Go Cheat Sheet

▶ Table of Contents

Credits

Most example code taken from A Tour of Go, which is an excellent introduction to Go. If you're new to Go, do that tour. Seriously.

Go in a Nutshell

- Imperative language
- Statically typed
- Syntax tokens similar to C (but less parentheses and no semicolons) and the structure to Oberon-2
- Compiles to native code (no JVM)
- No classes, but structs with methods
- Interfaces
- No implementation inheritance. There's type embedding, though.
- Functions are first class citizens
- Functions can return multiple values
- Has closures
- Pointers, but not pointer arithmetic
- Built-in concurrency primitives: Goroutines and Channels

Hello World

File `hello.go`:

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello Go")
}
```

```
$ go run hello.go
```

Operators

Arithmetic Comparison

Operator	Description		Operator	Description
`+`	addition		`==`	equal
`-`	subtraction		`!=`	not equal
`*`	multiplication		`<`	less than
`/`	quotient		`<=`	less than or equal
`%`	remainder		`>`	greater than
`&`	bitwise and		`>=`	greater than or equal
`|`	bitwise or			
`^`	bitwise xor			
`&^`	bit clear (and not)			
`<<`	left shift			
`>>`	right shift			

Logical Others

Operator	Description		Operator	Description
`&&`	logical and		`&`	address of or pointer creator
`||`	logical or		`*`	dereference pointer
`!`	logical not		`<-`	send/receive operator (see `Channels` section)

Declarations

Type goes after identifier!

```go
var foo int                 // declaration without initialization
var foo int = 42            // declaration with initialization
var foo, bar int = 42, 1302 // declare and init multiple vars at once
var foo = 42                // type omitted, will be inferred
foo := 42                   // shorthand, only in func bodies, omit var
                            // keyword, type is always implicit
const constant = "This is a constant"
```

```go
// iota can be used for incrementing numbers, starting from 0
const (
    _ = iota
    a
    b
    c = 1 << iota
    d
)
    fmt.Println(a, b)   // 1 2 (0 is skipped)
    fmt.Println(c, d)   // 8 16 (2^3, 2^4)
```

Functions

```go
// a simple function
func functionName() {}

// function with parameters (again, types go after identifiers)
func functionName(param1 string, param2 int) {}

// multiple parameters of the same type
func functionName(param1, param2 int) {}

// return type declaration
func functionName() int {
    return 42
}

// Can return multiple values at once
func returnMulti() (int, string) {
    return 42, "foobar"
}
var x, str = returnMulti()

// Return multiple named results simply by return
func returnMulti2() (n int, s string) {
    n = 42
    s = "foobar"
    // n and s will be returned
    return
}
var x, str = returnMulti2()
```

Functions As Values And Closures

```go
func main() {
    // assign a function to a name
    add := func(a, b int) int {
        return a + b
    }
    // use "add" to call the function
    fmt.Println(add(3, 4))
}
```

A closure is a function that captures the variables from its surrounding scope. This allows the function to access those variables even when it is executed outside of that scope. Closures are commonly used for creating functions with private variables or for functional programming techniques.

```go
// scope() is a function that returns a closure.
// Closures, lexically scoped: Functions can access values that were
// in scope when defining the function
func scope() func() int{
    outer_var := 2
    foo := func() int { return outer_var}
    return foo
}

func another_scope() func() int{
    // won't compile because outer_var and foo not defined in this scope
    outer_var = 444
    return foo
}

// outer() return a closure
func outer() (func() int, int) {
    outer_var := 2
    inner := func() int {
        outer_var += 99 // outer_var from outer scope is mutated.
        return outer_var
    }
    inner()
    return inner, outer_var // return inner func and mutated outer_var 101
}
```

Variadic Functions

A variadic function can accept a variable number of arguments.

```go
func main() {
        fmt.Println(adder(1, 2, 3))     // 6
        fmt.Println(adder(9, 9))        // 18

        nums := []int{10, 20, 30}
        fmt.Println(adder(nums...))     // 60
}
```

```go
// By using ... before the type name of the last parameter you can indicate that it takes zero or more of those parameters.
// The function is invoked like any other function except we can pass as many arguments as we want.
func adder(args ...int) int {
	total := 0
	for _, v := range args { // Iterates over the arguments whatever the number.
		total += v
	}
	return total
}
```

Built-in Types

```
bool

string

int  int8  int16  int32  int64
uint uint8 uint16 uint32 uint64 uintptr

byte // alias for uint8

rune // alias for int32, it is 4 bytes. It is used for a character (good for a unicode character)

float32 float64

complex64 complex128
```

All Go's predeclared identifiers are defined in the builtin package.

Type Conversions

```go
var i int = 42
var f float64 = float64(i)
var u uint = uint(f)

// alternative syntax
i := 42
f := float64(i)
u := uint(f)
```

Packages

- Package declaration at top of every source file
- Executables are in package `main`
- Convention: package name == last name of import path (import path `math/rand` => package `rand`)
- Upper case identifier: exported (visible from other packages)
- Lower case identifier: private (not visible from other packages)

Control structures

If

```go
func main() {
	// Basic one
	if x > 10 {
		return x
	} else if x == 10 {
		return 10
	} else {
		return -x
	}

	// You can put one statement before the condition
	if a := b + c; a < 42 {
		return a
	} else {
		return a - 42
	}

	// Type assertion inside if
	var val interface{} = "foo"
	if str, ok := val.(string); ok {
		fmt.Println(str)
	}
}
```

Loops

```go
    // There's only `for`, no `while`, no `until`
    for i := 1; i < 10; i++ {
    }
    for ; i < 10;  { // while - loop
    }
    for i < 10  { // you can omit semicolons if there is only a condition
    }
    for { // you can omit the condition ~ while (true)
    }

    // use break/continue on current loop
```

```go
    // use break/continue with label on outer loop
here:
    for i := 0; i < 2; i++ {
        for j := i + 1; j < 3; j++ {
            if i == 0 {
                continue here
            }
            fmt.Println(j)
            if j == 2 {
                break
            }
        }
    }

there:
    for i := 0; i < 2; i++ {
        for j := i + 1; j < 3; j++ {
            if j == 1 {
                continue
            }
            fmt.Println(j)
            if j == 2 {
                break there
            }
        }
    }
```

Switch

```go
    // switch statement
    switch operatingSystem {
    case "darwin":
        fmt.Println("Mac OS Hipster")
        // cases break automatically, no fallthrough by default
    case "linux":
        fmt.Println("Linux Geek")
    default:
        // Windows, BSD, ...
        fmt.Println("Other")
    }

    // as with for and if, you can have an assignment statement before the switch value
    switch os := runtime.GOOS; os {
    case "darwin": ...
    }

    // you can also make comparisons in switch cases
    number := 42
    switch {
        case number < 42:
            fmt.Println("Smaller")
        case number == 42:
            fmt.Println("Equal")
        case number > 42:
            fmt.Println("Greater")
    }

    // cases can be presented in comma-separated lists
    var char byte = '?'
    switch char {
        case ' ', '?', '&', '=', '#', '+', '%':
            fmt.Println("Should escape")
    }
```

Arrays, Slices, Ranges

Arrays

```go
var a [10]int // declare an int array with length 10. Array length is part of the type!
a[3] = 42     // set elements
i := a[3]     // read elements

// declare and initialize
var a = [2]int{1, 2}
a := [2]int{1, 2} //shorthand
a := [...]int{1, 2} // elipsis -> Compiler figures out array length
```

Slices

```go
var a []int                              // declare a slice - similar to an array, but length is unspecified
var a = []int {1, 2, 3, 4}               // declare and initialize a slice (backed by the array given implicitly)
a := []int{1, 2, 3, 4}                   // shorthand
chars := []string{0:"a", 2:"c", 1: "b"}  // ["a", "b", "c"]

var b = a[lo:hi]       // creates a slice (view of the array) from index lo to hi-1
var b = a[1:4]         // slice from index 1 to 3
var b = a[:3]          // missing low index implies 0
var b = a[3:]          // missing high index implies len(a)
a =  append(a,17,3)    // append items to slice a
c := append(a,b...)    // concatenate slices a and b

// create a slice with make
a = make([]byte, 5, 5)  // first arg length, second capacity
a = make([]byte, 5)     // capacity is optional

// create a slice from an array
```

```go
x := [3]string{"Лайка", "Белка", "Стрелка"}
s := x[:] // a slice referencing the storage of x
```

Operations on Arrays and Slices

`len(a)` gives you the length of an array/a slice. It's a built-in function, not a attribute/method on the array.

```go
// loop over an array/a slice
for i, e := range a {
    // i is the index, e the element
}

// if you only need e:
for _, e := range a {
    // e is the element
}

// ...and if you only need the index
for i := range a {
}

// In Go pre-1.4, you'll get a compiler error if you're not using i and e.
// Go 1.4 introduced a variable-free form, so that you can do this
for range time.Tick(time.Second) {
    // do it once a sec
}
```

Maps

```go
m := make(map[string]int)
m["key"] = 42
fmt.Println(m["key"])

delete(m, "key")

elem, ok := m["key"] // test if key "key" is present and retrieve it, if so

// map literal
var m = map[string]Vertex{
    "Bell Labs": {40.68433, -74.39967},
    "Google":    {37.42202, -122.08408},
}

// iterate over map content
for key, value := range m {
}
```

Structs

There are no classes, only structs. Structs can have methods.

```go
// A struct is a type. It's also a collection of fields

// Declaration
type Vertex struct {
    X, Y float64
}

// Creating
var v = Vertex{1, 2}
var v = Vertex{X: 1, Y: 2} // Creates a struct by defining values with keys
var v = []Vertex{{1,2},{5,2},{5,5}} // Initialize a slice of structs

// Accessing members
v.X = 4

// You can declare methods on structs. The struct you want to declare the
// method on (the receiving type) comes between the the func keyword and
// the method name. The struct is copied on each method call(!)
func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

// Call method
v.Abs()

// For mutating methods, you need to use a pointer (see below) to the Struct
// as the type. With this, the struct value is not copied for the method call.
func (v *Vertex) add(n float64) {
    v.X += n
    v.Y += n
}
```

Anonymous structs: Cheaper and safer than using `map[string]interface{}` .

```go
point := struct {
        X, Y int
}{1, 2}
```

Pointers

```go
p := Vertex{1, 2}  // p is a Vertex
q := &p            // q is a pointer to a Vertex
r := &Vertex{1, 2} // r is also a pointer to a Vertex
```

```go
// The type of a pointer to a Vertex is *Vertex

var s *Vertex = new(Vertex) // new creates a pointer to a new struct instance
```

Interfaces

Used to implement Abstraction, Polymorphism, Decoupling and Testing.

```go
// Interface declaration
type Awesomizer interface {
    Awesomize() string
}

// Type Foo implements the Awesomizer interface
type Foo struct{}

func (foo Foo) Awesomize() string {
    return "Awesome!"
}

// Another type Bar implements the Awesomizer interface
type Bar struct{}

func (bar Bar) Awesomize() string {
    return "Super Awesome!"
}

// Function that accepts any type that implements the Awesomizer interface
func PrintAwesomeness(a Awesomizer) {
    fmt.Println(a.Awesomize())
}

func main() {
    foo := Foo{}
    bar := Bar{}

    PrintAwesomeness(foo)
    PrintAwesomeness(bar)
}
```

Embedding

Embedding is a way to include one struct or interface within another struct or interface. This allows the embedded type's methods to be accessed directly from the embedding type, providing a form of composition and reuse without inheritance.

```go
// ReadWriter implementations must satisfy both Reader and Writer
type ReadWriter interface {
    Reader
    Writer
}

// Server exposes all the methods that Logger has
type Server struct {
    Host string
    Port int
    *log.Logger
}

// initialize the embedded type the usual way
server := &Server{"localhost", 80, log.New(...)}

// methods implemented on the embedded struct are passed through
server.Log(...) // calls server.Logger.Log(...)

// the field name of the embedded type is its type name (in this case Logger)
var logger *log.Logger = server.Logger
```

Benefits of Embedding

Code Reuse. Embedding allows you to reuse the functionality of the embedded type without having to write wrapper methods. **Composition Over Inheritance.** Go promotes composition over inheritance, and embedding is a way to achieve that. It allows you to build complex types by combining simpler ones.

Errors

There is no exception handling. Instead, functions that might produce an error just declare an additional return value of type [`error`](). This is the `error` interface:

```go
// The error built-in interface type is the conventional interface for representing an error condition,
// with the nil value representing no error.
type error interface {
    Error() string
}
```

Here's an example:

```go
func sqrt(x float64) (float64, error) {
        if x < 0 {
                return 0, errors.New("negative value")
        }
```

```
		return math.Sqrt(x), nil
}

func main() {
	val, err := sqrt(-1)
	if err != nil {
		// handle error
		fmt.Println(err) // negative value
		return
	}
	// All is good, use `val`.
	fmt.Println(val)
}
```

Concurrency

In go, concurrency is implemented using Goroutines and Channels.

Goroutines

Goroutines are lightweight threads managed by Go (are not OS threads). `go f(a, b)` starts a new goroutine which runs the function `f`. You can use `sync.WaitGroup` to wait for goroutines to finish.

```
func doStuff(wg *sync.WaitGroup) {
	defer wg.Done() // Signal goroutine is done
	fmt.Println("Doing stuff...")
	time.Sleep(2 * time.Second)
	fmt.Println("Done with stuff")
}

func main() {
	var wg sync.WaitGroup

	wg.Add(1)
	go doStuff(&wg)

    // an anonymous function can be used with goroutines too
	wg.Add(1)
	go func(wg *sync.WaitGroup, x int) {
		defer wg.Done()
		fmt.Println("Doing anonymous stuff with", x)
		time.Sleep(4 * time.Second)
		fmt.Println("Done with anonymous stuff")
	}(&wg, 42)

	wg.Wait() // Wait for all goroutines in the WaitGroup to finish
	fmt.Println("Main function ended")
}
```

Channels

Channels are a way to communicate between goroutines. They allow you to send and receive values between different goroutines, enabling synchronization and data exchange.

```
ch := make(chan int) // create a channel of type int
ch <- 42             // Send a value to the channel ch.
v := <-ch            // Receive a value from ch

// Non-buffered channels block. Read blocks when no value is available, write blocks until there is a read.

// Create a buffered channel. Writing to a buffered channels does not block if less than <buffer size> unread values have been written.
ch := make(chan int, 100)

close(ch) // closes the channel (only sender should close)

// read from channel and test if it has been closed
v, ok := <-ch

// if ok is false, channel has been closed

// Read from channel until it is closed
for i := range ch {
    fmt.Println(i)
}

// select blocks on multiple channel operations, if one unblocks, the corresponding case is executed
func doStuff(channelOut, channelIn chan int) {
    select {
    case channelOut <- 42:
        fmt.Println("We could write to channelOut!")
    case x := <- channelIn:
        fmt.Println("We could read from channelIn")
    case <-time.After(time.Second * 1):
        fmt.Println("timeout")
    }
}
```

The following example demonstrates how to use channels to coordinate work between multiple goroutines. The `worker` function reads jobs from the `jobs` channel, processes them, and sends results to the `results` channel. It uses a `sync.WaitGroup` to signal when it is done.

```
func worker(id int, jobs <-chan int, results chan<- int, wg *sync.WaitGroup) {
    defer wg.Done()
    for j := range jobs {
        fmt.Printf("Worker %d started job %d\n", id, j)
        time.Sleep(time.Second) // Simulate some work with a sleep
```

```go
        fmt.Printf("Worker %d finished job %d\n", id, j)
        results <- j * 2 // Send the result to the results channel
    }
}

func main() {
    const numJobs = 5
    jobs := make(chan int, numJobs)
    results := make(chan int, numJobs)
    var wg sync.WaitGroup

    // Start 3 worker goroutines
    for w := 1; w <= 3; w++ {
        wg.Add(1)
        go worker(w, jobs, results, &wg)
    }

    // Send jobs to the jobs channel
    for j := 1; j <= numJobs; j++ {
        jobs <- j
    }
    close(jobs) // Close the jobs channel to indicate no more jobs

    // Wait for all workers to finish
    wg.Wait()
    close(results) // Close the results channel

    // Collect results
    for result := range results {
        fmt.Println("Result:", result)
    }
}
```

Established Facts About Channels (Channel Axioms)

A send to a nil channel blocks forever.

```go
var c chan string
c <- "Hello, World!"
// fatal error: all goroutines are asleep - deadlock!
```

A receive from a nil channel blocks forever.

```go
var c chan string
fmt.Println(<-c)
// fatal error: all goroutines are asleep - deadlock!
```

A send to a closed channel panics.

```go
var c = make(chan string, 1)
c <- "Hello, World!"
close(c)
c <- "Hello, Panic!"
// panic: send on closed channel
```

A receive from a closed channel returns the zero value immediately.

```go
var c = make(chan int, 2)
c <- 1
c <- 2
close(c)
for i := 0; i < 3; i++ {
    fmt.Printf("%d ", <-c)
}
// 1 2 0
```

Printing With Formatted I/O Functions

```go
fmt.Println("Hello, 你好, नमस्ते, Привет, ᎣᏏᏲ")                      // basic print with
newline
p := struct { X, Y int }{ 17, 2 }
fmt.Println( "My point:", p, "x coord=", p.X )                  // print structs, ints,
etc
s := fmt.Sprintln( "My point:", p, "x coord=", p.X )            // print to string
variable

fmt.Printf("%d hex:%x bin:%b fp:%f sci:%e",17,17,17,17.0,17.0)  // c-like format
s2 := fmt.Sprintf( "%d %f", 17, 17.0 )                          // formatted print to
string

// multi-line string literal, using back-tick at beginning and end
hellomsg := `
 "Hello" in Chinese is 你好 ('Ni Hao')
 "Hello" in Hindi is नमस्ते ('Namaste')
`
```

Reflection

Reflection uses the `reflect` package to inspect and manipulate types and values at runtime. `Type Switch` determines the type of an interface value at runtime and executes code based on that type.

While both can be used to work with dynamic types, reflection is more powerful and flexible, allowing deeper inspection and manipulation of types and values.

```go
func main() {
    var x int = 42
    t := reflect.TypeOf(x)
    v := reflect.ValueOf(x)
    fmt.Println("Type:", t)
    fmt.Println("Value:", v)
}
```

Type Switch

A type switch is like a regular switch statement, but the cases in a type switch specify types (not values) which are compared against the type of the value held by the given interface value.

```go
func do(i interface{}) {
	switch v := i.(type) {
	case int:
		fmt.Printf("Twice %v is %v\n", v, v*2)
	case string:
		fmt.Printf("%q is %v bytes long\n", v, len(v))
	default:
		fmt.Printf("I don't know about type %T!\n", v)
	}
}

func main() {
	do(21)
	do("hello")
	do(true)
}
```

Snippets

Files Embedding

Go programs can embed static files using the `"embed"` package as follows:

```go
package main

import (
	"embed"
	"log"
	"net/http"
)

// content holds the static content (2 files) for the web server.
//go:embed a.txt b.txt
var content embed.FS

func main() {
	http.Handle("/", http.FileServer(http.FS(content)))
	log.Fatal(http.ListenAndServe(":8080", nil))
}
```

[Full Playground Example]()

HTTP Server

```go
package main

import (
    "fmt"
    "net/http"
)

// define a type for the response
type Hello struct{}

// let that type implement the ServeHTTP method (defined in interface http.Handler)
func (h Hello) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, "Hello!")
}

func main() {
    var h Hello
    http.ListenAndServe("localhost:4000", h)
}

// Here's the method signature of http.ServeHTTP:
// type Handler interface {
//     ServeHTTP(w http.ResponseWriter, r *http.Request)
// }
```