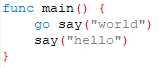
Go has several built-in primitives for concurrency.

# Goroutines

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.



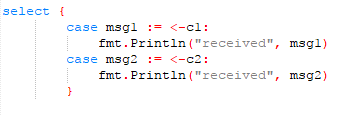
Goroutines have several differences from threads, perhaps most notably that the creation of a goroutine requires significantly less memory than a thread. Goroutines require only kilobytes of memory, while threads are created in orders of megabytes.

# Channels

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 from the first channel it received from.

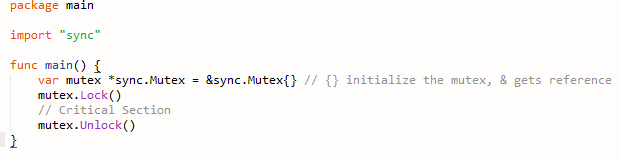
# Sync Package

There are several other types that are useful in the go language for concurrency, primarily in the form of the “sync” package. In order to access these we must include an import statement for the “sync” package. Full documentation can be found at <https://golang.org/pkg/sync/>, but we will highlight Mutex Locks, Wait Groups, and Atomic types



## Mutex Locks

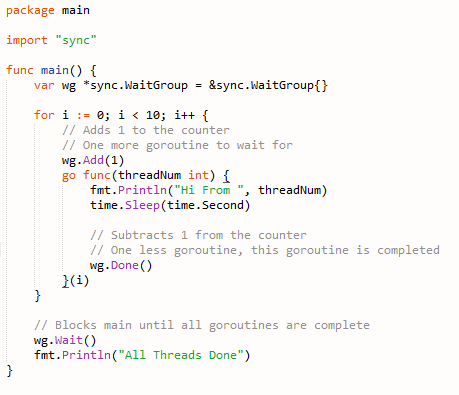
The mutex locks in go are similar to the mutex locks in pthreads. First, you must create a pointer reference to a mutex object. Then you call the Lock() function on the mutex reference, to start the critical section. After the critical section ends, you can call the Unlock() function on the mutex reference, to unlock the mutex.



## Wait Groups

Wait groups are the closest type in go to the barrier that we get in the standard go library. Waitgroups can let you sync up several Goroutines or block wait for a set amount of Goroutines to finish. After you initialize a pointer reference to a waitGroup, there are three methods available to use. You can Add a number to the current waitGroup counter. Normally, you add 1 for every goroutine you want to wait for. You can then call Done() to decrease the current waitGroup counter. Then you can call Wait() to block the current goroutine until the counter is 0.

This functionality is very useful to halt the main function until all the goroutine have finished since the main function will not wait for all Goroutines to finish normally.



## Atomic Operations

In the sync/atomic package, the go standard library offers the ability to use atomic statements instead of mutex locks. The library package, at <https://golang.org/pkg/sync/atomic/>, has a large amount of types for atomic operations and an interface to build into your own types to allow for atomic operations. By passing a reference to a type, you can increment it or add a value to it without creating a mutex lock yourself.

