Introduction to Goroutines

Let's get started with concurrency in Go by learning the basics of goroutines!

WE'LL COVER THE FOLLOWING

> A goroutine is a function or a method which is executed concurrently with the rest of the program.

Syntax

You can create a goroutine by using the following syntax:

```
go function/method(input parameters)
```

Yes, it is that simple!

The current goroutine evaluates the input parameters to the functions/methods which are executed in the new goroutines. Even our `main()` function is a goroutine which was invoked by the implicitly created goroutine managed by Go runtime.

Example

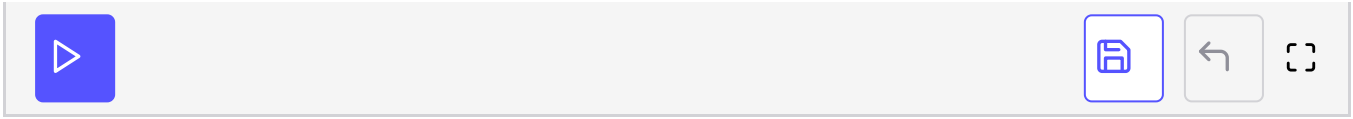
Let’s look at an example:

```
package main
import "fmt"
```

```
func WelcomeMessage(){
    fmt.Println("Welcome to Educative!")

}

func main() {
  go WelcomeMessage()
  fmt.Println("Hello World!")
}
```



Example

You can never be sure about the output of the above program. Sometimes it is just `Hello World!` while at other times the output may show both the strings `Welcome to Educative!` and `Hello World!` but in a different order.

Keep running the code and I hope you see the different kinds of output that the program generates.

Now let's discuss what was actually happening in the above program. On **line 9**, we create a goroutine which executes the function `WelcomeMessage()` concurrently. The program then moves to **line 10** and executes `fmt.Println("Hello World!")`. Remember here that the goroutine executing `WelcomeMessage()` was also running concurrently so whether you see `Hello World!` first or `Welcome to Educative!` depends on which code finishes first. But what happens when we only have `Hello World!` as the output?

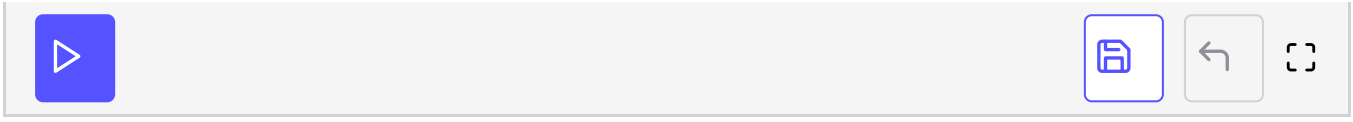
Recall that the main function is also a kind of goroutine, so when `fmt.Println("Hello World!")` executes, the `main` goroutine finishes and we exit the program regardless of what other goroutines might be doing in the background. Hence, we don't see the output from the `WelcomeMessage()` goroutine as the program exits even before the `WelcomeMessage()` function has finished.

Let's make this concept more clear in another example:

```
package main
import "fmt"

func WelcomeMessage(){
    fmt.Println("Welcome to Educative!")
}
```

```
func main() {
  go WelcomeMessage()
  go func(){
    fmt.Println("Hello World!")
  }()
}
```



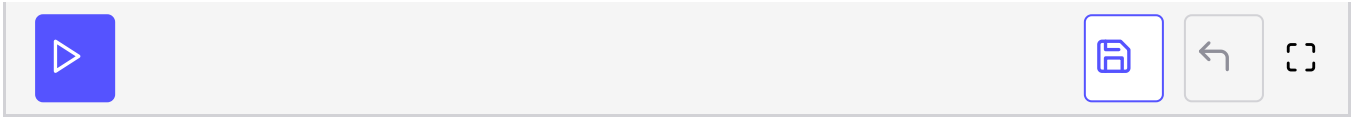
So no output, huh? If you try to figure it out, you'll realize that the main goroutine again finished causing the program to exit before we could get the results from the goroutines on **line 9** and **line 10**. Let us now pause our main goroutine and give the other goroutines time to finish as shown in the example below.

```
package main
import (
  "fmt"
  "time"
)

func WelcomeMessage(){
    fmt.Println("Welcome to Educative!")
}

func main() {
  go WelcomeMessage()
  go func(){
    fmt.Println("Hello World!")
  }()

  time.Sleep(time.Millisecond*200)
}
```



Now you can see the results from both goroutines we created in the `main` because we used the `time.Sleep()` function to cause a time delay which allows other goroutines to finish before we exit the program.

You might have realized that this is not an efficient solution but we will solve this problem by using **wait groups** from the **sync** package in the upcoming lessons.

As you are now familiar with the basics of goroutines, we'll dive deeper into what is happening behind the scenes in the next lesson!