
ON TRACK
with
GOLANG

Level 1–1

3, 2, 1… Go!

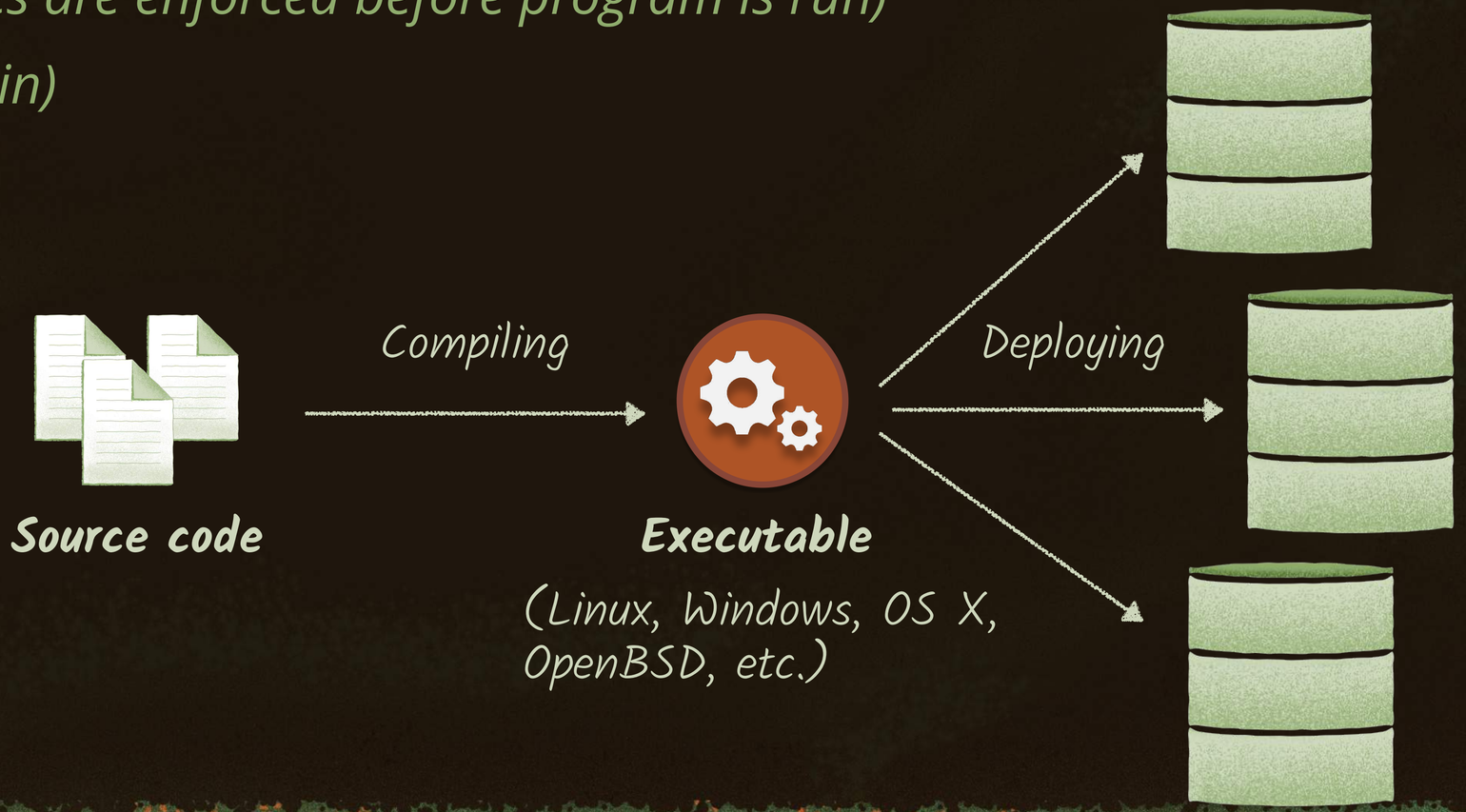
Taking the First Steps

ON TRACK with GOLANG

What Is Go?

Go is an **open-source** programming language created by Google in 2007. It makes it easy to build **simple, efficient** programs. Here are some of its main characteristics:

- Compiled *(compiler generates single executable file)*
- Statically typed *(types are enforced before program is run)*
- Fast *(concurrency built in)*
- Easy to deploy
- Fun to write!



Systems Programming

Go is a great language choice for writing lower-level programs that provide services to other systems. This type of programming is called **systems programming.**

Allows users to perform tasks
(Very common — your friends and family all use these)

Application Programs

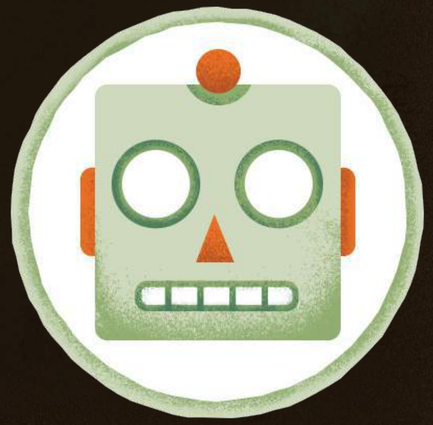
- E-commerce
- To-do lists
- Text editors
- Music players

vs.

Provides services to other systems
(By developers, for developers)

System Programs

- APIs
- Game engines
- Network applications
- CLI apps *(command line)*



What We'll Learn

In this course, we'll look at the most common features used when writing programs in Go.

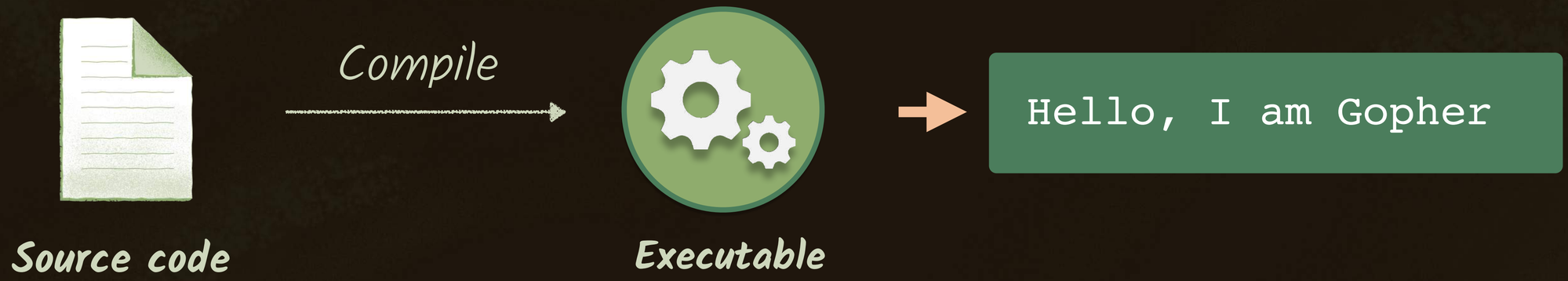
Things we'll learn:

- Building and running programs
- Importing and creating packages
- Basic constructs *(functions, variables, loops, conditionals)*
- Data types
- Concurrency with **goroutines**

ON TRACK with GOLANG

Our First Go Program

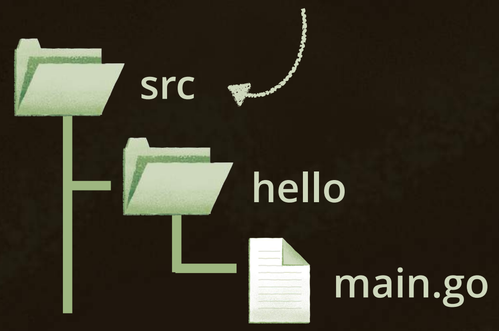
We'll start with a simple *Hello World* program in a single source code file. Once compiled, we'll use the executable file to run the program, which will print a message to the console.



main — the Entry Point

All runnable Go programs must have a main package and one main function.

The Go environment assumes all projects live under a src folder.



It's a convention in Go to use main as the name of single file programs.

src/hello/main.go

```
package main

func main() {

}
```

A package definition is always the very first thing in a Go source file.

The main() function is the entry point for all Go programs. It MUST have this name.

The func keyword declares a new function.

Printing From main()

Package **imports** go after package definition. The Println() function belongs to the fmt package.

src/hello/main.go

```
package main

import "fmt"

func main() {
  fmt.Println("Hello, I am Gopher")
}
```

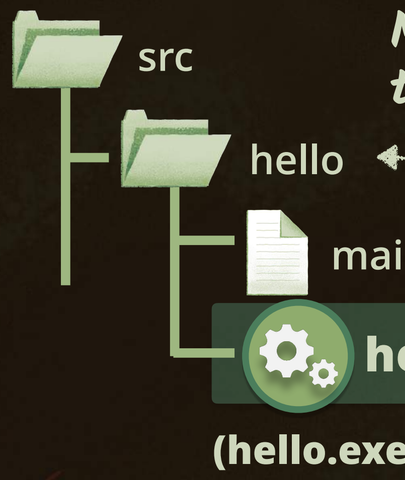
Packages used by the program must be explicitly imported.

No semicolons necessary!

This function from the fmt package prints a message to the standard output (a.k.a., console).

Building and Running With go build

A **compiler** reads source code and produces one executable file. We call this the **build process.**



(hello.exe *on Windows)*

Must be run from the project root

Output of the build command on Unix machines

src/hello/main.go

```
...
```

The build command is shipped with Go...

```
$ go build
```

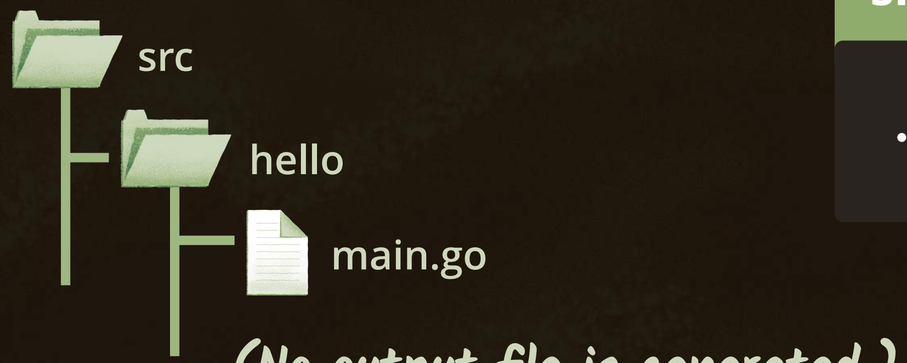
...it produces a single executable file named after the project.

```
$ ./hello
```

```
Hello, I am Gopher
```

Running with go run

Using go run**, we can build and run programs in one command, which makes things a bit easier.**



(No output file is generated.)

src/hello/main.go

```
...
```

The run command is shipped with Go...

```
$ go run main.go
```

```
Hello, I am Gopher
```

...it builds AND runs our code straight from the source file.

The gofmt Command

The gofmt command ships with Go and can be used to **automatically format** source code.

Valid syntax but hard to understand

src/hello/main.go

```
package
main
import"fmt"
func main(){
fmt.Println("Hello, I am Gopher")}
```

```
$ gofmt -w main.go
```

The -w flag is used to write the results to the original source file instead of printing to console.

Valid syntax AND easy to understand!

src/hello/main.go

```
package main

import "fmt"

func main() {
  fmt.Println("Hello, I am Gopher")
}
```

Formatting Source Code With gofmt

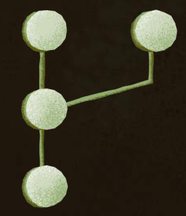
There are many benefits when following gofmt's formatting rules.

- Uncontroversial decisions *(less time spent arguing about code style)*
- Easier to **write, read**, and **maintain** programs

Many plugins are available for text editors, IDEs, and version control systems that make it easy to run gofmt automatically.

gofmt ***could be configured to run on every file save...***

...or before each commit.



http://go.codeschool.com/gofmt

Level 1–2

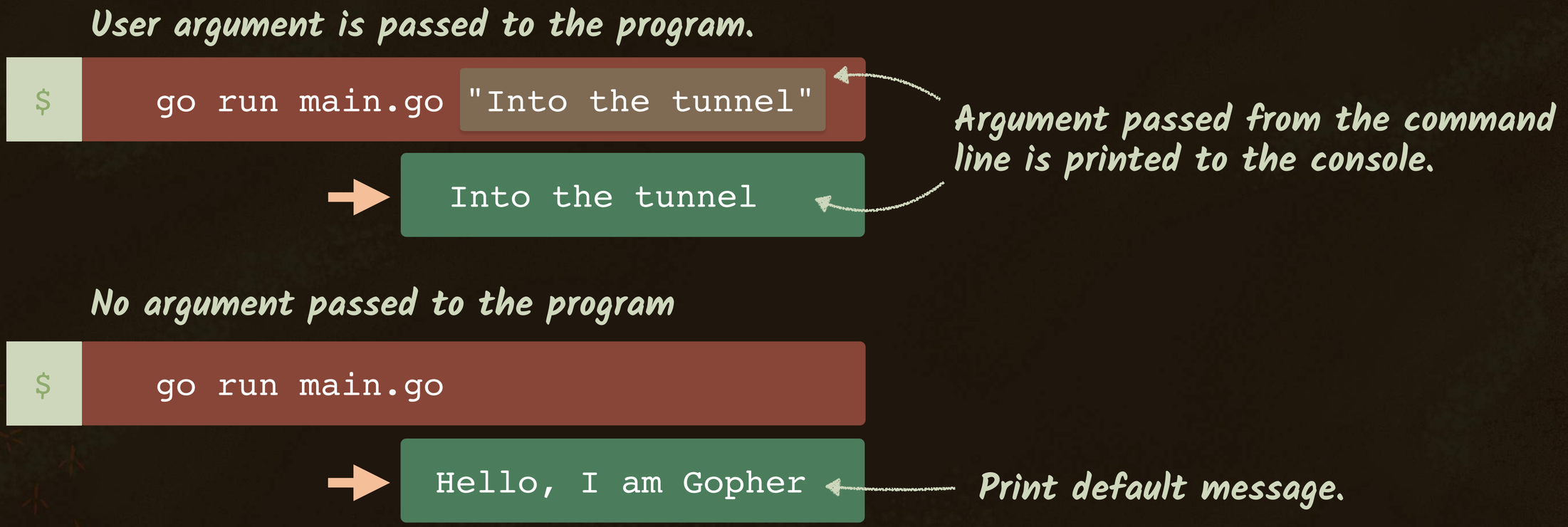
3, 2, 1… Go!

Conditionals, Args & Imports

ON TRACK with GOLANG

Printing Two Different Messages

We will write a program that reads a **user-supplied argument** from the command line and prints it to the console. If no argument is given, then a **default message** is printed.



Using Conditionals

There are no parentheses in if/else statements, and blocks **must be brace-delimited.**

src/hello/main.go

```
package main

import "fmt"

func main() {
  if                    {

  } else {

  }
}
```

Boolean expressions go here, and no parentheses are necessary.

Using Conditionals

The len() built-in function returns the **length of its argument.**

src/hello/main.go

```
package main

import "fmt"

func main() {
  if len(        ) > 1 {

  } else {

  }
}
```

Statement evaluates to true if len() returns a number greater than 1.

Built-in functions are functions that can be invoked directly without us having to import a package.

Reading Arguments

The os.Args array contains all arguments passed to the running program, including user-supplied arguments from the **command line.**

```go
package main

import (
  "fmt"
  "os"
)

func main() {
  if len(os.Args) > 1 {

  } else {

  }
}
```

Import package from the standard library

What a command-line argument looks like

```
$ go run main.go "Into the tunnel"
```

`os.Args[0]`

`os.Args[1]`

```
/var/folders/(...)/command-line-arguments/_obj/exe/main
```

Name of the temporary executable created by the go run *command*

An array with the program arguments, starting with the name of the executable and followed by any user-supplied arguments

Printing Arguments

We invoke fmt.Println() from both the if and else blocks. First, we pass it the user-supplied command-line argument (os.Args[1]), and, on the second block, a default greeting message.

```
package main

import (
  "fmt"
  "os"
)

func main() {
  if len(os.Args) > 1 {
    fmt.Println(os.Args[1])
  } else {
    fmt.Println("Hello, I am Gopher")
  }
}
```

User-supplied arguments start on index 1 of the array.

If no arguments are passed, then we print a default greeting message.

Running the Program With Arguments

If given an argument, then our program will print this argument to the console.

```
package main

import (
  "fmt"
  "os"
)

func main() {
  if len(os.Args) > 1 {
    fmt.Println(os.Args[1])
  } else {
    fmt.Println("Hello, I am Gopher")
  }
}
```

Returns 2...

...the if block is run...

```
$ go run main.go "Into the tunnel"
```

os.Args[1]

```
Into the tunnel
```

...and the argument is printed to the console.

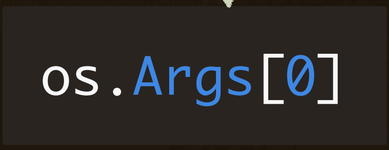
Running the Program With No Arguments

If **no argument** is supplied, then our program will print the default message to the console.

```go
package main

import (
  "fmt"
  "os"
)

func main() {
  if len(os.Args) > 1 {
    fmt.Println(os.Args[1])
  } else {
    fmt.Println("Hello, I am Gopher")
  }
}
```

Returns 1...

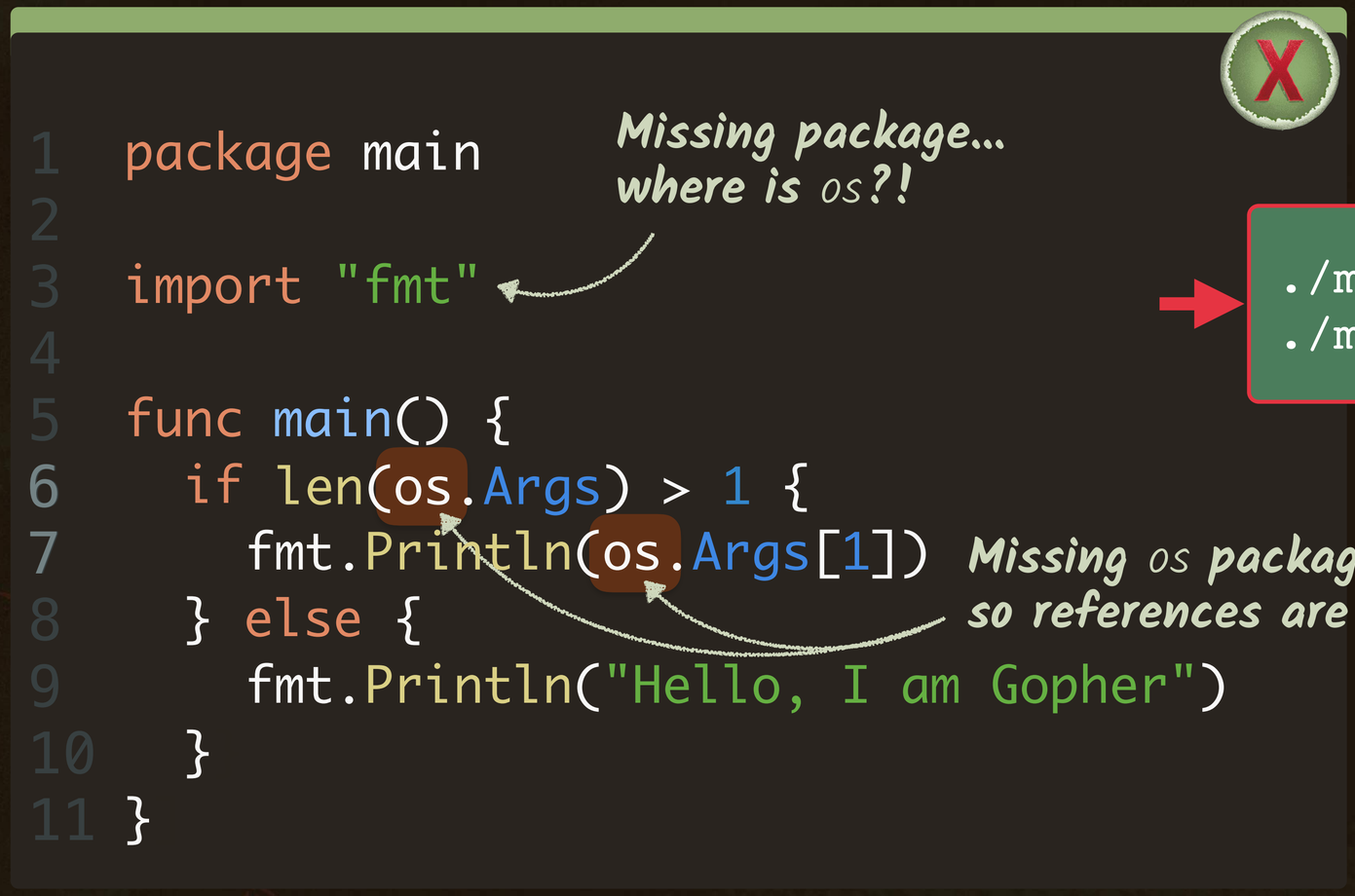
...the else *block is run...*



...and the default message is printed to the console.

Running With Missing Imports

Any missing package imports will **raise an error** during the build process.



```
1  package main
2
3  import "fmt"
4
5  func main() {
6    if len(os.Args) > 1 {
7      fmt.Println(os.Args[1])
8    } else {
9      fmt.Println("Hello, I am Gopher")
10   }
11 }
```

Missing package… where is os?!

Missing os package import, so references are invalid

```
$ go run main.go
```

```
./main.go:6: undefined: os in os.Args
./main.go:7: undefined: os in os.Args
```

The goimports Command

The goimports command ships with Go. It **detects missing packages** and automatically updates import statements in the source code.

The -w flag writes results to the original file instead of printing to the console.

```
$ goimports -w main.go
```

```
package main

import "fmt"

func main() {
  if len(os.Args) > 1 {
    fmt.Println(os.Args[1])
  } else {
    fmt.Println("Hello, Gopher")
  }
}
```

Detects os is being used, but it's not imported

```
package main

import (
  "fmt"
  "os"
)

func main() {
  if len(os.Args) > 1 {
    fmt.Print(os.Args[1])
  } ...
```

Adds missing package and formats code

Running With Fixed Imports

With the necessary packages imported, we can now run our program successfully.

Level 2–1

Underneath the Tracks

Variables & Types

ON TRACK with GOLANG

Repetitive References

We are currently referencing the same os.Args array from two different places. This is the start of what could become a code smell and unnecessary repetition.

```
package main

import
...

func main() {
  if len(os.Args) > 1 {
    fmt.Println(os.Args[1])
  } else {
    fmt.Println("Hello, Gopher")
  }
}
```

Multiple references to the same array

Declaring Variables With Type Inference

The `:=` operator tells Go to **automatically find out the data type** on the right being assigned to the **newly declared variable** on the left. This is known as **type inference.**

```
package main

import
...

func main() {
  args := os.Args
  if len(args) > 1 {
    fmt.Println(args[1])
  } else {
    fmt.Println("Hello, Gopher")
  }
}
```

Automatically finds out data type for the value returned from os.Args

Storing Values as Data Types

Given a value, we must determine **how much space** is needed to store this value for later reuse.

A value, like 1 *or* "hello"

HERE'S AN ACORN...

A data type, like int *or* string

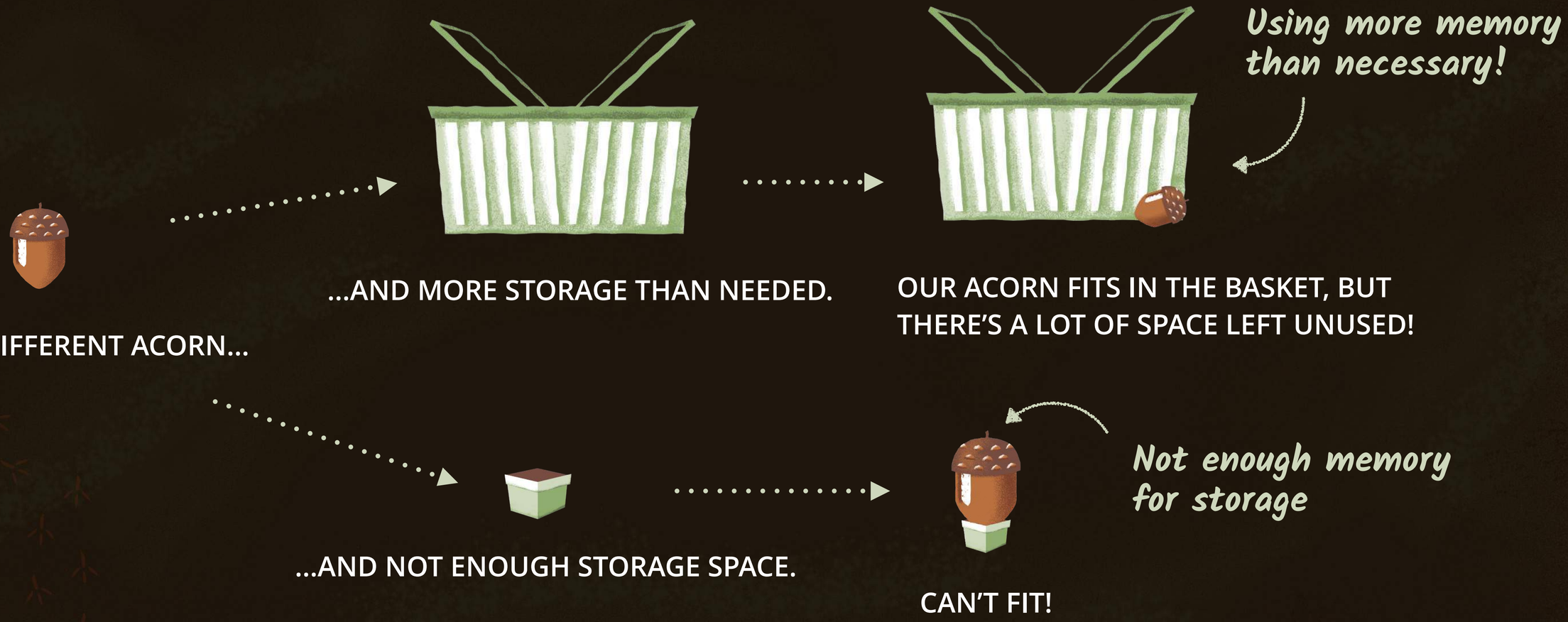
...AND SOME AVAILABLE STORAGE SPACE: A BASKET.

Assigning a value to a variable of a specific data type

THE BASKET CAN PERFECTLY ACCOMMODATE THE ACORN.

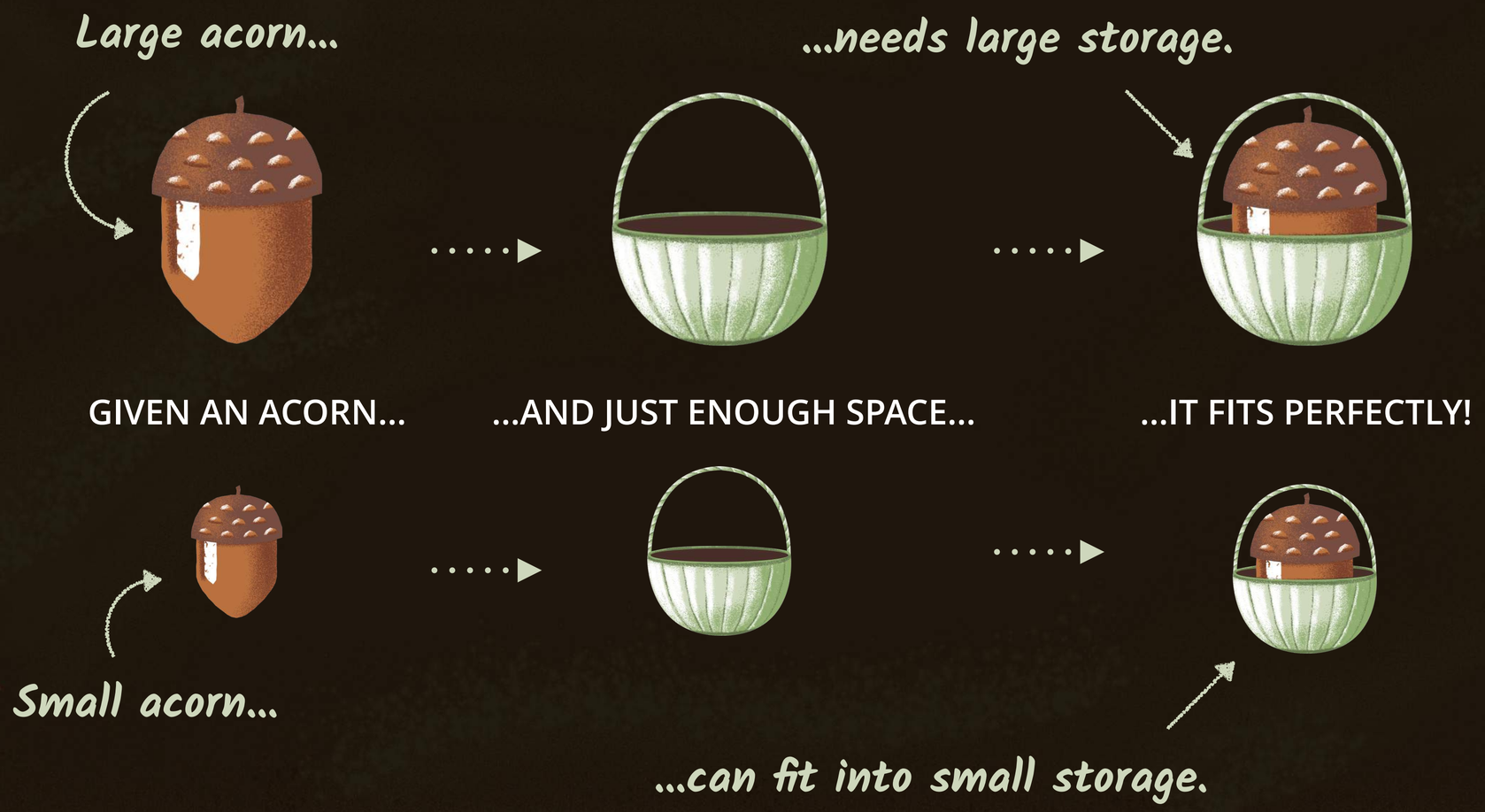
One Size Does Not Fit All

Taking up too much storage is a **waste of precious space**. On the other hand, not reserving enough space can limit the amount of data we can store.



Storing Values as Data Types

For every value, there's always a **proper data type** that determines how the value should be stored and the operations that can be done on values of that type.



Type Inference vs. Manual Type Declaration

There are **two ways** to declare variables in Go: **type inference** and **manual type declaration.**

Type Inference

Data type is inferred during assignment.

```
<variable-name> := <some-value>
```

Notice the special := operator.

The most common and preferred way

```
message := "Hello, Gopher"
```

Manual Type Declaration

Data type is declared prior to assignment.

```
var <variable-name> <data-type>
<variable-name> = <some-value>
```

The var keyword...

...and the = operator.

More verbose, but useful and often necessary

```
var message string
message = "Hello, Gopher"
```

How Static Typing Can Help

Static typing allows the Go compiler to check for type errors **before the program is run.**

Assigning a different value than what was expected makes the Go compiler mad... 42 is NOT a string!

```
var age string
age = 42
```

```
./main.go:5: cannot use 42 (type int)
                as type string in assignment
```

No errors when the correct data type is used... 42 is an int!

```
var age int
age = 42
```

Type Inference Requires Less Code

Most times, type inference and manual type declaration **can be used interchangeably,** but type inference is **less code to write and read.**

```
package main

import
...

func main() {
  args := os.Args
  if len(args) > 1 {
    fmt.Println(args[1])
  } else {
    fmt.Println("Hello, Gopher")
  }
}
```

Same thing

Manually declaring type

```
var args []string
args = os.Args
```

Brackets prefix indicates this is a collection of strings.

Common Data Types

Here are a few built-in data types commonly found in most Go programs.

Type	Data
int	42
string	"Hello"
bool	true or false
[]string	["acorn", "basket"]

Also called primitive data values

Level 2–2

Underneath the Tracks

Functions

ON TRACK with GOLANG

The Greeting Program

Let's write a program that **outputs a greeting message**. The message will be different depending on which hour of the day we run our program.

Grabbing Current Time

We'll start by grabbing the **current hour** of the day.

src/hello/main.go

```
package main

import (
  "fmt"
  "time"
)

func main() {
  hourOfDay := time.Now().Hour()

}
```

Import package from standard library.

Assign return value to a new variable using type inference.

The getGreeting() Function Signature

Named functions in Go must have **a name**, followed by **any arguments** they expect, and end with the **data type** they return (if any). This is commonly referred to as a **function signature.**

src/hello/main.go

```
...

func main() {
  hourOfDay := time.Now().Hour()
  greeting := getGreeting(hourOfDay)
}

func getGreeting(hour int) string {

}
```

Calling the new function passing the current hour of the day as its single argument

How do we find out which data type this is?

Expected return type for this function

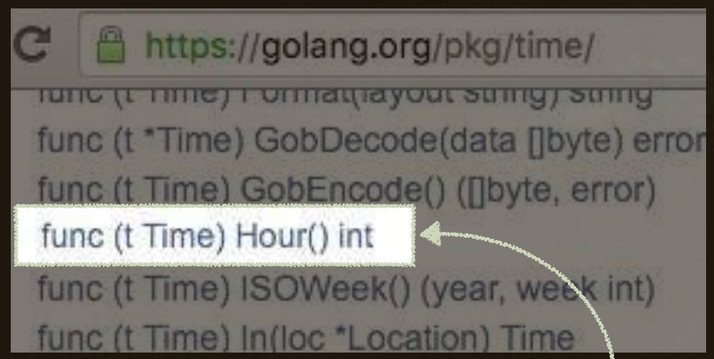
The Return Data Type

One way to find out the **data type returned by a function** from the Go standard library is by referring to the official documentation.

src/hello/main.go

```
...

func main() {
  hourOfDay := time.Now().Hour()
  greeting := getGreeting(hourOfDay)
}


func getGreeting(hour int) string {

}
```

http://go.codeschool.com/go-time-package



According to the docs for the time package, Hour() returns a value of type int.

The complete function signature

Returning From getGreeting()

We'll return a different greeting message based on the hour of the day passed as argument.

src/hello/main.go

```
...

func getGreeting(hour int) string {
  if hour < 12 {
    return "Good Morning"
  } else if hour < 18 {
    return "Good Afternoon"
  } else {
    return "Good Evening"
  }
}
```

We can nest if*s inside* else *clauses.*

Returning From getGreeting()

We run our program and see a different output depending on the current time of day.

src/hello/main.go

```
...

func main() {
  hourOfDay := time.Now().Hour()
  greeting := getGreeting(hourOfDay)
  fmt.Println(greeting)
}

func getGreeting(hour int) string {

}
```


```
$ go run main.go
```

→ Good Morning


```
$ go run main.go
```

→ Good Afternoon


```
$ go run main.go
```

→ Good Evening

Level 2–3

Underneath the Tracks

Working With Errors

ON TRACK with GOLANG

Don’t Wake the Gophers Up!

This program should not be allowed to run before 7AM. In case this happens, it should terminate immediately and return an **error code**.

AM 8:00

```
$ go run main.go
```

```
Good Morning
```

PM 2:00

```
$ go run main.go
```

```
Good Afternoon
```

PM 7:00

```
$ go run main.go
```

```
Good Evening
```

AM 6:00

```
$ go run main.go
```

```
Too early for greetings!
```

Our gopher friends are still asleep...

Declaring Multiple Return Values

In Go, we communicate errors via a **separate return value**. Let's update our function signature to return **two different values**: a string and an error.

src/hello/main.go

```
...

func getGreeting(hour int) (string, error) {

  if hour < 7 {


  }

  ...
}
```

Two values will now be returned from this function.

If it's earlier than 7AM, this block will be executed.

Returning With Error

If invoked before 7AM, the getGreeting() function will return an empty string **and a new error.**

```
...
import (
  "errors"
  ...
)

func getGreeting(hour int) (string, error) {
  var message string

  if hour < 7 {
    err := errors.New("Too early for greetings!")
    return message, err
  }
  ...
}
```

Import package from standard library.

Manually declaring the variable and data type.

Assigning a new error and returning it alongside an empty string message

Has not been assigned a value at this point, so defaults to zero value of empty string

Zero Values

A zero value in Go is the **default value** assigned to variables declared without an explicit initial value.

```
var message string
fmt.Print(message)
```

→ `""`

```
var age int
fmt.Print(age)
```

→ `0`

```
var isAdmin bool
fmt.Print(isAdmin)
```

→ `false`

http://go.codeschool.com/go-zero-value

Type	Zero Value
float	`0.0`
byte	`0`
function	`nil`
etc...	

Every primitive data type has an associated zero value to it.

Assigning a Message

We determine the appropriate greeting and assign it to the previously declared message variable.

```
...

func getGreeting(hour int) (string, error) {


   ...
   if hour < 7 { ... }

   if hour < 12 {
      message = "Good Morning"
   } else if hour < 18 {
      message = "Good Afternoon"
   } else {
      message = "Good Evening"
   }

}
```

Using = to assign a value because variable was manually declared previously

Returning With No Error

We use an **explicit nil** as the second return value. This indicates the function ran with no errors.

```
...

func getGreeting(hour int) (string, error) {
  ...
  if hour < 7 { ... }

  if hour < 12 {
    message = "Good Morning"
  } else if hour < 18 {
    message = "Good Afternoon"
  } else {
    message = "Good Evening"
  }
  return message, nil
}
```

A nil value for error tells the caller this function has no error.

Reading Multiple Values From a Function

We can assign multiple values at once by separating the new variables using a comma.

```
...

func main() {
  hourOfDay := time.Now().Hour()
  greeting, err := getGreeting(hourOfDay)

  fmt.Print(greeting)
}

func getGreeting(hour int) (string, error) {
  ...
}
```

Two values are now being returned from `getGreeting()`.

Checking for Errors

It is a common practice in Go to always check if an error exists before proceeding.

```
...

func main() {
  hourOfDay := time.Now().Hour()
  greeting, err := getGreeting(hourOfDay)
  if err != nil {



  }
  fmt.Print(greeting)
}

func getGreeting(hour int) (string, error) {
  ...
}
```

If err is NOT nil, then some error must have occurred!

Exiting With Error

The exit code 1 is a **POSIX standard** indicating the program has finished, but **errors have occurred.**

```
import (
  "os"
  ...
)

func main() {
  hourOfDay := time.Now().Hour()
  greeting, err := getGreeting(hourOfDay)
  if err != nil {
    fmt.Println(err)
    os.Exit(1)
  }
  fmt.Print(greeting)
}
...
```

We import our old friend, the os package.

Prints error to the console.

The Exit function takes an exit code of type int as argument and causes the program to terminate immediately.

Running the Complete Code

If we run the code now, it still prints all messages just like before and also an error message if it's before 7AM.

AM 8:00

```
$ go run main.go
```

```
Good Morning
```

PM 2:00

```
$ go run main.go
```

```
Good Afternoon
```

PM 7:00

```
$ go run main.go
```

```
Good Evening
```

AM 6:00

```
$ go run main.go
```

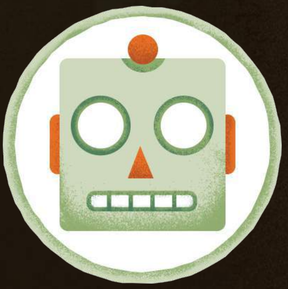
```
Too early for greetings!
```

Our gopher friends can now sleep in peace.

Where is the **exit code**?

Exit codes are used by **other programs** so they know whether or not an error occurred.

(Remember systems programming?)



Level 3–1

Following the Trail

The *for* Loop

ON TRACK with GOLANG

The Only Looping Construct in Go

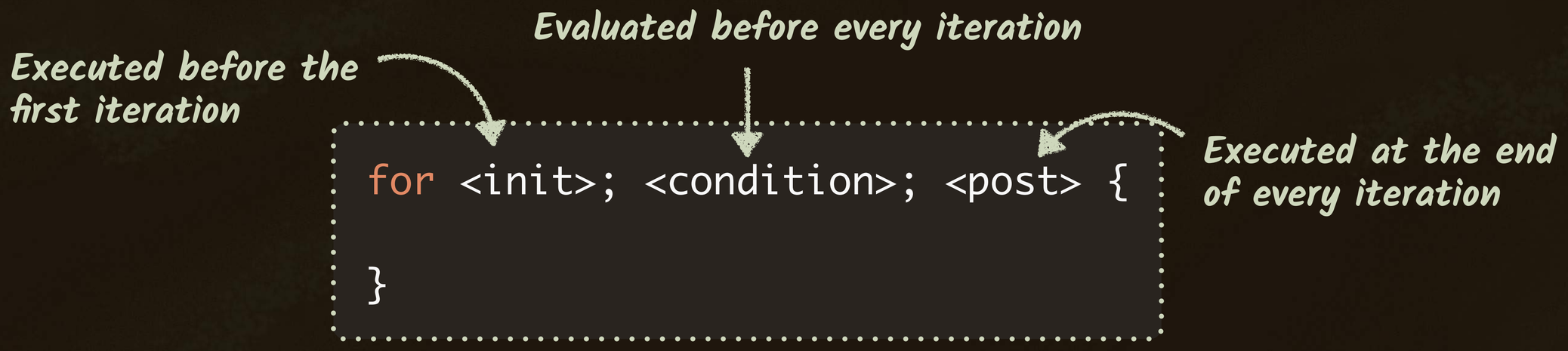
Unlike other popular languages, the for loop is **the only looping construct** in Go.

Other Languages →
- *for*
- *while*
- *do/while*
- *each, map, filter, find, indexOf, etc.*

Go → ***for***

The for Loop

There are **no parentheses** in for loops and three different components we can use to control the loop: the **init** statement, a **condition** expression, and a **post** statement.



A Complete for Loop

We can use the `:=` symbol on the **init** statement to create new variables using type inference.

src/hello/main.go

```
package main

import "fmt"

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

Create new variable using type inference.

Increments variable by 1 after each run.

Run this loop five times.

```
$ go run main.go
```

```
0
1
2
3
4
```

Print numbers from 0 to 4.

A for Loop With a Single Condition

The for loop components are **optional**. We can create loops with variables declared previously in the code and a **single condition expression.**

```
for <condition> {

}
```

Leave out init and post components.

src/hello/main.go

```
...

func main() {
  i := 0
  isLessThanFive := true
  for isLessThanFive {


    fmt.Println(i)
    i++
  }
}
```

Declare variables and assign initial values.

As long as condition expression is true, the loop will continue to run.

Increment counter i at the end of every run of the loop.

Breaking With a Condition

In order to break from a for loop with no **post** statement, we can change the variable used in the condition expression from inside the body of the loop.

src/hello/main.go

```
...

func main() {
  i := 0
  isLessThanFive := true
  for isLessThanFive {
    if i >= 5 {
      isLessThanFive = false
    }
    fmt.Println(i)
    i++
  }
}
```

```
$ go run main.go
```

```
0
1
2
3
4
5
```

Print numbers from 0 to 5.

Change the condition expression and stops the loop before the next run.

Writing for Loops With No Components

It's also common to write for loops with **no components at all.** To break out of these loops, we can use the break keyword.

src/hello/main.go

```
...

func main() {
  i := 0

  for {
    if i >= 5 {
      break
    }
    fmt.Println(i)
    i++
  }
}
```

The loop stops immediately.

Leave out ALL components.

```
for {
  ...
  break
  ...
}
```

Exit from the loop.

```
$ go run main.go
```

```
0
1
2
3
4
```

Does NOT print number 5

Writing Infinite Loops

Infinite loops are widely used in networking programs. They are useful for **setting up listeners** and **responding to connections.**

src/hello/main.go

```
...

func main() {

  for {
    someListeningFunction()
  }
}
```

Some function listening for connections from other programs

```
for {
  ...
}
```

Runs indefinitely

```
$ go run main.go
```

→ *(...)*

Does NOT exit the process

Level 3–2

Following the Trail

Arrays & Slices

ON TRACK with GOLANG

Declaring Arrays

When creating arrays via manual type declaration, we must set the **max number of elements.**

src/hello/main.go

```go
package main

import "fmt"

func main() {
  var langs [3]string

  fmt.Println(langs)
}
```

Holds no more than 3 values of type string

```
$ go run main.go
[ ]
```

Writing to Arrays

We can add elements to arrays by assigning to each specific index.

src/hello/main.go

```go
package main

import "fmt"

func main() {
  var langs [3]string

  langs[0] = "Go"
  langs[1] = "Ruby"
  langs[2] = "JavaScript"

  fmt.Println(langs)
}
```

Index count starts at 0.

```
$ go run main.go
[Go Ruby JavaScript]
```

Arrays Are Not Dynamic

Adding more elements to an array than what was initially expected **raises an out-of-bounds error.**

src/hello/main.go

```
1   package main
2
3   import "fmt"
4
5   func main() {
6     var langs [3]string
7
8     langs[0] = "Go"
9     langs[1] = "Ruby"
10    langs[2] = "JavaScript"
11    langs[3] = "LOLcode"
12    fmt.Println(langs)
13  }
```

Adding to nonexistent space

```
$ go run main.go
./main.go:11: invalid array index 3 (out of
              bounds for 3-element array)
```

Slices Are Like Arrays

The **slice** type is built on top of arrays to provide more power and convenience. Most array programming in Go is done with **slices** rather than simple arrays.

```
package main

import "fmt"

func main() {
  var langs []string

  fmt.Println(langs)
}
```

Leaving out max elements creates a slice with a zero value of nil.

```
$ go run main.go
[ ]
```

Slices Are Dynamic

A **nil** slice in Go behaves **the same as an empty slice**. It can be appended to using the built-in function append(), which takes two arguments: a slice and a variable number of elements.

```go
package main

import "fmt"

func main() {
  var langs []string

  langs = append(langs, "Go")
  langs = append(langs, "Ruby")
  langs = append(langs, "JavaScript")
  langs = append(langs, "LOLcode")
  fmt.Println(langs)
}
```

Returns a new slice that contains the new element

```
$ go run main.go
[Go Ruby JavaScript LOLcode]
```

If capacity is not sufficient, a new underlying array will be allocated.

Level 3–3

Following the Trail

Slice Literals & Looping With *range*

ON TRACK with GOLANG

Creating Slices With Initial Values

When we know beforehand which elements will be part of a slice, multiple calls to append will start looking **too verbose**. There's a better way...

```go
package main

import "fmt"

func main() {
  var a []string

  langs = append(langs, "Go")
  langs = append(langs, "Ruby")
  langs = append(langs, "JavaScript")

  fmt.Println(langs)
}
```

```
$ go run main.go
[Go Ruby JavaScript]
```

We can use a shorter syntax to create this slice in one line.

Slice Literals

A **slice literal** is a quick way to create slices with initial elements via type inference. We can pass elements between curly braces `{}`.

```go
package main

import "fmt"

func main() {
  langs := []string{"Go", "Ruby", "JavaScript"}
  fmt.Println(langs)
}
```

```
$ go run main.go
[Go Ruby JavaScript]
```

Element count is inferred from the number of initial elements.

Reading From a Slice by Index

One way to read from a slice is to **access elements by index**, just like an array.

```go
package main

import "fmt"

func main() {
  langs := []string{"Go", "Ruby", "JavaScript"}
  fmt.Println(langs[0])
  fmt.Println(langs[1])
  fmt.Println(langs[2])
}
```

Prints each individual element to the console.

```
$ go run main.go
```

```
Go
Ruby
JavaScript
```

Reading From a Slice With Unknown Size

While reading elements by index works, it doesn't scale well once our slice grows or when the exact number of elements **is not known until the program is run.**

```
...

func main() {
  langs := getLangs()

  fmt.Println(langs[???])

}

func getLangs() []string {
  ...
}
```

Based on the function signature, we can see this returns a slice, but the total number of elements is unknown.

Can't tell which index is valid.

The function signature tells us a slice will be returned from this function call.

Navigating a Slice With for and range

The range clause provides a way to iterate over slices. When only one value is used on the left of range, then this value will be the index for each element on the slice, one at a time.

```
...

func main() {
  langs := getLangs()

  for i := range langs {

  }
}

func getLangs() []string {
  ...
}
```

Loop runs once for each value in langs.

The index for each element is returned on each run of the loop.

Navigating a Slice With for and range

By using range on a slice, we can be sure the indices used are **always valid** for that slice.

```
...

func main() {
  langs := getLangs()

  for i := range langs {
    fmt.Println(langs[i])
  }
}

func getLangs() []string {
  ...
}
```

We can now safely use the index (of type int) to fetch each element from the slice...

```
$ go run main.go

Go
Ruby
JavaScript
```

...and print them to the console.

Unused Variables Produce Errors

Using range we can also read **the index** and **the element associated with it** at the same time. However, if we don't use a variable that's been assigned, then Go will produce an error.

```
...

func main() {
  langs := getLangs()

  for i, element := range langs {
    fmt.Println(element)
  }
}

func getLangs() []string {
  ...
}
```

Not used anywhere else in the code. This will produce an error!

For each run of the loop, this variable is assigned each actual element from the slice.

```
$ go run main.go
```

```
./main.go:8: i declared and not used
```

Ignoring Unused Variables With Underline

The underline character tells Go this value will not be used from anywhere else in the code.

```
...

func main() {
  langs := getLangs()

  for _, element := range langs {
    fmt.Println(element)
  }
}

func getLangs() []string {
  ...
}
```

We use the underline character to indicate variables that will not be used.

```
$ go run main.go
```

```
Go
Ruby
JavaScript
```

When given a single identifier, range assigns index, NOT the element.

```
for element := range langs {
  fmt.Println(element)
}
```

```
0
1
2
```

Level 4–1

Adding Structure to the Data

Structs

ON TRACK with GOLANG

Young Gophers Can Jump HIGH

Not many people know this, but a gopher's ability to jump is based on their age.

```
gopher1Name := "Phil"
gopher1Age := 30

if gopher1Age < 65 {
  highJump(gopher1Name)
} else {
  ...
}

func highJump(name string) {
  fmt.Println(name, "can jump HIGH")
}
```

Values for name and age are assigned to individual variables.

Phil can jump pretty high.

```
$ go run main.go
```

```
Phil can jump HIGH
```

Older Gophers Can Still Jump

Despite the odds, the more experienced gophers can still keep up with the youngsters!

```
...
gopher2Name := "Noodles"
gopher2Age := 90

if gopher2Age < 65 {
  ...
} else {
  lowJump(gopher2Name)
}

func lowJump(name string) {
  fmt.Println(name, "can still jump")
}
```

Two new values and two new variables

Although not as young as Phil, Noodles can still jump.

```
$ go run main.go
Noodles can still jump
```

Too Much Code at Once

Things start looking confusing when we begin working with multiple gophers. This is a sign our code is **leaking logic details.**

```
gopher1Name := "Phil"
gopher1Age := 30
gopher2Name := "Noodles"
gopher2Age := 90
```

Each new gopher requires TWO independent variables...

```
if gopher1Age < 65 {
   ...
}
if gopher2Age < 65 {
   ...
}
```

...and an additional `if` statement.

Being part of the same file, logic rules are exposed to caller of this code.

```
func highJump(name string) { ... }
func lowJump(name string) { ... }
```

```
$ go run main.go
```

```
Phil can jump HIGH
Noodles can still jump
```

Hiding Details

Even in procedural languages like Go, there are ways we can **hide unnecessary implementation details** from the caller. This practice is also known as **encapsulation.**

```
gopher1 := gopher{name: "Phil", age: 30}
gopher2 := gopher{name: "Noodles", age: 90}

fmt.Println(gopher1.jump())
fmt.Println(gopher2.jump())
```

You might not know what these mean just yet, but I bet they look intuitive...

Leaking implementation details

Encapsulating implementation details

How do we get here?

Declaring a New struct

We'll declare a new struct type for a gopher. A struct is a built-in type that allows us to group properties under a single name.

The name of the struct

The underlying primitive type

```
type gopher struct {
  name string
  age  int
}
```

Properties are variables internal to the struct.

The type keyword indicates a new type is about to be declared.

Creating a struct

The most common way to allocate memory and assign values to a struct is by calling its name, followed by the initial data wrapped in curly braces.

```
type gopher struct {
  name string
  age  int
}

func main() {
  gopher1 := gopher{name: "Phil", age: 30}
  gopher2 := gopher{name: "Noodles", age: 90}
}
```

Must be placed outside the main function.

Allocates memory and assigns result to variables

Using struct for Encapsulation of Behavior

A struct contains behavior in the form of **methods**. The way we define methods on a struct is by writing a regular function and specifying **the** struct **as the explicit receiver.**

```
type gopher struct {
  name string
  age  int
}

func (g gopher) jump() string {


}

func main() {

}
```

This is how we specify an explicit receiver for this function.

Using struct for Encapsulation of Behavior

From inside the method, we can access properties from the struct via the **explicit receiver.**

```
type gopher struct {
  name string
  age  int
}

func (g gopher) jump() string {
  if g.age < 65 {
    return g.name + " can jump HIGH"
  }
  return g.name + " can still jump"
}
```

Properties from the struct

Calling Methods

We can now call jump() on all gophers and avoid exposing the “jump logic” to the caller of this method.

```
type gopher struct { ... }

func (g gopher) jump() string {
  if g.age < 65 {
    return g.name + " can jump HIGH"
  }
  return g.name + " can still jump"
}

func main() {
  gopher1 := gopher{name: "Phil", age: 30}
  fmt.Println(gopher1.jump())
  gopher2 := gopher{name: "Noodles", age: 90}
  fmt.Println(gopher2.jump())
}
```

```
$ go run main.go
```

```
Phil can jump HIGH
Noodles can still jump
```

The jump() method acts on its receiver.

The “Tell, Don’t Ask” Principle

Rather than **asking** for data and acting on it, we instead **tell** the program what to do.

Asking

Asking for age and checking...

```
if gopherAge < 65 {
   highJump(gopherName)
} else {
   lowJump(gopherName)
}
```

...whether it has a high jump...

...or a low jump.

```
func highJump(name string) {
   ...
}
func lowJump(name string) {
   ...
}
```

VS.

Telling

```
type gopher struct {
   ...
}

func (g gopher) jump() string {
   ...
}

gopher1.jump()
```

*Telling gopher **what to do. Logic is encapsulated and hidden from the caller.***

Level 4–2

Adding Structure to the Data

Working With Pointers

ON TRACK with GOLANG

Validating a Gopher's Age

A new function will set a value on a property from the struct passed as argument.

```
type gopher struct {
  name string
  age  int
  isAdult bool
}
```

New property will determine whether a gopher is an adult.

A new function takes type gopher.

```
<new-function>(gopher)
```

If gopher is of age, property is set to true...

```
gopher.isAdult = true
```

...otherwise, it is set to false.

```
gopher.isAdult = false
```

This pattern of **modifying arguments passed to functions** can be found in parts of the Go standard library. The Scan() method from the database package is an example.

http://go.codeschool.com/go-db-sql

func (*Rows) Scan

```
func (rs *Rows) Scan(dest ...interface{}) error
```

Scan copies the columns in the current row into the values pointed at by dest. The number of values in dest must be the same as the number of columns in Rows.

New Property Defaults to Zero Value

The isAdult property from all new gophers defaults to **false** — the zero value for type bool, remember?

```
type gopher struct {
  name string
  age  int
  isAdult bool
}

func main() {
  phil := gopher{name: "Phil", age: 35}
  fmt.Println(phil)
}
```

```
$ go run main.go
{Phil 35 false}
```

No value assigned to isAdult, so it defaults to false.

Writing a Validation Function

The new function takes one argument and writes to the isAdult property of this argument. The function does **NOT** return anything.

```
type gopher struct {
  ...
}

func main() {
  phil := gopher{name: "Phil", age: 35}
  validateAge(phil)

}

func validateAge(g  ???? ) {
  g.isAdult = g.age >= 21
}
```

Passing type gopher as argument

Must accept a compatible type.

Passing structs **by Value**

Passing a struct as argument **creates a copy** of all the values assigned to its properties.

```
type gopher struct {
  name string
  age  int
  isAdult bool
}

func main() {
  phil := gopher{name: "Phil", age: 35}
  validateAge(phil)
  fmt.Println(phil)
}

func validateAge(g gopher) {
  g.isAdult = g.age >= 21
}
```

Creates a COPY of the original struct data...

...and the COPY of the data is received as argument.

Assigns true to the COPY of the data — not the original data!

```
$ go run main.go
{Phil 35 false}
```

Original value from struct is NOT changed.

Values and References in Music Playlists

Thinking about how playlists work can help us understand the difference between values and references.

Original songs from each artist's album

Albums

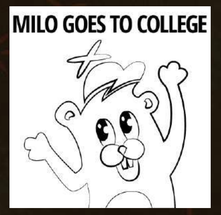
Are You Gonna Go My Way

1 - Song **a**

2 - Song **b**

3 - Song **c**

Milo Goes to College



1 - Song **d**

2 - Song **e**

3 - Song **f**

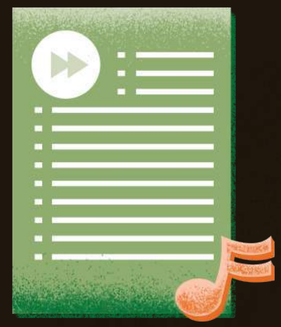
Playlist *(by value)*

Music for Programming

1 -

2 -

3 -



Favorite songs handpicked by us

Creating a Playlist With Values

We can implement a music playlist by **creating copies** of existing songs and storing those copies under each playlist.

Albums

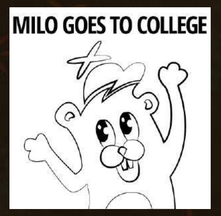
Are You Gonna Go My Way

1 - Song **a**

2 - Song **b**

3 - Song **c**

Milo Goes to College



1 - Song **d**

2 - Song **e**

3 - Song **f**

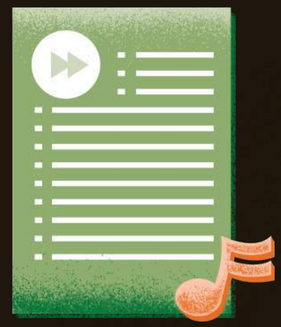
Playlist *(by value)*

Music for Programming

1 - Song **a**

2 - Song **c**

3 - Song **f**



Playlist contains copies of the original songs.

Creating a Playlist With References

A more efficient way to implement a playlist is by **storing references** to original songs. This avoids creating multiple copies of the same songs for each new playlist.

Albums

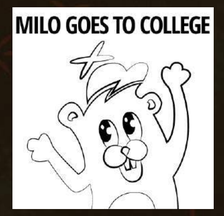
Are You Gonna Go My Way

1 - Song **a**

2 - Song **b**

3 - Song **c**

Milo Goes to College



1 - Song **d**

2 - Song **e**

3 - Song **f**

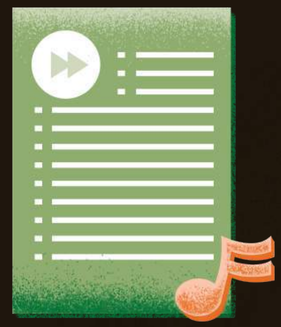
Playlist *(by reference)*

Music for Programming

1 -

2 -

3 -



Playlist slots point back to original songs.
No copies are created! (memory efficient)

Pass by Value

A) Passing Values *(default behavior)*

```
language := "Go"
```

1. Primitive value is assigned to new variable and stored in **new memory address** 0x10444444.

```
favoriteLanguage := language
```

2. A **new memory address** 0x10555555 **is allocated** for the new variable that receives **a copy of the original data.**

Two different memory addresses are used to store exact copies of the data.

Pass by Reference

B) Passing References

1. Primitive value is assigned to new variable and stored in new memory address **0x10444444**.
2. Using the & operator, a **reference** to the existing memory address is assigned to the new variable.

```
language := "Go"
favoriteLanguage := &language
```

Returns a pointer

A single memory address is used. (memory efficient)

Passing structs **by Reference**

In order to assign a struct **reference to a new variable, we use the** & **operator to return a pointer.**

```
type gopher struct {
  name string
  age  int
  isAdult bool
}

func main() {
  phil := &gopher{name: "Phil", age: 30}
  validateAge(phil)
  fmt.Println(phil)
}

func validateAge(          ) {
  g.isAdult = g.age >= 21
}
```

The & operator returns a pointer to this new struct.

Passes a reference to the original struct — NOT a copy of the values

Values and References Are Not the Same

Accepting references as function arguments requires a different syntax.

```
type gopher struct {
  name string
  age  int
  isAdult bool
}

func main() {
  phil := &gopher{name: "Phil", age: 30}
  validateAge(phil)
  fmt.Println(phil)
}

func validateAge(g gopher) {
  g.isAdult = g.age >= 21
}
```

```
$ go run main.go

main.go:15: cannot use phil (type *gopher) as
        type gopher in argument to validateAge
```

Wrong type!

Reading struct **References**

We use the * operator to access the value that the pointer points to (a.k.a., “dereferencing a pointer variable”).

```
type gopher struct {
  name string
  age  int
  isAdult bool
}

func main() {
  phil := &gopher{name: "Phil", age: 30}
  validateAge(phil)
  fmt.Println(phil)
}

func validateAge(g *gopher) {
  g.isAdult = g.age >= 21
}
```

The * operator indicates a pointer to the type *gopher*.

```
$ go run main.go
&{Phil 30 true}
```

Reference to underlying *struct*

Original value from *struct* is changed

Level 5–1

Gophers & Friends

Interfaces

ON TRACK with GOLANG

Calling jump() on Multiple Gophers

Static typing allows the Go compiler to ensure every element on the collection returned by getList() responds to the jump() method.

```
type gopher struct { ... }
func (g gopher) jump() string { ... }

func main() {
  gopherList := getList()
  for _, gopher := range gopherList {
    fmt.Println(gopher.jump())
  }
}

func getList() []*gopher {

}
```

All gophers respond to *jump()*...

...and here we grab a collection of gophers...

...so we can safely call *jump()* on each and every element from this collection.

The * symbol means return value is a slice of pointers to gopher.

Returning a Collection of struct Pointers

From the getList() function, we create two gophers, grab their pointers, and return them as part of a slice.

```
type gopher struct { ... }
func (g gopher) jump() string { ... }

func main() {
   ...
}

func getList() []*gopher {
   phil := &gopher{name: "Phil", age: 30}
   noodles := &gopher{name: "Noodles", age: 90}

   list := []*gopher{phil, noodles}
   return list
}
```

```
$ go run main.go
```

```
Phil can jump HIGH
Noodles can still jump
```

Creates two gophers and returns a pointer for each.

Creates a new slice with the gopher pointers and returns it from the function.

Other Types Can Also jump()

There can also be other structs, like horse, with different properties but **the same method signature.**

```
type gopher struct { ... }
func (g gopher) jump() string { ... }

type horse struct {
  name   string
  weight int
}

func (h horse) jump() string {
  if h.weight > 2500 {
    return "I'm too heavy, can't jump..."
  }
  return "I will jump, neigh!!"
}
```

Different properties...

...but exact same method signature.

Different Types Are... Different!

We cannot combine both Gopher and Horse structs **under a single slice of type** *gopher.

```
type gopher struct { ... }
func (g gopher) jump() string { ... }

type horse struct { ... }
func (h horse) jump() string { ... }

33 func getList() []*gopher {
34   phil := &gopher{name: "Phil", age: 30}
35   noodles := &gopher{name: "Noodles", age: 90}
36   barbaro := &horse{name: "Barbaro", weight: 2000}
37
38   list := []*gopher{gopher, noodles, barbaro}
39   return list
40 }
```

A horse is NOT a gopher!

```
$ go run main.go
```

```
main.go:38: cannot use horse (type *horse) as type
            *gopher in array or slice literal
```

Common Behavior With interface

Interfaces provide a way to **specify behavior:** *"If something can do* **this**, *then it can be used* **here**".

(type)

(method)

```
...

type jumper interface {
  jump() string
}

func getList() []jumper {
  phil := &gopher{name: "Phil", age: 30}
  noodles := &gopher{name: "Noodles", age: 90}
  barbaro := &horse{name: "Barbaro", weight: 2000}

  list := []jumper{ }
  return list
}
```

Method expected to be present in all types that implement this interface

Can be used as return type
*(The * symbol is not necessary when working with interfaces)*

All types part of this slice MUST respond to jump().

Combining Types That jump()

Types implement interfaces **implicitly**, simply by **implementing methods from the interface.**

```
func (g gopher) jump() string { ... }
func (h horse) jump() string { ... }

type jumper interface {
  jump() string
}

func getList() []jumper {
  phil := &gopher{name: "Phil", age: 30}
  noodles := &gopher{name: "Noodles", age: 90}
  barbaro := &horse{name: "Barbaro", weight: 2000}

  list := []jumper{gopher, noodles, barbaro}
  return list
}
```

Because gopher and horse both implement jump() with the exact same signature...

...we can add both types under the same slice of type jumper.

All Jumpers Can jump()

```go
type gopher struct { ... }
func (g gopher) jump() string { ... }
type horse struct { ... }
func (h horse) jump() string { ... }
type jumper interface {
  jump() string
}

func main() {
  jumperList := getList()
  for _, jumper := range jumperList {
    fmt.Println(jumper.jump())
  }
}

func getList() []jumper { ... }
```

Compiler does NOT care about naming, so these could be named something else too...

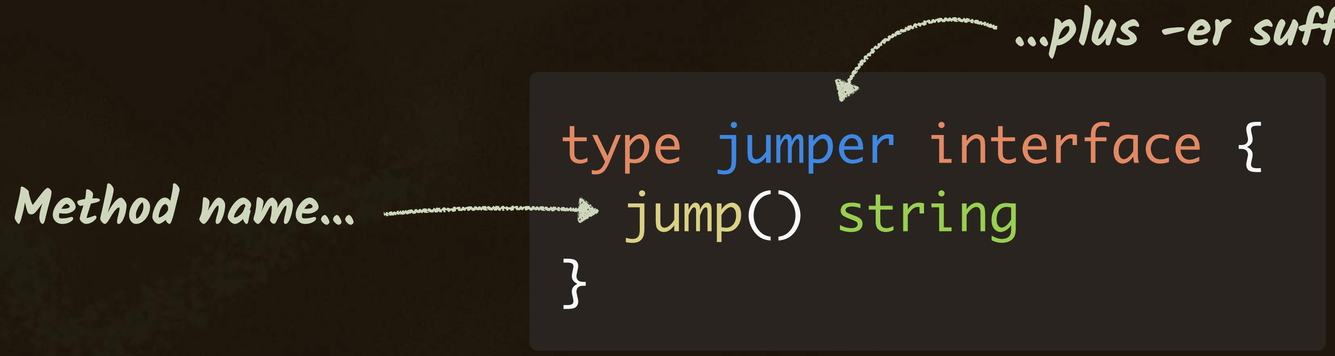
```go
list := getList()
for _, element := range list {
  fmt.Println(element.jump())
}
```

```
$ go run main.go
```

```
Phil can jump HIGH
Noodles can still jump
I will jump, Neigh!!
```

Naming Convention for interfaces

A convention for naming one-method interfaces in Go is to use the method name plus an **-er** suffix.



Examples from the Go standard library:

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

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

Visit **http://go.codeschool.com/go-io** for full documentation

Level 5–2

Gophers & Friends

Creating Packages

ON TRACK with GOLANG

When Single Files Grow Too Long

As we add **more code to the main file**, keeping logic for our program inside a single file gets complicated.

src/hello/main.go

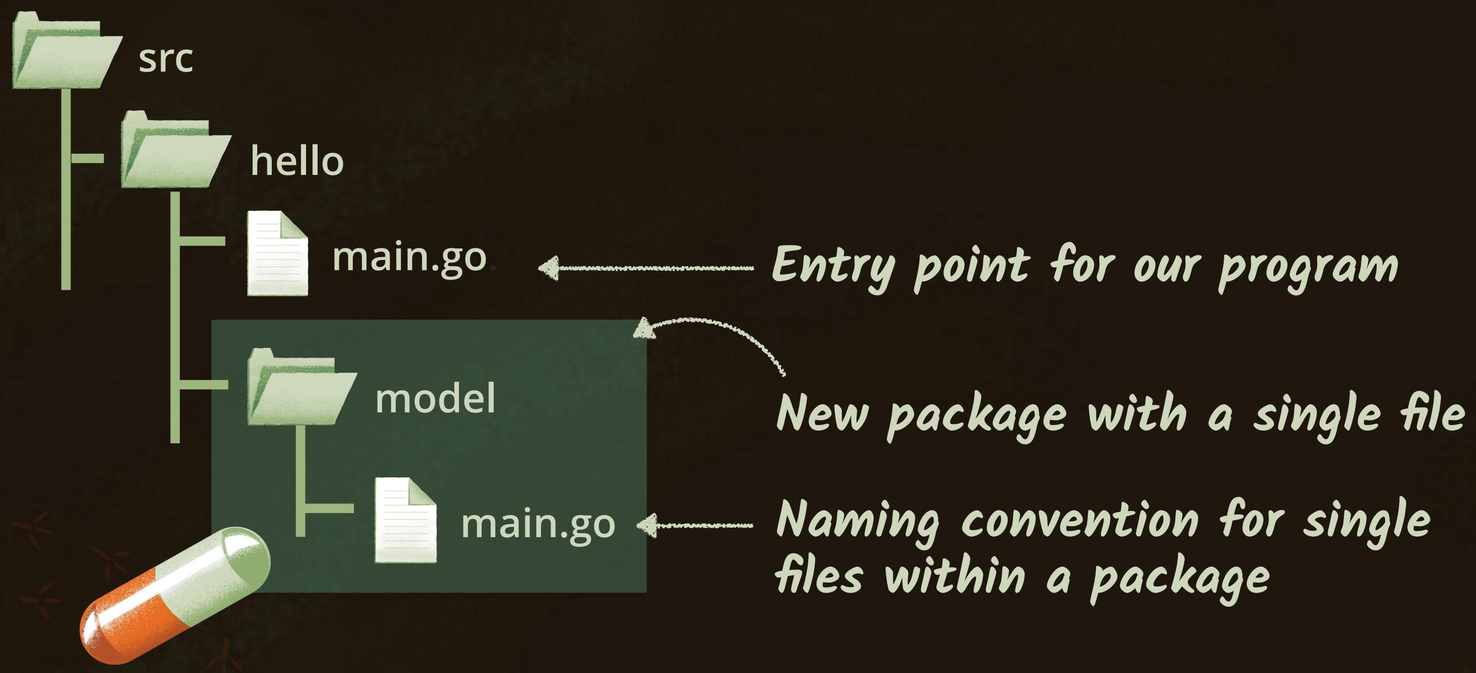
```
package main
import ...

type gopher struct { ... }
func (g gopher) jump() string { ... }
type horse struct { ... }
func (h horse) jump() string { ... }
type jumper interface { ... }

func getList() []jumper { ... }

func main() {
   ...
}
```

Too much code to look at!

Code inside this function is what really matters.

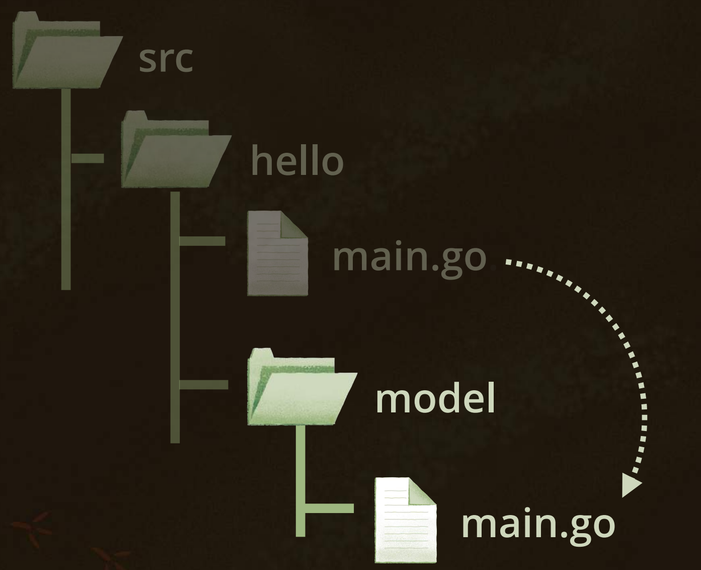
Creating a New Package Inside a Project

One way to refactor files growing too long is to create **project packages.** A new **package** is a **folder** within the project that holds logic for a specific part of the program.



Moving Code to New Package

In order to be accessed from outside packages, identifiers must be **explicitly exported** by adopting an **uppercase naming convention for the first letter.**



Move struct definitions and methods to new package.

src/hello/model/main.go

```
package model
```

New package definition

```
type gopher struct { ... }
func (g gopher) jump() string { ... }
type horse struct { ... }
func (h horse) jump() string { ... }
type jumper interface { ... }
```

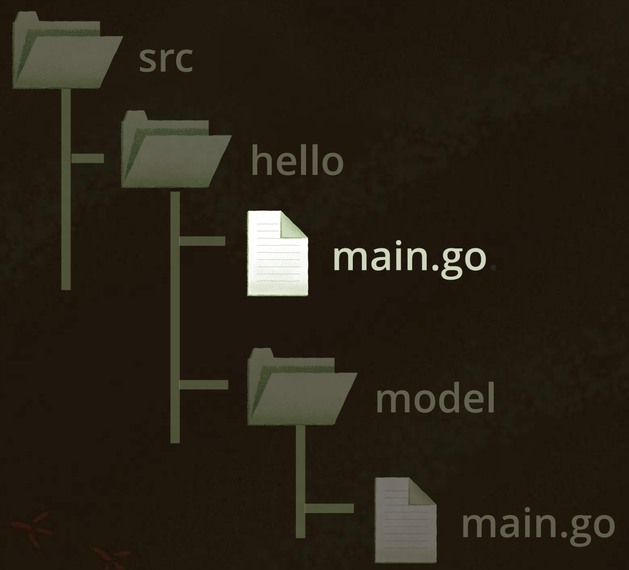
*Moved here from hello/main.go **with no changes***

```
func GetList() []jumper { ... }
```

Capitalized name means this function can now be accessed from outside packages.

Importing Package and Calling Functions

From the main source code file, we can import our new package by using the import statement followed by the project name (**hello**) and the new package name (**model**).



src/hello/main.go

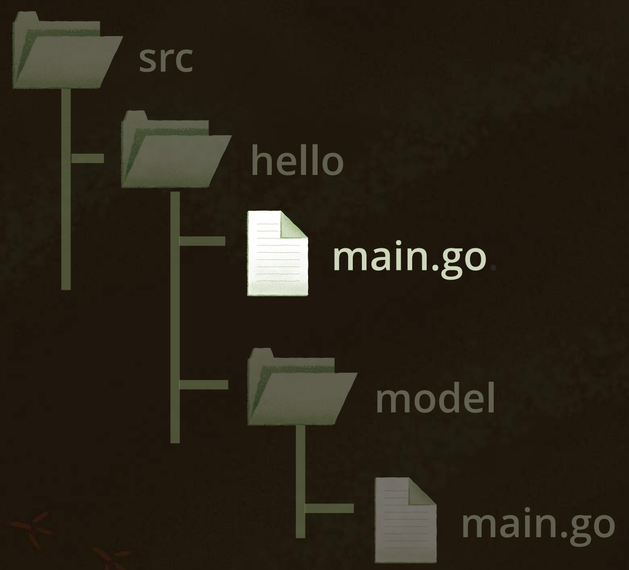
```
package main

import (
  "fmt"
  "hello/model"
)

func main() {
  jumperList := model.GetList()
  for _, jumper := range jumperList {

  }
}
```

Import new package.

Function namespaced by package name

Understanding Export Errors

References to unexported identifiers will cause the Go compiler to **raise errors.**



src/hello/main.go

```
1  package main
2
3  import (
4    "fmt"
5    "hello/model"
6  )
7
8  func main() {
9    jumperList := model.GetList()
10   for _, jumper := range jumperList {
11     fmt.Println(jumper.jump())
12   }
13 }
```

```
$ go run main.go
./main.go:11: jumper.jump undefined
  (cannot refer to unexported field
   or method jump)
```

Whoops! Looks like we forgot to export this method.

Exporting Methods

Interface methods and their corresponding implementations must also be capitalized in order to be invoked from other packages.

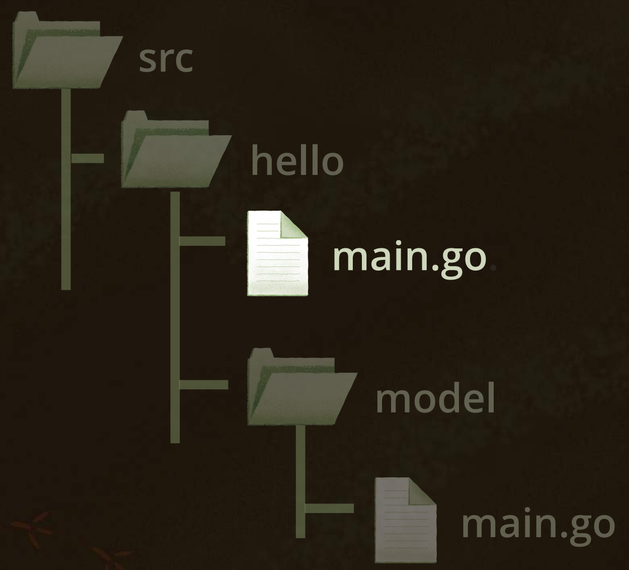
src/hello/model/main.go

```
package model

type gopher struct { ... }
func (g gopher) Jump() string { ... }
type horse struct { ... }
func (h horse) Jump() string { ... }

type jumper interface {
  Jump()
}

func GetList() []jumper {
  ...
}
```

Only the method names need to be exported — NOT the structs or interface.

Running With Correct Package Imports

Our program can now run without errors, and the main file looks a lot cleaner!



src/hello/main.go

```go
package main

import (
  "fmt"
  "hello/model"
)

func main() {
  jumperList := model.GetList()
  for _, jumper := range jumperList {
    fmt.Println(jumper.Jump())
  }
}
```

```
$ go run main.go
```

```
Phil can jump HIGH
Noodles can jump ok.
I will jump, Neigh!!
```

Level 5–3

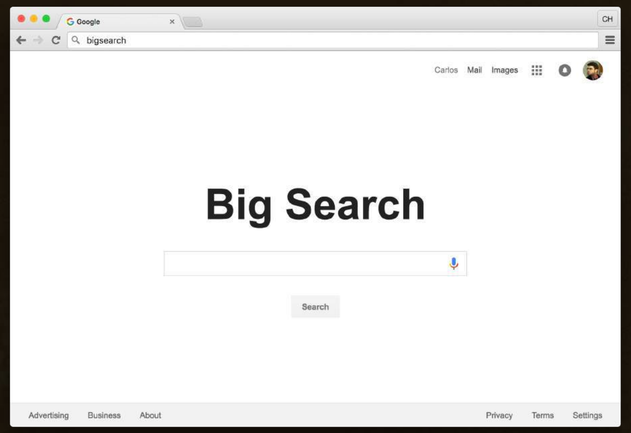
Gophers & Friends

goroutines

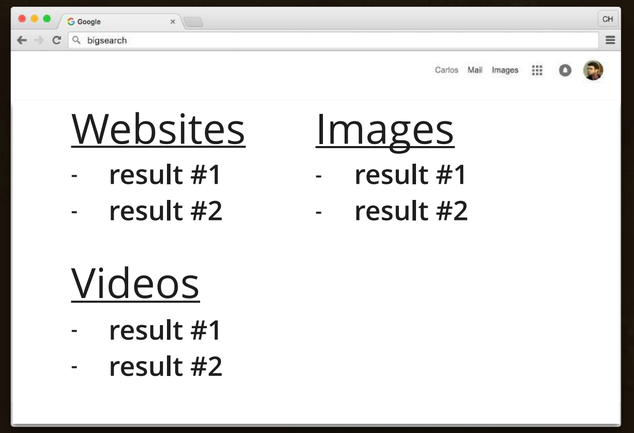
ON TRACK with GOLANG

The “Big Search” Website

This search engine will search the web for websites, images, and videos.

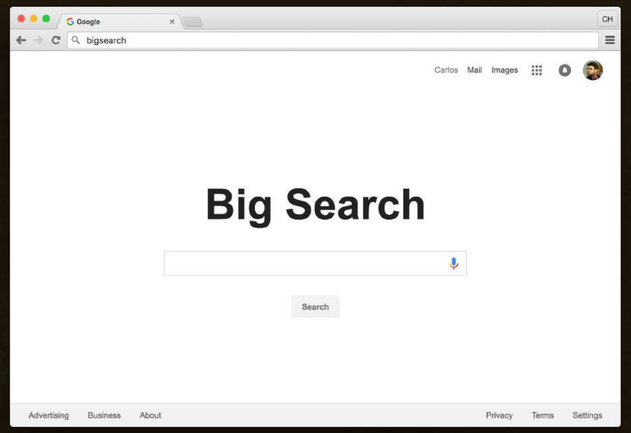


Display results for a search

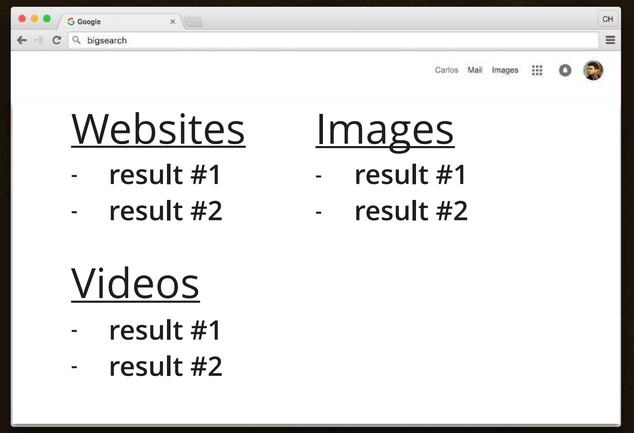


Sequential Programs

In sequential programs, before a new task starts, the previous one **must finish.**



Searches websites, images, and videos one at a time



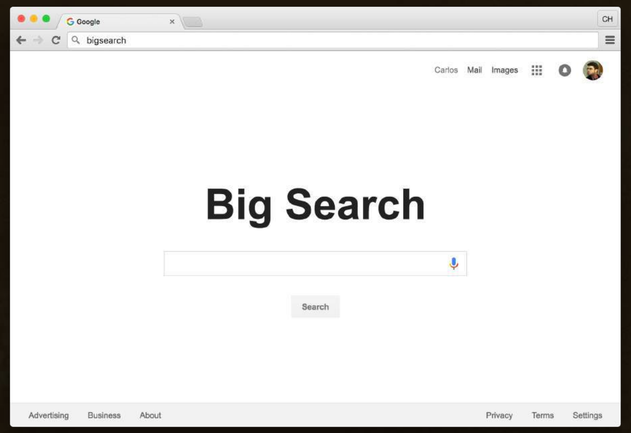
(Total: 9 seconds)

Websites *Images* *Videos*

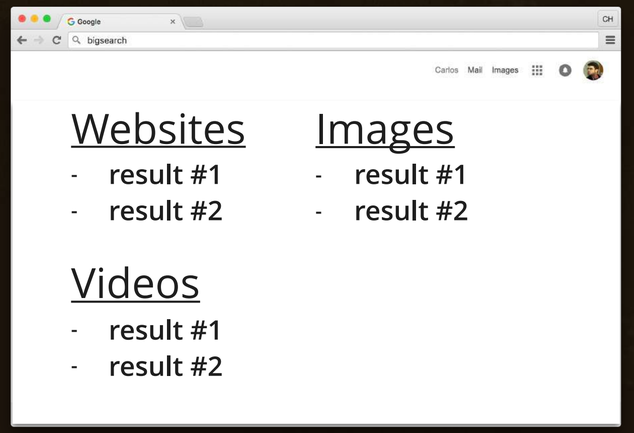
Takes as long as the sum of all of tasks

Concurrent Programs

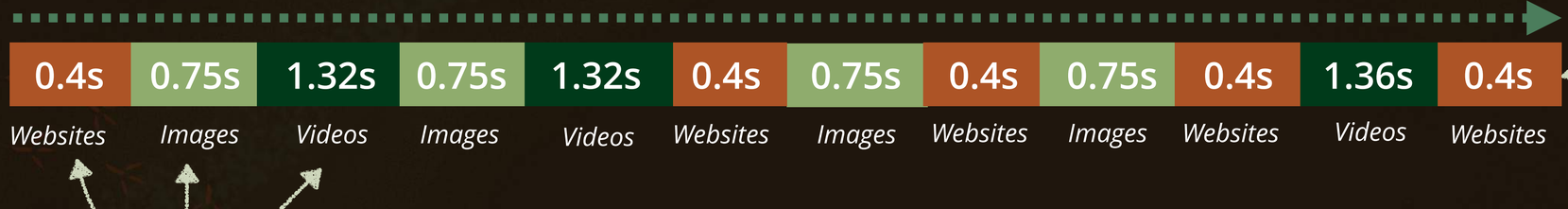
In concurrent programs, multiple tasks can be **executed independently** and may **appear simultaneous.**



Searches websites, images, and videos "at the same time"



(Total: 9 seconds)

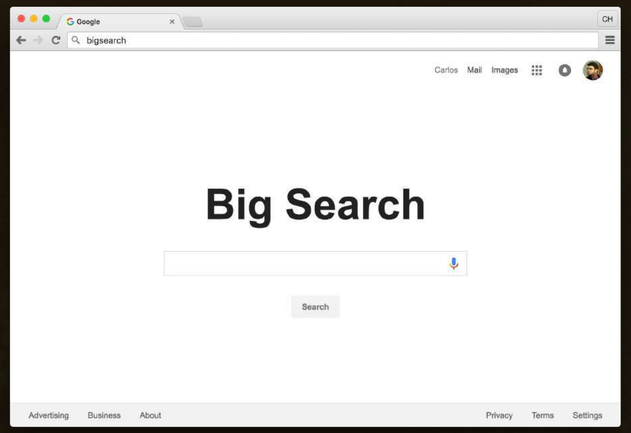


Can be executed in any order

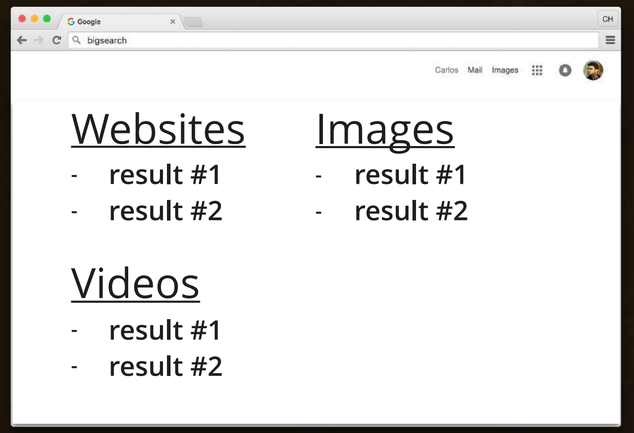
Also takes as long as the sum of all tasks, but tasks alternate time slices

Parallel Programs

In parallel programs, multiple tasks can be executed **simultaneously** (requires **multi-core** machines).



Searches websites, images, and videos **at the same time** *(really!)*



(Total: 4 seconds)

2 seconds

Websites

3 seconds

Images

4 seconds

Videos

All executed at the same time

Takes as long as the slowest task

Concurrency Allows Parallelism

Concurrency and parallelism are **NOT** the same thing. The former means **independent,** which is a necessary step toward the latter, which means **simultaneous.**

In short...

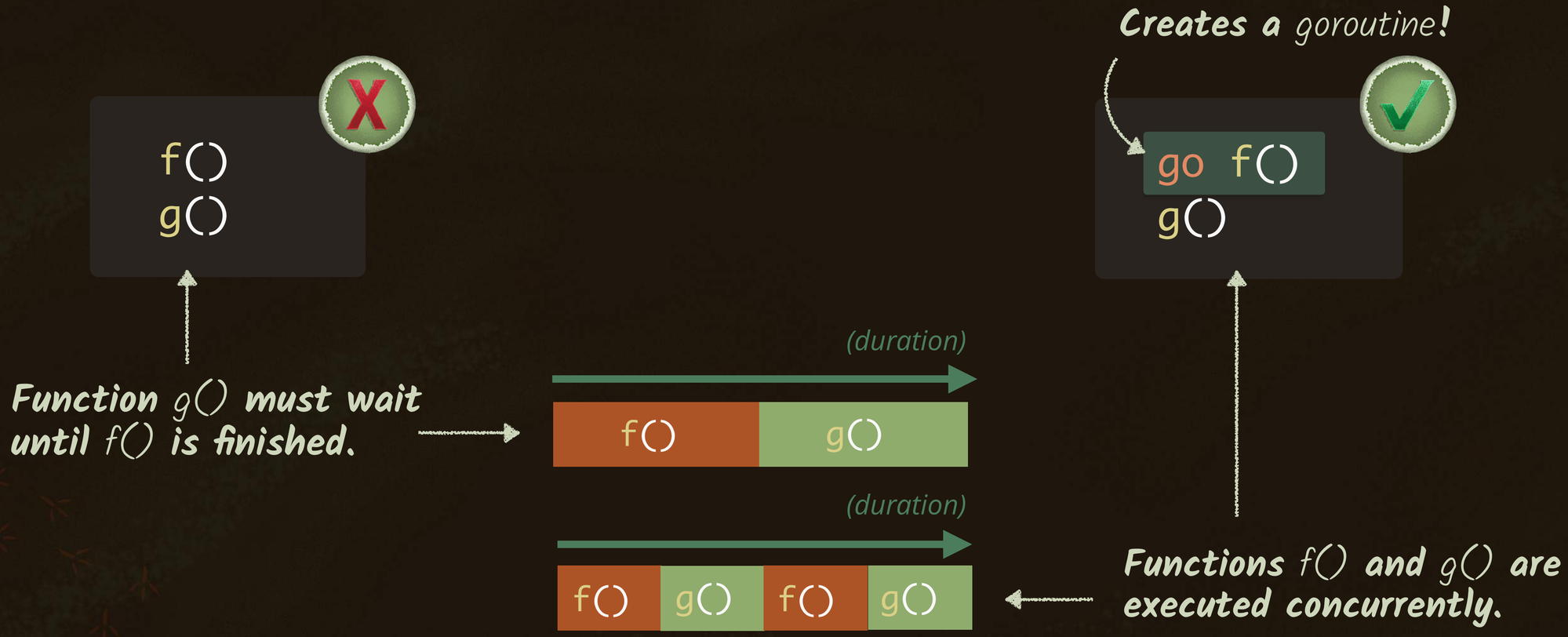
Concurrent *means* **Independent**

Parallel *means* **Simultaneous**

Go's **concurrency** model and goroutines make it simple to build **parallel programs** that take advantage of machines with **multiple processors** (most machines today).

Concurrency With goroutines

A goroutine is a special function that executes **concurrently** with other functions. We create them with the go keyword (yes, go is also a keyword!).



On a single-core machine, concurrent code is unlikely to perform better than sequential code.

Looping and Printing Names

Let's write a new program that iterates through a slice and invokes a function printName() for each item.

```
package main

import "fmt"

func main() {
  names := []string{"Phil", "Noodles", "Barbaro"}
  for _, name := range names {
    printName(name)
  }
}

func printName(n string) {

}
```

Tracking Duration With the time Tool

The printName() function takes one argument and simply prints it to the console. We'll run our program with go run and use the Unix time command to track the duration of the execution.

```
...

func main() {
  names := []string{"Phil", "Noodles", "Barbaro"}
  for _, name := range names {
    printName(name)
  }
}

func printName(n string) {
  fmt.Println("Name: ", n)
}
```

Prints argument to the console

Determines duration of command passed to it.

```
$ time go run main.go
```

```
Name: Phil
Name: Noodles
Name: Barbaro

real 0m0.321s
```

Less than half a second

Heavy Processing Comes Into Play

Let's simulate a **time-consuming task** on printName(). We'll do this by adding a very costly mathematical operation using the math package from Go's standard library.

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

func main() { ... }

func printName(n string) {
  result := 0.0
  for i := 0; i < 100000000; i++ {
    result += math.Pi * math.Sin(float64(len(n)))
  }
  fmt.Println("Name: ", n)
}
```

Import new package.

Time-consuming computation keeps the processor busy!

Sequential Tasks Are Blocking

When we add **heavy processing** to printName(), we can see a big **time increase on execution time.**

```
...
func main() {
  names := []string{"Phil", "Noodles", "Barbaro"}
  for _, name := range names {
    printName(name)
  }
}

func printName(n string) {
  result := 0.0
  for i := 0; i < 100000000; i++ {
    result += math.Pi * math.Sin(float64(len(n)))
  }
  fmt.Println("Name: ", n)
}
```

Each call to this function blocks the processor for almost 5 seconds!

```
$ time go run main.go
```

```
Name: Phil
Name: Noodles
Name: Barbaro

real 0m11.603s
```

Went from 0.3 to 11.6 seconds!

Running sequentially

Going Concurrent

Go programs **are NOT automatically aware** of newly created goroutines, **so the main function exits before the** goroutines **are finished.**

```
...
func main() {
  names := []string{"Phil", "Noodles", "Barbaro"}
  for _, name := range names {
    go printName(name)
  }
}

func printName(n string) {
  ...
}
```

Executes each function call on a new goroutine.

```
$ time go run main.go
```

```
real 0m0.314s
```

Back to being fast, but no names listed

Worry not, my friend. There's a built-in solution for this...

Adding Synchronization With WaitGroup

On the sync package from Go's standard library, there's a WaitGroup data type. We can use this type to make our program wait for goroutines to finish.

```
...
import (
  "fmt"
  "sync"
  "math"
)

func main() {
  names := []string{"Phil", "Noodles", "Barbaro"}
  var wg sync.WaitGroup
  ...
}
...
```

Import new package.

Declare a new variable of the sync.WaitGroup data type.

Waiting on goroutines

The Add **method sets the number of** goroutines **to wait for, and the** Wait **method prevents the program from exiting before all** goroutines **being tracked by our** WaitGroup **are finished executing.**

```
...

func main() {
  names := []string{"Phil", "Noodles", "Barbaro"}
  var wg sync.WaitGroup
  wg.Add(len(names))
  for _, name := range names {
    go printName(name)
  }
  wg.Wait()
}
...
```

The call to len() *returns the total number of names...*

...which is equal to the number of goroutines *we create inside the loop.*

Prevents program from exiting.

Updating WaitGroup

The Done **method must be called from each function that runs on a** goroutine **once it's finished. This gives the** WaitGroup **an update — like saying,** ***"Hey, there's one less*** goroutine ***you need to wait for."***

```
...
func main() {
  var wg sync.WaitGroup
  wg.Add(len(names))
  ...
  for _, name := range names {
    go printName(name, &wg)
  }
  wg.Wait()
}

func printName(n string, wg *sync.WaitGroup) {
  ...
  wg.Done()
}
```

Must pass a reference to *WaitGroup* ***so that we call*** *Done* ***on the original value and NOT on a copy.***

Inform the *WaitGroup* ***that the*** *goroutine* ***running this function is now finished!***

Single CPU — Concurrent and Synchronized

If we run our final code **specifying a single processor**, there's **no noticeable performance improvement.**

```
...
func main() {
  ...
  var wg sync.WaitGroup
  wg.Add(len(names))
  for _, name := range names {
    go printName(name, &wg)
  }
  wg.Wait()
}

func printName(n string) {
  ...
  wg.Done()
}
```

```
$ time GOMAXPROCS=1 go run main.go
```

Run program on a single processor.

```
Name: Phil
Name: Noodles
Name: Barbaro

real 0m11.675s
```

Still slow

Running concurrent

Multiple CPUs — Parallel and Synchronized

The Go runtime **defaults to using all processors available**. Most machines today have more than one processor and our concurrent Go code can run **in parallel** with no changes!

```
...
func main() {
  ...
  var wg sync.WaitGroup
  wg.Add(len(names))
  for _, name := range names {
    go printName(name, &wg)
  }
  wg.Wait()
}

func printName(n string) {
  ...
  wg.Done()
}
```

Absence of GOMAXPROCS means all processors available will be used!

```
$ time go run main.go
```

```
Name: Phil
Name: Noodles
Name: Barbaro

real 0m4.172s
```

From 11 to 4.1 seconds!

Running in parallel!

```
printName("Phil", &wg)
printName("Noodles", &wg)
printName("Barbaro", &wg)
```