Channels

This lesson explains in detail the workings of channels in Go

About channels

Channels are pipes through which you can send and receive values using the channel operator, `<-`.

Environment Variables

Key:	Value:
GOPATH	/go

```
ch <- v    // Send v to channel ch.
v := <-ch  // Receive from ch, and
           // assign value to v.
```

(The data flows in the direction of the arrow.)

Like maps and slices, channels must be created before use:

```
ch := make(chan int)
```

By default, sends and receives block wait until the other side is ready. This allows goroutines to synchronize without explicit locks or condition variables.

Example

Environment Variables

Key:	Value:
GOPATH	/go

```go
package main

import "fmt"

func sum(a []int, c chan int) {
        sum := 0
        for _, v := range a {
                sum += v
        }
        c <- sum // send sum to c
}

func main() {
        a := []int{7, 2, 8, -9, 4, 0}

        c := make(chan int)
        go sum(a[:len(a)/2], c)
        go sum(a[len(a)/2:], c)
        x, y := <-c, <-c // receive from c

        fmt.Println(x, y, x+y)
}
```

In the example above, all lines before line **17** will execute sequentially. Line **17** will invoke a non-blocking go-routine for the `sum` function and will pass the channel `c` and the first half of the array in it. As the code does not block, the function invoked by line 17 and line 18 of the main program executes parallelly. Line **18** will take the second half of the array and the channel `c` as inputs.

Hence, both the go-routines invoked in line **17** and **18** will execute in parallel.

Line **19** will not execute until we receive a value from the channels. When the go-routines invoked by line **17** and **18** will pass values to the channels (as specified by `c <- sum`), line **19** of the main will then execute.

Buffered channels

Channels can be *buffered*. Provide the buffer length as the second argument to `make` to initialize a buffered channel:

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

Sends to a buffered channel block only when the buffer is full. Receives block when the buffer is empty.

```
package main

import "fmt"

func main() {
    c := make(chan int, 2)
    c <- 1
    c <- 2
    fmt.Println(<-c)
    fmt.Println(<-c)
}
```

But if you do:

Environment Variables

Key:	Value:
GOPATH	/go

```
package main

import "fmt"

func main() {
        c := make(chan int, 2)
        c <- 1
        c <- 2
        c <- 3
        fmt.Println(<-c)
        fmt.Println(<-c)
        fmt.Println(<-c)
}
```

You are getting a deadlock!

That’s because we overfilled the buffer without letting the code a chance to read/remove a value from the channel.

However, this version using a goroutine would work fine:

Environment Variables

Key:	Value:
GOPATH	/go

```
package main

import "fmt"

func main() {
        c := make(chan int, 2)
        c <- 1
        c <- 2
        c3 := func() { c <- 3 }
        go c3()
        fmt.Println(<-c)
        fmt.Println(<-c)
    fmt.Println(<-c)
}
```

The reason is that we are adding an extra value from inside a go routine, so our code doesn't block the main thread. The goroutine is being called before the channel is being emptied, but that is fine, the goroutine will wait until the channel is available. We then read the first value from the channel, which frees a spot and our goroutine can push its value to the channel.

Click on Go tour page to test out the example below yourself by modifying it.

```
package main

import "fmt"

func main() {
        ch := make(chan int, 2)
        ch <- 1
        ch <- 2
        fmt.Println(<-ch)
        fmt.Println(<-ch)
}
```

Having trouble visualizing channels? No worries, the next lesson explains channels through illustrations to help you better understand the concept.