Cheatography

go files Cheat Sheet

by cizixs (cizixs) via cheatography.com/43449/cs/12942/

Basic Operations

create empty file	`newFile, err := os.Create("test.txt")`
truncate a file	`err := os.Truncate("test.txt", 100)`
get file info	`fileInfo, err := os.State("test.txt")`
rename a file	`err := os.Rename(oldPath, newPath)`
delete a file	`err := os.Remove("test.txt")`
open a file for reading	`file, err := os.Open("test.txt")`
open a file	`file, err := os.Open("test.txt", os.O_APPEND, 0600)`
close a file	`err := file.Close()`
change file permision	`err := os.Chmod("test.txt", 0777)`
change file ownership	`err := os.Chown("test.txt", os.Getuid(), os.Getgid())`
change file timestamps	`err := os.Chtimes("test.txt", lastAccessTime, lastModifyTime)`

file open flag

`os.O_RDONLY`	open the file read only
`os.O_WRONLY`	open the file write only
`os.O_RDWR`	open the file read write
`os.O_APPEND`	append data to the file when writing
`os.O_CREATE`	create a new file if none exists
`os.O_EXCL`	used with `O_CREATE`, file must not exist
`os.O_SYNC`	open for synchronous I/O
`O_TRUNC`	if possible, truncate file when opened

When opening file with `os.OpenFile`, flags control how the file behaves.

Hard Link & Symbol Link

create a hard link	`err := os.Link("test.txt", "test_copy.txt")`
create a symbol link	`err := os.Symlink("test.txt", "test_sym.txt")`
get link file info	`fileInfo, err := os.Lstat("test_sym.txt")`
change link file owner	`err := os.Lchown("test_sym.txt", uid, gid)`
read a link	`dest, err := os.ReadLink("link_file.txt")`

A hard link creates a new pointer to the same place. A file will only be deleted from disk after all links are removed. Hard links only work on the same file system. A hard link is what you might consider a 'normal' link.

A symbolic link, or soft link, only reference other files by name. They can point to files on different filesystems. Not all systems support symlinks.

Read and Write

write bytes to file	`n, err := file.Write([]byte("hello, world!\n"))`
write string to file	`n, err := file.WriteString("Hello, world!\n")`
write at offset	`n, err := file.WriteAt([]byte("Hello"), 10)`
read to byte	`n, err := file.Read(byteSlice)`
read exactly n bytes	`n, err := io.ReadFull(file, byteSlice)`
read at least n bytes	`n, err := io.ReadAtLeast(file, byteSlice, minBytes)`
read all bytes of a file	`byteSlice, err := ioutil.ReadAll(file)`
read from offset	`n, err := file.ReadAt(byteSlice, 10)`

Cheatography

go files Cheat Sheet

by cizixs (cizixs) via cheatography.com/43449/cs/12942/

Work with directories

create a directory	`err := os.Mkdir("myDir", 0600)`
recursively create a directory	`err := os.MkdirAll("dir/subdir/myDir", 0600)`
delete a directory recursively	`err := os.RemoveAll("dir/")`
list directory files	`fileInfo, err := ioutil.ReadDir(".")`

Shortcuts

quick read from file	`byteSlice, err := ioutil.ReadFile("test.txt")`
quick write to file	`err := ioutil.WriteFile("test.txt", []byte("Hello"), 0666)`
copy file	`n, err := io.Copy(newFile, originFile)`
write string to file	`io.WriteString(file, "Hello, world")`

Temporary files and directories

create temp dir	`ioutil.TempDir(dir, prefix string) (name string, err error)`
create temp file	`ioutil.TempFile(dir, prefix string) (f *os.File, err error)`

References

Working with Files in Go

golang `os` standard library

golang `ioutil` standard library

golang `iou` standard library

By **cizixs** (cizixs)
cheatography.com/cizixs/
cizixs.com

Published 26th September, 2017.
Last updated 26th September, 2017.

Cheatography

GoLang Cheat Sheet

by tahir24434 via cheatography.com/67694/cs/24941/

Packages

Packages in Go supports modularity, encapsulation, separate compilation, and reuse.

Package declaration at top of every source file

Standalone executables program are in package main

If an entity is declared within a function, it is local to that function.

If declared outside of a function, however, it is visible in all files of the package to which it belongs

The case of the first letter of a name determines its visibility across package boundaries.

Upper case identifier: Exported i.e visible and accessible outside of its own package.

Lower case identifier: private (not accessible from other packages)

Pointers

```
var x int = 11
/*
*int is integerPointer type.
‘p’ will contain the address of an integer variable.
You can also say that p points to an int variable.
*/
var p *int
// Expression &var (address of var) yields a pointer to a variable.
p = &x // will contain address of x
// Expression *p points to the variable whose address p contains. *p is an alias for x.
fmt.Println(*p)
```

Arrays

Arrays and Structs are aggregate type
Arrays are homogeneous
Array is fixed length sequence of zero or more elements of particular type.

```
var a[3] int        // Array of 3 integers
var a[3]int = [3]int{1, 2, 3} // use an array literal to initialize an array with a list of values
a[len(a)-1]         // Print last element
q := [...]int{1, 2, 3} // with ellipsis ... , array length is determined by the number of initializer
```

Naming Convention

Declarations

There are four major kinds of declarations: **var, const, type, func**

var

```
var name type = expression
// Either the type or the =expression part may be omitted, but not both
// If the type is omitted, it is determined by the initializer expression.
If the expression is omitted, the initial value is the zero value for the type, which is 0 for numbers, false for booleans.
var foo int = 42 // declare and init. var name type = expression
var sep string              // implicit initialize
s, sep := "", ""            // Short variable declaration. name := expression
p := new(int) // p, of type *int, points to an unnamed int variable
// new(T) creates unnamed variable of type T, initialize it to the zero value of T and returns its address.
```

const

A constant is an identifier for a fixed value. The value of a variable can vary, but the value of a constant must remain constant.

```
const constant = "This is a constant"
const a float64 = 3.14
```

Function Declaration

A function declaration has a name, a list of parameters, an optional list of results

```
// function with params
func getFullName(firstName string, lastName string) {}
// Multiple params of the same type
func getFullName(firstName, lastName string) {}
// Can return type declaration
func getId() int
// Can return multiple values at once
func person() (int, string) {
  return 23, "vinay"
}
// Can return multiple named results
func person() (age int, name string) {
  age = 23 name = "vinay"
  return
}
var age, name = person()
// Can return function
func person() func() (string,string) {
  area:=func() (string,string) {
    return "street", "city"
  }
return area
}
```

Loops

```
// a name begins with a letter or an underscore and may have any number of additional letters, digits, and underscores
type playerScore struct // Use CamelCase
const MaxTime int
var fileClosed bool // Use the complete words in larger scopes
var arg []string // Use fewer letters in smaller scopes
var localAPI string // Use All caps for acronym
```

```
// There only for, no while, no until
for i := 1; i < len(os.Args); i++ {}   // initialization; condition; post {}
for condition {}       // While loop
// 'range' produces a pair of values: the index and the value of the element at that index. '_' is called blank identifier.
for _, arg := range os.Args[1:]
```


By **tahir24434**
cheatography.com/tahir24434/

Published 10th November, 2020.
Last updated 6th March, 2021.

Cheatography

Golang Naming Conventions Cheat Sheet

by [deleted] via cheatography.com/70653/cs/17920/

Why naming is important?

Critical for Readability = Maintainability

The naming is important because it is very critical for readability and if you can't read the code, you can't properly maintain it.

Imagine a book that you don't understand, and someone comes to you and asks you to fix the typos in it.

Can you really do it, without understanding it?

There are only two hard things in Computer Science: cache invalidation and naming things.
- Phil Karlton

Non-Idiomatic

```
func Read(buffer *Buffer, inBuffer []byte) (size
int, err error) {
        if buffer.empty() {
                buffer.Reset()
        }
        size = copy(
                inBuffer,
                buffer.buffer[buffer.offset:])
        buffer.offset += size
        return size, nil
}
```

This code is unnecessarily verbose. Everything has been declared in English words, which generally should be avoided. From the readability and maintainability perspective, this code is not good.

Idiomatic

```
func Read(b *Buffer, p []byte) (n int, err error) {
        if b.empty() {
                b.Reset()
        }
        n = copy(p, b.buf[b.off:])
        b.off += n
        return n, nil
}
```

This code is very concise and idiomatic and it's easy to understand and maintain.

References

Abbreviation in Go

golang `bytes` standard library

Inanc Gumus - Learn Go Programming

Use the first few letters of the words

```
var fv string // flag value
```

Use fewer letters in smaller scopes

```
var bytesRead int // number of bytes read ✖
var n int // number of bytes read ✔
```

Use the complete words in larger scopes

```
package file
var fileClosed bool
```

Imagine that this variable is declared in the package block of the `file` package.

It's a package level variable and therefore it's in a larger scope. Don't use abbreviations there and don't mix caps in the name. `file` starts with a lowercase letter.

Cheatography

Golang Naming Conventions Cheat Sheet

by [deleted] via cheatography.com/70653/cs/17920/

Use mixedCaps like this

```
type playerScore struct
```

Use all caps for acronyms

```
var localApi string ✖
var localAPI string ✔
```

Do not stutter

```
player.PlayerScore ✖
player.Score ✔
```

Do not use under_scores oR LIKE_THIS

```
const MAX_TIME int ✖
const MaxTime int ✔
const N int ✔
```

By **[deleted]**
cheatography.com/deleted-70653/

Published 15th November, 2018.
Last updated 15th November, 2018.

Abbreviation - Rules

Sound/Spelling

Abbreviations should be **pronounceable**.

Abbreviations should have **at least one vowel**.

Abbreviations should not split up **plosive/liquid** combinations but as **plosive/plosive**, for example, the **ct** in **dictionary** or **pt** in **caption**.

Abbreviations should not have more than **three consonants** in a row and should usually **end in a consonant**, unless the vowel is needed for discrimination, for example, **alg** and **algo**.

All of the letters in the abbreviation should be present in the long form and in the same order, and need not appear in sequence in the long form, for example, **recv** and **receive**.

Exceptions/Limitations

There are a few exceptions to the above rules for common, well-established forms.

ct and **pt** can be used for **ction** and **ption** if the abbreviation would be too short otherwise, for example, **act** and **opt**.

There are also other types of prefixing, for example, the three--letter prefixes used to distinguish field names in the same database table.

Examples would include **cusID** for **customer ID** and **ordID** for **order ID**.

Those prefixes don't need to follow the same rules.

Abbreviation - Rules (cont)

Length/Meaning and Interpretation

An abbreviation should be less than or equal to half the length of the original form.

Abbreviations should be **at least three letters long**.

Abbreviations should **not be whole words** that mean something else.

Abbreviations should not just consist of the prefix of a word, for example, **sym** for **symbol** or **syl** for **syllable**.

Abbreviations **shouldn't be ambiguous**. However, if the names are different that **no confusion** can result, they are **OK**.

Cheatography

Golang Naming Conventions Cheat Sheet

by [deleted] via cheatography.com/70653/cs/17920/

Abbreviation	
`var a int`	// **a**rray
`var arg []string`	// **arg**ument
`var b []byte`	// **b**uffer
`var b byte`	// **b**yte
`var bs bytes`	// **bs**
`var buf []byte`	// **buf**fer
`var c int`	// **c**apacity
`var c int`	// **c**haracter
`var dst int`	// **dst**ination
`var err error`	// **err**or value
`var fv string`	// **f**lag **v**alue
`var i int`	// **i**ndex
`var l int`	// **l**ength
`var m int`	// another nu**m**ber
`var msg string`	// **msg**
`var n int`	// **n**umber or number of
`var num int`	// **num**ber
`var off int`	// **off**set
`var op int`	// **op**eration
`var parsed bool`	// **parsed** ok?
`var pkg string`	// **pkg**
`var pos int`	// **pos**ition
`var r rune`	// **r**une
`var r io.Reader`	// **r**eader
`var s string`	// **s**tring
`var seen bool`	// has **seen**?
`var sep string`	// **sep**arator

Abbreviation (cont)	
`var src int`	// **src**
`var str string`	// **str**ing
`var v string`	// **v**alue
`var val string`	// **val**ue
`var w io.Writer`	// **w**riter
...the list goes on and on...	

By **[deleted]**
cheatography.com/deleted-70653/

Published 15th November, 2018.
Last updated 15th November, 2018.

Cheatography

GoLang fmt Printing Cheat Sheet

by Guillermo Pascual (gpascual) via cheatography.com/32181/cs/9868/

String and slice of bytes

%s	the uninterpreted bytes of the string or slice
%q	a double-quoted string safely escaped with Go syntax
%x	base 16, lower-case, two characters per byte
%X	base 16, upper-case, two characters per byte

General

%v	The value in a default format. When printing structs, the plus flag (%+v) adds field names.
%#v	a Go-syntax representation of the value
%T	a Go-syntax representation of the type of the value
%%	a literal percent sign; consumes no value

The default format for %v

bool:	%t
int, int8 etc.:	%d
uint, uint8 etc.:	%d, %x if printed with %#v
float32, complex64, etc:	%g
string:	%s
chan:	%p
pointer:	%p

Other flags

+	always print a sign for numeric values; guarantee ASCII-only output for %q (%+q).
-	pad with spaces on the right rather than the left (left-justify the field).
#	alternate format: add leading 0 for octal (%#o), 0x for hex (%#x); 0X for hex (%#X); suppress 0x for %p (%#p); for %q, print a raw (backquoted) string if strconv.CanBackquote returns true;
' ' (space)	leave a space for elided sign in numbers (% d); put spaces between bytes printing strings or slices in hex (% x, % X).
0	pad with leading zeros rather than spaces; for numbers, this moves the padding after the sign.

Boolean

%t	the word true or false

Integer

%b	base 2
%c	the character represented by the corresponding Unicode code point
%d	base 10
%o	base 8
%q	a single-quoted character literal safely escaped with Go syntax
%x	base 16, with lower-case letters for a-f
%X	base 16, with upper-case letters for A-F
%U	Unicode format: U+1234; same as "U+%04X"

Floating-point and complex constituents

%b	decimalless scientific notation with exponent a power of two, in the manner of strconv.FormatFloat with the 'b' format, e.g. -123456p-78
%e	scientific notation, e.g. -1.234456e+78
%E	scientific notation, e.g. -1.234456E+78
%f	decimal point but no exponent, e.g. 123.456
%F	synonym for %f
%g	%e for large exponents, %f otherwise
%G	%E for large exponents, %F otherwise

Floating-point Precision

%f	default width, default precision
%9f	width 9, default precision
%.2f	default width, precision 2
%9.2f	width 9, precision 2
%9.f	width 9, precision 0

Pointer

%p	base 16 notation, with leading 0x

By **Guillermo Pascual** (gpascual)
cheatography.com/gpascual/

Published 17th November, 2016.
Last updated 17th November, 2016.

Cheatography

GoLang Cheat Sheet

by Andan H M (andanhm) via cheatography.com/69306/cs/17474/

Go

Go (also referred to as GoLang) is an open source and lower level programming language designed and created at Google in 2009 by Robert Griesemer, Rob Pike and Ken Thompson, to enable users to easily write simple, reliable, and highly efficient computer programs

Besides its better-known aspects such as built-in concurrency and garbage collection

Go is a statically typed language, it is anti functional programming and anti OOP, as far as the designers concerned.

https://golang.org/

Feature

Language is very concise, simple and safe.

Compilation time is very fast.

Patterns which adapt to the surrounding environment similar to dynamic languages.

Inbuilt concurrency such as lightweight processes channels and select statements.

Supports the interfaces and the embedded types.

https://golang.org/doc/faq

Lack of essential features

No ternary operator ?:

No generic types

No exceptions

No assertions

No overloading of methods and operators

~~GOPATH is a mess~~

~~Package dependence manage tool~~

https://github.com/ksimka/go-is-not-good

Companies Using Golang

Google for “dozens of systems”

Docker a set of tools for deploying linux containers

Openshift a cloud computing platform as a service by Red Hat

Dropbox migrated few of their critical components from Python to Go

Netflix for two portions of their server architecture

Soundcloud for “dozens of systems”

ThoughtWorks some tools and applications around continuous delivery and instant messages (CoyIM)

Uber for handling high volumes of geofence-based queries.

BookMyShow for handling high volume of traffic, rapidly growing customer, to adapt new business solution and (cloud solution) distribution tools

https://www.qwentic.com/blog/companies-using-golang

Install

OSX brew install go

Run the command below to view your Go version:

```
go version
```

https://golang.org/doc/install

Directory layout

GOPATH=/home/user/go

```
/home/user/go/
  src/
    hello/
      main.go (package main)
  bin/
     hello (installed command)
  pkg/
    linux_amd64/ (installed package object)
    github.com/ (3rd party dependencies)
```

By **Andan H M** (andanhm)
cheatography.com/andanhm/
andanhm.me

Published 21st October, 2018.
Last updated 22nd October, 2018.

Cheatography

GoLang Cheat Sheet

by Andan H M (andanhm) via cheatography.com/69306/cs/17474/

Hello Word

```
package main
import "fmt"
func main() {
    fmt.Println("Hello, World!!")
}
```

Create a file named main.go in the directory src/hello inside your workspace/go path

go env Default go system environment

https://tour.golang.org

Running

```
$ cd $HOME/go/src/hello
$ go run main.go
Hello, World!!
$ go build
$ ./hello
Hello, World!!
```

Package

Package declaration at top of every source file

Executables are should be in package main

Upper case identifier: public (accessible from other packages)

Lower case identifier: private (not accessible from other packages)

Built-in Types

```
bool
string
int int8 int16 int32 int64
uint uint8 uint16 uint32 uint64 uintptr
byte // alias for uint8
rune // alias for int32 ~= a character (Unicode code point)
```

Built-in Types (cont)

```
float32 float64
complex64 complex128
```

Packages and Modules

Packages

> Go packages are folders that contain one more go files.

Modules

> A modules (starting with vgo and go 1.11) is a versioned collection of packages.

```
go get github.com/andanhm/go-prettytimee
go mod init github.com/andanhm/go-prettytime
```

Variable & Function Declarations

```
const country = "india"
// declaration without initialization
var age int
// declaration with initialization
var age int = 23
// declare and init multiple at once
var age, pincode int = 23, 577002
// type omitted, will be inferred
var age = 23
// simple function
func person() {
 // shorthand, only within func bodies
 // type is always implicit
 age := 23
}
// Can have function with params
```

Variable & Function Declarations (cont)

```
func person(firstName string, lastName string) {}
// Can have multiple params of the same type
func person(firstName, lastName string) {}
// Can return type declaration
func person() int {
    return 23
}
// Can return multiple values at once
func person() (int, string) {
    return 23, "vinay"
}
var age, name = person()
// Can return multiple named results
func person() (age int, name string) {
    age = 23
    name = "vinay"
    return
}
var age, name = person()
// Can return function
func person() func() (string,string) {
     area:=func() (string,string) {
          return "street", "city"
      }
      return area
}
```

If statement

```
if age < 18 {
      return errors.New("not allowed to enter")
}
// Conditional statement
if err := Request("google.com"); err != nil {
    return err
}
// Type assertion inside
var age interface{}
age = 23
if val, ok := age.(int); ok {
    fmt.Println(val)
}
```

Loop statement

```
for i := 1; i < 3; i++ {
}
// while loop syntax
for i < 3 {
}
// Can omit semicolons if there is only a condition
for i < 10 {
}
// while (true) like syntax
for {
}
```

Go don't have **while until**

Switch statement

```
// switch statement
switch runtime.GOOS {
  case "darwin": {
// cases break automatically
   }
   case "linux": {
   }
   default:
}
// can have an assignment statement before the switch
statement
switch os := runtime.GOOS; os {
case "darwin":
default:
}
// comparisons in switch cases
os := runtime.GOOS
switch {
case os == "darwin":
default:
}
// cases can be presented in comma-separated lists
switch os {
case "darwin", "linux":
}
```

Arrays, Slices

```
var a [3]int // declare an int array with length 3.
var a = [3]int {1, 2, 3} // declare and initialize a
slice
a := [...]int{1, 2} // elipsis -> Compiler figures out
array length
a[0] = 1 // set elements
i := a[0] // read elements
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([]int, 5, 5) // first arg length, second
capacity
a = make([]int, 5) // capacity is optional
// loop over an array/ slice / struct
for index, element := range a {

}
```

Maps & Struct

Maps

Maps are Go's built-in associative data type (`hashes` or `dicts`)

Struct

Structs are the way to create concrete user-defined types in Go. Struct types are declared by composing a fixed set of unique fields.

Example

```
type Address struct {
  Street string
  City string
}
type Employee struct {
  Name string
  Age int
  Address Address
}
// Can declare methods on structs.
func (emp Employee) Display() string {
  // accessing member
  name:=emp.Name
  return fmt.Sprintf("Name %s",name)
}
// Initialize the map with the type
// map key is city value employees working
bookmyshow := make(map[string][]Employee)
// Create new/updates the key value pair
bookmyshow["Pune"] = []Employee{}
bookmyshow["Bangalore"] = []Employee{
   Employee{
    Name: "Andan H M",
    Age: 23,
    Address: Address{
     Street: "KB Extension",
     City: "Davanagere",
   },
```

Example (cont)

```
 },
},
// Determains the the length of the map
_ = len(bookmyshow)
// read the item from the map
employees := bookmyshow["Bangalore"]
// loop over an array, slice, struct array
for index, element := range employees {
  // read the element from the struct
  fmt.Println(index, element.Display())
}
// Delete the key from the map
delete(bookmyshow, "Pune")
```

Interfaces

Interface type that specifies zero methods is known as the *empty interface*

```
var i interface{}
i = 42
// Reflection: type switch specify types
switch v := i.(type) {
 case int:
   fmt.Printf("(%v, %T)\n", i, i)
 case string:
  fmt.Printf("(%v, %T)\n", i, i)
 default:
  fmt.Printf("Unknow type %T!\n", v)
}
```

Interfaces are named collections of method signatures.

```
type error interface {
```

Interfaces (cont)

```
    Error() string
}
```

Accept interfaces, return structs

Error

The error type is an interface type.

`error` variable represents description of the error string

```
errors.New('user not found')
fmt.Errorf("%s user not found", "foo")
```

https://blog.golang.org/error-handling-and-go

HTTP Handler

```
package main
import (
 "io"
 "net/http"
)
func health(w http.ResponseWriter, r *http.Request) {
  w.WriteHeader(http.StatusOK)
  io.WriteString(w, "Ok")
}
func main() {
 http.HandleFunc("/health", health)
 http.ListenAndServe(":8080", nil)
}
```

A mini-toolkit/micro-framework to build web apps; with handler chaining, middleware and context injection, with standard library compliant HTTP handlers(i.e. http.HandlerFunc).

https://github.com/bnkamalesh/webgo

Unit Test

Go has a built-in testing command called go test and a package testing which combine to give a minimal but complete testing experience.

Standard tool-chain also includes benchmarking and code coverage

https://github.com/andanhm/gounittest

Concurrency

Goroutines

> Goroutines are lightweight threads managed by Go

Channels

> Channels are a typed conduit through which you can send and receive values with the channel operator (`<-`)

Example

```
package main
import "fmt"
func main() {
 n := 2

 // "make" the channel, which can be used
 // to move the int datatype
 out := make(chan int)
 // run this function as a goroutine
 // the channel that we made is also provided
 go Square(n, out)
 // Any output is received on this channel
 // print it to the console and proceed
 fmt.Println(<-out)
}
func Square(n int, out chan<- int) {
 result := n * n
```

Example (cont)

```
 //pipes the result into it
 out <- result
}
```

select statement lets a goroutine wait on multiple communication operations.

sync go build-in package provides basic synchronization primitives such as mutual exclusion locks.
https://golang.org/pkg/sync/

Defer, Panic, and Recover

Defer

> A defer statement pushes a function call onto a Last In First Out order list. The list of saved calls is executed after the surrounding function returns

Panic

> Panic is a built-in function that stops the ordinary flow of control and begins panicking.

Recover

> Recover is a built-in function that regains control of a panicking goroutine

```
func main() {
 defer func() {
 if r := recover(); r != nil {
   fmt.Println("Recovered", r)
  }
 }()
 panic("make panic")
}
```

Encoding

encoding is a built-in package defines interfaces shared by other packages that convert data to and from byte-level and textual representations

Go offers built-in support for encoding/gob, encoding/json, and encoding/xml

https://golang.org/pkg/encoding/

Example

```
package main
import (
 "encoding/json"
 "encoding/xml"
 "fmt"
)
type Employee struct {
 Name string `json:"name" xml:"name"`
 Age int `json:"age" xml:"age"`
}
func main() {
 emp := Employee{
  Name: "andan.h",
  Age: 27,
 }
 // Marshal: refers to the process of converting
 // the data or the objects into a byte-stream
 jsonData, _ := json.Marshal(emp)
 fmt.Println(string(jsonData))
 xmlData, _ := xml.Marshal(emp)
 fmt.Println(string(xmlData))
 // Unmarshal: refers to the reverse process of
 // converting the byte-stream back to data or object
 json.Unmarshal(jsonData, &emp)
 fmt.Println(emp)
}
```

Tool
https://godoc.org/golang.org/x/tools
https://dominik.honnef.co/posts/2014/12/an_incomplete_list_of_go_tools/
https://github.com/campoy/go-tooling-workshop

By **Andan H M** (andanhm)
cheatography.com/andanhm/
andanhm.me

Published 21st October, 2018.
Last updated 22nd October, 2018.