Go/CSP Reference Sheet

Compiling and Running Programs

Building	`go build pgm.go` Compiles to an executable named pgm.
Running	`go run pgm.go` Runs `pgm.go` without compiling.

Data Types

General	Variables can be declared explicitly or implicitly: `var x int; //explicit` `x := 1; //implicit`
`bool`	Boolean type
`int`	Integer type
`uint`	Unsigned Integer type
`int32`	32-bit integer (equivalent to a `rune`)
`float32` `float64`	32 and 64 bit floating points
`complex64` `complex128`	64 and 128 bit complex numbers
`string`	64 bit floating points
`const`	Constant type, similar to `#define` in C e.g., `const Pi = 3.14159`

Essential Packages/Libraries

`fmt`	Contains format print and I/O functions (similar to C I/O functions).
`os`	Includes program argument vector `os.Args`.
`strings`	Contains string functions.
`sync`	Contains synchronization functions.
`flag`	Contains tools for parsing command-line flags.
`ioutils`	Contains utilities for file I/O.

OS Interface

General	os functions can be imported with `import "os"` and accessed by `os.<FunctionName>`
`os.Args[n]`	Access element *n* of the argument vector.
`len(os.Args)`	Returns number of arguments in argument vector.
`os.Exit(n)`	Exit with status *n*.
`os.Environ()`	Returns an array of strings containing environment.

Strings, Slices, and Arrays

Array	Fixed size contiguous memory e.g., `var buffer [10]int`
Slices	Piece of an array (sharing memory with the array) e.g., `var midslice = buf[4:7]`
Strings	Read-only slice of bytes (can be many formats, e.g., Unicode, UTF-8, ASCII, etc.) e.g., `var Str string = "str"`

String Functions

General	String functions can be imported with `import "strings"` and accessed by `strings.<FunctionName>`
Contains	`Contains("123", "12")` Returns `true`.
`Split`	`Split("1 2 3", " ")` Returns the array [1, 2, 3].
`Join`	`Join(array,char)` Joins an array of strings (`array`), inserting `char` between each string (`char` can be `""`)
`len`	`len("Hello") // returns 5`
`"four"[n]`	Access index `n` of `"four"`

Communication through Channels

Spawning goroutines	`go <functionname>()`
Channels	Channels are pipes through which concurrent goroutines communicate, by sending values to each other.
Unbuffered Channels	goroutine will block unless another goroutine is waiting to read/write from the channel.
Buffered Channels	Channel contains a buffer that can be written/read without a goroutine waiting to read/write from the channel.
I/O with channels	Write to channel: e.g., `channel <- 1` Read from channel: `x = <- channel`
`select { case <I/O>: default: }`	Used to prevent a goroutine from blocking by selecting a ready channel.

Wait Groups

General	Allows for a parent goroutine to wait for a collection of other goroutines to terminate.
Declaring wait groups	`var wg = &sync.WaitGroup{}`
Adding	`wg.Add(n)` Adds `n` goroutines to the wait group.
Signaling	`wg.Done()` Calling goroutine signals it is finished.
Waiting	`wg.Wait()` Calling goroutine blocks until requisite child routines terminate.