WaitGroups

In this lesson, you will learn about the WaitGroups present in the "sync" package which will allow us some control over goroutines.

What is a `WaitGroup`?

> A WaitGroup blocks a program and waits for a set of goroutines to finish before moving to the next steps of execution.

How to use `WaitGroup`?

We can use WaitGroups through the following functions:

- `.Add(int)`: This function takes in an integer value which is essentially the number of goroutines which the waitgroup has to wait for. This function must be called before we execute a goroutine.
- `.Done()`: This function is called within the goroutine to signal that the goroutine has successfully executed.
- `.Wait()`: This function blocks the program until all the goroutines specified by `Add()` have invoked `Done()` from within.

Does this seem like a lot of information right now? Don't worry, it'll look easier once we see a sample structure to use the above functions.

```
var wg sync.WaitGroup

wg.Add(2) //called before running the goroutines
```

```
go func() {
    // Do work.
    wg.Done() //goroutine reached its completion; calling Done() to signal
}()
go func() {
    // Do work.
    wg.Done() //goroutine reached its completion; calling Done() to signal

}()

wg.Wait() //blocking the code until all the .Done() statements are executed
```

I hope things are a little clear now. Let’s make them completely clear by looking at examples.

Recall from the lesson when you were introduced to goroutines that the `main()` function exits the program without making sure that all the running sub goroutines have finished execution. We can solve this issue using WaitGroup.

Let’s first solve the issue from the first lesson of this chapter:

```go
package main
import ( "fmt"
         "sync" )

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

func main() {
  var wg sync.WaitGroup
  wg.Add(2)
  go func(){
    WelcomeMessage()
    wg.Done()
  }()
  go func(){
    fmt.Println("Hello World!")
    wg.Done()
  }()

  wg.Wait()
}
```

First of all, we import the `sync` package and declare a WaitGroup named `wg` on **line 10**. As we have two goroutines that have to be finished before we exit our program, we set the counter of `wg` to 2 by `wg.Add(2)` on **line 11**. Next, we write `wg.Done()` at the end of the execution of the goroutines (**line 14** and **line 18**). This is to give a signal that the goroutine has finished. Finally, we add `wg.Wait()` at the very end of our main program on **line 21**. As soon as the execution jumps to `wg.Wait()`, the program will halt until we get the signals from the goroutines by `wg.Done()`. `wg.Done()` decrements the counter once invoked. As soon as the counter reaches 0, the program moves on to the next statement after `wg.Wait()` which is none in this case and therefore, the program exits.

Also, notice a slight modification we made to the original program. Instead of writing `go WelcomeMessage()`, we invoked an anonymous function using goroutine and called `WelcomeMessage()` in that function. This is because the `WelcomeMessage()` function does not have access to `wg`.

Alternatively, we can pass `wg` by reference to the `WelcomeMessage()` function:

```go
package main
import (
  "fmt"
  "sync"
)

func WelcomeMessage(wg *sync.WaitGroup){
    fmt.Println("Welcome to Educative!")
    wg.Done()
}

func main() {
  var wg sync.WaitGroup

  wg.Add(2)

  go WelcomeMessage(&wg)
  go func(){
    fmt.Println("Hello World!")
    wg.Done()
  }()

  wg.Wait()

}
```

> **Note**: A WaitGroup must not be copied after first use.

This is because it can affect our counter which will disrupt the logic of our program.

In summary, waitgroups provide us with a way to make sure that all concurrent operations reach completion before the program exits. They can be vital in applications where you have to block code. However, they can only be used if we don’t want any output from those operations.

It’s time for an exercise. See you in the next lesson!