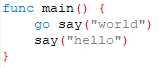
Go has several built-in primitives for concurrency.

Goroutines are one method of implementing concurrency into a program written in Go. Goroutines are essentially lightweight execution threads that are allocated to the CPU. Each CPU thread will run a goroutine until it is blocked, at which point the thread will swap it out for another goroutine.

Goroutines can be created very easily simply by prefixing a function call with “go”. For example, the below code example creates a goroutine running the “say” function.

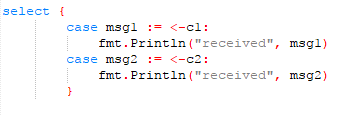


Goroutines have several differences from threads, perhaps most notably that the creation of a goroutine requires significantly less memory than a thread. Goroutines require only kilobytes of memory, while threads are created in orders of megabytes.

Another aspect Go uses for concurrency are channels. As their name suggests, channels serve as a *channel* through which data can be sent and received. Their main purpose is to transfer values between goroutines within a program. Channels are typed, so a channel of the int type can only transmit data of type int.

Once a channel is created, it can be sent data from a goroutine and goroutines can receive data from them. *<-ch* makes an executing goroutine block until it receives data from the channel. Data can be sent to the channel using a similar command *ch <- x*, where *x* is the data being sent to the channel.

Channels can also be utilized with the *select* operation. Using a select statement allows you to wait for multiple channels to send their data at once. The select will block until one of the cases within it can be run, at which point it executes that case.



In the above code snippet, each case waits to receive data and then prints the data it received.