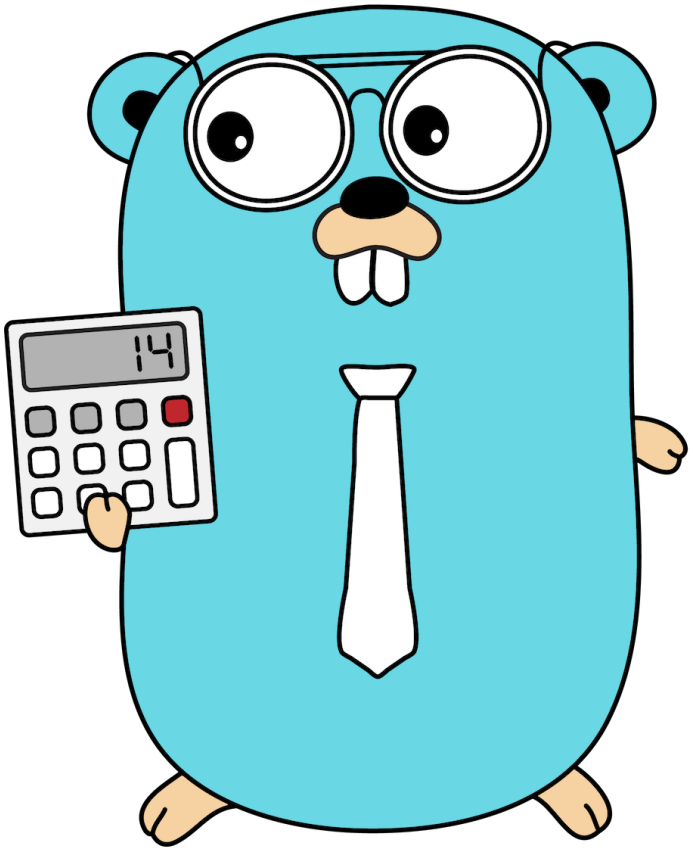
# Section 10

29 July 2019



## Introduction

Go was built to simplify and streamline the engineering work at Google. Its language features allow an average programmer to write scalable and concurrent programs easily. Golang has been described as “Useable C, disciplined Python.”

## Features of Golang

### Performance

Unlike other languages that have relied on runtime interpreters such as Java and Python, Go has been built from the ground up and any Go program compiles directly to machine code. Go is a language catered to today’s multi-core computing environments. It has lightweight threads called goroutines and also a runtime scheduler to minimize the context switch overhead associated with creating many threads in the operating system.

### Simplicity

*"Go was born out of frustration with existing languages and environments for systems programming. Programming had become too difficult and the choice of languages was partly to blame. One had to choose either efficient compilation, efficient execution, or ease of programming; all three were not available in the same mainstream language."*

– From the official Golang FAQ

Golang aims to address many challenges faced by the software engineering industry today by building a better programming language, whereas past attempts mainly focused on building tools to make the experience of working with other languages better and did not focus on the programming language itself. Common problems faced by the industry include long compile time, complex language syntax which makes the code hard to maintain, runtime dependency issues, to name just a few. Moreover, a big premise in Go is to have no change in the language. If you have experience with Swift or Python, you know what it is like to work with a constantly changing language.

### Concurrency

Go enables first-class support for concurrency with Goroutines and Channels. It is easy for programmers to write scalable programs that leverage the multi-core nature of modern-day CPUs. While other languages do support multi-threading, it often involves many library calls.

### Deployment

Go compiles everything into a statically linked binary file, so no dependencies are required. You just have a single binary to deploy.  Think about pip install for Python and the hassle you have to go through if you want to run your python program in a different environment.

# A Go Tutorial

## Hello World

```
// A package clause starts every source file. Main is a special name declaring an executable rather than a
library.
package main

// Import declaration declares library packages referenced in this file.
import (
    "fmt"       // A package in the Go standard library.
    m "math"    // Math library with local alias m.
    "os"        // OS functions like working with the file system
    "strconv"   // String conversions.
)

func main() {
    // Println outputs a line to stdout. It comes from the package fmt.
    fmt.Println("Hello world!")

    // Call another function within this package.
    beyondHello()
}

func beyondHello() {
    var x int   // Variable declaration. Variables must be declared before use.
    x = 3       // Variable assignment.
    y := 4      // "Short" declarations use := to infer the type, declare, and assign.
    sum, prod := learnMultiple(x, y)          // Function returns two values.
    fmt.Println("sum:", sum, "prod:", prod) // Simple output.
}

func learnMultiple(x, y int) (sum, prod int) {
    // Return two values.
    return x + y, x * y
}
```

## Types

```
func learnTypes() {
    str := "Learn Go!" // string type.
    f := 3.14195 // float64, an IEEE-754 64-bit floating point number.

    // var syntax with initializers.
    var u uint = 7 // Unsigned, but implementation dependent size as with int.
    var pi float32 = 22. / 7

    // Conversion syntax with a short declaration.
    n := byte('\n') // byte is an alias for uint8.

    // Arrays have size fixed at compile time.
    var a4 [4]int           // An array of 4 ints, initialized to all 0.

    // Slices have a dynamic size. Arrays and slices each have advantages
    // but use cases for slices are much more common.
    s3 := []int{4, 5, 9}
    s4 := make([]int, 4)    // Allocates slice of 4 ints, initialized to all 0.

    // Because they are dynamic, slices can be appended to on-demand.
    // To append elements to a slice, the built-in append() function is used.
    s := []int{1, 2, 3}     // Result is a slice of length 3.
    s = append(s, 4, 5, 6)  // Added 3 elements. Slice now has length of 6.
    fmt.Println(s) // Updated slice is now [1 2 3 4 5 6]

    // Maps are a dynamically growable associative array type, like the
    // hash or dictionary types of some other languages.
    m := map[string]int{"three": 3, "four": 4}
    m["one"] = 1
}
```

## Pointers

Go is fully garbage collected. It has pointers but no pointer arithmetic.

```
func learnMemory() (p, q *int) {
    // Named return values p and q have type pointer to int.
    p = new(int) // Built-in function new allocates memory.
    // The allocated int is initialized to 0, p is no longer nil.
    s := make([]int, 20) // Allocate 20 ints as a single block of memory.
    s[3] = 7             // Assign one of them.
    r := -2              // Declare another local variable.
    return &s[3], &r     // & takes the address of an object.
```

## Defer

A defer statement pushes a function call onto a list. The list of saved calls is executed AFTER the surrounding function returns. Defer is commonly used to close a file, so the function closing the file stays close to the function opening the file.

```
func learnDefer() (ok bool) {
    defer fmt.Println("deferred statements execute in reverse (LIFO) order.")
    defer fmt.Println("\nThis line is being printed first because")
    return true
}
```

### Interface and Struct

```
// Define Stringer as an interface type with one method, String.
type Stringer interface {
    String() string
}

// Define pair as a struct with two fields, ints named x and y.
type pair struct {
    x, y int
}

// Define a method on type pair. Pair now implements Stringer because Pair has defined all the methods
in the interface.
func (p pair) String() string { // p is called the "receiver"
    // Sprintf is another public function in package fmt.
    // Dot syntax references fields of p.
    return fmt.Sprintf("(%d, %d)", p.x, p.y)
}

func learnInterfaces() {
    p := pair{3, 4}
    fmt.Println(p.String()) // Call String method of p, of type pair.
    var i Stringer             // Declare i of interface type Stringer.
    i = p                      // Valid because pair implements Stringer
    // Call String method of i, of type Stringer. Output same as above.
    fmt.Println(i.String())

    // Functions in the fmt package call the String method to ask an object
    // for a printable representation of itself.
    fmt.Println(p) // Output same as above. Println calls String method.
    fmt.Println(i) // Output same as above.
}
```

### Error Handling

- An error in Go implements the error interface which has an Error() string method.
- Functions that could error typically return errors as the second return value. If the value is not nil, an error has occurred.
- It is also a common practice to propagate the error to higher layers of abstraction through simple returns (perhaps adding details to the error message).

```
func learnErrorHandling() {
    m := map[int]string{3: "three", 4: "four"}
```

```
    if x, ok := m[1]; !ok { // ok will be false because 1 is not in the map.
        fmt.Println("no one there")
    } else {
        fmt.Print(x) // x would be the value, if it were in the map.
    }
    // An error value communicates not just "ok" but more about the problem.
    if _, err := strconv.Atoi("non-int"); err != nil { // _ discards value
        // prints 'strconv.ParseInt: parsing "non-int": invalid syntax'
        fmt.Println(err)
    }
}
```

## Concurrency

It is helpful to think of goroutines as threads and go func() as pthread_create. However, goroutines are, in fact, user-level threads, not kernel threads which makes them less expensive to create.

```
func learnConcurrency() {
    c := make(chan int)

    // Start three concurrent goroutines. Numbers will be incremented
    // concurrently, perhaps in parallel if the machine is capable and
    // properly configured. All three send to the same channel.
    go inc(0, c) // go is a statement that starts a new goroutine.
    go inc(10, c)
    go inc(-805, c)
    // Read three results from the channel and print them out.
    // There is no telling in what order the results will arrive!
    fmt.Println(<-c, <-c, <-c) // channel on right, <- is "receive" operator.

    cs := make(chan string)       // Another channel, this one handles strings.
    go func() { c <- 84 }()       // Start a new goroutine just to send a value.
    go func() { cs <- "wordy" }() // Again, for cs this time.

    // Select has syntax like a switch statement but each case involves
    // a channel operation. It selects a case at random out of the cases
    // that are ready to communicate.
    select {
    case i := <-c: // The value received can be assigned to a variable,
        fmt.Printf("it's a %T", i)
    case <-cs: // or the value received can be discarded.
        fmt.Println("it's a string")
    }
    // At this point a value was taken from either c or cs. One of the two
    // goroutines started above has completed, the other will remain blocked.
}
```

## Examples

### Worker-queue Pattern

In this example, we'll look at how to implement a worker pool using goroutines and channels.

```
package main

import "fmt"
import "time"

// Here's the worker, of which we'll run several
// concurrent instances. These workers will receive
// work on the `jobs` channel and send the corresponding
// results on `results`. We'll sleep a second per job to
// simulate an expensive task.

func worker(id int, jobs <-chan int, results chan<- int) {
    for j := range jobs {
        fmt.Println("worker", id, "started  job", j)
        time.Sleep(time.Second)
        fmt.Println("worker", id, "finished job", j)
        results <- j * 2
    }
}

func main() {

    // In order to use our pool of workers we need to send
    // them work and collect their results. We make 2
    // channels for this.
    jobs := make(chan int, 100)
    results := make(chan int, 100)

    // This starts up 3 workers, initially blocked
    // because there are no jobs yet.
    for w := 1; w <= 3; w++ {
        go worker(w, jobs, results)
    }

    // Here we send 5 `jobs` and then `close` that
    // channel to indicate that's all the work we have.
    for j := 1; j <= 5; j++ {
        jobs <- j
    }
    close(jobs)

    // Finally we collect all the results of the work.
    // This also ensures that the worker goroutines have
```

```
    // finished. An alternative way to wait for multiple
    // goroutines is to use a [WaitGroup](waitgroups).
    for a := 1; a <= 5; a++ {
        <-results
    }
}
```

## Reference


1. https://talks.golang.org/2012/splash.article
2. https://golang.org/doc/faq
3. https://gobyexample.com/worker-pools
4. https://learnxinyminutes.com/docs/go/