GO BOOTCAMP

Everything you need to know to get started with Go.

Matt Aimonetti

Go Bootcamp

Everything you need to know to get started with Go.

Matt Aimonetti

Contents

Preface

Last updated: July 28, 2016

Cover art by Erick Zelaya sponsored by Ardan Studios

The Gopher character on the cover is based on the Go mascot designed by Renée French and copyrighted under the Creative Commons Attribution 3.0 license.

This book is a companion book to the Go Bootcamp event organized for the first time in Santa Monica CA in March 2014.

You don't need to install Go to read this book and follow along. This book was designed to delay the setup process and let you evaluate Go without having to go through the installation process. All examples and solutions are available online in an environment allowing you to execute code right in the browser.

Chapter 1

Intro

The very first Go Bootcamp event was put together by Matt Aimonetti from Splice with the support of a long list of volunteers.

- Francesc Campoy
- Mitchell Hashimoto
- Evan Phoenix
- Jeremy Saenz
- Nic Williams
- Ross Hale

Content and review has been provided by various members of the community, if you want to add content or provide corrections, feel free to send a pull request to this book's git repo.

Git hosting for this book is provided by GitHub and is available here.

This material is going to grow over time and is meant as a reference to people learning Go. Feel free to use this book to help you organize your own Go Bootcamp events or to learn Go. Don't forget to mention upcoming events in the mailing list below.

If you have any questions or want to know about future events, please consider joining the Go Bootcamp Mailing List

1.1 Objectives

After the event, we expect the attendees to leave with the following knowledge, skills and attributes.

1.1.1 Knowledge

- pros/cons of static typing in Go
- what makes Go unique
- what is Go particularly good at
- what are the challenging parts of Go

1.1.2 Skills

- know how to do data modeling with Go
- know how to organize code in packages
- know how to test code
- know how to write documentation

- know how to use JSON marshaling
- know how to build a web API (depending on exercises)
- know how to test a web API (depending on exercises)
- know how to cross compile
- know how to use the key go tools

1.1.3 Attitudes

- value the potential of the Go language
- can argue when to use Go vs using “legacy language”
- consider using Go for a future project

Chapter 2

The Basics

Go is often referred to as a "simple" programming language, a language that can be learned in a few hours if you already know another language. Go was designed to feel familiar and to stay as simple as possible, the entire language specification fits in just a few pages.

There are a few concepts we are going to explore before writing our first application.

2.1 Variables & inferred typing

The var statement declares a list of variables with the type declared last.

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

Or even

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

Variables can also be declared one by one:

```
var name     string
var age      int
var location string
```

A var declaration can include initializers, one per variable.

```
var (
        name     string = "Prince Oberyn"
        age      int    =  32
        location string = "Dorne"
)
```

If an initializer is present, the type can be omitted, the variable will take the type of the initializer (inferred typing).

```
var (
        name     = "Prince Oberyn"
        age      =  32
        location = "Dorne"
)
```

You can also initialize variables on the same line:

```
var (
        name, location, age = "Prince Oberyn", "Dorne", 32
)
```

Inside a function, the `:=` short assignment statement can be used in place of a var declaration with implicit type.

```
func main() {
        name, location := "Prince Oberyn", "Dorne"
        age := 32
        fmt.Printf("%s (%d) of %s", name, age, location)
}
```

See in Playground

A variable can contain any type, including functions:

```
func main() {
        action := func() {
                //doing something
        }
        action()
}
```

See in Playground

Outside a function, every construct begins with a keyword (**var**, **func**, and so on) and the **:=** construct is not available.

- Go's declaration Syntax

2.2 Constants

Constants are declared like variables, but with the **const** keyword.

Constants can only be character, string, boolean, or numeric values and cannot be declared using the **:=** syntax. An untyped constant takes the type needed by its context.

```
const Pi = 3.14
const (
        StatusOK                    = 200
        StatusCreated               = 201
        StatusAccepted              = 202
        StatusNonAuthoritativeInfo = 203
        StatusNoContent             = 204
        StatusResetContent          = 205
        StatusPartialContent        = 206
)
```

```
package main

import "fmt"
```

```
const (
        Pi    = 3.14
        Truth = false
        Big   = 1 << 62
        Small = Big >> 61
)

func main() {
        const Greeting = ""
        fmt.Println(Greeting)
        fmt.Println(Pi)
        fmt.Println(Truth)
        fmt.Println(Big)
}
```

See in Playground

2.3 Printing Constants and Variables

While you can print the value of a variable or constant using the built-in **print** and **println** functions, the more idiomatic and flexible way is to use the **fmt** package

```
func main() {
        cylonModel := 6
        fmt.Println(cylonModel)
}
```

fmt.Println prints the passed in variables' values and appends a newline. **fmt.Printf** is used when you want to print one or multiple values using a defined format specifier.

```
func main() {
        name := "Caprica-Six"
        aka := fmt.Sprintf("Number %d", 6)
        fmt.Printf("%s is also known as %s",
                name, aka)
}
```

See in Playground

Read the **fmt** package documentation to see the available flags to create a format specifier.

2.4 Packages and imports

Every Go program is made up of packages. Programs start running in package **main**.

```
package main

func main() {
        print("Hello, World!\n")
}
```

If you are writing an executable code (versus a library), then you need to define a **main** package and a **main()** function which will be the entry point to your software.

By convention, the package name is the same as the last element of the import path. For instance, the "math/rand" package comprises files that begin with the statement package rand.

Import statement examples:

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

Or grouped:

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

- Go Tour: Packages

- Go Tour: Imports

Usually, non standard lib packages are namespaced using a web url. For instance, I ported to Go some Rails logic, including the cryptography code used in Rails 4. I hosted the source code containing a few packages on github, in the following repository http://github.com/mattetti/goRailsYourself

To import the crypto package, I would need to use the following import statement:

```
import "github.com/mattetti/goRailsYourself/crypto"
```

2.5 Code location

The snippet above basically tells the compiler to import the crypto package available at the `github.com/mattetti/goRailsYourself/crypto` path. It doesn't mean that the compiler will automatically pull down the repository, so where does it find the code?

You need to pull down the code yourself. The easiest way is to use the `go get` command provided by Go.

```
$ go get github.com/mattetti/goRailsYourself/crypto
```

This command will pull down the code and put it in your Go path. When installing Go, we set the `GOPATH` environment variable and that is what's used to store binaries and libraries. That's also where you should store your code (your workspace).

```
$ ls $GOPATH
bin        pkg        src
```

The `bin` folder will contain the Go compiled binaries. You should probably add the bin path to your system path.

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

Finally the `src` folder contains all the Go source code organized by import path:

```
$ ls $GOPATH/src
bitbucket.org        code.google.com        github.com        launchpad.net
```

```
$ ls $GOPATH/src/github.com/mattetti
goblin                          goRailsYourself                jet
```

When starting a new program or library, it is recommended to do so inside the `src` folder, using a fully qualified path (for instance: `github.com/<your username>/<project name>`)

2.6 Exported names

After importing a package, you can refer to the names it exports (meaning variables, methods and functions that are available from outside of the package). In Go, a name is exported if it begins with a capital letter. `Foo` is an exported name, as is `FOO`. The name `foo` is not exported.

See the difference between:

```
import (
    "fmt"
    "math"
)

func main() {
    fmt.Println(math.pi)
}
```

and

```
func main() {
    fmt.Println(math.Pi)
}
```

`Pi` is exported and can be accessed from outside the page, while `pi` isn't available.

```
cannot refer to unexported name math.pi
```

- See in Playground

Use the provided Go documentation or godoc.org to find exported names.

- Exported names example

2.7 Functions, signature, return values, named results

A function can take zero or more typed arguments. The type comes after the variable name. Functions can be defined to return any number of values that are always typed.

```
package main

import "fmt"

func add(x int, y int) int {
    return x + y
}

func main() {
    fmt.Println(add(42, 13))
}
```

- Go Tour Functions example

In the following example, instead of declaring the type of each parameter, we only declare one type that applies to both.

```
package main

import "fmt"

func add(x, y int) int {
    return x + y
}

func main() {
    fmt.Println(add(42, 13))
}
```

- See in Playground

In the following example, the **location** function returns two string values.

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

        switch city {
        case "Los Angeles", "LA", "Santa Monica":
                region, continent = "California", "North America"
        case "New York", "NYC":
                region, continent = "New York", "North America"
        default:
                region, continent = "Unknown", "Unknown"
        }
        return region, continent
}

func main() {
        region, continent := location("Santa Monica")
        fmt.Printf("Matt lives in %s, %s", region, continent)
}
```

- See in playground

Functions take parameters. In Go, functions can return multiple “result parameters”, not just a single value. They can be named and act just like variables.

If the result parameters are named, a return statement without arguments returns the current values of the results.

```
func location(name, city string) (region, continent string) {
        switch city {
        case "New York", "LA", "Chicago":
                continent = "North America"
        default:
                continent = "Unknown"
        }
        return
}

func main() {
        region, continent := location("Matt", "LA")
        fmt.Printf("%s lives in %s", region, continent)
}
```

- See in Playground

I personally recommend against using named return parameters because they often cause more confusion than they save time or help clarify your code.

2.8 Pointers

Go has pointers, but no pointer arithmetic. Struct fields can be accessed through a struct pointer. The indirection through the pointer is transparent (you can directly call fields and methods on a pointer).

Note that by default Go passes arguments by value (copying the arguments), if you want to pass the arguments by reference, you need to pass pointers (or use a structure using reference values like slices (Section 4.2) and maps (Section 4.4).

To get the pointer of a value, use the `&` symbol in front of the value; to dereference a pointer, use the `*` symbol.

Methods are often defined on pointers and not values (although they can be defined on both), so you will often store a pointer in a variable as in the example below:

```
client := &http.Client{}
resp, err := client.Get("http://gobootcamp.com")
```

2.9 Mutability

In Go, only constants are immutable. However because arguments are passed by value, a function receiving a value argument and mutating it, won't mutate the original value.

```
package main

import "fmt"

type Artist struct {
        Name, Genre string
        Songs       int
}

func newRelease(a Artist) int {
        a.Songs++
        return a.Songs
}

func main() {
        me := Artist{Name: "Matt", Genre: "Electro", Songs: 42}
        fmt.Printf("%s released their %dth song\n", me.Name, newRelease(me))
        fmt.Printf("%s has a total of %d songs", me.Name, me.Songs)
}
```

```
Matt released their 43th song
Matt has a total of 42 songs
```

See in Playground

As you can see the total amount of songs on the `me` variable's value wasn't changed. To mutate the passed value, we need to pass it by reference, using a pointer.

```
package main

import "fmt"

type Artist struct {
        Name, Genre string
        Songs       int
}

func newRelease(a *Artist) int {
        a.Songs++
        return a.Songs
}

func main() {
        me := &Artist{Name: "Matt", Genre: "Electro", Songs: 42}
        fmt.Printf("%s released their %dth song\n", me.Name, newRelease(me))
        fmt.Printf("%s has a total of %d songs", me.Name, me.Songs)
}
```

See in Playground

The only change between the two versions is that **`newRelease`** takes a pointer to an **`Artist`** value and when I initialize our **`me`** variable, I used the **`&`** symbol to get a pointer to the value.

Another place where you need to be careful is when calling methods on values as explained a bit later (Section 6.3)

Chapter 3

Types

3.1 Basic types

```
bool
string

Numeric types:

uint        either 32 or 64 bits
int         same size as uint
uintptr     an unsigned integer 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        the set of all signed  8-bit integers (-128 to 127)
int16       the set of all signed 16-bit integers (-32768 to 32767)
int32       the set of all signed 32-bit integers (-2147483648 to 2147483647)
int64       the set of all signed 64-bit integers
            (-9223372036854775808 to 9223372036854775807)

float32     the set of all IEEE-754 32-bit floating-point numbers
float64     the set of all IEEE-754 64-bit floating-point numbers

complex64   the set of all complex numbers with float32 real and imaginary parts
complex128  the set of all complex numbers with float64 real and imaginary parts

byte        alias for uint8
rune        alias for int32 (represents a Unicode code point)
```

Example of some of the built-in types:

```
package main

import (
        "fmt"
        "math/cmplx"
)

var (
        goIsFun bool       = true
        maxInt  uint64     = 1<<64 - 1
        complex complex128 = cmplx.Sqrt(-5 + 12i)
)

func main() {
        const f = "%T(%v)\n"
        fmt.Printf(f, goIsFun, goIsFun)
        fmt.Printf(f, maxInt, maxInt)
        fmt.Printf(f, complex, complex)
}
```

```
bool(true)
uint64(18446744073709551615)
complex128((2+3i))
```

- See in Playground

3.2 Type conversion

The expression `T(v)` converts the value `v` to the type `T`. Some numeric conversions:

```
var i int = 42
var f float64 = float64(i)
var u uint = uint(f)
```

Or, put more simply:

```
i := 42
f := float64(i)
u := uint(f)
```

Go assignment between items of different type requires an explicit conversion which means that you manually need to convert types if you are passing a variable to a function expecting another type.

3.3 Type assertion

If you have a value and want to convert it to another or a specific type (in case of `interface{}`), you can use type assertion. A type assertion takes a value and tries to create another version in the specified explicit type.

In the example below, the `timeMap` function takes a value and if it can be asserted as a map of `interfaces{}` keyed by strings, then it injects a new entry called "updated_at" with the current time value.

```
package main

import (
        "fmt"
        "time"
)

func timeMap(y interface{}) {
        z, ok := y.(map[string]interface{})
        if ok {
                z["updated_at"] = time.Now()
        }
}

func main() {
        foo := map[string]interface{}{
                "Matt": 42,
        }
        timeMap(foo)
        fmt.Println(foo)
}
```

See in playground

The type assertion doesn't have to be done on an empty interface. It's often used when you have a function taking a param of a specific interface but the function inner code behaves differently based on the actual object type. Here is an example:

```
package main

import "fmt"

type Stringer interface {
        String() string
}

type fakeString struct {
        content string
}

// function used to implement the Stringer interface
func (s *fakeString) String() string {
        return s.content
}

func printString(value interface{}) {
        switch str := value.(type) {
        case string:
                fmt.Println(str)
        case Stringer:
                fmt.Println(str.String())
        }
}

func main() {
        s := &fakeString{"Ceci n'est pas un string"}
        printString(s)
        printString("Hello, Gophers")

}
```

- See in Playground

Another example is when checking if an error is of a certain type:

```
if err != nil {
  if msqlerr, ok := err.(*mysql.MySQLError); ok && msqlerr.Number == 1062 {
    log.Println("We got a MySQL duplicate :(")
```

```
  } else {
    return err
  }
}
```

- Read more in the Effective Go guide

3.4 Structs

A struct is a collection of fields/properties. You can define new types as structs or interfaces (Chapter 7). If you are coming from an object-oriented background, you can think of a struct to be a light class that supports composition but not inheritance. Methods are discussed at length in Chapter 6

You don't need to define getters and setters on struct fields, they can be accessed automatically. However, note that only exported fields (capitalized) can be accessed from outside of a package.

A struct literal sets a newly allocated struct value by listing the values of its fields. You can list just a subset of fields by using the **`"Name:" syntax`** (the order of named fields is irrelevant when using this syntax). The special prefix **`&`** constructs a pointer to a newly allocated struct.

```
package main

import (
        "fmt"
        "time"
)

type Bootcamp struct {
        // Latitude of the event
        Lat float64
        // Longitude of the event
        Lon float64
        // Date of the event
        Date time.Time
}

func main() {
        fmt.Println(Bootcamp{
                Lat:  34.012836,
```

```
                Lon:  -118.495338,
                Date: time.Now(),
        })
}
```

- See in Playground

Declaration of struct literals:

```
package main

import "fmt"

type Point struct {
        X, Y int
}

var (
        p = Point{1, 2}  // has type Point
        q = &Point{1, 2} // has type *Point
        r = Point{X: 1}  // Y:0 is implicit
        s = Point{}      // X:0 and Y:0
)

func main() {
        fmt.Println(p, q, r, s)
}
```

- See in playground

Accessing fields using the dot notation:

```
package main

import (
        "fmt"
        "time"
)

type Bootcamp struct {
        Lat, Lon float64
        Date     time.Time
}
```

```
func main() {
        event := Bootcamp{
                Lat: 34.012836,
                Lon: -118.495338,
        }
        event.Date = time.Now()
        fmt.Printf("Event on %s, location (%f, %f)",
                event.Date, event.Lat, event.Lon)

}
```

- See in Playground

3.5 Initializing

Go supports the **new** expression to allocate a zeroed value of the requested type and to return a pointer to it.

```
x := new(int)
```

As seen in (Section 3.4) a common way to "initialize" a variable containing a struct or a reference to one, is to create a struct literal. Another option is to create a constructor. This is usually done when the zero value isn't good enough and you need to set some default field values for instance.

Note that following expressions using **new** and an empty struct literal are equivalent and result in the same kind of allocation/initialization:

```
package main

import (
        "fmt"
)

type Bootcamp struct {
        Lat float64
        Lon float64
}
```

```
func main() {
        x := new(Bootcamp)
        y := &Bootcamp{}
        fmt.Println(*x == *y)
}
```

- See in playground

Note that slices (Section 4.2), maps (Section 4.4) and channels (Section 8.2) are usually allocated using `make` so the data structure these types are built upon can be initialized.

Resources:

- Allocation with `new` - effective Go
- Composite Literals - effective Go
- Allocation with `make` - effective Go

3.6 Composition vs inheritance

Coming from an OOP background a lot of us are used to inheritance, something that isn't supported by Go. Instead you have to think in terms of composition and interfaces.

The Go team wrote a short but good segment on this topic.

Composition (or embedding) is a well understood concept for most OOP programmers and Go supports it, here is an example of the problem it's solving:

```
package main

import "fmt"

type User struct {
        Id       int
        Name     string
        Location string
}
```

```
type Player struct {
        Id       int
        Name     string
        Location string
        GameId         int
}

func main() {
        p := Player{}
        p.Id = 42
        p.Name = "Matt"
        p.Location = "LA"
        p.GameId = 90404
        fmt.Printf("%+v", p)
}
```

- See in Playground

The above example demonstrates a classic OOP challenge, our **`Player`** struct has the same fields as the **`User`** struct but it also has a **`GameId`** field. Having to duplicate the field names isn't a big deal, but it can be simplified by composing our struct.

```
type User struct {
        Id             int
        Name, Location string
}

type Player struct {
        User
        GameId int
}
```

We can initialize a new variable of type **`Player`** two different ways. Using the dot notation to set the fields:

```
package main

import "fmt"

type User struct {
        Id             int
```

```
        Name, Location string
}

type Player struct {
        User
        GameId int
}

func main() {
        p := Player{}
        p.Id = 42
        p.Name = "Matt"
        p.Location = "LA"
        p.GameId = 90404
        fmt.Printf("%+v", p)
}
```

- See in Playground

The other option is to use a struct literal:

```
package main

import "fmt"

type User struct {
        Id              int
        Name, Location string
}

type Player struct {
        User
        GameId int
}

func main() {
        p := Player{
                User{Id: 42, Name: "Matt", Location: "LA"},
                90404,
        }
        fmt.Printf(
                "Id: %d, Name: %s, Location: %s, Game id: %d\n",
                p.Id, p.Name, p.Location, p.GameId)
        // Directly set a field defined on the Player struct
        p.Id = 11
        fmt.Printf("%+v", p)
}
```

- See in Playground

When using a struct literal with an implicit composition, we can't just pass the composed fields. We instead need to pass the types composing the struct. Once set, the fields are directly available.

Because our struct is composed of another struct, the methods on the **`User`** struct is also available to the **`Player`**. Let's define a method to show that behavior:

```
package main

import "fmt"

type User struct {
        Id              int
        Name, Location string
}

func (u *User) Greetings() string {
        return fmt.Sprintf("Hi %s from %s",
                u.Name, u.Location)
}

type Player struct {
        User
        GameId int
}

func main() {
        p := Player{}
        p.Id = 42
        p.Name = "Matt"
        p.Location = "LA"
        fmt.Println(p.Greetings())
}
```

- See in Playground

As you can see this is a very powerful way to build data structures but it's even more interesting when thinking about it in the context of interfaces. By composing one of your structure with one implementing a given interface, your structure automatically implements the interface.

Here is another example, this time we will look at implementing a **Job** struct that can also behave as a logger.

Here is the explicit way:

```
package main

import (
        "log"
        "os"
)

type Job struct {
        Command string
        Logger  *log.Logger
}

func main() {
        job := &Job{"demo", log.New(os.Stderr, "Job: ", log.Ldate)}
        // same as
        // job := &Job{Command: "demo",
        //             Logger: log.New(os.Stderr, "Job: ", log.Ldate)}
        job.Logger.Print("test")
}
```

- See in playground

Our **Job** struct has a field called **Logger** which is a pointer to another type (log.Logger)

When we initialize our value, we set the logger so we can then call its **Print** function by chaining the calls: **job.Logger.Print()**

But Go lets you go even further and use implicit composition. We can skip defining the field for our logger and now all the methods available on a pointer to **log.Logger** are available from our struct:

```
package main

import (
        "log"
        "os"
)

type Job struct {
```

```
        Command string
        *log.Logger
}

func main() {
        job := &Job{"demo", log.New(os.Stderr, "Job: ", log.Ldate)}
        job.Print("starting now...")
}
```

- See in Playground

Note that you still need to set the logger and that's often a good reason to use a constructor (custom constructors are used when you need to set a structure before using a value, see (Section 11.5)). What is really nice with the implicit composition is that it allows to easily and cheaply make your structs implement interfaces. Imagine that you have a function that takes variables implementing an interface with the `Print` method. By adding `*log.Logger` to your struct (and initializing it properly), your struct is now implementing the interface without you writing any custom methods.

3.7 Exercise

Looking at the `User` / `Player` example, you might have noticed that we composed `Player` using `User` but it might be better to compose it with a pointer to a `User` struct. The reason why a pointer might be better is because in Go, arguments are passed by value and not reference. If you have a small struct that is inexpensive to copy, that is fine, but more than likely, in real life, our `User` struct will be bigger and should not be copied. Instead we would want to pass by reference (using a pointer). (Section 2.9 & Section 6.3 discuss more in depth how calling a method on a type value vs a pointer affects mutability and memory allocation)

Modify the code to use a pointer but still be able to initialize without using the dot notation.

```
package main

import "fmt"

type User struct {
        Id              int
        Name, Location string
}

func (u *User) Greetings() string {
        return fmt.Sprintf("Hi %s from %s",
                u.Name, u.Location)
}

type Player struct {
        *User
        GameId int
}

func main() {
        // insert code
}
```

- See in Playground

Question: We defined the `Greetings` method on a pointer to a `User` type. How come we were able to call it directly on the value?

3.7.1 Solution

```
package main

import "fmt"

type User struct {
        Id              int
        Name, Location string
}

func (u *User) Greetings() string {
        return fmt.Sprintf("Hi %s from %s",
                u.Name, u.Location)
}

type Player struct {
```

```
        *User
        GameId int
}

func NewPlayer(id int, name, location string, gameId int) *Player {
        return &Player{
                User:   &User{id, name, location},
                GameId: gameId,
        }
}

func main() {
        p := NewPlayer(42, "Matt", "LA", 90404)
        fmt.Println(p.Greetings())
}
```

- See in Playground

Answer: That is because methods defined on a pointer are also automatically available on the value itself. The example didn't use a pointer on purpose, so the dot notation was working right away. In the pointer solution, a zero value player is composed of a nil pointer of type `User` and therefore, we can't call a field on a nil pointer.

Chapter 4

Collection Types

4.1 Arrays

The type `[n]T` is an array of `n` values of type `T`.

The expression:

```
var a [10]int
```

declares a variable `a` as an array of ten integers.

An array's length is part of its type, so arrays cannot be resized. This seems limiting, but don't worry; Go provides a convenient way of working with arrays.

```
package main

import "fmt"

func main() {
    var a [2]string
    a[0] = "Hello"
    a[1] = "World"
    fmt.Println(a[0], a[1])
    fmt.Println(a)
}
```

- Golang tour page

You can also set the array entries as you declare the array:

```
package main

import "fmt"

func main() {
    a  := [2]string{"hello", "world!"}
    fmt.Printf("%q", a)
}
```

- See in Playground

Finally, you can use an ellipsis to use an implicit length when you pass the values:

```
package main

import "fmt"

func main() {
        a := [...]string{"hello", "world!"}
        fmt.Printf("%q", a)
}
```

- See in Playground

4.1.1 Printing arrays

Note how we used the **`fmt`** package using **`Printf`** and used the **`%q`** "verb" to print each element quoted.

If we had used **`Println`** or the **`%s`** verb, we would have had a different result:

```
package main

import "fmt"
```

```
func main() {
        a := [2]string{"hello", "world!"}
        fmt.Println(a)
        // [hello world!]
        fmt.Printf("%s\n", a)
        // [hello world!]
        fmt.Printf("%q\n", a)
        // ["hello" "world!"]
}
```

- See in Playground

4.1.2 Multi-dimensional arrays

You can also create multi-dimensional arrays:

```
package main

import "fmt"

func main() {
        var a [2][3]string
        for i := 0; i < 2; i++ {
                for j := 0; j < 3; j++ {
                        a[i][j] = fmt.Sprintf("row %d - column %d", i+1, j+1)
                }
        }
        fmt.Printf("%q", a)
        // [["row 1 - column 1" "row 1 - column 2" "row 1 - column 3"]
        //  ["row 2 - column 1" "row 2 - column 2" "row 2 - column 3"]]
}
```

- See in Playground

Trying to access or set a value at an index that doesn't exist will prevent your program from compiling, for instance, try to compile the following code:

```
package main

func main() {
    var a [2]string
    a[3] = "Hello"
}
```

You will see that the compiler will report the following error:

```
Invalid array index 3 (out of bounds for 2-element array)
```

That's because our array is of length 2 meaning that the only 2 available indexes are 0 and 1. Trying to access index 3 results in an error that tells us that we are trying to access an index that is out of bounds since our array only contains 2 elements and we are trying to access the 4th element of the array.

Slices, the type that we are going to see next is more often used, due to the fact that we don't always know in advance the length of the array we need.

4.2 Slices

Slices wrap arrays to give a more general, powerful, and convenient interface to sequences of data. Except for items with explicit dimension such as transformation matrices, most array programming in Go is done with slices rather than simple arrays.

Slices hold references to an underlying array, and if you assign one slice to another, both refer to the same array. If a function takes a slice argument, changes it makes to the elements of the slice will be visible to the caller, analogous to passing a pointer to the underlying array.

A slice points to an array of values and also includes a length. Slices can be resized since they are just a wrapper on top of another data structure.

`[]T` is a slice with elements of type `T`.

```go
package main

import "fmt"

func main() {
    p := []int{2, 3, 5, 7, 11, 13}
    fmt.Println(p)
    // [2 3 5 7 11 13]
}
```

- Go tour page

4.2.1 Slicing a slice

Slices can be re-sliced, creating a new slice value that points to the same array.
The expression

```
s[lo:hi]
```

evaluates to a slice of the elements from `lo` through `hi-1`, inclusive. Thus

```
s[lo:lo]
```

is empty and

```
s[lo:lo+1]
```

has one element.
Note: `lo` and `hi` would be integers representing indexes.

```
package main

import "fmt"

func main() {
        mySlice := []int{2, 3, 5, 7, 11, 13}
        fmt.Println(mySlice)
        // [2 3 5 7 11 13]

        fmt.Println(mySlice[1:4])
        // [3 5 7]

        // missing low index implies 0
        fmt.Println(mySlice[:3])
        // [2 3 5]

        // missing high index implies len(s)
        fmt.Println(mySlice[4:])
        // [11 13]
}
```

- See in Playground
- Go tour page

4.2.2 Making slices

Besides creating slices by passing the values right away (slice literal), you can also use **make**. You create an empty slice of a specific length and then populate each entry:

```
package main

import "fmt"

func main() {
        cities := make([]string, 3)
        cities[0] = "Santa Monica"
        cities[1] = "Venice"
        cities[2] = "Los Angeles"
        fmt.Printf("%q", cities)
        // ["Santa Monica" "Venice" "Los Angeles"]
}
```

- See in Playground

It works by allocating a zeroed array and returning a slice that refers to that array.

4.2.3 Appending to a slice

Note however, that you would get a runtime error if you were to do that:

```
cities := []string{}
cities[0] = "Santa Monica"
```

As explained above, a slice is seating on top of an array, in this case, the array is empty and the slice can't set a value in the referred array. There is a way to do that though, and that is by using the **append** function:

```
package main

import "fmt"

func main() {
        cities := []string{}
        cities = append(cities, "San Diego")
        fmt.Println(cities)
        // [San Diego]
}
```

- See in Playground

You can append more than one entry to a slice:

```
package main

import "fmt"

func main() {
        cities := []string{}
        cities = append(cities, "San Diego", "Mountain View")
        fmt.Printf("%q", cities)
        // ["San Diego" "Mountain View"]
}
```

- See in Playground

And you can also append a slice to another using an ellipsis:

```
package main

import "fmt"

func main() {
        cities := []string{"San Diego", "Mountain View"}
        otherCities := []string{"Santa Monica", "Venice"}
        cities = append(cities, otherCities...)
        fmt.Printf("%q", cities)
        // ["San Diego" "Mountain View" "Santa Monica" "Venice"]
}
```

- See in Playground

Note that the ellipsis is a built-in feature of the language that means that the element is a collection. We can't append an element of type slice of strings (`[]string`) to a slice of strings, only strings can be appended. However, using the ellipsis (`...`) after our slice, we indicate that we want to append each element of our slice. Because we are appending strings from another slice, the compiler will accept the operation since the types are matching.

You obviously can't append a slice of type `[]int` to another slice of type `[]string`.

4.2.4 Length

At any time, you can check the length of a slice by using `len`:

```
package main

import "fmt"

func main() {
        cities := []string{
                "Santa Monica",
                "San Diego",
                "San Francisco",
        }
        fmt.Println(len(cities))
        // 3
        countries := make([]string, 42)
        fmt.Println(len(countries))
        // 42
}
```

- See in Playground

4.2.5 Nil slices

The zero value of a slice is nil. A nil slice has a length and capacity of 0.

```go
package main

import "fmt"

func main() {
    var z []int
    fmt.Println(z, len(z), cap(z))
    // [] 0 0
    if z == nil {
        fmt.Println("nil!")
    }
    // nil!
}
```

- See in Playground
- Go tour page

4.2.6 Resources

For more details about slices:

- Go slices, usage and internals
- Effective Go - slices
- Append function documentation
- Slice tricks
- Effective Go - slices
- Effective Go - two-dimensional slices
- Go by example - slices

4.3 Range

The **range** form of the for loop iterates over a **slice** (Section 4.2) or a **map** (Section 4.4). Being able to iterate over all the elements of a data structure is very useful and **range** simplifies the iteration.

```
package main

import "fmt"

var pow = []int{1, 2, 4, 8, 16, 32, 64, 128}

func main() {
    for i, v := range pow {
        fmt.Printf("2**%d = %d\n", i, v)
    }
}
```

Which will print:

```
2**0 = 1
2**1 = 2
2**2 = 4
2**3 = 8
2**4 = 16
2**5 = 32
2**6 = 64
2**7 = 128
```

- Go tour page

You can skip the index or value by assigning to _. If you only want the index, drop the ", value" entirely.

```
package main

import "fmt"

func main() {
    pow := make([]int, 10)
    for i := range pow {
```

```
        pow[i] = 1 << uint(i)
    }
    for _, value := range pow {
        fmt.Printf("%d\n", value)
    }
}
```

- Go tour page

4.3.1 Break & continue

As if you were using a normal for loop, you can stop the iteration anytime by using **`break`**:

```
package main

import "fmt"

func main() {
        pow := make([]int, 10)
        for i := range pow {
                pow[i] = 1 << uint(i)
                if pow[i] >= 16 {
                        break
                }
        }
        fmt.Println(pow)
        // [1 2 4 8 16 0 0 0 0 0]
}
```

- See in Playground

You can also skip an iteration by using **`continue`**:

```
package main

import "fmt"

func main() {
        pow := make([]int, 10)
        for i := range pow {
```

```
                if i%2 == 0 {
                        continue
                }
                pow[i] = 1 << uint(i)
        }
        fmt.Println(pow)
        // [0 2 0 8 0 32 0 128 0 512]
}
```

- See in Playground

4.3.2 Range and maps

Range can also be used on maps (Section 4.4), in that case, the first parameter isn't an incremental integer but the map key:

```
package main

import "fmt"

func main() {
        cities := map[string]int{
                "New York":    8336697,
                "Los Angeles": 3857799,
                "Chicago":     2714856,
        }
        for key, value := range cities {
                fmt.Printf("%s has %d inhabitants\n", key, value)
        }
}
```

Which will output:

```
New York has 8336697 inhabitants
Los Angeles has 3857799 inhabitants
Chicago has 2714856 inhabitants
```

- See in Playground

4.3.3 Exercise

Given a list of names, you need to organize each name within a slice based on its length.

```
package main

var names = []string{"Katrina", "Evan", "Neil", "Adam", "Martin", "Matt",
        "Emma", "Isabella", "Emily", "Madison",
        "Ava", "Olivia", "Sophia", "Abigail",
        "Elizabeth", "Chloe", "Samantha",
        "Addison", "Natalie", "Mia", "Alexis"}

func main() {
        // insert your code here
}
```

- See in Playground

After you implement your solution, you should get the following output (slice of slice of strings):

```
[[] [] [Ava Mia] [Evan Neil Adam Matt Emma] [Emily Chloe]
[Martin Olivia Sophia Alexis] [Katrina Madison Abigail Addison Natalie]
[Isabella Samantha] [Elizabeth]]
```

4.3.4 Solution

```
package main

import "fmt"

var names = []string{"Katrina", "Evan", "Neil", "Adam", "Martin", "Matt",
        "Emma", "Isabella", "Emily", "Madison",
        "Ava", "Olivia", "Sophia", "Abigail",
        "Elizabeth", "Chloe", "Samantha",
        "Addison", "Natalie", "Mia", "Alexis"}

func main() {
        var maxLen int
```

```
        for _, name := range names {
                if l := len(name); l > maxLen {
                        maxLen = l
                }
        }
        output := make([][]string, maxLen)
        for _, name := range names {
                output[len(name)-1] = append(output[len(name)-1], name)
        }

        fmt.Printf("%v", output)
}
```

- See in Playground

There are a few interesting things to note. To avoid an out of bounds insert, we need our **output** slice to be big enough. But we don't want it to be too big. That's why we need to do a first pass through all the names and find the longest. We use the longest name length to set the length of the **output** slice length. Slices are zero indexed, so when inserting the names, we need to get the length of the name minus one.

4.4 Maps

Maps are somewhat similar to what other languages call "dictionaries" or "hashes".

A map maps keys to values. Here we are mapping string keys (actor names) to an integer value (age).

```
package main

import "fmt"

func main() {
        celebs := map[string]int{
                "Nicolas Cage":       50,
                "Selena Gomez":       21,
                "Jude Law":           41,
                "Scarlett Johansson": 29,
        }

        fmt.Printf("%#v", celebs)
}
```

```
map[string]int{"Nicolas Cage":50, "Selena Gomez":21, "Jude Law":41,
    "Scarlett Johansson":29}
```

- See in Playground

When not using map literals like above, maps must be created with make (not new) before use. The nil map is empty and cannot be assigned to.

Assignments follow the Go convention and can be observed in the example below.

```
package main

import "fmt"

type Vertex struct {
        Lat, Long float64
}

var m map[string]Vertex

func main() {
        m = make(map[string]Vertex)
        m["Bell Labs"] = Vertex{40.68433, -74.39967}
        fmt.Println(m["Bell Labs"])
}
```

- See in Playground

When using map literals, if the top-level type is just a type name, you can omit it from the elements of the literal.

```
package main

import "fmt"

type Vertex struct {
        Lat, Long float64
}

var m = map[string]Vertex{
```

```
        "Bell Labs": {40.68433, -74.39967},
        // same as "Bell Labs": Vertex{40.68433, -74.39967}
        "Google": {37.42202, -122.08408},
}

func main() {
        fmt.Println(m)
}
```

- See in Playground

4.4.1 Mutating maps

Insert or update an element in map m:

```
m[key] = elem
```

Retrieve an element:

```
elem = m[key]
```

Delete an element:

```
delete(m, key)
```

Test that a key is present with a two-value assignment:

```
elem, ok = m[key]
```

- See in Playground

If **key** is in **m**, **ok** is true. If not, **ok** is false and **elem** is the zero value for the map's element type. Similarly, when reading from a map if the key is not present the result is the zero value for the map's element type.

4.4.2 Resources

- Go team blog post on maps
- Effective Go - maps

4.4.3 Exercise

Implement WordCount.

```
package main

import (
    "code.google.com/p/go-tour/wc"
)

func WordCount(s string) map[string]int {
    return map[string]int{"x": 1}
}

func main() {
    wc.Test(WordCount)
}
```

- See in Playground
- Online assignment

It should return a map of the counts of each “word” in the string `s`. The `wc.Test` function runs a test suite against the provided function and prints success or failure.

You might find strings.Fields helpful.

4.4.4 Solution

```
package main

import (
        "code.google.com/p/go-tour/wc"
```

```
        "strings"
)

func WordCount(s string) map[string]int {
        words := strings.Fields(s)
        count := map[string]int{}
        for _, word := range words {
                count[word]++
        }
        return count
}

func main() {
        wc.Test(WordCount)
}
```

- See in Playground

Chapter 5

Control flow

5.1 If statement

The `if` statement looks as it does in C or Java, except that the `( )` are gone and the `{ }` are required. Like `for`, the `if` statement can start with a short statement to execute before the condition. Variables declared by the statement are only in scope until the end of the `if`. Variables declared inside an if short statement are also available inside any of the else blocks.

- `If` statement example

```
if answer != 42 {
    return "Wrong answer"
}
```

- `If` with a short statement

```
if err := foo(); err != nil {
    panic(err)
}
```

- `If` and `else` example

5.2 For Loop

Go has only one looping construct, the for loop. The basic for loop looks as it does in C or Java, except that the ( ) are gone (they are not even optional) and the { } are required. As in C or Java, you can leave the pre and post statements empty.

- For loop example

```
sum := 0
for i := 0; i < 10; i++ {
    sum += i
}
```

- For loop without pre/post statements

```
sum := 1
for ; sum < 1000; {
    sum += sum
}
```

- For loop as a **while** loop

```
sum := 1
for sum < 1000 {
    sum += sum
}
```

- Infinite loop

```
for {
  // do something in a loop forever
}
```

5.3 Switch case statement

Most programming languages have some sort of switch case statement to allow developers to avoid doing complex and ugly series of `if else` statements.

```
package main

import (
        "fmt"
        "time"
)

func main() {
        now := time.Now().Unix()
        mins := now % 2
        switch mins {
        case 0:
                fmt.Println("even")
        case 1:
                fmt.Println("odd")
        }
}
```

- See in Playground

There are a few interesting things to know about this statement in Go:

- You can only compare values of the same type.
- You can set an optional default statement to be executed if all the others fail.
- You can use an expression in the case statement, for instance you can calculate a value to use in the case:

```
package main

import "fmt"

func main() {
        num := 3
```

```
        v := num % 2
        switch v {
        case 0:
                fmt.Println("even")
        case 3 - 2:
                fmt.Println("odd")
        }
}
```

- See in Playground
- You can have multiple values in a case statement:

```
package main

import "fmt"

func main() {
        score := 7
        switch score {
        case 0, 1, 3:
                fmt.Println("Terrible")
        case 4, 5:
                fmt.Println("Mediocre")
        case 6, 7:
                fmt.Println("Not bad")
        case 8, 9:
                fmt.Println("Almost perfect")
        case 10:
                fmt.Println("hmm did you cheat?")
        default:
                fmt.Println(score, " off the chart")
        }
}
```

- See in Playground
- You can execute all the following statements after a match using the **`fallthrough`** statement:

```
package main

import "fmt"

func main() {
        n := 4
        switch n {
        case 0:
                fmt.Println("is zero")
                fallthrough
        case 1:
                fmt.Println("is <= 1")
                fallthrough
        case 2:
                fmt.Println("is <= 2")
                fallthrough
        case 3:
                fmt.Println("is <= 3")
                fallthrough
        case 4:
                fmt.Println("is <= 4")
                fallthrough
        case 5:
                fmt.Println("is <= 5")
                fallthrough
        case 6:
                fmt.Println("is <= 6")
                fallthrough
        case 7:
                fmt.Println("is <= 7")
                fallthrough
        case 8:
                fmt.Println("is <= 8")
                fallthrough
        default:
                fmt.Println("Try again!")
        }
}
```

```
is <= 4
is <= 5
is <= 6
is <= 7
is <= 8
Try again!
```

- See in Playground

You can use a **break** statement inside your matched statement to exit the switch processing:

```
package main

import (
        "fmt"
        "time"
)

func main() {
        n := 1
        switch n {
        case 0:
                fmt.Println("is zero")
                fallthrough
        case 1:
                fmt.Println("<= 1")
                fallthrough
        case 2:
                fmt.Println("<= 2")
                fallthrough
        case 3:
                fmt.Println("<= 3")
                if time.Now().Unix()%2 == 0 {
                        fmt.Println("un pasito pa lante maria")
                        break
                }
                fallthrough
        case 4:
                fmt.Println("<= 4")
                fallthrough
        case 5:
                fmt.Println("<= 5")
        }
}
```

- See in Playground

```
<= 1
<= 2
<= 3
un pasito pa lante maria
```

5.4 Exercise

You have 50 bitcoins to distribute to 10 users: Matthew, Sarah, Augustus, Heidi, Emilie, Peter, Giana, Adriano, Aaron, Elizabeth The coins will be distributed based on the vowels contained in each name where:

a: 1 coin e: 1 coin i: 2 coins o: 3 coins u: 4 coins

and a user can't get more than 10 coins. Print a map with each user's name and the amount of coins distributed. After distributing all the coins, you should have 2 coins left.

The output should look something like that:

```
map[Matthew:2 Peter:2 Giana:4 Adriano:7 Elizabeth:5 Sarah:2 Augustus:10 Heidi:5 Emilie:6 Aaron:
Coins left: 2
```

Note that Go doesn't keep the order of the keys in a map, so your results might not look exactly the same but the key/value mapping should be the same.

Here is some starting code:

```
package main

import "fmt"

var (
        coins = 50
        users = []string{
                "Matthew", "Sarah", "Augustus", "Heidi", "Emilie",
                "Peter", "Giana", "Adriano", "Aaron", "Elizabeth",
        }
        distribution = make(map[string]int, len(users))
)

func main() {
        fmt.Println(distribution)
        fmt.Println("Coins left:", coins)
}
```

- See in Playground

5.5 Solution

```
package main

import "fmt"

var (
        coins = 50
        users = []string{
                "Matthew", "Sarah", "Augustus", "Heidi", "Emilie",
                "Peter", "Giana", "Adriano", "Aaron", "Elizabeth",
        }
        distribution = make(map[string]int, len(users))
)

func main() {
        coinsForUser := func(name string) int {
                var total int
                for i := 0; i < len(name); i++ {
                        switch string(name[i]) {
                        case "a", "A":
                                total++
                        case "e", "E":
                                total++
                        case "i", "I":
                                total = total + 2
                        case "o", "O":
                                total = total + 3
                        case "u", "U":
                                total = total + 4
                        }
                }
                return total
        }

        for _, name := range users {
                v := coinsForUser(name)
                if v > 10 {
                        v = 10
                }
                distribution[name] = v
                coins = coins - v
        }
        fmt.Println(distribution)
        fmt.Println("Coins left:", coins)
}
```

- See in Playground

Chapter 6

Methods

While technically Go isn't an Object Oriented Programming language, types and methods allow for an object-oriented style of programming. The big difference is that Go does not support type inheritance but instead has a concept of interface.

In this chapter, we will focus on Go's use of methods and interfaces.

Note: A frequently asked question is "what is the difference between a function and a method". A method is a function that has a defined receiver, in OOP terms, a method is a function on an instance of an object.

Go does not have classes. However, you can define methods on struct types.

The *method receiver* appears in its own argument list between the **`func`** keyword and the method name. Here is an example with a **`User`** struct containing two fields: **`FirstName`** and **`LastName`** of string type.

```
package main

import (
        "fmt"
)

type User struct {
        FirstName, LastName string
}

func (u User) Greeting() string {
        return fmt.Sprintf("Dear %s %s", u.FirstName, u.LastName)
}
```

```
func main() {
        u := User{"Matt", "Aimonetti"}
        fmt.Println(u.Greeting())
}
```

See in playground

Note how methods are defined outside of the struct, if you have been writing Object Oriented code for a while, you might find that a bit odd at first. The method on the `User` type could be defined anywhere in the package.

6.1 Code organization

Methods can be defined on any file in the package, but my recommendation is to organize the code as shown below:

```
package models

// list of packages to import
import (
        "fmt"
)

// list of constants
const (
        ConstExample = "const before vars"
)

// list of variables
var (
        ExportedVar    = 42
        nonExportedVar = "so say we all"
)

// Main type(s) for the file,
// try to keep the lowest amount of structs per file when possible.
type User struct {
        FirstName, LastName string
        Location            *UserLocation
}

type UserLocation struct {
        City    string
        Country string
```

```
}

// List of functions
func NewUser(firstName, lastName string) *User {
        return &User{FirstName: firstName,
                LastName: lastName,
                Location: &UserLocation{
                        City:    "Santa Monica",
                        Country: "USA",
                },
        }
}

// List of methods
func (u *User) Greeting() string {
        return fmt.Sprintf("Dear %s %s", u.FirstName, u.LastName)
}
```

In fact, you can define a method on **any** type you define in your package, not just structs. You cannot define a method on a type from another package, or on a basic type.

6.2 Type aliasing

To define methods on a type you don't "own", you need to define an alias for the type you want to extend:

```
package main

import (
        "fmt"
        "strings"
)

type MyStr string

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

func main() {
        fmt.Println(MyStr("test").Uppercase())
}
```

Playground Example

```
package main

import (
    "fmt"
    "math"
)

type MyFloat float64

func (f MyFloat) Abs() float64 {
    if f < 0 {
        return float64(-f)
    }
    return float64(f)
}

func main() {
    f := MyFloat(-math.Sqrt2)
    fmt.Println(f.Abs())
}
```

Playground Example

6.3 Method receivers

Methods can be associated with a named type (`User` for instance) or a pointer to a named type (`*User`). In the two type aliasing examples above, methods were defined on the value types (`MyStr` and `MyFloat`).

There are two reasons to use a pointer receiver. First, to avoid copying the value on each method call (more efficient if the value type is a large struct). The above example would have been better written as follows:

```
package main

import (
        "fmt"
)

type User struct {
        FirstName, LastName string
```

```
}

func (u *User) Greeting() string {
        return fmt.Sprintf("Dear %s %s", u.FirstName, u.LastName)
}

func main() {
        u := &User{"Matt", "Aimonetti"}
        fmt.Println(u.Greeting())
}
```

See in playground

Remember that Go passes everything by value, meaning that when `Greeting()` is defined on the value type, every time you call `Greeting()`, you are copying the `User` struct. Instead when using a pointer, only the pointer is copied (cheap).

The other reason why you might want to use a pointer is so that the method can modify the value that its receiver points to.

```
package main

import (
        "fmt"
        "math"
)

type Vertex struct {
        X, Y float64
}

func (v *Vertex) Scale(f float64) {
        v.X = v.X * f
        v.Y = v.Y * f
}

func (v *Vertex) Abs() float64 {
        return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
        v := &Vertex{3, 4}
        v.Scale(5)
        fmt.Println(v, v.Abs())
}
```

See in playground

In the example above, `Abs()` could be defined on the value type or the pointer since the method doesn't modify the receiver value (the vertex). However `Scale()` has to be defined on a pointer since it does modify the receiver. `Scale()` resets the values of the `X` and `Y` fields.

Go tour page

Chapter 7

Interfaces

An interface type is defined by a set of methods. A value of interface type can hold any value that implements those methods.

Here is a refactored version of our earlier example. This time we made the greeting feature more generic by defining a function called `Greet` which takes a param of interface type `Namer`. `Namer` is a new interface we defined which only defines one method: `Name()`. So `Greet()` will accept as param any value which has a `Name()` method defined.

To make our `User` struct implement the interface, we defined a `Name()` method. We can now call `Greet` and pass our pointer to `User` type.

```
package main

import (
        "fmt"
)

type User struct {
        FirstName, LastName string
}

func (u *User) Name() string {
        return fmt.Sprintf("%s %s", u.FirstName, u.LastName)
}

type Namer interface {
        Name() string
}
```

```
func Greet(n Namer) string {
        return fmt.Sprintf("Dear %s", n.Name())
}

func main() {
        u := &User{"Matt", "Aimonetti"}
        fmt.Println(Greet(u))
}
```

See in playground

We could now define a new type that would implement the same interface and our **`Greet`** function would still work.

```
package main

import (
        "fmt"
)

type User struct {
        FirstName, LastName string
}

func (u *User) Name() string {
        return fmt.Sprintf("%s %s", u.FirstName, u.LastName)
}

type Customer struct {
        Id       int
        FullName string
}

func (c *Customer) Name() string {
        return c.FullName
}

type Namer interface {
        Name() string
}

func Greet(n Namer) string {
        return fmt.Sprintf("Dear %s", n.Name())
}

func main() {
        u := &User{"Matt", "Aimonetti"}
        fmt.Println(Greet(u))
        c := &Customer{42, "Francesc"}
```

```
        fmt.Println(Greet(c))
}
```

See in playground

7.1 Interfaces are satisfied implicitly

A type implements an interface by implementing the methods.

There is no explicit declaration of intent.

Implicit interfaces decouple implementation packages from the packages that define the interfaces: neither depends on the other.

It also encourages the definition of precise interfaces, because you don't have to find every implementation and tag it with the new interface name.

```
package main

import (
        "fmt"
        "os"
)

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

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

type ReadWriter interface {
        Reader
        Writer
}

func main() {
        var w Writer

        // os.Stdout implements Writer
        w = os.Stdout

        fmt.Fprintf(w, "hello, writer\n")
}
```

See in playground
Package io defines Reader and Writer so you don't have to.

7.2 Errors

An error is anything that can describe itself as an error string. The idea is captured by the predefined, built-in interface type, error, with its single method, **`Error`**, returning a string:

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

The **`fmt`** package's various print routines automatically know to call the method when asked to print an error.

```
package main

import (
    "fmt"
    "time"
)

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)
    }
}
```

See in Playground

7.3 Exercise: Errors

Online assignment

Copy your `Sqrt` function from the earlier exercises (Section 5.4) and modify it to return an `error` value.

`Sqrt` should return a non-nil error value when given a negative number, as it doesn't support complex numbers.

Create a new type

```
type ErrNegativeSqrt float64
```

and make it an error by giving it a

```
func (e ErrNegativeSqrt) Error() string
```

method such that `ErrNegativeSqrt(-2).Error()` returns `cannot Sqrt negative number: -2`.

Note: a call to `fmt.Print(e)` inside the Error method will send the program into an infinite loop. You can avoid this by converting e first: `fmt.Print(float64(e` Why?

Change your `Sqrt` function to return an `ErrNegativeSqrt` value when given a negative number.

7.3.1 Solution

```
package main

import (
        "fmt"
)

type ErrNegativeSqrt float64

func (e ErrNegativeSqrt) Error() string {
        return fmt.Sprintf("cannot Sqrt negative number: %g", float64(e))
}

func Sqrt(x float64) (float64, error) {
        if x < 0 {
                return 0, ErrNegativeSqrt(x)
        }

        z := 1.0
        for i := 0; i < 10; i++ {
                z = z - ((z*z)-x)/(2*z)
        }
        return z, nil
}

func main() {
        fmt.Println(Sqrt(2))
        fmt.Println(Sqrt(-2))
}
```

See in playground

Tip: When doing an inferred declaration of a float, you can omit the decimal value and do the following:

```
z := 1.
// same as
// z := 1.0
```

Chapter 8

Concurrency

Concurrent programming is a large topic but it's also one of the most interesting aspects of the Go language.

Concurrent programming in many environments is made difficult by the subtleties required to implement correct access to shared variables. Go encourages a different approach in which shared values are passed around on channels and, in fact, never actively shared by separate threads of execution. Only one goroutine has access to the value at any given time. Data races cannot occur, by design. To encourage this way of thinking we have reduced it to a slogan:

Do not communicate by sharing memory; instead, share memory by communicating.

This approach can be taken too far. Reference counts may be best done by putting a mutex around an integer variable, for instance. But as a high-level approach, using channels to control access makes it easier to write clear, correct programs.

Although Go's approach to concurrency originates in Hoare's Communicating Sequential Processes (CSP), it can also be seen as a type-safe generalization of Unix pipes.

- Rob Pike's concurrency slides (IO 2012)
- Video of Rob Pike at IO 2012
- Video of Concurrency is not parallelism (Rob Pike)

8.1 Goroutines

A goroutine is a lightweight thread managed by the Go runtime.

```
go f(x, y, z)
```

starts a new goroutine running:

```
f(x, y, z)
```

The evaluation of `f`, `x`, `y`, and `z` happens in the current goroutine and the execution of `f` happens in the new goroutine.

Goroutines run in the same address space, so access to shared memory must be synchronized. The sync package provides useful primitives, although you won't need them much in Go as there are other primitives.

```
package main

import (
        "fmt"
        "time"
)

func say(s string) {
        for i := 0; i < 5; i++ {
                time.Sleep(100 * time.Millisecond)
                fmt.Println(s)
        }
}

func main() {
        go say("world")
        say("hello")
}
```

See in playground
Go tour page

8.2 Channels

Channels are a typed conduit through which you can send and receive values with the channel operator, `<-`.

```
ch <- v    // Send v to channel ch.
v := <-ch  // Receive from ch, and
           // assign value to v.
```

(The data flows in the direction of the arrow.)

Like maps (Section 4.4) and slices (Section 4.2), channels must be created before use:

```
ch := make(chan int)
```

By default, sends and receives block wait until the other side is ready. This allows goroutines to synchronize without explicit locks or condition variables.

```
package main

import "fmt"

func sum(a []int, c chan int) {
        sum := 0
        for _, v := range a {
                sum += v
        }
        c <- sum // send sum to c
}

func main() {
        a := []int{7, 2, 8, -9, 4, 0}

        c := make(chan int)
        go sum(a[:len(a)/2], c)
        go sum(a[len(a)/2:], c)
        x, y := <-c, <-c // receive from c

        fmt.Println(x, y, x+y)
}
```

See in playground
Go tour page

8.2.1 Buffered channels

Channels can be *buffered*. Provide the buffer length as the second argument to `make` to initialize a buffered channel:

```
ch := make(chan int, 100)
```

Sends to a buffered channel block only when the buffer is full. Receives block when the buffer is empty.

```
package main

import "fmt"

func main() {
    c := make(chan int, 2)
    c <- 1
    c <- 2
    fmt.Println(<-c)
    fmt.Println(<-c)
}
```

See in playground

But if you do:

```
package main

import "fmt"

func main() {
        c := make(chan int, 2)
        c <- 1
        c <- 2
        c <- 3
        fmt.Println(<-c)
        fmt.Println(<-c)
        fmt.Println(<-c)
}
```

See in playground

You are getting a deadlock:

```
fatal error: all goroutines are asleep - deadlock!
```

That's because we overfilled the buffer without letting the code a chance to read/remove a value from the channel.

However, this version using a goroutine would work fine:

```
package main

import "fmt"

func main() {
        c := make(chan int, 2)
        c <- 1
        c <- 2
        c3 := func() { c <- 3 }
        go c3()
        fmt.Println(<-c)
        fmt.Println(<-c)
    fmt.Println(<-c)
}
```

See in playground

The reason is that we are adding an extra value from inside a go routine, so our code doesn't block the main thread. The goroutine is being called before the channel is being emptied, but that is fine, the goroutine will wait until the channel is available. We then read a first value from the channel, which frees a spot and our goroutine can push its value to the channel.

Go tour page

8.3 Range and close

A sender can close a channel to indicate that no more values will be sent. Receivers can test whether a channel has been closed by assigning a second parameter to the receive expression: after

```
v, ok := <-ch
```

`ok` is false if there are no more values to receive and the channel is closed.

The loop `for i := range ch` receives values from the channel repeatedly until it is closed.

Note: Only the sender should close a channel, never the receiver. Sending on a closed channel will cause a panic.

Another note: Channels aren't like files; you don't usually need to close them. Closing is only necessary when the receiver must be told there are no more values coming, such as to terminate a range loop.

```
package main

import (
    "fmt"
)

func fibonacci(n int, c chan int) {
    x, y := 0, 1
    for i := 0; i < n; i++ {
        c <- x
        x, y = y, x+y
    }
    close(c)
}

func main() {
    c := make(chan int, 10)
    go fibonacci(cap(c), c)
    for i := range c {
        fmt.Println(i)
    }
}
```

See in playground

Go tour page

8.4 Select

The **select** statement lets a goroutine wait on multiple communication operations.

A **select** blocks until one of its cases can run, then it executes that case. It chooses one at random if multiple are ready.

```
package main

import "fmt"

func fibonacci(c, quit chan int) {
    x, y := 0, 1
    for {
        select {
        case c <- x:
            x, y = y, x+y
        case <-quit:
            fmt.Println("quit")
            return
        }
    }
}

func main() {
    c := make(chan int)
    quit := make(chan int)
    go func() {
        for i := 0; i < 10; i++ {
            fmt.Println(<-c)
        }
        quit <- 0
    }()
    fibonacci(c, quit)
}
```

See in playground

8.4.1 Default case

The default case in a **select** is run if no other case is ready.

Use a **default** case to try a send or receive without blocking:

```
select {
case i := <-c:
    // use i
default:
    // receiving from c would block
}
```

```
package main

import (
    "fmt"
    "time"
)

func main() {
    tick := time.Tick(100 * time.Millisecond)
    boom := time.After(500 * time.Millisecond)
    for {
        select {
        case <-tick:
            fmt.Println("tick.")
        case <-boom:
            fmt.Println("BOOM!")
            return
        default:
            fmt.Println("    .")
            time.Sleep(50 * time.Millisecond)
        }
    }
}
```

See in playground
Go tour page

8.4.2 Timeout

```
package main

import (
        "fmt"
        "log"
        "net/http"
        "time"
```

```
)

func main() {
        response := make(chan *http.Response, 1)
        errors := make(chan *error)

        go func() {
                resp, err := http.Get("http://matt.aimonetti.net/")
                if err != nil {
                        errors <- &err
                }
                response <- resp
        }()
        for {
                select {
                case r := <-response:
                        fmt.Printf("%s", r.Body)
                        return
                case err := <-errors:
                        log.Fatal(*err)
                case <-time.After(200 * time.Millisecond):
                        fmt.Printf("Timed out!")
                        return
                }
        }
}
```

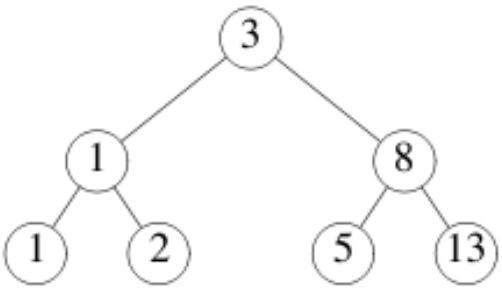
See in playground but note that in playground, you won't get a response due to sandboxing.

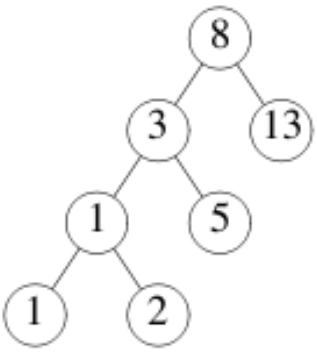
We are using the **`time.After`** call as a timeout measure to exit if the request didn't give a response within 200ms.

8.5 Exercise: Equivalent Binary Trees

Online Assignment

There can be many different binary trees with the same sequence of values stored at the leaves. For example, here are two binary trees storing the sequence 1, 1, 2, 3, 5, 8, 13.





A function to check whether two binary trees store the same sequence is quite complex in most languages. We'll use Go's concurrency and channels to write a simple solution.

This example uses the `tree` package, which defines the type:

```
type Tree struct {
    Left  *Tree
    Value int
    Right *Tree
}
```

1. Implement the Walk function.

2. Test the Walk function.

The function `tree.New(k)` constructs a randomly-structured binary tree holding the values `k`, `2k`, `3k`, ..., `10k`.

Create a new channel `ch` and kick off the walker:

```
go Walk(tree.New(1), ch)
```

Then read and print 10 values from the channel. It should be the numbers `1`, `2`, `3`, ..., `10`.

1. Implement the Same function using `Walk` to determine whether `t1` and `t2` store the same values.
2. Test the `Same` function.

`Same(tree.New(1), tree.New(1))` should return `true`, and `Same(tree.New(1) tree.New(2))` should return `false`.

8.5.1 Solution

If you print `tree.New(1)` you will see the following tree:

```
((((1 (2)) 3 (4)) 5 ((6) 7 ((8) 9))) 10)
```

To implement the `Walk` function, we need two things:

- walk each side of the tree and print the values
- close the channel so the `range` call isn't stuck.

We need to set a recursive call and for that, we are defining a non-exported `recWalk` function, the function walks the left side first, then pushes the value to the channel and then walks the right side. This allows our range to get the values in the right order. Once all branches have been walked, we can close the channel to indicate to the range that the walking is over.

```
package main

import (
        "code.google.com/p/go-tour/tree"
        "fmt"
)

// Walk walks the tree t sending all values
// from the tree to the channel ch.
func Walk(t *tree.Tree, ch chan int) {
        recWalk(t, ch)
        // closing the channel so range can finish
```

```
        close(ch)
}

// recWalk walks recursively through the tree and push values to the channel
// at each recursion
func recWalk(t *tree.Tree, ch chan int) {
        if t != nil {
                // send the left part of the tree to be iterated over first
                recWalk(t.Left, ch)
                // push the value to the channel
                ch <- t.Value
                // send the right part of the tree to be iterated over last
                recWalk(t.Right, ch)
        }
}

// Same determines whether the trees
// t1 and t2 contain the same values.
func Same(t1, t2 *tree.Tree) bool {
        ch1 := make(chan int)
        ch2 := make(chan int)
        go Walk(t1, ch1)
        go Walk(t2, ch2)

        for {
                x1, ok1 := <-ch1
                x2, ok2 := <-ch2
                switch {
                case ok1 != ok2:
                        // not the same size
                        return false
                case !ok1:
                        // both channels are empty
                        return true
                case x1 != x2:
                        // elements are different
                        return false
                default:
                        // keep iterating
                }
        }
}

func main() {
        ch := make(chan int)
        go Walk(tree.New(1), ch)
        for v := range ch {
                fmt.Println(v)
        }
        fmt.Println(Same(tree.New(1), tree.New(1)))
        fmt.Println(Same(tree.New(1), tree.New(2)))
```

```
}
```

The comparison of the two trees is trivial once we know how to extract the values of each tree. We just need to loop through the first tree (via the channel), read the value, get the value from the second channel (walking the second tree) and compare the two values.

See in playground

Chapter 9

Get Setup

9.1 OS X

The easiest way to install Go for development on OS X is to use homebrew.

Using **Terminal.app** install homebrew:

```
$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Hom
```

Once homebrew is installed, install Go to be able to crosscompile:

```
$ brew install go --cross-compile-common
```

In just a few minutes, go should be all installed and you should be almost ready to code. However we need to do two small things before we start:

9.1.1 Setup your paths

By convention, all your Go code and the code you will import, will live inside a workspace. This convention might seem rigid at first, but it quickly becomes clear that such a convention (like most Go conventions) makes our life much easier.

Before starting, we need to tell Go, where we want our workspace to be, in other words, where our code will live. Let's create a folder named "go" in our home directory and set our environment to use this location.

```
$ mkdir $HOME/go
$ export GOPATH=$HOME/go
```

Note that if we open a new tab or restart our machine, Go won't know where to find our workspace. For that, you need to set the export in your profile:

```
$ open $HOME/.bash_profile
```

Add a new entry to set **GOPATH** and add the workspace's bin folder to your system path:

```
export GOPATH=$HOME/go
export PATH=$PATH:$GOPATH/bin
```

This will allow your Go workspace to always be set and will allow you to call the binaries you compiled.

Official resource

9.1.2 Install mercurial and bazaar

Optionally but highly recommended, you should install mercurial and bazaar so you can retrieve 3rd party libraries hosted using these version control systems.

```
$ brew install hg bzr
```

9.2 Windows

Install the latest version by downloading the latest installer.

Official resource

9.3 Linux

Install from one of the official linux packages Setup your path, as explained in Section 9.1.1

9.4 Extras

Installing **Godoc**, **vet** and **Golint**, three very useful Go tools from the Go team, is highly recommended:

```
$ go get golang.org/x/tools/cmd/godoc
$ go get golang.org/x/tools/cmd/vet
$ go get github.com/golang/lint/golint
```

Official resource

Chapter 10

Get Your Feet Wet

One of the best way to learn technical skills is to actually dive in as soon as we have acquired the basics.

The following code was written by someone who just started learning Go. Beginners often make the same mistakes so assume that this is your code and you are now tasked to refactor it. The end goal is to create a CLI to interface with the Pivotal Tracker API.

Cli Rescue repo

Fork the project as explained in the readme

Find a one or more people and work with them to see how you would address this refactoring. Time to rescue this project!


BAYWATCH

Chapter 11

Tips and Tricks

This section will grow over time but the main goal is to share some tricks experienced developers discovered over time. Hopefully this tips will get new users more productive faster.

11.1 140 char tips

- leave your object oriented brain at home. Embrace the interface. @mikegehard
- Learn to do things the Go way, don't try to force your language idioms into Go. @DrNic
- It's better to over do it with using interfaces than use too few of them. @evanphx
- Embrace the language: simplicity, concurrency, and composition. @francesc
- read all the awesome docs that they have on golang.org. @vbatts
- always use `gofmt`. @darkhelmetlive
- read a lot of source code. @DrNic

- Learn and become familiar with tools and utilities, or create your own! They are as vital to your success as knowing the language. @coreyprak

11.2 Alternate Ways to Import Packages

There are a few other ways of importing packages. We'll use the `fmt` package in the following examples:

- `import format "fmt"` - Creates an alias of `fmt`. Preceed all `fmt` package content with `format.` instead of `fmt.`.
- `import . "fmt"` - Allows content of the package to be accessed directly, without the need for it to be preceded with `fmt`.
- `import _ "fmt"` - Suppresses compiler warnings related to `fmt` if it is not being used, and executes initialization functions if there are any. The remainder of `fmt` is inaccessible.

See this blog post for more detailed information.

11.3 goimports

Goimports is a tool that updates your Go import lines, adding missing ones and removing unreferenced ones.

It acts the same as gofmt (drop-in replacement) but in addition to code formatting, also fixes imports.

11.4 Organization

Go is a pretty easy programming language to learn but the hardest thing for developers at first is how to organize their code. Rails became popular for many reasons and scaffolding was one of them. It gave new developers clear directions and places to put their code and idioms to follow.

To some extent, Go does the same thing by providing developers with great tools like `go fmt` and by having a strict compiler that won't compile unused variables or unused import statements.

11.5 Custom Constructors

A question I often hear is when should I use custom constructors like `NewJob`. My answer is that in most cases you don't need to. However, whenever you need to set your value at initialization time and you have some sort of default values, it's a good candidate for a constructor. In the above example, adding a constructor makes a lot of sense so we can set a default logger.

```
package main

import (
        "log"
        "os"
)

type Job struct {
        Command string
        *log.Logger
}

func NewJob(command string) *Job {
        return &Job{command, log.New(os.Stderr, "Job: ", log.Ldate)}
}

func main() {
        NewJob("demo").Print("starting now...")
}
```

11.6 Breaking down code in packages

See this blog post on refactoring Go code, the first part talks about package organization.

11.7 Sets

You might want to find a way to extract unique value from a collection. In other languages, you often have a set data structure not allowing duplicates. Go doesn't have that built in, however it's not too hard to implement (due to a lack of generics, you do need to do that for most types, which can be cumbersome).

```
// UniqStr returns a copy if the passed slice with only unique string results.
func UniqStr(col []string) []string {
        m := map[string]struct{}{}
        for _, v := range col {
                if _, ok := m[v]; !ok {
                        m[v] = struct{}{}
                }
        }
        list := make([]string, len(m))

        i := 0
        for v := range m {
                list[i] = v
                i++
        }
        return list
}
```

See in playground

I used a few interesting tricks that are interesting to know. First, the map of empty structs:

```
m := map[string]struct{}{}
```

We create a map with the keys being the values we want to be unique, the associated value doesn't really matter much so it could be anything. For instance:

```
m := map[string]bool{}
```

However I chose an empty structure because it will be as fast as a boolean but doesn't allocate as much memory.

The second trick can been seen a bit further:

```
if _, ok := m[v]; !ok {
  m[v] = struct{}{}
}
```

What we are doing here, is simply check if there is a value associated with the key **v** in the map **m**, we don't care about the value itself, but if we know that we don't have a value, then we add one.

Once we have a map with unique keys, we can extract them into a new slice of strings and return the result.

Here is the test for this function, as you can see, I used a table test, which is the idiomatic Go way to run unit tests:

```
func TestUniqStr(t *testing.T) {

        data := []struct{ in, out []string }{
                {[]string{}, []string{}},
                {[]string{"", "", ""}, []string{""}},
                {[]string{"a", "a"}, []string{"a"}},
                {[]string{"a", "b", "a"}, []string{"a", "b"}},
                {[]string{"a", "b", "a", "b"}, []string{"a", "b"}},
                {[]string{"a", "b", "b", "a", "b"}, []string{"a", "b"}},
                {[]string{"a", "a", "b", "b", "a", "b"}, []string{"a", "b"}},
                {[]string{"a", "b", "c", "a", "b", "c"}, []string{"a", "b", "c"}},
        }

        for _, exp := range data {
                res := UniqStr(exp.in)
                if !reflect.DeepEqual(res, exp.out) {
                        t.Fatalf("%q didn't match %q\n", res, exp.out)
                }
        }

}
```

See in the playground

11.8 Dependency package management

Unfortunately, Go doesn't ship with its own dependency package management system. Probably due to its roots in the C culture, packages aren't versioned and explicit version dependencies aren't addressed.

The challenge is that if you have multiple developers on your project, you want all of them to be on the same version of your dependencies. Your dependencies might also have their own dependencies and you want to make sure everything is in a good state. It gets even tricker when you have multiple projects using different versions of the same dependency. This is typically the case in a CI environment.

The Go community came up with a lot of different solutions for these problems. But for me, none are really great so at Splice we went for the simplest working solution we found: gpm

Gpm is a simple bash script, we end up modifying it a little so we could drop the script in each repo. The bash script uses a custom file called `Godeps` which lists the packages to install.

When switching to a different project, we run the project `gpm` script to pull down or set the right revision of each package.

In our CI environment, we set `GOPATH` to a project specific folder before running the test suite so packages aren't shared between projects.

11.9 Using errors

Errors are very important pattern in Go and at first, new developers are surprised by the amount of functions returning a value and an error.

Go doesn't have a concept of an exception like you might have seen in other programming languages. Go does have something called `panic` but as its name suggests they are really exceptional and shouldn't be rescued (that said, they can be).

The error handling in Go seems cumbersome and repetitive at first, but quickly becomes part of the way we think. Instead of creating exceptions that bubble up and might or might not be handled or passed higher, errors are part

of the response and designed to be handled by the caller. Whenever a function might generate an error, its response should contain an error param.

Andrew Gerrand from the Go team wrote a great blog post on errors I strongly recommend you read it.

Effective Go section on errors

11.10 Quick look at some compiler's optimizations

You can pass specific compiler flags to see what optimizations are being applied as well as how some aspects of memory management. This is an advanced feature, mainly for people who want to understand some of the compiler optimizations in place.

Let's take the following code example from an earlier chapter:

```
package main

import "fmt"

type User struct {
        Id              int
        Name, Location string
}

func (u *User) Greetings() string {
        return fmt.Sprintf("Hi %s from %s",
                u.Name, u.Location)
}

func NewUser(id int, name, location string) *User {
        id++
        return &User{id, name, location}
}

func main() {
        u := NewUser(42, "Matt", "LA")
        fmt.Println(u.Greetings())
}
```

- See in Playground

Build your file (here called `t.go`) passing some `gcflags`:

```
$ go build -gcflags=-m t.go
# command-line-arguments
./t.go:15: can inline NewUser
./t.go:21: inlining call to NewUser
./t.go:10: leaking param: u
./t.go:10: leaking param: u
./t.go:12: (*User).Greetings ... argument does not escape
./t.go:15: leaking param: name
./t.go:15: leaking param: location
./t.go:17: &User literal escapes to heap
./t.go:15: leaking param: name
./t.go:15: leaking param: location
./t.go:21: &User literal escapes to heap
./t.go:22: main ... argument does not escape
```

The compiler notices that it can inline the **`NewUser`** function defined on line 15 and inline it on line 21. Dave Cheney has a great post about why Go's inlining is helping your programs run faster.

Basically, the compiler moves the body of the **`NewUser`** function (L15) to where it's being called (L21) and therefore avoiding the overhead of a function call but increasing the binary size.

The compiler creates the equivalent of:

```
func main() {
        id := 42 + 1
        u := &User{id, "Matt", "LA"}
        fmt.Println(u.Greetings())
}
```

On a few lines, you see the potentially alarming **`leaking param`** message. It doesn't mean that there is a memory leak but that the param is kept alive even after returning. The "leaked params" are:

- On the **`Greetings`**'s method: **`u`** (receiver)
- On the **`NewUser`**'s functon: **`name`**, **`location`**

The reason why **`u`** "leaks" in the **`Greetings`** method is because it's being used in the **`fmt.Sprintf`** function call as an argument. **`name`** and **`location`**

are also "leaked" because they are used in the `User`'s literal value. Note that `id` doesn't leak because it's a value, only references and pointers can leak.

X `argument does not escape` means that the argument doesn't "escape" the function, meaning that it's not used outside of the function so it's safe to store it on the stack.

On the other hand, you can see that `&User literal escapes to heap`. What it means is that the address of a literal value is used outside of the function and therefore can't be stored on the stack. The value *could* be stored on the stack, except a pointer to the value escapes the function, so the value has to be moved to the heap to prevent the pointer referring to incorrect memory once the function returns. This is always the case when calling a method on a value and the method uses one or more fields.

11.11 Expvar

TODO package

11.12 Set the build id using git's SHA

It's often very useful to burn a build id in your binaries. I personally like to use the SHA1 of the git commit I'm committing. You can get the short version of the sha1 of your latest commit by running the following `git` command from your repo:

```
git rev-parse --short HEAD
```

The next step is to set an exported variable that you will set at compilation time using the `-ldflags` flag.

```
package main

import "fmt"
```

```
// compile passing -ldflags "-X main.Build <build sha1>"
var Build string

func main() {
        fmt.Printf("Using build: %s\n", Build)
}
```

See in playground

Save the above code in a file called **example.go**. If you run the above code, **Build** won't be set, for that you need to set it using **go build** and the **-ldflags**.

```
$ go build -ldflags "-X main.Build a1064bc" example.go
```

Now run it to make sure:

```
$ ./example
Using build: a1064bc
```

Now, hook that into your deployment compilation process, I personally like Rake to do that, and this way, every time I compile, I think of Jim Weirich.

11.13 How to see what packages my app imports

It's often practical to see what packages your app is importing. Unfortunately there isn't a simple way to do that, however it is doable via the **go list** tool and using templates.

Go to your app and run the following.

```
$ go list -f '{{join .Deps "\n"}}' |
  xargs go list -f '{{if not .Standard}}{{.ImportPath}}{{end}}'
```

Here is an example with the clirescue refactoring example:

```
$ cd $GOPATH/src/github.com/GoBootcamp/clirescue
$ go list -f '{{join .Deps "\n"}}' |
  xargs go list -f '{{if not .Standard}}{{.ImportPath}}{{end}}'
github.com/GoBootcamp/clirescue/cmdutil
github.com/GoBootcamp/clirescue/trackerapi
github.com/GoBootcamp/clirescue/user
github.com/codegangsta/cli
```

If you want the list to also contain standard packages, edit the template and use:

```
$ go list -f '{{join .Deps "\n"}}' |  xargs go list -f '{{.ImportPath}}'
```

11.14 Iota: Elegant Constants

Some concepts have names, and sometimes we care about those names, even (or especially) in our code.

```
const (
    CCVisa            = "Visa"
    CCMasterCard      = "MasterCard"
    CCAmericanExpress = "American Express"
)
```

At other times, we only care to distinguish one thing from the other. There are times when theres no inherently meaningful value for a thing. For example, if were storing products in a database table we probably dont want to store their category as a string. We dont care how the categories are named, and besides, marketing changes the names all the time.

We care only that theyre distinct from each other.

```
const (
    CategoryBooks    = 0
    CategoryHealth   = 1
    CategoryClothing = 2
)
```

Instead of 0, 1, and 2 we could have chosen 17, 43, and 61. The values are arbitrary.

Constants are important but they can be hard to reason about and difficult to maintain. In some languages like Ruby developers often just avoid them. In Go, constants have many interesting subtleties that, when used well, can make the code both elegant and maintainable.

11.14.1 Auto Increment

A handy idiom for this in golang is to use the iota identifier, which simplifies constant definitions that use incrementing numbers, giving the categories exactly the same values as above.

```
const (
    CategoryBooks = iota // 0
    CategoryHealth       // 1
    CategoryClothing     // 2
)
```

11.14.2 Custom Types

Auto-incrementing constants are often combined with a custom type, allowing you to lean on the compiler.

```
type Stereotype int

const (
    TypicalNoob Stereotype = iota // 0
    TypicalHipster                // 1
    TypicalUnixWizard             // 2
    TypicalStartupFounder         // 3
)
```

If a function is defined to take an int as an argument rather than a Stereotype, it will blow up at compile-time if you pass it a Stereotype:

```
func CountAllTheThings(i int) string {
                return fmt.Sprintf("there are %d things", i)
}

func main() {
    n := TypicalHipster
    fmt.Println(CountAllTheThings(n))
}

// output:
// cannot use TypicalHipster (type Stereotype) as type int in argument to CountAllTheThings
```

The inverse is also true. Given a function that takes a Stereotype as an argument, you cant pass it an int:

```
func SoSayethThe(character Stereotype) string {
    var s string
    switch character {
    case TypicalNoob:
        s = "I'm a confused ninja rockstar."
    case TypicalHipster:
        s = "Everything was better we programmed uphill and barefoot in the snow on the SUTX 59
    case TypicalUnixWizard:
        s = "sudo grep awk sed %!#?!!1!"
    case TypicalStartupFounder:
        s = "exploit compelling convergence to syndicate geo-targeted solutions"
    }
    return s
}

func main() {
    i := 2
    fmt.Println(SoSayethThe(i))
}
```

// output: // cannot use i (type int) as type Stereotype in argument to SoSayethThe

```
Theres a dramatic twist, however. You could pass a number constant, and it would work:

```go
func main() {
    fmt.Println(SoSayethThe(0))
}

// output:
```
```

```
// I'm a confused ninja rockstar.
```

This is because constants in Go are loosely typed until they are used in a strict context.

Skipping Values

Imagine that youre dealing with consumer audio output. The audio might not have any output whatsoever, or it could be mono, stereo, or surround.

Theres some underlying logic to defining no output as 0, mono as 1, and stereo as 2, where the value is the number of channels provided.

So what value do you give Dolby 5.1 surround?

On the one hand, its 6 channel output, but on the other hand, only 5 of those channels are full bandwidth channels (hence the 5.1 designation - with the .1 referring to the low-frequency effects channel).

Either way, we dont want to simply increment to 3.

We can use underscores to skip past the unwanted values.

```go
type AudioOutput int

const (
    OutMute AudioOutput = iota // 0
    OutMono                    // 1
    OutStereo                  // 2
    _
    _
    OutSurround                // 5
)
```

Expressions

The iota can do more than just increment. Or rather, iota always increments, but it can be used in expressions, storing the resulting value in the constant.

Here were creating constants to be used as a bitmask.

```go
type Allergen int

const (
    IgEggs Allergen = 1 << iota // 1 << 0 which is 00000001
    IgChocolate                 // 1 << 1 which is 00000010
```

```
    IgNuts                      // 1 << 2 which is 00000100
    IgStrawberries              // 1 << 3 which is 00001000
    IgShellfish                 // 1 << 4 which is 00010000
)
```

This works because when you have only an identifier on a line in a const group, it will take the previous expression and reapply it, with the incremented iota. In the language of [the spec](http://golang.org/ref/spec#Iota), this is called _implicit repetition of the last non-empty expression list_.

If youre allergic to eggs, chocolate, and shellfish, and flip those bits to the on position (mapping the bits right to left), then you get a bit value of 00010011, which corresponds to 19 in decimal.

```go
fmt.Println(IgEggs | IgChocolate | IgShellfish)

// output:
// 19
```

Theres a great example in [Effective Go](https://golang.org/doc/effective_go.html#constants) for defining orders of magnitude:

```go
type ByteSize float64

const (
    _           = iota              // ignore first value by assigning to blank identifier
    KB ByteSize = 1 << (10 * iota)  // 1 << (10*1)
    MB                              // 1 << (10*2)
    GB                              // 1 << (10*3)
    TB                              // 1 << (10*4)
    PB                              // 1 << (10*5)
    EB                              // 1 << (10*6)
    ZB                              // 1 << (10*7)
    YB                              // 1 << (10*8)
)
```

Today I learned that after zettabyte we get yottabyte. #TIL

But Wait, There's More

What happens if you define two constants on the same line?
What is the value of Banana? 2 or 3? And what about Durian?

```go
```

```
const (
    Apple, Banana = iota + 1, iota + 2
    Cherimoya, Durian
    Elderberry, Fig
)
```

The iota increments on the next line, rather than as soon as it gets referenced.

```go
// Apple: 1
// Banana: 2
// Cherimoya: 2
// Durian: 3
// Elderberry: 3
// Fig: 4
```

Which is messed up, because now you have constants with the same value.

So, Yeah

There's a lot more to be said about constants in Go, and you should probably
read Rob Pike's [blog post on the subject](http://blog.golang.org/constants)
over on the golang blog.

_This was first published by Katrina Owen on the [Splice
blog](https://splice.com/blog/iota-elegant-constants-golang/)._

Web resources

* [Dave Cheney](https://twitter.com/davecheney) maintains a [list of resources](http://dave.che

Chapter 12

Exercises

We have 4 exercises to choose from, each exercise focuses on different challenges. Try to tackle the exercise of your choice by pairing with at least one person, ideally try to have 4 people by group.

Fork the original repo (with the instructions), work on it and send a pull request when you are done.

If your group wants to work on a new exercise, please create an assignment and send it to me (Matt) so I can add it to the repository and the following list.

- avatar me
  - Hashing
  - image manipulation
- remote commands
  - Concurrency (channels, go routines)
  - Network interface
  - depending on the commands you implement
- copernic 2000
  - concurrency

  - consumption of web resources (http/json/XML)
  - sorting
  - data sets
  - data storage

- Godoc API
  - Building a web API
  - Testing a web API
  - JSON encoding/decoding
  - Exploring godoc
  - Extending existing packages or
  - Shelling out