Go Cheat Sheet

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.

Original HTML Cheat Sheet by Ariel Mashraki (a8m):
https://github.com/a8m/go-lang-cheat-sheet

Go in a Nutshell

- Imperative language
- Statically typed
- Syntax similar to Java/C/C++, but less parentheses and no semicolons
- 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

Basic Syntax

Hello World

File `hello.go`:

```
package main

import "fmt"

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

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

Operators

Arithmetic

Operator	Description
+	addition
-	subtraction
*	multiplication
/	quotient
%	remainder
&	bitwise AND
|	bitwise OR
^	bitwise XOR
&^	bit clear (AND NOT)
<<	left shift
>>	right shift (logical)

Comparison

Operator	Description
==	equal
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

Logical

Operator	Description
&&	logical AND
||	logical OR
!	logical NOT

Other

Operator	Description
&	address of / create pointer
*	dereference pointer
<-	send / receive operator (see 'Channels' below)

Declarations

Type goes **after** identifier!

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

Functions

```
// 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) {}
```

Functions (cont)

```
// 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

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

// 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 - outer_var and foo not defined in this scope
    outer_var = 444
    return foo
}

// Closures: don't mutate outer vars, instead redefine them!
func outer() (func() int, int) {
    outer_var := 2       // NOTE outer_var is outside inner's scope
    inner := func() int {
        outer_var += 99  // attempt to mutate outer_var
        return outer_var // => 101 (but outer_var is a newly redefined
                         //  variable visible only inside inner)
    }
    return inner, outer_var // => 101, 2 (still!)
}
```

Functions (cont)

Variadic Functions

```
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
}

// Using  ... before the type name of the last parameter indicates
//  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 { // Iterate over all args
        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 ~= (Unicode code point) - Very Viking

float32 float64

complex64 complex128
```

Type Conversions

```
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

```
func main() {
    // Basic one
    if x > 0 {
        return x
    } 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{}
    val = "foo"
    if str, ok := val.(string); ok {
        fmt.Println(str)
    }
}
```

Loops

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

Control structures (cont)

Switch

```
    // 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, an assignment statement before the
    //  switch value is allowed
    switch os := runtime.GOOS; os {
        case "darwin": ...
    }
```

Arrays, Slices, Ranges

Arrays

```
var a [10]int  // int array with length 10. Length is part of 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

```
var a []int    // a slice – like an array, but length is unspecified
var a = []int {1, 2, 3, 4}  // declare and initialize a slice
                            //  (backed by given array 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)

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

Arrays, Slices, Ranges (cont)

```
// create a slice from an array
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.

```
// 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, it is a compiler error to not use i and e.
// Go 1.4 introduced a variable-free form:
for range time.Tick(time.Second) {
    // do it once a sec
}
```

Maps

```
var m map[string]int
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, retrieve if so

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

Structs

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

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

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

// Creating
var v = Vertex{1, 2}
var v = Vertex{X: 1, Y: 2} // Creates a struct by defining values
                           //  with keys

// 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 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{}`.

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

Pointers

```
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

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

var s *Vertex = new(Vertex) // create ptr to a new struct instance
```

Interfaces

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

// types do *not* declare to implement interfaces
type Foo struct {}

// instead, types implicitly satisfy an interface if they implement
all required methods
func (foo Foo) Awesomize() string {
    return "Awesome!"
}
```

Embedding

There is no subclassing in Go. Instead, there is interface and struct embedding (composition).

```
// 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(...)

// Field name of an embedded type is its type name ('Logger' here)
var logger *log.Logger = server.Logger
```

Errors

There is no exception handling. Functions that might produce an error just declare an additional return value of type `Error`. This is the `Error` interface:

```
type error interface {
    Error() string
}
```

Errors (cont)

A function that might return an error:

```
func doStuff() (int, error) {
}

func main() {
    result, error := doStuff()
    if (error != nil) {
        // handle error
    } else {
        // all is good, use result
    }
}
```

Concurrency

Goroutines

Goroutines are lightweight threads (managed by Go, not OS threads). `go f(a, b)` starts a new goroutine which runs `f` (given `f` is a function).

```
// just a function (which can be later started as a goroutine)
func doStuff(s string) {
}

func main() {
    // using a named function in a goroutine
    go doStuff("foobar")

    // using an anonymous inner function in a goroutine
    go func (x int) {
        // function body goes here
    }(42)
}
```

Channels

```
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 if a value already has been written
// but not 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(c) // closes the channel (only sender should close)

// Read from channel and test if it has been closed
// If ok is false, channel has been closed
v, ok := <-ch

// 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")
    }
}
```

Channel Axioms

```
// I. A send to a nil channel blocks forever
var c chan string
c <- "Hello, World!"
// fatal error: all goroutines are asleep - deadlock!


// II. A receive from a nil channel blocks forever
var c chan string
fmt.Println(<-c)
// fatal error: all goroutines are asleep - deadlock!


// III. A send to a closed channel panics
var c = make(chan string, 1)
c <- "Hello, World!"
close(c)
```

Channels (cont)

```
c <- "Hello, Panic!"
// panic: send on closed channel


// IV. A receive from a close channel returns the zero value
// immediately
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
```

Snippets

HTTP Server

```
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)
// }
```