Go (programming language)

Go is a high-level, general-purpose programming language that is statically typed and compiled. It is known for the simplicity of its syntax and the efficiency of development that it enables through the inclusion of a large standard library supplying many needs for common projects.[13] It was designed at Google[14] in 2007 by Robert Griesemer, Rob Pike, and Ken Thompson, and publicly announced in November 2009.[4] It is syntactically similar to C, but also has garbage collection, structural typing,[8] and CSP-style concurrency.[15] It is often referred to as **Golang** to avoid ambiguity and because of its former domain name, `golang.org`, but its proper name is Go.[16]

There are two major implementations:

- The original, self-hosting[17] compiler toolchain, initially developed inside Google;[18]
- A frontend written in C++, called gofrontend,[19] originally a GCC frontend, providing gccgo, a GCC-based Go compiler;[20] later extended to also support LLVM, providing an LLVM-based Go compiler called gollvm.[21]

A third-party source-to-source compiler, GopherJS,[22] transpiles Go to JavaScript for front-end web development.

History

Go was designed at Google in 2007 to improve programming productivity in an era of multicore, networked machines and large codebases.[23] The designers wanted to address criticisms of other languages in use at Google, but keep their useful characteristics:[24]

- Static typing and run-time efficiency (like C)
- Readability and usability (like Python)[25]

Go	
Paradigm	Multi-paradigm: concurrent, imperative, functional,[1] object-oriented[2][3]
Designed by	Robert Griesemer Rob Pike Ken Thompson[4]
Developer	The Go Authors[5]
First appeared	November 10, 2009
Stable release	1.26.0[6] / 10 February 2026
Typing discipline	Inferred, static, strong,[7] structural,[8][9] nominal
Memory management	Garbage collection
Implementation language	Go, Assembly language (gc); C++ (gofrontend)
OS	DragonFly BSD, FreeBSD, Linux, macOS, NetBSD, OpenBSD,[10] Plan 9,[11] Solaris, Windows
License	3-clause BSD[5] + patent grant[12]
Filename extensions	.go
Website	go.dev (https://go.dev)
Major implementations	
gc, gofrontend, gold	
Influenced by	
C, Oberon-2, Limbo, Active Oberon, communicating sequential processes, Pascal,	

- High-performance networking and multiprocessing

Its designers were primarily motivated by their shared dislike of C++.[26][27][28]

Oberon, Smalltalk, Newsqueak, Modula-2, Alef, APL, BCPL, Modula, occam
Influenced
Crystal, V

Go was publicly announced in November 2009,[29] and version 1.0 was released in March 2012.[30][31] Go is widely used in production at Google[32] and in many other organizations and open-source projects.

In retrospect the Go authors judged Go to be successful due to the overall engineering work around the language, including the runtime support for the language's concurrency feature.

> Although the design of most languages concentrates on innovations in syntax, semantics, or typing, Go is focused on the software development process itself. ... The principal unusual property of the language itself—concurrency—addressed problems that arose with the proliferation of multicore CPUs in the 2010s. But more significant was the early work that established fundamentals for packaging, dependencies, build, test, deployment, and other workaday tasks of the software development world, aspects that are not usually foremost in language design.[33]

Branding and styling

The gopher mascot was introduced in 2009 for the open source launch of the language. Renée French, who had designed the rabbit mascot for Plan 9, adapted the gopher from an earlier WFMU T-shirt design.[34]

Go's mascot is a cartoon gopher.

In November 2016, the Go and Go Mono fonts were released by type designers Charles Bigelow and Kris Holmes specifically for use by the Go project. Go is a humanist sans-serif resembling Lucida Grande, and Go Mono is monospaced. Both fonts adhere to the WGL4 character set and were designed to be legible with a large x-height and distinct letterforms. Both Go and Go Mono adhere to the DIN 1450 standard by having a slashed zero, lowercase `i` with a tail, and an uppercase `I` with serifs.[35][36]

In April 2018, the original logo was redesigned by brand designer Adam Smith. The new logo is a modern, stylized GO slanting right with trailing streamlines. The gopher mascot remained the same.[37]

Generics

The lack of support for generic programming in initial versions of Go drew considerable criticism.[38] The designers expressed an openness to generic programming and noted that built-in functions *were* in fact type-generic, but are treated as special cases; Pike called this a weakness that might be changed at some point.[39] The Google team built at least one compiler for an experimental Go dialect with generics, but did not release it.[40]

In August 2018, the Go principal contributors published draft designs for generic programming and error handling and asked users to submit feedback.[41][42] However, the error handling proposal was eventually abandoned.[43]

In June 2020, a new draft design document[44] was published that would add the necessary syntax to Go for declaring generic functions and types. A code translation tool, *go2go*, was provided to allow users to try the new syntax, along with a generics-enabled version of the online Go Playground.[45]

Generics were finally added to Go in version 1.18 on March 15, 2022.[46]

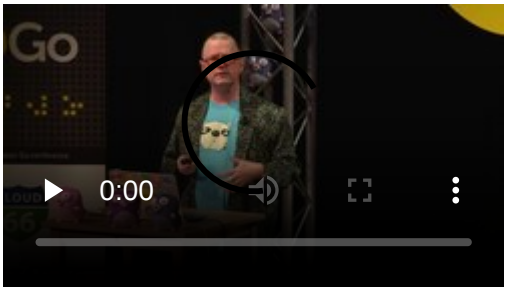
Versioning

Go 1 guarantees compatibility[47] for the language specification and major parts of the standard library. All versions up through the current Go 1.24 release[48] have maintained this promise.

Go uses a `go1.[major].[patch]` versioning format, such as `go1.24.0` and each major Go release is supported until there are two newer major releases. Unlike most software, Go calls the second number in a version the major, i.e., in `go1.24.0` the `24` is the major version.[49] This is because Go plans to never reach 2.0, prioritizing backwards compatibility over potential breaking changes.[50]

Design

Go is influenced by C (especially the Plan 9 dialect[51]), but with an emphasis on greater simplicity and safety. It consists of:



2015 lecture of Rob Pike (one of the Go creators)

- A syntax and environment adopting patterns more common in dynamic languages:[52]
  - Optional concise variable declaration and initialization through type inference (`x := 0` instead of `var x int = 0;` or `var x = 0;`)
  - Fast compilation[53]
  - Remote package management (`go get`)[54] and online package documentation[55]
- Distinctive approaches to particular problems:
  - Built-in concurrency primitives: light-weight processes (goroutines), channels, and the `select` statement
  - An interface system in place of virtual inheritance, and type embedding instead of non-virtual inheritance
  - A toolchain that, by default, produces statically linked native binaries without external Go dependencies
- A desire to keep the language specification simple enough to hold in a programmer's head,[56] in part by omitting features that are common in similar languages.
- 25 reserved words

Syntax

Go's syntax includes changes from C aimed at keeping code concise and readable. A combined declaration/initialization operator was introduced that allows the programmer to write `i := 3` or `s := "Hello, world!"`, without specifying the types of variables used. This contrasts with C++'s `int i = 3;` and `string s = "Hello, world!";` (though since C23 type inference has been supported using `auto`, like C++). Go also removes the requirement to use parentheses in if statement conditions.

Semicolons still terminate statements;[a] but are implicit when the end of a line occurs.[b]

Methods may return multiple values, and returning a `result, err` pair is the conventional way a method indicates an error to its caller in Go.[c] Go adds literal syntaxes for initializing struct parameters by name and for initializing maps and slices. As an alternative to C's three-statement `for` loop, Go's `range` expressions allow concise iteration over arrays, slices, strings, maps, and channels.[59]

Keywords

Go contains the following 25 keywords:[60]

- `break`
- `case`
- `chan`
- `const`
- `continue`
- `default`
- `defer`
- `else`
- `fallthrough`
- `for`
- `func`
- `go`
- `goto`
- `if`
- `import`
- `interface`
- `map`
- `package`
- `range`
- `return`
- `select`
- `struct`
- `switch`
- `type`
- `var`

Types

Go has a number of built-in types, including numeric ones (`byte`, `int64`, `float32`, etc.), Booleans, and byte strings (`string`). Strings are immutable; built-in operators and keywords (rather than functions) provide concatenation, comparison, and UTF-8 encoding/decoding.[61] Record types can be defined with the `struct` keyword.[62]

Go contains the following primitives:[63]

- `bool`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`

- `uint32`
- `int64`
- `uint64`
- `int`
- `uint`
- `uintptr`
- `float32`
- `float64`
- `complex64`
- `complex128`
- `string`

Note that `byte` is an alias for `uint8` and `rune` is an alias for `int32`.

For each type `T` and each non-negative integer constant `n`, there is an array type denoted `[n]T`; arrays of differing lengths are thus of different types. Dynamic arrays are available as "slices", denoted `[]T` for some type `T` (compare to other languages like C/C++ and Java, where instead the arrays are denoted `T[]`) These have a length and a *capacity* specifying when new memory needs to be allocated to expand the array. Several slices may share their underlying memory.[39][64][65]

Pointers are available for all types, and the pointer-to-`T` type is denoted `*T` (similar to Rust; compare to other languages like C/C++ and C#, where pointers are denoted `T*`). Address-taking and indirection use the `&` and `*` operators, as in C, or happen implicitly through the method call or attribute access syntax.[66][67] There is no pointer arithmetic,[d] except via the special `unsafe.Pointer` type in the standard library.[68]

For a pair of types `K`, `V`, the type `map[K]V` is the type mapping type-`K` keys to type-`V` values, which can be thought of as equivalent to `Map<K, V>` in other languages. The Go Programming Language specification does not give any performance guarantees or implementation requirements for map types, though it is usually implemented as a hash table (equivalent to `HashMap<K, V>` in other languages). Hash tables are built into the language, with special syntax and built-in functions. `chan T` is a *channel* that allows sending values of type `T` between concurrent Go processes.[69]

Aside from its support for interfaces, Go's type system is nominal: the `type` keyword can be used to define a new *named type*, which is distinct from other named types that have the same layout (in the case of a `struct`, the same members in the same order). Some conversions between types (e.g., between the various integer types) are pre-defined and adding a new type may define additional conversions, but conversions between named types must always be invoked explicitly.[70] For example, the `type` keyword can be used to define a type for IPv4 addresses, based on 32-bit unsigned integers as follows:

```
type ipv4addr uint32
```

With this type definition, `ipv4addr(x)` interprets the `uint32` value `x` as an IP address. Simply assigning `x` to a variable of type `ipv4addr` is a type error.[71]

Constant expressions may be either typed or "untyped"; they are given a type when assigned to a typed variable if the value they represent passes a compile-time check.[72]

Function types are indicated by the `func` keyword; they take zero or more parameters and return zero or more values, all of which are typed. The parameter and return values determine a function type; thus, `func(string, int32) (int, error)` is the type of functions that take a `string` and a 32-bit

signed integer, and return a signed integer (of default width) and a value of the built-in interface type `error`.[73]

Any named type has a method set associated with it. The IP address example above can be extended with a method for checking whether its value is a known standard:

```
// ZeroBroadcast reports whether addr is 255.255.255.255.
func (addr ipv4addr) ZeroBroadcast() bool {
    return addr == 0xFFFFFFFF
}
```

Due to nominal typing, this method definition adds a method to `ipv4addr`, but not on `uint32`. While methods have special definition and call syntax, there is no distinct method type.[74]

Interface system

Go provides two features that replace class inheritance.

The first is *embedding*, which can be viewed as an automated form of composition.[75]

The second are its *interfaces*, which provides runtime polymorphism.[76]:266 Interfaces are a class of types and provide a limited form of structural typing in the otherwise nominal type system of Go. An object which is of an interface type is also of another type, much like C++ objects being simultaneously of a base and derived class. The design of Go interfaces was inspired by protocols from the Smalltalk programming language.[77] Multiple sources use the term duck typing when describing Go interfaces.[78][79] Although the term duck typing is not precisely defined and therefore not wrong, it usually implies that type conformance is not statically checked. Because conformance to a Go interface is checked statically by the Go compiler (except when performing a type assertion), the Go authors prefer the term *structural typing*.[80]

The definition of an interface type lists required methods by name and type. Any object of type T for which functions exist matching all the required methods of interface type I is an object of type I as well. The definition of type T need not (and cannot) identify type I. For example, if `Shape`, `Square` and `Circle` are defined as

```
import "math"

type Shape interface {
    Area() float64
}

// Note: no "implements" declaration
type Square struct {
    side float64
}

func (sq Square) Area() float64 {
    return sq.side * sq.side
}

// No "implements" declaration here either
type Circle struct {
    radius float64
}

func (c Circle) Area() float64 {
```

```
    return math.Pi * math.Pow(c.radius, 2)
}
```

then both a `Square` and a `Circle` are implicitly a `Shape` and can be assigned to a `Shape`-typed variable.[76]:263–268 In formal language, Go's interface system provides structural rather than nominal typing. Interfaces can embed other interfaces with the effect of creating a combined interface that is satisfied by exactly the types that implement the embedded interface and any methods that the newly defined interface adds.[76]:270

The Go standard library uses interfaces to provide genericity in several places, including the input/output system that is based on the concepts of `Reader` and `Writer`.[76]:282–283

Besides calling methods via interfaces, Go allows converting interface values to other types with a run-time type check. The language constructs to do so are the *type assertion*,[81] which checks against a single potential type:

```
var shp Shape = Square{5}
square, ok := shp.(Square) // Asserts Square type on shp, should work
if ok {
    fmt.Printf("%#v\n", square)
} else {
    fmt.Println("Can't print shape as Square")
}
```

and the *type switch*,[82] which checks against multiple types:

```
func (sq Square) Diagonal() float64 { return sq.side * math.Sqrt2 }
func (c Circle) Diameter() float64 { return 2 * c.radius }

func LongestContainedLine(shp Shape) float64 {
    switch v := shp.(type) {
    case Square:
        return v.Diagonal() // Or, with type assertion, shp.(Square).Diagonal()
    case Circle:
        return v.Diameter() // Or, with type assertion, shp.(Circle).Diameter()
    default:
        return 0 // In practice, this should be handled with errors
    }
}
```

The *empty interface* `interface{}` is an important base case because it can refer to an item of *any* concrete type. It is similar to the `Object` class in Java or C# or `void*` in C or `Any` in C++ and Rust and is satisfied by any type, including built-in types like `int`.[76]:284 Code using the empty interface cannot simply call methods (or built-in operators) on the referred-to object, but it can store the `interface{}` value, try to convert it to a more useful type via a type assertion or type switch, or inspect it with Go's `reflect` package.[83] Because `interface{}` can refer to any value, it is a limited way to escape the restrictions of static typing, like `void*` in C but with additional run-time type checks.

The `interface{}` type can be used to model structured data of any arbitrary schema in Go, such as JSON or YAML data, by representing it as a `map[string]interface{}` (map of string to empty interface). This recursively describes data in the form of a dictionary with string keys and values of any type.[84]

Interface values are implemented using pointer to data and a second pointer to run-time type information.[85] Like some other types implemented using pointers in Go, interface values are `nil` if uninitialized.[86]

Generic code using parameterized types

Since version 1.18, Go supports generic code using parameterized types.[87]

Functions and types now have the ability to be generic using type parameters. These type parameters are specified within square brackets, right after the function or type name.[88] The compiler transforms the generic function or type into non-generic by substituting *type arguments* for the type parameters provided, either explicitly by the user or type inference by the compiler.[89] This transformation process is referred to as type instantiation.[90]

Interfaces now can define a set of types (known as type set) using | (Union) operator, as well as a set of methods. These changes were made to support type constraints in generics code. For a generic function or type, a constraint can be thought of as the type of the type argument: a meta-type. This new `~T` syntax will be the first use of ~ as a token in Go. `~T` means the set of all types whose underlying type is `T`.[91]

```
type Number interface {
    ~int | ~float64 | ~float32 | ~int32 | ~int64
}

func Add[T Number](nums ...T) T {
    var sum T
    for _, v := range nums {
        sum += v
    }
    return sum
}

func main() {
    add := Add[int]              // Type instantiation
    println(add(1, 2, 3, 4, 5)) // 15

    res := Add(1.1, 2.2, 3.3, 4.4, 5.5) // Type Inference
    println(res)                        // +1.650000e+001
}
```

Enumerated types

Go uses the `iota` keyword to create enumerated constants.[92][93]

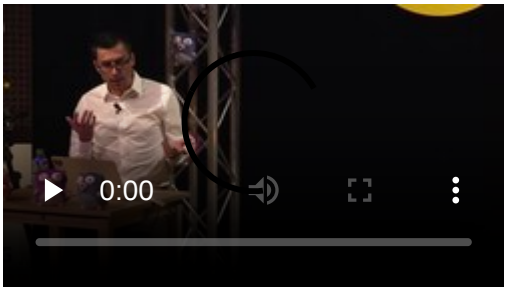
```
type ByteSize int

const (
    _           = iota // ignore first value by assigning to blank identifier; 0
    KB ByteSize = 1 << (10 * iota) // 1 << (10 * 1) == 1 << 10 == 1024; in binary 10000000000
    MB // 1 << (10 * 2) == 1048576;                                    in binary
100000000000000000000
    GB // 1 << (10 * 3) == 1073741824;                                 in binary
1000000000000000000000000000000
)
```

Package system

In Go's package system, each package has a path (e.g., `"compress/bzip2"` or `"golang.org/x/net/html"`) and a name (e.g., `bzip2` or `html`). By default other packages' definitions must *always* be prefixed with the other package's name. However the name used can be changed from the package name, and if imported as `_`, then no package prefix is required. Only the *capitalized* names from other packages are accessible: `io.Reader` is public but `bzip2.reader` is not.[94] The `go get` command can retrieve packages stored in a remote repository[95] and developers are encouraged to develop packages inside a base path corresponding to a source repository (such as example.com/user_name/package_name) to reduce the likelihood of name collision with future additions to the standard library or other external libraries.[96]

Concurrency: goroutines and channels

The Go language has built-in facilities, as well as library support, for writing concurrent programs. The runtime is asynchronous: program execution that performs, for example, a network read will be suspended until data is available to process, allowing other parts of the program to perform other work. This is built into the runtime and does not require any changes in program code. The go runtime also automatically schedules concurrent operations (goroutines) across multiple CPUs; this can achieve parallelism for a properly written program.[97]



DotGo 2015 - Matt Aimonetti - Applied concurrency in Go

The primary concurrency construct is the *goroutine*, a type of green thread.[98]:280–281 A function call prefixed with the `go` keyword starts a function in a new goroutine. The language specification does not specify how goroutines should be implemented, but current implementations multiplex a Go process's goroutines onto a smaller set of operating-system threads, similar to the scheduling performed in Erlang and Haskell's Glasgow Haskell Compiler (GHC) runtime implementation.[99]:10

While a standard library package featuring most of the classical concurrency control structures (mutex locks, etc.) is available,[99]:151–152 idiomatic concurrent programs instead prefer *channels*, which send messages between goroutines.[100] Optional buffers store messages in FIFO order[101]:43 and allow sending goroutines to proceed before their messages are received.[98]:233

Channels are typed, so that a channel of type `chan T` can only be used to transfer messages of type `T`. Special syntax is used to operate on them; `<-ch` is an expression that causes the executing goroutine to block until a value comes in over the channel `ch`, while `ch <- x` sends the value `x` (possibly blocking until another goroutine receives the value). The built-in `switch`-like `select` statement can be used to implement non-blocking communication on multiple channels; see below for an example. Go has a memory model describing how goroutines must use channels or other operations to safely share data.[102]

The existence of channels does not by itself set Go apart from actor model-style concurrent languages like Erlang, where messages are addressed directly to actors (corresponding to goroutines). In the actor model, channels are themselves actors, therefore addressing a channel just means to address an actor. The

actor style can be simulated in Go by maintaining a one-to-one correspondence between goroutines and channels, but the language allows multiple goroutines to share a channel or a single goroutine to send and receive on multiple channels.[99]:147

From these tools one can build concurrent constructs like worker pools, pipelines (in which, say, a file is decompressed and parsed as it downloads), background calls with timeout, "fan-out" parallel calls to a set of services, and others.[103] Channels have also found uses further from the usual notion of interprocess communication, like serving as a concurrency-safe list of recycled buffers,[104] implementing coroutines (which helped inspire the name *goroutine*),[105] and implementing iterators.[106]

Concurrency-related structural conventions of Go (channels and alternative channel inputs) are derived from Tony Hoare's communicating sequential processes model. Unlike previous concurrent programming languages such as Occam or Limbo (a language on which Go co-designer Rob Pike worked),[107] Go does not provide any built-in notion of safe or verifiable concurrency.[108] While the communicating-processes model is favored in Go, it is not the only one: all goroutines in a program share a single address space. This means that mutable objects and pointers can be shared between goroutines; see § Lack of data race safety, below.

Suitability for parallel programming

Although Go's concurrency features are not aimed primarily at parallel processing,[97] they can be used to program shared-memory multi-processor machines. Various studies have been done into the effectiveness of this approach.[109] One of these studies compared the size (in lines of code) and speed of programs written by a seasoned programmer not familiar with the language and corrections to these programs by a Go expert (from Google's development team), doing the same for Chapel, Cilk and Intel TBB. The study found that the non-expert tended to write divide-and-conquer algorithms with one `go` statement per recursion, while the expert wrote distribute-work-synchronize programs using one goroutine per processor core. The expert's programs were usually faster, but also longer.[110]

Lack of data race safety

Go's approach to concurrency can be summarized as "don't communicate by sharing memory; share memory by communicating".[111] There are no restrictions on how goroutines access shared data, making data races possible. Specifically, unless a program explicitly synchronizes via channels or other means, writes from one goroutine might be partly, entirely, or not at all visible to another, often with no guarantees about ordering of writes.[108] Furthermore, Go's *internal data structures* like interface values, slice headers, hash tables, and string headers are not immune to data races, so type and memory safety can be violated in multithreaded programs that modify shared instances of those types without synchronization.[112][113] Instead of language support, safe concurrent programming thus relies on conventions; for example, Chisnall recommends an idiom called "aliases xor mutable", meaning that passing a mutable value (or pointer) over a channel signals a transfer of ownership over the value to its receiver.[99]:155 The gc toolchain has an optional data race detector that can check for unsynchronized access to shared memory during runtime since version 1.1,[114] additionally a best-effort race detector is also included by default since version 1.6 of the gc runtime for access to the `map` data type.[115]

Binaries

The linker in the gc toolchain creates statically linked binaries by default; therefore all Go binaries include the Go runtime.[116][117]

Omissions

Go deliberately omits certain features common in other languages, including (implementation) inheritance, assertions,[e] pointer arithmetic,[d] implicit type conversions, untagged unions,[f] and tagged unions.[g] The designers added only those facilities that all three agreed on.[120]

Of the omitted language features, the designers explicitly argue against assertions and pointer arithmetic, while defending the choice to omit type inheritance as giving a more useful language, encouraging instead the use of interfaces to achieve dynamic dispatch[h] and composition to reuse code. Composition and delegation are in fact largely automated by `struct` embedding; according to researchers Schmager *et al.*, this feature "has many of the drawbacks of inheritance: it affects the public interface of objects, it is not fine-grained (i.e, no method-level control over embedding), methods of embedded objects cannot be hidden, and it is static", making it "not obvious" whether programmers will overuse it to the extent that programmers in other languages are reputed to overuse inheritance.[75]

Exception handling was initially omitted in Go due to lack of a "design that gives value proportionate to the complexity".[121] An exception-like `panic`/`recover` mechanism that avoids the usual `try-catch` control structure was proposed[122] and released in the March 30, 2010 snapshot.[123] The Go authors advise using it for unrecoverable errors such as those that should halt an entire program or server request, or as a shortcut to propagate errors up the stack within a package.[124][125] Across package boundaries, Go includes a canonical error type, and multi-value returns using this type are the standard idiom.[4]

Style

The Go authors put substantial effort into influencing the style of Go programs:

- Indentation, spacing, and other surface-level details of code are automatically standardized by the `gofmt` tool. It uses tabs for indentation and blanks for alignment. Alignment assumes that an editor is using a fixed-width font.[126] `golint` does additional style checks automatically, but has been deprecated and archived by the Go maintainers.[127]
- Tools and libraries distributed with Go suggest standard approaches to things like API documentation (`godoc`),[128] testing (`go test`), building (`go build`), package management (`go get`), and so on.
- Go enforces rules that are recommendations in other languages, for example banning cyclic dependencies, unused variables[129] or imports,[130] and implicit type conversions.
- The *omission* of certain features (for example, functional-programming shortcuts like `map` and Java-style `try`/`finally` blocks) tends to encourage a particular explicit, concrete, and imperative programming style.
- On day one the Go team published a collection of Go idioms,[128] and later also collected code review comments,[131] talks,[132] and official blog posts[133] to teach Go style and coding philosophy.

Tools

The main Go distribution includes tools for building, testing, and analyzing code:

- `go build`, which builds Go binaries using only information in the source files themselves, no separate makefiles
- `go test`, for unit testing and microbenchmarks as well as fuzzing
- `go fmt`, for formatting code
- `go install`, for retrieving and installing remote packages
- `go vet`, a static analyzer looking for potential errors in code
- `go run`, a shortcut for building and executing code
- `go doc`, for displaying documentation
- `go generate`, a standard way to invoke code generators
- `go mod`, for creating a new module, adding dependencies, upgrading dependencies, etc.
- `go tool`, for invoking developer tools (added in Go version 1.24)

It also includes profiling and debugging support, fuzzing capabilities to detect bugs, runtime instrumentation (for example, to track garbage collection pauses), and a data race detector.

Another tool maintained by the Go team but is not included in Go distributions is `gopls`, a language server that provides IDE features such as intelligent code completion to Language Server Protocol compatible editors.[134]

An ecosystem of third-party tools adds to the standard distribution, such as `gocode`, which enables code autocompletion in many text editors, `goimports`, which automatically adds/removes package imports as needed, and `errcheck`, which detects code that might unintentionally ignore errors.

Examples

Hello world

```
package main

import "fmt"

func main() {
    fmt.Println("hello world")
}
```

where "fmt" is the package for *formatted I/O*, similar to C's `<stdio.h>` or C++ `<print>`.[135]

Concurrency

The following simple program demonstrates Go's concurrency features to implement an asynchronous program. It launches two lightweight threads ("goroutines"): one waits for the user to type some text, while the other implements a timeout. The `select` statement waits for either of these goroutines to send a message to the main routine, and acts on the first message to arrive (example adapted from David Chisnall's book).[99]:152

```
package main

import (
    "fmt"
    "time"
)

func readword(ch chan string) {
    fmt.Println("Type a word, then hit Enter.")
    var word string
    fmt.Scanf("%s", &word)
    ch <- word
}

func timeout(t chan bool) {
    time.Sleep(5 * time.Second)
    t <- false
}

func main() {
    t := make(chan bool)
    go timeout(t)

    ch := make(chan string)
    go readword(ch)

    select {
    case word := <-ch:
        fmt.Println("Received", word)
    case <-t:
        fmt.Println("Timeout.")
    }
}
```

Testing

The testing package provides support for automated testing of go packages.[136] Target function example:

```
func ExtractUsername(email string) string {
    at := strings.Index(email, "@")
    return email[:at]
}
```

Test code (note that **assert** keyword is missing in Go; tests live in <filename>_test.go at the same package):

```
import (
    "testing"
)

func TestExtractUsername(t *testing.T) {
    t.Run("withoutDot", func(t *testing.T) {
        username := ExtractUsername("r@google.com")
        if username != "r" {
            t.Fatalf("Got: %v\n", username)
```

```
        }
    })

    t.Run("withDot", func(t *testing.T) {
        username := ExtractUsername("jonh.smith@example.com")
        if username != "jonh.smith" {
            t.Fatalf("Got: %v\n", username)
        }
    })
}
```

It is possible to run tests in parallel.

Web app

The net/http (https://pkg.go.dev/net/http)[137] package provides support for creating web applications.

This example would show "Hello world!" when localhost:8080 is visited.

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func helloFunc(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello world!")
}

func main() {
    http.HandleFunc("/", helloFunc)
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Applications

Go has found widespread adoption in various domains due to its robust standard library and ease of use.[138]

Popular applications include:

- Caddy — a web server that automates the process of setting up HTTPS[139]
- Docker — a platform for containerization, aiming to ease the complexities of software development and deployment[140]
- Kubernetes — automates the deployment, scaling, and management of containerized applications[141]
- CockroachDB — a distributed SQL database engineered for scalability and strong consistency[142]
- Hugo — a static site generator that prioritizes speed and flexibility, allowing developers to create websites efficiently[143]

Reception

The interface system, and the deliberate omission of inheritance, were praised by Michele Simionato, who likened these characteristics to those of Standard ML, calling it "a shame that no popular language has followed [this] particular route".[144]

Dave Astels at Engine Yard wrote in 2009:[145]

> Go is extremely easy to dive into. There are a minimal number of fundamental language concepts and the syntax is clean and designed to be clear and unambiguous. Go *is* still experimental and still a little rough around the edges.

Go was named Programming Language of the Year by the TIOBE Programming Community Index in its first year, 2009, for having a larger 12-month increase in popularity (in only 2 months, after its introduction in November) than any other language that year, and reached 13th place by January 2010,[146] surpassing established languages like Pascal. By June 2015, its ranking had dropped to below 50th in the index, placing it lower than COBOL and Fortran.[147] But as of January 2017, its ranking had surged to 13th, indicating significant growth in popularity and adoption. Go was again awarded TIOBE Programming Language of the Year in 2016.[148]

Bruce Eckel has stated:[149]

> The complexity of C++ (even more complexity has been added in the new C++), and the resulting impact on productivity, is no longer justified. All the hoops that the C++ programmer had to jump through in order to use a C-compatible language make no sense anymore -- they're just a waste of time and effort. Go makes much more sense for the class of problems that C++ was originally intended to solve.

A 2011 evaluation of the language and its `gc` implementation in comparison to C++ (GCC), Java and Scala by a Google engineer found:

> Go offers interesting language features, which also allow for a concise and standardized notation. The compilers for this language are still immature, which reflects in both performance and binary sizes.
>
> —R. Hundt[150]

The evaluation got a rebuttal from the Go development team. Ian Lance Taylor, who had improved the Go code for Hundt's paper, had not been aware of the intention to publish his code, and says that his version was "never intended to be an example of idiomatic or efficient Go"; Russ Cox then optimized the Go code, as well as the C++ code, and got the Go code to run almost as fast as the C++ version and more than an order of magnitude faster than the code in the paper.[151]

- Go's `nil` combined with the lack of algebraic types leads to difficulty handling failures and base cases.[152][153]
- Go has been criticized for focusing on simplicity of implementation rather than correctness and flexibility; as an example, the language uses POSIX file semantics on all platforms, and therefore provides incorrect information on platforms such as Windows (which do not follow the aforementioned standard).[154][155]
- A study showed that it is as easy to make concurrency bugs with message passing as with shared memory, sometimes even more.[156]

Naming dispute

On November 10, 2009, the day of the general release of the language, Francis McCabe, developer of the Go! programming language (note the exclamation point), requested a name change of Google's language to prevent confusion with his language, which he had spent 10 years developing.[157] McCabe raised concerns that "the 'big guy' will end up steam-rollering over" him, and this concern resonated with the more than 120 developers who commented on Google's official issues thread saying they should change the name, with some[158] even saying the issue contradicts Google's motto of: Don't be evil.[159]

On October 12, 2010, the filed public issue ticket was closed by Google developer Russ Cox (@rsc) with the custom label "Unfortunate" accompanied by the following comment:

> "There are many computing products and services named Go. In the 11 months since our release, there has been minimal confusion of the two languages."[159]

See also

Free and open-source software portal

- Fat pointer
- Fyne (software) — widget toolkit for creating GUIs with Go
- Comparison of programming languages

Notes

a. But "To allow complex statements to occupy a single line, a semicolon may be omitted before a closing) or }".[57]
b. "if the newline comes after a token that could end a statement, [the lexer will] insert a semicolon".[58]
c. Usually, exactly one of the result and error values has a value other than the type's zero value; sometimes both do, as when a read or write can only be partially completed, and sometimes neither, as when a read returns 0 bytes. See Semipredicate problem: Multivalued return.
d. Language FAQ "Why is there no pointer arithmetic? Safety ... never derive an illegal address that succeeds incorrectly ... using array indices can be as efficient as ... pointer

arithmetic ... simplify the implementation of the garbage collector...."[4]

e. Language FAQ "Why does Go not have assertions? ...our experience has been that programmers use them as a crutch to avoid thinking about proper error handling and reporting...."[4]
f. Language FAQ "Why are there no untagged unions...? [they] would violate Go's memory safety guarantees."[4]
g. Language FAQ "Why does Go not have variant types? ... We considered [them but] they overlap in confusing ways with interfaces.... [S]ome of what variant types address is already covered, ... although not as elegantly."[4] (The tag of an interface type[118] is accessed with a type assertion[119]).
h. Questions "How do I get dynamic dispatch of methods?" and "Why is there no type inheritance?" in the language FAQ.[4]

References

1. "Codewalk: First-Class Functions in Go" (https://go.dev/doc/codewalk/functions/). "Go supports first class functions, higher-order functions, user-defined function types, function literals, closures, and multiple return values. This rich feature set supports a functional programming style in a strongly typed language."
2. "Is Go an object-oriented language?" (https://golang.org/doc/faq#Is_Go_an_object-oriented_language). Retrieved April 13, 2019. "Although Go has types and methods and allows an object-oriented style of programming, there is no type hierarchy."
3. "Go: code that grows with grace" (https://talks.golang.org/2012/chat.slide#5). Retrieved June 24, 2018. "Go is Object Oriented, but not in the usual way."
4. "Language Design FAQ" (https://go.dev/doc/faq). *The Go Programming Language*. January 16, 2010. Archived (https://web.archive.org/web/20250728222723/https://go.dev/doc/faq) from the original on July 28, 2025. Retrieved February 27, 2010.
5. "Text file LICENSE" (https://go.dev/LICENSE). *The Go Programming Language*. Archived (https://web.archive.org/web/20250716050530/https://go.dev/LICENSE) from the original on July 16, 2025. Retrieved October 5, 2012.
6. "Release 1.26.0" (https://github.com/golang/go/releases/tag/go1.26.0). February 10, 2026. Retrieved February 13, 2026.
7. "The Go Programming Language Specification" (https://go.dev/ref/spec#Introduction). *The Go Programming Language*. Archived (https://web.archive.org/web/20250813094237/https://go.dev/ref/spec#Introduction) from the original on August 13, 2025.
8. "Why doesn't Go have "implements" declarations?" (https://go.dev/doc/faq#implements_interface). *The Go Programming Language*. Archived (https://web.archive.org/web/20250728222723/https://go.dev/doc/faq#implements_interface) from the original on July 28, 2025. Retrieved October 1, 2015.
9. Pike, Rob [@rob_pike] (December 22, 2014). "Go has structural typing, not duck typing. Full interface satisfaction is checked and required" (https://web.archive.org/web/20220407025913/https://twitter.com/rob_pike/status/546973312543227904) (Tweet). Archived from the original (https://x.com/rob_pike/status/546973312543227904) on April 7, 2022. Retrieved March 13, 2016 – via Twitter.
10. "lang/go: go-1.4" (http://ports.su/lang/go). *OpenBSD ports*. December 23, 2014. Retrieved January 19, 2015.
11. "Go Porting Efforts" (http://go-lang.cat-v.org/os-ports). *Go Language Resources*. cat-v. January 12, 2010. Retrieved January 18, 2010.

12. "Additional IP Rights Grant" (https://go.dev/PATENTS). *The Go Programming Language*. Archived (https://web.archive.org/web/20250330185005/https://go.dev/PATENTS) from the original on March 30, 2025. Retrieved October 5, 2012.
13. "Go Introduction" (https://www.w3schools.com/go/go_introduction.php). *www.w3schools.com*. Retrieved November 23, 2024.
14. Kincaid, Jason (November 10, 2009). "Google's Go: A New Programming Language That's Python Meets C++" (https://techcrunch.com/2009/11/10/google-go-language/). *TechCrunch*. Retrieved January 18, 2010.
15. Metz, Cade (May 5, 2011). "Google Go boldly goes where no code has gone before" (https://www.theregister.com/2011/05/05/google_go/). *The Register*. Archived (https://web.archive.org/web/20250728222723/https://go.dev/doc/faq) from the original on July 28, 2025.
16. "Is the language called Go or Golang?" (https://go.dev/doc/faq#go_or_golang). Retrieved March 16, 2022. "The language is called Go."
17. "Go 1.5 Release Notes" (https://golang.org/doc/go1.5#implementation). Retrieved January 28, 2016. "The compiler and runtime are now implemented in Go and assembler, without C."
18. "The Go programming language" (https://github.com/golang/go). *GitHub*. Retrieved November 1, 2024.
19. "gofrontend" (https://github.com/golang/gofrontend). *GitHub*. Retrieved November 1, 2024.
20. "gccgo" (https://gcc.gnu.org/onlinedocs/gcc-14.2.0/gccgo/). Retrieved November 1, 2024. "gccgo, the GNU compiler for the Go programming language"
21. "Gollvm" (https://go.googlesource.com/gollvm/). Retrieved November 1, 2024. "Gollvm is an LLVM-based Go compiler."
22. "A compiler from Go to JavaScript for running Go code in a browser: Gopherjs/Gopherjs" (https://github.com/gopherjs/gopherjs). *GitHub*. Archived (https://web.archive.org/web/20231212143621/https://github.com/gopherjs/gopherjs) from the original on December 12, 2023.
23. "Go at Google: Language Design in the Service of Software Engineering" (https://talks.golang.org/2012/splash.article). Retrieved October 8, 2018.
24. Pike, Rob (April 28, 2010). "Another Go at Language Design" (http://www.stanford.edu/class/ee380/Abstracts/100428.html). *Stanford EE Computer Systems Colloquium*. Stanford University. Video available (https://www.youtube.com/watch?v=7VcArS4Wpqk).
25. "Frequently Asked Questions (FAQ) - The Go Programming Language" (https://golang.org/doc/faq#different_syntax). *The Go Programming Language*. Retrieved February 26, 2016.
26. Binstock, Andrew (May 18, 2011). "Dr. Dobb's: Interview with Ken Thompson" (https://web.archive.org/web/20130105013259/https://www.drdobbs.com/open-source/interview-with-ken-thompson/229502480). *Dr. Dobb's*. Archived from the original (http://www.drdobbs.com/open-source/interview-with-ken-thompson/229502480) on January 5, 2013. Retrieved February 7, 2014.
27. Pike, Rob (2012). "Less is exponentially more" (http://commandcenter.blogspot.mx/2012/06/less-is-exponentially-more.html).
28. Griesemer, Robert (2015). "The Evolution of Go" (https://talks.golang.org/2015/gophercon-goevolution.slide#4).
29. Griesemer, Robert; Pike, Rob; Thompson, Ken; Taylor, Ian; Cox, Russ; Kim, Jini; Langley, Adam. "Hey! Ho! Let's Go!" (https://opensource.googleblog.com/2009/11/hey-ho-lets-go.html). *Google Open Source*. Retrieved May 17, 2018.

30. Shankland, Stephen (March 30, 2012). "Google's Go language turns one, wins a spot at YouTube: The lower-level programming language has matured enough to sport the 1.0 version number. And it's being used for real work at Google" (https://www.cnet.com/news/googles-go-language-turns-one-wins-a-spot-at-youtube/). News. *CNet*. CBS Interactive Inc. Retrieved August 6, 2017. "Google has released version 1 of its Go programming language, an ambitious attempt to improve upon giants of the lower-level programming world such as C and C++."
31. "Release History" (https://golang.org/doc/devel/release.html). *The Go Programming Language*.
32. "Go FAQ: Is Google using Go internally?" (https://golang.org/doc/faq#internal_usage). Retrieved March 9, 2013.
33. The Go Programming Language and Environment. Communications of the ACM. https://dl.acm.org/doi/pdf/10.1145/3488716
34. "The Go Gopher - The Go Programming Language" (https://go.dev/blog/gopher). *go.dev*. Retrieved February 9, 2023.
35. "Go fonts" (https://blog.golang.org/go-fonts). Go. November 16, 2016. Retrieved March 12, 2019.
36. "Go Font TTFs" (https://github.com/golang/image/tree/master/font/gofont/ttfs). *GitHub*. Retrieved April 2, 2019.
37. "Go's New Brand" (https://blog.golang.org/go-brand). *The Go Blog*. Retrieved November 9, 2018.
38. Merrick, Alice (March 9, 2021). "Go Developer Survey 2020 Results" (https://go.dev/blog/survey2020-results#:~:text=Among%20the%2026%25%20of%20respondents%20who%20said%20Go%20lacks%20language%20features%20they%20need%2C%2088%25%20selected%20generics%20as%20a%20critical%20missing%20feature.). *Go Programming Language*. Retrieved March 16, 2022.
39. Pike, Rob (September 26, 2013). "Arrays, slices (and strings): The mechanics of 'append' " (https://blog.golang.org/slices). *The Go Blog*. Retrieved March 7, 2015.
40. "E2E: Erik Meijer and Robert Griesemer" (http://channel9.msdn.com/Blogs/Charles/Erik-Meijer-and-Robert-Griesemer-Go). *Channel 9*. Microsoft. May 7, 2012.
41. "Go 2 Draft Designs" (https://go.googlesource.com/proposal/+/master/design/go2draft.md). Retrieved September 12, 2018.
42. "The Go Blog: Go 2 Draft Designs" (https://blog.golang.org/go2draft). August 28, 2018.
43. "Proposal: A built-in Go error check function, "try" " (https://github.com/golang/go/issues/32437). *Go repository on GitHub*. Retrieved March 16, 2022.
44. "Type Parameters — Draft Design" (https://go.googlesource.com/proposal/+/refs/heads/master/design/go2draft-type-parameters.md). *go.googlesource.com*.
45. "Generics in Go" (https://bitfieldconsulting.com/golang/generics). *bitfieldconsulting.com*. December 17, 2021.
46. "Go 1.18 is released!" (https://go.dev/blog/go1.18). *Go Programming Language*. March 15, 2022. Retrieved March 16, 2022.
47. "Go 1 and the Future of Go Programs" (https://golang.org/doc/go1compat). *The Go Programming Language*.
48. "Go 1.24 Release Notes" (https://go.dev/doc/go1.24). *The Go Programming Language*.
49. "Release History" (https://golang.org/doc/devel/release.html#policy). *The Go Programming Language*.
50. "Backward Compatibility, Go 1.21, and Go 2" (https://go.dev/blog/compat). *The Go Programming Language*.
51. "A Quick Guide to Go's Assembler" (https://go.dev/doc/asm). *go.dev*. Retrieved December 31, 2021.

52. Pike, Rob (November 10, 2009). "The Go Programming Language" (https://www.youtube.com/watch?v=rKnDgT73v8s). YouTube. Retrieved July 1, 2011.
53. Pike, Rob (November 10, 2009). *The Go Programming Language* (https://www.youtube.com/watch?v=rKnDgT73v8s#t=8m53) (flv) (Tech talk). Google. Event occurs at 8:53.
54. "Download and install packages and dependencies" (https://golang.org/cmd/go/#hdr-Download_and_install_packages_and_dependencies). See godoc.org (http://godoc.org) for addresses and documentation of some packages.
55. "GoDoc" (http://godoc.org). *godoc.org*.
56. Pike, Rob. "The Changelog" (https://web.archive.org/web/20131020101046/http://5by5.tv/changelog/100) (Podcast). Archived from the original (http://5by5.tv/changelog/100) on October 20, 2013. Retrieved October 7, 2013.
57. "Go Programming Language Specification, §Semicolons" (https://golang.org/ref/spec#Semicolons). *The Go Programming Language*.
58. "Effective Go, §Semicolons" (https://golang.org/doc/effective_go.html#semicolons). *The Go Programming Language*.
59. "The Go Programming Language Specification" (https://golang.org/ref/spec#For_statements). *The Go Programming Language*.
60. "Keywords and Identifiers in Go" (https://go101.org/article/keywords-and-identifiers.html). *go101.org*. Retrieved October 10, 2025.
61. Pike, Rob (October 23, 2013). "Strings, bytes, runes and characters in Go" (https://blog.golang.org/strings).
62. Doxsey, Caleb. "Structs and Interfaces — An Introduction to Programming in Go" (https://web.archive.org/web/20181231043059/http://www.golang-book.com/books/intro/9). *www.golang-book.com*. Archived from the original (https://www.golang-book.com/books/intro/9) on December 31, 2018. Retrieved October 15, 2018.
63. "Go Type System Overview" (https://go101.org/article/type-system-overview.html). *go101.org*. Retrieved October 10, 2025.
64. Gerrand, Andrew. "Go Slices: usage and internals" (https://blog.golang.org/go-slices-usage-and-internals).
65. The Go Authors. "Effective Go: Slices" (https://golang.org/doc/effective_go.html#slices).
66. The Go authors. "Selectors" (https://golang.org/ref/spec#Selectors).
67. The Go authors. "Calls" (https://golang.org/ref/spec#Calls).
68. "Go Programming Language Specification, §Package unsafe" (https://golang.org/ref/spec#Package_unsafe). *The Go Programming Language*.
69. "The Go Programming Language Specification" (https://go.dev/ref/spec#Channel_types). *go.dev*. Retrieved December 31, 2021.
70. "The Go Programming Language Specification" (https://golang.org/ref/spec#Assignability). *The Go Programming Language*.
71. "A tour of go" (https://go.dev/tour/basics/13). *go.dev*.
72. "The Go Programming Language Specification" (https://golang.org/ref/spec#Constants). *The Go Programming Language*.
73. "The Go Programming Language Specification" (https://go.dev/ref/spec#Function_types). *go.dev*. Retrieved December 31, 2021.
74. "The Go Programming Language Specification" (https://golang.org/ref/spec#Calls). *The Go Programming Language*.
75. Schmager, Frank; Cameron, Nicholas; Noble, James (2010). *GoHotDraw: evaluating the Go programming language with design patterns*. Evaluation and Usability of Programming Languages and Tools. ACM.
76. Balbaert, Ivo (2012). *The Way to Go: A Thorough Introduction to the Go Programming Language*. iUniverse.

77. "The Evolution of Go" (https://talks.golang.org/2015/gophercon-goevolution.slide#19). *talks.golang.org*. Retrieved March 13, 2016.
78. Diggins, Christopher (November 24, 2009). "Duck Typing and the Go Programming Language" (http://www.drdobbs.com/architecture-and-design/duck-typing-and-the-go-programming-langu/228701527). *Dr. Dobb's, The world of software development*. Retrieved March 10, 2016.
79. Ryer, Mat (December 1, 2015). "Duck typing in Go" (https://medium.com/@matryer/golang-advent-calendar-day-one-duck-typing-a513aaed544d#.ebm7j81xu). Retrieved March 10, 2016.
80. "Frequently Asked Questions (FAQ) - The Go Programming Language" (https://golang.org/doc/faq). *The Go Programming Language*.
81. "The Go Programming Language Specification" (https://golang.org/ref/spec#Type_assertions). *The Go Programming Language*.
82. "The Go Programming Language Specification" (https://golang.org/ref/spec#Type_switches). *The Go Programming Language*.
83. "reflect package" (https://pkg.go.dev/reflect). *pkg.go.dev*.
84. "map[string]interface{} in Go" (https://bitfieldconsulting.com/golang/map-string-interface). *bitfieldconsulting.com*. June 6, 2020.
85. "Go Data Structures: Interfaces" (http://research.swtch.com/interfaces). Retrieved November 15, 2012.
86. "The Go Programming Language Specification" (https://golang.org/ref/spec#Interface_types). *The Go Programming Language*.
87. "Go 1.18 Release Notes: Generics" (https://go.dev/doc/go1.18#generics). *Go Programming Language*. March 15, 2022. Retrieved March 16, 2022.
88. "Type Parameters Proposal" (https://go.googlesource.com/proposal/+/HEAD/design/43651-type-parameters.md). *go.googlesource.com*. Retrieved June 25, 2023.
89. "The Go Programming Language Specification - The Go Programming Language" (https://go.dev/ref/spec). *go.dev*. Retrieved June 25, 2023.
90. "An Introduction To Generics - The Go Programming Language" (https://go.dev/blog/intro-generics). *go.dev*. Retrieved June 25, 2023.
91. "Type Parameters Proposal" (https://go.googlesource.com/proposal/+/HEAD/design/43651-type-parameters.md#using-a-constraint). *go.googlesource.com*. Retrieved June 25, 2023.
92. "Effective Go" (https://golang.org/doc/effective_go.html#constants). *golang.org*. The Go Authors. Retrieved May 13, 2014.
93. "Go Wiki: Iota - The Go Programming Language" (https://go.dev/wiki/Iota). *go.dev*. Retrieved May 15, 2025. "Go's iota identifier is used in const declarations to simplify definitions of incrementing numbers. Because it can be used in expressions, it provides a generality beyond that of simple enumerations"
94. "A Tutorial for the Go Programming Language" (https://golang.org/doc/go_tutorial.html). *The Go Programming Language*. Retrieved March 10, 2013. "In Go the rule about visibility of information is simple: if a name (of a top-level type, function, method, constant or variable, or of a structure field or method) is capitalized, users of the package may see it. Otherwise, the name and hence the thing being named is visible only inside the package in which it is declared."
95. "go" (https://golang.org/cmd/go/#hdr-Download_and_install_packages_and_dependencies). *The Go Programming Language*.

96. "How to Write Go Code" (https://golang.org/doc/code.html). *The Go Programming Language*. "The packages from the standard library are given short import paths such as "fmt" and "net/http". For your own packages, you must choose a base path that is unlikely to collide with future additions to the standard library or other external libraries. If you keep your code in a source repository somewhere, then you should use the root of that source repository as your base path. For instance, if you have an Example account at example.com/user, that should be your base path"
97. Pike, Rob (September 18, 2012). "Concurrency is not Parallelism" (https://vimeo.com/49718712).
98. Donovan, Alan A. A.; Kernighan, Brian W. (2016). *The Go programming language*. Addison-Wesley professional computing series. New York, Munich: Addison-Wesley. ISBN 978-0-13-419044-0.
99. Chisnall, David (2012). *The Go Programming Language Phrasebook* (https://books.google.com/books?id=scyH562VXZUC). Addison-Wesley. ISBN 9780132919005.
100. "Effective Go" (https://golang.org/doc/effective_go.html#sharing). *The Go Programming Language*.
101. Summerfield, Mark (2012). *Programming in Go: Creating Applications for the 21st Century*. Addison-Wesley.
102. "The Go Memory Model" (https://golang.org/ref/mem). Retrieved April 10, 2017.
103. "Go Concurrency Patterns" (https://talks.golang.org/2012/concurrency.slide). *The Go Programming Language*.
104. Graham-Cumming, John (August 24, 2013). "Recycling Memory Buffers in Go" (https://blog.cloudflare.com/recycling-memory-buffers-in-go). *The Cloudflare Blog*.
105. "tree.go" (https://golang.org/doc/play/tree.go).
106. Cheslack-Postava, Ewen. "Iterators in Go" (http://ewencp.org/blog/golang-iterators/).
107. Kernighan, Brian W. "A Descent Into Limbo" (http://www.vitanuova.com/inferno/papers/descent.html).
108. "The Go Memory Model" (https://golang.org/doc/go_mem.html). Retrieved January 5, 2011.
109. Tang, Peiyi (2010). *Multi-core parallel programming in Go* (https://web.archive.org/web/20160909032631/http://www.ualr.edu/pxtang/papers/acc10.pdf) (PDF). Proc. First International Conference on Advanced Computing and Communications. Archived from the original (http://www.ualr.edu/pxtang/papers/acc10.pdf) (PDF) on September 9, 2016. Retrieved May 14, 2015.
110. Nanz, Sebastian; West, Scott; Soares Da Silveira, Kaue. *Examining the expert gap in parallel programming* (http://se.inf.ethz.ch/people/west/expert-gap-europar-2013.pdf) (PDF). Euro-Par 2013. CiteSeerX 10.1.1.368.6137 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.368.6137).
111. Go Authors. "Share Memory By Communicating" (https://go.dev/doc/codewalk/sharemem/).
112. Cox, Russ. "Off to the Races" (http://research.swtch.com/gorace).
113. Pike, Rob (October 25, 2012). "Go at Google: Language Design in the Service of Software Engineering" (https://talks.golang.org/2012/splash.article). Google, Inc. "There is one important caveat: Go is not purely memory safe in the presence of concurrency."
114. "Introducing the Go Race Detector" (https://go.dev/blog/race-detector). *The Go Blog*. Retrieved June 26, 2013.
115. "Go 1.6 Release Notes - The Go Programming Language" (https://go.dev/doc/go1.6). *go.dev*. Retrieved November 17, 2023.
116. "Frequently Asked Questions (FAQ) - the Go Programming Language" (https://golang.org/doc/faq).
117. "A Story of a Fat Go Binary" (https://medium.com/@jondot/a-story-of-a-fat-go-binary-20edc6549b97). September 21, 2018.

118. "Go Programming Language Specification, §Interface types" (https://golang.org/ref/spec#Interface_types). *The Go Programming Language*.
119. "Go Programming Language Specification, §Type assertions" (https://golang.org/ref/spec#Type_assertion). *The Go Programming Language*.
120. "All Systems Are Go" (http://www.informit.com/articles/article.aspx?p=1623555). *informIT* (Interview). August 17, 2010. Retrieved June 21, 2018.
121. "Language Design FAQ" (https://web.archive.org/web/20091113154906/http://golang.org/doc/go_lang_faq.html#absent_features). November 13, 2009. Archived from the original (https://golang.org/doc/go_lang_faq.html#absent_features) on November 13, 2009.
122. "Proposal for an exception-like mechanism" (https://groups.google.com/group/golang-nuts/browse_thread/thread/1ce5cd050bb973e4). *golang-nuts*. March 25, 2010. Retrieved March 25, 2010.
123. "Weekly Snapshot History" (https://golang.org/doc/devel/weekly.html#2010-03-30). *The Go Programming Language*.
124. "Panic And Recover" (https://code.google.com/p/go-wiki/wiki/PanicAndRecover). Go wiki.
125. "Effective Go" (https://golang.org/doc/effective_go.html#panic). *The Go Programming Language*.
126. "gofmt" (https://golang.org/cmd/gofmt/). *The Go Programming Language*. Retrieved February 5, 2021.
127. "golang/lint public archive" (https://github.com/golang/lint). *github.com*. November 30, 2022.
128. "Effective Go" (https://golang.org/doc/effective_go.html). *The Go Programming Language*.
129. "Unused local variables" (https://yourbasic.org/golang/unused-local-variables/). *yourbasic.org*. Retrieved February 11, 2021.
130. "Unused package imports" (https://yourbasic.org/golang/unused-imports/). *yourbasic.org*. Retrieved February 11, 2021.
131. "Code Review Comments" (https://github.com/golang/go/wiki/CodeReviewComments). *GitHub*. Retrieved July 3, 2018.
132. "Talks" (https://talks.golang.org/). Retrieved July 3, 2018.
133. "Errors Are Values" (https://blog.golang.org/errors-are-values). Retrieved July 3, 2018.
134. "tools/gopls/README.md at master · golang/tools" (https://github.com/golang/tools/blob/master/gopls/README.md). *GitHub*. Retrieved November 17, 2023.
135. "fmt" (https://golang.org/pkg/fmt/). *The Go Programming Language*. Retrieved April 8, 2019.
136. "testing" (https://golang.org/pkg/testing/). *The Go Programming Language*. Retrieved December 27, 2020.
137. "http package - net/http - Go Packages" (https://pkg.go.dev/net/http). *pkg.go.dev*. Retrieved November 23, 2024.
138. Lee, Wei-Meng (November 24, 2022). "Introduction to the Go Programming Language" (https://web.archive.org/web/20230605071554/https://www.codemag.com/Article/2011051/Introduction-to-the-Go-Programming-Language). *Component Developer Magazine*. Archived from the original (https://www.codemag.com/Article/2011051/Introduction-to-the-Go-Programming-Language) on June 5, 2023. Retrieved September 8, 2023.
139. Hoffmann, Frank; Neumeyer, Mandy (August 2018). "Simply Secure" (https://web.archive.org/web/20230528175545/https://www.linux-magazine.com/Issues/2018/213/Caddy). *Linux Magazine*. No. 213. Archived from the original (https://www.linux-magazine.com/Issues/2018/213/Caddy) on May 28, 2023. Retrieved September 8, 2023.
140. Lee, Wei-Meng (August 31, 2022). "Introduction to Containerization Using Docker" (https://www.codemag.com/Article/2103061/Introduction-to-Containerization-Using-Docker). *CODE Magazine*. Archived (https://web.archive.org/web/20230530073551/https://www.codemag.com/Article/2103061/Introduction-to-Containerization-Using-Docker) from the original on May 30, 2023. Retrieved September 8, 2023.

141. Pirker, Alexander (February 24, 2023). "Kubernetes Security for Starters" (https://www.codemag.com/Article/2303071/Kubernetes-Security-for-Starters). *CODE Magazine*. Archived (https://web.archive.org/web/20230401212416/https://codemag.com/Article/2303071/Kubernetes-Security-for-Starters) from the original on April 1, 2023. Retrieved September 8, 2023.
142. Taft, Rebecca; Sharif, Irfan; Matei, Andrei; Van Benschoten, Nathan; Lewis, Jordan; Grieger, Tobias; Niemi, Kai; Woods, Andy; Birzin, Anne; Poss, Raphael; Bardea, Paul; Ranade, Amruta; Darnell, Ben; Gruneir, Bram; Jaffray, Justin; Zhang, Lucy; Mattis, Peter (June 11, 2020). "CockroachDB: The Resilient Geo-Distributed SQL Database". *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. pp. 1493–1509. doi:10.1145/3318464.3386134 (https://doi.org/10.1145%2F3318464.3386134). ISBN 978-1-4503-6735-6.
143. Hopkins, Brandon (September 13, 2022). "Static Site Generation with Hugo" (https://www.linuxjournal.com/content/static-site-generation-hugo). *Linux Journal*. Archived (https://web.archive.org/web/20230408065506/https://www.linuxjournal.com/content/static-site-generation-hugo) from the original on April 8, 2023. Retrieved September 8, 2023.
144. Simionato, Michele (November 15, 2009). "Interfaces vs Inheritance (or, watch out for Go!)" (http://www.artima.com/weblogs/viewpost.jsp?thread=274019). artima. Retrieved November 15, 2009.
145. Astels, Dave (November 9, 2009). "Ready, Set, Go!" (https://web.archive.org/web/20181019164102/https://www.engineyard.com/blog/ready-set-go). engineyard. Archived from the original (https://www.engineyard.com/blog/ready-set-go) on October 19, 2018. Retrieved November 9, 2009.
146. jt (January 11, 2010). "Google's Go Wins Programming Language Of The Year Award" (http://jaxenter.com/google-s-go-wins-programming-language-of-the-year-award-10069.html). jaxenter. Retrieved December 5, 2012.
147. "TIOBE Programming Community Index for June 2015" (http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html). TIOBE Software. June 2015. Retrieved July 5, 2015.
148. "TIOBE Index" (https://www.tiobe.com/tiobe-index/). *TIOBE*. Retrieved July 15, 2024.
149. Eckel, Bruce (August 27, 2011). "Calling Go from Python via JSON-RPC" (http://www.artima.com/weblogs/viewpost.jsp?thread=333589). Retrieved August 29, 2011.
150. Hundt, Robert (2011). *Loop recognition in C++/Java/Go/Scala* (https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf) (PDF). Scala Days.
151. Metz, Cade (July 1, 2011). "Google Go strikes back with C++ bake-off" (https://www.theregister.co.uk/2011/07/01/go_v_cpluplus_redux/). *The Register*.
152. Yager, Will. "Why Go is not Good" (http://yager.io/programming/go.html). Retrieved November 4, 2018.
153. Dobronszki, Janos. "Everyday Hassles in Go" (https://crufter.com/everyday-hassles-in-go). Retrieved November 4, 2018.
154. "I want off Mr. Golang's Wild Ride" (https://fasterthanli.me/articles/i-want-off-mr-golangs-wild-ride). February 28, 2020. Retrieved November 17, 2020.
155. "proposal: os: Create/Open/OpenFile() set FILE_SHARE_DELETE on windows #32088" (https://github.com/golang/go/issues/32088). *GitHub*. May 16, 2019. Retrieved November 17, 2020.
156. Tu, Tengfei (2019). "Understanding Real-World Concurrency Bugs in Go" (https://songlh.github.io/paper/go-study.pdf) (PDF). "For example, around 58% of blocking bugs are caused by message passing. In addition to the violation of Go's channel usage rules (e.g., waiting on a channel that no one sends data to or close), many concurrency bugs are caused by the mixed usage of message passing and other new semantics and new libraries in Go, which can easily be overlooked but hard to detect"

157. Brownlee, John (November 13, 2009). "Google didn't google "Go" before naming their programming language' " (https://web.archive.org/web/20151208143907/http://www.geek.com/news/google-didnt-google-go-before-naming-their-programming-language-977351/). Archived from the original (http://www.geek.com/news/google-didnt-google-go-before-naming-their-programming-language-977351/) on December 8, 2015. Retrieved May 26, 2016.
158. Claburn, Thomas (November 11, 2009). "Google 'Go' Name Brings Accusations Of Evil' " (https://web.archive.org/web/20100722010320/http://www.informationweek.com/news/software/web_services/showArticle.jhtml?articleID=221601351). InformationWeek. Archived from the original (http://www.informationweek.com/news/software/web_services/showArticle.jhtml?articleID=221601351) on July 22, 2010. Retrieved January 18, 2010.
159. "Issue 9 - go — I have already used the name for *MY* programming language" (https://github.com/golang/go/issues/9#issuecomment-66047478). *Github*. Google Inc. Retrieved October 12, 2010.

Further reading

- Donovan, Alan; Kernighan, Brian (October 2015). *The Go Programming Language* (https://www.informit.com/store/go-programming-language-9780134190440) (1st ed.). Addison-Wesley Professional. p. 400. ISBN 978-0-13-419044-0.
- Bodner, Jon (March 2021). *Learning Go* (https://www.oreilly.com/library/view/learning-go/9781492077206/) (1st ed.). O'Reilly. p. 352. ISBN 9781492077213.

External links

- Official website (https://go.dev)