

Go cheatsheet

Proudly sponsored by

New: DigitalOcean Marketplace Deploy your favorite dev tools with 1-Click Apps

ethical ad by CodeFund

Hello world

hello.go

```
package main
```

```
import "fmt"
```

```
func main() {  
    message := greetMe("world")  
    fmt.Println(message)  
}
```

```
func greetMe(name string) string {  
    return "Hello, " + name + "!"  
}
```

```
$ go build
```

Or try it out in the [Go repl](#), or [A Tour of Go](#).

Variables

Variable declaration

```
var msg string  
msg = "Hello"
```

Shortcut of above (Infers type)

```
msg := "Hello"
```

Constants

```
const Phi = 1.618
```

Constants can be cl

See: [Constants](#)

Basic types

Strings

```
str := "Hello"
```

```
str := `Multiline
string`
```

Strings are of type string.

Numbers

Typical types

```
num := 3           // int
num := 3.          // float64
num := 3 + 4i      // complex128
num := byte('a')  // byte (alias for uint8)
```

Other types

```
var u uint = 7      // uint (unsigned)
var p float32 = 22.7 // 32-bit float
```

Arrays

```
// var numbers [5]
numbers := [...]i
```

Arrays have a fixed

Slices

```
slice := []int{2,
```

Pointers

```
func main () {
    fmt.Println("Value is", b)
}
```

```
func getPointer () (myPointer *int) {
    a := 234
}
```

Pointers point to a memory location of a variable. Go is fully garbage-collected.

See: Pointers

Type conversions

```
i := 2
f := float64(i)
u := uint(i)
```

See: Type conversions

Flow control

Conditional

```
rest()

groan()

- work()
}
```

See: [If](#)

Statements in if

```
fmt.Println("Uh oh")
}
```

A condition in an if statement can be preceded with a statement
See: [If with a short statement](#)

Switch

```
switch day {
case "sunday":
    // cases don't
    fallthrough

case "saturday"
    rest()

default:
    work()
}
```

For loop

For-Range loop

```
entry := []string{"Jack", "John", "Jones"}
for i, val := range entry {
    fmt.Printf("At position %d, the character %s is present\n", i, val)
}
```

See: [For-Range loops](#)

Functions

Lambdas

```
return x > 10000
}
```

Functions are first class objects.

Multiple return types

```
a, b := getMessage()
```

```
func getMessage() (a string, b string) {
}
```

Named return va

```
func split(sum int) (x, y int) {
    x = sum * 4 / 9
    y = sum - x
}
```

By defining the return values

See: [Named return values](#)

Packages

Importing

```
import "fmt"
import "math/rand"
```

```
import (
    "fmt"      // gives fmt.Println
    "math/rand" // gives rand.Intn
)
```

Aliases

```
r.Intn()
```

Packages

```
package hello
```

Exporting names

```
func Hello () {
    ...
}
```

Exported names begin with a capital letter

See: [Exported names](#)

Both are the same.

See: Importing

Every package file has to start with package.

Concurrency

Goroutines

```
func main() {
    // A "channel"

    // Start concurrent routines

    // ...

    // Read 3 results
    // (Since our goroutines are concurrent,
    // the order isn't guaranteed)
    fmt.Println(<-ch, <-ch, <-ch)
}

func push(name string, ch chan string) {
    msg := "Hey, " + name

}
```

Channels are concurrency-safe communication objects, used in goroutines.

Buffered channels

```
ch <- 1
ch <- 2
ch <- 3
// fatal error:
// all goroutines are asleep - deadlock!
```

Buffered channels limit the amount of messages it can keep

See: Buffered channels

Closing channels

Closes a channel

```
ch <- 1
ch <- 2
ch <- 3
```

Iterates across a chan

```
...
}
```

Closed if ok == false

```
v, ok := <- ch
```

See: Range and close

See: [Goroutines](#), [Channels](#)

⌚ Error control

Defer

```
func main() {
    fmt.Println("Working...")
}
```

Defers running a function until the surrounding function returns. The arguments are evaluated until later.

See: [Defer](#), [panic](#) and [recover](#)

Deferring functions

```
func main() {
    fmt.Println("Working...")
}
```

Lambdas are better suited for defer blocks.

⌚ Structs

Defining

Literals

```
v := Vertex{X: 1, Y: 2}
```

```
// Field names can be omitted
v := Vertex{1, 2}
```

Pointers to structs

```
v := &Vertex{1, 2}
v.X = 2
```

Doing v.X is the same as

```
func main() {  
    v := Vertex{1, 2}  
    v.X = 4  
    fmt.Println(v.X, v.Y)  
}
```

```
// Y is implicit  
v := Vertex{X: 1}
```

You can also put field names.

See: [Structs](#)

Methods

Receivers

```
type Vertex struct {  
    X, Y float64  
}
```

```
    return math.Sqrt(v.X * v.X + v.Y * v.Y)  
}
```

```
v := Vertex{1, 2}  
v.Abs()
```

There are no classes, but you can define functions with receivers.

See: [Methods](#)

Mutation

```
    v.X = v.X * f  
    v.Y = v.Y * f  
}
```

```
v := Vertex{6, 12}  
v.Scale(0.5)  
// `v` is updated
```

By defining your receiver as a pointer (*Vertex), you can mutate the original value.

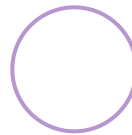
See: [Pointer receivers](#)

References

A tour of Go (tour.golang.org)
Golang wiki (github.com)
Awesome Go (awesome-go.com)
Go by Example (gobyexample.com)
Effective Go (golang.org)
JustForFunc Youtube (youtube.com)
Style Guide (github.com)

► **6 Comments** for this cheatsheet. Write yours!

Search 381+ cheatsheets



Over 381 curated cheatsheets, by developers for developers.

Devhints home

Other C-like cheatsheets

C Preprocessor
cheatsheet

C# 7
cheatsheet

Top cheatsheets

Elixir
cheatsheet

ES2015+
cheatsheet

React.js
cheatsheet

Vimdiff
cheatsheet

Vim
cheatsheet

Vim scripting
cheatsheet