Goroutines: Explained Further

Let's get to know what is happening behind the scenes of a goroutine.

WE'LL COVER THE FOLLOWING

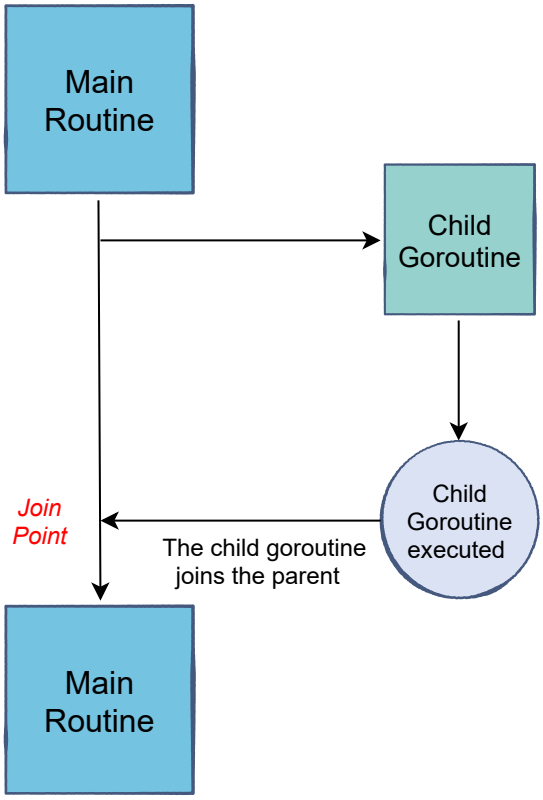
So far, you already know that the go statement runs a function in a separate thread of execution. The simplicity of the `go` command to create concurrent processes is what sets the Golang apart from other languages. Previously, developers used threads to implement concurrency in C++/Java which made it a bit complicated. But with Golang, you simply type `go function/method` and the program creates a goroutine which will execute concurrently and *immediately* return to the next line of the code in the parent routine. In doing so, the program will ignore any return values from the goroutine. This ease with which concurrency can be implemented using Go has made it popular in the concurrent programming world.

The Fork-Join Model

Go uses the idea of the fork-join model of concurrency behind goroutines. The fork-join model essentially implies that a child thread/process splits from its parent thread/process to run concurrently with the parent process. After completing its execution, the child process merges back into the parent process. The point where it joins back is called the *join point*. Goroutines work in a similar fashion. Sometimes you won't have a join point in your program, for example, in cases where goroutines only print onto the console and exit. On the other hand, if you'll have a join point, you'll have to synchronize your

goroutine with the rest of the program.

Here is an illustration to help you better understand the fork-join model:



The fork-join model

Goroutines are not necessarily running in parallel!

Clear your mind here if you have any misconceptions regarding this concept about goroutines. Goroutines don't necessarily run in parallel. They can be running sequentially in the background or can start running at different times. On the other hand, parallelism really depends on the underlying processors and CPUs available. Therefore, goroutines might give the illusion of running in parallel while just executing concurrently. This is evident from the example below:

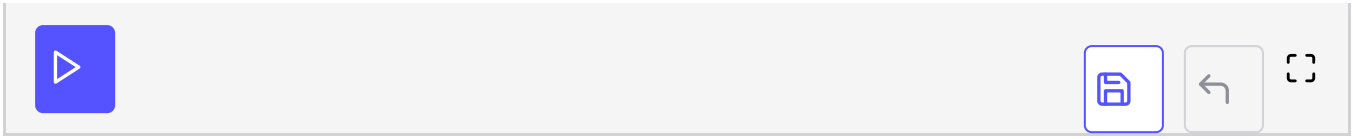
```
package main
import "fmt"

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

func main() {
  go WelcomeMessage()

  fmt.Println("Hello World!")
```

```
}
```



Prints from the goroutine and the main routine are printed at *separate times and not at the same time*. Sometimes, the goroutine waits for the main routine to print first while at other times the main routine completes its execution and exits without waiting for the goroutine to complete itself.

All goroutines have the same address space

You might have realized this by yourself but let me iterate again that all goroutines in a single program share the same address space. Hence, it is the programmer's job to make sure that they implement concurrency while keeping in mind the prevention of race conditions and the handling of synchronization issues between concurrent operations.

Goroutines as light-weight threads

What are threads?

> Threads are chunks of code which can be executed separately by the processor.

Goroutines can be thought of as light-weight threads, though bear in mind that they are not actual OS threads. A single thread may run thousands of goroutines in them using the Go runtime scheduler which uses cooperative scheduling. This implies that if the current goroutine is blocked or has been completed, the scheduler will move the other goroutines to another OS thread. Hence, we achieve efficiency in scheduling where no routine is blocked forever.

Additionally, goroutines are less costly than threads as they take up only a few kilobytes of memory.

Besides, goroutines have their own call stack which can grow and shrink dynamically. This also gives us an edge over threads as the size of the threads have to be specified earlier.

As a result, a goroutine uses far fewer resources than threads. Combining this with the clean API provided by the Go runtime for the programmer, it is easier to implement concurrency in Go than in C++/Java.

Hope you were able to absorb all the information about goroutines. Finally, it's time we learn about channels which you have been waiting for so long!

See you in the next lesson.