Select Statement

This lesson will introduce you to multi-way concurrent control in Go: the select statement.

WE'LL COVER THE FOLLOWING

> The select statement blocks the code and waits for multiple channel operations simultaneously.

Syntax

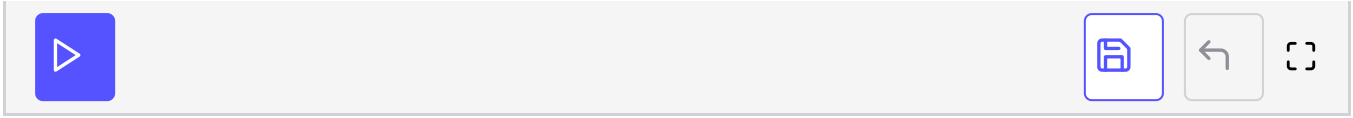
The syntax for the `select` statement is as follows:

```
select {
  case channel operation:
    statement(s);
       . //more cases
       .
       .
  default : //Optional
    statement(s);
}
```

Example

Let's try to understand the usage of a select statement with a simple example consisting of two channels which are communicating using send/receive operations:

```
package main

import (
        "fmt"
        "time"
)

func main() {
        channel1 := make(chan string)
        channel2 := make(chan string)

        go func() {
                for i := 0; i < 5; i++ {
                        channel1 <- "I'll print every 100ms"
                        time.Sleep(time.Millisecond * 100)

                }
        }()

        go func() {
                for i := 0; i < 5; i++ {
                        channel2 <- "I'll print every 1s"
                        time.Sleep(time.Second * 1)

                }
        }()

        for i := 0; i < 5; i++ {

                fmt.Println(<-channel1)
                fmt.Println(<-channel2)

        }
}
```



In the code above, we have two goroutines, each of which is sending messages on each of the two channels.

The first goroutine sends a message `I'll print every 100ms` and then waits for 100ms before sending the message again in the next iteration.

```
go func() {
    for i := 0; i < 5; i++ {
            channel1 <- "I'll print every 100ms"
            time.Sleep(time.Millisecond * 100)

        }
    }()
```

On the other hand, the second goroutine functions similarly and sends a message `I'll print every 1s` in each of the five iterations with a delay of 1s.

```
go func() {
        for i := 0; i < 5; i++ {
            channel2 <- "I'll print every 1s"
            time.Sleep(time.Second * 1)

        }
    }()
```

The interesting thing happens in this part of the code:

```
for i := 0; i < 5; i++ {

        fmt.Println(<-channel1)
        fmt.Println(<-channel2)

    }
```

`fmt.Println(<-channel1)` receives and prints the message `I'll print every 100ms` and the program moves to `fmt.Println(<-channel2)`. After receiving and printing the message from the second goroutine, we get done with our first iteration. However, in the second iteration, after `channel1` receives the message sent, our program is blocked by the second goroutine as it is waiting for 1 second of the previous iteration. Note that `channel1` is always ready to send messages but is blocked by the delay in the second goroutine. `channel1` has to wait for 1 second for `fmt.Println(<-channel2)` to execute every time although the first goroutine may have finished entirely in less than a second!

The expected output is shown below:

```
I'll print every 1s or I'll print every 100ms (depends on which goroutine is ready first)
I'll print every 100ms
I'll print every 100ms
I'll print every 100ms
I'll print every 100ms
```

But instead we have the two statements printed turn by turn because of the blocking of `channel2`. Hence, the actual output is as follows:

```
I'll print every 100ms
I'll print every 1s
```

```
I'll print every 100ms
I'll print every 1s
I'll print every 100ms

I'll print every 1s
I'll print every 100ms
I'll print every 1s
I'll print every 100ms
I'll print every 1s
```

Here, the select statement comes to the rescue:

```
package main

import (
        "fmt"
        "time"
)

func main() {
        channel1 := make(chan string)
        channel2 := make(chan string)

        go func() {
                for i := 0; i < 5; i++ {
                        channel1 <- "I'll print every 100ms"
                        time.Sleep(time.Millisecond * 100)

                }
        }()

        go func() {
                for i := 0; i < 5; i++ {
                        channel2 <- "I'll print every 1s"
                        time.Sleep(time.Second * 1)

                }
        }()

        for i := 0; i < 5; i++ {
                select {
                case message1 := <-channel1:
                        fmt.Println(message1)
                case message2 := <-channel2:
                        fmt.Println(message2)
                }
        }
}
```

The select statement will pick up any channel operation that is ready. As you can see from the output, there are more `I'll print every 100ms` statements as compared to `I'll print every 1s` because `channel1` is ready after every 100ms whereas `channel2` takes 1 second in order to get ready to send/receive

100ms whereas `channel2` takes 1 second in order to get ready to send/receive messages.

Default Case

Let's add a default case to our select statement and see what happens:

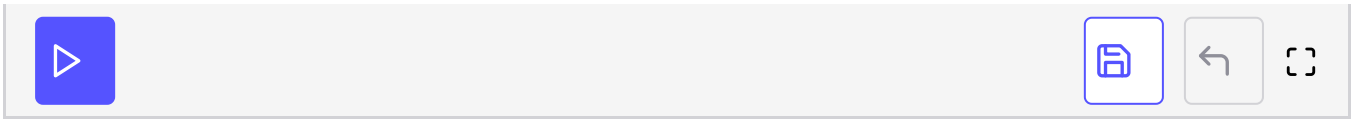
```
package main

import (
        "fmt"
        "time"
)

func main() {
        channel1 := make(chan string)
        channel2 := make(chan string)

        go func() {
                for i := 0; i < 5; i++ {
                        channel1 <- "I'll print every 100ms"
                        time.Sleep(time.Millisecond * 100)

                }
        }()

        go func() {
                for i := 0; i < 5; i++ {
                        channel2 <- "I'll print every 1s"
                        time.Sleep(time.Second * 1)

                }
        }()

        for i := 0; i < 5; i++ {
                select {
                case message1 := <-channel1:
                        fmt.Println(message1)
                case message2 := <-channel2:
                        fmt.Println(message2)
                default:
                        fmt.Println("No channel is ready")
                }
        }
}
```

The default case executes when every other send/receive operation is blocked. But what happens if we have more than one send/receive operation ready at the same time? Well, the select statement chooses one of them at random.

Nil Channel

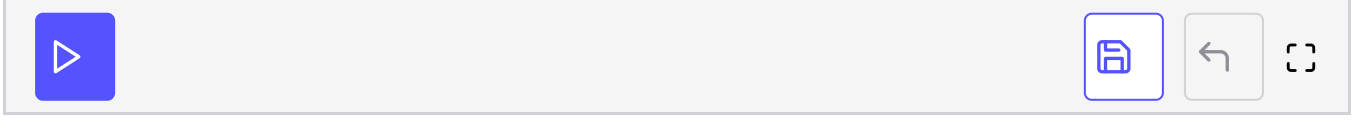
Nil channel will block the select statement forever, giving an error stating a deadlock:

```
package main

import (
        "fmt"
)

func main() {
        channel1 := make(chan string)
        channel1 = nil

        select {
        case message1 := <-channel1:
                fmt.Println(message1)
        }
}
```



If we want to disable the send/receive operations of a channel in select statement, we can use nil channels. Let's add other cases and the default case to resolve the deadlock:

```
package main

import (
        "fmt"
)

func main() {
        channel1 := make(chan string)
        channel1 = nil
        channel2 := make(chan string)

        select {
        case message1 := <-channel1:
                fmt.Println(message1)
        case message2 := <-channel2:
                fmt.Println(message2)
        default:
                fmt.Println("No channel is ready")
        }
}
```



Empty Select

Sometimes, the select statement is just there to block the code. This happens if no case is given for the select statement to proceed as it needs at least one ready case for itself to unblock.

```
select {}
```

That’s all there was to select statements. Let’s challenge ourselves to an exercise in the next lesson.