Channels

This lesson will introduce you to channels - one of the most vital concepts when it comes to communication in concurrency using Go.

WE'LL COVER THE FOLLOWING

> A channel is a pipe between goroutines to synchronize execution and communicate by sending/receiving data.

Channels are based on the idea of Communicating Sequential Processes (CSP) put forward by Hoare in 1978.

First of all, let's just cover the basic syntax.

Creating a Channel

You can create a channel with the following syntax:

```
channelName := make(chan datatype)
```

Initializing A Channel

The `datatype` is the type of data that you will pass on your channel. For example, to create a channel named `result` of type `int`, you can write:

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

Sending on a Channel

You can send data over your channel by using the following syntax:

```
channelName <- data
```

Sending Data To A Channel

Let's continue with our example of `result`. If you want to send `2` on `result`, you can do so using the following code:

```
result <- 2
```

The sending of data over the channel will block the code from proceeding further until the receive operation receives the data sent on to the channel.

The same goes for receiving data on a channel. The receive operation blocks the code until and unless some data is sent by the send operation.

Receiving on a Channel

You can receive data from a channel by using the following syntax:

```
data := <- channelName
```

Receiving Data From A Channel

Hence, in order to receive and store the value `2` sent on the `result` channel, we'll do the following:

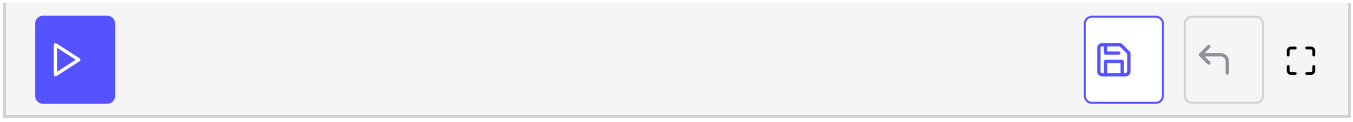
```
value := <- result
```

We can also receive by using `<-` as a unary operator as shown below:

```
<- result
```

Let's go over an example of using channels in Go:

```
package main
import "fmt"
func sendValues(myIntChannel chan int){

  for i:=0; i<5; i++ {
    myIntChannel <- i //sending value
  }

}

func main() {
  myIntChannel := make(chan int)

  go sendValues(myIntChannel)

  for i:=0; i<5; i++ {
    fmt.Println(<-myIntChannel) //receiving value
  }
}
```



Let's get into what happened above. In a goroutine created on **line 14**, the function `sendValues` was sending values over `myIntChannel` by using a for-loop. On the other hand, on **line 17**, `myIntChannel` was receiving values and the program was printing them onto the console. The most important point to note is that both the following statements were blocking operations:

- `myIntChannel <- i`
- `<-myIntChannel`

Hence, the program when blocked on `myIntChannel <- i` was unblocked by the `<-myIntChannel` statement. This was only possible as they were running concurrently.

Let's get more clarity by modifying the above code a little bit:

```
package main
import "fmt"
func sendValues(myIntChannel chan int){

  for i:=0; i<5; i++ {
    myIntChannel <- i //sending value
  }

}
```

```
func main() {
  myIntChannel := make(chan int)

  go sendValues(myIntChannel)

  for i:=0; i<6; i++ {
    fmt.Println(<-myIntChannel) //receiving value
  }
}
```

If you run the code above, you'll get the following error:

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan receive]:
main.main()
    /usercode/main.go:17 +0xa0
exit status 2
```

Finally, we reached a deadlock. Any idea why it happened? Let's figure it out.

So I just changed the for loop condition in the main loop from `i < 5` to `i < 6`. As a result, the main routine is blocked on `<-myIntChannel` because the sending operation has sent only 5 values which were received by the 5 iterations of the loop. However, for the 6th iteration, there is no sending operation that will send value on the channel. Therefore, the program is blocked on the receiving operation resulting in a deadlock.

One way to fix this problem is by closing the channel.

Closing a Channel

You can close a channel in the following way:

```
close(channelName)
```

Closing A Channel

Closing a channel means that you can no longer communicate on it. Please

note that it only makes sense for a sender, not a receiver, to close a channel

because the receiver does not know if it has received everything or not. Now let's try closing the channel:

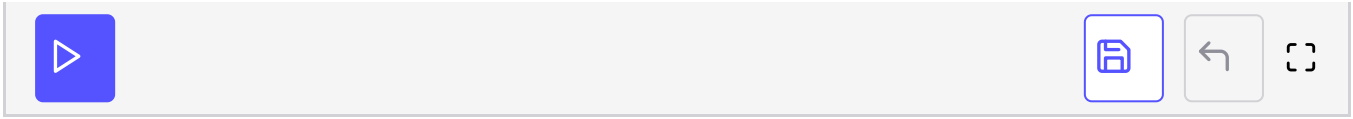
```
package main
import "fmt"
func sendValues(myIntChannel chan int){

  for i:=0; i<5; i++ {
    myIntChannel <- i //sending value
  }
  close(myIntChannel)
}

func main() {
  myIntChannel := make(chan int)

  go sendValues(myIntChannel)

  for i:=0; i<6; i++ {
    fmt.Println(<-myIntChannel) //receiving value
  }
}
```

You can see that when we close the channel after all our send operations, the receive operation returns `0` without blocking on the 6th iteration.

Additionally, the receive operation returns another value with the data to indicate whether the channel is open or not. Let's see how we can use it to solve our problem:

```
package main
import "fmt"
func sendValues(myIntChannel chan int){

  for i:=0; i<5; i++ {
    myIntChannel <- i //sending value
  }
  close(myIntChannel)
}

func main() {
  myIntChannel := make(chan int)

  go sendValues(myIntChannel)

  for i:=0; i<6; i++ {
    value, open := <-myIntChannel
```

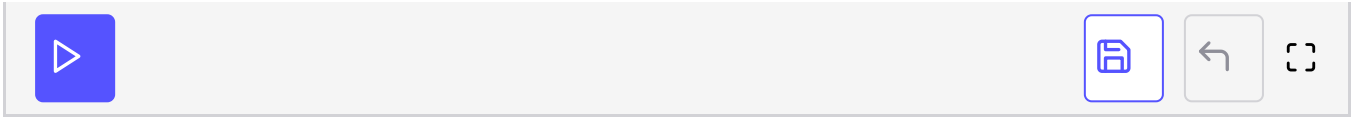
```
    if !open {
       break;
    }

    fmt.Println(value) //receiving value
  }
}
```



Here, we check if the channel is open or not using `open` (**line 18**) and break the loop if the channel is closed (**line 19**).

Another way to implement the same functionality as above is by using the `range` function on **line 16**:

```
package main
import "fmt"
func sendValues(myIntChannel chan int){

  for i:=0; i<5; i++ {
    myIntChannel <- i //sending value
  }
  close(myIntChannel)
}

func main() {
  myIntChannel := make(chan int)

  go sendValues(myIntChannel)

  for value := range myIntChannel {
    fmt.Println(value) //receiving value
  }
}
```



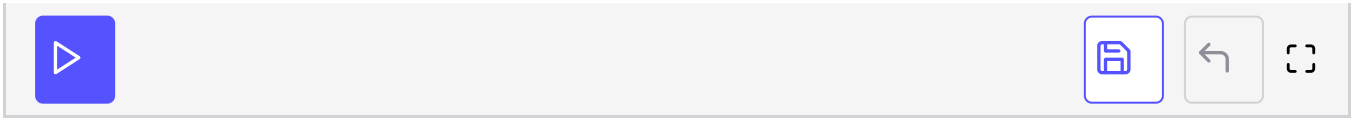
Now we got exactly the same results but we don't have to manually check if the channel is open or not. This was just syntactic sugar.

Deferring the closing of a Channel

The `defer` function defers the execution of a function until the end of the surrounding function.

Have a look at the example below:

```
package main
import "fmt"
func sendValues(myIntChannel chan int){

  for i:=0; i<5; i++ {
    myIntChannel <- i //sending value
  }

}

func main() {
  myIntChannel := make(chan int)
  defer close(myIntChannel)
  go sendValues(myIntChannel)

  for i:=0; i<5; i++ {
    fmt.Println(<-myIntChannel) //receiving value
  }
}
```



In general, it is good practice to defer the closing of channels in the main program so that we clean up everything ourselves.

Hope you loved all the information on channels. Let’s learn more about them in the next lesson.