Introduction to Go

Ravi Jaya

Agenda

This workshop is aimed at programmers with experience in another programming language and want to learn how to apply their skills to Go.

This workshop consists of five main sections:

- Basic Syntax
- Advanced Syntax
- Development environment and tooling
- The standard library
- Packages and applications

After each section we'll have time for questions.

As we work through the day there will be less of me talking, and more exercises to help you learn Go via doing.

The Go programming language

- Modern
- Compact, concise, general-purpose
- Imperative, statically type-checked, dynamically type-safe
- Garbage-collected
- Compiles to native code, statically linked
- Fast compilation, efficient execution

Designed by programmers for programmers!

Source: Robert Griesemer, A Taste of Go. (https://talks.golang.org/2014/taste.slide#2)

3

Speed

Go is often noted for its fast compilation, but this is only one facet of the idea of Go being a *fast* language to program in.

"False dichotomy of static languages being 'slow and painful' and dynamic languages being 'fast and fun'" -Rob Pike

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

Go is an attempt to demonstrate that a language can be both fast in terms of the programs it produces and fast (productive?) for the programmers.

Safety

- Typed, and type safe

```
var i int = -1
var u uint = 200
i = u   // nope, incompatible types
```

- Array accesses are bounds checked

```
s := make([]string, 10)
x := s[20] // will panic at runtime
```

- All memory is zeroed if not explicitly initialised

```
var q int       // initialised to 0
var f = 7       // initialised to 7, type defaults to int
```

- No implicit conversions; booleans and integers are not aliases

```
i := 2
if i { ... }    // nope, no coercion to bool
```

5

Good support for concurrency and communication

- Multicore CPUs are a reality.
- Multiprocessing is not a solution.
- Networking support baked into the standard library, integrated into the runtime.

6

Garbage collected

Go is a garbage collected language.

- Eliminates the bookkeeping errors related to ownership of shared values.
- Eliminates an entire class of use after free and memory leak bugs.
- Enables simpler, cleaner, APIs.

The garbage collector handles heaps into the 100's of GB range, and is designed for extremely low "stop the world" pauses.

In Go 1.7 these pauses are now less than 100 microseconds.

7

Opinionated

Go is an opinionated language.

- Unused local variables are an error.
- Unused imports are also an error.
- The compiler does not issue warnings, only errors.
- A single way to format code as defined by `go fmt`.

8

Hello, http!

```
package main

import (
    "html/template"
    "log"
    "net/http"
)

const templ = `<html><head><title>Hello</title></head><body>
Hello {{ .RemoteAddr }}
You sent me a {{ .Method }} request for {{ .URL }}
</body></html>`

func HelloServer(w http.ResponseWriter, req *http.Request) {
    log.Println(req.URL)
    t := template.Must(template.New("html").Parse(templ))
    t.Execute(w, req)
}

func main() {
    log.Println("please connect to http://localhost:7777/")
    http.HandleFunc("/", HelloServer)
    log.Fatal(http.ListenAndServe(":7777", nil))
}
```

Run

Basic Syntax

10

Basic Syntax

In this section we will walk through the basic syntax of a short Go program.

For this section all the code exercises will be inside this slide deck itself.

At the end of this section you'll know:

- How to declare constants and variables
- How to write `for` loops and use `if`.
- How types work.
- How to write your own functions.
- How packages and `import` statements work.

11

Constants

A good place to start talking about Go are *Constants*.

Constants are values that do not change.

Here are some examples of constants:

```
1
"hello"
false
1.3
```

These are called *literal constants* because the value of evaluating each of these is the constant itself.

12

Types of constants

There are six kinds of constants

- integer constants: `1`, `0`, `-6`, `99999999999999999`, ...
- floating point constants: `3.14`, `7.5e-2`, ...
- complex number constants (rare): `1 - 0.707i`, ...
- string constants: `"Hello, 東京"`, ...
- rune constants: `'a'`, `'す'`, `'シ'`, `'1'`, ...
- boolean constants: `true`, `false`

13

const

To make a constant, we declare one with the `const` keyword.

```
const name = "David"
println(name)
```

Run

Whenever you see a **Run** button on a slide, you can execute the code.

- Press the **Run** button now and see what happens.
- Press the **Close** button to close the output window.

You can also edit the code right here in the slide.

- Edit the code to replace my name, with yours.
- All source code in Go is UTF-8, you can use Kanji.
- Press the **Run** button to see the results.
- If you make a mistake, just reload the page.

14

Identifiers

This is a good time to talk about *identifiers*.

An identifier is a name your give to something in your code.

In Go an identifier is any word that starts with a *letter*.

```
const students = 22
const country = "日本"
println(students, country)
```

Run

A letter is anything that Unicode defines to be a letter, this includes Kanji, Cyrillic, Latin, etc.

- Identifiers are not restricted to ASCII, edit the slide and change `students` to `学生の` and press **Run**.
- If you make a mistake, just reload the page.

15

Identifiers (cont.)

An identifier must start with a Unicode letter, or the underscore character, _.

Numbers are not permitted at the start of an identifier, but can appear later in the identifier.

```
const 1fruit = "apple"
const 五vehicle = 20
```

Run

Here are some examples of *invalid* identifiers.

- Change the name of the identifier so that the program compiles.
- Press the **Run** button to see the results.
- If you make a mistake, just reload the page.

16

Comments

Go supports two kinds of comments:

- Inline comments, which start with a double forward slash, `//`.
- Block comments, which start with a forward slash and a star, `/*`, and end with a star and forward slash, `*/`.

```
// const a = 1
/*
const b = 2
const c = 3
*/

println(a, b, c)
```

Run

Code that is commented out will not be compiled or run.

- Remove the comments in this program so it compiles.
- Press the **Run** button to see the results, it should compile and print 1 2 3.
- If you make a mistake, just reload the page.

17

Declarations

There are six kinds of *declarations* in Go, we've seen one of them already.

They are:

- `const`: declares a new constant.
- `var`: declares a new variable.
- `type`: declares a new type.
- `func`: declares a new function, or method.
- `package`: declares the package this `.go` source file belongs to.
- `import`: declares that this package imports declarations from another.

We'll cover each of the remaining five declarations in this section.

18

Variables

A variable holds a value that can be changed over time.

You *declare* a new variable with the `var` declaration.

```
var π = 3.14159
var radius = 6371.0 // radius of the Earth in km
var circumference = 2 * π * radius

println("Circumference of the earth is", circumference, "km")
```

Run

Just like constants, variable *identifiers* can be any any valid Unicode word.

- Press the **Run** button to see the result.

note: The circumference is printed in scientific notation 😕, that's ok, we'll talk about how to fix it later when we talk about the `fmt` package.

19

Unused variable declarations

Unused variables are often the source of bugs. If you declare a variable in the scope of your function but do not use it, the Go compiler will complain.

If the variable is unused, you should delete it, or assign it temporarily to the magic variable called _.

```
var cats = 20
var dogs = 30
var people = 7
var 自転車 = 4

// println(cats + dogs)
// _ = people
```

Run

This program has four unused variables.

- Fix the program by removing the comment from the two lines at the bottom.
- Delete, or comment out, the declaration of 自転車 as it is unused.
- Press the **Run** button to check that your program compiles.

20

Statements

A *statement* is a command to do something:

- *Declare a new integer variable x and assign it the value 10*.
- *Add 4 to the value of x and store the result in y*.
- *While x less than 20 perform these statements*.

There are several kinds of statements in Go, we've seen a few already. The most common statements are:

- Declarations: `const pi = 3.14159265359`, `type Counter int32`, ...
- Assignments: `count = count + 1`, ...
- Increment/Decrement: `x++`, `y--`
- Conditionals: `if … { … } else { … }`, `switch { … }`
- Loops: `for … { … }`
- Control flow: `break`, `fallthrough`, `continue`, `goto`, `return`.

Assignment

It is common that you need to change the value stored in a variable, this is called *Assignment*.

```
var x = 1
println(x)

x = 2
println(x)

var y = x + 2
println(y)
```

Run

In this example, we declare

- Press **Run** to check that program prints, 1, 2, 4.

Assignment (cont.)

You can only assign a new value to *variables*.

```
const x = 1
x = x + 1
println(x)
```

Run

It is a syntax error to assign a new value to a constant.

- Press **Run** and see the program does not compile.
- Edit the sample code to change the declaration of x from a constant (`const`) to a variable (`var`).
- Press **Run** and check the program compiles and prints the correct answer, 2.

23

Increment and decrement

Go supports a limited form of variable post-increment and post-decrement, ie. `x++`, `x--`.

```
var i = 1
i++
println(i)
```

Run

- Press **Run** and see the program prints 2,
- Edit the sample code to subtract 1 from `i` using the decrement operator `i--`.
- Press **Run** and check the program compiles and prints the correct answer, 0.

24

Increment and decrement (cont.)

`i++` and `i--` are *statements*, not an *expressions*, they do not produce a value.

```
var i = 1

var j = i++
println(i, j)
```

Run

This program has a syntax error.

- Edit the sample code to correct the error by moving the `i++` statement above the declaration of `var j`.
- Press **Run** and check the program compiles and prints the correct answer, 2, 2.

Looping

Go has a single `for` loop construct that combines

- `while` *condition* `{ ... }`
- `do { ... } while` *condition*
- `do { ... } until` *condition*

into one syntax.

- `for` *(init statement)*; *condition*; *(post statement)* `{ ... }`

The parts of a `for` statement are:

- init statement: used to initalise the loop variable; `i = 0`.
- condition: user to test if the loop is done; `i < 10`, `true` means keep looping.
- post statement: user to increment the loop variable; `i++`, `i = i - 1`.

Looping (cont.)

Let's practice using loops.

```
var i = 0
for i = 1; i < 11; i++ {
    println(i)
}
```

Run

This program counts from 1 to 10, can you make it print only the *even* numbers.

- Edit the program to make it print only the even numbers between 1 and 10, inclusive.
- Press **Run** and check the program compiles and prints the correct answer, 2, 4, 6, 8, 10.

note: you do not need to put `(` braces around the `for` condition `)`. In fact, if you do it's a syntax error. Try it.

27

Looping (cont.)

Let's practice counting down, this is where you would use a `while` loop in other languages.

```
var i = 10
for i > 0 {
    println(i)
    i--
}
```

Run

This program counts down from 10 to 1, can you make it count from 7 to 3.

- Edit the program to make it print only the numbers from 7 down to 3.
- Press **Run** and check the program compiles and prints the correct answer, 7, 6, 5, 4, 3.

note: this `for` loop only has a *condition*, there is no *init statement* or *post statement*, so we can omit the semicolons, `;`.

Conditional statements

Go has two conditional statements, `if` and `switch`.

`if` is used to choose between two choices based on a *condition*.

```
if v > 0 {
        println("v is greater than zero")
} else {
        println("v is less than or equal to zero")
}
```

In some cases the body of the `else` block may be omitted. This is very common when checking preconditions in a method of a function.

```
if v == 0 {
        // nothing to do
        return
}
// handle v
```

if

Let's revisit our previous even numbers `for` loop example.

```
var i = 0
for i = 1; i < 11; i++ {
    // if i%2 == 0 {
        println(i)
    // }
}
```

Run

This program counts from 1 to 10, can you make it print only the *even* numbers.

- Uncomment the `if i%2 == 0 {` and `}` lines to make the program print only the even numbers between 1 and 10.
- Press **Run** and check the program compiles and prints the correct answer, 2, 4, 6, 8, 10.

note: % is the *modulo* operator in Go. All even numbers divide wholly by 2, leaving 0 as their *modulo* (or remainder).

30

continue

Unlike languages like Java, `if` statements in Go are often used as *guard clauses*.

We say that when everything is true the code reads from the top to the bottom of the page.

We can rewrite the previous program using a new statement, `continue`, which *skips* the body of the loop.

```
var i = 0
for i = 1; i < 11; i++ {
    // if i%2 == 1 {
    //    continue
    // }
    println(i)
}
```

Run

- Uncomment the `if i%1 == 1 {`, `continue`, and `}` lines.
- Press **Run** and check the program compiles and prints the correct answer, 2, 4, 6, 8, 10.

break

This program is an *infinite loop*. There is no *condition expression* in the `for` loop.

We can use the `break` statement, which breaks out of the *current* loop, to fix it.

```
var i = 1
for {
    // if i > 10 {
    //    break
    // }
    if i%2 == 0 {
        println(i)
    }
    i++
}
```

Run

- Uncomment the `if i > 10 {`, `break`, and `}` lines.
- Press **Run** and check the program compiles and prints the correct answer, 2, 4, 6, 8, 10.

Type inference

In the examples so far we've avoided talking about types, this is because Go supports *type inference*.

Type inference lets you omit the type of a variable during declaration.

For example:

```
var i = 7
```

Go sees that `i` is being declared and initalised with the value 7, so the compiler infers the type of `i` is an `int`.

However if we did

```
var s = "seven"
```

Go sees that `s` is being initalised with the *string* `"seven"`, so the compiler infers the type of `s` is a `string`.

Explicit type declaration

Sometimes you will want to tell Go to use a specific type.

You do this when you declare a variable:

```
var i int = 4
var s string = "おはようございます"
```

Which tells the compiler that you are declaring `i` explicitly to be an `int` with the value 7, and `s` to be a `string` with the value "おはようございます"

We'll talk about the kinds of types that Go supports on the next slide.

Types

Go is a strongly typed language, like Java, C, C++, and Python. Go has nine kinds of types, they are:

- strings: `string`.
- signed integers: `int8`, `int16`, `int32`, `int64`.
- unsigned integers: `uint8`, `uint6`, `uint32`, `uint64`.
- aliases: `byte`, `rune`, `int`, `uint`.
- booleans: `bool`.
- IEEE floating point: `float32`, `float64`.
- Complex types: `complex64`, `complex128`.
- Compound types: `array`, `slice`, `map`, `struct`.
- Pointer types: `*int`, `*bytes.Buffer`.

String types

Strings are the most common data type in Go programs.

String types behave much as you would expect in other languages.

```
var salutation = "Hello"
var name = "David"

var greeting = salutation + " " + name
println(greeting)
```

Run

Strings may be *concatenated* with the + operator.

- Press **Run** to see the result.

note: In Go an empty string is "", not `null` or `nil`.

36

Integer types

Integer types are the second most common in Go.

Integer types come in two types; *signed* and *unsigned*.

Integer types also come in several sizes, represented by the number of bits they represent;

- Signed integers: `int8`, `int16`, `int32`, `int64`.
- Unsigned integers: `uint8`, `uint16`, `uint32`, `uint64`.

Go has two integer types

- `int`, alias for `int32` or `int64` depending on platform.
- `uint`, alias for `uint32` or `uint64` depending on platform.

whose size depends on the platform you used to build your Go program.

Integer types (cont.)

Why does Go support so many kinds of integer types?

Different *sized* integer types can accommodate different ranges of numbers.

```
var x int8 = 400
var y uint = -7

println(x, y)
```

Run

This program contains two type errors.

- Press **Run** to discover the two type errors.
- Fix the program by changing the declared types of x and y.
- Press **Run** and check the program compiles and prints the correct answer, 400, -7

38

Functions

Now it's time to talk about *functions*.

All Go you write is made up of functions, in fact, you've been writing functions all along.

```
func main() {
    var x int8 = 400
    var y uint = -7

    println(x, y)
}
```

Run

This is the same program as the previous example, showing the *function declaration*, which was previously hidden.

- All Go programs start in a function called `main`, we call this the *main function*.

39

func

You can declare your own functions with the `func` declaration.

A function's name must be a valid identifier, just like `const` and `var`.

```
func hello() {
    // println("こんにちは")
}

func main() {
    var i int = 0
    for i = 1; i < 4; i++ {
        hello()
    }
}
```

Run

This program declares two functions, `main` and `hello`.

- Uncomment the `println("こんにちは")` statement in `func hello`.
- Press **Run** and check the program compiles and prints こんにちは three times.

note: `println` is a *built in* function provided, you don't need to declare it.

Function parameters

To make functions useful, you often need to pass *arguments* to a function.

To pass an argument to a function, the type of the argument and the type of the function's *formal parameter* must be the same.

```
func hello(name int) {
    println("こんにちは " + name)
}

func main() {
    hello("David")
}
```

Run

In this program the type of the argument passed to `hello` does not match the type of the `name` parameter.

- Fix the declaration of `hello` so that the type of the parameter matches the type of the argument provided.
- Press **Run** and check the program compiles and prints `こんにちは David`.

Packages

A package is the unit in which software is shared and reused in Go. All Go code is arranged into packages.

Each source file in a package must begin with the same `package` declaration.

A package's name must be a valid identifier, just like `const`, `var`, and `func`.

```
package main

func hello(name string) {
    println("こんにちは " + name)
}

func main() {
    hello("David")
}
```

Run

This is the same example as the previous slide, revealing the package declaration.

`package main` is the name of the package which contains the entry point to your program, `func main`.

42

main packages

This program has the wrong package declaration.

```
package greeting

func main() {
    println("Hello Women Who Go!")
}
```

Run

- Press **Run** and observe the compile error.
- Change the package declaration to make the program compile.
- Press **Run** to check that program prints its greeting.

43

Import

The final declaration we'll cover in this section is the *import* declaration.

The `import` declaration allows you to use code from other *packages* into your package.

When you `import` a package, the *public* types, functions, variables, types, and constants, are available with a prefix of the package's name.

```
time.Now    // denotes the Now function in package time
```

Note that

```
import "fmt"
import "time"
```

and

```
import (
        "fmt"
        "time"
)
```

both import the `fmt` and `time` packages. The syntax is different, but they are equivalent. 44

Import (cont.)

The `import` declaration must appear *after* the `package` declaration, but before any `type`, `const`, `var`, or `func` declarations.

```
package main

var now = time.Now()

import "fmt"
import "time"

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

Run

This program does not compile as the `import` declaration is in the wrong place.

- Move the `var now = time.Now()` declaration *below* the `import` declaration.
- Press **Run** to check that program prints the current time.

Import (cont.)

Packages contain both *public* and *private* symbols. We also call these *exported* and *not exported*, respectively.

```
package main

import "fmt"
import "time"

func main() {
    var now = time.now()
    fmt.println(now)
}
```

Run

This program does not compile as it refers to two *private* symbols.

- Fix the program by using the correct case for `Println` and `Time`.
- Press **Run** to check that program prints the current time.

note: If you are running this slide from gotalks.golang.org (https://go-talks.golang.org/), the time may be reported as `2009-11-10 23:00:00 +0000 UTC`. This is a technical limitation.

fmt package

Do you remember this program?

```
package main

import "fmt"

func main() {
    var π = 3.14159
    var radius = 6371.0 // radius of the Earth in km
    var circumference = 2 * π * radius

    fmt.Println("Circumference of the earth is", circumference, "km")
}
```

Run

Here it is again, showing the `func`, `package` and `import` declarations making it a complete Go program.

This program also uses the `Println` function from the `fmt` package, which is more capable than the built in `println` function.

- Press **Run** to see how `fmt.Println` prints this output.

Recap

Now you know the basics of Go!

You've learnt:

- How to declare constants and variables
- How to write `for` loops and use `if`.
- How types work.
- How to write your own functions.
- How packages and `import` statements work.

Time for a quick break!

48

Advanced Syntax

49

Advanced Syntax

This section builds on the previous by exploring each of the things we learnt in a little more detail.

In this section we'll use the Go Playground, a simple online code editor to perform our exercises.

Coding style

All Go code is formatted according to a single style guide which is enforced with a tool called `gofmt`

Having one single style that all Go code is formatted in improves readability and avoids the time wasted arguing about code formatting.

"Gofmt's style is no one's favorite, yet gofmt is everyone's favorite."

Go Proverb (https://go-proverbs.github.io/).

The Go playground can format your code according to the canonical Go style.

- Follow this link (https://play.golang.org/p/0Hz57BQdTA) and press the **Format** button to see this in action.

Zero value

In previous examples we've written code like this

```
var name = "go"
var counter = 1
```

Which both *declares* and *initalises* the variables `counter` and `name` respectively. What happens if we have code like this?

```
package main

import "fmt"

func main() {
    var name string
    var counter int

    fmt.Printf("%q, %v", name, counter)
}
```

What will this print?

- Follow this link (https://play.golang.org/p/jioUB0t_LW) and press the **Run** button to see this in action.

Zero value (cont.)

In Go, there is no unitialised memory. The Go runtime will always ensure that the memory allocated for each variable is initalised before use.

If we write something like

```
var name string
var counter int
```

Then the memory assigned to the variables `name` and `counter` will be zeroed, as we have not provided an *initaliser*.

- The value of `name` will be `""` because that is the value of a string with zero length.
- The value of `counter` will be zero, because that is the value of an `int` if we wrote 0 to its memory location.

Zero value (cont.)

Every type in Go has an associated *zero value*. The value of that variable if we wrote zeros to its memory.

- The zero value for integer types: `int`, `int8`, `uint`, `uint64`, etc, is `0`.
- The zero value for floating point types: `float32`, `float64`, `complex128`, etc, is `0.0`.
- The zero value for arrays is the zero value for each element, ie. `[3]int` is `0`, `0`, `0`.
- The zero value for slices is `nil`.
- The zero value for structs is the zero value for each field.

Equality

As Go is a strongly typed language, for two variables to be equal, both their *type and their value* must be equal.

Trying to compare two variables of *different* types is detected at runtime.

```
package main

import "fmt"

func main() {
    var x uint = 700
    var y int = 700

    fmt.Println(x == y)
}
```

- Follow this link (https://play.golang.org/p/LwiJm2xuXK) and press the **Run** button.
- Fix the program by declaring x and y to be the *same* type.
- Press the **Run** button and confirm the program now prints `true`.

Type conversions

Sometimes you have variables of different integer types, you can *convert* from one type to another using a conversion *expression*.

The expression `T(v)` converts the value v to the type T.

```
package main

import "fmt"

func main() {
    var x uint = 700
    var y int = x

    fmt.Println(y)
}
```

In this example the assignment of `y = x` fails because x and y are different integer types.

- Follow this link (https://play.golang.org/p/wvG41C0lH4) and press the **Run** button.
- Fix the program by *converting* x to an `int` with `int(x)`.
- Press the **Run** button and confirm the program now prints 700.

Type conversions (cont.)

If you have variables of different *widths*, you can *convert* from one type to another.

```
package main

import "fmt"

func main() {
    var x int16 = 32000
    var y int64 = x

    fmt.Println(y)
}
```

- Follow this link (https://play.golang.org/p/l4Q48pWAla) and press the **Run** button.
- Fix the program by *converting* `x` to an `int64` with `int64(x)`.
- Press the **Run** button and confirm the program now prints 32000.

Type conversions (cont.)

We can do the opposite and convert a wider type to a narrower type.

```
package main

import "fmt"

func main() {
    var x int64 = 64000
    var y int16 = int16(x)

    fmt.Println(y)
}
```

- Follow this link (https://play.golang.org/p/NbNwRjbmRu) and press the **Run** button. Does it print the answer you expected?
- Fix the program by *declaring* y as an `int32`.
- Press the **Run** button and confirm the program now prints 64000.

Integer overflow

Whenever you declare a variable in Go, you have to choose how many bits of memory it will consume.

When you convert a variable with a *smaller* number of bits to a variable with a larger number of bits, this is fine, because they all fit.

When you convert a variable with a *larger* number of bits to a variable with a smaller number of bits there is a risk of truncation, because there are less bits available to represent your number.

- Follow this link (https://play.golang.org/p/NbNwRjbmRu) and press the **Run** button. Does it print the answer you expected?
- Fix the program by reducing the value of x. Hint: the value needs to be less than 33,000.
- Press the **Run** button and confirm the program now prints y correctly.

Short declaration syntax

As you've probably noticed, Go has several ways to declare variables. All three of these are the same

```
var x = 0
var x int = 0
var x int
```

If you've come from a language like Ruby or Python, you're probably wondering if this very common operation can be made more concise. Indeed it can.

```
x := 0
```

This is what we call a *short declaration*, which is the same as

```
var x int = 0
```

Short declaration is very common in Go programs, you'll see it everywhere, so let's do some exercises to familarise you with its use.

60

Short declaration syntax (cont.)

A common use of the short declaration syntax is in `for` loops. Consider this program

```
var i int
for i = 1; i < 11; i++ {
        fmt.Println(i)
}
```

This can be also written as

```
for i := 1; i < 11; i++ {
        fmt.Println(i)
}
```

This program (https://play.golang.org/p/3Vl75w72JO) contains two `var` declarations and two `for` loops.

- Follow this link (https://play.golang.org/p/3Vl75w72JO) and press the **Run** button.
- Rewrite the program using the short declaration syntax; there should be no `var` declarations, only `:=`.

61

Slices

The next kind of type to discuss is the *Slice*.

A slice is an ordered collection of values of a *single* type.

The syntax for declaring a slice variable is very similar to declaring a *scalar* variable.

```
var i int      // an int called i
var j []int    // a slice of ints called j
```

In this example,

- `i` is a variable of type `int`.
- `j` is a variable of type `[]int`, that is, a slice of `int`.

Slices are very important in Go programs, so we'll spend a bit of time discussing them.

note: A slice is *not* an array. Go also supports arrays, but you'll see later than they aren't very common, or very easy to use, so we won't discuss them at the moment.

How large is a slice?

If I declare a slice, `[]int`, how many items can it hold?

The *zero value* of a slice is empty, that is, it has a *length* of zero; it can hold 0 items.

```
package main

import "fmt"

func main() {
    var i []int
    fmt.Println(len(i))
}
```

We can retrieve the length of a slice with the built-in `len` function.

- Follow this link (https://play.golang.org/p/gZYvdE2zbT) and press the **Run** button.
- Did you guess the right answer?

63

Making a slice

We can create a slice with space to hold items using the built-in `make` function.

```
package main

import "fmt"

func main() {
    var i []int
    i = make([]int, 20)
    fmt.Println(len(i))
}
```

In this example, on the first line `` `var i []int` `` declares `i` to be a slice of `int`.

On the second line, `i` is *assigned* the result of `make([]int, 20)`.

- Follow this link (https://play.golang.org/p/i_IWqjik6u) and press the **Run** button.
- Did `fmt.Println(len(i))` print the result you expected?

64

Making a slice (cont.)

Because declaring a slice variable and initalising it with `make` is a common operation, it is common to see the *short variable declaration* used to combine this operations.

```
package main

import "fmt"

func main() {
    i := make([]int, 20)
    fmt.Println(len(i))
}
```

This example declares `i` *and* initalises it to be a slice of `int` with a length of 20.

- Follow this link (https://play.golang.org/p/b92SJ0Gx9s) and press the **Run** button.
- Did `fmt.Println(len(i))` print the result you expected?

65

Slice exercises

Let's do a quick exercise to familarise yourself with using slices.

```
package main

import "fmt"

func main() {

    // declarations go here

    fmt.Println(len(i), len(f), len(s))
}
```

- Follow this link (https://play.golang.org/p/AJk1Jgp1iE) for instructions.
- Declare a variable called `i` which is a slice of 5 `int`.
- Declare a variable called `f` which is a slice of 9 `float64`.
- Declare a variable called `s` which is a slice of 4 `string`.
- Does your program print the expected result, `5 9 4`?

Index expressions

To access, or assign, the contents of a slice element at index `i`, use the form `s[i]`.

Slices are zero indexed, so `s[0]` is the 1st element, `s[1]` is the second element, and so on.

When the *index expression* appears on the *left hand side* of the equals operator, =

```
s[7] = 20
```

We are assigning the number 20 to the 8'th element of the slice `s`.

When the *index expression* appears on the *right hand side* of the equals operator, =

```
x := s[7]
```

We are assigning the value at the 8th element of `s` to the variable `x`. 67

Slice zero value

We saw earlier that the *zero value* of the slice

```
var s []int
```

was and empty slice, a slice with length of zero.

What is the value of each of the elements of a newly created, with make, slice?

```
package main

import "fmt"

func main() {
    x := make([]int, 5)
    for i := 0; i < len(x); i++ {
        fmt.Println(x[i])
    }
}
```

Run

- Follow this link (https://play.golang.org/p/kGh_C1l6KW) and press the **Run** button.
- Did the program print the result you expected?

Slice initialisation

We want to create an `[]int` slice of the first 10 prime numbers, how could we do this?

One solution could be to create the slice and assign a value to each element in the slice.

```
package main

import "fmt"

func main() {
    primes := make([]int, 10)
    primes[0] = 2
    primes[1] = 3
    primes[2] = 5
    primes[3] = 7
    primes[4] = 11
    primes[5] = 13
    primes[6] = 17
    primes[7] = 19
    primes[8] = 23
    primes[9] = 29
    fmt.Println(primes)
}
```

Run

69

Slice initialisation (cont.)

Doing this manually is verbose and boring; how would you do this for the first 50 primes?

Go supports a method of assignment where we both *declare* and *initalise* the slice at once.

```
package main

import "fmt"

func main() {
    primes := []int{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
    fmt.Println(primes)
}
```

Run

This is called the *composite literal* syntax.

- Follow this link (https://play.golang.org/p/P-eBqzPCWh) and complete the exercise.

append

So far we've been using slices with a known length. You can extend the contents of a slice with the built-in append function.

```
package main

import "fmt"

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

    primes = append(primes, 13)
    fmt.Println(len(primes), primes)

    primes = append(primes, 17, 19, 23)
    fmt.Println(len(primes), primes)
}
```

Run

append increases the length of the slice to accommodate the new items, then returns a new slice value.

You can append multiple values in one statement, providing they are all the same type.

Further reading: Arrays, slices (and strings): The mechanics of 'append' (blog.golang.org)

(https://blog.golang.org/slices)

Subslices

What if we have a large slice, and want to refer to only a part of it.

We call this slicing a slice, or *subslicing*.

Subslicing looks similar to the *indexing* operation we saw a few slide ago, except it refers to a range of slice indexes.

```
package main

import "fmt"

func main() {
    brothers := []string{"chico", "harpo", "groucho", "gummo", "zeppo"}
    fmt.Println(brothers)

    wellknown := brothers[0:3]
    fmt.Println(wellknown)
}
```

Run

The expression `brothers[0:3]` evaluates to a slice of the 1st to 3rd Marx brother.

- Follow this link (https://play.golang.org/p/d1-jl42aTF) and complete the exercise.

Subslices (cont.)

An important thing to remember when slicing a slice, is that both slices refer to the *same* underying data.

```
package main

import "fmt"

func main() {
    a := []int{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
    fmt.Println(a)

    b := a[0:5]
    for i := 0; i < len(b); i++ {
        b[i] = b[i] * -1
    }
    fmt.Println(b)

    fmt.Println(a)
}
```

Run

To create two independent slice values, you would use the copy function, which we'll discuss later.

Bounds checking

Each slice has a length which is decided when it is made.

You can increase the length of the slice with the append function, and create a smaller slice from a larger one using the slice operator.

What happens if you accidentally exceed the bounds of the slice?

```
package main

import "fmt"

func main() {
    primes := []int{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}

    fmt.Println(primes[-1])
    fmt.Println(primes[10])
}
```

Run

- Follow this link (https://play.golang.org/p/mlWm0a1amp) and press the **Run** button.
- Comment out `fmt.Println(primes[-1])` and see what happens.

Multiple assignment

Go allows you to perform multiple assignments and declarations in one statement.

For example, if we wanted to declare, x, y, and z, with the values 1, 2, and 3 respectively. We could write

```
var x = 1
var y = 2
var z = 3
```

We can write the same thing like this

```
var x, y, z = 1, 2, 3
```

- Follow this link (https://play.golang.org/p/d2hDJQAfkt) for some examples of multiple declaration.

75

Multiple return values

Multiple assignment is important to understand because you can return multiple values from a function.

```
func f(i int)
```

This is a function declaration for `f` which takes one argument, an `int`.

```
func g(i int, j int, k string) int
```

This is a function declaration for g, which takes three arguments, two `int` s and a `string`, and returns an `int`.

```
func h(i, j int) (int, int, string)
```

This is a function declaration for h, which takes two arguments, two `int`s, and returns three values, two `int` s and a `string`.

Multiple return values (cont.)

Your program must return the number of values specified in the function signature.

```
package main

import "fmt"

// Double returns a number two times larger than i.
func A(i int) int {
    return i * 2
}

// Swap switches the values of a and b
func Swap(a int, b int) (int, int) {
    return b, a
}

func main() {
    fmt.Println(A(200))

    a, b := Swap(7, 9)
    fmt.Println(a, b)
}
```

Run

77

Assigning multiple return values

When you call a function that returns multiple values, you must assign *all* of them or *none* of them.

```
func f() (int, bool, string)

func main() {
        a, b, c := f()
        fmt.Println(a, b, c)
}
```

`f` returns three values, so we assign them to `a`, `b`, and `c` using the short declaration syntax.

If we wanted to use only the first and third values we can ignore the second by assigning it to the underscore variable, `_`.

```
func main() {
        a, _, c := f()
        fmt.Println(a, c)
}
```

Maps

Go has a built in Hash Map type, called a `map`.

Maps map values of key type K to values of type V

```
var m map[string]int
```

Just like making a slice, making a map is accomplished with the `make` built-in.

```
package main

import "fmt"

func main() {
    m := make(map[string]int)
    fmt.Println(m)
}
```

Run

79

Inserting values into a map

Inserting a value into a map looks similar to assigning a value to a slice element.

```
package main

import "fmt"

func main() {
    days := make(map[int]string)
    days[1] = "Monday"
    days[2] = "Tuesday"
    days[3] = "Wednesday"
    days[4] = "Thursday"
    days[5] = "Friday"
    days[6] = "Saturday"
    days[7] = "Sunday"
    fmt.Println(days)
}
```

Run

If an entry already exits with that key, it will be overwritten.

- Follow this link (https://play.golang.org/p/a-V5I0nZ5l) and complete the exercise.

note: maps are always *unordered*.

Compact literal initalisation

Just like slices, maps support compact literal initalisation, which declares and initalises the map.

```
package main

import "fmt"

func main() {
    days := map[int]string{
        1: "Monday",
        2: "Tuesday",
        3: "Wednesday",
        4: "Thursday",
        5: "Friday",
        6: "Saturday",
        7: "Sunday",
    }
    fmt.Println(days)
}
```

Run

81

Retrieving a values from a map

Just like a slice, you can retrieve the value stored in a map with the syntax `m[key]`.

If it is present the value will be returned, if not the *zero value* will be returned.

```
package main

import "fmt"

func main() {
    // map of planets to their number of moons
    moons := map[string]int{
        "Mercury": 0,
        "Venus":   0,
        "Earth":   1,
        "Mars":    2,
        "Jupiter": 67,
    }

    fmt.Println("Earth:", moons["Earth"])
    fmt.Println("Neptune:", moons["Neptune"])
}
```

Run

82

Checking if a map value exists

In the previous slide we saw that `moons["Neptune"]` returned 0.

How can we tell if Neptune actually has no moons, or if 0 was returned because there is no entry for Neptune?

Map look ups support a second syntax.

```
package main

import "fmt"

func main() {
    moons := map[string]int{"Mercury": 0, "Venus": 0, "Earth": 1, "Mars": 2, "Jupiter": 67}

    n, found := moons["Earth"]
    fmt.Println("Earth:", n, "Found:", found)

    n, found = moons["Neptune"]
    fmt.Println("Neptune:", n, "Found:", found)
}
```

Run

83

Deleting a value from a map

To delete a value from a map, you use the built in `delete` function.

```
package main

import "fmt"

func main() {
    moons := map[string]int{"Mercury": 0, "Venus": 0, "Earth": 1, "Mars": 2, "Jupiter": 67}

    const planet = "Mars"

    n, found := moons[planet]
    fmt.Println(planet, n, "Found:", found)

    delete(moons, planet)

    n, found = moons[planet]
    fmt.Println(planet, n, "Found:", found)
}
```

Run

84

Iterating over a map

If we wanted to print out all the values in a map we can use a form of the `for` syntax which is known as `range`.

```
package main

import "fmt"

func main() {
    cities := map[string]int{
        "Tokyo": 33200000, "New York": 17800000,
        "Sau Paulo": 17700000, "Delhi": 14300000,
        "Moscow": 10500000,
    }

    for name, pop := range cities {
        fmt.Println("City:", name, "Population", pop)
    }
}
```

Run

`range` loops over each entry in the map, assigning the map key to `name`, and the map value to `pop`.

85

Range over slices

We say previously that `for range` works with maps, it also works with slices.

```
package main

import "fmt"

func main() {
    primes := []int{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}

    for i, p := range primes {
        fmt.Println("The", i, "th prime is", p)
    }
}
```

Run

- Follow this link (https://play.golang.org/p/AmQW-OrPC1) and complete the exercise.
- If you cannot figure it out, don't worry, there is an answer on the next slide.

Switch

If you completed the previous exercise you may have written something like this

```
func main() {
    primes := []int{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}

    for i, p := range primes {
        i++
        if i == 1 {
            fmt.Println("The", i, "st prime is", p)
        } else if i == 2 {
            fmt.Println("The", i, "nd prime is", p)
        } else if i == 3 {
            fmt.Println("The", i, "rd prime is", p)
        } else {
            fmt.Println("The", i, "th prime is", p)
        }
    }
}
```

Run

Heavily nested `if else if` blocks are discouraged in Go.

Instead we can use the other condition statement, `switch`.

87

Switch (cont.)

switch can be used

```
func main() {
    primes := []int{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}

    for i, p := range primes {
        i++
        switch i {
        case 1:
            fmt.Println("The", i, "st prime is", p)
        case 2:
            fmt.Println("The", i, "nd prime is", p)
        case 3:
            fmt.Println("The", i, "rd prime is", p)
        default:
            fmt.Println("The", i, "th prime is", p)
        }
    }
}
```

Run

88

fmt

Let's conclude this section by talking about the fmt package.

`fmt` standard for formatted printing; the name is inherited from Go's Plan 9 legacy.

We've use `fmt.Println` a lot up to this point, but the `fmt` package has many other useful functions.

We'll focus on `fmt.Printf`, the `f` stands for *formatted output*.

Here is an example.

```
func main() {
    name := "David"
    age := 27

    fmt.Printf("Hello my name is %s, my age is %d", name, age)
}
```

Run

- Follow this link (https://play.golang.org/p/UuzPWcwDrE) to experiment.

89

Formatting verbs

If you're used to languages like Python or C, you're probably used to the idea of *formatting verbs*.

The `fmt` package supports a large number of formatting verbs and modifiers. In the previous example you saw %s and %d, for `string` and `int` respectively.

The `fmt` package is smart enough to spot when you use the wrong formatting verb, or don't provide enough arguments to `fmt.Printf`.

```
func main() {
    name := "David"
    age := 27

    fmt.Printf("Hello my name is %d, my age is %d", name)

    _ = age // keep the compiler happy
}
```

Run

90

Formatting verbs (cont.)

Having to choose the correct verb that matches the type of the value you want to print is boring.

To make it easier to use `fmt.Printf` in the simple case, you can use the %v verb, which know how to print *any* value.

```
func main() {
    name := "David"
    age := 27

    fmt.Printf("Hello my name is %v, my age is %v", name, age)
}
```

Run

- Follow this link (https://play.golang.org/p/B-E7JOJ1Db) to experiment with a few more formatting verbs.

Recap

Now you know most of Go!

You've learnt:

- How Go code is formatted.
- How what the zero value is and how it works.
- Equality and type conversions
- The short declaration syntax
- Multiple assignment.
- How slices work.
- How maps work.
- How the `fmt` package works.

Time for lunch!

Development environment and tooling

93

Development environment and tooling

This section focuses on the developer experience

- Installing Go.
- Setting up a `$GOPATH` workspace.
- Introduction to the go command.
- Writing unit tests with the go test command.

Installing Go

The next few slides give examples of how to install Go on various systems.

The official installation documentation is here:

golang.org/doc/install (https://golang.org/doc/install)

The current version of Go is 1.7.4.

- New minor releases, Go 1.8, Go 1.9, etc, ship twice a year.
- Historically 2–4 bug fix or security releases occur during each six month period.

Installing Go on OS X

Installing Go on OS X (also known as `darwin`) can be accomplished with

Mac OS Installer

The Go project provides an `.pkg` installer, use:

storage.googleapis.com/golang/go1.7.4.darwin-amd64.pkg (https://storage.googleapis.com/golang/go1.7.4.darwin-amd64.pkg)

Homebrew

If you use Homebrew to manage third party software on your Mac, use:

```
brew install golang
```

MacPorts

If you use MacPorts to manage third party software on your Mac, use:

```
sudo port install go
```

96

Installing Go on Windows

MSI Installer

The Go project provides a `.msi` installer for 32bit and 64bit Windows, use:

Windows 64bit MSI installer (https://storage.googleapis.com/golang/go1.7.4.windows-amd64.msi)

Windows 32bit MSI installer (https://storage.googleapis.com/golang/go1.7.4.windows-386.msi)

.zip file

The Go project provides a `.zip` file of the Go compiler and standard library, use:

Windows 64bit zip file (https://storage.googleapis.com/golang/go1.7.4.windows-amd64.zip)

Windows 32bit zip file (https://storage.googleapis.com/golang/go1.7.4.windows-386.zip)

Unzip the zip file and place it at `C:\Go`.

Installing Go on Linux

Depending on your Linux distribution they may have an up to date version of Go, but this is rare.

The most reliable way to install the latest version of Go is to untar

storage.googleapis.com/golang/go1.7.4.linux-amd64.tar.gz (https://storage.googleapis.com/golang/go1.7.4.linux-amd64.tar.gz)

to the directory

```
/usr/local
```

and add

```
/usr/local/go/bin
```

to your $PATH.

```
echo "export PATH=$PATH:/usr/local/go/bin" >> ~/.bash_profile
```

98

Go tool

Your Go installation comes with a tool we call the go tool, because that's its name.

The go tool can

- compile your programs
- run your tests
- display documentation for a package
- fetch packages from the internet.

$GOPATH

The go tool works inside a workspace where *all* your Go source code is stored.

All the source code for this workshop is included with this repository.

You can set `$GOPATH` to be the base directory where you checked out this repository. eg.

```
% export GOPATH=$HOME/introduction-to-go
```

Using a workspace allows you to import code from other packages with a fixed name. eg.

```
import "github.com/pkg/profile"
```

Will import the code for the `profile` package stored in

```
$GOPATH/src/github.com/pkg/profile
```

go build

Go is a compiled language, so the usual work flow is

- Edit code
- `go build`
- Run program

Let's try building a Go program

- `cd $GOPATH/src/helloworld`
- Read the source for `hello.go`
- Build the source with `go build`
- Run the program `./helloworld`

101

What time is it (exercise)

Let's write a clock, a program that prints out the current time

- `cd $GOPATH/src/whattimeisit`
- edit `main.go` and finish the program (if you get stuck the answer is in `answer.go`)
- Build the program with, `go build` (if you make an error, go back and edit `main.go`)
- Run your program `./whattimeisit`, it should print something like this

```
The current time is 2016-12-05 12:33:18.222821474 +0900 JST
```

Testing

I wanted to spend some time on testing because for the rest of the day we'll be using tests to complete code katas.

The testing package can also contain benchmark functions and examples, which show up in godoc.

You should include tests for each package that you write.

The `testing` package is ideal for *unit tests*. It's *ok* for functional tests, but not really suitable for complex integration tests.

go test

go `test` is the unit testing framework built into the Go standard library. It lives in the `testing` package.

Tests live in `_test.go` files, eg. the `strings` package has these files:

- `strings.go` functions to manipulate UTF-8 encoded strings.
- `strings_test.go` tests for the `strings` package.

Each test is a function in the form

```
func TestNameOfTest(t *testing.T) { ... }
```

- `NameOfTest` is the name of your test, it *must* start with an upper case letter.
- Test functions take a `testing.T` value, which provides helpers like `t.Error` and `t.Fail`.

Writing tests

Let's write some tests using our own version of the strings package, called `simplestrings`.

- The code for this exercise is in `$GOPATH/src/simplestrings/`
- Read the source code for `simplestrings.go`

Together we'll write some tests for the functions in our `simplestrings` package.

We'll use the test coverage (see next slide) tool to check our work.

Test coverage

`go test` can report coverage

```
go test -coverprofile=cover.out
```

This produces a coverage file, `cover.out`

- `go tool cover -func=cover.out` will print the coverage report
- `go tool cover -html=cover.out` will open the report in a browser

Protip: I use these little shell functions to make this easier

```
cover () {
  t=$(mktemp -t cover)
  go test $COVERFLAGS -coverprofile=$t $@ && go tool cover -func=$t && unlink $t
}

cover-web() {
  t=$(mktemp -t cover)
  go test $COVERFLAGS -coverprofile=$t $@ && go tool cover -html=$t && unlink $t
}
```

The standard library

Go ships with a rich standard library of packages. This includes

- file input / output
- string handling
- compression
- encoding and decoding of JSON and XML
- network handling
- HTTP client and server

Katas

For the rest of the day let's practice writing Go code together.

The Ruby community have a tradition of *code katas*, small exercises to make the test pass.

As we work through the katas, we'll visit some more of the Go standard library. 108

String formatting

As a warm up, let's do a small kata together.

The code for this kata is in `$GOPATH/src/katas/sprintf`

If you get stuck, the answers are in `$GOPATH/src/katas/sprintf/answers_test.go`. 109

Structs

So far we've discussed two kinds of types; *primitive* types and *slice* types.

Go supports what we call *compound* types, that is, types that are *composed* of other types.

These are called *struct* (for *structure*) types. We declare a struct like this:

```
type Point struct {
        X int
        Y int
}
```

`Point` is a position in two dimensional space, it has two fields, `X` and `Y`.

- Follow this link (https://play.golang.org/p/fAnPV1MojK) to complete the example

Methods

To this point we've talked about functions, which belong to a package

```
// Max returns the larger of a or b.
func Max(a, b int) int
```

In Go, you can attach a function to a type that you declare, this is called a *method*.

```
type Point struct { X, Y int }

func (p Point) String() string {
        return fmt.Sprintf("point: x=%d, y=%d", p.X, p.Y)
}
```

Any type that implements a `String() string` method will be used by the `fmt` package when it prints the value.

- Complete the kata in `$GOPATH/src/katas/methods` by making all the tests pass.

Pointers

Whenever you pass a value to a function or method, the value is *copied*.

In Go, the method's receiver is also a value, so it's copied when you call a method.

```
type Point struct{ X, Y int }

func (p Point) String() string { return fmt.Sprintf("point: x=%d, y=%d", p.X, p.Y) }

func (p Point) Move(x, y int) {
    p.X += x
    p.Y += y
}

func main() {
    p := Point{X: 20, Y: 20}
    fmt.Println(p)

    p.Move(20, 10)
    fmt.Println(p)
}
```

Run

- Complete the kata in `$GOPATH/src/katas/pointers` by making the test pass.

Interfaces

Go is an object oriented language; we have methods on types, but Go does not support inheritance or sub-classes.

Go supports polymorphism, *has a* (not *is a*) with *interfaces*.

An interface declaration looks like this:

```
type Reader interface {
        Read(buf []byte) (int, error)
}
```

This is the `io.Reader` (https://golang.org/pkg/io/#Reader) interface declaration from the `io` (https://golang.org/pkg/io/) package.

Go does not have an *implements* keyword, any type with the correct set of methods *is* an implementation of the interface.

113

Reading input

This kata asks you to write a function that reads lines from an io.Reader and returns a string containing all the lines read.

The code for this kata is in `$GOPATH/src/katas/input` 114

Readers

To familarise you with the `io.Reader` implementations available in the `io` package, this kata is all about Readers.

- Complete the kata in `$GOPATH/src/katas/readers` by making the test pass.
- If you get stuck, consult the documentation in the `io` (https://golang.org/pkg/io/) package.

115

Error handling

You probably spotted that lots of methods and functions in the Go standard library return a value of type `error`.

`error` is a *predeclared type*, just like `int`, `string`, etc.

`error` is an interface, it's declaration is

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

Any type that has an `Error() string` method, *implements* the `error` interface. 116

Nil

`nil` is Go's version of `NULL`, `null`, `void`.

- The zero value of an interface type is `nil`.
- The zero value of a pointer type is `nil`.

Go uses the `error` interface and a simple convention to implement error handling.

- If no error occurs, the `err` value returned from a function or method will equal nil.
- If an error occurs, the `err` value returned from a function or method will not equal nil.

```
if err != nil {
        // cleanup and handle error
}
```

117

Counting the number of lines in a file

Now we know about `io.Reader`'s, `error`'s, we can write some more useful programs.

The code for this kata is in `$GOPATH/src/katas/countlines`

Note:

- go `test` always executes from the package's source directory, this makes it simple to include fixtures for your tests.
- The go tool ignores any directory called `testdata`, or starts with a `.` or _.

defer

In `CountLines` from our previous example, if an error happened during reading lines, `f` may not be closed.

Go has a keyword `defer` to ensure operations *always* happen.

```
func CountLines(path string) int {
    f, err := os.Open(path)
    if err != nil {
        log.Fatal(err)
    }
    defer f.Close()

    sc := bufio.NewScanner(f)
    var lines int
    for sc.Scan() {
        lines++
    }
    if err := sc.Err(); err != nil {
        log.Fatal(err)
    }
    return lines
}
```

Error handling (cont.)

In the previous counting example, if an error happened, the program would exit.

In this kata, we'll handle errors by returning them to the caller.

The code for this kata is in `$GOPATH/src/katas/errorhandling`

120

Passing in a reader

Let's turn out `Countlines` function into a program.

The code for this kata is in `$GOPATH/src/katas/linecount`

Complete the program so it reads the number of lines sent to it via stdin.

```
% cat testdata/moby.txt | ./linecount
22659
```

Handling multiple files

Let's extend our `linecount` program to handle files passed on the command line.

The code for this kata is in `$GOPATH/src/katas/countmanyfiles`

Complete the program so it counts the lines in files passed via the command line.

```
% ./countmanyfiles testdata/*.txt
testdata/dracula.txt    15973
testdata/pride-and-prejudice.txt        13427
testdata/sherlock.txt   13052
```

Reading all the *.txt files in a directory

In the previous example we used the shell to list files to process.

In this kata, let's extend our `countmanyfiles` program to walk the directory listing itself.

To do this we use the `ReadDir` (https://golang.org/pkg/os/#File.Readdir) method on `os.File` (https://golang.org/pkg/os/#File).

Note: Be careful to only read *files*, not directories, and do not read files that don't end in `.txt`

The code for this kata is in `$GOPATH/src/katas/countdir`

Complete the program so it counts the lines in files passed via the command line.

```
% ./countdir testdata/
testdata/christmas-carol.txt    4236
testdata/tom-sawyer.txt 9209
```

123

Let's take a break

124

HTTP request

The Go standard library supports writing HTTP clients and servers with the `net/http` (https://golang.org/pkg/net/http/) package.

Using the `net/http` package is very straight forward:

```
resp, err := http.Get("http://example.com/")
if err != nil {
        // handle error
}
```

`resp` is a `http.Response` (https://golang.org/pkg/net/http/#Response) structure, which has lots of interesting fields.

Let's write a simple HTTP client that can fetch HTTP URLs.

```
% ./httpget http://httpbin.org/ip
{
  "origin": "125.203.122.114"
}
```

The code for this kata is in `$GOPATH/src/katas/httpget`

JSON parsing

The service at `http://httpbin.org/` returns JSON bodies.

The `encoding/json` (https://golang.org/pkg/encoding/json/) package can decode JSON data into a map.

```
result := make(map[string]string)
dec := json.NewDecoder(resp.Body)
err := dec.Decode(&result)
if err != nil {
        // handle error
}
```

Let's use this to write a program that will tell us our public IP address.

```
% ./whatismyip
My IP address is: 125.203.122.114
```

The code for this kata is in `$GOPATH/src/katas/whatismyip`

JSON encoding

If you're writing a RESTful web service it's common to have to return JSON encoded data.

In Go the `encoding/json` (https://godoc.org/encoding/json) package can turn Go maps and data structures into JSON.

```
type Person struct {
    Name    string
    City    string
    Country string
}

func main() {
    p := Person{Name: "Dave", City: "Sydney", Country: "Australia"}
    var b bytes.Buffer
    enc := json.NewEncoder(&b)
    err := enc.Encode(p)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(b.String())
}
```

Run

Controlling JSON encoding

The `encoding/json` package requires the fields of a struct to be public (start with an upper case letter), this means the keys in your JSON document will be upper case.

We can fix this and control the output of the JSON with a *tag*.

The format of the JSON tag is documented on the `json.Encode` (https://golang.org/pkg/encoding/json/#Marshal) method.

```
type Person struct {
    Name    string `json:"name"`
    City    string `json:",omitempty"`
    Country string
}

func main() {
    p := Person{Name: "Dave", Country: "Australia"}
    var b bytes.Buffer
    enc := json.NewEncoder(&b)
    err := enc.Encode(p)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(b.String())
}
```

Run

128

JSON encoding kata

The code for this kata is in `$GOPATH/src/katas/jsonenc`

129

Writing HTTP servers

Go's `net/http` (https://golang.org/pkg/net/http) library can be used to write production web applications.

Writing web servers in Go can be as simple as a few lines. Here is an example:

```
package main

import (
    "fmt"
    "net/http"
)

func index(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "This is the index page")
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", index)
    http.ListenAndServe(":8000", mux)
}
```

Run

130

Writing http servers (cont.)

This is a simple HTTP handler

```
func index(w http.ResponseWriter, r *http.Request) {
       fmt.Fprintln(w, "This is the index page")
}
```

it takes two parameters

- `w`, a `http.ResponseWriter` which is used to send data to the client
- `r`, a `http.Request` which contains the uri, query parameters, and request body.

```
mux := http.NewServeMux()
```

`http.NewServeMux` returns a `ServeMux`, what we usually call a HTTP router. 131

Writing http servers (cont.)

```
mux.HandleFunc("/", index)
```

Registers our `index` function, with the top level route, `"/"`.

```
http.ListenAndServe(":8000", mux)
```

Opens a socket on port 8080 and sends and requests to our `mux` (our router).

`http.ListenAndServe` won't return unless something happens to that listening socket (wifi drops, cable unplugs)

go get

go get is a simple wrapper around `git` that knows how fetch packages from GitHub (and a few other places).

If a package's import path follows the go get convention, go get knows how to fetch the source code and download it to your `$GOPATH`.

go get isn't a complete dependency manager like rubygems or npm, etc, but it good enough for writing many applications.

We'll use go get to fetch the source for `gorilla/mux`, a more advanced HTTP router

```
% go get github.com/gorilla/mux
% ls $GOPATH/src/github.com/gorilla/mux
LICENSE                 context_gorilla.go      context_native_test.go  mux_test.go
README.md               context_gorilla_test.go doc.go                  old_test.go
bench_test.go           context_native.go       mux.go                  regexp.go
route.go
```

Using gorilla/mux

The standard HTTP router, `http.ServeMux` is quite basic, so we'll upgrade to the `gorilla/mux` http router.

```
package main

import (
    "fmt"
    "net/http"

    "github.com/gorilla/mux"
)

func greet(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    name := vars["name"]
    fmt.Fprintf(w, "今日は %s", name)
}

func main() {
    r := mux.NewRouter()
    r.HandleFunc("/greet/{name}", greet)
    http.ListenAndServe(":8000", r)
}
```

Run

Line counting HTTP service

Let's write a HTTP service that counts the lines in a book via http.

Features:

- When the client requests `/books/{book}` we look up the book and return the number of lines counted.
- The response to the client should be in JSON format and include the number of lines and the title of the book.
- Book directory is configurable.

```
% ./httplinecount ../../../books/ &
[1] 17554
% curl http://localhost:8080/books/moby.txt
{"title":"moby.txt","lines":"22659"}
```

The code for this kata is in `$GOPATH/src/katas/httplinecount` 135

Time for a break

136

Documenting packages with comments

Go code is traditionally documented with comments in the source code. This is similar to Python's heredoc convention.

Here are some examples

```
// simplestrings provides simple helper functions to work with strings
package simplestrings

// APIVersion is the version of this package's API
const APIVersion = 3

// NextID returns the next ID in the sequence
func NextID() uint64 { ... }
```

Notes

- Comments directly precede the thing they apply to, don't put an extra newline in between the comment and the symbol
- You should comment both Public and private symbols, but godoc will only show you the documents

Scope

We've talked about all the ways to declare a variable in Go, now we need to discuss scope.

```
package main

import "fmt"

func f(x int) {
    for x := 0; x < 10; x++ {
        fmt.Println(x)
    }
}

var x int

func main() {
    var x = 200
    f(x)
}
```

This program declares x four times. All four x 's are *different* because they exist in different scopes.

- Follow this link (https://play.golang.org/p/nIcOXVXgwl) and press the **Run** button.

Scope (cont.)

The scope of a declaration is bound to the closest pair of curly braces, `{` and `}`.

```
package main

import "fmt"

func f() {
    x := 200
    fmt.Println("inside f: x =", x)
}

func main() {
    x := 100
    fmt.Println("inside main: x =", x)
    f()
    fmt.Println("inside main: x =", x)
}
```

In this example, we declare `x` to be 100 inside `main`, and 200 inside `f`.

- Follow this link (https://play.golang.org/p/Xfi3GOhTiC) and press the **Run** button.
- Did the program print what you expected?

Scope (cont.)

What do you expect this program will print?

```
package main

import "fmt"

func f() {
    x := 200
    fmt.Println("inside f: x =", x)
}

var x = 50

func main() {
    fmt.Println("inside main: x =", x)
    x := 100
    fmt.Println("inside main: x =", x)
    f()
    fmt.Println("inside main: x =", x)
}
```

- Follow this link (https://play.golang.org/p/7uxrebFzmK) and press the **Run** button.
- Did you guess the right answer?

Scope (cont.)

What do you expect this program will print?

```
package main

import "fmt"

func main() {
    x := 100
    for i := 0; i < 5; i++ {
        x := i
        fmt.Println(x)
    }
    fmt.Println(x)
}
```

- Follow this link (https://play.golang.org/p/7hpZre9LhI) and press the **Run** button.
- Did you guess the right answer?

Shadowing

What you are seeing is called *shadowing*.

```
var x = 100

func main() {
    var x = 200
    fmt.Println(x)
}
```

Most of you will be comfortable with a *function scoped* variable shadowing a *package scoped* variable.

```
func f() {
    var x = 99
    if x > 90 {
        x := 60
        fmt.Println(x)
    }
}
```

But a *block scoped* variable shadowing a *function scoped* variable may be surprising.

Question time

143

Question time

This is your time for questions

Ask me anything!

144

Conclusion

145

Conclusion

```
:)
```

146

Thank you