

Golang Fundamentals

"Go is expressive, concise, clean, and efficient"

Overview

Introductions

You:

- Who are you?
- What is your job title?
- What do you do here?
- What's your programming background?

Me:

- Who am I?

History

- Developed at Google (2009)
- Robert Griesemer, Rob Pike, Ken Thompson
- Born out of frustration and shared dislike of C++

4

Goals of a Programming Language

- efficient compilation
- efficient execution
- or ease of programming
- ...all three were not available in the same language!

Programmers were choosing ease over safety and efficiency by moving to dynamically typed languages (e.g., Python and JavaScript) rather than C++ or, to a lesser extent, Java

5

"Did the C++ committee really believe that was wrong with C++ was that it didn't have enough features?" –Rob Pike

"Clumsy type systems drive people to dynamically typed languages" -Robert Griesemer

"Old programs read like quiet conversations between a well-spoken research worker and a well-studied mechanical colleague, not as a debate with a compiler. Who'd have guessed sophistication brought such noise?" -Dick Gabriel

"The complexity of C++, and the resulting impact on productivity, is no longer justified. All the hoops that the C++ programmer had to jump through in order to use a C-compatible language make no sense anymore—they're just a waste of time and effort. Go makes much more sense for the

class of problems that C++ was originally intended to solve." -Bruce Eckel

Philosophy

A new programming language which addressed common criticisms of other languages while keeping the good parts:

- static typing
- scalability to large systems (think Java, C++)
- networking and multiprocessing
- not requiring IDEs, but supporting them well

Another goal is...speed!

- should take at most a few seconds to build a large executable
- and coding itself should be speedy

Philosophy (cont'd)

- Programming = too much bookkeeping, repetition, and clerical work
- The sophistication is worthwhile—no one wants to go back to the old languages—but can it be more quietly achieved?
- Reduce the amount of *typing* in both senses of the word
- Reduce clutter and complexity
- No forward declarations and no header files—everything is declared exactly once
- Initialization is expressive, automatic, and easy to use

11

Sweet Spots

- Anywhere you need speed and scalability
- Back-end (web) development
- Automation
- Command-line tools
- Concurrency
- Systems programming (operating systems, utility software, device drivers, compilers, and linkers)
- What projects/tools do you know of that are written in Go...?

12

Written in Golang

- Docker
- Kubernetes
- Terraform
- Netflix
- Dropbox
- SoundCloud
- Uber
- Drone

13

Principles

Clean and Simple–Go strives to keep things small and beautiful. You should be able to do a lot in only a few lines of code.

Concurrent–Easy to "fire off" functions to be run as lightweight threads.

Channels–Communication with these goroutines is done, either via shared state or via channels.

Fast–Compilation is fast and execution is fast. The aim is to be as fast as C. Compilation time is measured in seconds.

Safe–Explicit casting and strict rules when converting one type to another. Go has garbage collection. No free()!...the language takes care of it.

Standard format–A Go program can be formatted in (almost) any way the programmers want, but an official format exists. The rule is very simple: The output of the filter gofmt is the officially endorsed format.

Postfix types-The type specification is given after the variable name, thus var a int, instead of int a.

UTF-8-UTF-8 is everywhere, in strings and in the program code. Finally you can use $\Phi=\Phi+1$ in your source code.

Open Source-The Go license is completely open source.

Fun-Programming with Go should be fun!

Let's Get...Up and Running

Installation

1. Download from <https://golang.org/dl>
2. Installer should put go in your \$PATH (you may need to restart any open Terminal sessions for the change to take effect)
3. Create workspace dir \$HOME/go (or set \$GOPATH to use a different location)
4. Create the directory src/hello inside your workspace
5. Inside that directory create a file hello.go containing the following (OK to copy/paste)

```
package main

func main() {
    println("Hey, let's go!")
}
```

Run

- click the Run button above and on any slide which has that button

17

Compiling/Running

1. cd \$HOME/go/src/hello

2. go build

- this will create an executable in your directory

3. ./hello

4. go install

- this will install the binary in \$GOPATH/bin (optional)
- you can also compile and run in one step: go run hello.go

Compiling/Running/Packaging

Go code (source and compiled) can be found in your GOPATH

- if not set, defaults to ~/go

src contains source code

bin contains executables

pkg contains compiled versions of the available libraries so the compiler can link against them without recompiling them

go tool

- go run will compile and run your code
- go build will produce an executable
- go install will install it in \$GOPATH/bin

The Golang Playground

- receives a Go program
- compiles, links, and runs the program inside a sandbox
- some limits on what we can do there: e.g., no keyboard input, random numbers are deterministic, time is constant (i.e., not current)

[click here to try it](https://play.golang.org/) (<https://play.golang.org/>)

20

Editors

many editors have Golang plugins

- MacVim
- Atom
- Sublime
- Eclipse
- ...remember that Golang's philosophy is that IDEs are not required

21

Primitive Types

Golang Basic Types

bool true or false

string immutable

uint either 32 or 64 bits

int same size as uint

uintptr an unsigned int large enough to store the uninterpreted bits of a pointer value

uint8 the set of all unsigned 8-bit integers (0 to 255)

uint16 the set of all unsigned 16-bit integers (0 to 65535)

uint32 the set of all unsigned 32-bit integers (0 to 4294967295)

uint64 the set of all unsigned 64-bit integers (0 to 18446744073709551615)

int8/int16/32/64 the set of all signed 8-/16-/32-/64-bit integers

float32/64 the set of all IEEE-754 32-/64-bit floating-point numbers

complex64/128 set of all complex numbers with float32/64 real and imaginary parts

byte alias for uint8

rune alias for int32 (represents a Unicode code point)

23

Declaring Variables

The var statement is used to declare one or more variables

```
var (  
    name      string  
    age       int  
    almaMater string  
)
```

or

```
var (  
    name, almaMater string  
    age              int  
)
```

or individually:

```
var name      string  
var age       int  
var almaMater string
```

Initializing

all variables initialized to zero unless specified

```
var (  
    name      string = "Ranveer Singh"  
    age       int    = 34  
    almaMater string = "Indiana University"  
)
```

types are inferred if initializers are present

```
var (  
    name      = "Ranveer Singh"  
    age       = 34  
    almaMater = "Indiana University"  
)
```

single line initialization

```
var (  
    name, almaMater, age = "Ranveer Singh", "Indiana University", 34  
)
```

Declaration + Initialization Shorthand

- inside a function, the shorthand `:=` can be used

```
func main() {  
    name, almaMater := "Ranveer Singh", "Indiana University"  
    age := 34  
    println(name, age, almaMater)  
}
```

Run

- outside functions, everything begins with a keyword (`var`, `func`, etc.) and the `:=` shorthand is not available

26

Constants

- like variables, but declared with the `const` keyword (and of course can't be changed)
- integer, floating point, string, rune, boolean, and complex numbers
- can't use `:=` with constants
- type inferred if not supplied
- typically written in MixedCase or all caps

```
const SpeedOfLight = 2.99792458e8 // meters per second
const PI = 3.1415926 // already defined in math package
const SeparatorChar rune = ':'
const SpaceChar byte = ' '
const Message string = "1...2...3...Go!"
const (
    StatusOK          = 0
    StatusConnectionReset = 1
    StatusOtherError     = 2
)
```

Constants (cont'd)

- `iota` is a predeclared identifier that represents successive untyped integer constants

```
const (
    StatusOK          = iota // 0
    StatusConnectionReset   // 1
    StatusOtherError        // 2
)

func main() {
    println(StatusOK)
    println(StatusConnectionReset)
}
```

Run

28

Constants (cont'd)

- iota can be used in expressions so you can do things like this

```
type ByteSize float64

const (
    _           = iota // ignore first value by assigning to empty
    KB ByteSize = 1 << (10 * iota) // << is left shift operator
    MB
    GB
    TB
    PB
    EB
    ZB
)
func main() {
    println("GB =", GB)
    println("TB =", TB)
}
```

Run

- $1 \ll 10 = 0b1000000000 = 2^{10} = 1024$

29

Type Conversion/Casting

- $T(v)$ converts the value v to the type T

```
var p int = int(PI)
var f float64 = float64(p)
var u uint = uint(f)
```

- no automatic type promotion (as is done in C and some other languages)
- **EDIT THE CODE** to fix the problem after running the below and identifying the problem

```
func myfunc(f float64) int {
    return int(f)
}

func main() {
    x := 3
    y := myfunc(x)
    println(y)
}
```

Run

30

Converting to/from Strings

[strconv package](https://golang.org/pkg/strconv) (<https://golang.org/pkg/strconv>)

- most common conversions are `strconv.Atoi()` (ASCII to integer) and `strconv.Itoa()` (vice versa)
- super useful functions in there

31

Outputting

The fmt Package

as we've seen, `print` and `println` are built in to Golang

...but they may eventually go away (<https://golang.org/ref/spec#Bootstrapping>)

the `fmt` package contains versions that are more flexible (and it's considered more idiomatic to use `fmt`)

believe it or not, it's pronounced "fumpt"

33

fmt.Printf

- formatted print a la C/C++/Java
- %s = print a string value
- %d = print an integer value
- %f = print a float value

```
import (
    "fmt" // we import packages in order to use them
    "math" // we'll use the math package too
)

func main() {
    var name, age = "Ranveer Singh", 34
    fmt.Printf("Name is %s\n", name)
    fmt.Printf("%s is %d years old\n", name, age)
    fmt.Printf("The square root of %d is %f\n", age, math.Sqrt(float64(age)))
}
```

Run

34

fmt.Printf (cont'd)

- %v = print the value in a default format
- %T = print the type of the variable
- %t = print a Boolean value as true or false

```
import "fmt"

func main() {
    var f float64 = 38730204.3832
    fmt.Printf("%v\n", f)    // print f as a "value"
    fmt.Printf("%f\n", f)    // print f as a float
    fmt.Printf("%.2f\n", f) // restrict f to 2 places after decimal
    var i int = 13
    fmt.Printf("|%5d|\n", i) // print i 5 spaces wide
    fmt.Printf("|%-5d|\n", i) // how is this different?
    var b bool = false
    fmt.Printf("%t\n", b)
    fmt.Printf("%T %T %T\n", f, i, b)
}
```

Run

fmt.Sprintf

- "print" into a string, rather than to the screen
- also Sprint and Sprintln

```
package main

import "fmt"

func main() {
    f := 1.23456789
    i := 123456789
    s := fmt.Sprintf("Float value %.3f and int |%15d|", f, i)
    fmt.Println(s)
}
```

Run

36

Logging

Golang log package used for debugging (<https://golang.org/pkg/log/>)

(by the way, we can use the godoc tool if we are offline)

- by default, log messages go to stderr
- can be configured to send log messages to files

much better than fmt, but somewhat lacking as loggers go

- doesn't provide levels (e.g., debug, notice, warning, etc.)
- doesn't let you configure separate loggers for each package
- Google gives you glog (<https://godoc.org/github.com/golang/glog>)
- also consider loggo (<https://godoc.org/github.com/juju/loggo>)

Simple Logging Example

```
package main

import (
    "fmt"
    "log"
    "time"
)

func main() {
    // set up a logger to include the date and time in microseconds
    log.SetFlags(log.Ldate | log.Lmicroseconds)
    // set up a prefix which for every log message
    log.SetPrefix("MyApp ")
    // Start program?
    log.Println("Kaaryakram shuroo karen")
    // do some stuff...
    time.Sleep(103871 * time.Microsecond)
    log.Println("Database connection dropped.")
    // cf. with standard fmt.Println
    fmt.Println("Database connection dropped.")
}
```

Run

38

Packages

What's a Package?

- every Go program is made up of packages, which are file(s) containing functions, variables, constants, etc.
- every Go program file has to be associated with a package via the package command at the top
- programs start running in

```
package main
```

40

Package Naming

- by convention, the package name is the same as the last element of the import path
- e.g., to import the `math/rand` package, one would type

```
import "math/rand"
```

but one would refer to objects inside the package, like so

```
rand.Int()
```

Source files for the `rand` package are prefaced with

```
package rand
```

[Click here to view source code](#) (<https://github.com/golang/go/blob/master/src/math/rand/rand.go>)

41

The main Package

- as we know, execution begins in package main
- when building a standalone executable, main must exist
- when building a reusable package, there won't be a main

42

Importing Packages

- importing makes packages available in our programs
- packages can be given a nickname in the event of a name collision (two packages with the same name)
- you can import a package such that you don't need to use the package name prefix when accessing functions and data inside the package (not recommended)

```
package main

import "math/rand"
import myrand "other/rand"
import . "fmt"

func main() {
    rand.Seed(...)
    myrand.Something(...)
    println("Hello, world!")
}
```

Packages: Identifiers Beginning with Upper Case Letters are Exported

- if you want something to be private, start its name with a lower case letter
- if you want something to be public, start its name with an upper case letter
- no public or private keyword

```
package example

var private = "This is not exported."
var Public = "This variable IS exported."

func nothing(x int) int {
    return x - 1
}

func Something(x int) int {
    return x + 1
}
```

Exercise: Packages

- create your own package in your \$GOPATH/src directory
- be sure to export at least one function and one constant
- build your package with go build
- install your package with go install
- write a program in different subdirectory which imports your package

45

Possibly Painful Pedantry?

- Go complains about unused variables and imports
- ...and the Go compiler has no warnings
- you may be annoyed by this, but the Go team explains it well...

```
package main

import (
    "fmt"
    "math"
)

func main() {
    x := 1
    fmt.Println("Hello!")
}
```

Run

[view in playground](https://play.golang.org/p/OzTo_vd9ADo) (https://play.golang.org/p/OzTo_vd9ADo)

46

Possibly Painful Pedantry? (cont'd)

"The presence of an unused variable may indicate a bug, while unused imports just slow down compilation, an effect that can become substantial as a program accumulates code and programmers over time. For these reasons, Go refuses to compile programs with unused variables or imports, trading short-term convenience for long-term build speed and program clarity.

The reason for having no warnings. If it's worth complaining about, it's worth fixing in the code. And if it's not worth fixing, it's not worth mentioning.

Nowadays, most Go programmers use a tool, `goimports`, which automatically rewrites a Go source file to have the correct imports, eliminating the unused imports issue in practice."

47

Functions

Functions: Declaration, Arguments, Return Values

- a named (maybe) bit of code which is typically reused
- introduced with `func` keyword
- accept any number of arguments, return any number of values

```
// return the square of a number
func squareit(val float64) float64 {
    return val * val
}

// return sum of two values
func sum(x int, y int) int { // equivalent: func sum(x, y int) int
    return x + y
}

func main() { // return nothing
    fmt.Printf("%f\n", squareit(4.5))
    fmt.Println(sum(13, -2))
}
```

Run

49

main Function

- as we've seen, the main program must be inside a function called `main`
- this function takes no arguments and returns nothing...

```
package main

func main() { // minimal Golang program
}
```

50

Returning Multiple Values

- Go functions can return multiple values (much like Python)
- unlike Python, types must of course be specified
- the blank identifier `_` can be used to discard a value (also like Python)

```
package main

import "fmt"

func addmul(x, y int) (int, int) {
    return x + y, x * y
}

func main() {
    s, p := addmul(3, 5)
    fmt.Println("sum =", s, "product =", p)
    _, r := addmul(4, -8)
    fmt.Println("product =", r)
}
```

Run

Exercise: Functions

- write a function called `circleinfo` which accepts a `float64` radius and returns two values, the area of the circle, and the circumference
- $\text{area} = \text{Pi} * \text{radius} * \text{radius}$
- $\text{circumference} = \text{Pi} * \text{radius} * 2$
- `Pi` is defined in the `math` package

52

More About Return Values

- you can name the return values
- you still need a return statement
- **EDIT THE CODE** to remove the named return values and be sure it still runs properly

```
package main

import "fmt"

// if you name the parameters, you don't have to return them
// directly in the return statement...but I don't recommend it
func multival(x int) (square, cube int) {
    square = x * x
    cube = x * x * x
    return
}

func main() {
    fmt.Println(multival(3))
}
```

Run

Builtin Functions

- docs in <https://golang.org/pkg/builtin> but code is not actually in a package
- we've already seen `print` and `println`
- `complex`, `real`, and `imag` are for imaginary numbers

```
func append(slice []Type, elems ...Type) []Type
func cap(v Type) int {} // capacity of...
func close(c chan<- Type)
func complex(r, i FloatType) ComplexType
func copy(dst, src []Type) int
func delete(m map[Type]Type1, key Type)
func imag(c ComplexType) FloatType
func len(v Type) int
func make(t Type, size ...IntegerType) Type
func new(Type) *Type
func panic(v interface{}) 
func print(args ...Type)
func println(args ...Type)
func real(c ComplexType) FloatType
func recover() interface{}
```

Anonymous Functions

- functions that live in memory, but have no name
- they still may be referred to
- often called lambda functions in other languages

```
// This function takes a function as a parameter. The function
// passed in must take an int argument and return an int.
func runsomething(f func(int) int, i int) int {
    return f(i)
}

func main() {
    // Here we call the above function and pass in an
    // anonymous function (or function literal)
    x := runsomething(
        func(a int) int {
            return a * 2
        }, -5)
    fmt.Println(x)
}
```

Run

Anonymous Functions (cont'd)

- functions are first class objects, that is, they can be assigned to variables as well as passed in to other functions

```
// This function takes a function as a parameter. The function
// passed in must take an int argument and return an int.
func runsomething(f func(int) int, i int) int {
    return f(i)
}

func main() {
    // Here we make the variable lambda refer to
    // the function, and then pass lambda into the
    // function above.
    lambda := func(a int) int {
        return a * 2
    }
    fmt.Println(runsomething(lambda, -5))
}
```

Run

init Function

- each source file can define its own niladic (zero argument) init function (actually each file can have multiple init functions)
- can be used to set up whatever state is required
- init is called after 1) all imported packages have been initialized; 2) all variable declarations in the package have evaluated their initializers
- a common use of init functions is to verify or repair correctness of the program state before real execution begins
- let's see an example...

57

init Function (cont'd)

```
package main

import "fmt"

var _ int = setup() // happens first

func init() { // happens second
    fmt.Println("init!")
}

func init() { // happens third
    fmt.Println("something else")
}

func setup() int {
    fmt.Println("calling setup()")
    return 1
}

func main() { // happens last
    fmt.Println("main")
}
```

Run

Calling Functions

- Golang is pass-by-value ONLY (like C, which was a starting point of sorts)
- that means if you pass a variable or other object to a function, the function cannot change the variable/object because it receives a copy of it

```
package main

import "fmt"

// this function cannot change the variable passed in to it
func increment(num int) {
    num++
}

func main() {
    val := 1
    increment(val)
    fmt.Println("Value after incrementing:", val)
}
```

Run

59

How to Modify Parameters to a Function?

- in order for a function to modify its parameters, the function must accept a *pointer* to the object it wants to modify
- a *pointer* is a variable which contains the memory address of another variable
- there are two operators for dealing with pointers
- the & operator takes the *address* of the object that follows it
- the * operator *dereferences* a pointer, that is, it gives you access to the object at the address contained in the pointer

60

Pointers

```
package main

import "fmt"

func main() {
    x := 1
    px := &x // The & operator takes the address of what follows it
    fmt.Printf("The value of x is %d\n", x)
    fmt.Printf("The address of x is %v\n", px)
    fmt.Printf("The canonical form of px is %#v\n", px)
}
```

Run

- now let's revisit our earlier attempt to have a function change its parameter, but this time we'll use pointers...

61

Pointers (cont'd)

```
package main

import "fmt"

// the argument to this function is a pointer to an integer
// "star int" = pointer to int
func increment(nump *int) {
    fmt.Printf("the value of nump is %v\n", nump)
    // now we increment the object that nump is pointing to
    *nump++
}

func main() {
    val := 1
    // this time we call increment, but pass a pointer to the val variable
    ptr_to_val := &val
    fmt.Printf("ptr_to_val is the address of val... %v\n", ptr_to_val)
    increment(ptr_to_val)
    fmt.Println("Value after incrementing:", val)
}
```

Run

- (of course we wouldn't use the intermediate variable `ptr_to_val`, it's there for instructive purposes)

62

Exercise: Pointers and Functions

Write a function `doubler` which takes a string argument and doubles it, i.e., "Golang" would become "GolangGolang"

- note that '+' is the way to concatenate two strings
- your function should not return anything, it should modify its argument directly
- write a main program to test your function
- once you've done that, make it so that `doubler` calls a second function, `doublerHelper` which actually does the doubling (to understand how to pass pointers around)

63

Functions as Arguments

```
package main

import "fmt"

// this function takes three arguments: 1) a function which takes two ints
// as parameters, 2) an int, and 3) another int and then invokes that function,
// passing the two ints as parameters to the called function
func funcrunner(f func(x, y int), x, y int) {
    f(x, y)
}

// this function takes 2 ints and prints their values
func printer(a, b int) {
    fmt.Println("a and b are", a, b)
}

func main() {
    // pass the printer function as the first argument
    // and 1 and 3 as the additional arguments
    funcrunner(printer, 1, 3)
}
```

Run

64

Getting Input from the User

Command-Line Args

- import the os package to gain access to the command-line arguments
- os.Args is an array of strings that we can inspect
- we can't run these in the playground—we need to compile and run locally, then pass command-line args

```
package main

import (
    "fmt"
    "os"
)

func main() {
    fmt.Println("All command-line args:", os.Args)
    fmt.Println("Number of args =", len(os.Args))
    fmt.Println("Arg 0 is", os.Args[0])
    fmt.Println("Args 1-3 are", os.Args[1:4])
}
```

Exercise: Command Line Args

- write a program which takes two command line args representing integers and prints out their sum
- test it on your machine since it won't run in the playground
- hint: remember Atoi in the strconv package (<https://golang.org/pkg/strconv>) and ignore errors for now

67

Interactive User Input

- `fmt` package contains `Scan`, `Scanf`, `Scanln` functions to read from keyboard
- as with the previous examples, we can't run these examples in the playground, since they require keyboard input
- let's start with `Scanln`, which reads until a newline is seen (but spaces count as newlines here)

```
package main

import "fmt"

func main() {
    var input string
    fmt.Print("Enter text: ")
    fmt.Scanln(&input)
    fmt.Println("I think you just entered...", input)
}
```

fmt.Scan

- read from standard input, storing successive space-separated values into successive arguments
- newlines count as space

```
package main

import "fmt"

func main() {
    var input string
    var n int
    fmt.Print("Enter text: ")
    fmt.Scan(&input, &n)
    fmt.Println("I think you just entered...", input, n)
}
```

fmt.Scanf

- reads text from standard input, storing successive space-separated values into successive arguments as determined by the format string
- newlines in the input must match newlines in the format
- one exception: the verb %c always scans the next char in the input, even if it is a space (or tab etc.) or newline

```
package main

import "fmt"

func main() {
    var word string
    var int1, int2 int
    var float1 float32

    fmt.Print("Enter text: ")
    fmt.Scanf("%d %f %d\n%s", &int1, &float1, &int2, &word)
    fmt.Println("I think you just entered...", int1, float1, int2, word)
}
```

Control Structures

if / else

- no parentheses required
- braces always required
- else clause is of course optional

```
package main

import "fmt"

func main() {
    x := 27
    if x % 2 == 0 {
        fmt.Println(x, "is even")
    } else {
        fmt.Println(x, "is odd")
    }
}
```

Run

72

compound if statement

- a statement can precede conditionals
- any variables declared in this statement are available in all branches
- very idiomatic in Golang

```
func somenumber() int {  
    return -7  
}  
  
func main() {  
    if num := somenumber(); num < 0 {  
        fmt.Println(num, "is negative")  
    } else if num < 10 {  
        fmt.Println(num, "has 1 digit")  
    } else {  
        fmt.Println(num, "has multiple digits")  
    }  
}
```

Run

73

switch Statement

- express conditionals across many branches
- no parens
- simplest form...

```
import "time"
import "math/rand"

func main() {
    sec := time.Now().Unix() // get current time as seconds since 1970
    rand.Seed(sec)          // "seed" the random number generator
    i := rand.Int31n(4)      // generate random number between 0 and 3

    switch i {
    case 0:
        fmt.Println("zero...")
    case 1:
        fmt.Println("one...")
    case 2:
        fmt.Println("two...")
    }
    fmt.Println("ok")
}
```

Run

switch: Match Against Multiple Expressions

```
package main

import "fmt"

func location(city string) (string, string) {
    var region string
    var continent string

    switch city {
    case "Delhi", "Hyderabad", "Mumbai", "Kolkata", "Chennai", "Kochi":
        region, continent = "India", "Asia"
    case "Lafayette", "Louisville", "Boulder":
        region, continent = "Colorado", "USA"
    default:
        region, continent = "Unknown", "Unknown"
    }
    return region, continent
}

func main() {
    region, continent := location("Lafayette")
    fmt.Printf("Dave works in %s, %s\n", region, continent)
}
```

Run

switch: Invoke a Function

```
package main

import (
    "fmt"
    "time"
)

func main() {
    // time in playground is a constant, so should be run on your machine...
    switch time.Now().Weekday().String() {
        case "Monday", "Friday", "Sunday":
            fmt.Println("It's a 6-letter day")
        default:
            fmt.Println("It's NOT a 6-letter day")
    }
    // let's print out the day to check
    fmt.Println(time.Now().Weekday().String())
}
```

Run

76

switch: Condition Omitted

- condition can be omitted altogether, simulating an if
- ...or think of it as switch true

```
import (
    "fmt"
    "math/rand"
    "time"
)

func main() {
    rand.Seed(time.Now().Unix()) // use seconds since 1970 to seed random number generator
    r := rand.Float64()          // generate random number between 0 and 1

    switch {
    case r > 0.1:
        fmt.Println("Common case, 90% of the time")
    default:
        fmt.Println("10% of the time")
    }
}
```

Run

switch: Regular Expressions

- if you want to match a particular pattern, you can use Golang's regular expression package in conjunction with having switch call a function

```
import "regexp"

func main() {
    var email = regexp.MustCompile(`^[@]+@[^.]+\.[^.]+`)
    var phone = regexp.MustCompile(`^([()?[0-9][0-9][0-9](). \-]*[0-9][0-9][0-9][.\-]?[0-9][0-9]`)

    contact := "foo@bar.com"

    switch {
        case email.MatchString(contact):
            fmt.Println(contact, "is an email")
        case phone.MatchString(contact):
            fmt.Println(contact, "is a phone number")
        default:
            fmt.Println(contact, "is not recognized")
    }
}
```

Run

78

switch: fallthrough

- in C, each branch of a switch statement must be terminated with a break statement, or else execution will "fall through" to the next branch-WITHOUT RUNNING THE TEST-after a branch is executed
- in Go, if you want to fall through, you must explicitly state it

```
num := 1111

switch {
case num > 1000:
    fmt.Println("greater than 1000")
    fallthrough // go to next statement

case num > 100:
    fmt.Println("greater than 100") // (this is next statement)
    fallthrough // go to next statement

case num > 10:
    fmt.Println("greater than 10") // (this is next statement)

default:
    fmt.Println("less than 10")
}
```

Run

Exercise: Fizzbuzz

Write a function which accepts an integer and returns

- "fizz" if the number is divisible by 3
- "buzz" if the number is divisible by 5
- "fizzbuzz" if the number is divisible by BOTH 3 and 5
- otherwise it just returns the string version of the number, e.g., "4"

Test your function with these inputs: 3, 5, 15, 4

80

for Loop

Go only has one looping construct

- basic for loop looks as it does in C/C++/Java, except
- no parens
- braces required
- as in C/C++/Java, you can have empty pre and post statements

```
func main() {  
    sum := 0  
    for i := 1; i <= 100; i++ {  
        sum += i  
    }  
    fmt.Println("sum of 1..100 is", sum)  
}
```

Run

81

Exercise: for Loops

implement a square root function using Newton's method:

- starting with some guess for the square root of x, we can adjust it based on how close guess^2 is to x, producing a better guess:
- $\text{guess} = \text{guess} - (\text{guess}^2 - x) / (2 * \text{guess})$
- repeating the above makes the guess better and better
- use a starting guess of 1.0, regardless of the input (it works quite well)
- repeat the calculation 10 times and print each guess along the way

OR: implement the factorial function, $n! = n * (n - 1) * \dots * 1$

82

for Loop: Empty Pre Statement

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func main() {
    sec := time.Now().Unix()    // seconds since 1970...
    rand.Seed(sec)              // seed the random number generator
    num := rand.Int31n(10) + 5 // generate random num between 5 and 14

    for ; num > 0; num-- {
        fmt.Println(num)
    }
    fmt.Println("Blast off!")
}
```

Run

83

for loop as while loop

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func main() {
    var num int64
    rand.Seed(time.Now().Unix()) // seed the random number generator

    for num != 5 {
        num = rand.Int63n(15)
        fmt.Println(num)
    }
}
```

Run

Exercise: for Loop as while Loop

Write a function which implements the Collatz Conjecture:

- it should accept an integer ≥ 1 (return false if < 1)
- if it's even, divide it by 2
- if it's odd, multiply it by 3 and add 1
- stop when the result is 1
- return true for success

85

for loop as Infinite Loop

- as we are no doubt used to from other languages, we can break out of the enclosing loop with the keyword `break`

```
import (
    "fmt"
    "math/rand"
    "time"
)

func main() {
    var num int32
    sec := time.Now().Unix() // seconds since 1970...
    rand.Seed(sec)           // seed the random number generator

    for {
        fmt.Println("Infinite loop...")
        if num = rand.Int31n(10); num == 5 {
            fmt.Println("break!")
            break
        }
        fmt.Println(num)
    }
}
```

Run

Exercise: for Loop as Infinite Loop

- write an infinite loop that reads a string using `fmt.Scanf` and then uses `strconv.Atoi` to convert it to an integer
- if the conversion fails to produce an integer, print an error message and prompt the user again
- since we don't yet know how to deal with errors returned from functions, just examine the int returned from `strconv.Atoi` and treat 0 as an error
- stop the loop when the user complies, i.e., an integer is entered

87

continue Statement

- skips the current iteration of the loop
- you don't really need continue, since any code written with a continue can be written without one
- they can be handy in cases where you don't want to execute the loop until you've checked some error conditions

```
package main

import "fmt"

func main() {
    sum := 0
    for num := 1; num <= 100; num++ {
        if num % 5 == 0 {
            continue
        }
        sum += num
    }
    fmt.Println("sum 1..100 excluding nums divisible by 5 is", sum)
}
```

Run

goto ("Considered Harmful")



[Edsger Dijkstra - Go To Statement Considered Harmful](https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf) (<https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf>)

[A Case against the GO TO Statement](https://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF) (<https://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF>)

- Go has a goto statement—judicious use of it makes your code easier to understand
- jump to a named label
- must not skip over any variable declarations
- a goto outside a block cannot jump to a label inside that block

goto Statement: Cleanly Exiting a Function

- you may allocate resources and need to exit in multiple places
- putting cleanup code at end of the function with goto simplifies code
- you don't have to write cleanup code at every "exit point" of function

90

goto Statement: Cleanly Exiting a Function (example)

```
func foo(bar int) int {
    return_value := 0

    if ret1, err := do_something(bar); err {
        goto error_1
    }

    if ret2, err := init_stuff(ret1); err {
        goto error_2
    }

    if ret3, err := prepare_stuff(ret2); err {
        goto error_3
    }

    return_value = do_the_thing(ret3)
error_3:
    cleanup_3()
error_2:
    cleanup_2()
error_1:
    cleanup_1()
    return return_value
}
```

break with a Label

- useful inside multiple nested loops

```
package main

import "fmt"

func main() {
loop:
    for i := 1; i <= 5; i++ {
        for j := 1; j <= 5; j++ {
            if i*j == 9 {
                fmt.Println("---")
                break loop // skip rest of both loops
            }
            fmt.Println(i, j)
        }
    }
}
```

Run

92

continue with a Label

```
package main

import "fmt"

func main() {
loop:
    for i := 1; i <= 5; i++ {
        for j := 1; j <= 5; j++ {
            if i*j == 9 {
                fmt.Println("---")
                continue loop // skip rest of this loop
            }
            fmt.Println(i, j)
        }
        fmt.Println("j loop finished")
    }
}
```

Run

defer

defer pushes a function call onto a stack, and the saved calls are executed just prior to the surrounding function returning

commonly used to simplify functions with cleanup actions

three rules

- deferred function's arguments are evaluated when the defer statement is evaluated
- deferred function calls are executed in Last In First Out order
- deferred functions may read/assign to the returning function's named return values

```
func main() {  
    for i := 1; i <= 4; i++ {  
        defer fmt.Println("deferred", -i)  
        fmt.Println("regular", i)  
    }  
}
```

Run

Exercise: defer

- write a program which repeatedly asks to enter an integer
- your program should quit when the user enters a 0
- if the number is negative, the program should print an error and quit
- your program should perform a "cleanup action" when it quits, and this cleanup action should occur regardless of whether it quits normally or if there was an error (negative number)
- put your cleanup action in a deferred anonymous function

95

panic

- built-in function that stops the ordinary flow of control and begins "panicking"
- when function F calls `panic()`, execution of F stops, any deferred functions in F are executed normally, and then F returns to its caller
- to the caller, F then behaves like a call to `panic()`
- process continues up the stack until all functions have returned, at which point the program crashes
- panics can also be caused by runtime errors, such as out-of-bounds array accesses
- let's see an example of `defer` plus `panic()`...

96

panic (cont'd)

```
package main

import "fmt"

func main() {
    f()
    fmt.Println("Back from f()")
}

func f() {
    fmt.Println("Calling g()")
    g(0)
    fmt.Println("Back from g()")
}

func g(i int) {
    if i > 3 {
        fmt.Println("Panicking!")
        panic("Panic in g() (major)")
    }
    defer fmt.Println("Defer in g()", i)
    fmt.Println("Printing in g()", i)
    g(i + 1)
}
```

Run

recover

- built-in function that regains control of a panicking goroutine
- only useful inside deferred functions
- during normal execution, a call to recover() will return nil (like NULL in other languages) and have no other effect
- if current goroutine is panicking, a call to recover() will capture the value passed to panic() and resume normal execution
- let's add recover() to previous example...

98

recover (cont'd)

```
func main() {
    f()
    fmt.Println("Back from f()")
}

func f() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered in f", r)
        }
    }()
    fmt.Println("Calling g()")
    g(0)
    fmt.Println("Back from g()")
}

func g(i int) {
    if i > 3 {
        fmt.Println("Panicking!")
        panic("Panic in g() (major)")
    }
    defer fmt.Println("Defer in g", i)
    fmt.Println("Printing in g", i)
    g(i + 1)
}
```

Run

Exercise: panic/recover

- modify the previous exercise to panic when a negative number is entered (we wouldn't really use panic for something like this)
- use the math/rand module to panic randomly, say 10% of the time, with a different panic message
- have the calling function recover, but only if the panic was due to a negative number-if the random panic occurs, the program should crash with a panic

100

Scope

Scoping Rules

- the scope of a predeclared identifier is the universe block, e.g., true, false, nil, iota, types, and built-in functions
- the scope of an identifier denoting a constant, type, variable, or function (but not method) declared at top level (outside any function) is the package block
- the scope of the package name of an imported package is the file block of the file containing the import declaration
- the scope of a function parameter or result variable is the function body
- the scope of a constant or variable declared inside a function begins at the end of the ConstSpec or VarSpec and ends at the end of the innermost containing block
- an identifier declared in a block may be redeclared in an inner block

while the identifier of the inner declaration is in scope, it denotes the entity declared by the inner declaration

102

Scope Example

```
package main

import "fmt"

func main() {
    f := 1
    // i defined only inside the for block
    for i := 1; i < 2; i++ {
        fmt.Println("i is in scope here", i)
    }
    // fmt.Println(i) => i NOT in scope here
    if j := 4.5; j > 10 {
        fmt.Println("j is in scope here:", j)
    } else {
        fmt.Println("j also is in scope here:", j)
    }
    // fmt.Println(j) => j NOT in scope here
    if f == 1 {
        f := "new f, inside this block, hides the outer f"
        fmt.Println("in the block, f is", f)
    }
    fmt.Println(f)
}
```

Run

103

Data Structures

Arrays

- both type of element and length are part of the type
- cannot be resized
- zero-valued by default
- brittle, but still useful

```
func main() {  
    var a [10]int // array of 10 integers, initialized to 0 by default  
    fmt.Println(a)  
    for i := 0; i < len(a); i++ {  
        a[i] = i + 101  
    }  
    fmt.Println(a)  
    fmt.Println("a[3] =", a[3]) // array indexing  
}
```

Run

105

Arrays (cont'd)

- can be initialized when declared

```
func main() {  
    // Arrays can be initialized when they are declared  
    cities := [5]string{"Hyderabad", "Delhi", "Pune", "Kochi", "Chennai"}  
    fmt.Println("Cities:", cities)  
}
```

Run

106

Arrays (cont'd)

- no need to specify size when initializing

```
func main() {  
    // ellipsis (...) can be used in place of size  
    // if array is being initialized  
    nums := [...]float32{ 1.4, 2.67, -3.56 }  
    fmt.Println("len(nums) =", len(nums))  
    chars := [...]rune{ 'G', 'o', 'l', 'a', 'n', 'g', '€' }  
    fmt.Printf("%q\n", chars) // %q == quoted  
    fmt.Println(chars) // values are actually int32  
}
```

Run

107

2-D Arrays

- array of arrays
- each row is a 1-D array

```
func main() {  
    var twoD [3][5]int // 2-dimensional array  
    for i := 0; i < 3; i++ {  
        for j := 0; j < 5; j++ {  
            twoD[i][j] = (i + 1) * (j + 1)  
        }  
        fmt.Println("Row", i, twoD[i])  
    }  
    fmt.Println("\nAll at once:", twoD)  
    look(twoD[1]) // send one row to a function...  
}  
  
func look(thing [5]int) {  
    fmt.Println("I see:", thing)  
}
```

Run

108

Slices

- key datatype in Go
- more powerful interface to sequences than arrays
- slices are typed only by elements they contain (not number of elements)
- to create an empty slice with non-zero length, use builtin make function

```
func main() {  
    // create a slice of strings of size 3  
    s := make([]string, 3)  
    fmt.Printf("initial slice: %v, len = %d\n", s, len(s))  
    s[0] = "zero" // set just like an array  
    s[1] = "one"  
    s[2] = "two"  
    fmt.Println("now:", s)  
    fmt.Println("get:", s[2]) // get like an array  
}
```

Run

109

Slices: append()

- `append()` returns a new slice containing appended items
- Why is the syntax of `append()` the way it is?

```
func main() {  
    s := make([]string, 3) // slice of strings of size 3  
    fmt.Println("initial slice:", s)  
    s[0] = "zero" // set just like an array  
    s[1] = "one"  
    s[2] = "two"  
    fmt.Println("now:", s)  
    fmt.Println("get:", s[2]) // get like an array  
    fmt.Println("len of slice:", len(s))  
    s = append(s, "three")  
    s = append(s, "four", "five")  
    fmt.Println("after append:", s)  
}
```

Run

110

Slicing Slices

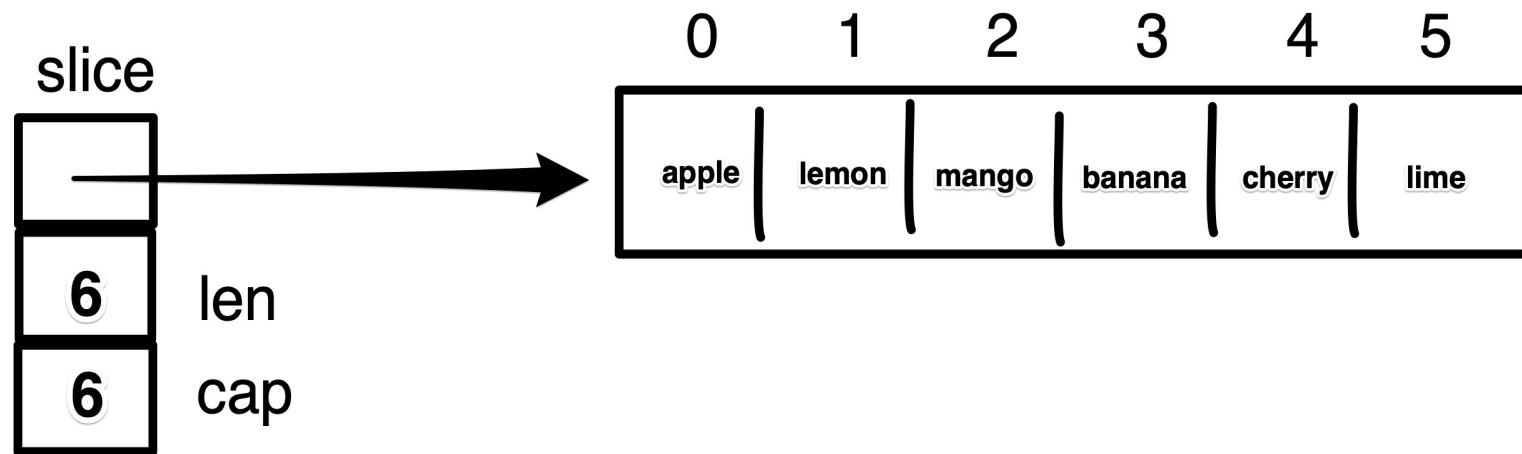
- slices (and arrays) support a slicing operator (much like Python)

```
func main() {  
    slice := make([]string, 0)  
    slice = append(slice, "apple", "lemon", "mango", "banana", "cherry", "lime")  
    fmt.Println(slice)  
    // Slices support a "slice" operator with the syntax slice[low:high]  
    slice1 := slice[2:5]  
    fmt.Println("slice1:", slice1)  
    // This slices up to (but excluding) `s[5]`.  
    slice2 := slice[:5]  
    fmt.Println("slice2:", slice2)  
    // And this slices up from (and including) `s[2]`.  
    slice3 := slice[2:]  
    fmt.Println("slice3:", slice3)  
}
```

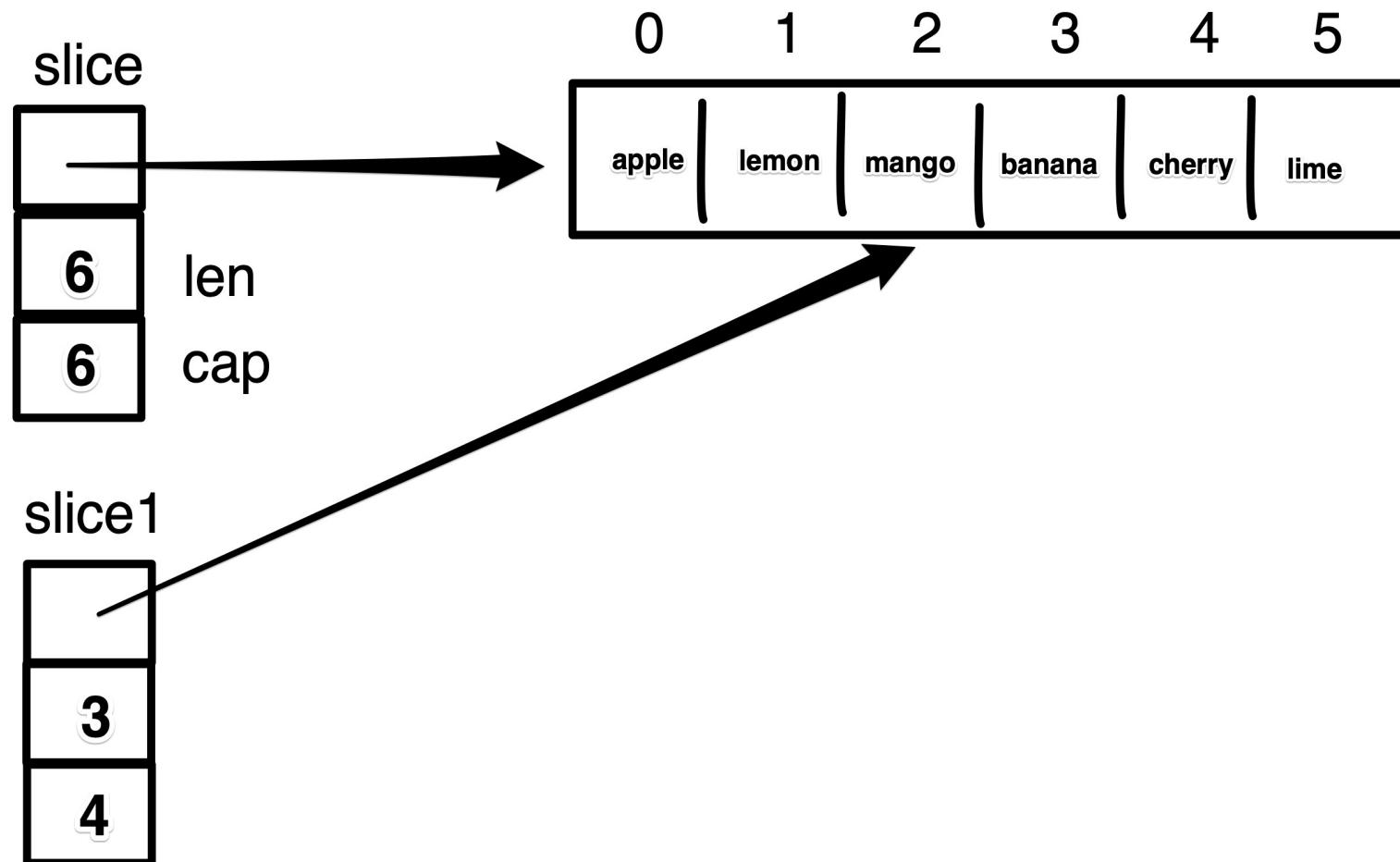
Run

111

What does slice our look like?



...and slice1?



Slices: copy()

- if we want to make a copy of a slice we need to use the `copy()` function
- ...but first we need to make a new slice to receive the copy

```
func main() {  
    s := make([]string, 3)  
    s[0], s[1], s[2] = "zero", "one", "two"  
    fmt.Println("initial slice:", s)  
    copy_of_s := make([]string, len(s))  
    copy(copy_of_s, s)  
    copy_of_s[0] = "ZERO!"  
    fmt.Printf("%v\n%v", copy_of_s, s)  
}
```

Run

114

Slices without make()

- we can declare a slice inline, without using make()

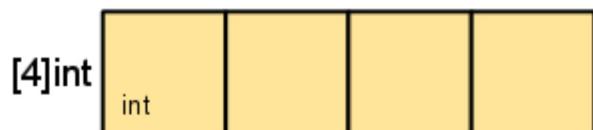
```
func main() {  
    s := []string{"raspberry", "orange", "guava"}  
    fmt.Println("declared slice:", s)  
    t := []string{"cherry", "banana"}  
    s = append(s, t[0], t[1])  
    fmt.Println(s, len(s), cap(s))  
}
```

Run

115

Slices vs. Arrays

- an array type definition specifies both length and element type
- for e.g., [4]int is an array of 4 ints



- a slice, on the other hand, is a descriptor of a segment of an array
- a slice consists of a pointer to the array, the length of the segment, and its capacity



blog.golang.org/go-slices-usage-and-internals (<https://blog.golang.org/go-slices-usage-and-internals>)

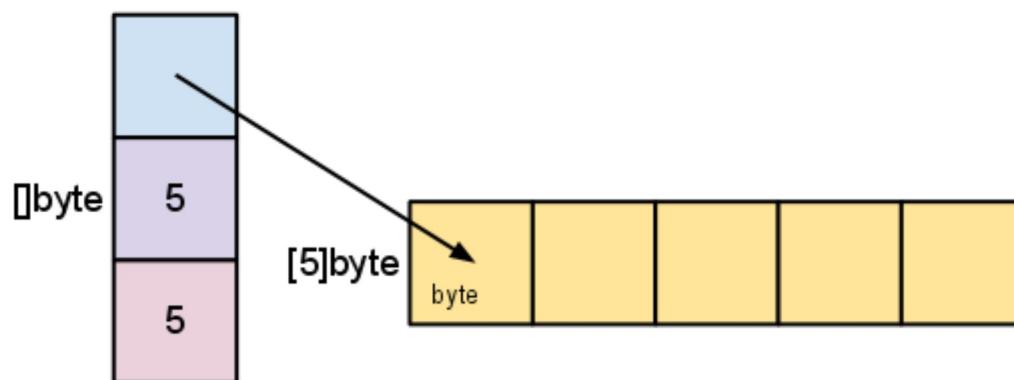
116

Slices vs. Arrays (cont'd)

- consider the following slice

```
s := make([]byte, 5)
```

- under the hood it looks like this



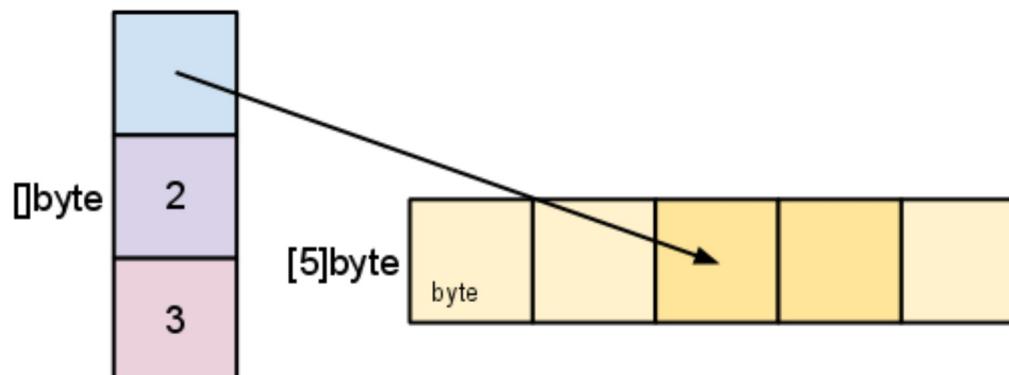
- what if we slice that slice?

blog.golang.org/go-slices-usage-and-internals (<https://blog.golang.org/go-slices-usage-and-internals>)

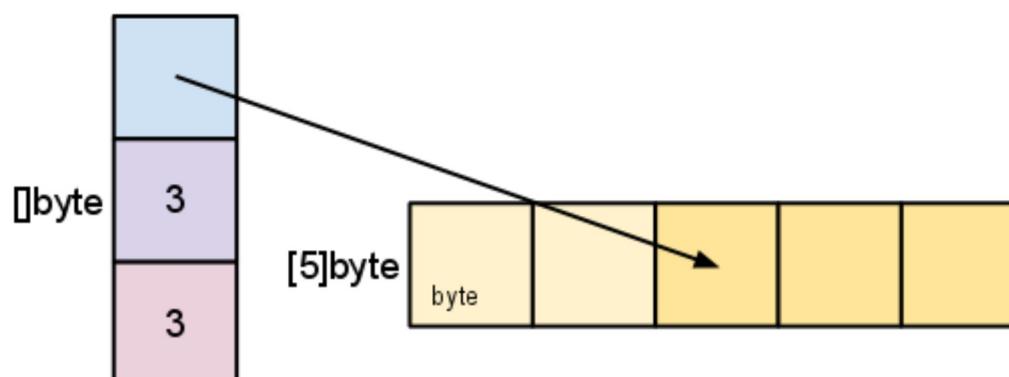
117

Slices vs. Arrays (cont'd)

```
s = s[2:4]
```



```
s = s[:cap(s)]
```



nil slice vs. empty slice

- the zero value of a slice is nil
- a nil slice has a length and capacity of 0, but has no underlying array
- a nil slice is not the same as an empty slice, which have an underlying array of size 0

```
func main() {  
    var n []int  
    z := make([]int, 0)  
  
    fmt.Printf("n is %#v %d %d\n", n, len(n), cap(n))  
    if n == nil {  
        fmt.Println("n is nil!\n")  
    }  
  
    fmt.Printf("z is %#v %d %d\n", z, len(z), cap(z))  
    if z == nil {  
        fmt.Println("z is nil!")  
    } else {  
        fmt.Println("z is NOT NOT NOT nil!")  
    }  
}
```

Run

Exercise: Slices

- there are two main ways to write a function that uses slices...either accept a slice as input and iterate through it, or have your function return a slice that it allocates from within the function
1. write a function `minmaxavg` which accepts a `float64` slice and returns four values: the minimum value in the slice, the maximum value in the slice, the average of all the values, and an indication of success (as we did previously)
 2. write a function `fib` that returns a slice containing the first n Fibonacci numbers
 - the Fibonacci sequence is 1, 1, 2, 3, 5, 8...
 - each number is the sum of the previous Fibonacci numbers
 - so `fib(6)` should return an `int` slice of size 6 containing 1, 1, 2, 3, 5, 8
 - in the event of an error, `fib` should return the nil slice

120

Variadic Functions

- can be called with any number of arguments
- e.g., `fmt.Println` is a variadic function

3. write a function counter which accepts a slice of strings and returns a new slice of the strings which contains 3 or more e's
- check out the `strings` package

```
import "fmt"

func variadic(words ...string) {
    for i := 0; i < len(words); i++ {
        fmt.Print(words[i])
        if len(words[i]) == 6 {
            fmt.Print(" ***")
        }
        fmt.Println()
    }
}

func main() {
    // Variadic functions can be called in the usual way with individual arguments
    variadic("this", "that", "hello", "Golang")
    variadic("C++", "Python")
    variadic()
```

{}

Run

Variadic Functions (cont'd)

- if you already have the args in a slice, you can pass them as individual args using the ... notation

```
func main() {  
    words := []string{"local", "locale", "localize"}  
    variadic(words...)  
}
```

Run

122

Exercise: Variadic Functions

- write a variadic function called product which takes a variable number of integers and returns their product
- if no integers are passed, product should return 1
- e.g., `product(1, 2, 3)` should return 6

123

Maps

- Go's built-in associative datatype (like Python dictionaries)
- key/value pairs
- a missing key does not cause an error (as it does in Python)

```
func main() {  
    continents := map[string]string{  
        "United States": "North America",  
        "India": "Asia",  
        "Namibia": "Africa",  
        "Germany": "Europe",  
        "Mexico": "North America",  
        "Ireland": "Europe",  
        "Iceland": "Europe",  
    }  
    c := "Iceland"  
    fmt.Println(c, "is in", continents[c])  
    fmt.Printf("%#v\n", continents)  
    c = "Freedomia"  
    fmt.Println(c, "is in", continents[c])  
}
```

Run

124

Maps (cont'd)

- we can also create an empty map and add to it later
- `len()` returns size of map (number of key/value pairs)
- `delete()` deletes items from a map

```
func main() {  
    // To create an empty map, use make  
    sbux := make(map[string]int)  
    // Set key/value pairs  
    sbux["tall"] = 12  
    sbux["grande"] = 16  
    sbux["venti"] = 20  
    fmt.Println(sbux)  
    fmt.Println("len:", len(sbux))  
    delete(sbux, "grande")  
    fmt.Println(sbux)  
  
    // Optional second return value indicates if key was in map.  
    // Used to disambiguate between missing keys and zero values.  
    _, present := sbux["grande"]  
    fmt.Println("grande present?", present)  
}
```

Run

125

range

- range iterates over elements in a variety of data structures

```
func main() {  
    nums := []int{5, 13, 27, 19, -1}  
    sum := 0  
    for index, num := range nums {  
        sum += num  
        fmt.Println("Added element", index)  
    }  
    fmt.Println("sum of", nums, "is", sum)  
}
```

Run

- if we don't need the index...

```
func main() {  
    nums := []int{5, 13, 27, 19, -1}  
    sum := 0  
    for _, num := range nums {  
        sum += num  
    }  
    fmt.Println("sum of", nums, "is", sum)  
}
```

Run

range on a Map

- ...iterates over key/value pairs

```
func main() {  
    continents := map[string]string{  
        "United States": "North America",  
        "India":          "Asia",  
        "Namibia":        "Africa",  
        "Germany":       "Europe",  
        "Mexico":         "North America",  
        "Ireland":        "Europe",  
        "Iceland":        "Europe",  
    }  
    for country, continent := range continents {  
        fmt.Println(country, "is in", continent)  
    }  
}
```

Run

127

range on a Map (cont'd)

- ...or just over the keys

```
func main() {  
    continents := map[string]string{  
        "United States": "North America",  
        "India":          "Asia",  
        "Namibia":        "Africa",  
        "Germany":       "Europe",  
        "Mexico":         "North America",  
        "Ireland":        "Europe",  
        "Iceland":        "Europe",  
    }  
    for country := range continents {  
        fmt.Println(country)  
    }  
}
```

Run

128

range on a string...

- ...iterates over Unicode code points
- the first value is the starting byte index of the rune
- the second value is the rune itself

```
func main() {  
    for index, char := range "Golang" {  
        // print each letter as a rune (int32) and character  
        fmt.Printf("%d %d %c\n", index, char, char)  
    }  
}
```

Run

129

Exercise: Maps

- use a map to translate Roman numerals into their Arabic equivalents
- load the map with Roman numerals M (1000), D (500), C (100), L (50), X (10), V (5), I (1)
- read in a Roman numeral
- print Arabic equivalent
- try it with $MCLX = 1000 + 100 + 50 + 10 = 1160$
- complain if invalid Roman digits (e.g., 'Z') are found
- EXTRA: if you have time, deal with the case where a smaller number precedes a larger number, e.g., $XC = 100 - 10 = 90$, or $MCM = 1000 + (1000-100) = 1900$

130

Additional Exercise: Maps

- create four calculator functions: add, sub, mul, div
- each function should take 2 integers and return an integer result
- create a map which maps the strings representing the operators to the *functions*
- that is, "+" would be mapped to add(), "-" would be mapped to sub()...
- have your program read in lines like "2 + 4" and have it determine the result by parsing the line and then using the operator ("+" in this case) to find the appropriate function to invoke

131

sort Package

- methods are specific to builtin type
- sorting occurs in place

```
import "sort"

func main() {
    strs := []string{"pear", "apple", "lemon"}
    sort.Strings(strs)
    fmt.Println("sorted:", strs)
    // Sorting integers
    ints := []int{7, 2, -4}
    sort.Ints(ints)
    fmt.Println("sorted:", ints)
    // Check if a slice is already sorted
    s := sort.StringsAreSorted(strs)
    fmt.Println("Are strings sorted?", s)
}
```

Run

132

User-Defined Types

struct

- typed collections of fields
- useful for grouping data together (like structs in other languages)

```
type Mountain struct {
    name      string
    elevation int
}

func main() {
    k2 := Mountain{"K2", 8611} // create a new struct
    fmt.Println(k2)
    fmt.Printf("%+v\n", k2)
    fmt.Printf("%#v\n", k2)
}
```

Run

134

struct (cont'd)

- you can name the fields when initializing a struct
- you can leave out fields, which will be set to zero
- & operator yields a pointer to the struct
- use a dot to access the fields

```
func main() {  
    everest := Mountain{elevation: 8848, name: "Mt. Everest"}  
    fmt.Printf("%#v\n", everest)  
    fmt.Println(Mountain{name: "Flat"})  
    fmt.Println(&Mountain{name: "Long's Peak", elevation: 4346})  
    nd := Mountain{name: "Nanda Devi", elevation: 7815}  
    fmt.Println(nd.name)  
    sp := &nd  
    fmt.Println(sp.elevation)  
    sp.elevation++  
    fmt.Println(sp.elevation)  
}
```

Run

135

Anonymous struct

- why an anonymous struct?
- e.g., grouped globals

```
var config struct {
    APIKey string
    IPAddr string
}
config.APIKey = "BADC0C0A"
config.IPAddr = "1.1.1.1"
```

- test tables

```
var indexRuneTests = []struct {
    s    string
    rune rune
    out int
}{
    {"a A x", 'A', 2},
    {"some_text=some_value", '=', 9},
    {"finster", 'E', -1},
}
```

Methods

- Go is not object oriented...or is it?
- Go does not have classes, but we can define methods on struct types
- types and methods allow for an OO style of programming
- Go does not support inheritance, but later we'll see the use of composition and interfaces
- suppose we define a type called triangle

```
type triangle struct {
    side1, side2, side3 int
}
```

- ...and we'd like to be able to perform certain operations on objects of type triangle

137

Methods (cont'd)

- here a method to compute the perimeter of a triangle...

```
// compute perimeter of triangle
func (t triangle) perimeter() int {
    return t.side1 + t.side2 + t.side3
}
```

- notice that the function (or method) above has a different syntax
- the above is a *method*, and the *method receiver* (the type that the method acts upon) appears in its own argument list between the `func` keyword and the method name
- here's a method to compute the area of a triangle using Heron's formula...

```
// compute area of triangle using Heron's formula
func (t triangle) area() int {
    s := (t.side1 + t.side2 + t.side3) / 2
    return int(math.Sqrt(float64(s * (s - t.side1) * (s - t.side2) * (s - t.side3))))
}
```

Methods (cont'd)

- to call a method on an object, we use a dotted notation common to object-oriented languages: *object.method(...)*

```
import "math"
type triangle struct {
    side1, side2, side3 int
}

// compute area of triangle using Heron's formula
func (t triangle) area() int {
    s := (t.side1 + t.side2 + t.side3) / 2
    return int(math.Sqrt(float64(s * (s - t.side1) * (s - t.side2) * (s - t.side3))))
}

// compute perimeter of triangle
func (t triangle) perimeter() int {
    return t.side1 + t.side2 + t.side3
}

func main() {
    t := triangle{3, 4, 5}
    fmt.Println("area: ", t.area(), "\nperim:", t.perimeter())
}
```

Run

Methods (cont'd)

- if we want to be able to modify the struct, our method will have to accept a pointer receiver

```
// scale the sides of a triangle
func (t *triangle) scale(factor int) {
    t.side1 *= factor
    t.side2 *= factor
    t.side3 *= factor
}
```

- we generally want all of our methods to receive either a pointer or a type, rather than having a mix of receiver types, but this will not be obvious until we know a bit more...

140

Methods (cont'd)

```
func (t *triangle) area() int {
    s := (t.side1 + t.side2 + t.side3) / 2
    return int(math.Sqrt(float64(s * (s - t.side1) * (s - t.side2) * (s - t.side3))))
}

func (t *triangle) perimeter() int {
    return t.side1 + t.side2 + t.side3
}

func (t *triangle) scale(factor int) {
    t.side1 *= factor
    t.side2 *= factor
    t.side3 *= factor
}

func main() {
    t := triangle{3, 4, 5}
    t.scale(3)
    fmt.Println("sides: ", t.side1, t.side2, t.side3)
}
```

Run

Defining Methods on Other Types

- you can define methods on any type in your package (not just struct)
- you can't define a method on a type from another package
- you can't define a method on a basic type, but you can create your own type via Go's type extensibility

```
import "strings"

type MyStr string // can't define a method on a basic type

func (s MyStr) Uppercase() string {
    return strings.ToUpper(string(s))
}

func main() {
    s := MyStr("Golang is fun!")
    fmt.Println(s.Uppercase())
}
```

Run

EDIT THE CODE so that the Uppercase method takes a parameter *suffix*, which is appended to the end of the string

142

Exercise: User-Defined Methods

- create a bank package which defines a type called a BankAccount
- at the very least, the BankAccount type should have an owner name (string) and a balance (float64)
- add methods Deposit, Withdraw, CheckBalance
- EXTRA: add a method to combine two accounts into a joint account

143

Embedding

- including ("embedding") a type as a nameless parameter within another type
- this makes the exported parameters and methods of the *embedded* type accessible through the *embedding* type
- compiler does this via promotion—the exported properties and methods of the embedded type are promoted to the embedding type

144

Embedding (cont'd)

- suppose in addition to our mountain type, we also want to represent a climb, in which a climber climbs a particular mountain...

```
type Climb struct {  
    Mountain  
    Climber string  
}
```

- the fields in Mountain (*name* and *elevation*) are promoted to be part of Climb 145

Embedding (cont'd)

- we can initialize a Climb in two ways, the first of which demonstrates the promotion of the fields from the embedded Mountain:

```
c1 := Climb{}
fmt.Println(c1)
c1.Name = "K2"
c1.Elevation = 8111 // i.e., missed the summit by 500 m
c1.Climber = "Arjun Climber"
fmt.Println(c1)
```

```
c2 := Climb{
    Mountain{"K2", 8111},
    "Arjun Climber",
}
fmt.Println(c2)
```

Run

- in addition to promotion of the fields, the methods of the embedded structs are also promoted...

146

Embedding (cont'd)

- let's make a silly method for Mountains called HowBig()

```
type Mountain struct {
    name      string
    elevation int
}

func (m Mountain) HowBig() string {
    if m.elevation > 8800 {
        return "HUGE"
    }
    return "NORMAL"
}

func main() {
    everest := Mountain{elevation: 8848, name: "Mt. Everest"}
    fmt.Println(everest.HowBig())
}
```

Run

147

Embedding (cont'd)

- now we'll demonstrate the object of type Climb can invoke the HowBig() method

```
type Climb struct {
    Mountain
    climber string
}

func (m Mountain) HowBig() string {
    if m.elevation > 8800 {
        return "HUGE"
    }
    return "NORMAL"
}

func main() {
    everest := Mountain{elevation: 8848, name: "Mt. Everest"}
    fmt.Println(everest.HowBig())
    arjun := Climb{Mountain{"K2", 8111}, "Arjun Climber"}
    fmt.Println(arjun.HowBig())
}
```

Run

148

Exercise: Embedding

- create a Customer type in your Bank package
- add a Customer method such as ChangeName
- embed a Customer inside a BankAccount object
- demonstrate that you can change the customer's name via the BankAccount object

149

Overloading

- suppose we want the *embedding* type to *override* a method in the *embedded* type
- that is, we want a method in the embedding type to have the same name as a method in the embedded type
- the method in Mountain will be "overridden" by the method in Climb

```
type Climb struct {
    Mountain
    Climber string
}

func (m Mountain) HowBig() string {
    if m.Elevation > 8800 {
        return "HUGE"
    }
    return "NORMAL"
}

func (c Climb) HowBig() string {
    return "OVERRIDE!"
}
```

150

Overloading (cont'd)

- ...but we can access the overriden method by referring to it explicitly

```
func (m Mountain) HowBig() string {
    if m.Elevation > 8800 {
        return "HUGE"
    }
    return "NORMAL"
}

// This method overrides the one above
func (c Climb) HowBig() string {
    return "OVERRIDE!"
}

func main() {
    c2 := Climb{
        Mountain{"K2", 8111}, "Arjun Climber",
    }
    // will call overriding method
    fmt.Println(c2.HowBig())
    // will call overridden method
    fmt.Println(c2.Mountain.HowBig())
}
```

Run

151

Encapsulation

- hiding implementation details when an interface to user-defined types is provided
- as a simple example, let's consider a singleton object (i.e., only one copy of the object exists at any given time)

```
package thing

// thing type is NOT exported
type thing struct {
    val int
}

// this variable is NOT exported
var privateThing thing

func ReadThing() int {
    return privateThing.val
}

func WriteThing(val int) {
    privateThing.val = val
}
```

Encapsulation (cont'd)

- via encapsulation, we can manipulate the singleton object, but can't see its implementation
- the package writer could change the implementation and as long as the same API was exposed, everything would still work

```
package main

import (
    "fmt"
    "thing"      // singleton, i.e., only one object
)

func main() {
    // try singleton "thing"
    fmt.Println(thing.ReadThing())
    thing.WriteThing(-5)
    fmt.Println(thing.ReadThing())
}
```

Interfaces

- an interface is a type, defined by a set of methods
- or, an interface is a named collection of method signatures
- to implement an interface, we implement all methods in the interface
- we don't need to tell the compiler we are implementing an interface—the compiler figures that out automatically

```
type MyInterface interface {
    Method() // for something to be of type MyInterface it must implement this method
}

type MyText string

func (text MyText) Method() { // this method means type MyText implements MyInterface
    fmt.Println(text)
}

func main() {
    var inter MyInterface = MyText("Hyderabad")
    inter.Method()
}
```

Run

154

Interfaces (cont'd)

```
type geometry interface {  
    area() float64  
    perim() float64  
}
```

here we implement geometry on rectangles:

```
type rect struct {  
    width, height float64  
}  
  
func (r rect) area() float64 {  
    return r.width * r.height  
}  
  
func (r rect) perim() float64 {  
    return 2*r.width + 2*r.height  
}
```

155

Interfaces (cont'd)

here we implement geometry on circles:

```
type circle struct {
    radius float64
}

func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}

func (c circle) perim() float64 {
    return 2 * math.Pi * c.radius
}
```

if a variable has an interface type, we can call methods in the interface

here's a generic measure function which works on any geometry

```
func measure(g geometry) {
    fmt.Println(g, g.area(), g.perim())
}
```

Interfaces (cont'd)

```
type geometry interface {
    area() float64
    perim() float64
}

func measure(g geometry) {
    fmt.Println(g)
    fmt.Println(g.area())
    fmt.Println(g.perim())
}

func main() {
    r := rect{width: 3, height: 4}
    c := circle{radius: 5}
    // The circle and rect struct types both implement
    // the geometry interface so we can use instances of
    // of these structs as arguments to measure()
    measure(r)
    measure(c)
}
```

Run

157

Interface Values

- under the hood, interface values can be thought of as a tuple of a value and a concrete type: (value, type)
- an interface value holds a value of a specific underlying concrete type

```
type MyInterface interface {
    Method()
}

type Text struct {
    String string
}

func (t Text) Method() {
    fmt.Println(t.String)
}

type MyFloat struct {
    val float64
}

func (f MyFloat) Method() {
    fmt.Println(f.val)
}
```

Interface Values (cont'd)

```
func describe(i MyInterface) {
    fmt.Printf("(%v, %T)\n", i, i)
}

func main() {
    var inter MyInterface
    inter = &Text{"Hello"} // pointer to Text object
    describe(inter)
    inter.Method()
    inter = MyFloat{math.Pi}
    describe(inter)
    inter.Method()
}
```

Run

159

Interface Values with nil Underlying Value

- if the value inside the interface is nil, the method will be called with a nil receiver
- in some languages this would yield a null pointer exception—it's common in Go to write methods that handle being called with a nil receiver

```
type MyInterface interface {
    Method()
}

type Text struct {
    String string
}

func (t *Text) Method() {
    if t == nil {
        fmt.Println("<nil>")
    } else {
        fmt.Println(t.String)
    }
}

func describe(i MyInterface) {
    fmt.Printf("(%v, %T)\n", i, i)
}
```

160

Interface Values with nil Underlying Value (cont'd)

```
func main() {
    var i MyInterface
    var t *Text

    i = t
    describe(i)
    i.Method()

    i = &Text{"hello"}
    describe(i)
    i.Method()
}
```

Run

161

nil Interface Value

- a nil interface value holds neither value nor type
- calling a method on a nil interface is a run-time error

```
type MyInterface interface {
    Method()
}

func describe(i MyInterface) {
    fmt.Printf("(%v, %T)\n", i, i)
}

func main() {
    var i MyInterface
    describe(i)
    i.Method()
}
```

Run

162

The Empty Interface

- the interface type that specifies zero methods is known as the empty interface:
`interface{}`
- an empty interface may hold values of any type, since every type implements at least zero methods
- use case-code that handles values of unknown type (e.g., `fmt.Println` takes any number of arguments of type `interface{}`)

```
func describe(i interface{}) {
    fmt.Printf("(%v, %T)\n", i, i)
}

func main() {
    var i interface{}
    describe(i)
    i = 42
    describe(i)
    i = "hello"
    describe(i)
}
```

Run

163

Type Assertions

- a type assertion provides access to an interface value's underlying concrete value

```
t := i.(T)
```

- the above statement asserts that the interface value *i* holds the concrete type *T* and assigns the underlying *T* value to the variable *t*
- if *i* does not hold a *T*, the statement will trigger a panic
- to test whether an interface value holds a specific type, a type assertion can return two values—the underlying value and a boolean that reports whether the assertion succeeded

```
t, ok := i.(T)
```

- if *i* holds a *T*, then *t* will be the underlying value and *ok* will be true
- otherwise, *ok* will be false and *t* will be the zero value of type *T*, and no panic occurs
- NOTE: this syntax is similar to that of reading from a map

Type Assertions (cont'd)

```
package main

import "fmt"

func main() {
    var i interface{} = "hello"
    s := i.(string)
    fmt.Println(s)
    s, ok := i.(string)
    fmt.Println(s, ok)
    f, ok := i.(float64)
    fmt.Println(f, ok)
    f = i.(float64) // panic
    fmt.Println(f)
}
```

Run

165

Type switch

- a switch construct that permits several type assertions in series
- like a regular switch statement, but the cases in a type switch specify types (not values), and those values are compared against the type of the value held by the given interface value

```
func check_type(i interface{}) {  
    switch v := i.(type) {  
    case int:  
        fmt.Printf("Twice %v is %v\n", v, v*2)  
    case string:  
        fmt.Printf("%q is %v bytes long\n", v, len(v))  
    default:  
        fmt.Printf("I don't know about type %T!\n", v)  
    }  
}  
  
func main() {  
    check_type(21)  
    check_type("hello")  
    check_type(true)  
}
```

Run

166

Stringers

- one of the most ubiquitous interfaces is Stringer

```
type Stringer interface {
    String() string
}
```

- a type that can describe itself as a string
- the fmt package (and many others) look for this interface to print values

```
type Person struct {
    Name, AlmaMater string
}

func (p Person) String() string {
    return fmt.Sprintf("%v (Alma Mater: %v)", p.Name, p.AlmaMater)
}

func main() {
    rs := Person{"Ranveer Singh", "Indiana University"}
    ab := Person{"Amitabh Bachchan", "Sherwood College"}
    fmt.Printf("%s\n%s\n", rs, ab)
}
```

Run

Exercise: Stringers

1. Add a Stringer method to your Bank object so that a BankAccount can be printed out in a nice human-readable way
2. Given the following type:

```
type IPAddr [4]byte
```

- Define a map which maps hostnames (string) to IPAddr
- Add a few hostnames to your map, e.g., "localhost.com" (the addresses don't have to be correct)
- Add a Stringer method to IPAddr so that it is printed out in dotted notation (e.g., "127.0.0.1")
- Test your code in the playground or locally

168

Error Handling

Error Handling

- Go does not have exceptions, as we're used to in Python, Java, etc.
- it's idiomatic to communicate errors via a separate return value
- the ability to return multiple values is an improvement over C, where programmers often overload a single result value to indicate error
- error detection and handling in Go is easy since we can use the same language constructs we're used to
- by convention, errors are the last return value and are of type error:

```
type error interface {  
    Error() string  
}
```

- ...in other words, any type that has an `Error()` method that returns a string
- let's see an example using Go's built-in errors package

170

Error Handling (cont'd)

- an error of `nil` means no error
- `errors.New()` returns a basic error which your functions can return

```
import "errors"

var FilenameErr = errors.New("Invalid filename!")
var NotFoundErr = errors.New("File not found!")

...
if something_bad_happened {
    return 0, NotFoundErr
}
```

171

Error Handling (cont'd)

- in addition, you can use `fmt.Error` to do `Printf`-like formatting and returns the result as an error created by `errors.New`

```
if f < 0 {  
    return 0, fmt.Errorf("math: square root of negative number %g", f)  
}
```

172

Error Handling (cont'd)

```
import "errors"

var CountErr = errors.New("Invalid count")
var EmptyErr = errors.New("Can't replicate empty string!")

// Like Python * for strings, i.e., string replication
func stringstar(str string, count int) (string, error) {
    result := ""
    if count < 1 {
        return "", CountErr
    }
    if str == "" {
        return "", EmptyErr
    }
    for i := 1; i <= count; i++ {
        result += str
    }
    return result, nil
}

func main() {
    res, err := stringstar("Golang", 4)
    fmt.Println("result:", res, "\nerror:", err)
}
```

Run

Error Handling (cont'd)

- we can create our own error type, perhaps adding a timestamp, as long as we implement the error interface

```
type MyError struct {
    When time.Time
    What string
}

func (e MyError) Error() string {
    return fmt.Sprintf("at %v, %s", e.When, e.What)
}

func run() error {
    return MyError{
        time.Now(),
        "it didn't work",
    }
}

func main() {
    if err := run(); err != nil {
        fmt.Println(err)
    }
}
```

Run

Error Handling (cont'd)

- if you need to access the data in a custom error, you need to use type assertion in order to get the error as an instance of your custom error:

```
type MyError struct {
    Num int
    Msg string
}

func (e MyError) Error() string {
    return fmt.Sprintf("%s (%d)", e.Msg, e.Num)
}

func do_something() (int, error) {
    return 0, MyError{42, "failure message"}
}

func main() {
    _, err := do_something()
    if me, ok := err.(MyError); ok { // assert err is of type MyError
        fmt.Println("Num =", me.Num)
        fmt.Println("What =", me.Msg)
    }
}
```

Run

Exercise : Error Handling

- modify your square root (or Withdraw) function from earlier to return a custom error as well as the existing return value
- test it with a main program that sends a valid argument as well as an invalid argument
- access the fields of the custom error programmatically using type assertion

176

Error Handling: Best Practices

always check for errors, not just when you expect them

whenever possible, error strings should identify their origin, e.g., a prefix naming the operation or package that generated the error

- e.g., "<package_name>: unknown format"

pre-define errors when possible

know when to panic:

- when the user explicitly says so
- during initialization (maybe, if can't recover)

Reading/Writing Files

Reading Files

- the interfaces `io.Reader` and `io.Writer` are central to input/output in Go

```
type Reader interface {
    Read(p []byte) (n int, err error)
}

type Writer interface {
    Write(p []byte) (n int, err error)
}
```

179

Reading Files

- here's an example using the os package to read from /etc/passwd

```
func check(e error) { // helper function for error checking
    if e != nil {
        panic(e)
    }
}

func main() {
    buf := make([]byte, 1024) // create a buffer
    f, e := os.Open("/etc/passwd")
    check(e)
    defer f.Close() // idiomatic to defer a Close just after Open
    for {
        n, e := f.Read(buf)
        if n == 0 { // 0 means we've hit EOF
            break
        }
        check(e)
        os.Stdout.Write(buf[:n])
    }
}
```

Run

180

Writing Files

```
func main() {
    // Dump a string (or just bytes) into a file
    d1 := []byte("hello\ngo\n")
    err := ioutil.WriteFile("/tmp/dat1", d1, 0644)
    check(err)
    // For more granular writes, open a file for writing
    f, err := os.Create("/tmp/dat2")
    check(err)
    defer f.Close()
    d2 := []byte{115, 111, 109, 101, 10} // "some"
    n2, err := f.Write(d2)           // you can Write byte slices as you'd expect
    check(err)
    fmt.Printf("wrote %d bytes\n", n2)
    n3, err := f.WriteString("writes\n") // write a string
    fmt.Printf("wrote %d bytes\n", n3)
    f.Sync() // flushes writes to stable storage
}
```

Run

181

More Reading Files

```
func main() {
// Slurping a file's entire contents into memory
    dat, err := ioutil.ReadFile("/tmp/dat1")
    check(err)
    fmt.Println(string(dat))
    // or...
    f, err := os.Open("/tmp/dat1")
    check(err)
    defer f.Close()
    // Read up to 5 bytes from beginning of file.
    // Note how many actually were read.
    b1 := make([]byte, 5)
    n1, err := f.Read(b1)
    check(err)
    fmt.Printf("%d bytes: %s\n", n1, string(b1))
    o2, err := f.Seek(6, 0) // seek to known location in file and read from there
    check(err)
    b2 := make([]byte, 2)
    n2, err := f.Read(b2)
    check(err)
    fmt.Printf("%d bytes @ %d: %s\n", n2, o2, string(b2))
```

More Reading Files (cont'd)

```
// io package provides helpful functions for file reading. Reads
// like the ones above can be more robustly implemented with ReadAtLeast.
o3, err := f.Seek(6, 0)
check(err)
b3 := make([]byte, 2)
n3, err := io.ReadAtLeast(f, b3, 2)
check(err)
fmt.Printf("%d bytes @ %d: %s\n", n3, o3, string(b3))
_, err = f.Seek(0, 0) // no built-in rewind, but Seek(0, 0) does the trick
check(err)
// bufio package implements a buffered reader-useful both for efficiency
// with many small reads and additional reading methods it provides
r4 := bufio.NewReader(f)
b4, err := r4.Peek(5)
check(err)
fmt.Printf("5 bytes: %s\n", string(b4))
}
```

Run

183

Reading a Text File Line by Line

- easiest to use a Scanner in the `bufio` package
- substitute `os.Stdin` to read from standard input

```
import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    f, err := os.Open("poem.txt")
    if err != nil {
        panic(err)
    }
    scanner := bufio.NewScanner(f)
    for scanner.Scan() {
        fmt.Println(scanner.Text())
    }
    if err := scanner.Err(); err != nil {
        fmt.Fprintln(os.Stderr, "reading standard input:", err)
    }
}
```

Run

Exercise: Reading from Files

- write a Go program to read a file and count the number of occurrences of each word in the file
- use a map indexed by word, to count the occurrences
- treat **The** and **the** as the same word when counting
- NOTE: the strings package has several functions that will be helpful
- NOTE 2: ideally we would sort by counts, from most common to least common, but this is much more difficult to do in Go than it is in Python
- EXTRA: remove punctuation, so **Hamlet, == Hamlet**

[Test on Shakespeare's Hamlet](http://bit.ly/BillShak) (<http://bit.ly/BillShak>)

185

Concurrency

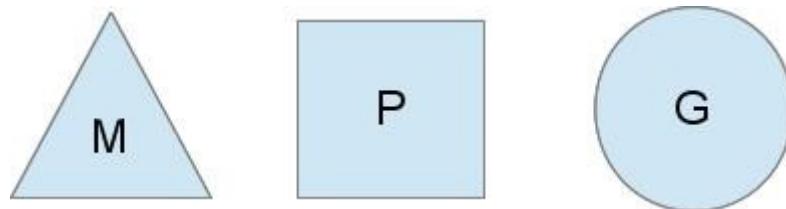
What is Concurrency?

- Hint: it's not parallelism
- concurrency = the composition of independently executing things (e.g., functions)
- parallelism = simultaneous execution of multiple things (which may or may not be related)
- concurrency is about dealing with many things at once (i.e., it's about structure)
- parallelism is about doing many things at once (i.e., it's about execution)
- consider an OS which manages a number of I/O devices: mouse, keyboard, video, etc.
- it's concurrent, but if only one CPU, clearly not parallel

187

Go Scheduler Overview

- a goroutine (G) is like a lightweight thread which we will examine shortly
- in order for an M (operating system thread) to execute a G it must acquire a (logical) processor (P), also known as a context

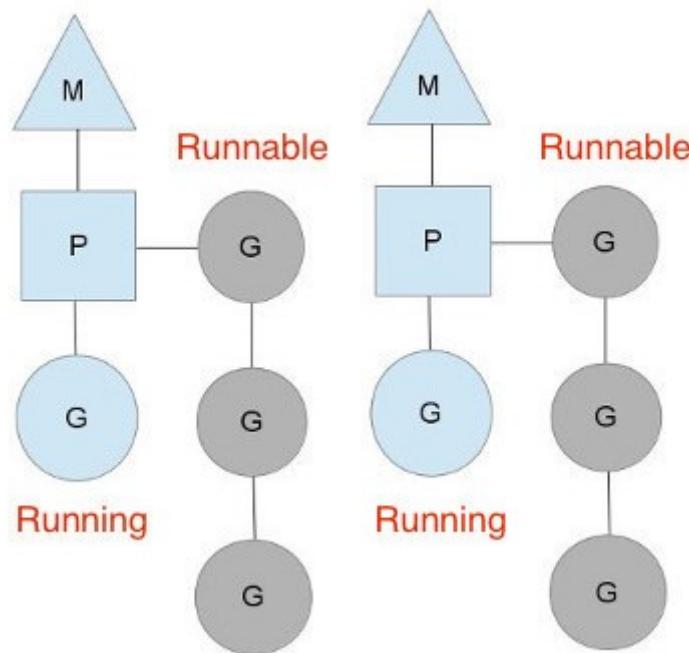


morsmachine.dk/go-scheduler (<http://morsmachine.dk/go-scheduler>)

188

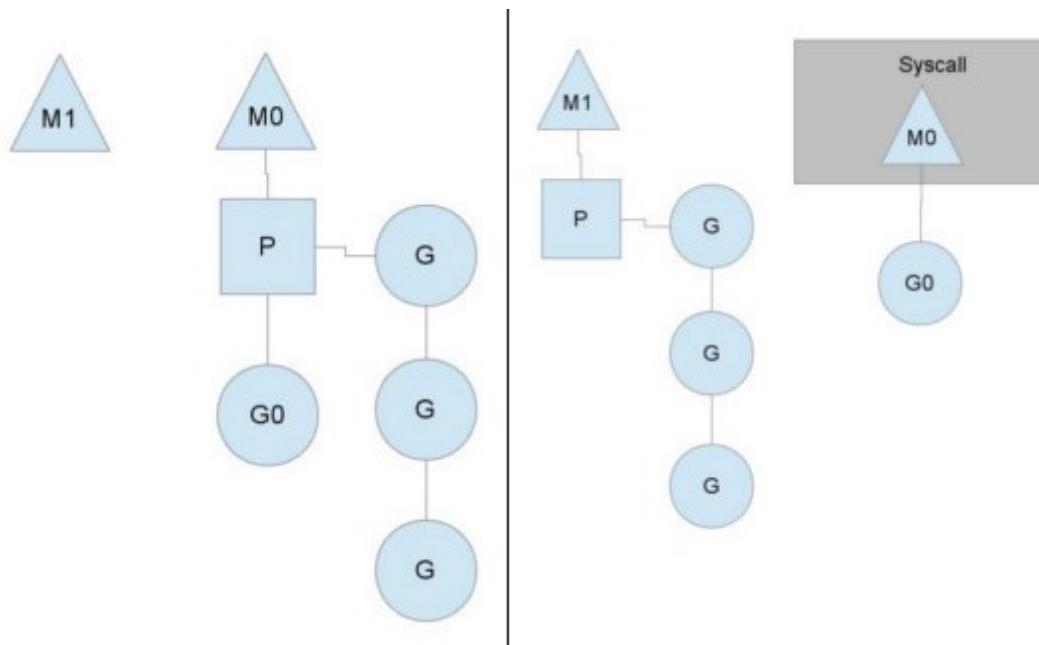
Go Scheduler Overview (cont'd)

M will run G until it stops:



- making a system call (e.g., I/O)
- blocking on a channel operation (discussed shortly)
- pre-empted (which only happens at a safe point—when it makes a function call)⁸⁹

Go Scheduler Overview (cont'd)



if G blocks on a system call, it releases the P but remains running on the M

- a system monitor thread wakes up every so often
- if it sees runnable G and available P it will wake a sleeping M (or start a new one)- that M will acquire a P and run a runnable G

when a G completes a system call it will either reacquire the P or be marked runnable and go to sleep

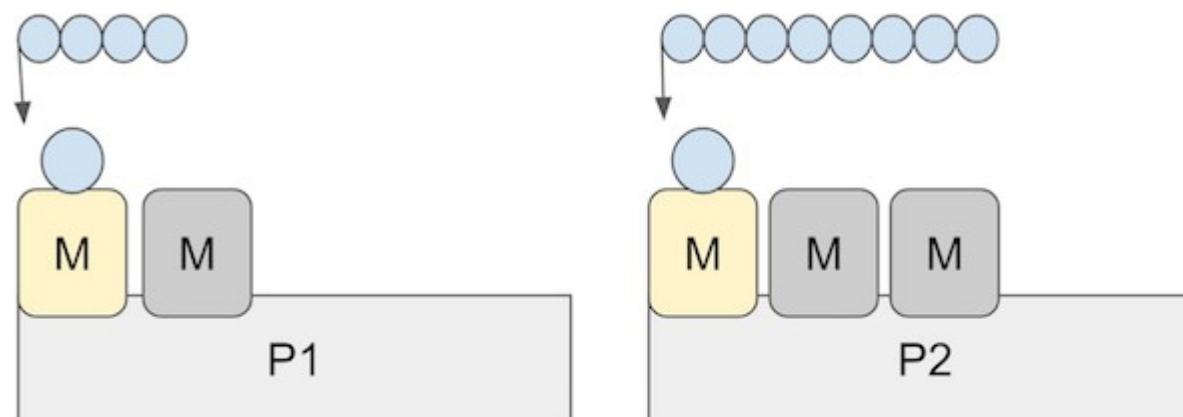
190

Go Scheduler Overview (cont'd)

if G blocks on a channel operation, it gets placed in a queue

- M looks for another runnable G
- if no runnable G, M will release P and go to sleep

when a channel operation succeeds, it will wake the other goroutine, mark it runnable, and, if there is an available P wake up a M to run it



191

Concurrency: Go vs. Others

concurrent programming is often difficult due to the subtleties required to correctly access shared data

Go takes a different approach—shared data are passed back and forth on channels

- they are not, in fact, actively shared by separate threads
- only one goroutine has access to the data at any given time
- data races cannot occur, by design

to encourage this way of thinking Golang has reduced it to a slogan:

192

**Do not communicate by sharing memory;
instead, share memory by communicating!**

Concurrency: Go vs. Others (cont'd)

- of course, this approach can be taken too far...
- e.g., reference counts may be best implemented by putting a mutex (essentially, a "lock") around an integer

Go includes low-level concurrency primitives as well, but you often need not concern yourself with low-level details

194

Goroutines

- a lightweight thread of execution (not an actual OS thread)
- invoked by keyword go preceding a function call (or anonymous function)

```
package main

import (
    "fmt"
    "time"
)
func thrice(from string) {
    for i := 1; i <= 3; i++ {
        time.Sleep(10 * time.Millisecond)
        fmt.Println(from, i)
    }
}
func main() {
    thrice("direct") // no concurrency at this point
    go thrice("goroutine") // invoke the function in a goroutine
    go thrice("hey!") // and another...concurrent with main program
    time.Sleep(1000 * time.Millisecond) // sleep 1 second
}
```

Run

195

Gouroutines (cont'd)

- another example...run directly, not in playground

```
package main

import "fmt"

var done bool = false

func ask() {
    var input string
    fmt.Print("Enter some text: ")
    fmt.Scanln(&input)
    done = true
}

func main() {
    var square, count uint64 = 0, 0
    go ask()
    for ; !done; count++ {
        square = count * count
    }
    fmt.Printf("While waiting, computed %d * %d = %d\n", count, count, square)
}
```

Channels

- typed "conduits" through which you can send and receive data using the channel operator <-
- data moves in the direction of the arrow
- create a new channel using built-in function make()
- sending and receiving are blocking operations

```
func pinger(c chan string) {  
    fmt.Println("pinger about to send")  
    c <- "ping...ping...ping" // send a value into a channel ... blocking operation  
}  
  
func main() {  
    mychan := make(chan string) // Create a new channel  
    go pinger(mychan) // run pinger() as a goroutine  
    fmt.Println("goroutine started...waiting 3 seconds before receiving")  
    time.Sleep(3 * time.Second)  
    msg := <-mychan // receive value from channel ... this is a blocking operation  
    fmt.Println(msg)  
}
```

Run

Exercise: Relay Race

- write a program to simulate a relay race with 4 runners
- create a func called runner with an int channel as a parameter
- the function should receive the "baton" via that channel (in reality it will receive a runner number, from 1-4)
- once the baton is received, it will print a message saying "Runner x is running" and then it will fire off the next goroutine representing the next runner in line
- when runner 4 receives the baton, it is the last runner, so it should not fire off another goroutine

198

Buffered Channels

- channels are unbuffered by default, i.e., they only accept a send operation if there is a corresponding receive operation ready to receive the data
- buffered channels receive and hold send ops without a corresponding receiver

```
import "fmt"
import "time"

func main() {
    // make a buffered channel of up to 4 strings
    bchan := make(chan string, 4)
    // We can send 4 values without a corresponding receive
    bchan <- "1"    // will not block
    bchan <- "2"    // will not block
    bchan <- "3"    // will not block
    bchan <- "go!"  // will not block
    fmt.Println("4 values sent...")
    time.Sleep(1 * time.Second)
    for i := 1; i <= 4; i++ { // Later we can receive these values as usual
        fmt.Println(<-bchan)
    }
}
```

Run

199

Buffered Channels (cont'd)

- but if you try to put too much in the buffered channel...

```
package main

import "fmt"

func main() {
    // make a buffered channel of up to 2 strings
    bchan := make(chan string, 2)

    bchan <- "1"
    bchan <- "2"
    bchan <- "3"

    // Later we can receive these values as usual
    for i := 1; i <= 3; i++ {
        fmt.Println(<-bchan)
    }
}
```

Run

200

Buffered Channels (cont'd)

- but if we use a goroutine, it will work...

```
package main

import "fmt"

func main() {
    // make a buffered channel of up to 2 strings
    bchan := make(chan string, 2)

    bchan <- "1"
    bchan <- "2"
    go func() {
        bchan <- "3"
    }()
}

// Later we can receive these values as usual
for i := 1; i <= 3; i++ {
    fmt.Println(<-bchan)
}
}
```

Run

201

Channel Synchronization

- channels can be used to synchronize execution across goroutines
- we will use a blocking receive to wait for our goroutine to finish
- (here's a case where the parameter name is the same as the variable name, on purpose)

```
func worker(done chan bool) {  
    for i := 1; i <= 5; i++ {  
        fmt.Println("working...")  
        time.Sleep(600 * time.Millisecond)  
    }  
    done <- true // we are done!  
}  
func main() {  
    done := make(chan bool)  
    go worker(done)  
    // block until we receive a notification  
    <-done  
    fmt.Println("done!")  
}
```

Run

202

Channel Directions

- when using a channel as a parameter to a function, you can specify whether the channel is meant to send or receive

```
// It would be a compile-time error to try to receive on this channel
func ping(schan chan<- string, msg string) {
    fmt.Printf("ping sending %#v\n", msg)
    schan <- msg
}

// Accepts one channel for receives and a second for sends
func pong(rchan <-chan string, schan chan<- string) {
    msg := <-rchan // receive a message
    fmt.Printf("pong received %#v\n", msg)
    schan <- msg
    fmt.Println("and sent it...")
}

func main() {
    pings := make(chan string, 1)
    pongs := make(chan string, 1)
    ping(pings, "secret message")
    pong(pings, pongs)
    fmt.Printf("main received %#v\n", <-pongs)
}
```

Run

The select Statement

- lets a goroutine wait on multiple communication operations
- blocks until one of its cases can run, then it executes that case
- if more than one are ready it chooses one at random

```
c1, c2 := make(chan string), make(chan string)
go func() { // wait 3 seconds, then send
    time.Sleep(3 * time.Second)
    c1 <- "one"
}()
go func() { // wait 1 seconds, then send
    time.Sleep(1 * time.Second)
    c2 <- "two"
}()
for i := 0; i < 2; i++ { // await both of these values simultaneously
    select {
        case msg1 := <-c1:
            fmt.Println("received", msg1)
        case msg2 := <-c2:
            fmt.Println("received", msg2)
    }
}
```

Run

Timeouts

we need to be able to time out if

- we're connecting to external resources
- we need to bound execution time
- channels plus the select statement let us time out elegantly

```
func main() {  
    c1 := make(chan string, 1)  
  
    go func() { // sleep 2 seconds, then send  
        time.Sleep(2 * time.Second)  
        c1 <- "Secret message"  
    }()  
  
    select { // wait up to 1 second to get message on c1  
    case res := <-c1:  
        fmt.Println(res)  
    // time.After waits for duration to elapse then sends current time on the returned channel  
    case <-time.After(1 * time.Second):  
        fmt.Println("Timed out; Message not received.")  
    }  
}
```

Run

Non-Blocking Channel Operations

- sends and receives on channels are blocking
- ...but, we can use `select` with `default` to implement non-blocking send/receive

```
messages := make(chan string)
select { // non-blocking send
    case messages <- "hi":
        fmt.Println("sent message")
    default:
        fmt.Println("no message sent")
}

go func() {
    messages <- "Here's a message"
}()
time.Sleep(1 * time.Second)

select { // non-blocking receive
    case msg := <-messages:
        fmt.Println("received message:", msg)
    default:
        fmt.Println("no message received")
}
```

Run

Non-Blocking Channel Operations (cont'd)

- here is a non-blocking multi-way select

```
package main

import "fmt"

func main() {
    messages := make(chan string)
    signals := make(chan bool)
    // Multi-way non-blocking select
    select {
        case msg := <-messages:
            fmt.Println("received message", msg)
        case sig := <-signals:
            fmt.Println("received signal", sig)
        default:
            fmt.Println("no activity")
    }
}
```

Run

207

Closing Channels

```
jobs := make(chan int, 5) // holds work to be done
done := make(chan bool)

go func() {
    for {
        j, more := <-jobs // more will be false if channel closed
        if more {
            fmt.Println("running job", j)
            time.Sleep(time.Second)
        } else {
            fmt.Println("no more work to do")
            done <- true // signal main program
            return
        }
    }
}()

for j := 1; j <= 3; j++ { // sends 3 jobs to the worker over the jobs channel, then close
    jobs <- j
    fmt.Println("sent job", j)
}

close(jobs)
fmt.Println("sent all jobs")

<-done // await worker
```

Run

Range on a Channel

- receive values repeatedly from a channel until it is closed
- we can close a channel with data still in it, and have the data received afterwards

```
// Compute Fibonacci sequence up to n and send results to the channel
func fibonacci(n int, c chan int) {
    x, y := 1, 1
    for i := 0; i < n; i++ {
        c <- x
        x, y = y, x+y
    }
    close(c) // close channel to signal all done
}

func main() {
    c := make(chan int, 10) // buffered channel, 10 slots
    go fibonacci(cap(c), c) // run fibonacci up to capacity of channel
    for i := range c {
        fmt.Println(i)
    }
}
```

Run

209

Exercise: Select/Concurrency

- modify the previous Fibonacci example to use an unbuffered channel, rather than a buffered channel
- add a second unbuffered channel called quit, which is used by the fibonacci routine to know there is no more work to do (in other words, fibonacci should listen for a message on the quit channel, and if it receives the message, it should stop generating Fibonacci numbers)
- the above means that the fibonacci function will no longer have a loop from 1 to 10

210

Signal Handling

- sometimes we'd like our programs to intelligently handle Linux signals
- e.g., want a server to gracefully shut down when it receives a SIGTERM
- doesn't work in playground, run directly!

```
import "os"
import "os/signal"
import "syscall"

func main() {
    sigs := make(chan os.Signal, 1) // channel to receive os.Signal notifications
    done := make(chan bool, 1) // channel to notify us when the program can exit
    // Register the given channel to receive signal notifications
    signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM)
    go func() { // Execute a blocking receive for signals
        sig := <-sigs
        fmt.Println("\n", sig)
        done <- true
    }()
    fmt.Println("awaiting signal") // Wait here until goroutine is done
    <-done
    fmt.Println("exiting")
}
```

Run

Waiting for Goroutines

- a `sync.WaitGroup` waits for a collection of goroutines to finish

```
var wg sync.WaitGroup

func process(thing string) {
    defer wg.Done() // decrement counter when goroutine completes
    time.Sleep(time.Duration(len(thing) * 100) * time.Millisecond)
    fmt.Println(thing, "processed")
}

func main() {
    var items = []string{
        "short",
        "longer",
        "even longer",
        "supercalifragilisticexpialidocious",
    }
    for _, item := range items {
        wg.Add(1) // increment the WaitGroup counter
        go process(item)
    }
    fmt.Println("main waiting...")
    wg.Wait() // wait for all processing to complete
}
```

Run

Exercise: Waiting for Goroutines

- create 25,000 goroutines, each of which sleeps a random number of seconds before completing
- after sleeping, the go routine should print a message like *Goroutine n done* where n is the number of the goroutine
- when all 25,000 are done, the main program should notify you that all goroutines have finished

213

More Interesting Example

- go get a bunch of URLs
- spin up a goroutine for each one

```
func main() {  
    var wg sync.WaitGroup  
  
    var urls = []string{  
        "http://www.golang.org/", "http://www.google.com/", "http://www.zzz1983x.com/",  
    }  
    for _, url := range urls {  
        wg.Add(1) // increment the WaitGroup counter  
        go func(url string) { // launch goroutine to fetch URL  
            defer wg.Done() // decrement counter when goroutine completes  
            if resp, err := http.Get(url); err == nil {  
                fmt.Println(url, resp.Status)  
            } else {  
                fmt.Println(url, err)  
            }  
        }(url)  
    }  
    wg.Wait() // Wait for all HTTP fetches to complete  
}
```

Run

Mutual Exclusion (sync.Mutex)

- what if we just want to make sure only one goroutine can access a variable at a time to avoid conflicts?
- this concept is called mutual exclusion and the data structure that provides it is typically called a mutex
- Go provides sync.Mutex and its two methods, Lock and Unlock
- if a sync.Mutex object is already locked when a goroutine tried to lock it, the goroutine will block until the holder of the lock unlocks it

```
import "sync"
var state = make(map[int]int) // For our example the `state` will be a map
var mutex sync.Mutex          // This `mutex` will synchronize access to `state`
var readOps uint64            // We'll keep track of how many read...
var writeOps uint64           // ...and write operations we do
```

Mutual Exclusion (cont'd)

```
import "sync/atomic" // also "math/rand" and "time"

func read_state() {
    total := 0
    for {
        key := rand.Intn(5)          // pick a random map entry
        mutex.Lock()                // lock the shared state
        total += state[key]         // read it!
        mutex.Unlock()              // unlock
        atomic.AddUint64(&readOps, 1) // atomically increment count
        time.Sleep(time.Millisecond)
    }
}

func write_state() {
    for {
        key := rand.Intn(5)          // pick a random map position
        val := rand.Intn(100)         // pick a random number to write
        mutex.Lock()                // lock the shared state
        state[key] = val            // write it!
        mutex.Unlock()              // unlock
        atomic.AddUint64(&writeOps, 1) // atomically increment count
        time.Sleep(time.Millisecond)
    }
}
```

Mutual Exclusion (cont'd)

```
func main() {
    for r := 0; r < 100; r++ { // Start 100 goroutines to execute repeated reads
        go read_state()
    }
    for w := 0; w < 10; w++ { // Start 10 goroutines to simulate writes
        go write_state()
    }
    time.Sleep(time.Second)
    readOpsFinal := atomic.LoadUint64(&readOps) // Report final operation counts
    fmt.Println("readOps:", readOpsFinal)
    writeOpsFinal := atomic.LoadUint64(&writeOps)
    fmt.Println("writeOps:", writeOpsFinal)
    mutex.Lock()
    fmt.Println("state:", state)
    mutex.Unlock()
}
```

Run

217

Race Detection

- a data race occurs when two goroutines access the same variable concurrently and at least one of the accesses is a write
- among the most common/most difficult to debug issues in concurrent systems

```
package main

import "fmt"

func main() {
    c := make(chan bool)
    m := make(map[string]string)
    go func() {
        m["1"] = "a" // First conflicting access.
        c <- true
    }()
    m["2"] = "b" // Second conflicting access.
    <-c
    for k, v := range m {
        fmt.Println(k, v)
    }
}
```

Run

Race Detection (cont'd)

- The Go race detector can detect data races while your program is running, but the executable must be instrumented with special code to perform the detection
- To add race detection to your binary:

```
go build -race mycmd
```

and then run it

- You can, as you'd expect do this all in one step

```
go run -race mysrc.go
```

- And you can use `go install` to add race detection to a package

```
go install -race mypkg
```

219

Exercise

- create a program which uses a goroutine to simulate a database server
- your "database server" should simply be a goroutine which accepts an integer representing some "work" to do and it sleeps for that amount of time before accepting more work
- your main routine should get integers from the user, and pass those to the "server"
- once you have that working, make it more complex...
- add a second server goroutine, and use a select statement to send the work to whichever server is able to accept it (to test this, send a big chunk of work, i.e., a large int, to one of them, and then be sure the other can do subsequent work in the meantime)

220

Exercise (cont'd)

- still more...how about a load balancer goroutine which spins up new server routines as needed up to some maximum (i.e., if it sees that no servers are available, it spins up more)
- to do this effectively you'll likely need reflection, so let's simplify by just trying to send to each server, and if they are all busy, launch a new goroutine
- you'll likely need an array/slice of channels
- make it so each server goroutine shuts down if it hasn't done work for a while, so even if you hit your max of, say 10, servers, they should start shutting down as the workload decreases

221

Exercise (cont'd)

- although we normally would share memory by communicating...
- add a shared queue, which can simply a slice of ints that you add to by appending, and remove from by slicing the queue, i.e.,

```
// remove an item from the queue
item = queue[0]
queue = queue[1:]
```

- wrap the queue in a Mutex to prevent simultaneous accesses
- the queue holds work to be performed and is filled by the main routine and drained by the load balancer, or remove the load balancer and let the workers remove something from the queue as they are free
- also let workers randomly reject work and place it back in the wqqueue...

222

Testing

Testing

- Go comes with great unit testing capabilities built in
- let's build a small package and see how to test it

```
package stack

// A simple stack object with 10 slots for integers
type Stack struct {
    ptr  int
    items [10]int
}
// Push an item onto the stack
func (s *Stack) Push(i int) {
    s.items[s.ptr] = i
    s.ptr++
}
// Pop an item from the stack
func (s *Stack) Pop() int {
    s.ptr--
    return s.items[s.ptr]
}
```

224

Testing (cont'd)

- let's create test file(s) named `*_test.go` in the same folder as our code
- first we have to import the testing package

golang.org/pkg/testing (<https://golang.org/pkg/testing>)

- then create one or more functions named `Test*` which accept a `*testing.T` (that object manages state for the test)

```
package stack

import "testing"

func TestPushPop(t *testing.T) {
    s := Stack{} // cf. s := new(Stack)
    s.Push(1)
    if s.Pop() != 1 {
        t.Log("Pop() failed to return 1")
        t.Fail()
    }
}
```

Testing (cont'd)

```
$ go test -v
==== RUN    TestPushPop
--- PASS: TestPushPop (0.00s)
PASS
ok      stack    0.006s
```

226

Exercise: Testing

add additional methods to the Stack module:

- Top: returns the top element but does not remove it
- IsEmpty: return a Boolean indicating whether stack is empty
- Size: returns number of elements on stack (could be used by IsEmpty)
- add tests for each of these methods

add tests to your bank package

- try to Deposit/Withdraw a negative amount
- try to Withdraw more than your balance
- try to create a bankAccount with no name
- anything else you can think of...

227

"Example" Tests

- like unit tests, but instead of verifying results programmatically, it verifies output
- the correct output lives in comments after the commands

```
func ExampleSalutations() {  
    fmt.Println("hello, and")  
    fmt.Println("goodbye")  
    // Output:  
    // hello, and  
    // goodbye  
}
```

- output from running the examples

```
$ go test -v  
=== RUN   ExampleSalutations  
--- PASS: ExampleSalutations (0.00s)  
PASS  
ok      example    0.005s
```

228

"Example" Tests (cont'd)

- the comment prefix "Unordered output:" matches any line order:

```
package perm_test

import "math/rand"

func ExamplePerm() {
    for _, value := range rand.Perm(4) {
        fmt.Println(value)
    }
    // Unordered output:
    // 2
    // 1
    // 3
    // 0
}
```

Exercise: Example Tests

- write a Golang program that takes a string message such as "Hello from Intuit", splits it into words, and then enters the words as keys in a map (the values are up to you)
- iterate through the map and output the keys, which will be in a random order
- write an "example" test to ensure the output is correct

230

Test Coverage

- test coverage refers to the percentage of statements in your code that are executed by your tests
- ideally, we want 100% coverage ("untested code is broken code")

```
package size

func Size(a int) string {
    switch {
    case a < 0:
        return "negative"
    case a == 0:
        return "zero"
    case a < 10:
        return "small"
    case a < 100:
        return "big"
    case a < 1000:
        return "huge"
    }
    return "enormous"
}
```

Test Coverage (cont'd)

- here's a test-let's see how it does

```
type Test struct { // a "Test" consists of an input and an output
    in int
    out string
}
var tests = []Test{ // Clearly are missing some cases here
    {-1, "negative"},
    {5, "small"},
}
func TestSize(t *testing.T) {
    for i, test := range tests {
        size := Size(test.in)
        if size != test.out {
            t.Errorf("#%d: Size(%d)=%s; want %s", i, test.in, size, test.out)
        }
    }
}
```

232

Test Coverage (cont'd)

```
$ go test  
PASS  
ok      size   0.08s
```

- the tests pass, but let's look at coverage

```
$ go test -cover  
PASS  
coverage: 42.9% of statements  
ok      size   0.008s
```

- coverage is quite poor—it's fairly obvious why in this toy example

233

Test Coverage (cont'd)

- go test runs the cover tool which rewrites the source code like this

```
func Size(a int) string {
    GoCover.Count[0] = 1
    switch {
    case a < 0:
        GoCover.Count[2] = 1 // each assignment statement is a single move instruction,
        return "negative"   // therefore FAST
    case a == 0:
        GoCover.Count[3] = 1
        return "zero"
    case a < 10:
        GoCover.Count[4] = 1
        return "small"
    case a < 100:
        GoCover.Count[5] = 1
        return "big"
    case a < 1000:
        GoCover.Count[6] = 1
        return "huge"
    }
    GoCover.Count[1] = 1    // 7 paths through the code, and only 3 of them were executed
    return "enormous"
}
```

Coverage Profile

```
$ go test -coverprofile=coverage.out
PASS
coverage: 42.9% of statements
ok      size   0.08s
```

- we can examine coverage per function (but not very interesting here since there's only one function)

```
$ go tool cover -func=coverage.out
size/size.go:3: Size          42.9%
total:           (statements) 42.9%
```

- let's see it graphically...

```
$ go tool cover -html=coverage.out
```

235

Coverage Profile (cont'd)

size/size.go (42.9%) not tracked not covered covered

```
package size

func Size(a int) string {
    switch {
    case a < 0:
        return "negative"
    case a == 0:
        return "zero"
    case a < 10:
        return "small"
    case a < 100:
        return "big"
    case a < 1000:
        return "huge"
    }
    return "enormous"
}
```

Sub-Tests

- we can use the `t.Run()` method to group tests together, in order to share setup and teardown and create hierarchical tests

```
func TestSize(t *testing.T) {  
    t.Log("Setup!") // common setup code can go here  
    t.Run("neg", func(t *testing.T) {  
        if Size(-1) != "negative" {  
            t.Error("-1 not negative")  
        }  
    })  
    t.Run("big", func(t *testing.T) {  
        if Size(9) != "big" {  
            t.Error("9 not big")  
        }  
    })  
    t.Log("Teardown!") // common teardown code can go here  
}
```

- each subtest has a unique name—the name of the top-level test plus the sequence of names passed to `t.Run()`, separated by slashes, with an optional trailing sequence number for disambiguation

Sub-Tests (cont'd)

- we can specify which test should run using command-line args

```
go test -run ''          # Run all tests (same as `go test`__)
go test -run Size        # Run top-level tests matching "Size", such as "TestSize"
go test -run Size/ne     # For top-level tests matching "Size", run subtests matching "ne"
go test -run /ne         # For all top-level tests, run subtests matching "ne"
```

238

Testing Best Practices

put tests in a different package

- if they're in the same package they have access to the package internals
- in a different package you are limited to the exposed API

put internal tests in a different file, e.g., foo_internal_test.go

- good for TDD

use test tables

aim for 100% coverage

- but only if your tests are actually testing what they should be

239

Benchmarking

Benchmarks

- examine the performance of your Go code
- part of the testing package
- placed inside `*_test.go` files, prefaced with `Benchmark` as opposed to `Test`
- benchmark functions take a single argument, a pointer to a `testing.B`
- executed with `go test -bench <regexp>`
- use `.` to run all benchmarks
- benchmarks are run several times by the testing package
- the value of `b.N` will increase each time until the benchmark runner is satisfied with the stability of the benchmark

241

Benchmarks (cont'd)

```
func Fib(n int) int {
    if n < 2 {
        return n
    }
    return Fib(n - 1) + Fib(n - 2)
}

func BenchmarkFib10(b *testing.B) {
    // run the Fib function b.N times
    for n := 0; n < b.N; n++ {
        Fib(10)
    }
}

.
```

[How to Write Benchmarks in Go](https://dave.cheney.net/2013/06/30/how-to-write-benchmarks-in-go) (<https://dave.cheney.net/2013/06/30/how-to-write-benchmarks-in-go>)

242

Benchmarks (cont'd)

```
$ go test -bench=.  
goos: darwin  
goarch: amd64  
pkg: fib  
BenchmarkFib10-4      5000000      359 ns/op  
PASS  
ok    fib    2.184s
```

- PASS output is from running tests
- use -run <regexp> if you want to avoid running tests
- second line is number of times test was run (i.e., b.N), followed by average execution time for the test

[How to Write Benchmarks in Go](https://dave.cheney.net/2013/06/30/how-to-write-benchmarks-in-go) (<https://dave.cheney.net/2013/06/30/how-to-write-benchmarks-in-go>)

243

Benchmarking Various Inputs

- given that the Fib function is the classic recursive implementation, we'd expect it to slow down exponentially for large values of n
- we can rewrite our benchmark slightly using a common pattern

```
func benchmarkFib(i int, b *testing.B) {  
    for n := 0; n < b.N; n++ {  
        Fib(i)  
    }  
}  
  
func BenchmarkFib1(b *testing.B) { benchmarkFib(1, b) }  
func BenchmarkFib2(b *testing.B) { benchmarkFib(2, b) }  
func BenchmarkFib3(b *testing.B) { benchmarkFib(3, b) }  
func BenchmarkFib10(b *testing.B) { benchmarkFib(10, b) }  
func BenchmarkFib20(b *testing.B) { benchmarkFib(20, b) }  
func BenchmarkFib40(b *testing.B) { benchmarkFib(40, b) }
```

- note that `benchmarkFib` has to be private to prevent the testing driver from trying to run it directly

[How to Write Benchmarks in Go](https://dave.cheney.net/2013/06/30/how-to-write-benchmarks-in-go) (<https://dave.cheney.net/2013/06/30/how-to-write-benchmarks-in-go>)

244

Benchmarking Various Inputs (cont'd)

```
$ go test -bench=.
goos: darwin
goarch: amd64
pkg: fib
BenchmarkFib1-4          1000000000      2.59 ns/op
BenchmarkFib2-4          2000000000      7.07 ns/op
BenchmarkFib3-4          1000000000      12.0 ns/op
BenchmarkFib10-4         3000000          433 ns/op
BenchmarkFib20-4         30000           52993 ns/op
BenchmarkFib40-4          2             779718761 ns/op
PASS
ok    fib    12.440s
```

- each benchmark runs for a minimum of 1 second by default
- if the second has not elapsed when the Benchmark function returns, the value of b.N is increased in the sequence 1, 2, 5, 10, 20, 50, ... and the function run again
- the final BenchmarkFib40 only ran twice with the average just under 1s per run
- we can increase the minimum benchmark time using the -benctime flag to produce a more accurate result

Benchmarking Gotchas

- two examples of a faulty benchmark...

```
func BenchmarkFibWrong(b *testing.B) {
    for n := 0; n < b.N; n++ {
        Fib(n)
    }
}

func BenchmarkFibWrong2(b *testing.B) {
    Fib(b.N)
}
```

- these benchmarks will never complete because the runtime of the benchmark increases as `b.N` grows

246

Documentation

Documentation

- Go takes documentation seriously
- documentation is integral to making software accessible and maintainable
- docs should be coupled to the code so they evolve along with the code
- historically it hasn't been so easy to produce docs
- Python has Docstrings, Java has Javadoc ... Go has godoc
- the easier it is for programmers to produce good docs, the better for everyone
- godoc parses Go source code—including comments—and produces documentation as HTML or plain text
- the result is documentation tightly coupled with the code it documents
- conceptually related to Docstring and Javadoc but simpler
- comments read by godoc are not language constructs (like Docstrings)
- the comments don't have to have their own machine-readable syntax (like Javadoc)

How Does godoc Work?

- to document a type/variable/constant/function/package, write a regular comment directly preceding its declaration, w/no intervening blank line
- godoc presents that comment as text alongside the item it documents
- for e.g., this is the documentation for `fmt.Fprint`

```
// Fprint formats using the default formats for its operands and writes to w.  
// Spaces are added between operands when neither is a string.  
// It returns the number of bytes written and any write error encountered.  
func Fprint(w io.Writer, a ...interface{}) (n int, err error) {
```

the above comment is a complete sentence, beginning with the name of the element it describes, which is an important convention allowing:

- documentation to be generated in a variety of formats, from plain text to HTML to UNIX man pages
- makes it read better when tools truncate it for brevity

How Does godoc Work? (cont'd)

- comments on package declarations should provide general package documentation, and can be short-like the sort package's brief description:

```
// Package sort provides primitives for sorting slices and user-defined  
// collections.  
package sort
```

- comments not adjacent to a top-level declaration are omitted from godoc's output
- ...except for top-level comments beginning with the word BUG(who), which are included in the Bugs section of the package documentation
- the who part is the user name of someone who could provide more information
- for e.g., this is a known issue from the bytes package:

```
// BUG(r): The rule Title uses for word boundaries does not handle Unicode  
// punctuation properly.
```

- let's try it with our stack package...

250

Resources

[Official Docs](https://golang.org/doc) (<https://golang.org/doc>)

[Playground](https://play.golang.org/) (<https://play.golang.org/>)

[Tour of Go](https://tour.golang.org/welcome/1) (<https://tour.golang.org/welcome/1>)

[Go Modules](https://blog.golang.org/migrating-to-go-modules) (<https://blog.golang.org/migrating-to-go-modules>)

[Effective Go](https://golang.org/doc/effective_go.html) (https://golang.org/doc/effective_go.html)

[Go for Pythonistas](https://talks.golang.org/2013/go4python.slide) (<https://talks.golang.org/2013/go4python.slide>)

[Concurrency is not Parallelism](https://blog.golang.org/concurrency-is-not-parallelism) (<https://blog.golang.org/concurrency-is-not-parallelism>)

[Less is Exponentially More](https://commandcenter.blogspot.com/2012/06/less-is-exponentially-more.html) (<https://commandcenter.blogspot.com/2012/06/less-is-exponentially-more.html>)

[Go frameworks, libraries and software!](https://awesome-go.com/) (<https://awesome-go.com/>)

[Go Slice Tricks](https://github.com/golang/go/wiki/SliceTricks) (<https://github.com/golang/go/wiki/SliceTricks>)

[Go 2.0 Draft: Error Handling](https://go.googlesource.com/proposal/+/master/design/go2draft-error-handling.md) (<https://go.googlesource.com/proposal/+/master/design/go2draft-error-handling.md>)

Go 2.0 Error Handling — Problem Overview

(<https://go.googlesource.com/proposal/+/master/design/go2draft-error-handling-overview.md>)

Thank you