# GO language

The current version of the GO language is described on the [go.dev](https://go.dev/ref/spec) site. Windows users would use [choco](https://community.chocolatey.org/packages/golang) application for operations with golang:

choco install golang

choco upgrade golang

choco uninstall golang

**Go** tools expect a certain layout of the source code. **GOROOT** and **GOPATH** are environment variables that define this layout. Windows users can to inspect these variables with commands:

echo %GOROOT%

echo %GOPATH%

Linux users would use command **printenv**:

printenv GOROOT

printenv GOPATH

**GOROOT** is a variable that defines where your Go SDK is located. You do not need to change this variable, unless you plan to use different Go versions.

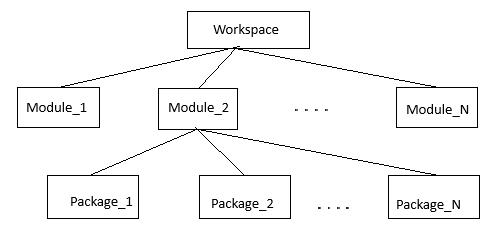
**GOPATH** is a variable that defines the root of your workspace. By default, the workspace directory is a directory that is named go within your user home directory (**~/go** for Linux and macOS, **%USERPROFILE%/go** for Windows). **GOPATH** stores your code base and all the files that are necessary for your development. You can use another directory as your workspace by configuring **GOPATH** for different scopes. **GOPATH** is the root of your workspace and contains the following folders:

* src/: location of Go source code (for example, .go, .c, .g, .s).
* pkg/: location of compiled package code (for example, .a).
* bin/: location of compiled executable programs built by Go.

Recent versions of GO **no longer use this variable**. Use workspaces in the new projects. Workspace is created with the **go work init** command, see to the next paragraph.

## Structure of the application

The diagram below shows the structure of the GO program:



**Workspace** starts in a root directory of the project. The file **go.work** must be created in the directory. This file may be created with any text editor but it would be better to use **go work init** command. Every **module** starts in a separate subdirectory with **go.mod** file. Use **go mod init** command for creating the file. Package is collection of GO files with the same name:

package **dartlib**

function Hello() string { return "hello!" }

…

Package may consist of multiple files but they must be placed in the same directory. Only one package can be placed in a specific directory. The only exception for different package definitions under the same directory is for tests, where the recommended way of defining the package is adding \_test on it (e.g. **package main\_test** or **package dart\_test**).

mkdir sql-ledger-go

cd sql-ledger-go

mkdir loginModule && cd loginModule && **go mod init** loginModule

type nul > login.go

cd ..

mkdir mainModule && cd mainModule && **go mod init** mainModule

type nul > main.go

cd ..

**go work init** ./mainModule ./loginModule

This sequence of commands will create the following structure:

sql-ledger-go

| go.work

| LICENSE

| README.md

|

+---loginModule

| go.mod

| login.go

|

\---mainModule

go.mod

main.go

// [Tutorial: Getting started with multi-module workspaces - The Go Programming Language](https://go.dev/doc/tutorial/workspaces)

Create a new project in the WEB repository when you are working with GIT. Drop root directory with the **git clone** command.

A local module can call a function from another module, as long as both modules belong to the same workspace. The calling syntax is identical to calling modules from the repository. Let the function **Calculate** is described in the module loginModule:

// loginModule/login.go

**package loginmodule**

func Calculate(x, y int) int {

return x + y

}

mainModule/main.go can call this function in this way:

// mainModule/main.go

**package main**

import (

"fmt"

loginmodule "loginModule"

)

func main() {

fmt.Println(loginmodule.Calculate(3, 4))

}

### Importing Packages

In Go, the basic unit of reusable code is called a package. Even the simplest Go program is its own package, and probably uses at least one other package. Like most languages, Go comes with a built-in library of reusable code that you can use for common tasks. The import command allows you to use external packages:

package main

**import "fmt"**

func main() {

fmt.Println("Hello, World!")

}

The **import** is a Go keyword that tells the Go compiler which other packages you want to use in this file. Here you import the **fmt** package that comes with the standard library. The fmt package provides formatting and printing functions that can be useful when developing. Collect all imports into a single operator using parentheses:

package main

**import (**

"fmt"

"strings"

**)**

func main() {

fmt.Println("Please enter your name.")

var name string

fmt.Scanln(&name)

name = strings.TrimSpace(name)

fmt.Printf("Hi, %s! I'm Go!", name)

}

Like most languages, Go comes with a built-in library of reusable code that you can use for common tasks. You don’t need to write your own code to format and print strings, or send an HTTP request, for example. The [Go standard library](https://pkg.go.dev/std) has packages for both of those tasks and many others. Import standard packages by name:

package main

**import "math/rand"**

func main() {

for i := 0; i < 5; i++ {

println(rand.Intn(10))

}

}

One of the most popular packages for generating UUIDs is **github.com/google/uuid**. Third-party packages are always known by their **fully-qualified names**, which include the **site** that hosts the code (e.g. github.com), the **user or organization** that develops it (e.g. google), and the **base name** (e.g. uuid). You will use a package’s fully-qualified name when importing it, when reading its documentation on **pkg.go.dev**, and elsewhere. When referencing it in your code statements, however, you will only use the base name.

You can create modules that will be used in several applications. For that purpose, create a new repository in your archive (GitHub, GitLab, ...). Drop it to your local drive and create a **go.mod** file in the top directory. Use **fully-qualified** name:

go mod init github.com/sammy/random

Use go get command for importing this module into another one:

**go get github.com/sammy/random**

...

package test

**import random**

…

The documentation for **math/rand** says that it actually generates pseudo-random numbers and is “unsuitable for security-sensitive work”. For that kind of work, you would use **crypto/rand** instead. You could write a program to compare the performance of these two rand packages, but you cannot reference both packages by the rand name throughout such a program. To get around this, Go allows you to choose an alternative local name (**alias**) for a package when importing it. Put alias to the left of the fully-qualified package name:

import (

“math/rand”

**crand** “crypto/rand”

)

Use the alias (**crand**) calling functions from crypto/rand.

### The main package

In Go, main is actually a special package name which indicates that the package contains the code for an executable application. That is, it indicates that the package contains code that can be built into a binary and run.

Any package with the name main must also contain a main() function somewhere in the package which acts as the entry point for the program. If it doesn't, and you try to run it, you will get this error:

$ go run \*.go

function main is undeclared in the main package

It's conventional for your main() function to live in a file with the filename main.go. Technically it doesn't have to, but following this convention makes the application entry point easier to find for anyone reading your code in the future.

As an aside, if you try to build or run a non-main package it will also result in an error.:

$ go run \*.go

package command-line-arguments is not a main package

### Internal packages

1. The core of using internal packages lies in their ability to manage visibility and establish clear boundaries within a Go program. By using internal packages, developers can ensure that certain parts of their codebase are "shielded" from external use, thus maintaining a clean and well-defined interface for users and other parts of the program.
2. To take advantage of this feature, you simply need to place the desired package within a directory named internal, or within any subdirectory of an internal directory. The Go toolchain enforces access restrictions based on this convention.
3. When the go command sees an import of a package with internal/ in the import path, it verifies that the importing package is within the tree rooted at the parent of the internal/ directory.
4. For example, a package **/a/b/c/internal/d/e/f** can only be imported by code in the directory tree rooted at **/a/b/c**. It cannot be imported by code in /a/b/g or in any other repository.
5. Internal packages provide a useful mechanism for controlling visibility to parts of your project.
6. // [Importing Packages in Go | DigitalOcean](https://www.digitalocean.com/community/tutorials/importing-packages-in-go)
7. // https://www.digitalocean.com/community/tutorials/importing-packages-in-go
8. // [An introduction to Packages, Imports and Modules in Go – Alex Edwards](https://www.alexedwards.net/blog/an-introduction-to-packages-imports-and-modules)
9. // https://www.alexedwards.net/blog/an-introduction-to-packages-imports-and-modules
10. //[Go Modules Reference - The Go Programming Language](https://go.dev/ref/mod" \l "workspaces)

## Types

[Understanding Golang Type System - The New Stack](https://thenewstack.io/understanding-golang-type-system/)

GO language provides rich set of built-in **basic types**. They are listed in [BasicKind](https://cs.opensource.google/go/go/+/refs/tags/go1.20.7:src/go/types/basic.go;l=8) iota. **Aggregate** types consists of arrays and structures. Pointers, slices, maps, functions, and channels represent **reference** types. **Interfaces** are a separate group of compositional types, which we will consider in the section "Interfaces".

### Aggregate types

#### Arrays

Array stores collection of items with the same type (homogeneous) and are stored in continuous memory locations. Array has **fixed** size. Each element of the array can be accessed with the help of an index. Unlike other programming languages, an **array** is a **value** type. Array declaration:

a := [3]int{1,2,3} // Given fixed size

a := [...]int{1,2,3} // Array literal with length inferred

var a [3]int

a[0] = 1

a[1] = 2

a[2] = 3

The run-time copies array refer into the different underlying data, pass an address if you want the same location:

b := a // b is a copy of a

c := &a // c and a occupy the same location

#### Structures

A **struct** (short for structure) is used to create a collection of members of different data types, into a single variable. To declare a structure in Go, use the **type** and **struct** keywords:

type struct\_name struct {

member1 datatype;

member2 datatype;

member3 datatype;

...

}

To access any member of a structure, use the dot operator (.) between the structure variable name and the structure member:

package main

import "fmt"

type Creature struct {

Name string

}

func main() {

c := Creature{

Name: "Sammy the Shark",

}

fmt.Println(**c.Name**)

}

Within the body of main, we create an instance of Creature by placing a pair of braces after the name of the type, Creature, and then specifying values for that instance’s fields. The instance in c will have its Name field set to “Sammy the Shark”. If every field value will be provided during the instantiation of a struct, you can omit the field names (short declaration):

package main

import "fmt"

type Creature struct {

Name string

Type string

}

func main() {

c := **Creature{"Sammy", "Shark"}**

fmt.Println(c.Name, "the", c.Type)

}

You must provide values for each field in the **struct** when using the short declaration.

If a field name begins with a capital letter, it will be readable and writeable by code outside of the package where the struct was defined (exported field). If the field begins with a lowercase letter, only code within that struct’s package will be able to read and write that field:

package main

import "fmt"

type Creature struct {

Name string

Type string

password string

}

func main() {

c := Creature{

Name: "Sammy",

Type: "Shark",

**password: "secret"**,

}

fmt.Println(c.Name, "the", c.Type)

fmt.Println("Password is", c.password)

}

Field **password** is accessible in **main** package only.

You can also define an **inline struct**. Inline struct definitions appear on the right-hand side of a variable assignment. You must provide an instantiation of them immediately after by providing an additional pair of braces with values for each of the fields you define:

package main

import "fmt"

func main() {

c := struct {

Name string

Type string

}{

Name: "Sammy",

Type: "Shark",

}

fmt.Println(c.Name, "the", c.Type)

}

Methods

In GO, only variables can be specified inside a structure. Functions are described outside the structure and are connected using the **receiver**:

type my\_type struct { }

...

func (**m my\_type**) my\_func() int {

//code

}

In this example, the expression before the function name **(m my\_type)** defines the **receiver** and its **type**. A function written this way is called a **method**. Remember that the receiver is actually a hidden parameter of the method and is passed by value. If a method changes the fields inside the structure, it is necessary to pass the **pointer**, otherwise the changes will not be saved:

package main

import (

"fmt"

"math"

)

type Vertex struct {

X, Y float64

}

func (v Vertex) Abs() float64 { // copy of the receiver

return math.Sqrt(v.X\*v.X + v.Y\*v.Y)

}

func (v \*Vertex) Scale(f float64) { // pointer to the receiver

v.X = v.X \* f

v.Y = v.Y \* f

}

func main() {

v := Vertex{3, 4}

v.Scale(10) // Compiler will convert **v** into the pointer of this value

fmt.Println(v.Abs())

}

Methods can be bound not only to structures, but also to any **custom** type described in the same module:

...

type **WalkerFunc** func() // Custom type

func (**wf WalkerFunc**) Walk() { // Method attached to WalkerFunc

fmt.Println("start walking")

wf()

fmt.Printf("stop walking\n\n")

}

…

### Reference types

#### Pointers

When you write software in Go you’ll be writing functions and methods. You pass data to these functions as arguments. Sometimes, the function needs a local copy of the data, and you want the original to remain unchanged. GO allows that passing parameter by value:

package main

import "fmt"

func add(left int, right int) int {

var total int = left + right

left = 0

return total

}

func main() {

var (

left1 int = 4

right1 int = 5

)

var total int = add(left1, right1)

fmt.Println("total = ", total)

fmt.Println("left1 = ", left1, "\nright1 = ", right1)

}

The application will print:

go run .\main.go

total = 9

left = 4

right = 5

The left1 variable will not be changed because "add" function obtains a value of this variable. It makes local copy of the left1 variable.

Other times, you may want the function to be able to alter the data in the original variable. In this case, you don’t need to send the actual data to the function; you just need to tell the function where the data is located in memory. A data type called a pointer holds the memory address of the data, but not the data itself.

If you place an ampersand (&) in front of a variable name, you are stating that you want to get the address, or a pointer to that variable:

package main

import "fmt"

func main() {

var creature string = "shark"

**var pointer \*string = &creature //** Pointer to the creature

fmt.Println("creature =", creature)

fmt.Println("pointer =", pointer)

}

This application will print:

creature = shark

pointer = 0xc0000721e0 <-- This value may be different on your computer

If you want to print out the value of the variable being pointed at from the pointer variable, you need to **dereference** that variable. The following code uses the \* operator to dereference the pointer variable and retrieve its value:

package main

import "fmt"

func main() {

var creature string = "shark"

var pointer \*string = &creature

fmt.Println("creature =", creature)

fmt.Println("pointer =", pointer)

fmt.Println("\*pointer =", **\*pointer**)

}

This application will print:

creature = shark

pointer = 0xc000028070

\*pointer = shark

If you want to modify the value stored at the pointer variable’s location, you can use the dereference operator as well:

package main

import "fmt"

func main() {

var creature string = "shark"

var pointer \*string = &creature

fmt.Println("creature =", creature)

fmt.Println("pointer =", pointer)

fmt.Println("\*pointer =", \*pointer)

**\*pointer = "jellyfish"**

fmt.Println("After assignment: pointer =", pointer)

fmt.Println("\*pointer =", \*pointer)

fmt.Println("creature =", creature)

}

This application will print:

creature = shark

**pointer = 0xc000104030**

\*pointer = shark

After assignment: **pointer = 0xc000104030**

\*pointer = jellyfish

creature = jellyfish

This example demonstrates that we actually changed the value of the **creature** variable, pointer's value remains unchanged.

When you write a function, you can define arguments to be passed ether by value, or by reference. Passing by value means that a copy of that value is sent to the function, and any changes to that argument within that function only effect that variable within that function, and not where it was passed from. However, if you pass by reference, meaning you pass a pointer to that argument, you can change the value from within the function, and also change the value of the original variable that was passed in.

Pointers are best used when passing large arrays or structures to a function. In this way, you will avoid unnecessary copying, save RAM and the program will run faster.

The same rule applies to member functions: use pointers when the receiver is a large structure or array.

In Go, pointers are used to hold the memory address of a value. However, unlike languages such as C, Go does not support pointer arithmetic, which means you cannot directly increment or decrement pointers using `++` or `--` operators. Some operations on pointers are defined in [unsafe](https://pkg.go.dev/unsafe) package.

package main

import (

"fmt"

"**unsafe**"

)

func main() {

arr := [3]int{1, 2, 3}

ptr := &arr[0]

fmt.Println(\*ptr) // output: 1

**ptr = (\*int)(unsafe.Pointer(uintptr(unsafe.Pointer(ptr)) +**

**uintptr(unsafe.Sizeof(arr[0]))))**

fmt.Println(\*ptr) // output: 2

}

Please exercise caution when using the unsafe package, as it bypasses some of Go's safety mechanisms. It's generally recommended to **avoid using unsafe** unless absolutely necessary and to explore alternative approaches whenever possible.

#### Slices

In Go, slice is an abstraction over arrays. It consists of a pointer to the backing array, the length of the slice, and its capacity. The length is the number of elements in the slice, while the capacity is the number of elements in the underlying array. Here's how a slice is defined in [Go runtime](https://github.com/golang/go/blob/cff7267e0d77f02d582c613c272b6f8ebf1c0412/src/runtime/slice.go" \l "L15):

type slice struct {

array unsafe.Pointer

len int

cap int

}

The **array** field is a pointer to the (first element of the) backing array. The **len** field is the length of the slice, denoting the number of items in the slice, and the **cap** field is the capacity of the slice. Capacity stores the number of elements in the backing array, and it is the maximum number of elements the slice can hold without reallocating the backing array.

The type specification for a slice is []T, where T is the type of the elements of the slice. Unlike an array type, a slice type has no specified length. A slice literal is declared just like an array literal, except you leave out the element count:

letters := []string{"a", "b", "c", "d"}

A slice can be created with the built-in function called make, which has the signature,

func make([]T, len, cap) []T

where T stands for the element type of the slice to be created. The make function takes a type, a length, and an optional capacity.

var s []byte

s = make([]byte, 5) // capacity is 5

// s == []byte{0, 0, 0, 0, 0}

The length and capacity of a slice can be inspected using the built-in len and cap functions:

len(s) == 5

cap(s) == 5

The zero value of a slice is nil. The len and cap functions will both return 0 for a nil slice. A slice can also be formed by “slicing” an existing slice or array. Slicing is done by specifying a half-open range with two indices separated by a colon:

b := []byte{'g', 'o', 'l', 'a', 'n', 'g'}

// b[1:4] == []byte{'o', 'l', 'a'}, sharing the same storage as b

The start and end indices of a slice expression are optional; they default to zero and the slice’s length respectively:

// b[:2] == []byte{'g', 'o'}

// b[2:] == []byte{'l', 'a', 'n', 'g'}

// b[:] == b

A slice may be created from given array:

x := [3]string{"Лайка", "Белка", "Стрелка"}

s := x[:] // a slice referencing the storage of x

Slicing does not copy the slice’s data. It creates a new slice value that points to the original array. Therefore, modifying the elements (not the slice itself) of a re-slice modifies the elements of the original slice:

d := []byte{'r', 'o', 'a', 'd'}

e := d[2:] // e == []byte{'a', 'd'}

e[1] = 'm'

// e == []byte{'a', 'm'}

// d == []byte{'r', 'o', 'a', 'm'}

For increasing the capacity of a slice one must create a new, larger slice and copy the contents of the original slice into it. The **copy** function copies data from a source slice to a destination slice. It returns the number of elements copied:

func copy(dst, src []T) int

This function supports copying between slices of different lengths (it will copy only up to the smaller number of elements). In addition, copy can handle source and destination slices that share the same underlying array, handling overlapping slices correctly:

t := make([]byte, len(s), (cap(s)+1)\*2)

copy(t, s)

s = t

Go provides a built-in append function that’s good for most purposes; it has the signature

func append(s []T, x ...T) []T

The append function appends the elements x to the end of the slice s, and grows the slice if a greater capacity is needed:

a := make([]int, 1)

// a == []int{0}

a = append(a, 1, 2, 3)

// a == []int{0, 1, 2, 3}

Slice is a structure, so when you specify it in the function parameter of a function, the GO environment passes a copy. The function below demonstrates this:

package main

import "fmt"

func testSlice(arg []int) {

arg[0] = 10

newArg := append(arg, 7, 8, 9)

**arg = newArg**  // This assignment has no sence

}

func main() {

data := []int{1, 2, 3}

fmt.Println(data)

fmt.Println("Size: ", len(data), "\tCapacity: ", cap(data))

testSlice(data)

fmt.Println("After update: ", data)

fmt.Println("Size: ", len(data), "\tCapacity: ", cap(data))

}

It prints:

[1 2 3]

Size: 3 Capacity: 3

After update: [**10** 2 3]

Size: 3 Capacity: 3

First element of the array was changed but no additional items were added. There are two ways to fix an error: pass a pointer to the function, or to return a corrected slice:

package main

import "fmt"

func testSlice(arg []int) []int {

arg[0] = 10

**return append(arg, 7, 8, 9)**

}

func main() {

data := []int{1, 2, 3}

fmt.Println(data)

fmt.Println("Size: ", len(data), "\tCapacity: ", cap(data))

data = testSlice(data)

fmt.Println("After update: ", data)

fmt.Println("Size: ", len(data), "\tCapacity: ", cap(data))

}

This function will print:

[1 2 3]

Size: 3 Capacity: 3

After update: [10 2 3 7 8 9]

Size: 6 Capacity: 6

#### Maps

One of the most useful data structures in computer science is the hash table. Many hash table implementations exist with varying properties, but in general they offer fast lookups, adds, and deletes. Go provides a built-in map type that implements a hash table. A Go map type looks like this:

map[KeyType]ValueType

where KeyType may be any type that is [comparable](https://go.dev/ref/spec" \l "Comparison_operators), and ValueType may be any type at all, including another map!

This variable m is a map of string keys to int values:

var m map[string]int

Map types are reference types, like pointers or slices, and so the value of m above is nil; it doesn’t point to an initialized map. A nil map behaves like an empty map when reading, but attempts to write to a nil map will cause a runtime panic. To initialize a map, use the built in make function:

m = make(map[string]int)

The make function allocates and initializes a hash map data structure and returns a map value that points to it.

Use indexes for reading and writing map elements:

m["route"] = 66

i := m["route"]

If the requested key doesn’t exist, we get the value type’s zero value. In this case the value type is int, so the zero value is 0:

j := m["root"] // j == 0

The built in len function returns on the number of items in a map:

n := len(m)

The built in delete function removes an entry from the map:

delete(m, "route")

The delete function doesn’t return anything, and will do nothing if the specified key doesn’t exist. A two-value assignment tests for the existence of a key:

i, ok := m["route"]

In this statement, the first value (**i**) is assigned the value stored under the key "route". If that key doesn’t exist, i is the value type’s zero value (0). The second value (**ok**) is a bool that is true if the key exists in the map, and false if not.

To test for a key without retrieving the value, use an underscore in place of the first value:

\_, ok := m["route"]

To iterate over the contents of a map, use the range keyword:

for key, value := range m {

fmt.Println("Key:", key, "Value:", value)

}

To initialize a map with some data, use a map literal:

commits := map[string]int{

"rsc": 3711,

"r": 2138,

"gri": 1908,

"adg": 912,

}

The same syntax may be used to initialize an empty map, which is functionally identical to using the make function:

m = map[string]int{}

This example traverses a linked list of Nodes and prints their values. It uses a map of Node pointers to detect cycles in the list:

type Node struct {

Next \*Node

Value interface{}

}

var first \*Node

visited := make(map[\*Node]bool)

for n := first; n != nil; n = n.Next {

if visited[n] {

fmt.Println("cycle detected")

break

}

visited[n] = true

fmt.Println(n.Value)

}

Another instance of helpful zero values is a map of slices. Appending to a nil slice just allocates a new slice, so it’s a one-liner to append a value to a map of slices; there’s no need to check if the key exists. In the following example, the slice people is populated with Person values. Each Person has a Name and a slice of Likes. The example creates a map to associate each like with a slice of people that like it.

Another instance of helpful zero values is a map of slices. Appending to a nil slice just allocates a new slice, so it’s a one-liner to append a value to a map of slices; there’s no need to check if the key exists. In the following example, the slice people is populated with Person values. Each Person has a Name and a slice of Likes. The example creates a map to associate each like with a slice of people that like it.

type Person struct {

Name string

Likes []string

}

var people []\*Person

likes := make(map[string][]\*Person)

for \_, p := range people {

for \_, l := range p.Likes {

likes[l] = append(likes[l], p)

}

}

To print a list of people who like cheese:

for \_, p := range likes["cheese"] {

fmt.Println(p.Name, "likes cheese.")

}

Note that since both **range** and **len** treat a nil slice as a zero-length slice, the last example will work even if nobody likes cheese.

Keys

Keys of a map may be of any type that is comparable. There are 6 comparison operators in a GO language:

== equal

!= not equal

< less

<= less or equal

> greater

>= greater or equal

In any comparison, the first operand must be assignable to the type of the second operand, or vice versa. The equality operators == and != apply to operands of **comparable** types. The ordering operators <, <=, >, and >= apply to operands of **ordered** types. These terms and the result of the comparisons are defined as follows:

* **Boolean** types are **comparable**. Two boolean values are equal if they are either both true or both false.
* **Integer** types are **comparable** and **ordered**. Two integer values are compared in the usual way.
* **Floating-point** types are **comparable** and **ordered**. Two floating-point values are compared as defined by the IEEE 754 standard.
* **Complex** types are **comparable**. Two complex values u and v are equal if both real(u) == real(v) and imag(u) == imag(v).
* **String** types are **comparable** and **ordered**. Two string values are compared lexically byte-wise.
* **Pointer** types are **comparable**. Two pointer values are equal if they point to the same variable or if both have value nil. Pointers to distinct zero-size variables may or may not be equal.
* **Channel** types are comparable. Two channel values are equal if they were created by the same call to make or if both have value nil.
* **Interface** types that are not type parameters are **comparable**. Two interface values are equal if they have identical dynamic types and equal dynamic values or if both have value nil.
* A value x of **non-interface type X** and a value t **of interface type T** can be compared if type X is comparable and X implements T. They are equal if t's dynamic type is identical to X and t's dynamic value is equal to x.
* **Struct** types are **comparable** if all their field types are comparable. Two struct values are equal if their corresponding non-blank field values are equal. The fields are compared in source order, and comparison stops as soon as two field values differ (or all fields have been compared).
* **Array** types are **comparable** if their array element types are comparable. Two array values are equal if their corresponding element values are equal. The elements are compared in ascending index order, and comparison stops as soon as two element values differ (or all elements have been compared).
* **Type** parameters are **comparable** if they are strictly comparable (see below).

A comparison of two interface values with identical dynamic types causes a run-time panic if that type is not comparable. This behavior applies not only to direct interface value comparisons but also when comparing arrays of interface values or structs with interface-valued fields.

Slice, map, and function types are not comparable. However, as a special case, a slice, map, or function value may be compared to the predeclared identifier nil. Comparison of pointer, channel, and interface values to nil is also allowed and follows from the general rules above.

A type is strictly comparable if it is comparable and not an interface type nor composed of interface types. Specifically:

* Boolean, numeric, string, pointer, and channel types are strictly comparable.
* Struct types are strictly comparable if all their field types are strictly comparable.
* Array types are strictly comparable if their array element types are strictly comparable.
* Type parameters are strictly comparable if all types in their type set are strictly comparable.

The use of structural keys can greatly simplify the processing of MAPs. Consider the following two-dimensional MAP:

hits := make(map[string]map[string]int)

Each key of the outer map is the path to a web page with its own inner map. Each inner map key is a two-letter country code. This expression retrieves the number of times an Australian has loaded the documentation page:

n := hits["/doc/"]["au"]

Unfortunately, adding data to such a MAP is quite tricky:

func add(m map[string]map[string]int, path, country string) {

mm, ok := m[path]

if !ok {

mm = make(map[string]int) // Create an empty map

m[path] = mm

}

mm[country]++

}

...

add(hits, "/doc/", "au")

A design that uses a single map with a struct key does away with all that complexity:

type Key struct {

Path, Country string

}

hits := make(map[Key]int)

When a Vietnamese person visits the home page, incrementing (and possibly creating) the appropriate counter is a one-liner:

hits[Key{"/", "vn"}]++

And it’s similarly straightforward to see how many Swiss people have read the spec:

n := hits[Key{"/ref/spec", "ch"}]

Concurrency

Maps are **not safe** for concurrent use. If you need to read from and write to a map from concurrently executing goroutines, the accesses must be mediated by some kind of synchronization mechanism. One common way to protect maps is with [sync.RWMutex](https://pkg.go.dev/sync" \l "RWMutex). The zero value for a **RWMutex** is an unlocked mutex.

This statement declares a counter variable that is an anonymous struct containing a map and an embedded sync.RWMutex.

var counter = struct{

**sync.RWMutex**

m map[string]int

}{m: make(map[string]int)}

To read from the counter, take the read lock:

**counter.RLock()**

n := counter.m["some\_key"]

**counter.RUnlock()**

fmt.Println("some\_key:", n)

To write to the counter, take the write lock:

**counter.Lock()**

counter.m["some\_key"]++

**counter.Unlock()**

Iteration order

When iterating over a map with a range loop, the iteration order is not specified and is not guaranteed to be the same from one iteration to the next. If you require a stable iteration order you must maintain a separate data structure that specifies that order. This example uses a separate sorted slice of keys to print a **map[int]string** in key order:

import "sort"

var m map[int]string

var keys []int

for k := range m {

keys = append(keys, k)

}

sort.Ints(keys)

for \_, k := range keys {

fmt.Println("Key:", k, "Value:", m[k])

}

Latest GO has function Keys. It returns iterator that you can directly use in a range clause:

import (

"maps"

)

func main() {

m := map[string]int{"alpha": 1, "bravo": 2}

for k := range **maps.Keys(m)** {

// ...

}

}

If you need the map keys into a slice, you have to collect them from the iterator using slices.Collect:

import (

"fmt"

"maps"

"slices"

)

func main() {

m := map[string]int{"alpha": 1, "bravo": 2}

**keys := slices.Collect(maps.Keys(m))**

fmt.Println(keys) // [alpha bravo]

}

This article follows the [Go maps in action - The Go Programming Language](https://go.dev/blog/maps) site.

// [Types in the Go Programming Language | by Vladimir Vivien | Learning the Go Programming Language | Medium](https://medium.com/learning-the-go-programming-language/types-in-the-go-programming-language-65e945d0a692)

https://golangdocs.com/functions-in-golang

// Pointers, slices, maps, functions, and channels

## Interfaces

In Go language, the [interface](https://www.geeksforgeeks.org/interfaces-in-golang/) is a custom type that is used to specify a set of one or more method signatures. Interface is an abstract type, so you are not allowed to create an instance of the interface but you are allowed to create a variable of an interface type and this variable can be assigned with a concrete type value that has all methods the interface requires. In Go language, you can create an interface using the following syntax:

type interface\_name interface{

// Method signatures

}

In the Go language, it is necessary to implement all the methods declared in the interface for implementing an interface. The **go** language **interfaces** are implemented implicitly. It does not contain any specific keyword to implement an interface just like other languages:

type ianimal interface {

speak()

sayHello()

}

type animal struct {

}

func (animal) speak() {

fmt.Println("???")

}

func (nml animal) sayHello() {

nml.speak()

}

Type **animal** implements interface **ianimal**.

Important Points:

* The zero value of the interface is nil.
* The interface type consists of two subtypes one is static and dynamic. The **static** type is the interface itself. A **dynamic** type occurs after a value is assigned to a variable of an interface type.
* An interface with no methods (**interface{}**) is called an **empty interface** and has the alias **any**. Value of any type may be assigned to a variable of the **emty interface** type. This interface was widely used for programming containers before the advent of generic functions.

#### Operations with interfaces

Declaration of a variable

Interface is custom type thus declaration a variable of this type follows common rules:

var pet ianimal

The variable pet will have the value nil. It is possible to declare a variable with an initial value:

var pet ianimal = dog{}

Such a declaration is possible if the structure **dog** supports the interface **ianimal**.

Assignment

A value of any custom type may be assigned to a variable of the interface type. The go compiler will reject the assignment if a type of the value does not suppport an interface:

package main

import "fmt"

type ianimal interface {

speak()

}

type dog struct {

}

func (dog) speak() {

fmt.Println("gav gav")

}

type cat struct {

}

func (cat) sing() {

fmt.Println("miau miau")

}

func main() {

var flint, robber ianimal

flint = dog{}

**robber = cat{}**

flint.speak()

robber.speak()

}

The compiler will capture an error because **cat** has no the speak() function. Change the name of the sing() function and this example will compile:

func (cat) speak() { // sing()

Interfaces provide go with what other languages call polymorphism:

func main() {

var myPet ianimal

myPet = dog{}

**myPet.speak()**

myPet = cat{}

**myPet.speak()**

}

The same operator **myPet.speak()** will print two different values:

gav gav

miau miau

The static type of the **myPet** variable is the **ianimal** interface, but after assignments this variable takes on two different dynamic types: **dog** and **cat**.

#### Type Assertions

Type assertion is a process to extract the values of the interface:

value, ok := a.(T)

Here, **a** is the value or the expression of the interface, and **T** is the type also known as asserted type. The **value** contains the dynamic value of the **a** and **ok** will set to **true** if a dynamic type of the **a** is equal to **T**.The operation sets **ok** to **false**, **value** will have zero if the type of the **a** is not equal to **T**. The ok variable can be omitted:

value := a.(T)

but the operation will panics if the dynamic type of the interface does not match **T**.

#### Type Switch

Type switch is used to compare the concrete type of an interface with the multiple types provide in the case statements. It is similar to type assertion with only one difference, i.e, **case** specifies **types**, not the values. You can also compare a type to the interface type.

func myfun(a interface{}) {

switch a.(type) {

case int:

fmt.Println("Type: int, Value:", a.(int))

case string:

fmt.Println("\nType: string, Value: ", a.(string))

case float64:

fmt.Println("\nType: float64, Value: ", a.(float64))

default:

fmt.Println("\nType not found")

}

}

#### Composition

The name of another interface can be specified in the list of interface methods. The new interface inherits all the methods of the nested interface. This operation is called **interface** **composition** and is exactly the same as **object composition**.

type BasicDatabase interface {

CreateTable(string) error

DeleteTable(string) error

}

type SpecificDatabase interface {

BasicDatabase

CreateUserRecord(User) error

}

## Context

Many functions in Go use the **context** package to gather additional information about the environment they’re being executed in, and will typically provide that **context** to the functions they also call. By using the **context.Context** interface in the context package and passing it from function to function, programs can convey that information from the beginning function of a program, such as main, all the way to the deepest function call in the program. The Context function of an **http.Request**, for example, will provide a **context.Context** that includes information about the client making the request and will end if the client disconnects before the request is finished. By passing this **context.Context** value into a function that then makes a call to the **QueryContext** function of a sql.DB, the database query will also be stopped if it’s still running when the client disconnects. Empty context may be created using 2 functions:

1. **context.TODO()** - TODO returns a non-nil, empty [Context]. Code should use context.TODO when it's unclear which Context to use or it is not yet available (because the surrounding function has not yet been extended to accept a Context parameter).
2. **context.Background()** - Background returns a non-nil, empty [Context]. It is never canceled, has no values, and has no deadline. It is typically used by the main function, initialization, and tests, and as the top-level Context for incoming requests.

An example below demonstrates, how to create an empty context and pass it to child function. The variable’s name is **ctx**, which is commonly used for context values. It’s also recommended to put the **context.Context** parameter as the first parameter in a function. Type context.Context is an interface thus is passed to the **doSomething()** function by reference.

package main

import (

"context"

"fmt"

)

func doSomething(ctx context.Context) {

fmt.Println("Doing something!")

}

func main() {

ctx := context.TODO()

doSomething(ctx)

}

To add a new value to a context, use the **context.WithValue** function.

WithValue(parent Context, key, val any)

The function accepts three parameters: the parent context.Context, the key, and the value. The parent context is the context to add the value to while preserving all the other information about the parent context. The key is then used to retrieve the value from the context. The provided key must be **comparable** and should not be of type string or any other built-in type to avoid collisions between packages using context. Users of WithValue should define their own types for keys. To avoid allocating when assigning to an interface{}, context keys often have concrete type struct{}. The value can be any data type. The context.WithValue will then return a new context.Context value with the value added to it. Parent context remains unchanged.

import (

"context"

"fmt"

)

func main() {

**type favContextKey string**

f := func(ctx context.Context, k favContextKey) {

if v := ctx.Value(k); v != nil {

fmt.Println("found value:", v)

return

}

fmt.Println("key not found:", k)

}

**k := favContextKey("language")**

**ctx := context.WithValue(context.Background(), k, "Go")**

f(ctx, k)

f(ctx, favContextKey("color"))

}

**Value** returns the value associated with this context for key, or **nil** if no value is associated with key:

ctx.Value(key any) any

The context.Context type provides a method called **Done** that can be checked to see whether a context has ended or not. This method returns a channel that is closed when the context is done. The following code example shows how a select statement could potentially be used in a long-running function that receives results from a channel, but also watches for when a context’s Done channel is closed:

ctx := context.Background()

resultsCh := make(chan \*WorkResult)

for {

select {

**case <- ctx.Done()**:

// The context is over, stop processing results

return

case result := <- resultsCh:

// Process the results received

}

}

### Canceling a Context

Canceling a context is the most straightforward and controllable way to end a context. Similar to including a value in a context with **context.WithValue**, it’s possible to associate a “cancel” function with a context using the **context.WithCancel** function. This function receives a parent context as a parameter and returns a new context as well as a function that can be used to cancel the returned context. Also, similar to **context.WithValue**, calling the cancel function returned will only cancel the context returned and any contexts that use it as a parent context.

package main

import (

"context"

"fmt"

"time"

)

func doSomething(ctx context.Context) {

ctx, cancelCtx := context.WithCancel(ctx)

printCh := make(chan int)

go doAnother(ctx, printCh)

for num := 1; num <= 3; num++ {

printCh <- num

}

cancelCtx()

time.Sleep(100 \* time.Millisecond)

fmt.Printf("doSomething: finished\n")

}

func doAnother(ctx context.Context, printCh <-chan int) {

for {

select {

case <-ctx.Done():

if err := ctx.Err(); err != nil {

fmt.Printf("doAnother err: %s\n", err)

}

fmt.Printf("doAnother: finished\n")

return

case num := <-printCh:

fmt.Printf("doAnother: %d\n", num)

}

}

}

func main() {

ctx := context.WithoutCancel(context.TODO())

doSomething(ctx)

}

### Giving a Context a Deadline

Using **context.WithDeadline** with a context allows you to set a deadline for when the context needs to be finished, and it will automatically end when that deadline passes.

...

func doSomething(ctx context.Context) {

deadline := time.Now().Add(1500 \* time.Millisecond)

**ctx, cancelCtx := context.WithDeadline(ctx, deadline)**

defer cancelCtx()

printCh := make(chan int)

go doAnother(ctx, printCh)

for num := 1; num <= 3; num++ {

select {

case printCh <- num:

time.Sleep(1 \* time.Second)

case <-ctx.Done():

break

}

}

cancelCtx()

time.Sleep(100 \* time.Millisecond)

fmt.Printf("doSomething: finished\n")

}

...

The **defer cancelCtx()** isn’t necessarily required because the other call will always be run, but it can be useful to keep it in case there are any return statements in the future that cause it to be missed. When a context is canceled from a deadline, the cancel function is still required to be called in order to clean up any resources that were used, so this is more of a safety measure.

### Giving a Context a Time Limit

With **context.WithDeadline** you provide a specific **time.Time** for the context to end, but by using the **context.WithTimeout** function you only need to provide a **time.Duration** value for how long you want the context to last.

func doSomething(ctx context.Context) {

ctx, cancelCtx := context.WithTimeout(ctx, 1500\*time.Millisecond)

defer cancelCtx()

printCh := make(chan int)

go doAnother(ctx, printCh)

for num := 1; num <= 3; num++ {

select {

case printCh <- num:

time.Sleep(1 \* time.Second)

case <-ctx.Done():

break

}

}

time.Sleep(100 \* time.Millisecond)

fmt.Printf("doSomething: finished\n")

}

// [How To Use Contexts in Go | DigitalOcean](https://www.digitalocean.com/community/tutorials/how-to-use-contexts-in-go)

## Signals

This endless application will run until we press CTRL+C:

package main

import "fmt"

func main() {

fmt.Println("Start worker")

myWorker()

fmt.Println("Finish worker")

}

func myWorker() error {

for {

err := doSomethingRepeatedly()

if err != nil {

return err

}

}

}

func doSomethingRepeatedly() error {

// ...

return nil

}

Terminal will report "exit status 0xc000013a" after killing the application with CTRL+C. Problems will arise if the program uses external resources that must be closed. No additional operations are performed before terminating the application.

Signals are a way for an operating system to send notifications to a running process in Golang. Signals can be used to perform various actions, such as stopping or restarting a process, or to trigger a specific behavior within a program. 15 signals are defined for Windows environment:

SIGHUP = Signal(0x1)

SIGINT = Signal(0x2)

SIGQUIT = Signal(0x3)

SIGILL = Signal(0x4)

SIGTRAP = Signal(0x5)

SIGABRT = Signal(0x6)

SIGBUS = Signal(0x7)

SIGFPE = Signal(0x8)

SIGKILL = Signal(0x9)

SIGSEGV = Signal(0xb)

SIGPIPE = Signal(0xd)

SIGALRM = Signal(0xe)

SIGTERM = Signal(0xf)

Linux environment has a collection of 35 signals:

SIGABRT = Signal(0x6)  
SIGALRM = Signal(0xe)  
SIGBUS = Signal(0x7)  
SIGCHLD = Signal(0x11)  
SIGCLD = Signal(0x11)  
SIGCONT = Signal(0x12)  
SIGFPE = Signal(0x8)  
SIGHUP = Signal(0x1)  
SIGILL = Signal(0x4)  
SIGINT = Signal(0x2)  
SIGIO = Signal(0x1d)  
SIGIOT = Signal(0x6)  
SIGKILL = Signal(0x9)  
SIGPIPE = Signal(0xd)  
SIGPOLL = Signal(0x1d)  
SIGPROF = Signal(0x1b)  
SIGPWR = Signal(0x1e)  
SIGQUIT = Signal(0x3)  
SIGSEGV = Signal(0xb)  
SIGSTKFLT = Signal(0x10)  
SIGSTOP = Signal(0x13)  
SIGSYS = Signal(0x1f)  
SIGTERM = Signal(0xf)  
SIGTRAP = Signal(0x5)  
SIGTSTP = Signal(0x14)  
SIGTTIN = Signal(0x15)  
SIGTTOU = Signal(0x16)  
SIGUNUSED = Signal(0x1f)  
SIGURG = Signal(0x17)  
SIGUSR1 = Signal(0xa)  
SIGUSR2 = Signal(0xc)  
SIGVTALRM = Signal(0x1a)  
SIGWINCH = Signal(0x1c)  
SIGXCPU = Signal(0x18)  
SIGXFSZ = Signal(0x19)

The application would listen (or you can say intercept) for two signals: **SIGINT**, **SIGTERM**.

* **SIGINT** means is **Signal Interrupted**, it is sent when the user types the **INTR** character (e.g. **Ctrl-C**). It can be caught or ignored. The intention is to provide a mechanism for an orderly, graceful shutdown.
* The **SIGTERM** signal is a generic signal used to cause program termination. Unlike **SIGKILL**, this signal can be blocked, handled, and ignored. It is the normal way to politely ask a program to terminate. The intention is to kill the process, gracefully or not, but to first allow it a chance to cleanup.

The Notify() function allows you to "catch" operating system signals and write them to a channel:

package main

import (

"fmt"

"os"

"os/signal"

"syscall"

)

func main() {

// Set up channel on which to send signal notifications.

// We must use a buffered channel or risk missing the signal

// if we're not ready to receive when the signal is sent.

sigs := make(chan os.Signal, 1)

**signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM)**

// signal.Notify(c) // This operator will capture all signals

fmt.Println("Waiting for signals...")

// Block until any signal is received.

**sig := <-sigs**

fmt.Printf("Received signal: %s\n", sig)

// Close all external resources here

}

This template will work for a single long-running process:

package main

import (

"fmt"

"log"

"os"

"os/signal"

"syscall"

)

var shouldStop = false

func main() {

complete := make(chan struct{})

go func() {

signals := make(chan os.Signal, 1)

signal.Notify(signals, os.Interrupt, syscall.SIGTERM)

<-signals

shouldStop = true

close(complete)

}()

fmt.Println("We are starting!")

if err := myWorker(); err != nil {

// Error in our worker:

log.Fatalf("Worker error: %v", err)

}

<-complete

fmt.Println("We are stopping ...")

}

func myWorker() error {

// Open all resources here

for !shouldStop {

err := doSomethingRepeatedly()

if err != nil {

return err

}

}

// Close all resources here

return nil

}

func doSomethingRepeatedly() error {

// ...

return nil

}

The template is a bit more complicated for multiple workers. It requires context now:

package main

import (

"context"

"fmt"

"log"

"net/http"

"os"

"os/signal"

"sync"

"syscall"

"time"

)

func main() {

ctx, cancel := context.WithTimeout(context.Background(), 10\*time.Second)

var wg sync.WaitGroup

go func() {

signals := make(chan os.Signal, 1)

signal.Notify(signals, os.Interrupt, syscall.SIGTERM)

<-signals

cancel()

}()

wg.Add(1)

go func() {

fmt.Println("Worker: start")

if err := myWorker(ctx); err != nil {

cancel()

}

fmt.Println("Worker: stop")

wg.Done()

}()

wg.Add(1)

go func() {

fmt.Println("Server: start")

if err := startServer(ctx); err != nil {

cancel()

}

fmt.Println("Server: stop")

wg.Done()

}()

wg.Wait()

}

func myWorker(ctx context.Context) error {

var shouldStop = false

go func() {

<-ctx.Done()

shouldStop = true

}()

for !shouldStop {

err := doSomethingRepeatedly()

if err != nil {

return err

}

}

return nil

}

func startServer(ctx context.Context) error {

var srv http.Server

go func() {

<-ctx.Done() // Wait for the context to be done

// Shutdown the server

if err := srv.Shutdown(context.Background()); err != nil {

// Error from closing listeners, or context timeout:

log.Printf("HTTP server Shutdown: %v", err)

}

}()

if err := srv.ListenAndServe(); err != http.ErrServerClosed {

// Error starting or closing listener:

return fmt.Errorf("HTTP server ListenAndServe: %w", err)

}

return nil

}

func doSomethingRepeatedly() error {

// ...

return nil

}

// [Gracefully shutting down multiple workers in Go | Stephen AfamO's Blog](https://stephenafamo.com/blog/posts/gracefully-shutting-down-multiple-workers-in-go)

[Interfaces in Golang - GeeksforGeeks](https://www.geeksforgeeks.org/interfaces-in-golang/)

[The power of single-method interfaces in Go - Eli Bendersky's website (thegreenplace.net)](https://eli.thegreenplace.net/2023/the-power-of-single-method-interfaces-in-go/)

[The power of single-method interfaces in Go (golang.ch)](https://golang.ch/the-power-of-single-method-interfaces-in-go/)

[Golang Blog | Learn Go, Free Interctive Go Tutorials (learngolangonline.com)](https://learngolangonline.com/)

[Iterators in Go — Bitfield Consulting](https://bitfieldconsulting.com/posts/iterators)

[Go 101 -Go 101](https://go101.org/)

Iterators:

[A deep dive into the iter package | Medium](https://medium.com/@g.macmount/go-iterators-a-deep-dive-into-the-iter-package-329139000910" \l ":~:text=The iter package is a Go library that offers a)

[Understanding Go 1.23 iterators - DEV Community](https://dev.to/wmdanor/understanding-go-123-iterators-3ai1" \l ":~:text=Many people seem to be confused by newly added iterators)

[A look at iterators in Go (Golang) | Eureka Engineering (medium.com)](https://medium.com/eureka-engineering/a-look-at-iterators-in-go-f8e86062937c" \l ":~:text=An iterator is a higher-order function that takes a function)

[Understanding Go 1.23 iterators - DEV Community](https://dev.to/wmdanor/understanding-go-123-iterators-3ai1" \l ":~:text=Understanding Go 1.23 iterators. %23 go %23 programming %23 tutorial)

[iter package - iter - Go Packages](https://pkg.go.dev/iter)