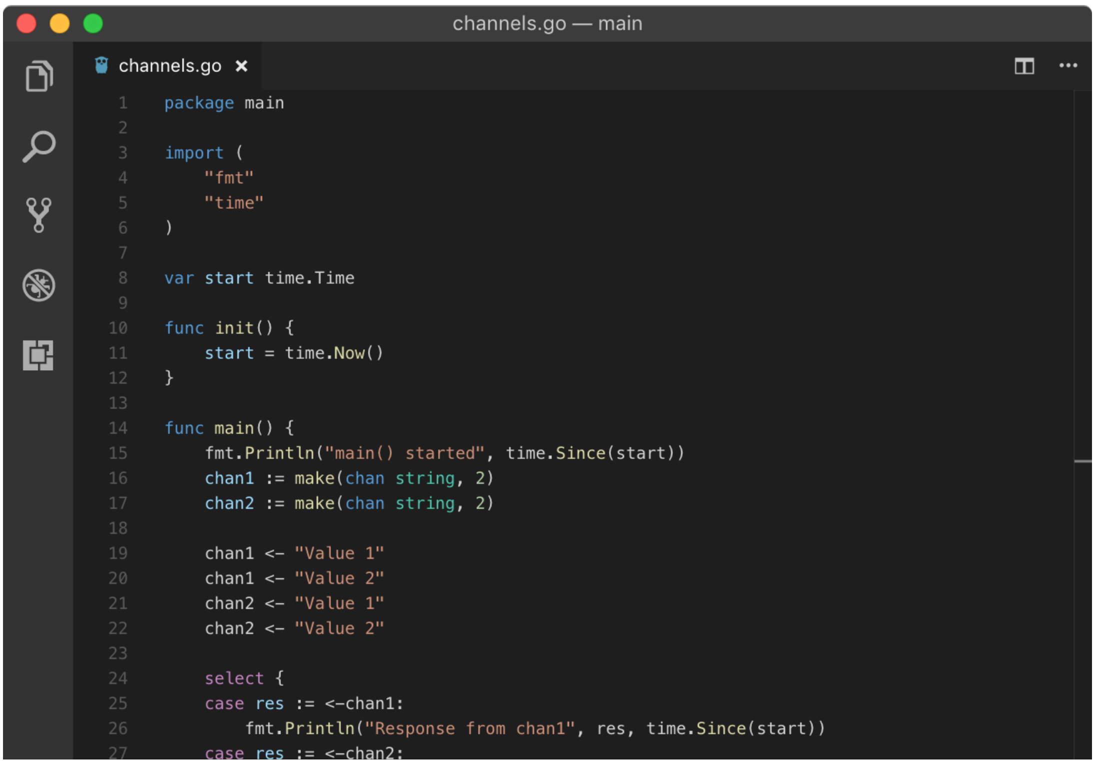
```
main() started 0s
service2() started 481µs
Response from service 2 Hello from service 2 981.1µs
main() stopped 981.1µs
```

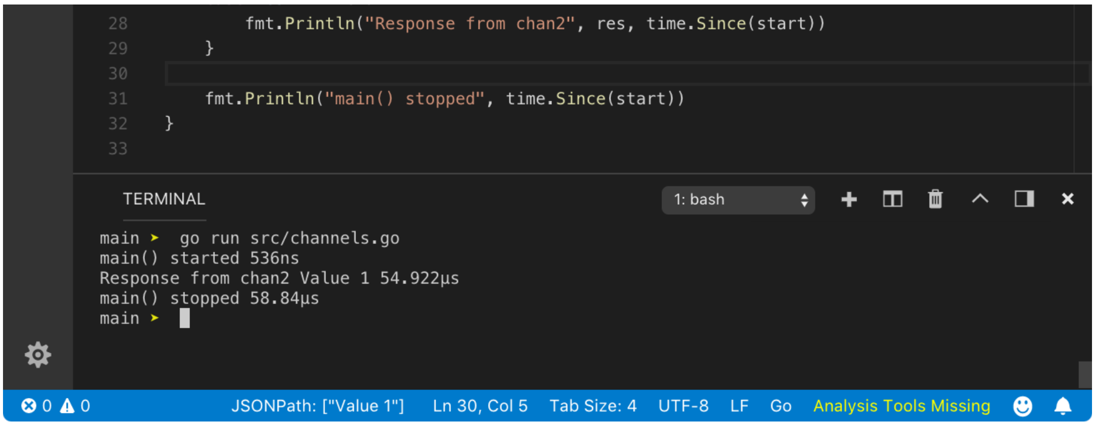
but sometimes, it can also be

```
main() started 0s
service1() started 484.8µs
Response from service 1 Hello from service 1 984µs
main() stopped 984µs
```

This happens because `chan1` and `chan2` operations happen at nearly the same time, but still, there is some time difference in execution and scheduling.

To simulate when all the cases are non-blocking and response is available at the same time, we can use a **buffered channel**.





https://play.golang.org/p/RLRGEmFQP3f

The above program yields the following result.

```
main() started 0s
Response from chan2 Value 1 0s
main() stopped 1.0012ms
```

In some cases, it can also be

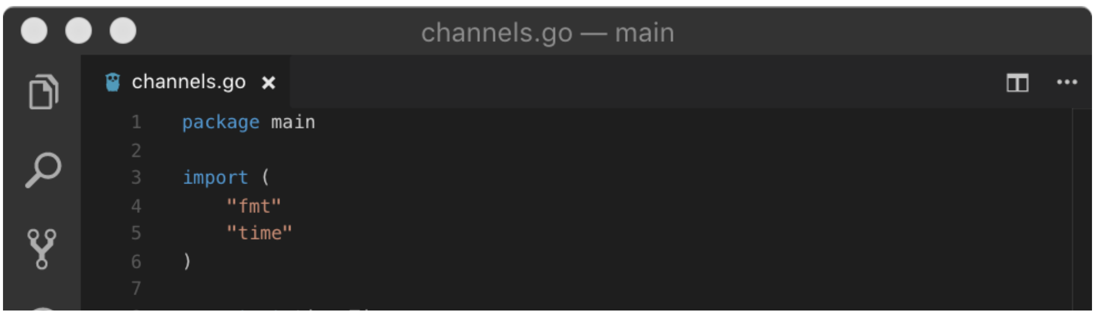
```
main() started 0s
Response from chan1 Value 1 0s
main() stopped 1.0012ms
```

In the above program, both channels have 2 values in their buffer. Since we are sending on 2 values in a channel of buffer capacity 2, these channel operations won't block and control will go to `select` statement. Since reading from the buffered channel is non-blocking operation until the entire buffer is empty and we are reading only one value in case condition, all case operations are non-blocking. Hence, Go runtime will select any `case` statement at random.

`default` case

Like `switch` statement, `select` statement also has `default` case. **A default case is non-blocking**. But that's not all, **`default` case makes `select` statement always non-blocking**. That means, `send` and `receive` operation on any channel (*buffered or unbuffered*) is always non-blocking.

If a value is available on any channel then `select` will execute that case. If not then it will immediately execute the `default` case.



```go
var start time.Time

func init() {
    start = time.Now()
}

func service1(c chan string) {
    fmt.Println("service1() started", time.Since(start))
    c <- "Hello from service 1"
}

func service2(c chan string) {
    fmt.Println("service2() started", time.Since(start))
    c <- "Hello from service 2"
}

func main() {
    fmt.Println("main() started", time.Since(start))

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

    go service1(chan1)
    go service2(chan2)

    select {
    case res := <-chan1:
        fmt.Println("Response from service 1", res, time.Since(start))
    case res := <-chan2:
        fmt.Println("Response from service 2", res, time.Since(start))
    default:
        fmt.Println("No response received", time.Since(start))
    }

    fmt.Println("main() stopped", time.Since(start))
}
```

TERMINAL

```
main ➤ go run src/channels.go
main() started 569ns
No response received 58.124µs
main() stopped 63.319µs
main ➤
```

JSONPath: ["main() started"] Ln 41, Col 1 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing

https://play.golang.org/p/rFMpc80EuT3

In the above program, since channels are unbuffered and value is not immediately available on both channel operations, `default` case will be executed. If the above `select` statement wouldn't have `default` case, `select` would have been blocking and the response would have been different.

Since with `default`, `select` is non-blocking, the scheduler does not get a call from `main` goroutine to schedule available goroutines. But we can do that manually by calling `time.Sleep`. This way, all goroutines will execute and die, returning control to `main` goroutine which will wake up after some time. When `main` goroutine wakes up, channels will have values immediately available.

channels.go — main

channels.go

```go
package main

import (
    "fmt"
```

```go
    "time"
)

var start time.Time

func init() {
    start = time.Now()
}

func service1(c chan string) {
    fmt.Println("service1() started", time.Since(start))
    c <- "Hello from service 1"
}

func service2(c chan string) {
    fmt.Println("service2() started", time.Since(start))
    c <- "Hello from service 2"
}

func main() {
    fmt.Println("main() started", time.Since(start))

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

    go service1(chan1)
    go service2(chan2)

    time.Sleep(3 * time.Second)

    select {
    case res := <-chan1:
        fmt.Println("Response from service 1", res, time.Since(start))
    case res := <-chan2:
        fmt.Println("Response from service 2", res, time.Since(start))
    default:
        fmt.Println("No response received", time.Since(start))
    }

    fmt.Println("main() stopped", time.Since(start))
}
```

TERMINAL 1: bash

```
main ➤ go run src/channels.go
main() started 603ns
service1() started 55.188µs
service2() started 92.761µs
Response from service 1 Hello from service 1 3.001191609s
main() stopped 3.001271357s
main ➤
```

JSONPath: ["main() started"] Ln 43, Col 5 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing

https://play.golang.org/p/eD0NHxHm9hN

Hence, the above program yields the following result.

```
main() started 0s
service1() started 0s
service2() started 0s
Response from service 1 Hello from service 1 3.0001805s
main() stopped 3.0001805s
```

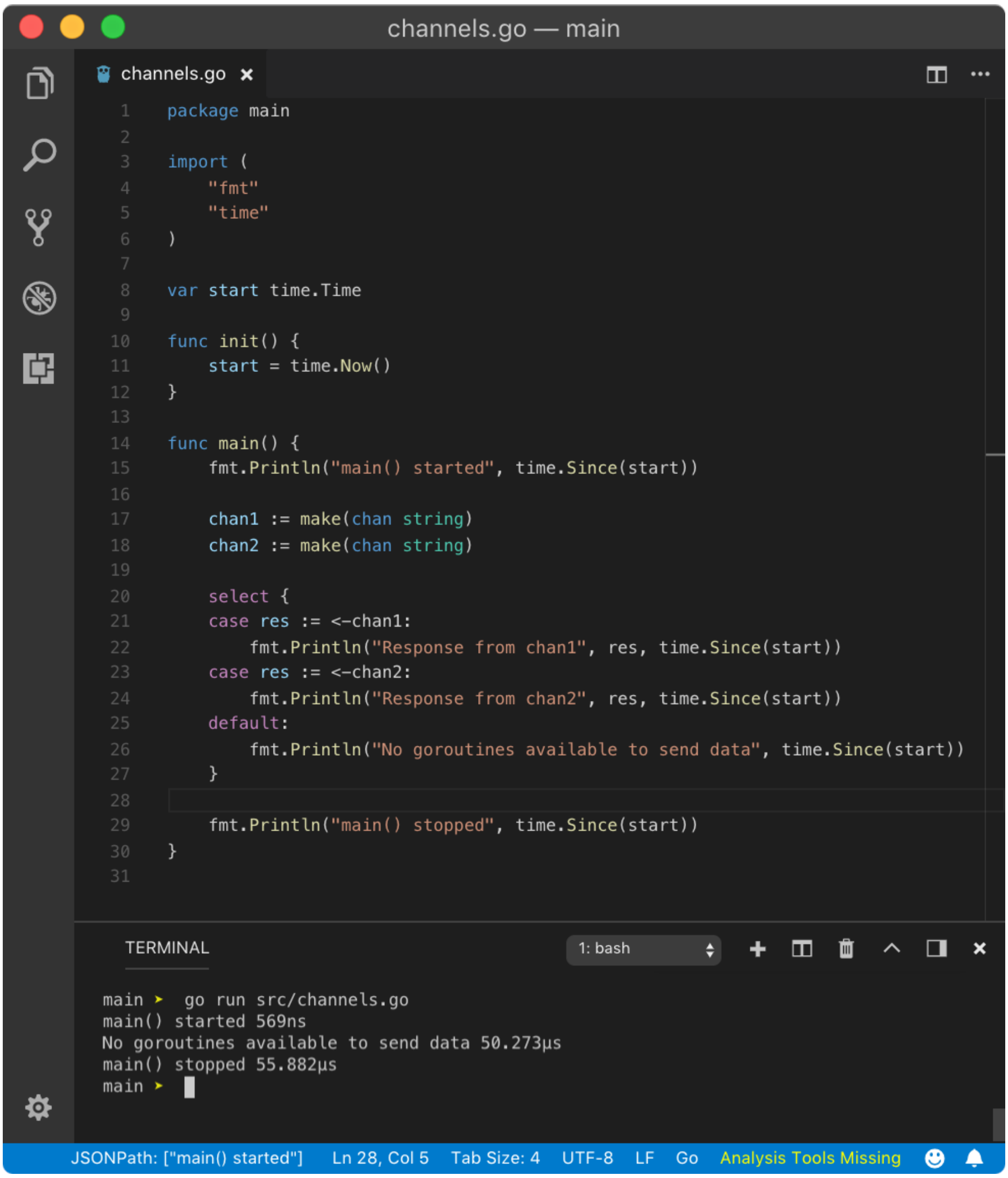
In some case, it could also be

```
main() started 0s
service1() started 0s
```

```
service2() started 0s
Response from service 2 Hello from service 2 3.0000957s
main() stopped 3.0000957s
```

Deadlock

`default` case is useful when no channels are available to send or receive data. To avoid deadlock, we can use `default` case. This is possible because all channel operations due to `default` case are non-blocking, Go does not schedule any other goroutines to send data to channels if data is not immediately available.

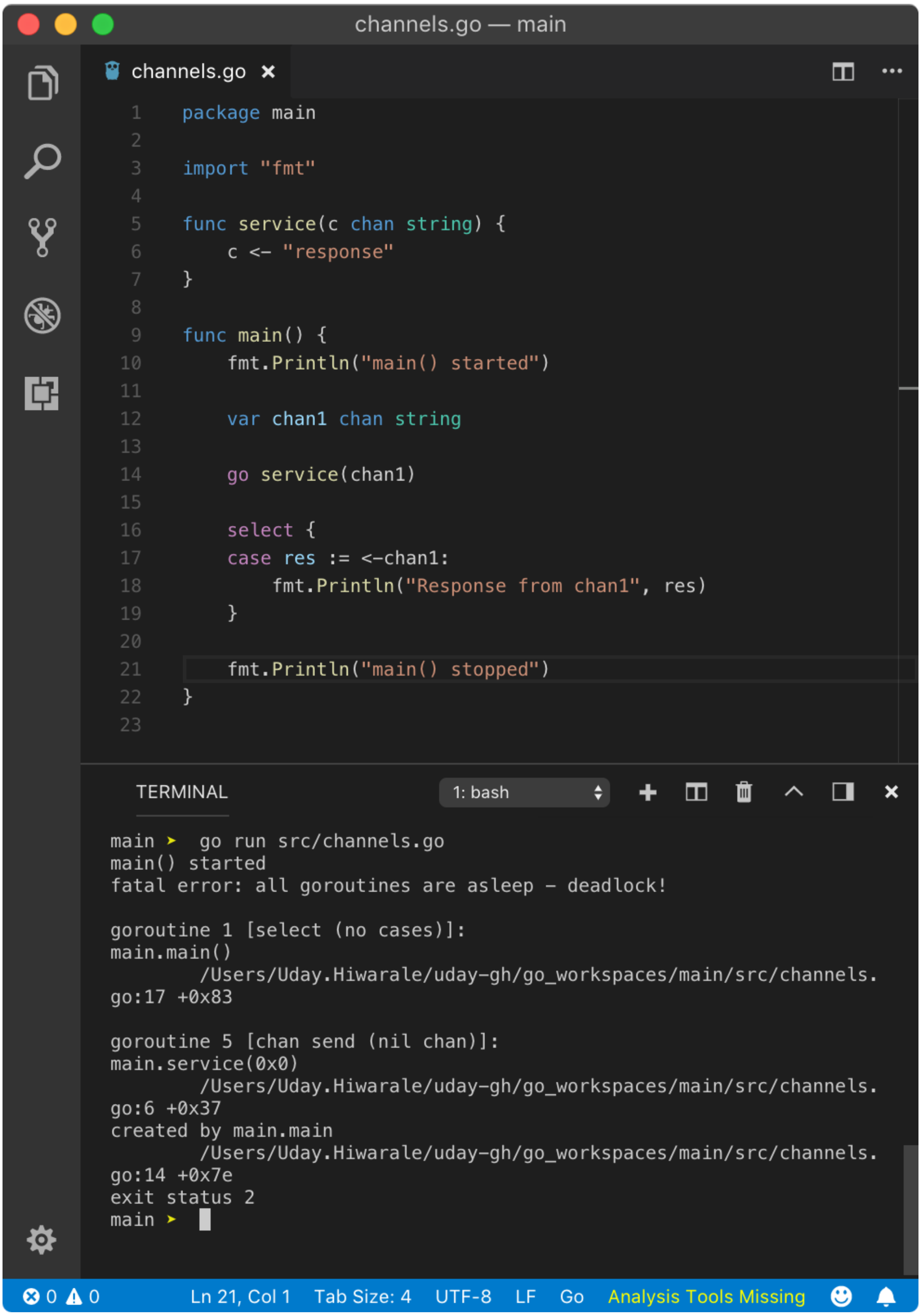


https://play.golang.org/p/S3Wxuqb8lMF

Similar to receive, in send operation, if other goroutines are sleeping (*not ready to receive value*), `default` case is executed.

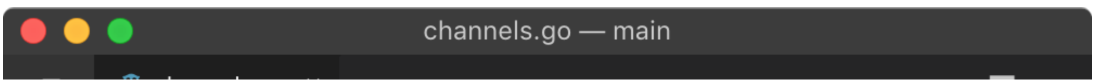
☛ `nil` channel

As we know, the default value of a channel is `nil`. Hence we can not perform `send` or `receive` operations on a nil channel. But in a case, when a `nil` channel is used in `select` statement, it will throw one of the below or both errors.



https://play.golang.org/p/uhraFubcF4S

From the above result, we can see that `select (no cases)` means that `select` statement is virtually empty because **cases with nil channel are ignored**. But as empty `select{}` statement blocks the `main` goroutine and `service` goroutine is scheduled in its place, channel operation on `nil` channels throws `chan send (nil chan)` error. To avoid this, we use `default` case.



channels.go

```go
package main

import "fmt"

func service(c chan string) {
    c <- "response"
}

func main() {
    fmt.Println("main() started")

    var chan1 chan string

    go service(chan1)

    select {
    case res := <-chan1:
        fmt.Println("Response from chan1", res)
    default:
        fmt.Println("No response")
    }

    fmt.Println("main() stopped")
}
```

TERMINAL

```
main ➤ go run src/channels.go
main() started
No response
main() stopped
main ➤
```

https://play.golang.org/p/upLsz52_CrE

Above program not-only ignores the `case` block but executes the `default` statement immediately. Hence scheduler does not get time to schedule `service` goroutine. But this is really bad design. You should always check a channel for `nil` value.

☛ Adding timeout

Above program is not very useful since only `default` case is getting executed. But sometimes, what we want is that any available services should respond in a desirable time, if it doesn't, then `default` case should get executed. This can be done using a `case` with a channel operation that unblocks after defined time. This channel operation is provided by `time` package's `After` function. Let's see an example.

channels.go — main

```go
package main

import (
    "fmt"
    "time"
)

var start time.Time
```

```go

func init() {
    start = time.Now()
}

func service1(c chan string) {
    time.Sleep(3 * time.Second)
    c <- "Hello from service 1"
}

func service2(c chan string) {
    time.Sleep(5 * time.Second)
    c <- "Hello from service 2"
}

func main() {
    fmt.Println("main() started", time.Since(start))

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

    go service1(chan1)
    go service2(chan2)

    select {
    case res := <-chan1:
        fmt.Println("Response from service 1", res, time.Since(start))
    case res := <-chan2:
        fmt.Println("Response from service 2", res, time.Since(start))
    case <-time.After(2 * time.Second):
        fmt.Println("No response received", time.Since(start))
    }

    fmt.Println("main() stopped", time.Since(start))
}

```

TERMINAL

```
main ➤  go run src/channels.go
main() started 2.564µs
No response received 2.00226428s
main() stopped 2.002396052s
main ➤
```

JSONPath: Error. Ln 33, Col 13 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing

https://play.golang.org/p/mda2t2IQK__X

Above program, yields the following result after 2 seconds.

```
main() started 0s
No response received 2.0010958s
main() stopped 2.0010958s
```

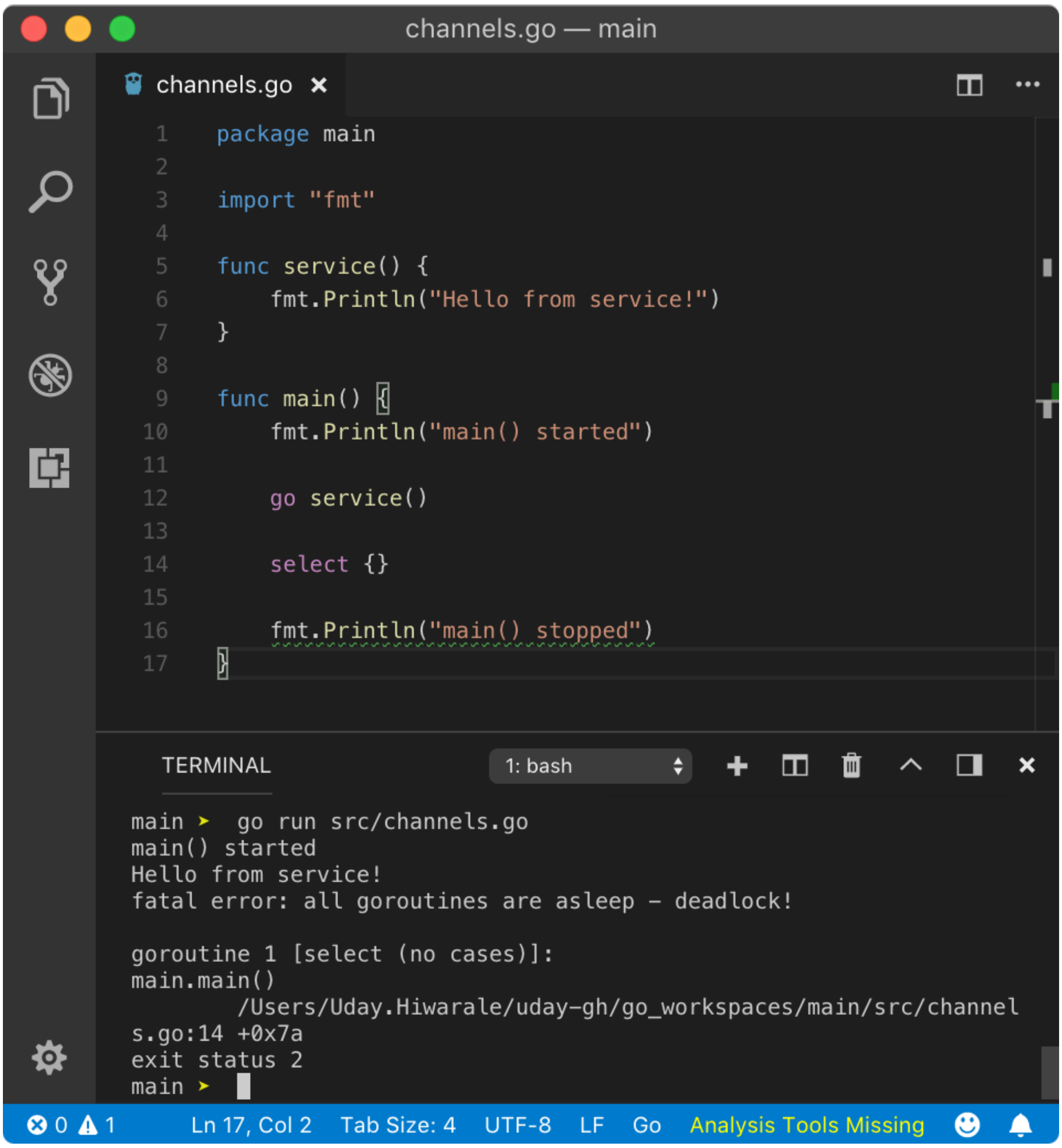
In the above program, `<-time.After(2 * time.Second)` unblocks after 2 seconds returning time at which it was unblocked, but here, we are not interested in its return value. Since it also acts like a goroutine, we have 3 goroutines out of which this one unblocks first. Hence, the case corresponding to that goroutine operation gets executed.

This is useful because you don't want to wait too long for a response from available services, where the user has to wait a long time before getting anything from the service.

If we add `10 * time.Second` in the above example, the response from `service1` will be printed, I guess that's obvious now.

☛ Empty select

Like `for{}` empty loop, an empty `select{}` syntax is also valid but there is a gotcha. As we know, `select` statement is blocked until one of the cases unblocks, and since there are no `case` statements available to unblock it, the main goroutine will block forever resulting in a deadlock.



https://play.golang.org/p/-pBd-BLMFOu

In the above program, as we know `select` will block the `main` goroutine, the scheduler will schedule another available goroutine which is `service`. But after that, it will die and the schedule has to schedule another available goroutine, but since `main` routine is blocked and no other goroutines are available, resulting in a **deadlock**.

```
main() started
Hello from service!
fatal error: all goroutines are asleep - deadlock!
```

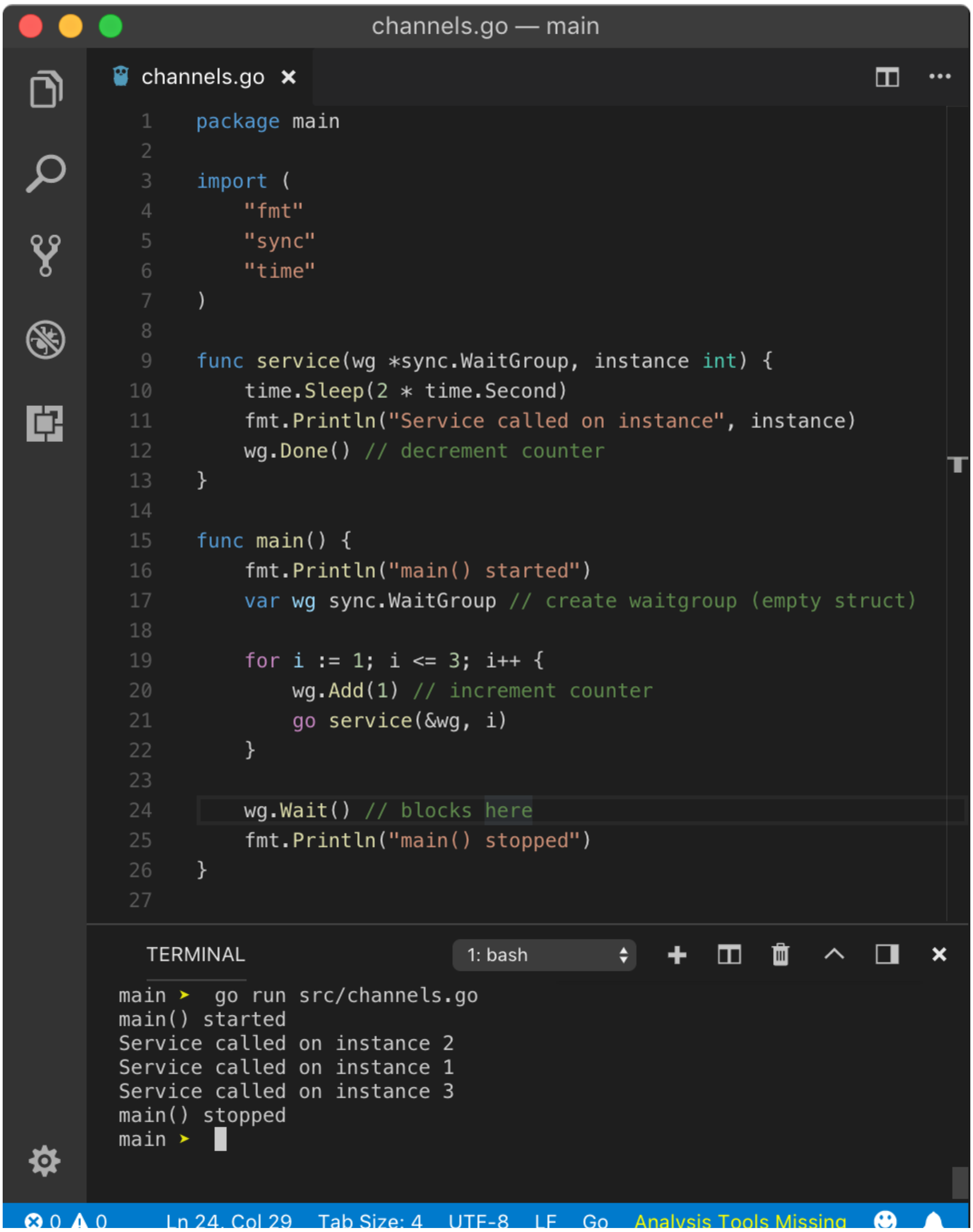
```
goroutine 1 [select (no cases)]:
main.main()
        program.Go:16 +0xba
exit status 2
```

☛ WaitGroup

Let's imagine a condition where you need to know if all goroutines finished their job. This is somewhat opposite to `select` where you needed only one condition to be `true`, but here you need **all conditions to be `true` in order to unblock the main goroutine**. Here the *condition is* successful channel operation.

WaitGroup is a `struct` with a `counter` value which tracks how many goroutines were spawned and how many have completed their job. This counter when reaches zero, means all goroutines have done their job.

Let's dive into an example and see the terminology.



https://play.golang.org/p/8qrAD9ceOfJ

In the above program, we created empty struct (*with zero-value fields*) `wg` of type `sync.WaitGroup`. WaitGroup struct has unexported fields like `noCopy`, `state1` and `sema` whose internal implementation we don't need to know. This struct has three methods viz. `Add`, `Wait` and `Done`.

`Add` method expects one `int` argument which is `delta` for the WaitGroup `counter`. The counter is nothing but an integer with default value 0. It holds how many goroutines are running. When WaitGroup is created, its counter value is 0 and we can increment it by passing `delta` as parameter using `Add` method. Remember, `counter` does not automatically understand when a goroutine was launched, hence we need to manually increment it.

`Wait` method is used to block the current goroutine from where it was called. Once `counter` reaches 0, that goroutine will unblock. Hence, we need something to decrement the counter.

`Done` method decrements the counter. It does not accept any argument, hence it only decrements the counter by 1.

In the above program, after creating `wg`, we ran `for` loop for 3 times. In each turn, we launched 1 goroutine and incremented the counter by 1. That means, now we have 3 goroutines waiting to be executed and WaitGroup counter is 3. Notice that, we passed a pointer to `wg` in goroutine. This is because in goroutine, once we are done with whatever the goroutine was supposed to do, we need to call `Done` method to decrement the counter. If `wg` was passed as a value, `wg` in `main` would not get decremented. This is pretty obvious.

After `for` loop has done executing, we still did not pass control to other goroutines. This is done by calling `Wait` method on `wg` like `wg.Wait()`. This will block the main goroutine until the counter reaches 0. Once the counter reaches 0 because from 3 goroutines, we called `Done` method on `wg` 3 times, `main` goroutine will unblock and starts executing further code.

Hence above program yields below result

```
main() started
Service called on instance 2
Service called on instance 3
Service called on instance 1
main() stopped
```

Above result might be different for you guys, as the order of execution of goroutines may vary.

> *`Add` method accept type of `int`, that means delta can also be negative. To know more about this, visit official documentation here.*

👉 Worker pool

As from the name, a `worker pool` is a collection of goroutines working concurrently to perform a job. In `WaitGroup`, we saw a collection of goroutines working concurrently but they did not have a specific job. Once you throw channels in them, they have some job to do and becomes a worker pool.

So the concept behind worker pool is maintaining a pool of worker goroutines which receives some task and returns the result. Once they all done with their job, we collect the result. All of these goroutines use the same channel for individual purpose.

Let's see a simple example with two channels viz. `tasks` and `results`.

channels.go — main

```go
package main

import (
    "fmt"
    "time"
)

// worker than make squares
func sqrWorker(tasks <-chan int, results chan<- int, instance int) {
    for num := range tasks {
        time.Sleep(time.Millisecond) // simulating blocking task
        fmt.Printf("[worker %v] Sending result by worker %v\n", instance, instance)
        results <- num * num
    }
}

func main() {
    fmt.Println("[main] main() started")

    tasks := make(chan int, 10)
    results := make(chan int, 10)

    // launching 3 worker goroutines
    for i := 0; i < 3; i++ {
        go sqrWorker(tasks, results, i)
    }

    // passing 5 tasks
    for i := 0; i < 5; i++ {
        tasks <- i * 2 // non-blocking as buffer capacity is 10
    }

    fmt.Println("[main] Wrote 5 tasks")

    // closing tasks
    close(tasks)

    // receving results from all workers
    for i := 0; i < 5; i++ {
        result := <-results // blocking because buffer is empty
        fmt.Println("[main] Result", i, ":", result)
    }

    fmt.Println("[main] main() stopped")
}
```

TERMINAL

```
main ➤ go run src/channels.go
[main] main() started
[main] Wrote 5 tasks
[worker 2] Sending result by worker 2
[worker 1] Sending result by worker 1
[main] Result 0 : 0
[main] Result 1 : 16
[worker 0] Sending result by worker 0
[main] Result 2 : 4
[worker 1] Sending result by worker 1
[main] Result 3 : 64
[worker 2] Sending result by worker 2
[main] Result 4 : 36
[main] main() stopped
```

https://play.golang.org/p/IYiMV1I4lCj

Don't worry, I am going to explain what's happening here.

- `sqrWorker` is a worker function which takes `tasks` channel, `results` channel and `id`. The job of this goroutine is to send squares of the number received from `tasks` channel to `results` channel.
- In the main function, we created `tasks` and `result` channel with buffer capacity `10`. Hence any send operation will be non-blocking until the buffer is full. Hence setting large buffer value is not a bad idea.
- Then we spawn multiple instances of `sqrWorker` as goroutines with above two channels and id parameter to get information on which worker is executing a task.
- Then we passed 5 tasks to the `tasks` channel which was non-blocking.
- Since we are done with tasks channel, we closed it. This is not necessary but it will save a lot of time in the future if some bugs get in.
- Then using for loop, with 5 iterations, we are pulling data from `results` channel. Since read operation on an empty buffer is blocking, a goroutine will be scheduled from the worker pool. Until that goroutine returns some result, the main goroutine will be blocked.
- Since we are simulating blocking operation in worker goroutine, that will call the scheduler to schedule another available goroutine until it becomes available. When worker goroutine becomes available, it writes to the `results` channel. As writing to buffered channel is non-blocking until the buffer is full, writing to `results` channel here is non-blocking. Also while current worker goroutine was unavailable, multiple other worker goroutines were executed consuming values in `tasks` buffer. After all worker goroutines consumed tasks, `for range` loop finishes when `tasks` channel buffer is empty. It won't throw deadlock error as `tasks` channel was closed.
- Sometimes, all worker goroutines can be sleeping, so `main` goroutine will wake up and works until `results` channel buffer is again empty.
- After all worker goroutines died, `main` goroutine will regain control and print remaining results from `results` channel and continue its execution.

Above example is a mouthful but brilliantly explain how multiple goroutines can feed on the same channel and get the job done elegantly. goroutines are powerful when worker's job is blocking. If you remove, `time.Sleep()` call, then only one goroutine will perform the job as no other goroutines are scheduled until `for range` loop is done and goroutine dies.

> *You can get different result like in previous example depending on how fast your system is because if all worker gorutines are blocked even for micro-second,* `main` *goroutine will wake up as explained.*

Now, let's use `WaitGroup` concept of synchronizing goroutines. Using the previous example with `WaitGroup` , we can achieve the same results but more elegantly.

channels.go — main

```go
package main

import (
	"fmt"
	"sync"
	"time"
)

// worker than make squares
func sqrWorker(wg *sync.WaitGroup, tasks <-chan int, results chan<- int, instance int) {
	for num := range tasks {
		time.Sleep(time.Millisecond)
		fmt.Printf("[worker %v] Sending result by worker %v\n", instance, instance)
		results <- num * num
	}

	// done with worker
	wg.Done()
}

func main() {
	fmt.Println("[main] main() started")

	var wg sync.WaitGroup

	tasks := make(chan int, 10)
	results := make(chan int, 10)

	// launching 3 worker goroutines
	for i := 0; i < 3; i++ {
		wg.Add(1)
		go sqrWorker(&wg, tasks, results, i)
	}

	// passing 5 tasks
	for i := 0; i < 5; i++ {
		tasks <- i * 2 // non-blocking as buffer capacity is 10
	}

	fmt.Println("[main] Wrote 5 tasks")

	// closing tasks
	close(tasks)

	// wait until all workers done their job
	wg.Wait()

	// receving results from all workers
	for i := 0; i < 5; i++ {
		result := <-results // non-blocking because buffer is non-empty
		fmt.Println("[main] Result", i, ":", result)
	}

	fmt.Println("[main] main() stopped")
}
```

TERMINAL

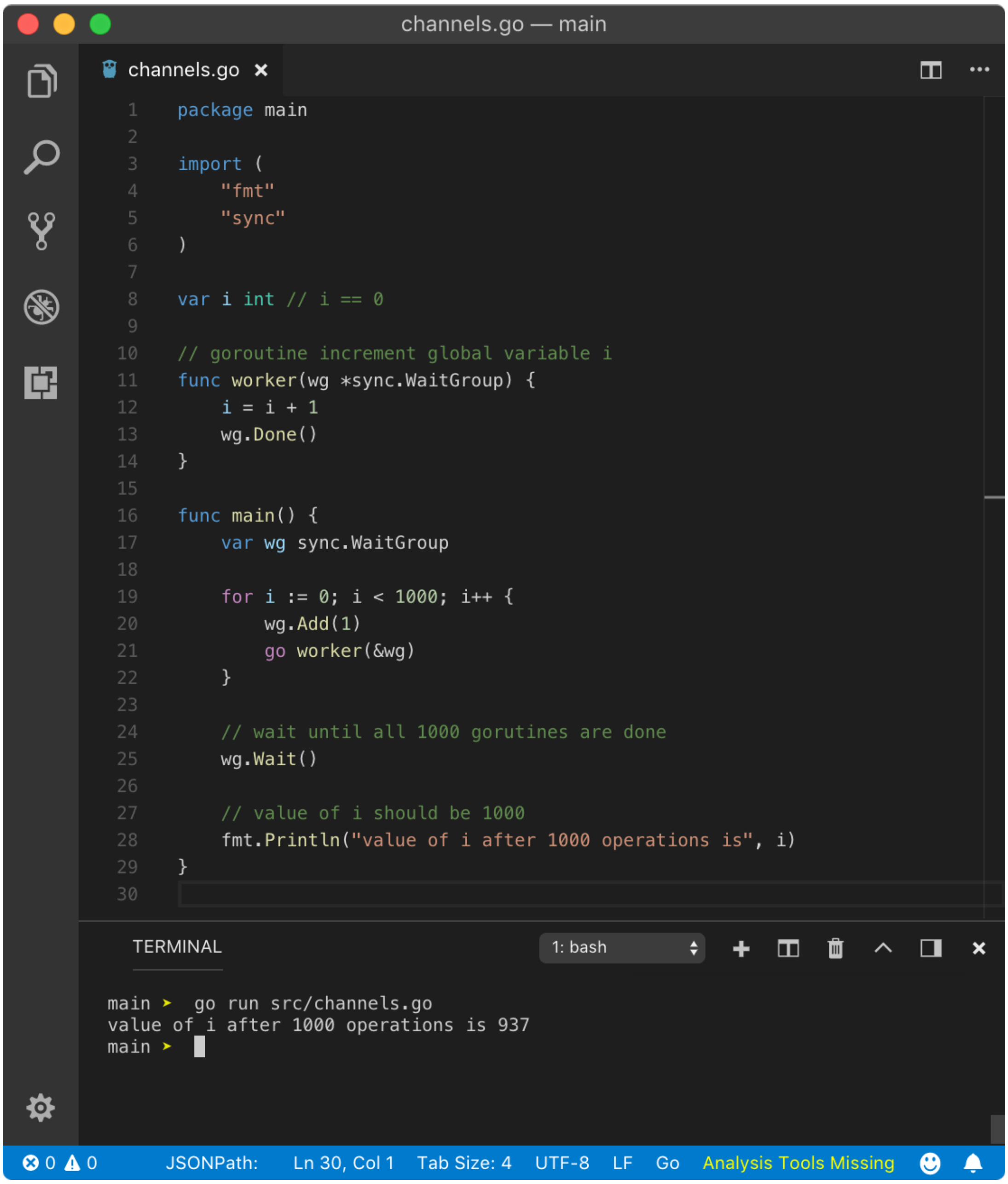
```
main ➤  go run src/channels.go
[main] main() started
[main] Wrote 5 tasks
[worker 2] Sending result by worker 2
[worker 1] Sending result by worker 1
[worker 0] Sending result by worker 0
[worker 2] Sending result by worker 2
[worker 1] Sending result by worker 1
[main] Result 0 : 4
[main] Result 1 : 16
[main] Result 2 : 0
[main] Result 3 : 36
[main] Result 4 : 64
[main] main() stopped
main ➤
```

https://play.golang.org/p/0rRfchn7sL1

Above result looks neat because read operations on `results` channel in the main goroutine is non-blocking as `result` channel is already populated by result while the main goroutine was blocked by `wg.Wait()` call. Using `waitGroup`, we can prevent lots of (*unnecessary*) context switching (*scheduling*), 7 here compared to 9 in the previous example. **But there is a sacrifice, as you have to wait until all jobs are done.**

☛ Mutex

Mutex is one of the easiest concepts in **Go**. But before I explain it, let's first understand what a race condition is. goroutines have their independent stack and hence they don't share any data between them. But there might be conditions where some data in heap is shared between multiple goroutines. In that case, multiple goroutines are trying to manipulate data at the same memory location resulting in unexpected results. I will show you one simple example.



https://play.golang.org/p/MQNepChxiEa

In the above program, we are spawning 1000 goroutines which increments the value of a global variable `i` which initially is at `0`. Since we are implementing `WaitGroup`, we want all 1000 goroutines to increment the value of `i` one by one resulting final value of `i` to be `1000`. When the main goroutine starts executing again after `wg.Wait()` call, we are printing `i`. Let's see the final result.

```
value of i after 1000 operations is 937
```

What? Why we got less than 1000? Looks like some goroutines did not work. But actually, our program had a race condition. Let's see what might have happened.

`i = i + 1` calculation has 3 steps

- (1) get value of i
- (2) increment value of i by 1
- (3) update value of i with new value

Let's imagine a scenario where different goroutines were scheduled in between these steps. For example, let's consider 2 goroutines out of the pool of 1000 goroutines viz. G1 and G2.

G1 starts first when `i` is `0`, run first 2 steps and `i` is now `1`. But before G1 updates value of `i` in step 3, new goroutine G2 is scheduled and it runs all steps. But in case of G2, value of `i` is still `0` hence after it executes step 3, `i` will be 1. Now G1 is again scheduled to finish step 3 and updates value of `i` which is `1` from step 2. In a perfect world where goroutines are scheduled after completing all 3 steps, successful operations of 2 goroutines would have produced the value of `i` to be `2` but that's not the case here. Hence, we can pretty much speculate why our program did not yield value of `i` to be `1000`.

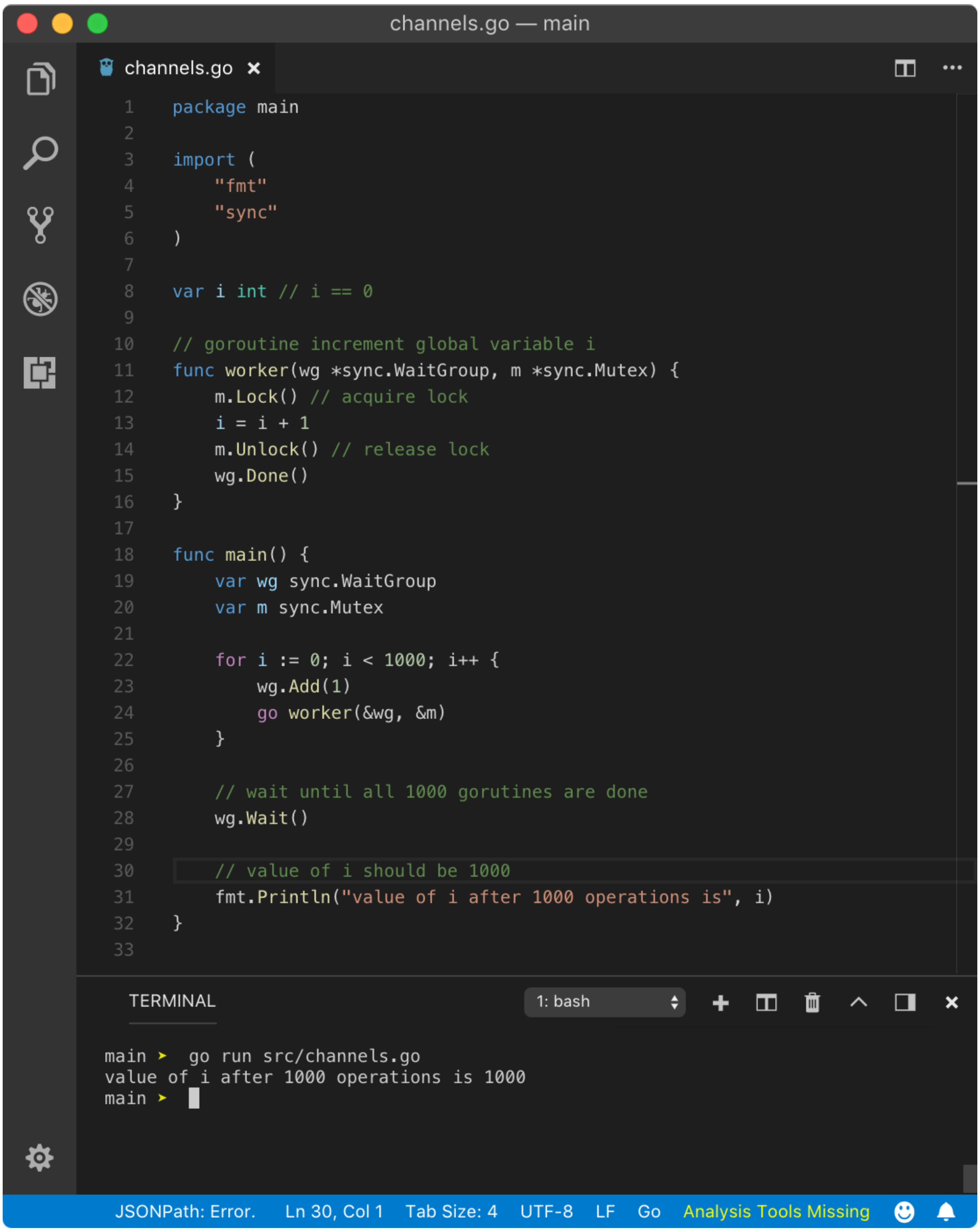
So far we learned that goroutines are cooperatively scheduled. Until unless a goroutine blocks with one of the conditions mentioned in `concurrency` lesson, another goroutine won't take its place. And since `i = i + 1` is not blocking, why Go scheduler schedules another goroutine?

You should definitely check out this answer on stackoverflow. **At any condition, you shouldn't rely on Go's scheduling algorithm and implement your own logic to synchronize different goroutines**.

One way to make sure that only one goroutine complete all 3 above steps at a time is by implementing the mutex. Mutex (*mutual exclusion*) is a concept in programming where only one routine (*thread*) can perform multiple operations at a time. This is done by one routine acquiring the lock on value, do whatever manipulation on the value it has to do and then releasing the lock. When the value is locked, no other routine can read or write to it.

In Go, the mutex data structure (a *map*) is provided by `sync` package. In Go, before performing any operation on a value which might cause race condition, we acquire a lock using `mutex.Lock()` method followed by code of operation. Once we are done with the operation, in above program `i = i + 1`, we unlock it using `mutext.Unlock()` method. When any other goroutine is trying to read or write value of `i` when the lock is present, that goroutine will block until the operation is unlocked from the first goroutine. Hence only 1 goroutine can get to read or write value of `i`, avoiding race conditions. Remember, any variables present in operation(s) between the lock and unlock will not be available for other goroutines until the whole operation(s) is unlocked.

Let's modify the previous example with a mutex.



https://play.golang.org/p/xVFAX_0Uig8

In the above program, we have created one mutex `m` and passed a pointer to it to all spawned goroutines. Before we begin operation on `i`, we acquired the lock on mutex `m` using `m.Lock()` syntax and after operation, we unlocked it using `m.Unlock()` syntax. Above program yields below result

```
value of i after 1000 operations is 1000
```

From the above result, we can see that mutex helped us resolve racing conditions. **But the first rule is to avoid shared resources between goroutines.**

> *You can test for race condition in Go using `race` flag while running a program like `Go run -race program.Go`. Read more about race detector here.*

. . .

Concurrency Patterns

There are tons of ways concurrency makes our day to day programming life easier. Following are few concepts and methodologies using which we can make programs faster and reliable.

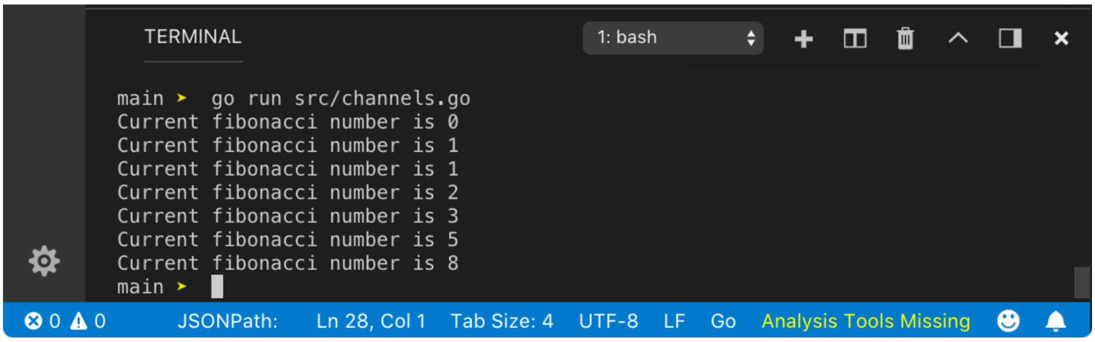
Generator

Using channels, we can implement a generator in a much better way. If a generator is computationally expensive, then we might as well do the generation of data concurrently. That way, the program doesn't have to wait until all data is generated. For example, generating a fibonacci series.

channels.go — main

channels.go

```go
package main

import "fmt"

// fib returns a channel which transports fibonacci numbers
func fib(length int) <-chan int {
    // make buffered channel
    c := make(chan int, length)

    // run generation concurrently
    go func() {
        for i, j := 0, 1; i < length; i, j = i+j, i {
            c <- i
        }
        close(c)
    }()

    // return channel
    return c
}

func main() {
    // read 10 fibonacci numbers from channel returned by `fib` function
    for fn := range fib(10) {
        fmt.Println("Current fibonacci number is", fn)
    }
}

```



https://play.golang.org/p/1_2MDeqQ3o5

Using `fib` function, we are getting a channel over which we can iterate and make use of data received from it. While inside `fib` function, since we have to return a `receive-only` channel, we are creating a buffered channel and returning it at the end. The return value of this function will convert this bi-directional channel to unidirectional receive-only channel. While using anonymous goroutine, we are pushing the fibonacci number to this channel using `for` loop. Once we are done with for loop, we are closing it from the inside of goroutine. In `main` goroutine, using `for range` on `fib` function call, we are getting direct access to this channel.

fan-in & fan-out

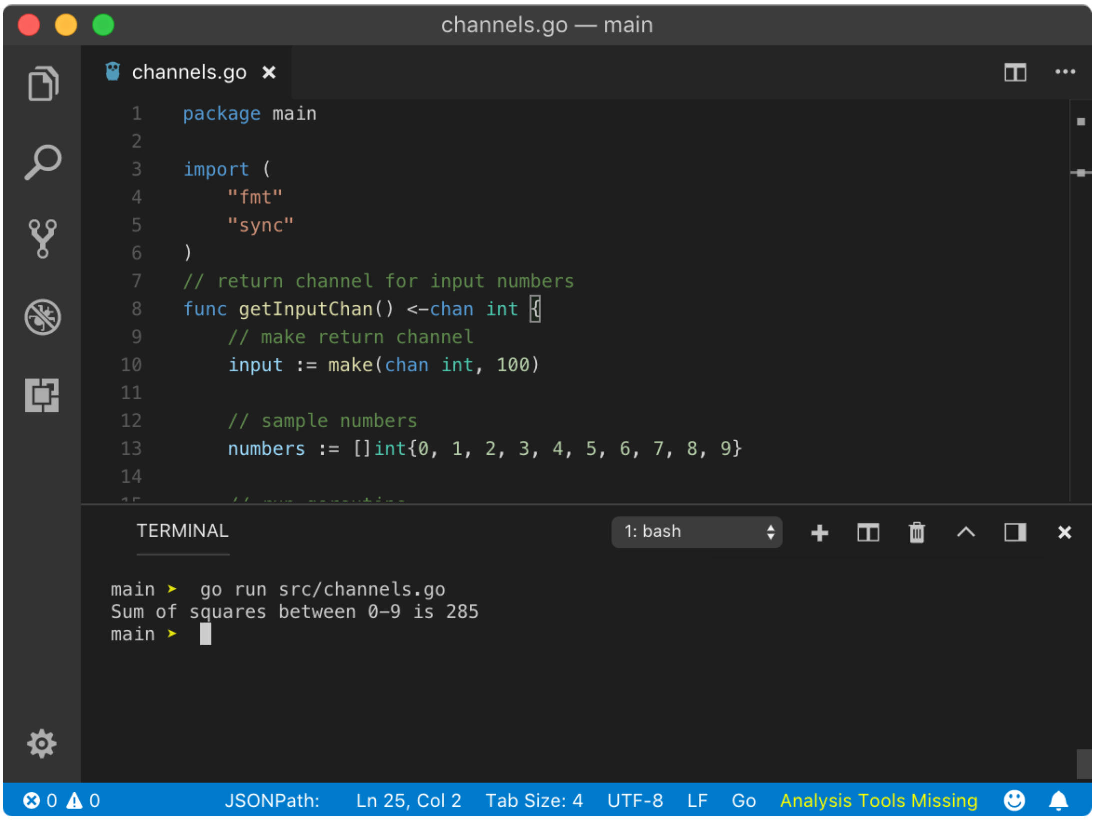
fan-in is a multiplexing strategy where the inputs of several channels are combined to produce an output channel. fan-out is demultiplexing strategy where a single channel is split into multiple channels.

```go
package main

import (
	"fmt"
	"sync"
)
// return channel for input numbers
func getInputChan() <-chan int {
	// make return channel
	input := make(chan int, 100)

	// sample numbers
	numbers := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

	// run goroutine
	go func() {
		for num := range numbers {
			input <- num
		}
		// close channel once all numbers are sent to channel
		close(input)
	}()

	return input
}

// returns a channel which returns square of numbers
func getSquareChan(input <-chan int) <-chan int {
	// make return channel
```

```
    output := make(chan int, 100)

    // run goroutine
    go func() {
        // push squares until input channel closes
        for num := range input {
            output <- num * num
        }

        // close output channel once for loop finishesh
        close(output)
    }()

    return output
}

// returns a merged channel of `outputsChan` channels
// this produce fan-in channel
// this is veriadic function
func merge(outputsChan ...<-chan int) <-chan int {
    // create a WaitGroup
    var wg sync.WaitGroup

    // make return channel
    merged := make(chan int, 100)

    // increase counter to number of channels `len(outputsChan)`
    // as we will spawn number of goroutines equal to number of channels received t
    wg.Add(len(outputsChan))

    // function that accept a channel (which sends square numbers)
    // to push numbers to merged channel
    output := func(sc <-chan int) {
        // run until channel (square numbers sender) closes
        for sqr := range sc {
            merged <- sqr
        }
        // once channel (square numbers sender) closes,
        // call `Done` on `WaitGroup` to decrement counter
        wg.Done()
    }

    // run above `output` function as groutines, `n` number of times
    // where n is equal to number of channels received as argument the function
    // here we are using `for range` loop on `outputsChan` hence no need to manuall
    for _, optChan := range outputsChan {
        go output(optChan)
    }

    // run goroutine to close merged channel once done
    go func() {
        // wait until WaitGroup finishesh
        wg.Wait()
        close(merged)
    }()

    return merged
}

func main() {
```

```go
func main() {
	// step 1: get input numbers channel
	// by calling `getInputChan` function, it runs a goroutine which sends number t
	chanInputNums := getInputChan()

	// step 2: `fan-out` square operations to multiple goroutines
	// this can be done by calling `getSquareChan` function multiple times where ir
	// `getSquareChan` function runs goroutines internally where squaring operatior
	chanOptSqr1 := getSquareChan(chanInputNums)
	chanOptSqr2 := getSquareChan(chanInputNums)

	// step 3: fan-in (combine) `chanOptSqr1` and `chanOptSqr2` output to merged ch
	// this is achieved by calling `merge` function which takes multiple channels a
	// and using `WaitGroup` and multiple goroutines to receive square number, we c
	// to `merged` channel and close it
	chanMergedSqr := merge(chanOptSqr1, chanOptSqr2)

	// step 4: let's sum all the squares from 0 to 9 which should be about `285`
	// this is done by using `for range` loop on `chanMergedSqr`
	sqrSum := 0

	// run until `chanMergedSqr` or merged channel closes
	// that happens in `merge` function when all goroutines pushing to merged chanr
	// check line no. 86 and 87
	for num := range chanMergedSqr {
		sqrSum += num
	}

	// step 5: print sum when above `for loop` is done executing which is after `ch
	fmt.Println("Sum of squares between 0-9 is", sqrSum)
}
```

fan-in-fan-out.go hosted with ❤ by GitHub view raw



https://play.golang.org/p/hATZmb6P1-u

I am not going to explain how the above program is working because I have added comments in the program explaining just that. The above program yields the following result.

```
Sum of squares between 0-9 is 285
```

. . .

(GitHub / Twitter)

Golang Golang Tutorial Go Programming Go Programming Language Concurrency