Developer Learning Solutions

Go Cheat Sheet

Importing Packages

```
import "fmt"
```

```
// import multiple packages
import (
    "fmt"
    r "math/rand"
)
...
fmt.Println(r.Intn(100))
```

Running Go Program

```
// build
$ go build hello.go
$ ./hello
```

```
// build and run
$ go run hello.go
```

Variables

```
// type inferred
var num1 = 5

// explicitly typed
var num2 int = 6

// declare and init
num3 := 7
num4 := num3

// declare as string
var str string

// declare as float and init
var rates float32 = 4.5

// declare as bool and init
var raining bool = false

// multiple declares and assign
var num5, num6 int = 8, 9

var (
    age = 25
    name = "Samuel"
)
```

String Interpolation

```
var num1 = 5
var rates float32 = 4.5

str := fmt.Sprintf(
    "num1 is %d and rates is
%.2f", num1, rates)
```

Type Conversion

```
num1 := 2
num2 := float32(num1)
num3 := uint(num1)

fmt.Println(num1) // 2
fmt.Println(num2) // 2
fmt.Println(num3) // 2

num2 = 3.5
num4 := int(num2)
fmt.Println(num2) // 3.5
fmt.Println(num4) // 3
```

```
sumStr := "123"

import "strconv"

// convert string to int
sum, err :=
    strconv.Atoi(sumStr)

// convert int to string
str := strconv.Itoa(sum)
```

Type Alias

```
type PeopleName string
var name PeopleName
name = "Jonathan"
```

Arrays

```
// array with 5 elements
var nums [5] int  // [0 0 0 0 0]
```

Slices

```
// creates a slice of 5
// elements, capacity = 5
x := make([] int, 5)
// [0 0 0 0 0]
```

```
// creates a slice of 2
// elements, capacity = 3
x = make([] int, 2, 3)
fmt.Println(x)     // [0 0]
```

```
primes := [] int {2, 3, 5, 7,
11, 13}
```

Slicing

```
var c[3] string
c[0] = "iOS"
c[1] = "Android"
c[2] = "Windows"

fmt.Println(c[0:2])
//[iOS Android]
```

Ranging

```
primes := [] int {2, 3, 5, 7,
                  11, 13}
for i, v := range primes {
    fmt.Println(i, v)
}
```

Decision Making

```
if true {
    fmt.Println(true)
} else {
    fmt.Println(false)
}
```

```
if sum := add(5,6); sum < limit
{
    fmt.Println(sum)
} else {
    fmt.Println(limit)
}
```

Switch

```
grade := "B"
switch grade {
case "A":
    fallthrough
case "B":
    fallthrough
case "C":
    fallthrough
case "D":
    fmt.Println("Passed")
case "F":
    fmt.Println("Failed")
default:
    fmt.Println("Undefined")
// Passed
```

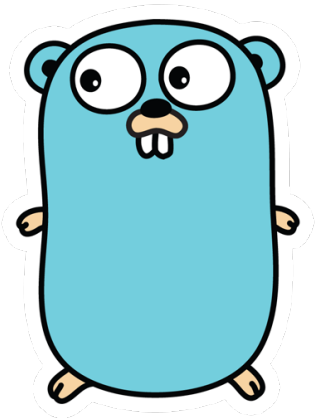
Structs

```
type Point struct {
    X int
    Y int
}
```

```
ptA := Point{5,6}

// ptB is a reference to ptA
ptB := &ptA
```

```
ptC := Point{X:2,Y:3}
```



Looping

```
for i:=0; i<5; i++ {
    fmt.Println(i)
}

// infinite loop
for {
}

// "while" loop
counter := 0
for counter <5 {
    fmt.Println(counter)
    counter++
}

// iterate through an array
for i,n:= range c {
    fmt.Println(i, n)
}
```

Developer Learning Solutions

Maps

```
// declaring a map type
var heights map[string] int

// initialize the map
heights = make(map [string] int)
```

```
heights["Peter"] = 178
delete(heights, "Mike")
```

```
// checking for a key
value, ok := heights["Matt"]
```

```
// iterating over a map
for k, v := range heights {
    fmt.Println(k,v)
}
```

```
// creating an empty map
weights := map[string] float32
{}
```

```
// create and init
weights = map[string] float32 {
    "Peter": 45.9,
    "Joan": 56.8,
}
```

Function

```
func doSomething() {
   fmt.Println("Hello")
}
```

```
// calling a function
doSomething()
```

Function with Parameters and Return Value

```
// returns int result
func addNum(num1 int, num2 int)
int {
    return num1 + num2
}
```

Variadic Functions

```
func addNums(nums ... int) int {
    total := 0
    for _, n := range nums {
        total += n
    }
    return total
}

fmt.Println(addNums(1,2,3,4,5))
```

Multiple Return Type Functions

```
func countOddEven(s string)
(int,int) {
    odds, evens := 0,0
    for _, c := range s {
        if int(c) % 2 == 0 {
            evens++
        } else {
            odds++
        }
    }
    return odds,evens
}
```

Named Return Type Functions

```
func countOddEven(s string)
(odds,evens int) {
    odds, evens = 0, 0
    for _, c := range s {
        if int(c) % 2 == 0 {
            evens++
        } else {
            odds++
        }
    }
    return
}
```

Anonymous Functions

```
// declare i to be a function
// that returns int
var i func() int

i = func() int {
    return 5
}
fmt.Println(i())
```

Closure

```
func fib() func() int {
    f1 := 0
    f2 := 1
    return func() int {
        f1, f2 = f2, (f1 + f2)
        return f1
    }
}

gen := fib()
for i := 0; i < 10; i++ {
    fmt.Println(gen())
}
```

Goroutines

```
func say(s string, times int) {
    for i := 0; i < times; i++ {
        time.Sleep(
            time.Millisecond)
        fmt.Println(s)
    }
}

go say("Hello", 3)
go say("Hello", 2)
```

Channels

```
// create a channel to store
// strings
ch := make(chan string)
```

```
// sending to a channel
ch <- "A"
```

```
// retrieving from a channel
s := <-ch
```

```
// closing a channel
close(ch)
```

```
// check if a channel is closed
v, ok := <-ch
```

Waiting on Channels

```
select {
case s1 := <-c1:
    fmt.Println(s1)
case s2 := <-c2:
    fmt.Println(s2)
}
```

Using Mutuex

```
import "sync"
var mutex = &sync.Mutex{}
```

```
func credit() {
    mutex.Lock()
    balance += 100
    mutex.Unlock()
    }
}
```

Atomic Counters

```
import "sync/atomic"
// adds 100 to balance
// atomically
atomic.AddInt64(&balance, 100)
```

WaitGroup

```
func credit(wg *sync.WaitGroup)
{
    defer wg.Done()
    atomic.AddInt64(&balance,
        100)
}

func debit(wg *sync.WaitGroup) {
    defer wg.Done()
    atomic.AddInt64(&balance,
        -100)
}

func main() {
    var wg sync.WaitGroup

    balance = 500
    wg.Add(1)
    go credit(&wg)

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

    wg.Wait()
    // code block until the
    // WaitGroup counter reaches
    // 0
    fmt.Println(balance)
}
```

Packages

```
package mystrings

func internalFunction() {
  // In Go, a name is exported
  // if it begins with a capital
  // letter
}

// Must begin with a capital
// letter in order to be
// exported
func CountOddEven(s string)
(odds,evens int) {
    ...
}
```