# 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.

**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.

## Structure of the application

The workspace concept used in current versions of GO makes it much easier to structure large applications. Now, when starting a new project, create a new directory with the \*.work file in it:

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

Large programs should be divided into modules. First, we create a directory for the entire application with separate directories for each module:

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

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))

}

//[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".

#### 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")

}

…

## 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)