

Programming in Golang



Learning GO
Programming with
hands-on experience

Programming in Golang

Learning golang Programming with hands-on experience

Alejandro Garcia

This book is for sale at <http://leanpub.com/programming-in-golang>

This version was published on 2020-08-24



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2020 Alejandro Garcia

Tweet This Book!

Please help Alejandro Garcia by spreading the word about this book on [Twitter](#)!

The suggested tweet for this book is:

This book on Golang is awesome !! <https://leanpub.com/programming-in-golang/>

The suggested hashtag for this book is [#golang](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

[#golang](#)

Contents

Other Books by Alejandro	1
Recommended Resources	5
Introduction	7
Who is this book for ?	7
What this book covers ?	7
Hello World	8
Values	9
Variables	10
Constants	12
For	14
If/Else	16
Switch	18
Arrays	20
Slices	22
Maps	25
Range	27
Functions	29
Multiple Return Values	31
Variadic Functions	32
Closures	34
Recursion	36

CONTENTS

Pointers	37
Structs	39
Methods	41
Interfaces	43
Errors	45
Goroutines	48
Channels	50
Channel Buffering	51
Channel Synchronization	52
Channel Directions	54
Select	55
Timeouts	57
Non-Blocking Channel Operations	59
Closing Channels	61
Range over Channels	63
Timers	64
Tickers	66
Worker Pools	68
WaitGroups	70
Rate Limiting	72
Atomic Counters	75
Mutexes	77
Stateful Goroutines	80
Sorting	83
Sorting by Functions	85

CONTENTS

Panic	87
Defer	89
Collection Functions	91
String Functions	94
String Formatting	96
Regular Expressions	100
JSON	103
XML	107
Time	110
Epoch	113
Time Formatting / Parsing	115
Random Numbers	117
Number Parsing	119
URL Parsing	121
SHA1 Hashes	123
Base64 Encoding	125
Reading Files	127
Writing Files	130
Line Filters	132
File Paths	134
Directories	136
Temporary Files and Directories	139
Testing	141
Command-Line Arguments	144
Command-Line Flags	145

CONTENTS

Command-Line Subcommands	148
Environment Variables	150
HTTP Clients	152
HTTP Servers	154
Context	156
Spawning Processes	158
Exec'ing Processes	160
Signals	162
Exit	164
Keep developing your GO skills	165
About the Author	166

Other Books by Alejandro



Up to Speed with Javascript in 59 minutes

Up to Speed with Javascript in 59 minutes¹

¹https://leanpub.com/javascript-in-59-minutes/?utm_source=ebook&utm_medium=leanpub&utm_campaign=golang



Learning Python Programming from Scratch

Learning Python Programming from Scratch²

²https://leanpub.com/learning-python-programming-from-scratch/?utm_source=ebook&utm_medium=leanpub&utm_campaign=golang



Javascript Snippets: Secret Code Snippets used by Powerful Developers to create Innovative Apps

[Javascript Snippets: Secret Code Snippets used by Powerful Developers to create Innovative Apps](https://leanpub.com/javascript-snippets/?utm_source=ebook&utm_medium=leanpub&utm_campaign=golang)³

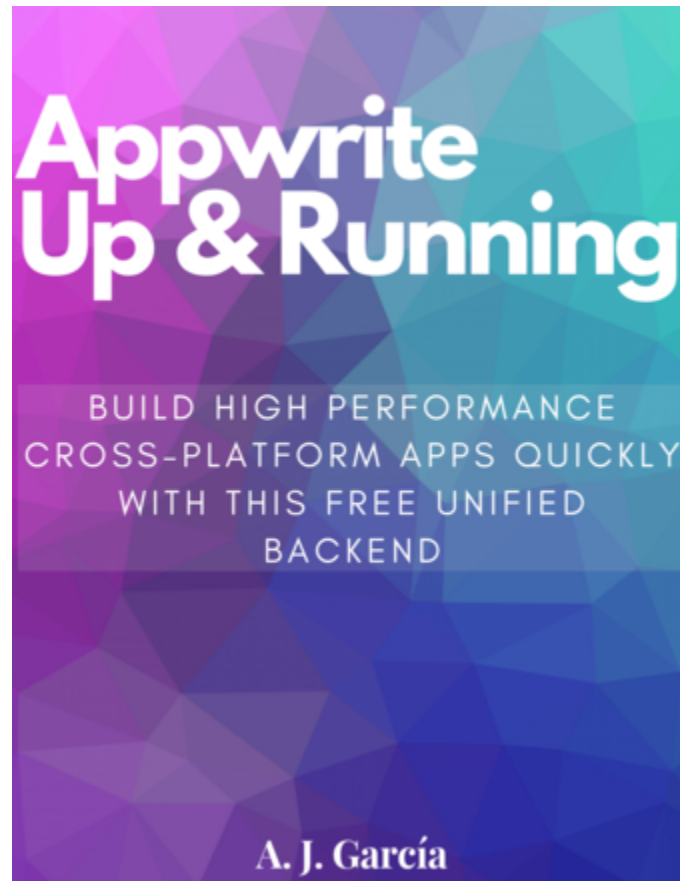
³https://leanpub.com/javascript-snippets/?utm_source=ebook&utm_medium=leanpub&utm_campaign=golang



Frontend Developer Interview Questions: Prep for the interview and get the job you want

Frontend Developer Interview Questions: Prep for the interview and get the job you want⁴

⁴https://leanpub.com/frontend-developer-interview-questions/?utm_source=ebook&utm_medium=leanpub&utm_campaign=golang



Appwrite Up and Running: Build High Performance cross-platform Apps quickly with this Free unified Backend

Appwrite Up and Running: Build High Performance cross-platform Apps quickly with this Free unified Backend⁵

Recommended Resources

If you are looking into video tutorials and trainings to polish your Javascript and programming skills take a look at the below options:

- [Udemy GO](#)⁶
- [Programming with Google Go Specialization](#)⁷

⁵https://leanpub.com/appwrite-up-and-running/?utm_source=ebook&utm_medium=leanpub&utm_campaign=golang

⁶<https://click.linksynergy.com/deeplink?id=NcMZU1aTATA&mid=39197&murl=https%3A%2F%2Fwww.udemy.com%2Fcourse%2Flearn-how-to-code%2F>

⁷<https://click.linksynergy.com/deeplink?id=NcMZU1aTATA&mid=40328&murl=https%3A%2F%2Fwww.coursera.org%2Fspecializations%2Fgoogle-golang>

Tuesday Messages

On Tuesday, I send some of my Free books to subscribers. You can expect updates, technical articles and courses recommendations. People tell me this content is so valuable to them that sometimes they ask me to send info on particular subjects.

If you would like to **receive the exclusive free content only for subscribers**, then you can join here: [Alejandro's Updates](https://alejandro-garcia-author.ck.page/2ed39ffaf)⁸

⁸<https://alejandro-garcia-author.ck.page/2ed39ffaf>

Introduction

Go, or Golang, is an open source programming language. It's statically typed and produces compiled machine code binaries. Developers say that Google's Go language is the C for the twenty-first century when it comes to syntax. However, this new programming language includes tooling that allows you to safely use memory, manage objects, collect garbage, and provide static (or strict) typing along with concurrency.

The world was first introduced to Go in 2009 thanks to Google's Rob Pike, Robert Griesemer, and Ken Thompson. The main goal of creating Go was to combine the best features of other programming languages:

- Ease of use together with state-of-the-art productivity
- High-level efficiency along with static typing
- Advanced performance for networking and the full use of multi-core power

It's a good idea to start learning GO today !! It has so many benefits and it's a demanded language career wise.

Some of the benefits of GO include:

- Golang is open source by nature.
- Golang is fast.
- Golang is concurrent.
- Golang is cross-platform.
- Golang has a garbage collector.

Who is this book for ?

This book is for someone interested in learning GO, it would be easier if you already know the basics of programming and can program in other languages like Python for example.

What this book covers ?

This book contains a collection of exercises for you to analyze the code, modify it and run it on your system so you can **learn the features of GO by doing**. It's hands on experience.

Each exercise **will provide some explanations**, the main idea is for you to understand an **explore even more the lines of code**.

Hello World

Our first program will print the classic “hello world” message. Here’s the full source code.

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     fmt.Println("hello world")
7 }
```

To run the program, put the code in hello-world.go and use go run.

Sometimes we’ll want to build our programs into binaries. We can do this using go build.

We can then execute the built binary directly.

Now that we can run and build basic Go programs, let’s learn more about the language.

Output:

```
1 $ go run hello-world.go
2 hello world
3
4 $ go build hello-world.go
5 $ ls
6 hello-world    hello-world.go
7
8
9 $ ./hello-world
10 hello world
```

Values

Go has various value types including strings, integers, floats, booleans, etc. Here are a few basic examples.

```
1 package main
2
3 import "fmt"
4
5 func main() {
6
7     fmt.Println("go" + "lang")
8
9     fmt.Println("1+1 =", 1+1)
10    fmt.Println("7.0/3.0 =", 7.0/3.0)
11
12    fmt.Println(true && false)
13    fmt.Println(true || false)
14    fmt.Println(!true)
15 }
```

Strings, which can be added together with +.

Integers and floats.

Booleans, with boolean operators as you'd expect.

Output:

```
1 $ go run values.go
2 go lang
3 1+1 = 2
4 7.0/3.0 = 2.3333333333333335
5 false
6 true
7 false
```

Variables

In Go, variables are explicitly declared and used by the compiler to e.g. check type-correctness of function calls.

```
1 package main
2
3 import "fmt"
4
5 func main() {
6
7     var a = "initial"
8     fmt.Println(a)
9
10    var b, c int = 1, 2
11    fmt.Println(b, c)
12
13    var d = true
14    fmt.Println(d)
15
16    var e int
17    fmt.Println(e)
18
19    f := "apple"
20    fmt.Println(f)
21 }
```

var declares 1 or more variables.

You can declare multiple variables at once.

Go will infer the type of initialized variables.

Variables declared without a corresponding initialization are zero-valued. For example, the zero value for an int is 0.

The := syntax is shorthand for declaring and initializing a variable, e.g. for var f string = “apple” in this case.

Output:

```
1  $ go run variables.go
2  initial
3  1 2
4  true
5  0
6  apple
```

Constants

Go supports constants of character, string, boolean, and numeric values.

```
1 package main
2
3 import (
4     "fmt"
5     "math"
6 )
7
8 const s string = "constant"
9
10 func main() {
11     fmt.Println(s)
12
13     const n = 5000000000
14
15     const d = 3e20 / n
16     fmt.Println(d)
17
18     fmt.Println(int64(d))
19
20     fmt.Println(math.Sin(n))
21 }
```

Output:

const declares a constant value.

A const statement can appear anywhere a var statement can.

Constant expressions perform arithmetic with arbitrary precision.

A numeric constant has no type until it's given one, such as by an explicit conversion.

A number can be given a type by using it in a context that requires one, such as a variable assignment or function call. For example, here `math.Sin` expects a `float64`.

```
1 $ go run constant.go
2 constant
3 6e+11
4 600000000000
5 -0.28470407323754404
```

For

for is Go's only looping construct. Here are some basic types of for loops.

```
1 package main
2
3 import "fmt"
4
5 func main() {
6
7     i := 1
8     for i <= 3 {
9         fmt.Println(i)
10        i = i + 1
11    }
12
13    for j := 7; j <= 9; j++ {
14        fmt.Println(j)
15    }
16
17    for {
18        fmt.Println("loop")
19        break
20    }
21
22    for n := 0; n <= 5; n++ {
23        if n%2 == 0 {
24            continue
25        }
26        fmt.Println(n)
27    }
28 }
```

The most basic type, with a single condition.

A classic initial/condition/after for loop.

for without a condition will loop repeatedly until you break out of the loop or return from the enclosing function.

You can also continue to the next iteration of the loop.

We'll see some other for forms later when we look at range statements, channels, and other data structures.

Output:

```
1  $ go run for.go
2  1
3  2
4  3
5  7
6  8
7  9
8  loop
9  1
10 3
11 5
```

If/Else

Branching with if and else in Go is straight-forward.

```
1 package main
2
3 import "fmt"
4
5 func main() {
6
7     if 7%2 == 0 {
8         fmt.Println("7 is even")
9     } else {
10        fmt.Println("7 is odd")
11    }
12
13    if 8%4 == 0 {
14        fmt.Println("8 is divisible by 4")
15    }
16
17    if num := 9; num < 0 {
18        fmt.Println(num, "is negative")
19    } else if num < 10 {
20        fmt.Println(num, "has 1 digit")
21    } else {
22        fmt.Println(num, "has multiple digits")
23    }
24 }
```

Here's a basic example.

You can have an if statement without an else.

A statement can precede conditionals; any variables declared in this statement are available in all branches.

Note that you don't need parentheses around conditions in Go, but that the braces are required.

There is no ternary if in Go, so you'll need to use a full if statement even for basic conditions.

Output:

```
1  $ go run if-else.go
2  7 is odd
3  8 is divisible by 4
4  9 has 1 digit
```

Switch

Switch statements express conditionals across many branches.

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func main() {
9
10     i := 2
11     fmt.Print("Write ", i, " as ")
12     switch i {
13     case 1:
14         fmt.Println("one")
15     case 2:
16         fmt.Println("two")
17     case 3:
18         fmt.Println("three")
19     }
20
21     switch time.Now().Weekday() {
22     case time.Saturday, time.Sunday:
23         fmt.Println("It's the weekend")
24     default:
25         fmt.Println("It's a weekday")
26     }
27
28     t := time.Now()
29     switch {
30     case t.Hour() < 12:
31         fmt.Println("It's before noon")
32     default:
33         fmt.Println("It's after noon")
34     }
35 }
```

```
36     whatAmI := func(i interface{}) {
37         switch t := i.(type) {
38             case bool:
39                 fmt.Println("I'm a bool")
40             case int:
41                 fmt.Println("I'm an int")
42             default:
43                 fmt.Printf("Don't know type %T\n", t)
44         }
45     }
46     whatAmI(true)
47     whatAmI(1)
48     whatAmI("hey")
49 }
```

Here's a basic switch.

You can use commas to separate multiple expressions in the same case statement. We use the optional default case in this example as well.

switch without an expression is an alternate way to express if/else logic. Here we also show how the case expressions can be non-constants.

A type switch compares types instead of values. You can use this to discover the type of an interface value. In this example, the variable t will have the type corresponding to its clause.

Output:

```
1  $ go run switch.go
2  Write 2 as two
3  It's a weekday
4  It's after noon
5  I'm a bool
6  I'm an int
7  Don't know type string
```

Arrays

In Go, an array is a numbered sequence of elements of a specific length.

```
1 package main
2
3 import "fmt"
4
5 func main() {
6
7     var a [5]int
8     fmt.Println("emp:", a)
9
10    a[4] = 100
11    fmt.Println("set:", a)
12    fmt.Println("get:", a[4])
13
14    fmt.Println("len:", len(a))
15
16    b := [5]int{1, 2, 3, 4, 5}
17    fmt.Println("dcl:", b)
18
19    var twoD [2][3]int
20    for i := 0; i < 2; i++ {
21        for j := 0; j < 3; j++ {
22            twoD[i][j] = i + j
23        }
24    }
25    fmt.Println("2d: ", twoD)
26 }
```

Here we create an array `a` that will hold exactly 5 ints. The type of elements and length are both part of the array's type. By default an array is zero-valued, which for ints means 0s.

We can set a value at an index using the `array[index] = value` syntax, and get a value with `array[index]`.

The builtin `len` returns the length of an array.

Use this syntax to declare and initialize an array in one line.

Array types are one-dimensional, but you can compose types to build multi-dimensional data structures.

Note that arrays appear in the form `[v1 v2 v3 ...]` when printed with `fmt.Println`.

You'll see slices much more often than arrays in typical Go. We'll look at slices next.

Output:

```
1  $ go run arrays.go
2  emp: [0 0 0 0 0]
3  set: [0 0 0 0 100]
4  get: 100
5  len: 5
6  dcl: [1 2 3 4 5]
7  2d:  [[0 1 2] [1 2 3]]
```

Slices

Slices are a key data type in Go, giving a more powerful interface to sequences than arrays.

```
1 package main
2
3 import "fmt"
4
5 func main() {
6
7     s := make([]string, 3)
8     fmt.Println("emp:", s)
9
10    s[0] = "a"
11    s[1] = "b"
12    s[2] = "c"
13    fmt.Println("set:", s)
14    fmt.Println("get:", s[2])
15
16    fmt.Println("len:", len(s))
17
18    s = append(s, "d")
19    s = append(s, "e", "f")
20    fmt.Println("apd:", s)
21
22    c := make([]string, len(s))
23    copy(c, s)
24    fmt.Println("cpy:", c)
25
26    l := s[2:5]
27    fmt.Println("s11:", l)
28
29    l = s[:5]
30    fmt.Println("s12:", l)
31
32    l = s[2:]
33    fmt.Println("s13:", l)
34
35    t := []string{"g", "h", "i"}
```

```
36     fmt.Println("dcl:", t)
37
38     twoD := make([][]int, 3)
39     for i := 0; i < 3; i++ {
40         innerLen := i + 1
41         twoD[i] = make([]int, innerLen)
42         for j := 0; j < innerLen; j++ {
43             twoD[i][j] = i + j
44         }
45     }
46     fmt.Println("2d: ", twoD)
47 }
```

Unlike arrays, slices are typed only by the elements they contain (not the number of elements). To create an empty slice with non-zero length, use the builtin `make`. Here we make a slice of strings of length 3 (initially zero-valued).

We can set and get just like with arrays.

`len` returns the length of the slice as expected.

In addition to these basic operations, slices support several more that make them richer than arrays. One is the builtin `append`, which returns a slice containing one or more new values. Note that we need to accept a return value from `append` as we may get a new slice value.

Slices can also be copy'd. Here we create an empty slice `c` of the same length as `s` and copy into `c` from `s`.

Slices support a “slice” operator with the syntax `slice[low:high]`. For example, this gets a slice of the elements `s[2]`, `s[3]`, and `s[4]`.

This slices up to (but excluding) `s[5]`.

And this slices up from (and including) `s[2]`.

We can declare and initialize a variable for slice in a single line as well.

Slices can be composed into multi-dimensional data structures. The length of the inner slices can vary, unlike with multi-dimensional arrays.

Note that while slices are different types than arrays, they are rendered similarly by `fmt.Println`.

Check out this great blog post by the Go team for more details on the design and implementation of slices in Go.

Now that we’ve seen arrays and slices we’ll look at Go’s other key builtin data structure: maps.

Output:

```
1  $ go run slices.go
2  emp: [ ]
3  set: [a b c]
4  get: c
5  len: 3
6  apd: [a b c d e f]
7  cpy: [a b c d e f]
8  s11: [c d e]
9  s12: [a b c d e]
10 s13: [c d e f]
11 dcl: [g h i]
12 2d: [[0] [1 2] [2 3 4]]
```

Maps

Maps are Go's built-in associative data type (sometimes called hashes or dicts in other languages).

```
1 package main
2
3 import "fmt"
4
5 func main() {
6
7     m := make(map[string]int)
8
9     m["k1"] = 7
10    m["k2"] = 13
11
12    fmt.Println("map:", m)
13
14    v1 := m["k1"]
15    fmt.Println("v1: ", v1)
16
17    fmt.Println("len:", len(m))
18
19    delete(m, "k2")
20    fmt.Println("map:", m)
21
22    _, prs := m["k2"]
23    fmt.Println("prs:", prs)
24
25    n := map[string]int{"foo": 1, "bar": 2}
26    fmt.Println("map:", n)
27 }
```

To create an empty map, use the builtin make: `make(map[key-type]val-type)`.

Set key/value pairs using typical `name[key] = val` syntax.

Printing a map with e.g. `fmt.Println` will show all of its key/value pairs.

Get a value for a key with `name[key]`.

The builtin `len` returns the number of key/value pairs when called on a map.

The builtin `delete` removes key/value pairs from a map.

The optional second return value when getting a value from a map indicates if the key was present in the map. This can be used to disambiguate between missing keys and keys with zero values like 0 or “”. Here we didn’t need the value itself, so we ignored it with the blank identifier `_`.

You can also declare and initialize a new map in the same line with this syntax.

Note that maps appear in the form `map[k:v k:v]` when printed with `fmt.Println`.

Output:

```
1 $ go run maps.go
2 map: map[k1:7 k2:13]
3 v1: 7
4 len: 2
5 map: map[k1:7]
6 prs: false
7 map: map[bar:2 foo:1]
```

Range

range iterates over elements in a variety of data structures. Let's see how to use range with some of the data structures we've already learned.

```
1 package main
2
3 import "fmt"
4
5 func main() {
6
7     nums := []int{2, 3, 4}
8     sum := 0
9     for _, num := range nums {
10         sum += num
11     }
12     fmt.Println("sum:", sum)
13
14     for i, num := range nums {
15         if num == 3 {
16             fmt.Println("index:", i)
17         }
18     }
19
20     kvs := map[string]string{"a": "apple", "b": "banana"}
21     for k, v := range kvs {
22         fmt.Printf("%s -> %s\n", k, v)
23     }
24
25     for k := range kvs {
26         fmt.Println("key:", k)
27     }
28
29     for i, c := range "go" {
30         fmt.Println(i, c)
31     }
32 }
```

Here we use range to sum the numbers in a slice. Arrays work like this too.

range on arrays and slices provides both the index and value for each entry. Above we didn't need the index, so we ignored it with the blank identifier `_`. Sometimes we actually want the indexes though.

range on map iterates over key/value pairs.

range can also iterate over just the keys of a map.

range on strings iterates over Unicode code points. The first value is the starting byte index of the rune and the second the rune itself.

Output:

```
1  $ go run range.go
2  sum: 9
3  index: 1
4  a -> apple
5  b -> banana
6  key: a
7  key: b
8  0 103
9  1 111
```

Functions

Functions are central in Go. We'll learn about functions with a few different examples.

```
1 package main
2
3 import "fmt"
4
5 func plus(a int, b int) int {
6
7     return a + b
8 }
9
10 func plusPlus(a, b, c int) int {
11     return a + b + c
12 }
13
14 func main() {
15
16     res := plus(1, 2)
17     fmt.Println("1+2 =", res)
18
19     res = plusPlus(1, 2, 3)
20     fmt.Println("1+2+3 =", res)
21 }
```

Here's a function that takes two ints and returns their sum as an int.

Go requires explicit returns, i.e. it won't automatically return the value of the last expression.

When you have multiple consecutive parameters of the same type, you may omit the type name for the like-typed parameters up to the final parameter that declares the type.

Call a function just as you'd expect, with name(args).

There are several other features to Go functions. One is multiple return values, which we'll look at next.

Output:

```
1 $ go run functions.go
2 1+2 = 3
3 1+2+3 = 6
```

Multiple Return Values

Go has built-in support for multiple return values. This feature is used often in idiomatic Go, for example to return both result and error values from a function.

```
1 package main
2
3 import "fmt"
4
5 func vals() (int, int) {
6     return 3, 7
7 }
8
9 func main() {
10
11     a, b := vals()
12     fmt.Println(a)
13     fmt.Println(b)
14
15     _, c := vals()
16     fmt.Println(c)
17 }
```

The (int, int) in this function signature shows that the function returns 2 ints.

Here we use the 2 different return values from the call with multiple assignment.

If you only want a subset of the returned values, use the blank identifier _.

Accepting a variable number of arguments is another nice feature of Go functions; we'll look at this next.

Output:

```
1 $ go run multiple-return-values.go
2 3
3 7
4 7
```

Variadic Functions

Variadic functions can be called with any number of trailing arguments. For example, `fmt.Println` is a common variadic function.

```
1 package main
2
3 import "fmt"
4
5 func sum(nums ...int) {
6     fmt.Print(nums, " ")
7     total := 0
8     for _, num := range nums {
9         total += num
10    }
11    fmt.Println(total)
12 }
13
14 func main() {
15
16     sum(1, 2)
17     sum(1, 2, 3)
18
19     nums := []int{1, 2, 3, 4}
20     sum(nums...)
21 }
```

Here's a function that will take an arbitrary number of ints as arguments.

Variadic functions can be called in the usual way with individual arguments.

If you already have multiple args in a slice, apply them to a variadic function using `func(slice...)` like this.

Another key aspect of functions in Go is their ability to form closures, which we'll look at next.

Output:

```
1 $ go run variadic-functions.go
2 [1 2] 3
3 [1 2 3] 6
4 [1 2 3 4] 10
```

Closures

Go supports anonymous functions, which can form closures. Anonymous functions are useful when you want to define a function inline without having to name it.

```
1 package main
2
3 import "fmt"
4
5 func intSeq() func() int {
6     i := 0
7     return func() int {
8         i++
9         return i
10    }
11 }
12
13 func main() {
14
15     nextInt := intSeq()
16
17     fmt.Println(nextInt())
18     fmt.Println(nextInt())
19     fmt.Println(nextInt())
20
21     newInts := intSeq()
22     fmt.Println(newInts())
23 }
```

This function `intSeq` returns another function, which we define anonymously in the body of `intSeq`. The returned function closes over the variable `i` to form a closure.

We call `intSeq`, assigning the result (a function) to `nextInt`. This function value captures its own `i` value, which will be updated each time we call `nextInt`.

See the effect of the closure by calling `nextInt` a few times.

To confirm that the state is unique to that particular function, create and test a new one.

The last feature of functions we'll look at for now is recursion.

Output:

```
1  $ go run closures.go
2  1
3  2
4  3
5  1
```

Recursion

Go supports recursive functions. Here's a classic factorial example.

```
1 package main
2
3 import "fmt"
4
5 func fact(n int) int {
6     if n == 0 {
7         return 1
8     }
9     return n * fact(n-1)
10 }
11
12 func main() {
13     fmt.Println(fact(7))
14 }
```

This fact function calls itself until it reaches the base case of fact(0).

Output:

```
1 $ go run recursion.go
2 5040
```

Pointers

Go supports pointers, allowing you to pass references to values and records within your program.

```
1 package main
2
3 import "fmt"
4
5 func zeroval(ival int) {
6     ival = 0
7 }
8
9 func zeroPtr(iptr *int) {
10     *iptr = 0
11 }
12
13 func main() {
14     i := 1
15     fmt.Println("initial:", i)
16
17     zeroval(i)
18     fmt.Println("zeroval:", i)
19
20     zeroPtr(&i)
21     fmt.Println("zeroPtr:", i)
22
23     fmt.Println("pointer:", &i)
24 }
```

We'll show how pointers work in contrast to values with 2 functions: `zeroval` and `zeroPtr`. `zeroval` has an `int` parameter, so arguments will be passed to it by value. `zeroval` will get a copy of `ival` distinct from the one in the calling function.

`zeroPtr` in contrast has an `*int` parameter, meaning that it takes an `int` pointer. The `*iptr` code in the function body then dereferences the pointer from its memory address to the current value at that address. Assigning a value to a dereferenced pointer changes the value at the referenced address.

The `&i` syntax gives the memory address of `i`, i.e. a pointer to `i`.

Pointers can be printed too.

zeroval doesn't change the *i* in *main*, but *zeroPtr* does because it has a reference to the memory address for that variable.

Output:

```
1 $ go run pointers.go
2 initial: 1
3 zeroval: 1
4 zeroPtr: 0
5 pointer: 0x42131100
```

Structs

Go's structs are typed collections of fields. They're useful for grouping data together to form records.

```
1  package main
2
3  import "fmt"
4
5  type person struct {
6      name string
7      age  int
8  }
9
10 func newPerson(name string) *person {
11
12     p := person{name: name}
13     p.age = 42
14     return &p
15 }
16
17 func main() {
18
19     fmt.Println(person{"Bob", 20})
20
21     fmt.Println(person{name: "Alice", age: 30})
22
23     fmt.Println(person{name: "Fred"})
24
25     fmt.Println(&person{name: "Ann", age: 40})
26
27     fmt.Println(newPerson("Jon"))
28
29     s := person{name: "Sean", age: 50}
30     fmt.Println(s.name)
31
32     sp := &s
33     fmt.Println(sp.age)
34
35     sp.age = 51
```

```
36     fmt.Println(sp.age)
37 }
```

This person struct type has name and age fields.

newPerson constructs a new person struct with the given name

You can safely return a pointer to local variable as a local variable will survive the scope of the function.

This syntax creates a new struct.

You can name the fields when initializing a struct.

Omitted fields will be zero-valued.

An & prefix yields a pointer to the struct.

It's idiomatic to encapsulate new struct creation in constructor functions

Access struct fields with a dot.

You can also use dots with struct pointers - the pointers are automatically dereferenced.

Structs are mutable.

Output:

```
1  $ go run structs.go
2  {Bob 20}
3  {Alice 30}
4  {Fred 0}
5  &{Ann 40}
6  &{Jon 42}
7  Sean
8  50
9  51
```

Methods

Go supports methods defined on struct types.

```
1 package main
2
3 import "fmt"
4
5 type rect struct {
6     width, height int
7 }
8
9 func (r *rect) area() int {
10     return r.width * r.height
11 }
12
13 func (r rect) perim() int {
14     return 2*r.width + 2*r.height
15 }
16
17 func main() {
18     r := rect{width: 10, height: 5}
19
20     fmt.Println("area: ", r.area())
21     fmt.Println("perim:", r.perim())
22
23     rp := &r
24     fmt.Println("area: ", rp.area())
25     fmt.Println("perim:", rp.perim())
26 }
```

This area method has a receiver type of `*rect`.

Methods can be defined for either pointer or value receiver types. Here's an example of a value receiver.

Here we call the 2 methods defined for our struct.

Go automatically handles conversion between values and pointers for method calls. You may want to use a pointer receiver type to avoid copying on method calls or to allow the method to mutate the receiving struct.

Next we'll look at Go's mechanism for grouping and naming related sets of methods: interfaces.

Output:

```
1  $ go run methods.go
2  area: 50
3  perim: 30
4  area: 50
5  perim: 30
```

Interfaces

Interfaces are named collections of method signatures.

```
1 package main
2
3 import (
4     "fmt"
5     "math"
6 )
7
8 type geometry interface {
9     area() float64
10    perim() float64
11 }
12
13 type rect struct {
14     width, height float64
15 }
16
17 type circle struct {
18     radius float64
19 }
20
21 func (r rect) area() float64 {
22     return r.width * r.height
23 }
24
25 func (r rect) perim() float64 {
26     return 2*r.width + 2*r.height
27 }
28
29 func (c circle) area() float64 {
30     return math.Pi * c.radius * c.radius
31 }
32
33 func (c circle) perim() float64 {
34     return 2 * math.Pi * c.radius
35 }
36
37 func measure(g geometry) {
38     fmt.Println(g)
```

```
36     fmt.Println(g.area())
37     fmt.Println(g.perim())
38 }
39
40 func main() {
41     r := rect{width: 3, height: 4}
42     c := circle{radius: 5}
43
44     measure(r)
45     measure(c)
46 }
```

Here's a basic interface for geometric shapes.

For our example we'll implement this interface on *rect* and *circle* types.

To implement an interface in Go, we just need to implement all the methods in the interface. Here we implement *geometry* on *rects*.

The implementation for *circles*.

If a variable has an interface type, then we can call methods that are in the named interface. Here's a generic *measure* function taking advantage of this to work on any *geometry*.

The *circle* and *rect* struct types both implement the *geometry* interface so we can use instances of these structs as arguments to *measure*.

Output:

```
1 $ go run interfaces.go
2 {3 4}
3 12
4 14
5 {5}
6 78.53981633974483
7 31.41592653589793
```

Errors

In Go it's idiomatic to communicate errors via an explicit, separate return value. This contrasts with the exceptions used in languages like Java and Ruby and the overloaded single result / error value sometimes used in C. Go's approach makes it easy to see which functions return errors and to handle them using the same language constructs employed for any other, non-error tasks.

```
1  package main
2
3  import (
4      "errors"
5      "fmt"
6  )
7
8  func f1(arg int) (int, error) {
9      if arg == 42 {
10
11          return -1, errors.New("can't work with 42")
12
13      }
14
15      return arg + 3, nil
16  }
17
18  type argError struct {
19      arg  int
20      prob string
21  }
22
23  func (e *argError) Error() string {
24      return fmt.Sprintf("%d - %s", e.arg, e.prob)
25  }
26
27  func f2(arg int) (int, error) {
28      if arg == 42 {
29
30          return -1, &argError{arg, "can't work with it"}
31      }
32
33      return arg + 3, nil
34  }
```

```

33 }
34
35 func main() {
36
37     for _, i := range []int{7, 42} {
38         if r, e := f1(i); e != nil {
39             fmt.Println("f1 failed:", e)
40         } else {
41             fmt.Println("f1 worked:", r)
42         }
43     }
44     for _, i := range []int{7, 42} {
45         if r, e := f2(i); e != nil {
46             fmt.Println("f2 failed:", e)
47         } else {
48             fmt.Println("f2 worked:", r)
49         }
50     }
51
52     _, e := f2(42)
53     if ae, ok := e.(*argError); ok {
54         fmt.Println(ae.arg)
55         fmt.Println(ae.prob)
56     }
57 }

```

By convention, errors are the last return value and have type error, a built-in interface.

`errors.New` constructs a basic error value with the given error message.

A nil value in the error position indicates that there was no error.

It's possible to use custom types as errors by implementing the `Error()` method on them. Here's a variant on the example above that uses a custom type to explicitly represent an argument error.

In this case we use `&argError` syntax to build a new struct, supplying values for the two fields `arg` and `prob`.

The two loops below test out each of our error-returning functions. Note that the use of an inline error check on the if line is a common idiom in Go code.

If you want to programmatically use the data in a custom error, you'll need to get the error as an instance of the custom error type via type assertion.

Output:

```
1 $ go run errors.go
2 f1 worked: 10
3 f1 failed: can't work with 42
4 f2 worked: 10
5 f2 failed: 42 - can't work with it
6 42
7 can't work with it
```

Goroutines

A goroutine is a lightweight thread of execution.

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func f(from string) {
9     for i := 0; i < 3; i++ {
10         fmt.Println(from, ":", i)
11     }
12 }
13
14 func main() {
15
16     f("direct")
17
18     go f("goroutine")
19
20     go func(msg string) {
21         fmt.Println(msg)
22     }("going")
23
24     time.Sleep(time.Second)
25     fmt.Println("done")
26 }
```

Suppose we have a function call `f(s)`. Here's how we'd call that in the usual way, running it synchronously.

To invoke this function in a goroutine, use `go f(s)`. This new goroutine will execute concurrently with the calling one.

You can also start a goroutine for an anonymous function call.

Our two function calls are running asynchronously in separate goroutines now. Wait for them to finish (for a more robust approach, use a `WaitGroup`).

When we run this program, we see the output of the blocking call first, then the interleaved output of the two goroutines. This interleaving reflects the goroutines being run concurrently by the Go runtime.

Next we'll look at a complement to goroutines in concurrent Go programs: channels.

Output:

```
1 $ go run goroutines.go
2 direct : 0
3 direct : 1
4 direct : 2
5 goroutine : 0
6 going
7 goroutine : 1
8 goroutine : 2
9 done
```

Channels

Channels are the pipes that connect concurrent goroutines. You can send values into channels from one goroutine and receive those values into another goroutine.

```
1 package main
2
3 import "fmt"
4
5 func main() {
6
7     messages := make(chan string)
8
9     go func() { messages <- "ping" }()
10
11     msg := <-messages
12     fmt.Println(msg)
13 }
```

Create a new channel with `make(chan val-type)`. Channels are typed by the values they convey.

Send a value into a channel using the `channel <-` syntax. Here we send “ping” to the `messages` channel we made above, from a new goroutine.

The `<-channel` syntax receives a value from the channel. Here we’ll receive the “ping” message we sent above and print it out.

When we run the program the “ping” message is successfully passed from one goroutine to another via our channel.

By default sends and receives block until both the sender and receiver are ready. This property allowed us to wait at the end of our program for the “ping” message without having to use any other synchronization.

Output:

```
1 $ go run channels.go
2 ping
```

Channel Buffering

By default channels are unbuffered, meaning that they will only accept sends (`chan <-`) if there is a corresponding receive (`<- chan`) ready to receive the sent value. Buffered channels accept a limited number of values without a corresponding receiver for those values.

```
1 package main
2
3 import "fmt"
4
5 func main() {
6
7     messages := make(chan string, 2)
8
9     messages <- "buffered"
10    messages <- "channel"
11
12    fmt.Println(<-messages)
13    fmt.Println(<-messages)
14 }
```

Here we make a channel of strings buffering up to 2 values.

Because this channel is buffered, we can send these values into the channel without a corresponding concurrent receive.

Later we can receive these two values as usual.

Output:

```
1 $ go run channel-buffering.go
2 buffered
3 channel
```

Channel Synchronization

We can use channels to synchronize execution across goroutines. Here's an example of using a blocking receive to wait for a goroutine to finish. When waiting for multiple goroutines to finish, you may prefer to use a WaitGroup.

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func worker(done chan bool) {
9     fmt.Print("working...")
10    time.Sleep(time.Second)
11    fmt.Println("done")
12
13    done <- true
14 }
15
16 func main() {
17
18     done := make(chan bool, 1)
19     go worker(done)
20
21     <- done
22 }
```

This is the function we'll run in a goroutine. The done channel will be used to notify another goroutine that this function's work is done.

Send a value to notify that we're done.

Start a worker goroutine, giving it the channel to notify on.

Block until we receive a notification from the worker on the channel.

If you removed the <- done line from this program, the program would exit before the worker even started.

Output:

```
1 $ go run channel-synchronization.go
2 working...done
```

Channel Directions

When using channels as function parameters, you can specify if a channel is meant to only send or receive values. This specificity increases the type-safety of the program.

```
1 package main
2
3 import "fmt"
4
5 func ping(pings chan<- string, msg string) {
6     pings <- msg
7 }
8
9 func pong(pings <-chan string, pongs chan<- string) {
10     msg := <-pings
11     pongs <- msg
12 }
13
14 func main() {
15     pings := make(chan string, 1)
16     pongs := make(chan string, 1)
17     ping(pings, "passed message")
18     pong(pings, pongs)
19     fmt.Println(<-pongs)
20 }
```

This ping function only accepts a channel for sending values. It would be a compile-time error to try to receive on this channel.

The pong function accepts one channel for receives (pings) and a second for sends (pongs).

Output:

```
1 $ go run channel-directions.go
2 passed message
```

Select

Go's select lets you wait on multiple channel operations. Combining goroutines and channels with select is a powerful feature of Go.

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func main() {
9
10     c1 := make(chan string)
11     c2 := make(chan string)
12
13     go func() {
14         time.Sleep(1 * time.Second)
15         c1 <- "one"
16     }()
17     go func() {
18         time.Sleep(2 * time.Second)
19         c2 <- "two"
20     }()
21
22     for i := 0; i < 2; i++ {
23         select {
24             case msg1 := <-c1:
25                 fmt.Println("received", msg1)
26             case msg2 := <-c2:
27                 fmt.Println("received", msg2)
28         }
29     }
30 }
```

For our example we'll select across two channels.

Each channel will receive a value after some amount of time, to simulate e.g. blocking RPC operations executing in concurrent goroutines.

We'll use select to await both of these values simultaneously, printing each one as it arrives.

We receive the values "one" and then "two" as expected.

Note that the total execution time is only ~2 seconds since both the 1 and 2 second Sleeps execute concurrently.

Output:

```
1 $ time go run select.go
2 received one
3 received two
4 Note that the total execution time is only ~2 seconds since both the 1 and 2 second \
5 Sleeps execute concurrently.
6
7 real    0m2.245s
```

Timeouts

Timeouts are important for programs that connect to external resources or that otherwise need to bound execution time. Implementing timeouts in Go is easy and elegant thanks to channels and *select*.

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func main() {
9
10     c1 := make(chan string, 1)
11     go func() {
12         time.Sleep(2 * time.Second)
13         c1 <- "result 1"
14     }()
15
16     select {
17     case res := <-c1:
18         fmt.Println(res)
19     case <-time.After(1 * time.Second):
20         fmt.Println("timeout 1")
21     }
22
23     c2 := make(chan string, 1)
24     go func() {
25         time.Sleep(2 * time.Second)
26         c2 <- "result 2"
27     }()
28     select {
29     case res := <-c2:
30         fmt.Println(res)
31     case <-time.After(3 * time.Second):
32         fmt.Println("timeout 2")
33     }
34 }
```

For our example, suppose we're executing an external call that returns its result on a channel `c1` after 2s. Note that the channel is buffered, so the send in the goroutine is nonblocking. This is a common pattern to prevent goroutine leaks in case the channel is never read.

Here's the select implementing a timeout. `res := <-c1` awaits the result and `<-Time.After` awaits a value to be sent after the timeout of 1s. Since select proceeds with the first receive that's ready, we'll take the timeout case if the operation takes more than the allowed 1s.

If we allow a longer timeout of 3s, then the receive from `c2` will succeed and we'll print the result.

Running this program shows the first operation timing out and the second succeeding.

Output:

```
1 $ go run timeouts.go
2 timeout 1
3 result 2
```

Non-Blocking Channel Operations

Basic sends and receives on channels are blocking. However, we can use select with a default clause to implement non-blocking sends, receives, and even non-blocking multi-way selects.

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     messages := make(chan string)
7     signals := make(chan bool)
8
9     select {
10    case msg := <-messages:
11        fmt.Println("received message", msg)
12    default:
13        fmt.Println("no message received")
14    }
15
16    msg := "hi"
17    select {
18    case messages <- msg:
19        fmt.Println("sent message", msg)
20    default:
21        fmt.Println("no message sent")
22    }
23
24    select {
25    case msg := <-messages:
26        fmt.Println("received message", msg)
27    case sig := <-signals:
28        fmt.Println("received signal", sig)
29    default:
30        fmt.Println("no activity")
31    }
32 }
```

Here's a non-blocking receive. If a value is available on messages then select will take the <-messages case with that value. If not it will immediately take the default case.

A non-blocking send works similarly. Here `msg` cannot be sent to the `messages` channel, because the channel has no buffer and there is no receiver. Therefore the default case is selected.

We can use multiple cases above the default clause to implement a multi-way non-blocking select. Here we attempt non-blocking receives on both `messages` and `signals`.

Output:

```
1 $ go run non-blocking-channel-operations.go
2 no message received
3 no message sent
4 no activity
```

Closing Channels

Closing a channel indicates that no more values will be sent on it. This can be useful to communicate completion to the channel's receivers.

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      jobs := make(chan int, 5)
7      done := make(chan bool)
8
9      go func() {
10         for {
11             j, more := <- jobs
12             if more {
13                 fmt.Println("received job", j)
14             } else {
15                 fmt.Println("received all jobs")
16                 done <- true
17                 return
18             }
19         }
20     }()
21
22     for j := 1; j <= 3; j++ {
23         jobs <- j
24         fmt.Println("sent job", j)
25     }
26     close(jobs)
27     fmt.Println("sent all jobs")
28
29     <- done
30 }
```

In this example we'll use a jobs channel to communicate work to be done from the main() goroutine to a worker goroutine. When we have no more jobs for the worker we'll close the jobs channel.

Here's the worker goroutine. It repeatedly receives from jobs with `j`, `more := <-jobs`. In this special 2-value form of receive, the `more` value will be `false` if jobs has been closed and all values in the channel have already been received. We use this to notify on `done` when we've worked all our jobs.

This sends 3 jobs to the worker over the jobs channel, then closes it.

We await the worker using the synchronization approach we saw earlier.

The idea of closed channels leads naturally to our next example: range over channels.

Output:

```
1 $ go run closing-channels.go
2 sent job 1
3 received job 1
4 sent job 2
5 received job 2
6 sent job 3
7 received job 3
8 sent all jobs
9 received all jobs
```

Range over Channels

In a previous example we saw how `for` and `range` provide iteration over basic data structures. We can also use this syntax to iterate over values received from a channel.

```
1 package main
2
3 import "fmt"
4
5 func main() {
6
7     queue := make(chan string, 2)
8     queue <- "one"
9     queue <- "two"
10    close(queue)
11
12    for elem := range queue {
13        fmt.Println(elem)
14    }
15 }
```

We'll iterate over 2 values in the queue channel.

This range iterates over each element as it's received from queue. Because we closed the channel above, the iteration terminates after receiving the 2 elements.

This example also showed that it's possible to close a non-empty channel but still have the remaining values be received.

Output:

```
1 $ go run range-over-channels.go
2 one
3 two
```

Timers

We often want to execute Go code at some point in the future, or repeatedly at some interval. Go's built-in timer and ticker features make both of these tasks easy. We'll look first at timers and then at tickers.

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func main() {
9
10     timer1 := time.NewTimer(2 * time.Second)
11
12     <-timer1.C
13     fmt.Println("Timer 1 fired")
14
15     timer2 := time.NewTimer(time.Second)
16     go func() {
17         <-timer2.C
18         fmt.Println("Timer 2 fired")
19     }()
20     stop2 := timer2.Stop()
21     if stop2 {
22         fmt.Println("Timer 2 stopped")
23     }
24
25     time.Sleep(2 * time.Second)
26 }
```

Timers represent a single event in the future. You tell the timer how long you want to wait, and it provides a channel that will be notified at that time. This timer will wait 2 seconds.

The `<-timer1.C` blocks on the timer's channel C until it sends a value indicating that the timer fired.

If you just wanted to wait, you could have used `time.Sleep`. One reason a timer may be useful is that you can cancel the timer before it fires. Here's an example of that.

Give the timer2 enough time to fire, if it ever was going to, to show it is in fact stopped.

The first timer will fire ~2s after we start the program, but the second should be stopped before it has a chance to fire.

Output:

```
1  $ go run timers.go
2  Timer 1 fired
3  Timer 2 stopped
```

Tickers

Timers are for when you want to do something once in the future - tickers are for when you want to do something repeatedly at regular intervals. Here's an example of a ticker that ticks periodically until we stop it.

```
1  package main
2
3  import (
4      "fmt"
5      "time"
6  )
7
8  func main() {
9
10     ticker := time.NewTicker(500 * time.Millisecond)
11     done := make(chan bool)
12
13     go func() {
14         for {
15             select {
16                 case <-done:
17                     return
18                 case t := <-ticker.C:
19                     fmt.Println("Tick at", t)
20             }
21         }
22     }()
23
24     time.Sleep(1600 * time.Millisecond)
25     ticker.Stop()
26     done <- true
27     fmt.Println("Ticker stopped")
28 }
```

Tickers use a similar mechanism to timers: a channel that is sent values. Here we'll use the select builtin on the channel to await the values as they arrive every 500ms.

Tickers can be stopped like timers. Once a ticker is stopped it won't receive any more values on its channel. We'll stop ours after 1600ms.

When we run this program the ticker should tick 3 times before we stop it.

Output:

```
1 $ go run tickers.go
2 Tick at 2012-09-23 11:29:56.487625 -0700 PDT
3 Tick at 2012-09-23 11:29:56.988063 -0700 PDT
4 Tick at 2012-09-23 11:29:57.488076 -0700 PDT
5 Ticker stopped
```

Worker Pools

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

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func worker(id int, jobs <-chan int, results chan<- int) {
9     for j := range jobs {
10         fmt.Println("worker", id, "started job", j)
11         time.Sleep(time.Second)
12         fmt.Println("worker", id, "finished job", j)
13         results <- j * 2
14     }
15 }
16
17 func main() {
18
19     const numJobs = 5
20     jobs := make(chan int, numJobs)
21     results := make(chan int, numJobs)
22
23     for w := 1; w <= 3; w++ {
24         go worker(w, jobs, results)
25     }
26
27     for j := 1; j <= numJobs; j++ {
28         jobs <- j
29     }
30     close(jobs)
31
32     for a := 1; a <= numJobs; a++ {
33         <-results
34     }
35 }
```

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.

In order to use our pool of workers we need to send them work and collect their results. We make 2 channels for this.

This starts up 3 workers, initially blocked because there are no jobs yet.

Here we send 5 jobs and then close that channel to indicate that's all the work we have.

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.

Our running program shows the 5 jobs being executed by various workers. The program only takes about 2 seconds despite doing about 5 seconds of total work because there are 3 workers operating concurrently.

Output:

```
1  $ time go run worker-pools.go
2  worker 1 started  job 1
3  worker 2 started  job 2
4  worker 3 started  job 3
5  worker 1 finished job 1
6  worker 1 started  job 4
7  worker 2 finished job 2
8  worker 2 started  job 5
9  worker 3 finished job 3
10 worker 1 finished job 4
11 worker 2 finished job 5
12 real    0m2.358s
```

WaitGroups

To wait for multiple goroutines to finish, we can use a wait group.

```
1 package main
2
3 import (
4     "fmt"
5     "sync"
6     "time"
7 )
8
9 func worker(id int, wg *sync.WaitGroup) {
10
11     defer wg.Done()
12
13     fmt.Printf("Worker %d starting\n", id)
14
15     time.Sleep(time.Second)
16     fmt.Printf("Worker %d done\n", id)
17 }
18
19 func main() {
20
21     var wg sync.WaitGroup
22
23     for i := 1; i <= 5; i++ {
24         wg.Add(1)
25         go worker(i, &wg)
26     }
27
28     wg.Wait()
29 }
```

This is the function we'll run in every goroutine. Note that a WaitGroup must be passed to functions by pointer.

On return, notify the WaitGroup that we're done.

Sleep to simulate an expensive task.

This WaitGroup is used to wait for all the goroutines launched here to finish.

Launch several goroutines and increment the WaitGroup counter for each.

Block until the WaitGroup counter goes back to 0; all the workers notified they're done.

The order of workers starting up and finishing is likely to be different for each invocation.

Output:

```
1  $ go run waitgroups.go
2  Worker 5 starting
3  Worker 3 starting
4  Worker 4 starting
5  Worker 1 starting
6  Worker 2 starting
7  Worker 4 done
8  Worker 1 done
9  Worker 2 done
10 Worker 5 done
11 Worker 3 done
```

Rate Limiting

Rate limiting is an important mechanism for controlling resource utilization and maintaining quality of service. Go elegantly supports rate limiting with goroutines, channels, and tickers.

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func main() {
9
10     requests := make(chan int, 5)
11     for i := 1; i <= 5; i++ {
12         requests <- i
13     }
14     close(requests)
15
16     limiter := time.Tick(200 * time.Millisecond)
17
18     for req := range requests {
19         <-limiter
20         fmt.Println("request", req, time.Now())
21     }
22
23     burstyLimiter := make(chan time.Time, 3)
24
25     for i := 0; i < 3; i++ {
26         burstyLimiter <- time.Now()
27     }
28
29     go func() {
30         for t := range time.Tick(200 * time.Millisecond) {
31             burstyLimiter <- t
32         }
33     }()
34 }
```

```

35     burstyRequests := make(chan int, 5)
36     for i := 1; i <= 5; i++ {
37         burstyRequests <- i
38     }
39     close(burstyRequests)
40     for req := range burstyRequests {
41         <-burstyLimiter
42         fmt.Println("request", req, time.Now())
43     }
44 }

```

First we'll look at basic rate limiting. Suppose we want to limit our handling of incoming requests. We'll serve these requests off a channel of the same name.

This limiter channel will receive a value every 200 milliseconds. This is the regulator in our rate limiting scheme.

By blocking on a receive from the limiter channel before serving each request, we limit ourselves to 1 request every 200 milliseconds.

We may want to allow short bursts of requests in our rate limiting scheme while preserving the overall rate limit. We can accomplish this by buffering our limiter channel. This `burstyLimiter` channel will allow bursts of up to 3 events.

Fill up the channel to represent allowed bursting.

Every 200 milliseconds we'll try to add a new value to `burstyLimiter`, up to its limit of 3.

Now simulate 5 more incoming requests. The first 3 of these will benefit from the burst capability of `burstyLimiter`.

Running our program we see the first batch of requests handled once every ~200 milliseconds as desired.

For the second batch of requests we serve the first 3 immediately because of the burstable rate limiting, then serve the remaining 2 with ~200ms delays each.

Output:

```

1  $ go run rate-limiting.go
2  request 1 2012-10-19 00:38:18.687438 +0000 UTC
3  request 2 2012-10-19 00:38:18.887471 +0000 UTC
4  request 3 2012-10-19 00:38:19.087238 +0000 UTC
5  request 4 2012-10-19 00:38:19.287338 +0000 UTC
6  request 5 2012-10-19 00:38:19.487331 +0000 UTC
7  For the second batch of requests we serve the first 3 immediately because of the bur\
8  stable rate limiting, then serve the remaining 2 with ~200ms delays each.
9

```

```
10 request 1 2012-10-19 00:38:20.487578 +0000 UTC
11 request 2 2012-10-19 00:38:20.487645 +0000 UTC
12 request 3 2012-10-19 00:38:20.487676 +0000 UTC
13 request 4 2012-10-19 00:38:20.687483 +0000 UTC
14 request 5 2012-10-19 00:38:20.887542 +0000 UTC
```

Atomic Counters

The primary mechanism for managing state in Go is communication over channels. We saw this for example with worker pools. There are a few other options for managing state though. Here we'll look at using the `sync/atomic` package for atomic counters accessed by multiple goroutines.

```
1 package main
2
3 import (
4     "fmt"
5     "sync"
6     "sync/atomic"
7 )
8
9 func main() {
10
11     var ops uint64
12
13     var wg sync.WaitGroup
14
15     for i := 0; i < 50; i++ {
16         wg.Add(1)
17
18         go func() {
19             for c := 0; c < 1000; c++ {
20
21                 atomic.AddUint64(&ops, 1)
22             }
23             wg.Done()
24         }()
25     }
26
27     wg.Wait()
28
29     fmt.Println("ops:", ops)
30 }
```

We'll use an unsigned integer to represent our (always-positive) counter.

A `WaitGroup` will help us wait for all goroutines to finish their work.

We'll start 50 goroutines that each increment the counter exactly 1000 times.

To atomically increment the counter we use `AddUint64`, giving it the memory address of our ops counter with the `&` syntax.

Wait until all the goroutines are done.

It's safe to access ops now because we know no other goroutine is writing to it. Reading atomics safely while they are being updated is also possible, using functions like `atomic.LoadUint64`.

We expect to get exactly 50,000 operations. Had we used the non-atomic `ops++` to increment the counter, we'd likely get a different number, changing between runs, because the goroutines would interfere with each other. Moreover, we'd get data race failures when running with the `-race` flag.

Next we'll look at mutexes, another tool for managing state.

Output:

```
1 $ go run atomic-counters.go
2 ops: 50000
```

Mutexes

In the previous example we saw how to manage simple counter state using atomic operations. For more complex state we can use a mutex to safely access data across multiple goroutines.

```
1  package main
2
3  import (
4      "fmt"
5      "math/rand"
6      "sync"
7      "sync/atomic"
8      "time"
9  )
10
11 func main() {
12
13     var state = make(map[int]int)
14
15     var mutex = &sync.Mutex{}
16
17     var readOps uint64
18     var writeOps uint64
19
20     for r := 0; r < 100; r++ {
21         go func() {
22             total := 0
23             for {
24
25                 key := rand.Intn(5)
26                 mutex.Lock()
27                 total += state[key]
28                 mutex.Unlock()
29                 atomic.AddUint64(&readOps, 1)
30
31                 time.Sleep(time.Millisecond)
32             }
33         }()
34     }
```

```

35
36     for w := 0; w < 10; w++ {
37         go func() {
38             for {
39                 key := rand.Intn(5)
40                 val := rand.Intn(100)
41                 mutex.Lock()
42                 state[key] = val
43                 mutex.Unlock()
44                 atomic.AddUint64(&writeOps, 1)
45                 time.Sleep(time.Millisecond)
46             }
47         }()
48     }
49
50     time.Sleep(time.Second)
51
52     readOpsFinal := atomic.LoadUint64(&readOps)
53     fmt.Println("readOps:", readOpsFinal)
54     writeOpsFinal := atomic.LoadUint64(&writeOps)
55     fmt.Println("writeOps:", writeOpsFinal)
56
57     mutex.Lock()
58     fmt.Println("state:", state)
59     mutex.Unlock()
60 }

```

For our example the state will be a map.

This mutex will synchronize access to state.

We'll keep track of how many read and write operations we do.

Here we start 100 goroutines to execute repeated reads against the state, once per millisecond in each goroutine.

For each read we pick a key to access, Lock() the mutex to ensure exclusive access to the state, read the value at the chosen key, Unlock() the mutex, and increment the readOps count.

Wait a bit between reads.

We'll also start 10 goroutines to simulate writes, using the same pattern we did for reads.

Let the 10 goroutines work on the state and mutex for a second.

Take and report final operation counts.

With a final lock of state, show how it ended up.

Running the program shows that we executed about 90,000 total operations against our mutex-synchronized state.

Next we'll look at implementing this same state management task using only goroutines and channels.

Output:

```
1 $ go run mutexes.go
2 readOps: 83285
3 writeOps: 8320
4 state: map[1:97 4:53 0:33 2:15 3:2]
```

Stateful Goroutines

In the previous example we used explicit locking with mutexes to synchronize access to shared state across multiple goroutines. Another option is to use the built-in synchronization features of goroutines and channels to achieve the same result. This channel-based approach aligns with Go's ideas of sharing memory by communicating and having each piece of data owned by exactly 1 goroutine.

```
1 package main
2
3 import (
4     "fmt"
5     "math/rand"
6     "sync/atomic"
7     "time"
8 )
9
10 type readOp struct {
11     key int
12     resp chan int
13 }
14 type writeOp struct {
15     key int
16     val int
17     resp chan bool
18 }
19
20 func main() {
21
22     var readOps uint64
23     var writeOps uint64
24
25     reads := make(chan readOp)
26     writes := make(chan writeOp)
27
28     go func() {
29         var state = make(map[int]int)
30         for {
31             select {
```

```

32         case read := <-reads:
33             read.resp <- state[read.key]
34         case write := <-writes:
35             state[write.key] = write.val
36             write.resp <- true
37     }
38 }
39 }()
40
41 for r := 0; r < 100; r++ {
42     go func() {
43         for {
44             read := readOp{
45                 key: rand.Intn(5),
46                 resp: make(chan int)}
47             reads <- read
48             <-read.resp
49             atomic.AddUint64(&readOps, 1)
50             time.Sleep(time.Millisecond)
51         }
52     }()
53 }
54
55 for w := 0; w < 10; w++ {
56     go func() {
57         for {
58             write := writeOp{
59                 key: rand.Intn(5),
60                 val: rand.Intn(100),
61                 resp: make(chan bool)}
62             writes <- write
63             <-write.resp
64             atomic.AddUint64(&writeOps, 1)
65             time.Sleep(time.Millisecond)
66         }
67     }()
68 }
69
70 time.Sleep(time.Second)
71
72 readOpsFinal := atomic.LoadUint64(&readOps)
73 fmt.Println("readOps:", readOpsFinal)
74 writeOpsFinal := atomic.LoadUint64(&writeOps)

```

```
75     fmt.Println("writeOps:", writeOpsFinal)
76 }
```

In this example our state will be owned by a single goroutine. This will guarantee that the data is never corrupted with concurrent access. In order to read or write that state, other goroutines will send messages to the owning goroutine and receive corresponding replies. These `readOp` and `writeOp` structs encapsulate those requests and a way for the owning goroutine to respond.

As before we'll count how many operations we perform.

The reads and writes channels will be used by other goroutines to issue read and write requests, respectively.

Here is the goroutine that owns the state, which is a map as in the previous example but now private to the stateful goroutine. This goroutine repeatedly selects on the reads and writes channels, responding to requests as they arrive. A response is executed by first performing the requested operation and then sending a value on the response channel `resp` to indicate success (and the desired value in the case of reads).

This starts 100 goroutines to issue reads to the state-owning goroutine via the reads channel. Each read requires constructing a `readOp`, sending it over the reads channel, and then receiving the result over the provided `resp` channel.

We start 10 writes as well, using a similar approach.

Let the goroutines work for a second.

Finally, capture and report the op counts.

Running our program shows that the goroutine-based state management example completes about 80,000 total operations.

For this particular case the goroutine-based approach was a bit more involved than the mutex-based one. It might be useful in certain cases though, for example where you have other channels involved or when managing multiple such mutexes would be error-prone. You should use whichever approach feels most natural, especially with respect to understanding the correctness of your program.

Output:

```
1  $ go run stateful-goroutines.go
2  readOps: 71708
3  writeOps: 7177
```

Sorting

Go's sort package implements sorting for builtins and user-defined types. We'll look at sorting for builtins first.

```
1 package main
2
3 import (
4     "fmt"
5     "sort"
6 )
7
8 func main() {
9
10     strs := []string{"c", "a", "b"}
11     sort.Strings(strs)
12     fmt.Println("Strings:", strs)
13
14     ints := []int{7, 2, 4}
15     sort.Ints(ints)
16     fmt.Println("Ints:  ", ints)
17
18     s := sort.IntsAreSorted(ints)
19     fmt.Println("Sorted: ", s)
20 }
```

Sort methods are specific to the builtin type; here's an example for strings. Note that sorting is in-place, so it changes the given slice and doesn't return a new one.

An example of sorting ints.

We can also use sort to check if a slice is already in sorted order.

Running our program prints the sorted string and int slices and true as the result of our AreSorted test.

Output:

```
1 $ go run sorting.go
2 Strings: [a b c]
3 Ints:    [2 4 7]
4 Sorted:  true
```

Sorting by Functions

Sometimes we'll want to sort a collection by something other than its natural order. For example, suppose we wanted to sort strings by their length instead of alphabetically. Here's an example of custom sorts in Go.

```
1 package main
2
3 import (
4     "fmt"
5     "sort"
6 )
7
8 type byLength []string
9
10 func (s byLength) Len() int {
11     return len(s)
12 }
13 func (s byLength) Swap(i, j int) {
14     s[i], s[j] = s[j], s[i]
15 }
16 func (s byLength) Less(i, j int) bool {
17     return len(s[i]) < len(s[j])
18 }
19
20 func main() {
21     fruits := []string{"peach", "banana", "kiwi"}
22     sort.Sort(byLength(fruits))
23     fmt.Println(fruits)
24 }
```

In order to sort by a custom function in Go, we need a corresponding type. Here we've created a `byLength` type that is just an alias for the builtin `[]string` type.

We implement `sort.Interface` - `Len`, `Less`, and `Swap` - on our type so we can use the `sort` package's generic `Sort` function. `Len` and `Swap` will usually be similar across types and `Less` will hold the actual custom sorting logic. In our case we want to sort in order of increasing string length, so we use `len(s[i])` and `len(s[j])` here.

With all of this in place, we can now implement our custom sort by converting the original fruits slice to `byLength`, and then use `sort.Sort` on that typed slice.

Running our program shows a list sorted by string length, as desired.

By following this same pattern of creating a custom type, implementing the three Interface methods on that type, and then calling `sort.Sort` on a collection of that custom type, we can sort Go slices by arbitrary functions.

Output:

```
1 $ go run sorting-by-functions.go
2 [kiwi peach banana]
```

Panic

A panic typically means something went unexpectedly wrong. Mostly we use it to fail fast on errors that shouldn't occur during normal operation, or that we aren't prepared to handle gracefully.

```
1 package main
2
3 import "os"
4
5 func main() {
6
7     panic("a problem")
8
9     _, err := os.Create("/tmp/file")
10    if err != nil {
11        panic(err)
12    }
13 }
```

We'll use panic throughout this site to check for unexpected errors. This is the only program on the site designed to panic.

A common use of panic is to abort if a function returns an error value that we don't know how to (or want to) handle. Here's an example of panicking if we get an unexpected error when creating a new file.

Running this program will cause it to panic, print an error message and goroutine traces, and exit with a non-zero status.

Note that unlike some languages which use exceptions for handling of many errors, in Go it is idiomatic to use error-indicating return values wherever possible.

Output:

```
1  $ go run panic.go
2  panic: a problem
3  goroutine 1 [running]:
4  main.main()
5      /.../panic.go:12 +0x47
6  ...
7  exit status 2
```

Defer

Defer is used to ensure that a function call is performed later in a program's execution, usually for purposes of cleanup. defer is often used where e.g. ensure and finally would be used in other languages.

```
1  package main
2
3  import (
4      "fmt"
5      "os"
6  )
7
8  func main() {
9
10     f := createFile("/tmp/defer.txt")
11     defer closeFile(f)
12     writeFile(f)
13 }
14
15 func createFile(p string) *os.File {
16     fmt.Println("creating")
17     f, err := os.Create(p)
18     if err != nil {
19         panic(err)
20     }
21     return f
22 }
23
24 func writeFile(f *os.File) {
25     fmt.Println("writing")
26     fmt.Fprintln(f, "data")
27
28 }
29
30 func closeFile(f *os.File) {
31     fmt.Println("closing")
32     err := f.Close()
33 }
```

```
34     if err != nil {  
35         fmt.Fprintf(os.Stderr, "error: %v\n", err)  
36         os.Exit(1)  
37     }  
38 }
```

Suppose we wanted to create a file, write to it, and then close when we're done. Here's how we could do that with defer.

Immediately after getting a file object with `createFile`, we defer the closing of that file with `closeFile`. This will be executed at the end of the enclosing function (`main`), after `writeFile` has finished.

It's important to check for errors when closing a file, even in a deferred function.

Running the program confirms that the file is closed after being written.

Output:

```
1 $ go run defer.go  
2 creating  
3 writing  
4 closing
```

Collection Functions

We often need our programs to perform operations on collections of data, like selecting all items that satisfy a given predicate or mapping all items to a new collection with a custom function.

In some languages it's idiomatic to use generic data structures and algorithms. Go does not support generics; in Go it's common to provide collection functions if and when they are specifically needed for your program and data types.

Here are some example collection functions for slices of strings. You can use these examples to build your own functions. Note that in some cases it may be clearest to just inline the collection-manipulating code directly, instead of creating and calling a helper function.

```
1 package main
2
3 import (
4     "fmt"
5     "strings"
6 )
7
8 func Index(vs []string, t string) int {
9     for i, v := range vs {
10         if v == t {
11             return i
12         }
13     }
14     return -1
15 }
16
17 func Include(vs []string, t string) bool {
18     return Index(vs, t) >= 0
19 }
20
21 func Any(vs []string, f func(string) bool) bool {
22     for _, v := range vs {
23         if f(v) {
24             return true
25         }
26     }
27     return false
28 }
```

```
28 }
29
30 func All(vs []string, f func(string) bool) bool {
31     for _, v := range vs {
32         if !f(v) {
33             return false
34         }
35     }
36     return true
37 }
38
39 func Filter(vs []string, f func(string) bool) []string {
40     vsf := make([]string, 0)
41     for _, v := range vs {
42         if f(v) {
43             vsf = append(vsf, v)
44         }
45     }
46     return vsf
47 }
48
49 func Map(vs []string, f func(string) string) []string {
50     vsm := make([]string, len(vs))
51     for i, v := range vs {
52         vsm[i] = f(v)
53     }
54     return vsm
55 }
56
57 func main() {
58
59     var strs = []string{"peach", "apple", "pear", "plum"}
60
61     fmt.Println(Index(strs, "pear"))
62
63     fmt.Println(Include(strs, "grape"))
64
65     fmt.Println(Any(strs, func(v string) bool {
66         return strings.HasPrefix(v, "p")
67     })))
68
69     fmt.Println(All(strs, func(v string) bool {
70         return strings.HasPrefix(v, "p")
```

```
71     )))
72
73     fmt.Println(Filter(strs, func(v string) bool {
74         return strings.Contains(v, "e")
75     })))
76
77     fmt.Println(Map(strs, strings.ToUpper))
78
79 }
```

Index returns the first index of the target string t, or -1 if no match is found.

Include returns true if the target string t is in the slice.

Any returns true if one of the strings in the slice satisfies the predicate f.

All returns true if all of the strings in the slice satisfy the predicate f.

Filter returns a new slice containing all strings in the slice that satisfy the predicate f.

Map returns a new slice containing the results of applying the function f to each string in the original slice.

Here we try out our various collection functions.

The above examples all used anonymous functions, but you can also use named functions of the correct type.

Output:

```
1  $ go run collection-functions.go
2
3  false
4  true
5  false
6  [peach apple pear]
7  [PEACH APPLE PEAR PLUM]
```

String Functions

The standard library's strings package provides many useful string-related functions. Here are some examples to give you a sense of the package.

```
1 package main
2
3 import (
4     "fmt"
5     s "strings"
6 )
7
8 var p = fmt.Println
9
10 func main() {
11
12     p("Contains: ", s.Contains("test", "es"))
13     p("Count: ", s.Count("test", "t"))
14     p("HasPrefix: ", s.HasPrefix("test", "te"))
15     p("HasSuffix: ", s.HasSuffix("test", "st"))
16     p("Index: ", s.Index("test", "e"))
17     p("Join: ", s.Join([]string{"a", "b"}, "-"))
18     p("Repeat: ", s.Repeat("a", 5))
19     p("Replace: ", s.Replace("foo", "o", "0", -1))
20     p("Replace: ", s.Replace("foo", "o", "0", 1))
21     p("Split: ", s.Split("a-b-c-d-e", "-"))
22     p("ToLower: ", s.ToLower("TEST"))
23     p("ToUpper: ", s.ToUpper("test"))
24     p()
25
26     p("Len: ", len("hello"))
27     p("Char: ", "hello"[1])
28 }
```

We alias `fmt.Println` to a shorter name as we'll use it a lot below.

Here's a sample of the functions available in strings. Since these are functions from the package, not methods on the string object itself, we need pass the string in question as the first argument to the function. You can find more functions in the strings package docs.

Not part of strings, but worth mentioning here, are the mechanisms for getting the length of a string in bytes and getting a byte by index.

Note that `len` and indexing above work at the byte level. Go uses UTF-8 encoded strings, so this is often useful as-is. If you're working with potentially multi-byte characters you'll want to use encoding-aware operations. See [strings](#), [bytes](#), [runes](#) and [characters](#) in Go for more information.

Output:

```
1 $ go run string-functions.go
2 Contains:  true
3 Count:    2
4 HasPrefix: true
5 HasSuffix: true
6 Index:    1
7 Join:     a-b
8 Repeat:   aaaaa
9 Replace:  f00
10 Replace: f0o
11 Split:   [a b c d e]
12 ToLower: test
13 ToUpper: TEST
14 Len:     5
15 Char: 101
```

String Formatting

Go offers excellent support for string formatting in the printf tradition. Here are some examples of common string formatting tasks.

```
1 package main
2
3 import (
4     "fmt"
5     "os"
6 )
7
8 type point struct {
9     x, y int
10 }
11
12 func main() {
13
14     p := point{1, 2}
15     fmt.Printf("%v\n", p)
16
17     fmt.Printf("%+v\n", p)
18
19     fmt.Printf("%#v\n", p)
20
21     fmt.Printf("%T\n", p)
22
23     fmt.Printf("%t\n", true)
24
25     fmt.Printf("%d\n", 123)
26
27     fmt.Printf("%b\n", 14)
28
29     fmt.Printf("%c\n", 33)
30
31     fmt.Printf("%x\n", 456)
32
33     fmt.Printf("%f\n", 78.9)
34
```

```

35     fmt.Printf("%e\n", 123400000.0)
36     fmt.Printf("%E\n", 123400000.0)
37
38     fmt.Printf("%s\n", "\"string\"")
39
40     fmt.Printf("%q\n", "\"string\"")
41
42     fmt.Printf("%x\n", "hex this")
43
44     fmt.Printf("%p\n", &p)
45
46     fmt.Printf("|%6d|%6d|\n", 12, 345)
47
48     fmt.Printf("|%6.2f|%6.2f|\n", 1.2, 3.45)
49
50     fmt.Printf("|%-6.2f|%-6.2f|\n", 1.2, 3.45)
51
52     fmt.Printf("|%6s|%6s|\n", "foo", "b")
53
54     fmt.Printf("|%-6s|%-6s|\n", "foo", "b")
55
56     s := fmt.Sprintf("a %s", "string")
57     fmt.Println(s)
58
59     fmt.Fprintf(os.Stderr, "an %s\n", "error")
60 }

```

Go offers several printing “verbs” designed to format general Go values. For example, this prints an instance of our point struct.

If the value is a struct, the `%+v` variant will include the struct’s field names.

The `%#v` variant prints a Go syntax representation of the value, i.e. the source code snippet that would produce that value.

To print the type of a value, use `%T`.

Formatting booleans is straight-forward.

There are many options for formatting integers. Use `%d` for standard, base-10 formatting.

This prints a binary representation.

This prints the character corresponding to the given integer.

`%x` provides hex encoding.

There are also several formatting options for floats. For basic decimal formatting use `%f`.

`%e` and `%E` format the float in (slightly different versions of) scientific notation.

For basic string printing use `%s`.

To double-quote strings as in Go source, use `%q`.

As with integers seen earlier, `%x` renders the string in base-16, with two output characters per byte of input.

To print a representation of a pointer, use `%p`.

When formatting numbers you will often want to control the width and precision of the resulting figure. To specify the width of an integer, use a number after the `%` in the verb. By default the result will be right-justified and padded with spaces.

You can also specify the width of printed floats, though usually you'll also want to restrict the decimal precision at the same time with the `width.precision` syntax.

To left-justify, use the `-` flag.

You may also want to control width when formatting strings, especially to ensure that they align in table-like output. For basic right-justified width.

To left-justify use the `-` flag as with numbers.

So far we've seen `Printf`, which prints the formatted string to `os.Stdout`. `Sprintf` formats and returns a string without printing it anywhere.

Output:

```

1  $ go run string-formatting.go
2  {1 2}
3  {x:1 y:2}
4  main.point{x:1, y:2}
5  main.point
6  true
7  123
8  1110
9  !
10 1c8
11 78.900000
12 1.234000e+08
13 1.234000E+08
14 "string"
15 "\"string\""
16 6865782074686973
17 0x42135100
18 |    12|   345|
19 |   1.20|   3.45|

```

```
20 |1.20 |3.45 |
21 |  foo|    b|
22 |foo  |b    |
23 a string
24 an error
```

Regular Expressions

Go offers built-in support for regular expressions. Here are some examples of common regexp-related tasks in Go.

```
1 package main
2
3 import (
4     "bytes"
5     "fmt"
6     "regexp"
7 )
8
9 func main() {
10
11     match, _ := regexp.MatchString("p([a-z]+)ch", "peach")
12     fmt.Println(match)
13
14     r, _ := regexp.Compile("p([a-z]+)ch")
15
16     fmt.Println(r.MatchString("peach"))
17
18     fmt.Println(r.FindString("peach punch"))
19
20     fmt.Println(r.FindStringIndex("peach punch"))
21
22     fmt.Println(r.FindStringSubmatch("peach punch"))
23
24     fmt.Println(r.FindStringSubmatchIndex("peach punch"))
25
26     fmt.Println(r.FindAllString("peach punch pinch", -1))
27
28     fmt.Println(r.FindAllStringSubmatchIndex(
29         "peach punch pinch", -1))
30
31     fmt.Println(r.FindAllString("peach punch pinch", 2))
32
33     fmt.Println(r.Match([]byte("peach")))
34 }
```

```
35     r = regexp.MustCompile("p([a-z]+)ch")
36     fmt.Println(r)
37
38     fmt.Println(r.ReplaceAllString("a peach", "<fruit>"))
39
40     in := []byte("a peach")
41     out := r.ReplaceAllFunc(in, bytes.ToUpper)
42     fmt.Println(string(out))
43 }
```

This tests whether a pattern matches a string.

Above we used a string pattern directly, but for other regexp tasks you'll need to Compile an optimized Regexp struct.

Many methods are available on these structs. Here's a match test like we saw earlier.

This finds the match for the regexp.

This also finds the first match but returns the start and end indexes for the match instead of the matching text.

The Submatch variants include information about both the whole-pattern matches and the submatches within those matches. For example this will return information for both `p([a-z]+)ch` and `([a-z]+)`.

Similarly this will return information about the indexes of matches and submatches.

The All variants of these functions apply to all matches in the input, not just the first. For example to find all matches for a regexp.

These All variants are available for the other functions we saw above as well.

Providing a non-negative integer as the second argument to these functions will limit the number of matches.

Our examples above had string arguments and used names like `MatchString`. We can also provide `[]byte` arguments and drop `String` from the function name.

When creating global variables with regular expressions you can use the `MustCompile` variation of `Compile`. `MustCompile` panics instead of returning an error, which makes it safer to use for global variables.

The `regexp` package can also be used to replace subsets of strings with other values.

The `Func` variant allows you to transform matched text with a given function.

Output:

```
1 $ go run regular-expressions.go
2 true
3 true
4 peach
5 [0 5]
6 [peach ea]
7 [0 5 1 3]
8 [peach punch pinch]
9 [[0 5 1 3] [6 11 7 9] [12 17 13 15]]
10 [peach punch]
11 true
12 p([a-z]+)ch
13 a <fruit>
14 a PEACH
```

JSON

Go offers built-in support for JSON encoding and decoding, including to and from built-in and custom data types.

```
1 package main
2
3 import (
4     "encoding/json"
5     "fmt"
6     "os"
7 )
8
9 type response1 struct {
10     Page    int
11     Fruits []string
12 }
13
14 type response2 struct {
15     Page    int    `json:"page"`
16     Fruits []string `json:"fruits"`
17 }
18
19 func main() {
20
21     bolB, _ := json.Marshal(true)
22     fmt.Println(string(bolB))
23
24     intB, _ := json.Marshal(1)
25     fmt.Println(string(intB))
26
27     fltB, _ := json.Marshal(2.34)
28     fmt.Println(string(fltB))
29
30     strB, _ := json.Marshal("gopher")
31     fmt.Println(string(strB))
32
33     slcD := []string{"apple", "peach", "pear"}
34     slcB, _ := json.Marshal(slcD)
```

```

35     fmt.Println(string(slcB))
36
37     mapD := map[string]int{"apple": 5, "lettuce": 7}
38     mapB, _ := json.Marshal(mapD)
39     fmt.Println(string(mapB))
40
41     res1D := &response1{
42         Page: 1,
43         Fruits: []string{"apple", "peach", "pear"}}
44     res1B, _ := json.Marshal(res1D)
45     fmt.Println(string(res1B))
46
47     res2D := &response2{
48         Page: 1,
49         Fruits: []string{"apple", "peach", "pear"}}
50     res2B, _ := json.Marshal(res2D)
51     fmt.Println(string(res2B))
52
53     byt := []byte(`{"num":6.13,"strs":["a","b"]}`)
54
55     var dat map[string]interface{}
56
57     if err := json.Unmarshal(byt, &dat); err != nil {
58         panic(err)
59     }
60     fmt.Println(dat)
61
62     num := dat["num"].(float64)
63     fmt.Println(num)
64
65     strs := dat["strs"].([]interface{})
66     str1 := strs[0].(string)
67     fmt.Println(str1)
68
69     str := `{"page": 1, "fruits": ["apple", "peach"]}`
70     res := response2{}
71     json.Unmarshal([]byte(str), &res)
72     fmt.Println(res)
73     fmt.Println(res.Fruits[0])
74
75     enc := json.NewEncoder(os.Stdout)
76     d := map[string]int{"apple": 5, "lettuce": 7}
77     enc.Encode(d)

```

78 }

We'll use these two structs to demonstrate encoding and decoding of custom types below.

Only exported fields will be encoded/decoded in JSON. Fields must start with capital letters to be exported.

First we'll look at encoding basic data types to JSON strings. Here are some examples for atomic values.

And here are some for slices and maps, which encode to JSON arrays and objects as you'd expect.

The JSON package can automatically encode your custom data types. It will only include exported fields in the encoded output and will by default use those names as the JSON keys.

You can use tags on struct field declarations to customize the encoded JSON key names. Check the definition of `response2` above to see an example of such tags.

Now let's look at decoding JSON data into Go values. Here's an example for a generic data structure.

We need to provide a variable where the JSON package can put the decoded data. This `map[string]interface{}` will hold a map of strings to arbitrary data types.

Here's the actual decoding, and a check for associated errors.

In order to use the values in the decoded map, we'll need to convert them to their appropriate type. For example here we convert the value in `num` to the expected `float64` type.

Accessing nested data requires a series of conversions.

We can also decode JSON into custom data types. This has the advantages of adding additional type-safety to our programs and eliminating the need for type assertions when accessing the decoded data.

In the examples above we always used bytes and strings as intermediates between the data and JSON representation on standard out. We can also stream JSON encodings directly to `os.Writers` like `os.Stdout` or even HTTP response bodies.

Output:

```
1  $ go run json.go
2  true
3  1
4  2.34
5  "gopher"
6  ["apple", "peach", "pear"]
7  {"apple":5, "lettuce":7}
8  {"Page":1, "Fruits":["apple", "peach", "pear"]}
9  {"page":1, "fruits":["apple", "peach", "pear"]}
10 map[num:6.13 strs:[a b]]
```

```
11 6.13
12 a
13 {1 [apple peach]}
14 apple
15 {"apple":5,"lettuce":7}
```

XML

Go offers built-in support for XML and XML-like formats with the `encoding.xml` package.

```
1 package main
2
3 import (
4     "encoding/xml"
5     "fmt"
6 )
7
8 type Plant struct {
9     XMLName xml.Name `xml:"plant"`
10    Id       int    `xml:"id,attr"`
11    Name     string `xml:"name"`
12    Origin   []string `xml:"origin"`
13 }
14
15 func (p Plant) String() string {
16     return fmt.Sprintf("Plant id=%v, name=%v, origin=%v",
17         p.Id, p.Name, p.Origin)
18 }
19
20 func main() {
21     coffee := &Plant{Id: 27, Name: "Coffee"}
22     coffee.Origin = []string{"Ethiopia", "Brazil"}
23
24     out, _ := xml.MarshalIndent(coffee, " ", " ")
25     fmt.Println(string(out))
26
27     fmt.Println(xml.Header + string(out))
28
29     var p Plant
30     if err := xml.Unmarshal(out, &p); err != nil {
31         panic(err)
32     }
33     fmt.Println(p)
34
35     tomato := &Plant{Id: 81, Name: "Tomato"}
```

```

36     tomato.Origin = []string{"Mexico", "California"}
37
38     type Nesting struct {
39         XMLName xml.Name `xml:"nesting"`
40         Plants   []*Plant `xml:"parent>child>plant"`
41     }
42
43     nesting := &Nesting{}
44     nesting.Plants = []*Plant{coffee, tomato}
45
46     out, _ = xml.MarshalIndent(nesting, " ", " ")
47     fmt.Println(string(out))
48 }

```

Plant will be mapped to XML. Similarly to the JSON examples, field tags contain directives for the encoder and decoder. Here we use some special features of the XML package: the XMLName field name dictates the name of the XML element representing this struct; id,attr means that the Id field is an XML attribute rather than a nested element.

Emit XML representing our plant; using MarshalIndent to produce a more human-readable output.

To add a generic XML header to the output, append it explicitly.

Use Unmarshal to parse a stream of bytes with XML into a data structure. If the XML is malformed or cannot be mapped onto Plant, a descriptive error will be returned.

The parent>child>plant field tag tells the encoder to nest all plants under <parent><child>...

Output:

```

1  $ go run xml.go
2  <plant id="27">
3      <name>Coffee</name>
4      <origin>Ethiopia</origin>
5      <origin>Brazil</origin>
6  </plant>
7  <?xml version="1.0" encoding="UTF-8"?>
8  <plant id="27">
9      <name>Coffee</name>
10     <origin>Ethiopia</origin>
11     <origin>Brazil</origin>
12 </plant>
13 Plant id=27, name=Coffee, origin=[Ethiopia Brazil]
14 <nesting>
15     <parent>
16         <child>

```

```
17      <plant id="27">
18          <name>Coffee</name>
19          <origin>Ethiopia</origin>
20          <origin>Brazil</origin>
21      </plant>
22      <plant id="81">
23          <name>Tomato</name>
24          <origin>Mexico</origin>
25          <origin>California</origin>
26      </plant>
27  </child>
28 </parent>
29 </nesting>
```

Time

Go offers extensive support for times and durations; here are some examples.

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func main() {
9     p := fmt.Println
10
11     now := time.Now()
12     p(now)
13
14     then := time.Date(
15         2009, 11, 17, 20, 34, 58, 651387237, time.UTC)
16     p(then)
17
18     p(then.Year())
19     p(then.Month())
20     p(then.Day())
21     p(then.Hour())
22     p(then.Minute())
23     p(then.Second())
24     p(then.Nanosecond())
25     p(then.Location())
26
27     p(then.Weekday())
28
29     p(then.Before(now))
30     p(then.After(now))
31     p(then.Equal(now))
32
33     diff := now.Sub(then)
34     p(diff)
35 }
```

```
36     p(diff.Hours())
37     p(diff.Minutes())
38     p(diff.Seconds())
39     p(diff.Nanoseconds())
40
41     p(then.Add(diff))
42     p(then.Add(-diff))
43 }
```

We'll start by getting the current time.

You can build a time struct by providing the year, month, day, etc. Times are always associated with a Location, i.e. time zone.

You can extract the various components of the time value as expected.

The Monday-Sunday Weekday is also available.

These methods compare two times, testing if the first occurs before, after, or at the same time as the second, respectively.

The Sub methods returns a Duration representing the interval between two times.

We can compute the length of the duration in various units.

You can use Add to advance a time by a given duration, or with a - to move backwards by a duration.

Output:

```
1  $ go run time.go
2  2012-10-31 15:50:13.793654 +0000 UTC
3  2009-11-17 20:34:58.651387237 +0000 UTC
4  2009
5  November
6  17
7  20
8  34
9  58
10 651387237
11 UTC
12 Tuesday
13 true
14 false
15 false
16 25891h15m15.142266763s
17 25891.25420618521
18 1.5534752523711128e+06
```

```
19 9.320851514226677e+07
20 93208515142266763
21 2012-10-31 15:50:13.793654 +0000 UTC
22 2006-12-05 01:19:43.509120474 +0000 UTC
```

Epoch

A common requirement in programs is getting the number of seconds, milliseconds, or nanoseconds since the Unix epoch. Here's how to do it in Go.

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func main() {
9
10     now := time.Now()
11     secs := now.Unix()
12     nanos := now.UnixNano()
13     fmt.Println(now)
14
15     millis := nanos / 1000000
16     fmt.Println(secs)
17     fmt.Println(millis)
18     fmt.Println(nanos)
19
20     fmt.Println(time.Unix(secs, 0))
21     fmt.Println(time.Unix(0, nanos))
22 }
```

Use `time.Now` with `Unix` or `UnixNano` to get elapsed time since the Unix epoch in seconds or nanoseconds, respectively.

Note that there is no `UnixMillis`, so to get the milliseconds since epoch you'll need to manually divide from nanoseconds.

You can also convert integer seconds or nanoseconds since the epoch into the corresponding time.

Output:

```
1  $ go run epoch.go
2  2012-10-31 16:13:58.292387 +0000 UTC
3  1351700038
4  1351700038292
5  1351700038292387000
6  2012-10-31 16:13:58 +0000 UTC
7  2012-10-31 16:13:58.292387 +0000 UTC
```

Time Formatting / Parsing

Go supports time formatting and parsing via pattern-based layouts.

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func main() {
9     p := fmt.Println
10
11     t := time.Now()
12     p(t.Format(time.RFC3339))
13
14     t1, e := time.Parse(
15         time.RFC3339,
16         "2012-11-01T22:08:41+00:00")
17     p(t1)
18
19     p(t.Format("3:04PM"))
20     p(t.Format("Mon Jan _2 15:04:05 2006"))
21     p(t.Format("2006-01-02T15:04:05.999999-07:00"))
22     form := "3 04 PM"
23     t2, e := time.Parse(form, "8 41 PM")
24     p(t2)
25
26     fmt.Printf("%d-%02d-%02dT%02d:%02d:%02d-00:00\n",
27         t.Year(), t.Month(), t.Day(),
28         t.Hour(), t.Minute(), t.Second())
29
30     ansic := "Mon Jan _2 15:04:05 2006"
31     _, e = time.Parse(ansic, "8:41PM")
32     p(e)
33 }
```

Here's a basic example of formatting a time according to RFC3339, using the corresponding layout constant.

Time parsing uses the same layout values as Format.

Format and Parse use example-based layouts. Usually you'll use a constant from time for these layouts, but you can also supply custom layouts. Layouts must use the reference time Mon Jan 2 15:04:05 MST 2006 to show the pattern with which to format/parse a given time/string. The example time must be exactly as shown: the year 2006, 15 for the hour, Monday for the day of the week, etc.

For purely numeric representations you can also use standard string formatting with the extracted components of the time value.

Parse will return an error on malformed input explaining the parsing problem.

Output:

```
1  $ go run time-formatting-parsing.go
2  2014-04-15T18:00:15-07:00
3  2012-11-01 22:08:41 +0000 +0000
4  6:00PM
5  Tue Apr 15 18:00:15 2014
6  2014-04-15T18:00:15.161182-07:00
7  0000-01-01 20:41:00 +0000 UTC
8  2014-04-15T18:00:15-00:00
9  parsing time "8:41PM" as "Mon Jan _2 15:04:05 2006": ...
```

Random Numbers

Go's math/rand package provides pseudorandom number generation.

```
1  package main
2
3  import (
4      "fmt"
5      "math/rand"
6      "time"
7  )
8
9  func main() {
10
11      fmt.Print(rand.Intn(100), ", ")
12      fmt.Print(rand.Intn(100))
13      fmt.Println()
14
15      fmt.Println(rand.Float64())
16
17      fmt.Print((rand.Float64()*5)+5, ", ")
18      fmt.Print((rand.Float64() * 5) + 5)
19      fmt.Println()
20
21      s1 := rand.NewSource(time.Now().UnixNano())
22      r1 := rand.New(s1)
23
24      fmt.Print(r1.Intn(100), ", ")
25      fmt.Print(r1.Intn(100))
26      fmt.Println()
27
28      s2 := rand.NewSource(42)
29      r2 := rand.New(s2)
30      fmt.Print(r2.Intn(100), ", ")
31      fmt.Print(r2.Intn(100))
32      fmt.Println()
33      s3 := rand.NewSource(42)
34      r3 := rand.New(s3)
35      fmt.Print(r3.Intn(100), ", ")
```

```
36     fmt.Print(r3.Intn(100))
37 }
```

For example, `rand.Intn` returns a random int `n`, $0 \leq n < 100$.

`rand.Float64` returns a float64 `f`, $0.0 \leq f < 1.0$.

This can be used to generate random floats in other ranges, for example $5.0 \leq f' < 10.0$.

The default number generator is deterministic, so it'll produce the same sequence of numbers each time by default. To produce varying sequences, give it a seed that changes. Note that this is not safe to use for random numbers you intend to be secret, use `crypto/rand` for those.

Call the resulting `rand.Rand` just like the functions on the `rand` package.

If you seed a source with the same number, it produces the same sequence of random numbers.

Output:

```
1  $ go run random-numbers.go
2  81,87
3  0.6645600532184904
4  7.123187485356329,8.434115364335547
5  0,28
6  5,87
7  5,87
```

Number Parsing

Parsing numbers from strings is a basic but common task in many programs; here's how to do it in Go.

```
1 package main
2
3 import (
4     "fmt"
5     "strconv"
6 )
7
8 func main() {
9
10     f, _ := strconv.ParseFloat("1.234", 64)
11     fmt.Println(f)
12
13     i, _ := strconv.ParseInt("123", 0, 64)
14     fmt.Println(i)
15
16     d, _ := strconv.ParseInt("0x1c8", 0, 64)
17     fmt.Println(d)
18
19     u, _ := strconv.ParseUint("789", 0, 64)
20     fmt.Println(u)
21
22     k, _ := strconv.Atoi("135")
23     fmt.Println(k)
24
25     _, e := strconv.Atoi("wat")
26     fmt.Println(e)
27 }
```

The built-in package `strconv` provides the number parsing.

With `ParseFloat`, this 64 tells how many bits of precision to parse.

For `ParseInt`, the 0 means infer the base from the string. 64 requires that the result fit in 64 bits.

`ParseInt` will recognize hex-formatted numbers.

A `ParseUint` is also available.

`Atoi` is a convenience function for basic base-10 int parsing.

Parse functions return an error on bad input.

Output:

```
1  $ go run number-parsing.go
2  1.234
3  123
4  456
5  789
6  135
7  strconv.ParseInt: parsing "wat": invalid syntax
```

URL Parsing

URLs provide a uniform way to locate resources. Here's how to parse URLs in Go.

```
1 package main
2
3 import (
4     "fmt"
5     "net"
6     "net/url"
7 )
8
9 func main() {
10
11     s := "postgres://user:pass@host.com:5432/path?k=v#f"
12
13     u, err := url.Parse(s)
14     if err != nil {
15         panic(err)
16     }
17
18     fmt.Println(u.Scheme)
19
20     fmt.Println(u.User)
21     fmt.Println(u.User.Username())
22     p, _ := u.User.Password()
23     fmt.Println(p)
24
25     fmt.Println(u.Host)
26     host, port, _ := net.SplitHostPort(u.Host)
27     fmt.Println(host)
28     fmt.Println(port)
29
30     fmt.Println(u.Path)
31     fmt.Println(u.Fragment)
32
33     fmt.Println(u.RawQuery)
34     m, _ := url.ParseQuery(u.RawQuery)
35     fmt.Println(m)
```

```
36     fmt.Println(m["k"][0])
37 }
```

We'll parse this example URL, which includes a scheme, authentication info, host, port, path, query params, and query fragment.

Parse the URL and ensure there are no errors.

Accessing the scheme is straightforward.

User contains all authentication info; call Username and Password on this for individual values.

The Host contains both the hostname and the port, if present. Use SplitHostPort to extract them.

Here we extract the path and the fragment after the #.

To get query params in a string of k=v format, use RawQuery. You can also parse query params into a map. The parsed query param maps are from strings to slices of strings, so index into [0] if you only want the first value.

Running our URL parsing program shows all the different pieces that we extracted.

Output:

```
1  $ go run url-parsing.go
2  postgres
3  user:pass
4  user
5  pass
6  host.com:5432
7  host.com
8  5432
9  /path
10 f
11 k=v
12 map[k: [v]]
13 v
```

SHA1 Hashes

SHA1 hashes are frequently used to compute short identities for binary or text blobs. For example, the git revision control system uses SHA1s extensively to identify versioned files and directories. Here's how to compute SHA1 hashes in Go.

```
1 package main
2
3 import (
4     "crypto/sha1"
5     "fmt"
6 )
7
8 func main() {
9     s := "sha1 this string"
10
11     h := sha1.New()
12
13     h.Write([]byte(s))
14
15     bs := h.Sum(nil)
16
17     fmt.Println(s)
18     fmt.Printf("%x\n", bs)
19 }
```

Go implements several hash functions in various `crypto/*` packages.

The pattern for generating a hash is `sha1.New()`, `sha1.Write(bytes)`, then `sha1.Sum([]byte{})`. Here we start with a new hash.

`Write` expects bytes. If you have a string `s`, use `[]byte(s)` to coerce it to bytes.

This gets the finalized hash result as a byte slice. The argument to `Sum` can be used to append to an existing byte slice: it usually isn't needed.

SHA1 values are often printed in hex, for example in git commits. Use the `%x` format verb to convert a hash results to a hex string.

Running the program computes the hash and prints it in a human-readable hex format.

You can compute other hashes using a similar pattern to the one shown above. For example, to compute MD5 hashes import `crypto/md5` and use `md5.New()`.

Note that if you need cryptographically secure hashes, you should carefully research hash strength!

Output:

```
1 $ go run sha1-hashes.go
2 sha1 this string
3 cf23df2207d99a74fbe169e3eba035e633b65d94
```

Base64 Encoding

Go provides built-in support for base64 encoding/decoding.

```
1 package main
2
3 import (
4     b64 "encoding/base64"
5     "fmt"
6 )
7
8 func main() {
9
10     data := "abc123!?$*&'-=@~"
11
12     sEnc := b64.StdEncoding.EncodeToString([]byte(data))
13     fmt.Println(sEnc)
14
15     sDec, _ := b64.StdEncoding.DecodeString(sEnc)
16     fmt.Println(string(sDec))
17     fmt.Println()
18
19     uEnc := b64.URLEncoding.EncodeToString([]byte(data))
20     fmt.Println(uEnc)
21     uDec, _ := b64.URLEncoding.DecodeString(uEnc)
22     fmt.Println(string(uDec))
23 }
```

This syntax imports the encoding/base64 package with the b64 name instead of the default base64. It'll save us some space below.

Here's the string we'll encode/decode.

Go supports both standard and URL-compatible base64. Here's how to encode using the standard encoder. The encoder requires a []byte so we convert our string to that type.

Decoding may return an error, which you can check if you don't already know the input to be well-formed.

This encodes/decodes using a URL-compatible base64 format.

The string encodes to slightly different values with the standard and URL base64 encoders (trailing + vs -) but they both decode to the original string as desired.

Output:

```
1 $ go run base64-encoding.go
2 YWJjMTIzIT8kKiYoKSctPUB+
3 abc123!?$*&(')-=@~
4 YWJjMTIzIT8kKiYoKSctPUB-
5 abc123!?$*&(')-=@~
```

Reading Files

Reading and writing files are basic tasks needed for many Go programs. First we'll look at some examples of reading files.

```
1  package main
2
3  import (
4      "bufio"
5      "fmt"
6      "io"
7      "io/ioutil"
8      "os"
9  )
10
11 func check(e error) {
12     if e != nil {
13         panic(e)
14     }
15 }
16
17 func main() {
18
19     dat, err := ioutil.ReadFile("/tmp/dat")
20     check(err)
21     fmt.Print(string(dat))
22
23     f, err := os.Open("/tmp/dat")
24     check(err)
25
26     b1 := make([]byte, 5)
27     n1, err := f.Read(b1)
28     check(err)
29     fmt.Printf("%d bytes: %s\n", n1, string(b1[:n1]))
30
31     o2, err := f.Seek(6, 0)
32     check(err)
33     b2 := make([]byte, 2)
34     n2, err := f.Read(b2)
```

```
35     check(err)
36     fmt.Printf("%d bytes @ %d: ", n2, o2)
37     fmt.Printf("%v\n", string(b2[:n2]))
38
39     o3, err := f.Seek(6, 0)
40     check(err)
41     b3 := make([]byte, 2)
42     n3, err := io.ReadAtLeast(f, b3, 2)
43     check(err)
44     fmt.Printf("%d bytes @ %d: %s\n", n3, o3, string(b3))
45
46     _, err = f.Seek(0, 0)
47     check(err)
48
49     r4 := bufio.NewReader(f)
50     b4, err := r4.Peek(5)
51     check(err)
52     fmt.Printf("5 bytes: %s\n", string(b4))
53
54     f.Close()
55 }
```

Reading files requires checking most calls for errors. This helper will streamline our error checks below.

Perhaps the most basic file reading task is slurping a file's entire contents into memory.

You'll often want more control over how and what parts of a file are read. For these tasks, start by Opening a file to obtain an `os.File` value.

Read some bytes from the beginning of the file. Allow up to 5 to be read but also note how many actually were read.

You can also Seek to a known location in the file and Read from there.

The `io` package provides some functions that may be helpful for file reading. For example, reads like the ones above can be more robustly implemented with `ReadAtLeast`.

There is no built-in rewind, but `Seek(0, 0)` accomplishes this.

The `bufio` package implements a buffered reader that may be useful both for its efficiency with many small reads and because of the additional reading methods it provides.

Close the file when you're done (usually this would be scheduled immediately after Opening with `defer`).

Output:

```
1 $ echo "hello" > /tmp/dat
2 $ echo "go" >> /tmp/dat
3 $ go run reading-files.go
4 hello
5 go
6 5 bytes: hello
7 2 bytes @ 6: go
8 2 bytes @ 6: go
9 5 bytes: hello
```

Writing Files

Writing files in Go follows similar patterns to the ones we saw earlier for reading.

```
1 package main
2
3 import (
4     "bufio"
5     "fmt"
6     "io/ioutil"
7     "os"
8 )
9
10 func check(e error) {
11     if e != nil {
12         panic(e)
13     }
14 }
15
16 func main() {
17
18     d1 := []byte("hello\ngo\n")
19     err := ioutil.WriteFile("/tmp/dat1", d1, 0644)
20     check(err)
21
22     f, err := os.Create("/tmp/dat2")
23     check(err)
24
25     defer f.Close()
26
27     d2 := []byte{115, 111, 109, 101, 10}
28     n2, err := f.Write(d2)
29     check(err)
30     fmt.Printf("wrote %d bytes\n", n2)
31
32     n3, err := f.WriteString("writes\n")
33     check(err)
34     fmt.Printf("wrote %d bytes\n", n3)
35 }
```

```
36     f.Sync()
37
38     w := bufio.NewWriter(f)
39     n4, err := w.WriteString("buffered\n")
40     check(err)
41     fmt.Printf("wrote %d bytes\n", n4)
42
43     w.Flush()
44
45 }
```

To start, here's how to dump a string (or just bytes) into a file.

For more granular writes, open a file for writing.

It's idiomatic to defer a Close immediately after opening a file.

You can Write byte slices as you'd expect.

A WriteString is also available.

Issue a Sync to flush writes to stable storage.

bufio provides buffered writers in addition to the buffered readers we saw earlier.

Use Flush to ensure all buffered operations have been applied to the underlying writer.

Try running the file-writing code.

Then check the contents of the written files.

Output:

```
1  $ go run writing-files.go
2  wrote 5 bytes
3  wrote 7 bytes
4  wrote 9 bytes
5
6  $ cat /tmp/dat1
7  hello
8  go
9  $ cat /tmp/dat2
10 some
11 writes
12 buffered
```

Line Filters

A line filter is a common type of program that reads input on stdin, processes it, and then prints some derived result to stdout. `grep` and `sed` are common line filters.

Here's an example line filter in Go that writes a capitalized version of all input text. You can use this pattern to write your own Go line filters.

```
1  package main
2
3  import (
4      "bufio"
5      "fmt"
6      "os"
7      "strings"
8  )
9
10 func main() {
11
12     scanner := bufio.NewScanner(os.Stdin)
13
14     for scanner.Scan() {
15
16         ucl := strings.ToUpper(scanner.Text())
17
18         fmt.Println(ucl)
19     }
20
21     if err := scanner.Err(); err != nil {
22         fmt.Fprintln(os.Stderr, "error:", err)
23         os.Exit(1)
24     }
25 }
```

Wrapping the unbuffered `os.Stdin` with a buffered scanner gives us a convenient `Scan` method that advances the scanner to the next token; which is the next line in the default scanner.

`Text` returns the current token, here the next line, from the input.

Write out the uppercased line.

Check for errors during Scan. End of file is expected and not reported by Scan as an error.

To try out our line filter, first make a file with a few lowercase lines.

Then use the line filter to get uppercase lines.

Output:

```
1 $ echo 'hello' > /tmp/lines
2 $ echo 'filter' >> /tmp/lines
3
4 $ cat /tmp/lines | go run line-filters.go
5 HELLO
6 FILTER
```

File Paths

The filepath package provides functions to parse and construct file paths in a way that is portable between operating systems; dir/file on Linux vs. dir\file on Windows, for example.

```
1  package main
2
3  import (
4      "fmt"
5      "path/filepath"
6      "strings"
7  )
8
9  func main() {
10
11      p := filepath.Join("dir1", "dir2", "filename")
12      fmt.Println("p:", p)
13
14      fmt.Println(filepath.Join("dir1//", "filename"))
15      fmt.Println(filepath.Join("dir1/../dir1", "filename"))
16
17      fmt.Println("Dir(p):", filepath.Dir(p))
18      fmt.Println("Base(p):", filepath.Base(p))
19
20      fmt.Println(filepath.IsAbs("dir/file"))
21      fmt.Println(filepath.IsAbs("/dir/file"))
22
23      filename := "config.json"
24
25      ext := filepath.Ext(filename)
26      fmt.Println(ext)
27
28      fmt.Println(strings.TrimSuffix(filename, ext))
29
30      rel, err := filepath.Rel("a/b", "a/b/t/file")
31      if err != nil {
32          panic(err)
33      }
34      fmt.Println(rel)
```

```
35
36     rel, err = filepath.Rel("a/b", "a/c/t/file")
37     if err != nil {
38         panic(err)
39     }
40     fmt.Println(rel)
41 }
```

Join should be used to construct paths in a portable way. It takes any number of arguments and constructs a hierarchical path from them.

You should always use Join instead of concatenating /s or \s manually. In addition to providing portability, Join will also normalize paths by removing superfluous separators and directory changes.

Dir and Base can be used to split a path to the directory and the file. Alternatively, Split will return both in the same call.

We can check whether a path is absolute.

Some file names have extensions following a dot. We can split the extension out of such names with Ext.

To find the file's name with the extension removed, use strings.TrimSuffix.

Rel finds a relative path between a base and a target. It returns an error if the target cannot be made relative to base.

Output:

```
1  $ go run file-paths.go
2  p: dir1/dir2/filename
3  dir1/filename
4  dir1/filename
5  Dir(p): dir1/dir2
6  Base(p): filename
7  false
8  true
9  .json
10 config
11 t/file
12 ../c/t/file
```

Directories

Go has several useful functions for working with directories in the file system.

```
1  package main
2
3  import (
4      "fmt"
5      "io/ioutil"
6      "os"
7      "path/filepath"
8  )
9
10 func check(e error) {
11     if e != nil {
12         panic(e)
13     }
14 }
15
16 func main() {
17
18     err := os.Mkdir("subdir", 0755)
19     check(err)
20
21     defer os.RemoveAll("subdir")
22
23     createEmptyFile := func(name string) {
24         d := []byte("")
25         check(ioutil.WriteFile(name, d, 0644))
26     }
27
28     createEmptyFile("subdir/file1")
29
30     err = os.MkdirAll("subdir/parent/child", 0755)
31     check(err)
32
33     createEmptyFile("subdir/parent/file2")
34     createEmptyFile("subdir/parent/file3")
35     createEmptyFile("subdir/parent/child/file4")
36 }
```

```

36
37     c, err := ioutil.ReadDir("subdir/parent")
38     check(err)
39
40     fmt.Println("Listing subdir/parent")
41     for _, entry := range c {
42         fmt.Println(" ", entry.Name(), entry.IsDir())
43     }
44
45     err = os.Chdir("subdir/parent/child")
46     check(err)
47
48     c, err = ioutil.ReadDir(".")
49     check(err)
50
51     fmt.Println("Listing subdir/parent/child")
52     for _, entry := range c {
53         fmt.Println(" ", entry.Name(), entry.IsDir())
54     }
55
56     err = os.Chdir("../..")
57     check(err)
58
59     fmt.Println("Visiting subdir")
60     err = filepath.Walk("subdir", visit)
61 }
62
63 func visit(p string, info os.FileInfo, err error) error {
64     if err != nil {
65         return err
66     }
67     fmt.Println(" ", p, info.IsDir())
68     return nil
69 }

```

Create a new sub-directory in the current working directory.

When creating temporary directories, it's good practice to defer their removal. `os.RemoveAll` will delete a whole directory tree (similarly to `rm -rf`).

Helper function to create a new empty file.

We can create a hierarchy of directories, including parents with `MkdirAll`. This is similar to the command-line `mkdir -p`.

ReadDir lists directory contents, returning a slice of `os.FileInfo` objects.

Chdir lets us change the current working directory, similarly to `cd`.

Now we'll see the contents of `subdir/parent/child` when listing the current directory.

`cd` back to where we started.

We can also visit a directory recursively, including all its sub-directories. `Walk` accepts a callback function to handle every file or directory visited.

`visit` is called for every file or directory found recursively by `filepath.Walk`.

Output:

```
1  $ go run directories.go
2  Listing subdir/parent
3      child true
4      file2 false
5      file3 false
6  Listing subdir/parent/child
7      file4 false
8  Visiting subdir
9      subdir true
10     subdir/file1 false
11     subdir/parent true
12     subdir/parent/child true
13     subdir/parent/child/file4 false
14     subdir/parent/file2 false
15     subdir/parent/file3 false
```

Temporary Files and Directories

Throughout program execution, we often want to create data that isn't needed after the program exits. Temporary files and directories are useful for this purpose since they don't pollute the file system over time.

```
1  package main
2
3  import (
4      "fmt"
5      "io/ioutil"
6      "os"
7      "path/filepath"
8  )
9
10 func check(e error) {
11     if e != nil {
12         panic(e)
13     }
14 }
15
16 func main() {
17
18     f, err := ioutil.TempFile("", "sample")
19     check(err)
20
21     fmt.Println("Temp file name:", f.Name())
22
23     defer os.Remove(f.Name())
24
25     _, err = f.Write([]byte{1, 2, 3, 4})
26     check(err)
27
28     dname, err := ioutil.TempDir("", "sampledir")
29     check(err)
30     fmt.Println("Temp dir name:", dname)
31
32     defer os.RemoveAll(dname)
33 }
```

```
34     fname := filepath.Join(dname, "file1")
35     err = ioutil.WriteFile(fname, []byte{1, 2}, 0666)
36     check(err)
37 }
```

The easiest way to create a temporary file is by calling `ioutil.TempFile`. It creates a file and opens it for reading and writing. We provide `""` as the first argument, so `ioutil.TempFile` will create the file in the default location for our OS.

Display the name of the temporary file. On Unix-based OSes the directory will likely be `/tmp`. The file name starts with the prefix given as the second argument to `ioutil.TempFile` and the rest is chosen automatically to ensure that concurrent calls will always create different file names.

Clean up the file after we're done. The OS is likely to clean up temporary files by itself after some time, but it's good practice to do this explicitly.

We can write some data to the file.

If we intend to write many temporary files, we may prefer to create a temporary directory. `ioutil.TempDir`'s arguments are the same as `TempFile`'s, but it returns a directory name rather than an open file.

Now we can synthesize temporary file names by prefixing them with our temporary directory.

Output:

```
1 $ go run temporary-files-and-directories.go
2 Temp file name: /tmp/sample610887201
3 Temp dir name: /tmp/samplendir898854668
```

Testing

Unit testing is an important part of writing principled Go programs. The testing package provides the tools we need to write unit tests and the go test command runs tests.

For the sake of demonstration, this code is in package main, but it could be any package. Testing code typically lives in the same package as the code it tests.

```
1 package main
2
3 import (
4     "fmt"
5     "testing"
6 )
7
8 func IntMin(a, b int) int {
9     if a < b {
10         return a
11     }
12     return b
13 }
14
15 func TestIntMinBasic(t *testing.T) {
16     ans := IntMin(2, -2)
17     if ans != -2 {
18
19         t.Errorf("IntMin(2, -2) = %d; want -2", ans)
20     }
21 }
22
23 func TestIntMinTableDriven(t *testing.T) {
24     var tests = []struct {
25         a, b int
26         want int
27     }{
28         {0, 1, 0},
29         {1, 0, 0},
30         {2, -2, -2},
31         {0, -1, -1},
32         {-1, 0, -1},
```

```

33     }
34
35     for _, tt := range tests {
36
37         testname := fmt.Sprintf("%d,%d", tt.a, tt.b)
38         t.Run(testname, func(t *testing.T) {
39             ans := IntMin(tt.a, tt.b)
40             if ans != tt.want {
41                 t.Errorf("got %d, want %d", ans, tt.want)
42             }
43         })
44     }
45 }

```

We'll be testing this simple implementation of an integer minimum. Typically, the code we're testing would be in a source file named something like `intutils.go`, and the test file for it would then be named `intutils_test.go`.

A test is created by writing a function with a name beginning with `Test`.

`t.Error*` will report test failures but continue executing the test. `t.Fail*` will report test failures and stop the test immediately.

Writing tests can be repetitive, so it's idiomatic to use a table-driven style, where test inputs and expected outputs are listed in a table and a single loop walks over them and performs the test logic.

`t.Run` enables running "subtests", one for each table entry. These are shown separately when executing `go test -v`.

Run all tests in the current project in verbose mode.

Output:

```

1  $ go test -v
2  == RUN    TestIntMinBasic
3  --- PASS: TestIntMinBasic (0.00s)
4  === RUN   TestIntMinTableDriven
5  === RUN   TestIntMinTableDriven/0,1
6  === RUN   TestIntMinTableDriven/1,0
7  === RUN   TestIntMinTableDriven/2,-2
8  === RUN   TestIntMinTableDriven/0,-1
9  === RUN   TestIntMinTableDriven/-1,0
10 --- PASS: TestIntMinTableDriven (0.00s)
11 --- PASS: TestIntMinTableDriven/0,1 (0.00s)
12 --- PASS: TestIntMinTableDriven/1,0 (0.00s)
13 --- PASS: TestIntMinTableDriven/2,-2 (0.00s)

```

```
14      --- PASS: TestIntMinTableDriven/0,-1 (0.00s)
15      --- PASS: TestIntMinTableDriven/-1,0 (0.00s)
16 PASS
17 ok      examples/testing    0.023s
```

Command-Line Arguments

Command-line arguments are a common way to parameterize execution of programs. For example, go run hello.go uses run and hello.go arguments to the go program.

```
1 package main
2
3 import (
4     "fmt"
5     "os"
6 )
7
8 func main() {
9
10     argsWithProg := os.Args
11     argsWithoutProg := os.Args[1:]
12
13     arg := os.Args[3]
14
15     fmt.Println(argsWithProg)
16     fmt.Println(argsWithoutProg)
17     fmt.Println(arg)
18 }
```

os.Args provides access to raw command-line arguments. Note that the first value in this slice is the path to the program, and os.Args[1:] holds the arguments to the program.

You can get individual args with normal indexing.

To experiment with command-line arguments it's best to build a binary with go build first.

Next we'll look at more advanced command-line processing with flags.

Output:

```
1 $ go build command-line-arguments.go
2 $ ./command-line-arguments a b c d
3 [./command-line-arguments a b c d]
4 [a b c d]
5 c
```

Command-Line Flags

Command-line flags are a common way to specify options for command-line programs. For example, in `wc -l` the `-l` is a command-line flag.

```
1 package main
2
3 import (
4     "flag"
5     "fmt"
6 )
7
8 func main() {
9
10     wordPtr := flag.String("word", "foo", "a string")
11
12     numbPtr := flag.Int("numb", 42, "an int")
13     boolPtr := flag.Bool("fork", false, "a bool")
14
15     var svar string
16     flag.StringVar(&svar, "svar", "bar", "a string var")
17
18     flag.Parse()
19
20     fmt.Println("word:", *wordPtr)
21     fmt.Println("numb:", *numbPtr)
22     fmt.Println("fork:", *boolPtr)
23     fmt.Println("svar:", svar)
24     fmt.Println("tail:", flag.Args())
25 }
```

Go provides a `flag` package supporting basic command-line flag parsing. We'll use this package to implement our example command-line program.

Basic flag declarations are available for string, integer, and boolean options. Here we declare a string flag `word` with a default value “foo” and a short description. This `flag.String` function returns a string pointer (not a string value); we'll see how to use this pointer below.

This declares `numb` and `fork` flags, using a similar approach to the `word` flag.

It's also possible to declare an option that uses an existing var declared elsewhere in the program. Note that we need to pass in a pointer to the flag declaration function.

Once all flags are declared, call `flag.Parse()` to execute the command-line parsing.

Here we'll just dump out the parsed options and any trailing positional arguments. Note that we need to dereference the pointers with e.g. `*wordPtr` to get the actual option values.

To experiment with the command-line flags program it's best to first compile it and then run the resulting binary directly.

Try out the built program by first giving it values for all flags.

Note that if you omit flags they automatically take their default values.

Trailing positional arguments can be provided after any flags.

Note that the flag package requires all flags to appear before positional arguments (otherwise the flags will be interpreted as positional arguments).

Use `-h` or `-help` flags to get automatically generated help text for the command-line program.

If you provide a flag that wasn't specified to the flag package, the program will print an error message and show the help text again.

Output:

```
1  $ go build command-line-flags.go
2  $ ./command-line-flags -word=opt -numb=7 -fork -svar=flag
3  word: opt
4  numb: 7
5  fork: true
6  svar: flag
7  tail: []
8
9  $ ./command-line-flags -word=opt
10 word: opt
11 numb: 42
12 fork: false
13 svar: bar
14 tail: []
15
16 $ ./command-line-flags -word=opt a1 a2 a3
17 word: opt
18 ...
19 tail: [a1 a2 a3]
20
21 $ ./command-line-flags -word=opt a1 a2 a3 -numb=7
22 word: opt
```

```
23 numb: 42
24 fork: false
25 svar: bar
26 tail: [a1 a2 a3 -numb=7]
27
28 $ ./command-line-flags -h
29 Usage of ./command-line-flags:
30 -fork=false: a bool
31 -numb=42: an int
32 -svar="bar": a string var
33 -word="foo": a string
34
35 $ ./command-line-flags -wat
36 flag provided but not defined: -wat
37 Usage of ./command-line-flags:
38 ...
```

Command-Line Subcommands

Some command-line tools, like the go tool or git have many subcommands, each with its own set of flags. For example, go build and go get are two different subcommands of the go tool. The flag package lets us easily define simple subcommands that have their own flags.

```
1  package main
2
3  import (
4      "flag"
5      "fmt"
6      "os"
7  )
8
9  func main() {
10
11      fooCmd := flag.NewFlagSet("foo", flag.ExitOnError)
12      fooEnable := fooCmd.Bool("enable", false, "enable")
13      fooName := fooCmd.String("name", "", "name")
14
15      barCmd := flag.NewFlagSet("bar", flag.ExitOnError)
16      barLevel := barCmd.Int("level", 0, "level")
17
18      if len(os.Args) < 2 {
19          fmt.Println("expected 'foo' or 'bar' subcommands")
20          os.Exit(1)
21      }
22
23      switch os.Args[1] {
24
25      case "foo":
26          fooCmd.Parse(os.Args[2:])
27          fmt.Println("subcommand 'foo'")
28          fmt.Println("  enable:", *fooEnable)
29          fmt.Println("  name:", *fooName)
30          fmt.Println("  tail:", fooCmd.Args())
31      case "bar":
32          barCmd.Parse(os.Args[2:])
33          fmt.Println("subcommand 'bar'")
```

```

34         fmt.Println("  level:", *barLevel)
35         fmt.Println("  tail:", barCmd.Args())
36     default:
37         fmt.Println("expected 'foo' or 'bar' subcommands")
38         os.Exit(1)
39     }
40 }

```

We declare a subcommand using the `NewFlagSet` function, and proceed to define new flags specific for this subcommand.

For a different subcommand we can define different supported flags.

The subcommand is expected as the first argument to the program.

Check which subcommand is invoked.

For every subcommand, we parse its own flags and have access to trailing positional arguments.

First invoke the foo subcommand.

Now try bar.

But bar won't accept foo's flags.

Output:

```

1  $ go build command-line-subcommands.go
2
3  $ ./command-line-subcommands foo -enable -name=joe a1 a2
4  subcommand 'foo'
5  enable: true
6  name: joe
7  tail: [a1 a2]
8
9  $ ./command-line-subcommands bar -level 8 a1
10 subcommand 'bar'
11 level: 8
12 tail: [a1]
13
14
15 $ ./command-line-subcommands bar -enable a1
16 flag provided but not defined: -enable
17 Usage of bar:
18 -level int
19 level

```

Environment Variables

Environment variables are a universal mechanism for conveying configuration information to Unix programs. Let's look at how to set, get, and list environment variables.

```
1 package main
2
3 import (
4     "fmt"
5     "os"
6     "strings"
7 )
8
9 func main() {
10
11     os.Setenv("FOO", "1")
12     fmt.Println("FOO:", os.Getenv("FOO"))
13     fmt.Println("BAR:", os.Getenv("BAR"))
14
15     fmt.Println()
16     for _, e := range os.Environ() {
17         pair := strings.SplitN(e, "=", 2)
18         fmt.Println(pair[0])
19     }
20 }
```

To set a key/value pair, use `os.Setenv`. To get a value for a key, use `os.Getenv`. This will return an empty string if the key isn't present in the environment.

Use `os.Environ` to list all key/value pairs in the environment. This returns a slice of strings in the form `KEY=value`. You can `strings.SplitN` them to get the key and value. Here we print all the keys.

Running the program shows that we pick up the value for FOO that we set in the program, but that BAR is empty.

The list of keys in the environment will depend on your particular machine.

If we set BAR in the environment first, the running program picks that value up.

Output:

```
1  $ go run environment-variables.go
2  FOO: 1
3  BAR:
4
5  TERM_PROGRAM
6  PATH
7  SHELL
8  ...
9
10 $ BAR=2 go run environment-variables.go
11 FOO: 1
12 BAR: 2
13 ...
```

HTTP Clients

The Go standard library comes with excellent support for HTTP clients and servers in the `net/http` package. In this example we'll use it to issue simple HTTP requests.

```
1 package main
2
3 import (
4     "bufio"
5     "fmt"
6     "net/http"
7 )
8
9 func main() {
10
11     resp, err := http.Get("http://gobyexample.com")
12     if err != nil {
13         panic(err)
14     }
15     defer resp.Body.Close()
16
17     fmt.Println("Response status:", resp.Status)
18
19     scanner := bufio.NewScanner(resp.Body)
20     for i := 0; scanner.Scan() && i < 5; i++ {
21         fmt.Println(scanner.Text())
22     }
23
24     if err := scanner.Err(); err != nil {
25         panic(err)
26     }
27 }
```

Issue an HTTP GET request to a server. `http.Get` is a convenient shortcut around creating an `http.Client` object and calling its `Get` method; it uses the `http.DefaultClient` object which has useful default settings.

Print the HTTP response status.

Print the first 5 lines of the response body.

Output:

```
1 $ go run http-clients.go
2 Response status: 200 OK
3 <!DOCTYPE html>
4 <html>
5 <head>
6 <meta charset="utf-8">
7 <title>Go by Example</title>
```

HTTP Servers

Writing a basic HTTP server is easy using the net/http package.

```
1 package main
2
3 import (
4     "fmt"
5     "net/http"
6 )
7
8 func hello(w http.ResponseWriter, req *http.Request) {
9
10    fmt.Fprintf(w, "hello\n")
11 }
12
13 func headers(w http.ResponseWriter, req *http.Request) {
14
15    for name, headers := range req.Header {
16        for _, h := range headers {
17            fmt.Fprintf(w, "%v: %v\n", name, h)
18        }
19    }
20 }
21
22 func main() {
23
24    http.HandleFunc("/hello", hello)
25    http.HandleFunc("/headers", headers)
26
27    http.ListenAndServe(":8090", nil)
28 }
```

A fundamental concept in net/http servers is handlers. A handler is an object implementing the `http.Handler` interface. A common way to write a handler is by using the `http.HandlerFunc` adapter on functions with the appropriate signature.

Functions serving as handlers take a `http.ResponseWriter` and a `http.Request` as arguments. The response writer is used to fill in the HTTP response. Here our simple response is just “hello\n”.

This handler does something a little more sophisticated by reading all the HTTP request headers and echoing them into the response body.

We register our handlers on server routes using the `http.HandleFunc` convenience function. It sets up the default router in the `net/http` package and takes a function as an argument.

Finally, we call the `ListenAndServe` with the port and a handler. `nil` tells it to use the default router we've just set up.

Run the server in the background.

Access the `/hello` route.

Output:

```
1 $ go run http-servers.go &
2
3 $ curl localhost:8090/hello
4 hello
```

Context

In the previous example we looked at setting up a simple HTTP server. HTTP servers are useful for demonstrating the usage of context.Context for controlling cancellation. A Context carries deadlines, cancellation signals, and other request-scoped values across API boundaries and goroutines.

```
1 package main
2
3 import (
4     "fmt"
5     "net/http"
6     "time"
7 )
8
9 func hello(w http.ResponseWriter, req *http.Request) {
10
11     ctx := req.Context()
12     fmt.Println("server: hello handler started")
13     defer fmt.Println("server: hello handler ended")
14
15     select {
16     case <-time.After(10 * time.Second):
17         fmt.Fprintf(w, "hello\n")
18     case <-ctx.Done():
19
20         err := ctx.Err()
21         fmt.Println("server:", err)
22         internalError := http.StatusInternalServerError
23         http.Error(w, err.Error(), internalError)
24     }
25 }
26
27 func main() {
28
29     http.HandleFunc("/hello", hello)
30     http.ListenAndServe(":8090", nil)
31 }
```

A context.Context is created for each request by the net/http machinery, and is available with the Context() method.

Wait for a few seconds before sending a reply to the client. This could simulate some work the server is doing. While working, keep an eye on the context's `Done()` channel for a signal that we should cancel the work and return as soon as possible.

The context's `Err()` method returns an error that explains why the `Done()` channel was closed.

As before, we register our handler on the `"/hello"` route, and start serving.

Run the server in the background.

Simulate a client request to `/hello`, hitting `Ctrl+C` shortly after starting to signal cancellation.

Output:

```
1  $ go run context-in-http-servers.go &
2
3  $ curl localhost:8090/hello
4  server: hello handler started
5  ^C
6  server: context canceled
7  server: hello handler ended
```

Spawning Processes

Sometimes our Go programs need to spawn other, non-Go processes. For example, the syntax highlighting on this site is implemented by spawning a pygmentize process from a Go program. Let's look at a few examples of spawning processes from Go.

```
1  package main
2
3  import (
4      "fmt"
5      "io/ioutil"
6      "os/exec"
7  )
8
9  func main() {
10
11      dateCmd := exec.Command("date")
12
13      dateOut, err := dateCmd.Output()
14      if err != nil {
15          panic(err)
16      }
17      fmt.Println("> date")
18      fmt.Println(string(dateOut))
19
20      grepCmd := exec.Command("grep", "hello")
21
22      grepIn, _ := grepCmd.StdinPipe()
23      grepOut, _ := grepCmd.StdoutPipe()
24      grepCmd.Start()
25      grepIn.Write([]byte("hello grep\ngoodbye grep"))
26      grepIn.Close()
27      grepBytes, _ := ioutil.ReadAll(grepOut)
28      grepCmd.Wait()
29
30      fmt.Println("> grep hello")
31      fmt.Println(string(grepBytes))
32
33      lsCmd := exec.Command("bash", "-c", "ls -a -l -h")
```

```

34     lsOut, err := lsCmd.Output()
35     if err != nil {
36         panic(err)
37     }
38     fmt.Println("> ls -a -l -h")
39     fmt.Println(string(lsOut))
40 }

```

We'll start with a simple command that takes no arguments or input and just prints something to stdout. The `exec.Command` helper creates an object to represent this external process.

`.Output` is another helper that handles the common case of running a command, waiting for it to finish, and collecting its output. If there were no errors, `dateOut` will hold bytes with the date info.

Next we'll look at a slightly more involved case where we pipe data to the external process on its stdin and collect the results from its stdout.

Here we explicitly grab input/output pipes, start the process, write some input to it, read the resulting output, and finally wait for the process to exit.

We omitted error checks in the above example, but you could use the usual `if err != nil` pattern for all of them. We also only collect the `StdoutPipe` results, but you could collect the `StderrPipe` in exactly the same way.

Note that when spawning commands we need to provide an explicitly delineated command and argument array, vs. being able to just pass in one command-line string. If you want to spawn a full command with a string, you can use `bash`'s `-c` option:

The spawned programs return output that is the same as if we had run them directly from the command-line.

Output:

```

1  $ go run spawning-processes.go
2  > date
3  Wed Oct 10 09:53:11 PDT 2012
4
5  > grep hello
6  hello grep
7  > ls -a -l -h
8  drwxr-xr-x  4 mark 136B Oct  3 16:29 .
9  drwxr-xr-x 91 mark 3.0K Oct  3 12:50 ..
10 -rw-r--r--  1 mark 1.3K Oct  3 16:28 spawning-processes.go

```

Exec'ing Processes

In the previous example we looked at spawning external processes. We do this when we need an external process accessible to a running Go process. Sometimes we just want to completely replace the current Go process with another (perhaps non-Go) one. To do this we'll use Go's implementation of the classic exec function.

```
1  package main
2
3  import (
4      "os"
5      "os/exec"
6      "syscall"
7  )
8
9  func main() {
10
11      binary, lookErr := exec.LookPath("ls")
12      if lookErr != nil {
13          panic(lookErr)
14      }
15
16      args := []string{"ls", "-a", "-l", "-h"}
17
18      env := os.Environ()
19
20      execErr := syscall.Exec(binary, args, env)
21      if execErr != nil {
22          panic(execErr)
23      }
24  }
```

For our example we'll exec ls. Go requires an absolute path to the binary we want to execute, so we'll use exec.LookPath to find it (probably /bin/ls).

Exec requires arguments in slice form (as apposed to one big string). We'll give ls a few common arguments. Note that the first argument should be the program name.

Exec also needs a set of environment variables to use. Here we just provide our current environment.

Here's the actual syscall.Exec call. If this call is successful, the execution of our process will end here and be replaced by the /bin/ls -a -l -h process. If there is an error we'll get a return value.

When we run our program it is replaced by ls.

Note that Go does not offer a classic Unix fork function. Usually this isn't an issue though, since starting goroutines, spawning processes, and exec'ing processes covers most use cases for fork.

Output:

```
1  $ go run execing-processes.go
2  total 16
3  drwxr-xr-x  4 mark 136B Oct  3 16:29 .
4  drwxr-xr-x 91 mark 3.0K Oct  3 12:50 ..
5  -rw-r--r--  1 mark 1.3K Oct  3 16:28 execing-processes.go
```

Signals

Sometimes we'd like our Go programs to intelligently handle Unix signals. For example, we might want a server to gracefully shutdown when it receives a SIGTERM, or a command-line tool to stop processing input if it receives a SIGINT. Here's how to handle signals in Go with channels.

```
1  package main
2
3  import (
4      "fmt"
5      "os"
6      "os/signal"
7      "syscall"
8  )
9
10 func main() {
11
12     sigs := make(chan os.Signal, 1)
13     done := make(chan bool, 1)
14
15     signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM)
16
17     go func() {
18         sig := <-sigs
19         fmt.Println()
20         fmt.Println(sig)
21         done <- true
22     }()
23
24     fmt.Println("awaiting signal")
25     <-done
26     fmt.Println("exiting")
27 }
```

Go signal notification works by sending `os.Signal` values on a channel. We'll create a channel to receive these notifications (we'll also make one to notify us when the program can exit).

`signal.Notify` registers the given channel to receive notifications of the specified signals.

This goroutine executes a blocking receive for signals. When it gets one it'll print it out and then notify the program that it can finish.

The program will wait here until it gets the expected signal (as indicated by the goroutine above sending a value on done) and then exit.

When we run this program it will block waiting for a signal. By typing ctrl-C (which the terminal shows as ^C) we can send a SIGINT signal, causing the program to print interrupt and then exit.

Output:

```
1  $ go run signals.go
2  awaiting signal
3  ^C
4  interrupt
5  exiting
```

Exit

Use `os.Exit` to immediately exit with a given status.

```
1 package main
2
3 import (
4     "fmt"
5     "os"
6 )
7
8 func main() {
9
10     defer fmt.Println("!")
11
12     os.Exit(3)
13 }
```

defers will not be run when using `os.Exit`, so this `fmt.Println` will never be called.

Exit with status 3.

Note that unlike e.g. C, Go does not use an integer return value from `main` to indicate exit status. If you'd like to exit with a non-zero status you should use `os.Exit`.

If you run `exit.go` using `go run`, the exit will be picked up by `go` and printed.

By building and executing a binary you can see the status in the terminal.

Note that the `!` from our program never got printed.

Output:

```
1 $ go run exit.go
2 exit status 3
3
4 $ go build exit.go
5 $ ./exit
6 $ echo $?
7 3
```

Keep developing your GO skills

This book is just a starting point for you to get inspired to keep discovering all the awesome features of GO.

Hopefully you are now more familiar with GO and you can start experimenting with all the libraries and frameworks developed for it, you will soon start enjoying the benefits of GO like the open source nature, it's speed and the fact that it is cross platform.

I want to say thanks for taking your time and reading the whole book, I really appreciate your commitment towards your education and professional success, and for that I'd like to point you in the right direction with the following incredible resources which will help you achieve your goals.

- Online Courses with discounts site: [Online Courses](#)⁹
- Facebook community for developers: [The Programming Hub](#)¹⁰
- Quora Group: [The Programming Hub](#)¹¹

I hope this book was useful, please feel free to contact me with any questions. Also I would appreciate your honest review or any feedback to improve the next editions of the book.

Thanks a lot for following along !!

⁹<https://coursesim.com>

¹⁰<https://www.facebook.com/theproghub>

¹¹https://www.quora.com/q/vlyubtzhdhpacqzr?invite_code=Yj9D6Z7H2Bf4CONidZmD

About the Author

Alejandro is a programmer and writer with the goal of inspiring people to learn about several tech related topics.

He's also a Udemy Instructor here's a link to his profile: [Alejandro Garcia Udemy Instructor](https://www.udemy.com/user/alejandro-garcia-172)¹²

Other Author Profiles

Amazon: [Alejandro on Amazon](https://www.amazon.com/Alejandro-Garcia/e/B08CF2H6TB)¹³

Leanpub: [Alejandro on Leanpub](https://leanpub.com/u/alejandro-garcia)¹⁴

Copyright

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor the publisher shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

This book contains some affiliate links, this means, if you purchase anything from those links there won't be an extra charge for you. In addition you will be supporting the future updates of this publication

Chapters of the book are based on the following repositories:

Go by Example Mark McGranaghan [CC Attribution 3.0](https://github.com/mmcgrana/gobyexample#license) ¹⁵

This book is an adaptation and transformation from the above mentioned repositories, this book is not from the original authors. The original books can be fetched from the repositories.

¹²<https://www.udemy.com/user/alejandro-garcia-172>

¹³<https://www.amazon.com/Alejandro-Garcia/e/B08CF2H6TB>

¹⁴<https://leanpub.com/u/alejandro-garcia>

¹⁵<https://github.com/mmcgrana/gobyexample#license>