Select

This lesson explains in detail the use of the select statement in Go, it's default case and concept of timeout

WE'LL COVER THE FOLLOWING

- Select Statement
- Example
- Default case
- Timeout

Select Statement

The `select` statement lets a goroutine wait on multiple communication operations.

A `select` blocks until one of its cases can run, then it executes that case. It chooses one at random if multiple are ready.

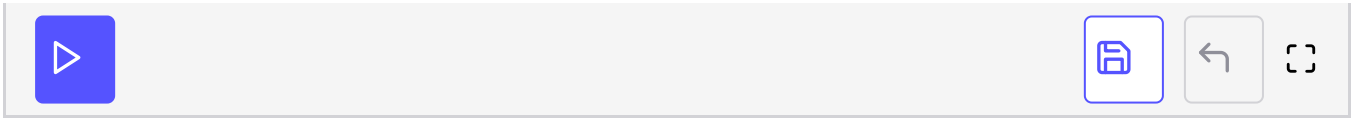
Example

Environment Variables

Key:	Value:
GOPATH	/go

```
package main

import "fmt"

func fibonacci(c, quit chan int) {
    x, y := 0, 1
    for {
        select { //waiting on a value for c
        case c <- x:
            x, y = y, x+y
        case <-quit:
            fmt.Println("quit")
            return
```

```
        }
    }
}

func main() {
    c := make(chan int)
    quit := make(chan int)
    go func() {
        for i := 0; i < 10; i++ {
            fmt.Println(<-c)
        }
        quit <- 0
    }()
    fibonacci(c, quit)
}
```



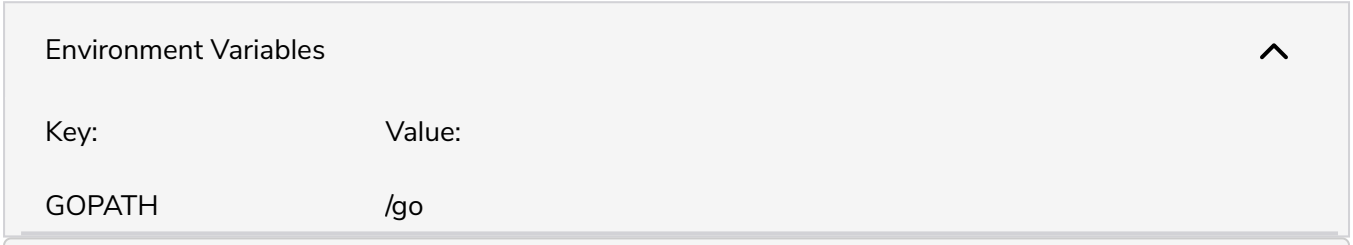
Default case

The default case in a `select` is run if no other case is ready.

Use a `default` case to try a send or receive without blocking:

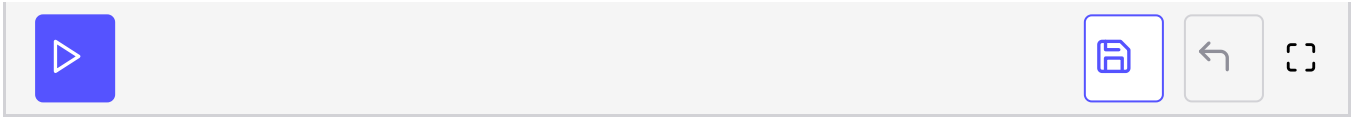
```
select {
case i := <-c:
    // use i
default:
    // receiving from c would block
}
```

Here's a complete example demonstrating the concept:




```
package main

import (
    "fmt"
    "time"
)

func main() {
    tick := time.Tick(100 * time.Millisecond)
    boom := time.After(500 * time.Millisecond)
    for {
        select {
        case <-tick:
```

```
            fmt.Println("tick.")
        case <-boom:
            fmt.Println("BOOM!")

            return
        default:
            fmt.Println("    .")
            time.Sleep(50 * time.Millisecond)
        }
    }
}
```



From the above example, you can see that first, the `default` case executes because none of the other *two* cases is ready. The moment a case is ready, such as `tick` case in the example above, the select command blocks till the case is run and `tick` is printed.

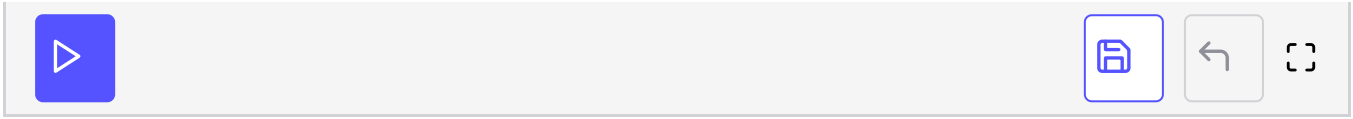
Timeout

Environment Variables

Key:	Value:
GOPATH	/go

```
package main

import (
        "fmt"
        "log"
        "net/http"
        "time"
)

func main() {
        response := make(chan *http.Response, 1)
        errors := make(chan *error)

        go func() {
                resp, err := http.Get("http://matt.aimonetti.net/")
                if err != nil {
                        errors <- &err
                }
                response <- resp
        }()
        for {
                select {
                case r := <-response:
                        fmt.Printf("%s", r.Body)
                        return
                case err := <-errors:
                        log.Fatal(*err)
```

```
                case <-time.After(200 * time.Millisecond):
                        fmt.Printf("Timed out!")
                        return

                }
        }
}
```



Note that in above example, you won't get a response due to sandboxing.

We are using the `time.After` call as a timeout measure to exit if the request didn't give a response within **200ms**.

Now that you are familiar with *goroutines* and their features. It's time to attempt a few exercise questions present in the next lesson.