Go Programming (Golang) Cheat Sheet

We created this Golang Cheat Sheet initially for students of our Go Programming (Golang): The Complete Developer's Guide. But we're now sharing it with any and all Developers that want to learn and remember some of the key functions and concepts of Go, and have a quick reference guide to the basics of Golang.

If you've stumbled across this cheatsheet and are just starting to learn Golang, you've made a great choice! It was created by Google to solve Google-sized problems. This has made it very popular with other companies solving massive scaling challenges and is a great laguage to learn if you're interested in becoming a DevOps Engineer or a Fullstack Developer.

However, if you're stuck in an endless cycle of YouTube tutorials and want to start building real world projects, become a professional developer, have fun and actually get hired, then come join the Zero To Mastery Academy. You'll learn Golang from actual industry professionals alongside thousands of students in our private Discord community.

You'll not only learn to become a top 10% Go Developer by learning advanced topics most courses don't cover. But you'll also build Golang projects, including a Pixl Art cross-platform desktop app that you can add to your portfolio and wow employers!

Just want the cheatsheet? No problem! Please enjoy and if you'd like to submit any suggestions, feel free to email us at support@zerotomastery.io

Contents

Comparison
Logical
Other

Data Types

Data Types
Signed Integers
Unsigned Integers
Floating Point Numbers

Declarations

Variables
Type Aliases
Constants
iota Enumeration Pattern

Functions

Functions
Function Literals / Closures
Variadics

fmt

fmt

Escape Sequences

Escape Sequences

Control Structures

- If
- Switch
- Loops

Arrays

- Arrays
- Slices

Maps

- Maps

Structures

- Structures
- Anonymous Structures

Pointers

- Pointers

Receiver Functions

- Receiver Functions

Interfaces

- Interfaces

Type Embedding

- Type Embedding
- Interfaces
- Structs

Concurrency

- Concurrency
- Defer
- Goroutines
- WaitGroups
- Mutex
- Channels

Errors

Errors

Testing

Testing

CLI

CLI

Modules

Modules

Packages

Packages

Operators

Mathematical

Operator	Description
`+`	add
`-`	subtract
`*`	multiply
`/`	divide
`%`	remainder / modulo
`+=`	add then assign
`-=`	subtract then assign
`*=`	multiply then assign
`/=`	divide then assign
`%=`	remainder / modulo
`(variable)++`	increment
`(variable)--`	decrement

Bitwise

Operator	Description
`&`	bitwise `and`
`|`	bitwise `or`
`^`	bitwise `xor`
`&=`	bitwise `and` then assign

Operator	Description
`|=`	bitwise `or` then assign
`^=`	bitwise `xor` then assign

Comparison

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

Logical

Operator	Description
`&&`	and
`||`	or
`!`	not

Other

Operator	Description
`<<`	left shift
`>>`	right shift

Data Types

Raw strings and raw runes will be displayed as-is (no escape sequences).

Type	Default	Notes
string	""	Create with "" (double quotes) or `` (backticks) for raw string; contains any number of Unicode code points
rune	0	Create with '' (single quotes) or `` (backticks) for raw rune; contains a single Unicode code point
bool	false	true / false

Signed Integers

Type	Default	Range
int8	0	-128..127
int16	0	-32768..32767
int	0	-2147483648..2147483647
int32	0	-2147483648..2147483647
rune	0	-2147483648..2147483647
int64	0	-9223372036854775808..9223372036854775807

Unsigned Integers

Type	Default	Range
uint8 byte	0	0..255
uint16	0	0.65535
uint uint32	0	0..4294967295
uint64	0	0..18446744073709551615
uintptr	0	<pointer size on target architecture>

Floating Point Numbers

| Type |Default | Notes | |———————-|————-|———————————————————————-| | float32 | 0 | 32-bit floating point | | float64 | 0 | 64-bit floating point | | complex64 | 0 | 32-bit floating point real & imaginary | | complex128 | 0 | 64-bit floating point real & imaginary |

Declarations

Variables

Naming convention is `camelCase`. Variable names can only be declared once per scope (function body, package, etc.).

```
var myVariable int  // uninitialized variable of type int
var b int = 3       // variable of type int set to 3
var c = 3           // variable set to 3. type inferred (i

var d, e, f = 1, 2, "f" // create 3 variables at once. typ

// block declaration
var (
    g int = 1
    h int = 2
    i     = "c"     // type inferred (string)
)

// Variables starting with a capital letter are public
// and can be accessed outside the package
var MyPublicVariable = 4
```

Shorthand notation creates and then assigns in a single statement. Shorthand can only be used within function bodies:

```
j := 3                 // create & assign; type inferred (in
k, m := 1, "sample"    // create & assign multiple; types in
n, _ := 1, 2           // ignore values with an underscore (
```

"*Comma, ok*" idiom allows re-use of last variable in compound create & assign:

```
a, ok := 1, 2   // create `a` and `ok`
b, ok := 3, 4   // create `b` and re-assign `ok`
a, ok := 5, 6   // ERROR: `a` has already been created; re
```

Type Aliases

Type aliases can help clarify intent. They exist as names only, and are equivalent to the aliased type.

```
type Distance int         // distance now refers to int
type Miles Distance       // miles now refers to int via Di

type Calculate = func(a, b int) int  // closure
```

Constants

Naming convention is `PascalCase`. Constant names can only be declared once per scope (function body, package, etc.).

```
const MyConstantValue       = 30     // type inferred (int
const MyConstantName string = "foo"  // explicit type

// block declaration
const (
    A = 0
    B = 1
    C = 2
)

// Constants starting with a capital letter are public
// and can be accessed outside the package
const SomeConstant = 10

// iota
const (
    A = iota  // 0
    B         // 1
    C         // 2
    _         // 3 (skipped)
    E         // 4
)
```

```
// start iota at specific value
const (
    A = iota + 3  // 3
    B             // 4
    C             // 5
)
```

iota Enumerations Pattern

```go type Direction byte const ( North Direction = iota // 0 East // 1 South // 2 West // 3 )

// `String` function used whenever type `Direction` is printed func (d Direction) String() string { // Array of string, indexed based on the constant value of Direction. // Cannot change order of constants with this implementation. return [ ]string{"North", "East", "South", "West"}[ d ]

func (d Direction) String() string { // resistant to changes in order of constants switch d { case North: return "North" case East: return "East" case South: return "South" case West: return "West" default: return "other direction" } }

```
<h2 id="functions">Functions</h2>

All function calls in Go are pass-by-value, meaning a copy

```go
// entry point to a Go program
func main() {}

func name(param1 int, param2 string) {}
//               ^ data type after parameter names

// return type of int
func name() int {
    return 1
}

// parameters of the same type only need 1 annotation
func sum(lhs, rhs int) int {
    return lhs + rhs
}

// multiple return values
func multiple() (string, string) {
    return "a", "b"
}
// call function
var a, b = multiple()

// return values can be set directly in function if named
func multipleNamed() (a string, b string) {
    a = "eh"
    b = "bee"
    return
}
// call function
```

```
var a, b = multipleNamed()

// Functions starting with a capital letter are public
// and can be accessed outside the package
func MyPublicFunc() {}
```

Function Literals

```
// function literals can be created inline
world := func() string {
    return "world"
}
// call function by using name assigned above
fmt.Printf("Hello, %s\n", world())

sample := 5
// closure
test := func() {
    // capture `sample` variable
    fmt.Println(sample)
}
test()  // output: 5
```

```
// closure as function parameter
func math(op func(lhs, rhs int) int, lhs, rhs int) int {
    //        |__closure signature_|
    // call `op` closure and return the result
    return op(lhs, rhs)
}

// with type alias
type MathOperation func(lhs, rhs int) int
func math(op MathOperation, lhs, rhs int) int {
    return op(lhs, rhs)
}
// create closure
add := func(lhs, rhs int) int {
        return lhs + rhs
}
// call function with closure
math(add, 2, 2)  // 4
```

```
// return closure from function
//
//                      |____return type____|
func minus(amount int) func(operand int) int {
    //                  |_closure signature_|
    //
    //     |_closure signature_|
    return func(operand int) int {
      // capture operand
      return operand - amount
    }
}
func main() {
    minusFive := minus(5)  // closure returned from functi
```

```
    minusFive(20)          // 15
}
```

Variadics

Variadics allow a function to accept any number of parameters.

```
// `nums` is treated like a slice of int
func sum(nums ...int) int {
    sum := 0
    // iterate through each argument to the function
    for _, n := range nums {
        sum += n
    }
    return sum
}

a := []int{1, 2, 3}
b := []int{4, 5, 6}

all := append(a, b...)     // slices can be expanded with
answer := sum(all...)      // each element will be an argu

// same as above
answer = sum(1, 2, 3, 4, 5, 6)    // many arguments
```

fmt

The `fmt` package is used to format strings. *Verbs* set what type of formatting is used for specific data types. See the docs for a full list of verbs.

Verb	Description
`%v`	default
`%+v`	for structs: field names & values; default otherwise
`%#v`	for structs: type name, field names, values; default otherwise
`%t`	"true" or "false" (boolean only)
`%c`	character representation of Unicode code point
`%b`	base 2
`%d`	base 10
`%o`	base 8
`%O`	base 8 w/ 0o prefix
`%x`	base 16 hexadecimal; lowercase a-f
`%X`	base 16 hexadecimal; uppercase A-F
`%U`	Unicode (U+0000)
`%e`	scientific notation

Verb	Description
`%p`	pointer address

```
fmt.Printf("custom format with %v, no newline at end\n", "
fmt.Print("simple")
fmt.Println("newline at end")

s1 := fmt.Sprintf("printf into a string")
s2 := fmt.Sprint("print into a string")
s3 := fmt.Sprintln("println into a string")

writer := bytes.NewBufferString("")
fmt.Fprintf(writer, "printf into a Writer")
fmt.Fprint(writer, "print into a Writer")
fmt.Fprintln(writer, "println into a Writer")
```

Escape Sequences

Escape sequences allow input of special or reserved characters within a string or rune.

Sequence	Result
`\\`	backslash: `\`
`\'`	single quote: `'`
`\"`	double quote: `"`
`\n`	newline
`\t`	horizontal tab
`\u`	Unicode (2 byte); example: `\u0041`
`\U`	Unicode (4 byte); example: `\uf09f9880`
`\0`	octal digit; example: `\077`
`\x`	hex byte; example: `\x8F`
`\v`	vertical tab
`\b`	backspace
`\a`	alert/bell
`\f`	form feed

Control Structures

If

```
if condition {
    // execute when true
} else {
```

```
    // execute when false
}

if condition == 1 {
    // ...
} else if condition == 2 {
    // ...
} else {
    // ...
}

// statement initialization: create variable and perform l
if i := someFunc(); i < 10 {
    // do something with i
} else {
    // do something else with i
}
```

Switch

`switch` can be used in place of long `if..else` chains. Cases are evaluated from top to bottom and always stop executing once a case is matched (unless the `fallthrough` keyword is provided).

```
x := 3
switch x {
case 1:
    fmt.Println("1")
case 2:
    fmt.Println("2")
case 3:
    fmt.Println("3")
// handle all other cases
default:
    fmt.Println("other:", x)
}

// switch works on strings as well
url := "example.com"
switch url {
case "example.com":
    fmt.Println("test")
case "google.com":
    fmt.Println("live")
default:
    fmt.Println("dev")
}

// Variables can be assigned and then `switched` upon.
// Cases can also contain conditionals.
switch result := calculate(5); {
case result > 10:
    fmt.Println(">10")
case result == 6:
    fmt.Println("==6")
case result < 10:
    fmt.Println("<10")
}

```

```
// case lists allow multiple values to trigger a case
switch x {
case 1, 2, 3:
    // ...
case 10, 20, 30:
    // ...
}

// fallthrough will execute the next case in the list
letter := 'a'
switch letter {
case ' ':
case 'a', 'e', 'i', 'o', 'u':
    fmt.Println("A vowel")
    fallthrough
case 'A', 'E', 'I', 'O', 'U':
    fmt.Println("Vowels are great")
default:
    fmt.Println("It's something else")
}
// output:
//    A vowel
//    Vowels are great
```

Loops

```
// C-style loop
// create variable i and as long as i < 10, increment i by
for i := 0; i < 10; i++ {
}

// while loop
i := 0
for i < 10 {
    i++
}

// infinite loop
for {
    if somethingHappened {
        // exit loop
        break
    } else if nothingHappened {
        // jump straight to next iteration
        continue
    }
}

// loop labels allow jumping to specific loops
outer:
    for r := 0; r < 5; r++ {
    inner:
        for c := 0; c < 5; c++ {
            if c%2 == 0 {
                // advance to the next iteration of `inner
                continue inner
            } else {
                // break from the `outer` loop
                break outer
            }
```

```
        }
    }
```

Arrays

Arrays in Go are fixed-size and set to default values for unspecified elements.

```go
var myArray [3]int   // create a 3 element array of int
myArray[0] = 1       // assign 1 to element at index 0
myArray[1] = 2       // assign 2 to element at index 1
myArray[2] = 3       // assign 3 to element at index 2

myArray := [3]int{1, 2, 3}   // create a 3 element array w
myArray := [...]int{1, 2, 3} // same as above; compiler fi
myArray := [4]int{1, 2, 3}   // create a 4 element array w

// iterate through an array by measuring length using buil
for i := 0; i < len(myArray); i++ {
    n := myArray[i]
    //...
}

// iterate using range
myArray := [...]int{1, 2, 3}
// `range` keyword iterates through collections
for index, element := range myArray {
    // `index` is the current index of the array
    // `element` is the corresponding data at `index`
}

// ignore index if not needed
for _, el := range myArray {
    // ...
}
```

Slices

```go var slice [ ]int // empty slice var slice = [ ]int{1, 2, 3} // create slice & underlying array mySlice := [ ]int{1, 2, 3} // create a slice & an underlying array (shorthand) item1 := mySlice[ 0 ] // access item at index 0 via slice

// 4 element array nums := [ ... ]int{1, 2, 3, 4} // 0 1 2 3 <- index // make a slice s1 := nums[ : ] // [ 1, 2, 3, 4 ] all s2 := nums[1: ] // [ 2, 3, 4 ] index 1 until end s3 := s2[1: ] // [ 3, 4 ] index 1 until end s4 := nums[ :2] // [ 1, 2 ] start until index 2 (exclusive) s5 := nums[1:3] // [ 2, 3 ] index 1 (inclusive) until index 3 (exclusive)

// append items to slice nums := [ ... ]int{1, 2, 3} nums = append(nums, 4, 5, 6) // nums == [ 1, 2, 3, 4, 5, 6 ]
```

// append a slice to another slice nums := [...]int{1, 2, 3} moreNums := [ ]int{4, 5, 6} nums = append(nums, moreNums...) // nums == [1, 2, 3, 4, 5, 6]

// preallocate a slice with specific number of elements slice := make([ ]int, 10) // int slice with 10 elements set to default (0)

// int slice; 5 elements set to default (0) // capacity of 10 elements before reallocation occurs slice := make([ ]int, 5, 10)

// multidimensional slices board := [ ][ ]string{ [ ]string{"", "", ""}, *// type annotation optional* {"", "", ""}, {"", "", "_"}, } board[0][0] = "X" board[2][2] = "O" board[1][2] = "X" board[1][0] = "O" board[0][2] = "X"

// iterate using range mySlice := [ ]int{1, 2, 3} // `range` keyword iterates through collections for index, element := range mySlice { // `index` is the current index of the array // `element` is the corresponding data at `index` }

// ignore index if not needed for _, el := range mySlice { // ... }

```
<h2 id="maps">Maps</h2>

Maps are key/value pairs. Equivalent to Dictionary, HashMa

```go
// empty map having key/value pairs of string/int
myMap := make(map[string]int)

// pre-populate with initial data
myMap := map[string]int{
    "item 1": 1,
    "item 2": 2,
    "item 3": 3,
}

myMap["item 4"] = 4;      // insert
two := myMap["item 2"]    // read
empty := myMap["item"]    // nonexistent item will return
delete(myMap, "item 1")   // remove from map

// determine if item exists
three, found := myMap["item 3"]
// `found` is a boolean
if !found {
    fmt.Println("item 3 not found!")
    return
}
fmt.Println(three)    // ok to use `three` here

// use `range` for iteration
myMap := map[string]int{
    "item 1": 1,
    "item 2": 2,
    "item 3": 3,
}
```
```

```
for key, value := range myMap {
    // ...
}

// ignore values
for key, _ := range myMap {
    // ...
}

// ignore keys
for _, value := range myMap {
    // ...
}
```

Structures

Structures are made of fields, which contain some data. Similar to a Class in other languages.

```
// definition
type Sample struct {
    field string  // type annotations required
    a, b  int     // same rules as function signatures
}

// instantiation
data := Sample{"data", 1, 2}  // all fields required when
data := Sample{}              // all default values

// omitted fields will have defaults
data := Sample{
    field: "data",
    b: 1,
    // `a` will be 0 (default)
}

f := data.a   // access field `a`
data.b = 2    // set field `b` to 2
```

Anonymous Structures

```
// anonymous structures can be created in functions
var sample struct {
    field string
    a, b int
}
sample.field = "test"

// shorthand (must provide values)
sample := struct {
    field string
    a, b int
}{
    "test",
```

```
    1, 2,
}
```

Pointers

Pointers are memory addresses stored in variables. They *point* to other variables, and can be *dereferenced* to access the data at the address they point to.

```
//
//   5    | value
//  ---
//  0x3   | memory address of value
//
num := 5            // value                      (5)
var numPtr *int     // pointer to int             (nil)
numPtr = &num       // use `&` to create pointer  (0x3)
five := *numPtr     // use `*` to access value    (5)
fmt.Println(five)  // output: 5

// function that takes a pointer to int
func increment(x *int) {
    // add 1 to the value
    *x += 1
}
i := 1
increment(&i)   // create pointer to `i`
fmt.Println(i)  // output: 2

var ptr *int    // default value of pointer is `nil`
```

Receiver Functions

Receiver functions allow functionality to be tied to structures. Use either all pointer receivers or all value receivers on a single structure to avoid compilation errors.

```
type Coordinate struct {
    X, Y int
}

// regular function
func shiftBy(x, y int, coord *Coordinate) {
    coord.X += x
    coord.Y += y
}

// receiver function
// use pointer to modify structure
func (coord *Coordinate) shiftBy(x, y int) {
```

```
    coord.X += x
    coord.Y += y
}
coord := Coordinate{5, 5}
shiftBy(1, 1, &coord)   // coord{6, 6}
coord.shiftBy(1, 1)     // coord{7, 7}

// receiver function
// original structure unmodified
func (coord Coordinate) shiftByValue(x, y int) Coordinate
    coord.X += x
    coord.Y += y
    return coord
}
coord := Coordinate{5, 5}
updated := coord.shiftByValue(1, 1)
fmt.Println(coord)      // output: {5, 5}
fmt.Println(updated)    // output: {6, 6}
```

Interfaces

Interfaces allow functionality to be specified for arbitrary data types.

```
// declare an interface
type MyInterface interface {
    MyFunc()
}

type MyType int

// Interfaces are implemented implicitly when
// all interface functions are implemented.
// Prefer pointer receivers over value receivers.
func (m *MyType) MyFunc() {}

// Any type that implements MyInterface can be used here.
// Interfaces are always pointers, so no need for *MyInter
func execute(m MyInterface) {
    m.MyFunc()
}

// cast int to MyType
num := MyType(3)
// MyType implements MyInterface, so we can call execute()
execute(&num)
```

```
type SomeType int
type SomeInterface interface {
    Foo()
    Bar()
}

// Avoid mixing pointer receivers and value receivers when
// implementing interfaces.
func (s SomeType) Foo() {}
```

```
func (s *SomeType) Bar() {}
func execute(s SomeInterface) {
    s.Foo()
}
s := SomeType(1)
execute(s)  // error: can only use &m
            // (even though we implemented it as a value r

execute(&s) // OK
```

```
// determine which type implements the interface
func run(s SomeInterface) {
    // allows using `s` as `*FooType`
    if foo, ok := s.(*FooType); ok {
        foo.Foo()
    }

    // allows using `s` as `*BarType`
    if bar, ok := s.(*BarType); ok {
        bar.Bar()
    }
}

f := FooType(1)
b := BarType(2)
run(&f)   // prints "foo"
run(&b)   // prints "bar"

// -- boilerplate for above --
type FooType int
type BarType int

type SomeInterface interface {
    Baz()
}

func (f *FooType) Baz() {}
func (f *FooType) Foo() {
    fmt.Println("foo")
}

func (b *BarType) Baz() {}
func (b *BarType) Bar() {
    fmt.Println("bar")
}

```

Type Embedding

Type embedding allows types to be "embedded" in another type. Doing this provides all the fields and functionality of the embedded types at the top level. Similar to inheritance in other languages.

Interfaces

Embedding an interface within another interface creates a composite interface. This composite interface requires all embedded interface functions to be implemented.

```
type Whisperer interface {
    Whisper() string
}

type Yeller interface {
    Yell() string
}

// embed 2 types
type Talker interface {
    Whisperer        // only the type name is used for emb
    Yeller
}

// we can use any Talker here
func talk(t Talker) {
    // Since we embedded both Whisperer and Yeller,
    // we can use both functions.
    fmt.Println(t.Yell())
    fmt.Println(t.Whisper())
}
```

Structs

Embedding a struct within another struct creates a composite structure. This composite structure will have access to all embedded fields and methods at the top level, through a concept known as method and field *promotion*.

```
type Account struct {
    accountId int
    balance   int
    name      string
}

func (a *Account) SetBalance(n int) {
    a.balance = n
}

type ManagerAccount struct {
    Account     // embed the Account struct
}

mgrAcct := ManagerAccount{Account{2, 30, "Cassandra"}}
// embedded type fields & functions are "promoted" and can
mgrAcct.SetBalance(50)
fmt.Println(mgrAcct.balance)   // 50
```

Concurrency

Go's concurrency model abstracts both threaded and asynchronous operations via the `go` keyword.

Defer

`defer` allows a function to be ran *after* the current function. It can be utilized for cleanup operations.

```
func foo() {
    // defer this function call until after foo() complete
    defer fmt.Println("done!")
    fmt.Println("foo'd")
}

foo()

// output:
//   foo'd
//   done!
```

Goroutines

Goroutines are green threads that are managed by Go. They can be both computation heavy and wait on external signals.

```
func longRunning() {
    time.Sleep(1000 * time.Millisecond)
    fmt.Println("longrunning() complete")
}

// spawn a new goroutine with the `go` keyword
go longRunning()        // this will run in the background
fmt.Println("goroutine running")
time.Sleep(1100 * time.Millisecond)
fmt.Println("program end")

// output:
//   goroutine running
//   longrunning() complete
//   program end
```

```
counter := 0
// create closure
wait := func(ms time.Duration) {
    time.Sleep(ms * time.Millisecond)
        // capture the counter
    counter += 1
}
fmt.Println("Launching goroutines")

// run closure 3 times in 3 different goroutines
```

```
go wait(100)
go wait(200)
go wait(300)

fmt.Println("Launched.     Counter =", counter)        //
time.Sleep(400 * time.Millisecond)
fmt.Println("Waited 400ms. Counter =", counter)   // 3

// output:
//   Launching goroutines
//   Launched.     Counter = 0
//   Launched.     Counter = 3
```

WaitGroups

The main thread of the program will *not* wait for goroutines to finish. `WaitGroup` provides a counter that can be waited upon until it reaches 0. This can be used to ensure that all work is completed by goroutines before exiting the program.

```
// create a new WaitGroup
var wg sync.WaitGroup

sum := 0
for i := 0; i < 20; i++ {
    wg.Add(1)         // add 1 to the WaitGroup counter
    value := 1
    // spawn a goroutine
   go func() {
        // defer execution of wg.Done()
        defer wg.Done()     // wg.Done() decrements the co
        sum += value
    }()
}
wg.Wait()      // wait for the counter to reach 0
fmt.Println("sum =", sum)  // 20
```

Mutex

`Mutex` (MUTual EXclusion) provides a lock that can only be accessed by one goroutine at a time. This is used to synchronize data across multiple goroutines. Attempting to lock a `Mutex` will block (wait) until it is safe to do so. Once locked, the protected data can be operated upon since all other goroutines are forced to wait until the lock is available. Unlock the `Mutex` once work is completed, so other goroutines can access it.

```
type SyncedData struct {
    inner map[string]int
    mutex sync.Mutex
}

func (d *SyncedData) Insert(k string, v int) {
```

```
    // Lock the Mutex before changing data.
    // This will wait until the Mutex is available
    // (and therefore safe to lock).
    d.mutex.Lock()
    d.inner[k] = v
    // Always unlock when done, so other goroutines
    // can access the data.
    d.mutex.Unlock()
}

func (d *SyncedData) Get(k string) int {
    d.mutex.Lock()        // Wait for Mutex to be unlocked
    data := d.inner[k]    // Do stuff.
    d.mutex.Unlock()      // Unlock so others can use data
    return data
}

func (d *SyncedData) GetDeferred(k string) int {
    d.mutex.Lock()
    // Use `defer` so the Mutex unlocks regardless of func
    defer d.mutex.Unlock()
    data := d.inner[k]
    return data
}

func (d *SyncedData) InsertDeferred(k string, v int) {
    d.mutex.Lock()
    defer d.mutex.Unlock()
    d.inner[k] = v
}

func main() {
    // Mutex abstracted behind SyncedData
    data := SyncedData{inner: make(map[string]int)}
    // Can be accessed by any number of goroutines since M
    // will wait to become unlocked.
    data.Insert("sample a", 5)
    data.InsertDeferred("sample b", 10)
}
```

Channels

Channels provide a communication pipe between goroutines. Data is *sent* into one end of the channel, and *received* at the other end of the channel.

```
// Create an unbuffered channel. Unbuffered
// channels always block (wait) until data is read
// on the receiving end.
channel := make(chan int)
go func() { channel <- 1 }()   // use `channel <-` to send
go func() { channel <- 2 }()
go func() { channel <- 3 }()

// non-deterministic ordering since we used goroutines
first := <-channel             // use `<- channel` to read
second := <-channel
third := <-channel
fmt.Println(first, second, third)
```

```
// closing the channel prevents any more data from being s
close(channel)
```

```
// Create a buffered channel with capacity for 2 messages.
// Buffered channels will not block until full.
channel := make(chan int, 2)
channel <- 1
channel <- 2

go func() {
    // Blocks because channel is full. Since we
    // are in a goroutine, main thread can continue
    channel <- 3
}()

// read from channels
first := <-channel
second := <-channel
third := <-channel
fmt.Println(first, second, third)
// output:
//   1
//   2
//   3
```

```
one := make(chan int)
two := make(chan int)

// infinite loop
for {
    // `select` will poll each channel trying to read data
    // There is no ordering like a `switch`; channels are
    select {
    case o := <-one:         // try to read from channel on
        fmt.Println("one:", o)
    case t := <-two:         // try to read from channel tw
        fmt.Println("two:", t)
        // use time.After() to create a timeout (maximum w
    case <-time.After(300 * time.Millisecond):
        fmt.Println("timed out")
        return
    // when there is no data to read from any channel, run
    default:
        fmt.Println("no data to receive")
        // sleep here so we don't consume all CPU cycles
        time.Sleep(50 * time.Millisecond)
    }
}
```

Errors

Go has no exceptions. Errors are returned as interface `error` values, or the program can abort completely with a panic.

```go
import "errors"        // helper package to work with erro

func divide(lhs, rhs int) (int, error) {
    if rhs == 0 {
        // create a new error with a message
        return 0, errors.New("cannot divide by zero")
    } else {
        // our success branch, `error` set to `nil`
        return lhs / rhs, nil
    }
}
// always check for errors
answer, err := div(10, 0)
if err != nil {
    // we have some error: print and return
    fmt.Println(err)
    return
}
// ok to use `answer` because err is nil here
fmt.Println(answer)

```

It is possible to create your own error types that contain relevant data related to the error.

```go
// stdlib Error interface
type Error interface {
    Error() string
}
// create your own error type
type DivError struct {
    a, b int
}
// always a receiver function
func (d *DivError) Error() string {
    return fmt.Sprintf("cannot divide %d by %d", d.a, d.b)
}

func div(lhs, rhs int) (int, error) {
    if rhs == 0 {
            // return the a pointer to the error type inst
        return 0, &DivError{lhs, rhs}
    } else {
        return lhs / rhs, nil
    }
}

// Use `errors.As` to determine if error is of specific ty
answer, err := div(10, 0)
var divError *DivError
if errors.As(err, &divError) {
    fmt.Println("div error")
} else {
    fmt.Println("other error")
}
```

Testing

Test files share the same name as the file under test, but with `_test` appended. They must also be part of the same package.

Tests can be ran with `go test`

```
sample/
  sample.go
  sample_test.go
```

sample.go:

```
package sample

func double(n int) int {
    return n * 2
}
```

sample_test.go:

```
package sample

import "testing"        // required

// function names must start with `Test` and
// have `t *testing.T` as only parameter
func TestDouble(t *testing.T) {
    // run function under test
    retVal := double(2)
    if retVal != 4 {
        t.Errorf("double(2)=%v, want 4", retVal)
    }
}

// test table
func TestDoubleTable(t *testing.T) {
    // anonymous struct containing input and expected outp
    table := []struct {
        input int
        want  int
    }{
        // input/output pairs
        {0, 0},
        {1, 2},
        {2, 4},
        {3, 6},
    }

    // iterate over the table
    for _, data := range table {
        // test the function
        result := double(data.input)
        // ensure that the result is what we wanted
        if result != data.want {
```

```
            t.Errorf("double(%v)=%v, want %v", data.input,
        }
    }
}

func TestSample(t *testing.T) {
    t.Fail()          // fail and continue test
    t.Errorf("msg")   // fail with message; continue test
    t.FailNow()       // fail and abort test
    t.Fatalf("msg")   // fail with message; abort test
    t.Logf("msg")     // Equivalent to Printf, but only whe
}
```

CLI

Command	Description
`go run ./sourceFile.go`	run source file containing `func main()`
`go build ./sourceFile.go`	build project from file containing `func main()`
`go test`	run test suite
`go clean`	remove cache and build artifacts
`go mod tidy`	download dependencies and remove unused dependencies
`go mod init`	create new Go project

Modules

Modules are composed of multiple packages. Each Go project has a `go.mod` file containing the Go version, module name, and dependencies.

go.mod:

```
module example.com/practice

go 1.17

require (
    example.com/package v1.2.3
)
```

Packages

Packages should have a single purpose. All files that are part of a single package are treated as one large file. ``` projectRoot/ go.mod package1/ package1.go extraStuff.go package2/ package2.go ```

```go
// using packages
package main

import "package1"
import (
    "package2"
    "namespace/packageName"
    . "pkg"    // glob import; no need to reference `pkg`
    pk "namespace/mySuperLongPackageName"    // rename to
)

FuncFromPkg()             // some function from the `pkg`
data := packageName.Data  // use packageName
pk.SomeFunc()             // use mySuperLongPackageName
```

Back To Top