

## THE GO LANGUAGE BASICS

---

# OR HOW TO BE MORE PRODUCTIVE AND HAVE FUN

# WHAT THIS CLASS IS (AND ISN'T)

- ▶ This class is not an intro to programming class. We assume you understand certain concepts like variables, conditionals and loop statements.
- ▶ This class does not teach the Go programming environment (build tools, etc...). All programs here are designed to be run in the Go playground (<http://play.golang.org>). The Go programming environment is detailed by many others sources on the web.
- ▶ This class is for those who have experience with a scripting language (Python, Perl, Ruby, ...)
- ▶ This class is not a comprehensive guide and does not always show the best way of completing a task. It is for bootstrapping students with needed material and the examples are used to teach concepts, not optimal methods.
- ▶ You like Gophers, because every Go presentation, conference, etc.. will be covered in them (including this one).

## CLASS MATERIALS

- ▶ You will need a copy of the slides used in this class. These can be found at:
  - ▶ [http://www.github.com/johnsiilver/go\\_basics/  
go\\_basics.pdf](http://www.github.com/johnsiilver/go_basics/go_basics.pdf)

WHY TO USE/NOT USE GO

---

OR WHY TO BOTHER WITH THIS CLASS



# WHY USE GO

- ▶ Go is type-safe, preventing many errors common to non-typed languages. This alone will increase your productivity 10x and make refactoring much simpler.
- ▶ Go is fast with a simple concurrency model.
- ▶ Go has low memory consumption compared to interpreted languages (Java, Python, Perl).
- ▶ Go is compiled. No need to distribute different versions of the interpreter or virtual machine.
- ▶ Go is quick to learn with few concepts not found in interpreted languages.
- ▶ Go is garbage collected, allowing an engineer to work on their code, not on which pointer type to use and when to allocate on the stack/heap.
- ▶ Go is fun. After 5k lines of code, you won't want to Go back (pun intended).

# WHY USE GO

268,843 Lines of Go in 2,632 files

- ▶ Go builds FAST!

No warm cache

Pachyderm server(btw)

real 0m 33.70s

user 1m 33.04s

sys 1m 13.16s

With a warm cache:

real 0m 6.67s

user 0m 16.07s

sys 0m 4.18s

# WHY NOT TO USE GO

- ▶ Go is faster for large programs than interpreted languages or languages using virtual machines. It is not faster than languages such as C++. If every nano-second equals money, C or C++ is probably your best bet. But only if you are a good C/C++ developer.\*
- ▶ Go is garbage collected. While this protects you from many mistakes that are common in C or C++, you no longer have control of when memory is freed and the GC adds delays when doing collection.
- ▶ While Go has a large and growing amount of software available for it, there are more libraries available for other languages.
- ▶ Go has virtually no UI tooling (outside of web frameworks) or an impressive IDE (like Visual Studio/ Xcode). If you are working on OS native user facing apps or with OpenGL/Direct X, Go is probably not the best choice (yet).
- ▶ The Go debugger situation is not stellar. While you can use the pprof debugger, it does not support the runtime. Third-party debuggers are not practical in many environments.

\*This is likely to draw criticism because interpreted language X is faster on a micro benchmark. In practice, for a large program written by a good developer, Go is faster than Java, CPython, and Javascript based programs.

Note: I'm being fairly broad with the term "interpreted" to include virtual machine languages that are semi-compiled.



# GO PACKAGES

---

## HOW GO ORGANIZES ITS CODE

# SOURCE FILES AND PACKAGES

- ▶ Go source files are named <file name>.go
- ▶ All Go files in the same directory will belong to the same package.
- ▶ A package can be made up of multiple files in the same directory.

# SOURCE FILES AND PACKAGES

- ▶ Package names are created using the **package** keyword followed by the package name at the top of the file.
- ▶ The package name must be the same as the directory.



stuff



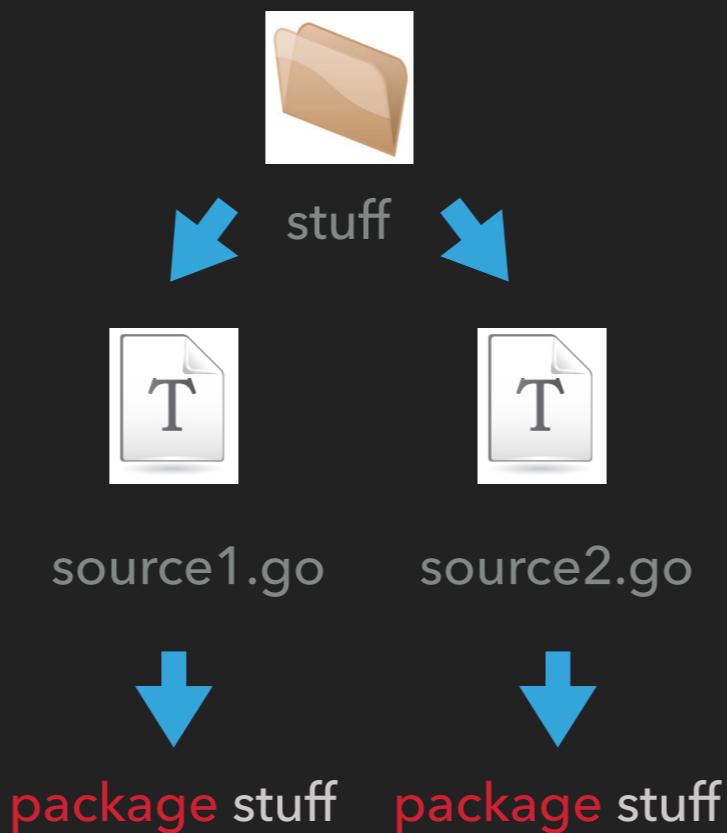
source.go



**package** stuff

# SOURCE FILES AND PACKAGES

- ▶ Two files with different names in the same directory should have the same package name and belong to the same package.



# SOURCE FILES AND PACKAGES

- ▶ Package main is special. Your directory does not have to be named “main” and it indicates where the compiler will compile a binary and not a library.
- ▶ Package main always contains a function called `main()`, which is the entrance point for execution. Similar to Python’s: `if __name__ == '__main__':`

```
package main

func main() {
    // Do something here.
}
```

## SOURCE FILES AND PACKAGES

- ▶ Importing a package into another file is done with the **import** keyword.

```
package main

import "fmt"

func main() {
    fmt.Println("Hello World")
}
```

Package **fmt** is part of the standard library

# SOURCE FILES AND PACKAGES

- ▶ You can simplify multiple imports by using () with the **import** keyword.

```
package main

import (
    "fmt"
    "os"
)

func main() {
    fmt.Println("Hello World")
    os.Exit(1)
}
```

(This prevents having to type **import** again and again)

# SOURCE FILES AND PACKAGES

- ▶ If you import a local package, the import statement is the path from your "src" directory (but not including it) to the directory containing the package.

```
package main

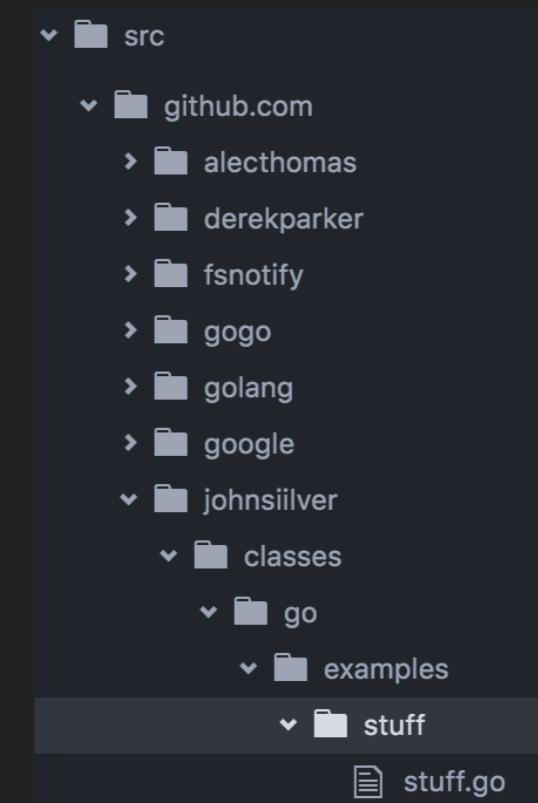
import (
    "fmt"

    "github.com/johnsiilver/classes/go/examples/stuff"
)

func main() {
    fmt.Printf("%d\n", stuff.Add(2,2))
}
```



(The arrow points to the package name)



Directory structure to this package

## SOURCE FILES AND PACKAGES

- ▶ The following is an example of a custom package providing a function to add two numbers together

```
package stuff

// Add returns the sum of x and y.
func Add(x, y int) int {
    return x + y
}
```

# SOURCE FILES AND PACKAGES

- ▶ This is an example of a binary that imports the stuff package and prints out 2+2

```
package main

import (
    "fmt"

    "github.com/johnsiilver/classes/go/examples/stuff"
)

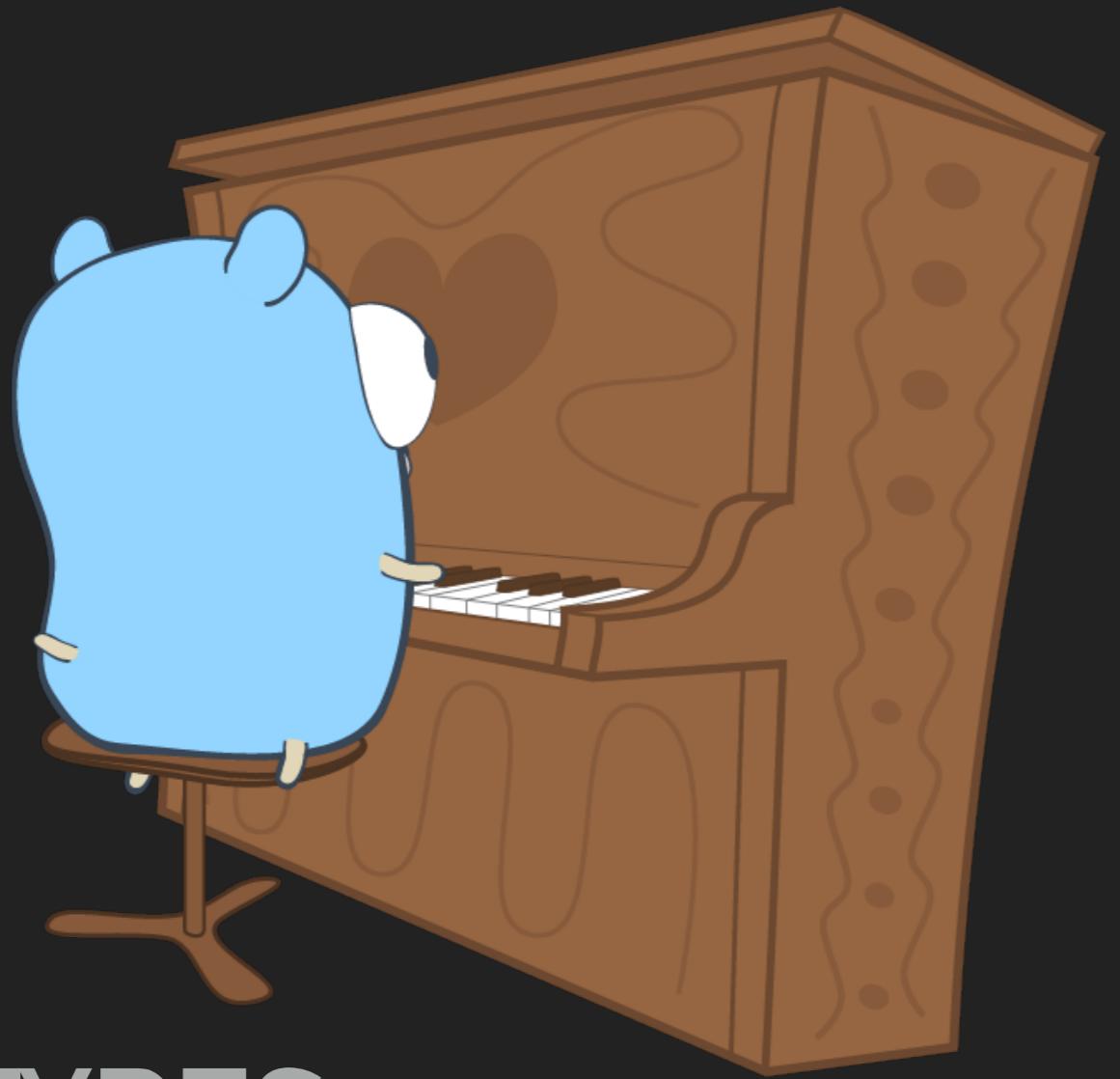
func main() {
    fmt.Printf("%d\n", stuff.Add(2,2))
}
```

# SOURCE FILES AND PACKAGES

- ▶ Standard library packages can be found at:
  - ▶ <http://golang.org/pkg>
- ▶ A great source for third-party packages is:
  - ▶ <https://godoc.org/>

# SUMMARY: SOURCE FILES AND PACKAGES

- ▶ Go source files are made up of files ending with `.go`
- ▶ Go code is divided into packages.
- ▶ All `.go` files in a directory should belong to the same package.
- ▶ The package name must be the same as the directory name.
- ▶ Packages are declared at the top of the file with:
  - ▶ `package <package name>`
  - ▶ `package main` is special, it is the entrance point for an executable.



GO TYPES

---

**EVERYONE HAS A TYPE**

# STATIC TYPING VS DYNAMIC TYPING

- ▶ Go uses static typing, meaning that once a variable has a type (`string`, `int`, `float`, ...), that variable can only store values of that type.
- ▶ Many languages have dynamic typing, meaning a variable can hold any type of value. One moment it is a `string`, the next it is an `int`.
- ▶ Static typing prevents errors in the compiler. You cannot store a `string` in a variable that has the `int` type. Dynamic typing is a large source of runtime errors.

# STATIC TYPING VS DYNAMIC TYPING

- ▶ The following is an example of dynamic typing in Python
- ▶ `x` starts as an integer
- ▶ `x` ends as a string

```
def someFunc():
    x = 3
    print(x)
    x = "hello"
    print(x)
```

# STATIC TYPING VS DYNAMIC TYPING

- ▶ At first this seems like an advantage, but various studies have shown this to increase runtime errors.
- ▶ It is easy to send the wrong types to function than was actually meant, especially if the argument list gets long. Many of these errors will not be exposed until runtime.

```
def Add(x, y):  
    return x + y
```

- ▶ You don't know what `x` and `y` are going to be.
- ▶ What will be returned is unknown.
- ▶ This will work if `x` and `y` are integers or strings.
- ▶ It will traceback during runtime if called and `x` is an `int` and `y` is a `string`.

# SO WHAT IS A TYPE

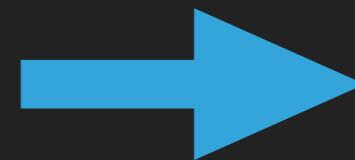
- ▶ A type defines what attributes data must possess to be placed in a variable.
- ▶ An `int32` type states the data stored in it must have the following attributes:
  - ▶ Is a whole number
  - ▶ Must  $\geq -2147483648$
  - ▶ Must be  $\leq 2147483647$
- ▶ A `string` type states the data stored in it must have the following attributes:
  - ▶ Must contain UTF-8 characters (a character encoding standard)

## SECTION 2

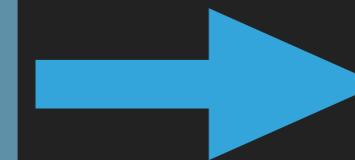
---

# SO WHAT IS A TYPE

People trying to get in an upscale bar



Bouncer



Upscale Bar



- Are you 21?
- Do you have the right attire?

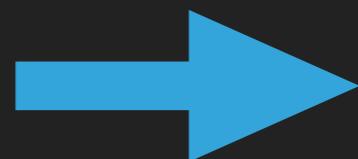
## SECTION 2

---

# SO WHAT IS A TYPE

Value to be stored

3

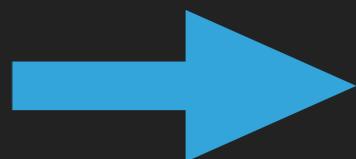


Compiler



Variable X of type int32

3



- Is it a whole number?
- $\geq -2147483648$
- $\leq 2147483647$

# SO WHAT IS A TYPE

- ▶ A variable of a type is incompatible with variables of other types:
  - ▶ For example, you cannot add an `int` and `int64` together, even though on most platforms they have the same attributes.
- ▶ Once a variable is declared with its type, no data of any other type can be stored in it.
- ▶ Some types can be converted from one type to another type.

# COMMON TYPES IN GO

- ▶ **int** - An integer that can be 32 or 64 bit, depending on the platform
- ▶ **string** - UTF8 character string
- ▶ **slices** - A growable Array holding a contiguous list of the same types of elements. Similar to Python lists or Ruby Arrays.
- ▶ **maps** - Like Python dictionaries or Perl hashes, stores key/value pairs. Keys must all be the same type and values must be the same type. Key type and value type can be different.
- ▶ **Structs** - Go's class like type. Similar to Python classes or C structs. Holds a collection of named attributes.

## SECTION 2

---

# NUMERIC TYPES IN GO (FOR REFERENCE)

- ▶ `uint8` the set of all unsigned 8-bit integers (0 to 255)
- ▶ `uint16` the set of all unsigned 16-bit integers (0 to 65535)
- ▶ `uint32` the set of all unsigned 32-bit integers (0 to 4294967295)
- ▶ `uint64` the set of all unsigned 64-bit integers (0 to 18446744073709551615)
  
- ▶ `int8` the set of all signed 8-bit integers (-128 to 127)
- ▶ `int16` the set of all signed 16-bit integers (-32768 to 32767)
- ▶ `int32` the set of all signed 32-bit integers (-2147483648 to 2147483647)
- ▶ `int64` the set of all signed 64-bit integers (-9223372036854775808 to 9223372036854775807)
  
- ▶ `float32` the set of all IEEE-754 32-bit floating point numbers
- ▶ `float64` the set of all IEEE-754 64-bit floating point numbers
  
- ▶ `complex64` the set of all complex numbers with float32 real and imaginary parts
- ▶ `complex128` the set of all complex numbers with float64 real and imaginary parts
  
- ▶ `uint` either 32 or 64 bit number (platform dependent)
- ▶ `int` either 32 or 64 bit number (platform dependent)

# REFERENCE TYPES\*

- ▶ **slices** - A growable array.
- ▶ **maps** - Key/Value pairing.
- ▶ **channels** - A synchronization primitive for communicating between goroutines (Go's answer to threads).

\***Note:** What makes a reference type different than other types will be discussed later

# OTHER TYPES

- ▶ **byte** - An alias for uint32
- ▶ **rune** - An alias for int32, represents a Unicode code point
- ▶ **uintptr** - An unsigned integer large enough to store the interpreted bits of a pointer value
- ▶ **array** - A sequence of elements of a single type. Fixed in length.
- ▶ **pointer type** - Denotes the set of all pointers to variables of a given type, called the base type of the pointer.
- ▶ **function type** - A function type denote the set of all functions with the same parameters and result types.
- ▶ **interface type** - Specifies a method set that a variable must provide.

# ZERO VALUE OF TYPES

Go provides that if the value of a variable is not declared, the value for the variable is set to the “Zero Value”.

- ▶ **Booleans** will be set to **false**
- ▶ **Integers** will be set to **0**
- ▶ **Floats** will be set to **0.0**
- ▶ **String** will be set to **""**
- ▶ **Pointers, functions, interfaces, slices, channels and maps** will start with the value of **nil**.



## VARIABLES

---

**HUH, WHAT ARE THEY GOOD FOR,  
ABSOLUTELY EVERYTHING**

-EDWIN STARR

# DECLARING A VARIABLE

- ▶ Variables are declared using the `var` keyword.
  - ▶ `var x int`, declares variable `x` and sets its type as an `int`.
- ▶ You can declare the initial value during variable declaration.
  - ▶ `var x int = 7`
- ▶ If you omit the type, Go will assign a type. Integers are commonly set to the `int` type.
  - ▶ `var x = 7`
- ▶ When inside a function, this can shortened via the `:=` operator. This says, “create a variable and assign the initial value”.
  - ▶ `x := 7` and `var x = 7` are equivalent.
  - ▶ `y := "hello world"` and `var y = "hello world"` are equivalent.

Yes

```
x := 7  
x = 3
```

No

```
x := 7  
x := 3
```

## SECTION 3

---

# DECLARING BASIC TYPES

```
// Declaring variables with the var keyword.  
var x int  
x = 7  
  
var y string = "Hello World"  
  
var z int = 7  
var a = 7  
  
// Declaring the same variables using :=  
x := 7  
y := "Hello World"  
x = 12 // Why not := ???  
  
// Declaring using := for a specific numeric type.  
x := int64(7)  
  
// Declaring multiple variables of the same type.  
var a, b, c int = 3, 2, 1  
  
// Declaring different variables, but using only one "var" keyword.  
var (  
    x float64  
    y string = "Hello World"  
    z uint32  
)
```

# EXERCISE

- ▶ Create a variable called `x` of type `int` with a value of `7` using the `var` keyword inside `main()`.
- ▶ Create a variable called `color` of type `string` with a value of “`yellow`” using the `:=` operator.
- ▶ Print out `x` and `color` using `fmt.Println()`
- ▶ Assign `x` the value of `10`.
- ▶ Print out `x`.
- ▶ Answer at: <https://play.golang.org/p/xr22BAZuqj>

# ARRAYS

- ▶ Rarely used, but important to understand.
- ▶ Arrays are a sequence of the same type of elements.
- ▶ Each entry in an **array** has an index, starting at **0**.
- ▶ **var x [2]string** declares an array of **2** string elements.
- ▶ Arrays cannot be resized.
- ▶ Useful in understanding how a **slice** works.

## SECTION 3

# ARRAYS

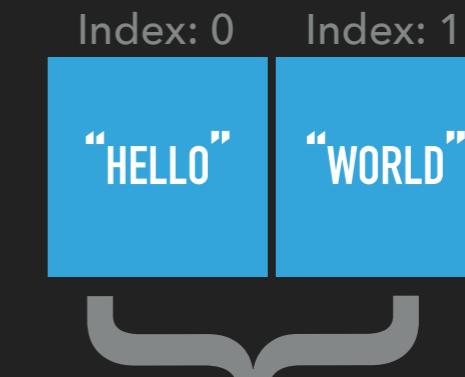
```
var myArray [2]string
```

Creates this in memory →



```
myArray[0] = "Hello"  
myArray[1] = "World"
```

Stores this in memory →



```
fmt.Println(myArray[0])  
fmt.Println(myArray[1])
```

Prints this to the screen →

```
Hello  
World
```

```
myArray[2] = "Won't work"
```

Gives compiler error →

invalid array index 2  
(out of bounds for 2-  
element array)

# REFERENCE TYPES

- ▶ Three special reference types:
  - ▶ Slices
  - ▶ Maps
  - ▶ Channels
- ▶ Reference types are created with the `make` command.  
ONLY reference types are made with `make`.

# SLICES

- ▶ Slices are a sequence of values like **arrays**.
- ▶ Each value is stored at an index starting at **0**.
- ▶ **Slices**, unlike **arrays**, can grow in size using the **append()** command.
- ▶ **Slice** data is actually stored in an **array**.
- ▶ **Slices** are views into an **array**, but do not store the underlying data itself.

## SECTION 3

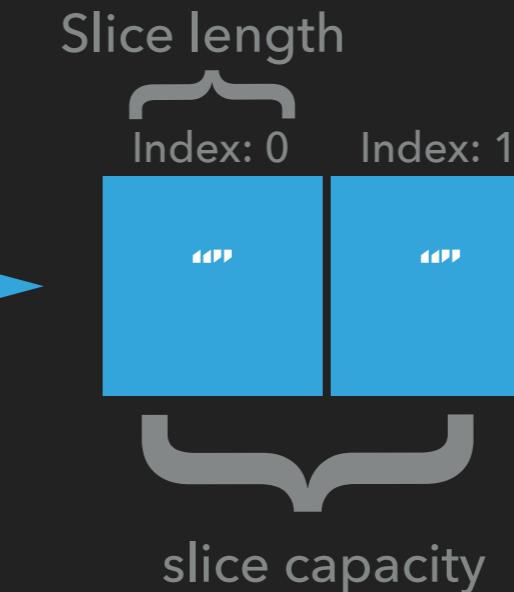
# SLICES

```
mySlice := make([]string, 1, 2)
```

The slice length

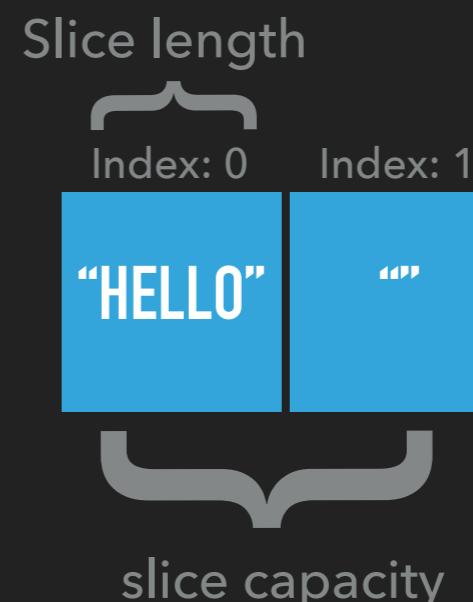
The slice's array capacity

Creates



```
mySlice[0] = "Hello"
```

Results in



```
mySlice[1] = "World"
```

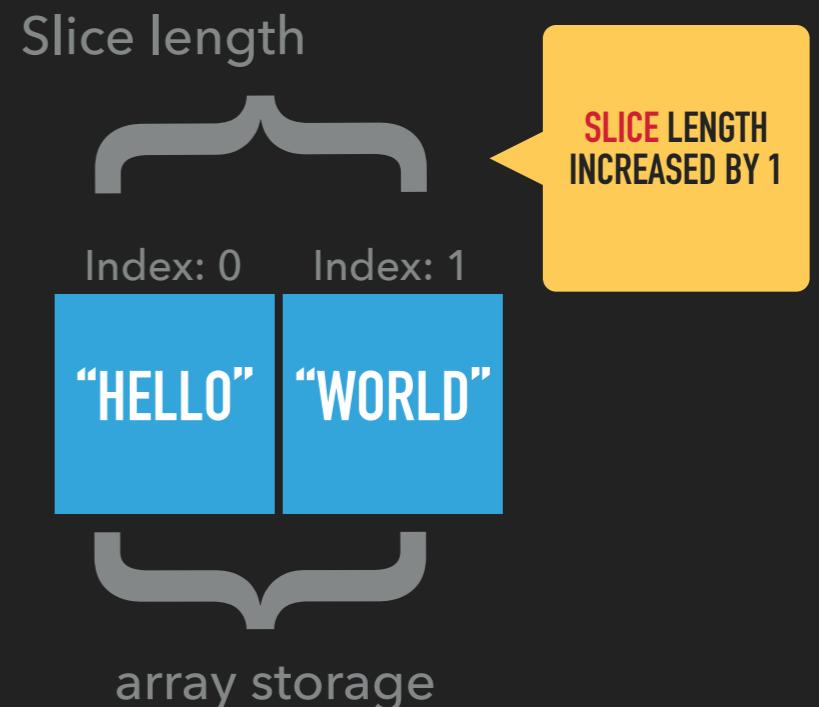
Results in

panic: runtime error: index out of range

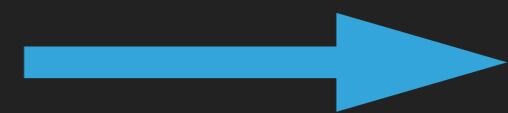
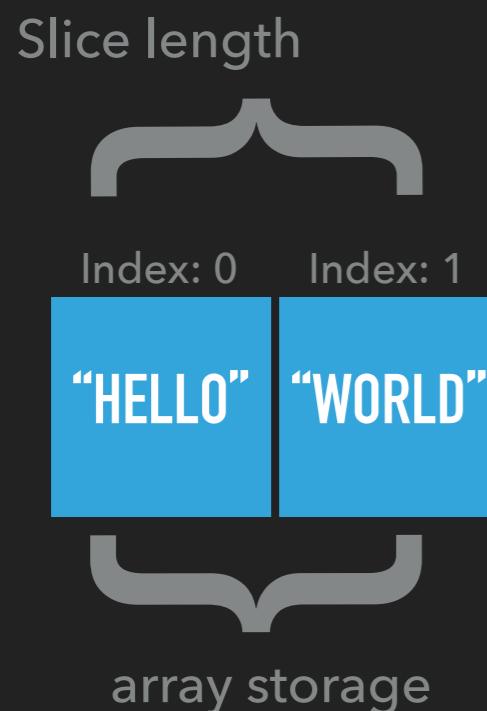
## SECTION 3

# SLICES

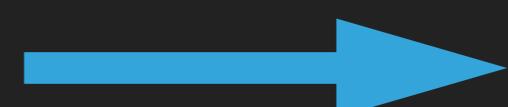
```
mySlice = append(mySlice, "World") → Results in →
```



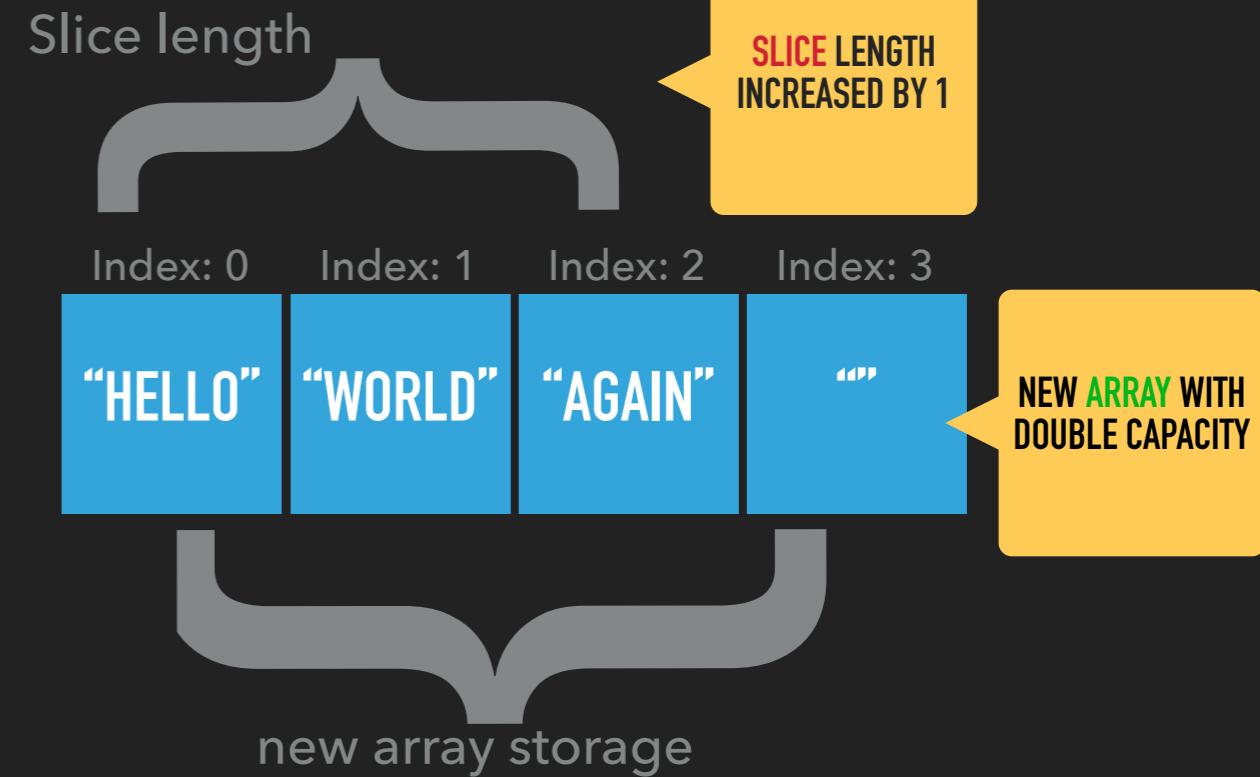
```
mySlice = append(mySlice, "Again")
```



Point slice at new array,  
increase length by 1



Copy data stored in  
array to new array  
(doubled in capacity)



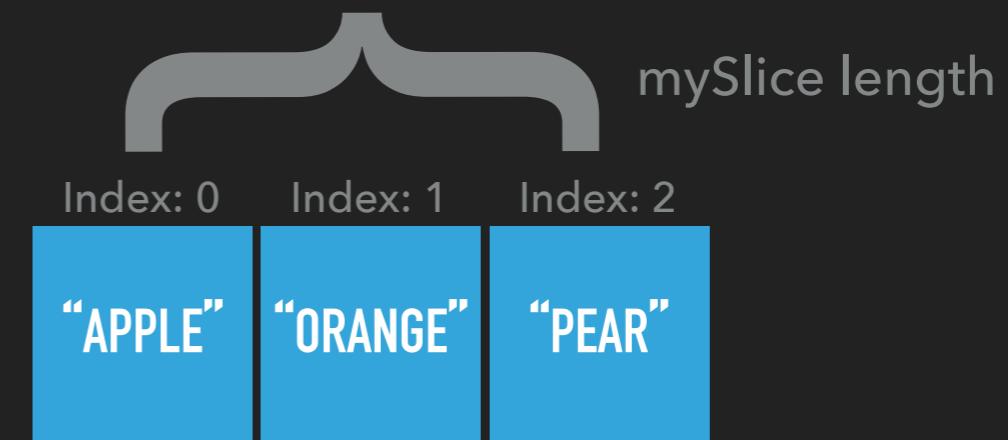
## SECTION 3

---

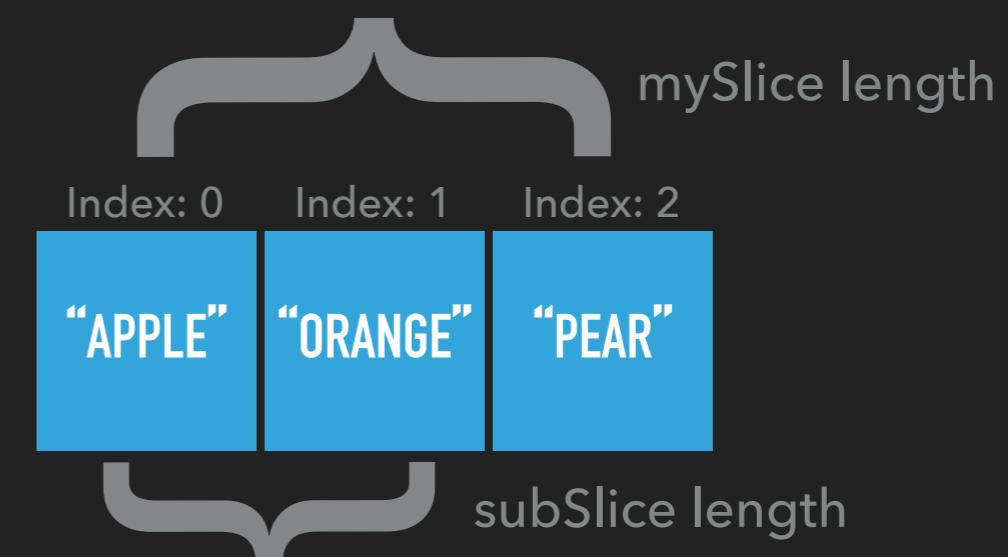
# SLICES

- You can create a new slice that has a different view of the data in another slice.

```
mySlice := make([]string, 0)
mySlice = append(mySlice, "apple")
mySlice = append(mySlice, "orange")
mySlice = append(mySlice, "pear")
```



```
subSlice = mySlice[0:2]
```



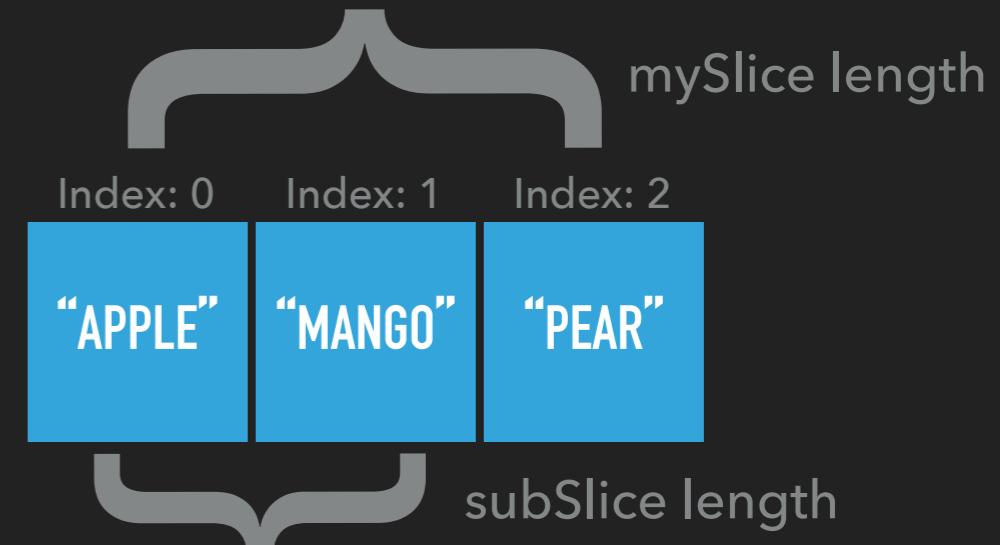
## SECTION 3

---

# SLICES

```
subSlice[1] = "mango"
```

Changes the underlying array



```
fmt.Println(subSlice[1])  
fmt.Println(mySlice[1])
```

Prints this to the screen

```
mango  
mango
```

```
subSlice[2]
```

Results in

panic: runtime error: index out of range

## SECTION 3

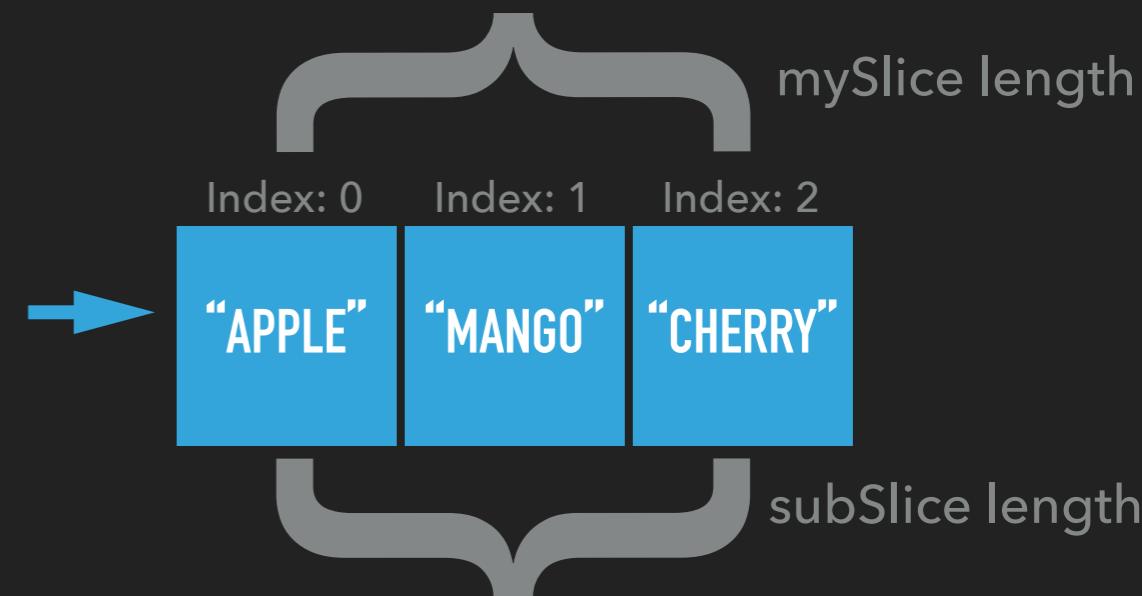
---

# SLICES

- Appending to the `subSlice` will change `mySlice`

```
subSlice = append(subSlice, "cherry")
```

Changes the underlying array and subSlice length



```
fmt.Println(subSlice[2])  
fmt.Println(mySlice[2])
```

Prints this to the screen

cherry  
cherry

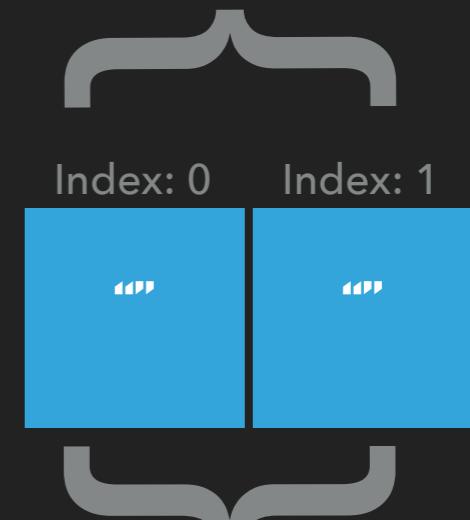
## SECTION 3

# SLICE GOTCHAS

- Using `make()` for a `slice` with only one number following it will create the length and capacity the same

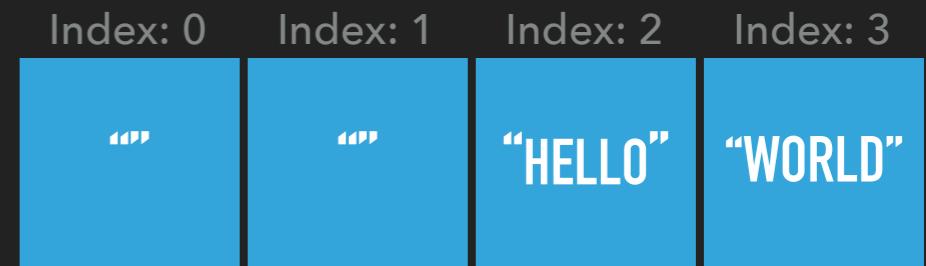
```
mySlice := make([]string, 2)
```

Slice length



```
mySlice = append(mySlice, "hello", "world")
```

Slice length



array storage

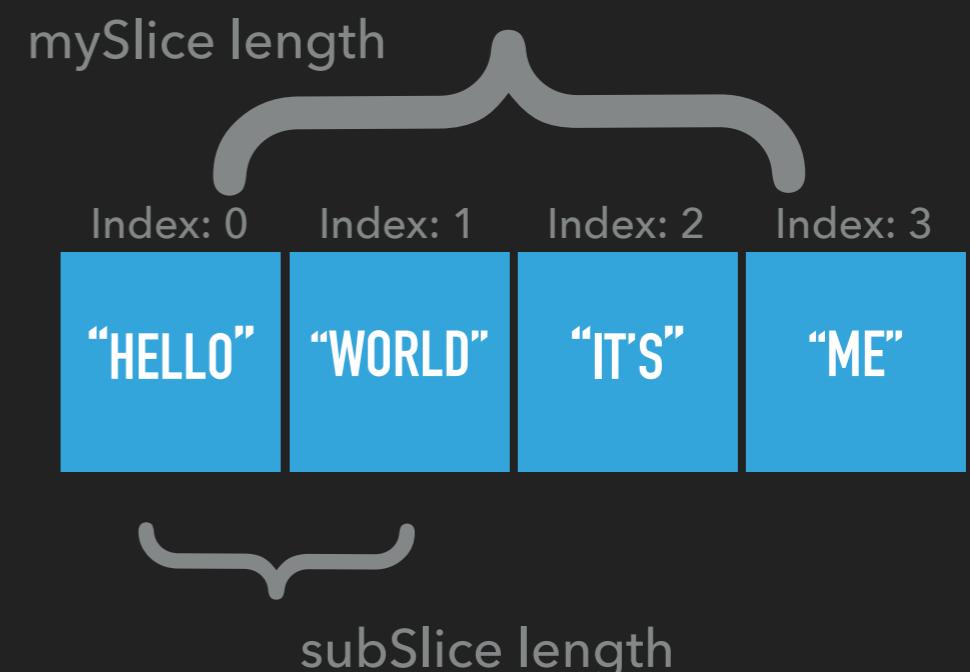
## SECTION 3

---

# SLICE GOTCHAS

- subSlice = mySlice[0:2] includes index 0 and 1, NOT 2

```
subSlice = mySlice[0:2]
```



### EXERCISE

- ▶ Create a variable called `fruits` of type `[]string`
- ▶ Add the following values to `fruits`: “apple”, “orange”, “mango”
- ▶ Print out the value at index `1` using `fmt.Println()`.
- ▶ Create a new slice called `californiaFruits` that is a slice of `fruits` containing only “apple” and “orange”.
- ▶ Print `californiaFruits` using `fmt.Printf("%#v\n", californiaFruits)`
- ▶ Answer at: <https://play.golang.org/p/e6QDM7XqVY>

# MAPS

- Maps are key/value pairs that can be used for quick lookups.
- Maps are declared using the map keyword.
- myMap := make(map[string]int) declares a map with keys that are strings and values that are ints.
- myMap["apple"] = 1, creates a key "apple" with a value of 1.
- myMap["apple"] will return 1.

## SECTION 3

---

# MAPS

```
// Creates a map with keys that are strings, values that are int.  
employeeID := make(map[string]int)  
  
// Assign key "john" to value 0.  
employeeID["john"] = 0  
  
// Assign key "bill" to value 1.  
employeeID["bill"] = 1  
  
// Print out Bill's employee id.  
fmt.Println(employeeID["bill"])  
  
// Printing out a non-existing key will provide the value's Zero Value.  
fmt.Println(employeeID["kate"])
```

Note: There is a way to detect if a key exists in a **map**. We will talk about it after **if** statements.

# MAPS: A FEW NOTES

- ▶ **Maps** can pre-allocate storage similar to a **slice** to prevent copying data when new data is added.
- ▶ `myMap := make(map[string]int, 10)` indicates that underlying storage should be able to hold **10** values.
- ▶ There is no length declaration as there is in **slices**.
- ▶ **Maps** are stored in random order. Printing **maps** out in a loop may result in different ordered items.

### EXERCISE

- ▶ Create a variable called `modelToVendor` of type `map[string]string`.
- ▶ Add the following value pairs to `modelToVendor`:
  - ▶ “focus”: “ford”
  - ▶ “prius”: “toyota”
- ▶ Print out the value at key “prius” using `fmt.Sprintf()`
- ▶ Answer at: <https://play.golang.org/p/TropD-RsiU>

## COMPOSITE LITERALS

- ▶ Composite literals are a quick method to create reference types, **arrays** or **struct** types that contain pre-defined values.

```
// Composite literal declaring an empty slice of strings.  
mySlice := []string{}  
  
// Composite literal declaring a slice of length 3 and capacity 3 storing  
// the value "apple" in index 0, "orange" in index 1, and "pear" in index 2.  
mySlice := []string{"apple", "orange", "pear"}  
  
// Composite literal declaring an empty map.  
myMap := map[string]string{}  
  
// Composite literal declaring a map with keys and associated values.  
myMap := map[string]string{  
    "key0": "value0",  
    "key1": "value1",  
    "key2": "value2",  
}
```

### EXERCISE

- ▶ Create the `modelToVendor` of type `map` again, but use a composite literal.
- ▶ Create the `fruits slice` again, but use a composite literal.
- ▶ Print out the vendor for the “`prius`” again using `fmt.Printf()`.
- ▶ Print out the entire `slice` of fruits using `fmt.Printf()`.
- ▶ Answer at: <https://play.golang.org/p/rpZddejV-r>



LOOPS

---

THERE CAN BE  
ONLY ONE!

# FOR LOOP - THE ONLY LOOP

- Go only has one type of loop, the **for** loop.
- The for loop syntax is:
  - **for** [init statement]; [conditional]; [post statement] {
    - **for** **x** := 0; **x** < 10; **x**++ {}
  - The [init statement] happens only once, on the entrance into the loop. Above, it creates a variable **x** of type **int** with the value of **0**.
  - The [conditional] is checked to see if it evaluates to true on every loop. If it is **true**, the loop continues, if not, the loop ends. Above, the loop will continue until **x** is equal to or greater than **10**.
  - The [post statement] is executed when a loop is completed before checking the [conditional] on the start of a new loop. Above, the loop will increment **x** by **1** at the end of every loop.



## FOR LOOP

- ▶ A **for** loop can also act as a **while** loop in other languages by omitting the **[init statement]** and the **[post statement]**.

```
x := 0
for x % 2 == 0 {
    fmt.Println("hello")
    x++
}
```

- ▶ The above loop will continue as long as the remainder of **x** divided by **2** is **0**. So this will loop exactly one time.
- ▶ **%** returns the remainder of a number divided by a number. **%** is called the modulus operator.

## FOR LOOP

- ▶ A **for** loop can be infinite by eliminating all statements.

```
for {  
    fmt.Println("hello")  
}
```

- ▶ The above loop will continue forever.
- ▶ You can use the term **break** inside a loop to end a loop.
- ▶ You can use the term **continue** inside a loop to stop the current loop execution and start from the top. This will still execute the [post statement].

## SECTION 4

---

### EXERCISE

- ▶ Use a **for** loop to print from **1** to **100** (but not **0**).
- ▶ Answer at: [https://play.golang.org/p/O6lj\\_wfla3](https://play.golang.org/p/O6lj_wfla3)

# FOR LOOP WITH RANGE

- ▶ `range` lets you extract key/value from `maps`, index/values from `slices/arrays`, and output from `channels` in a loop.
- ▶ `range` is always used on the same line as `for`.
- ▶ `range` output depends on the `type` you are extracting from.

## SECTION 4

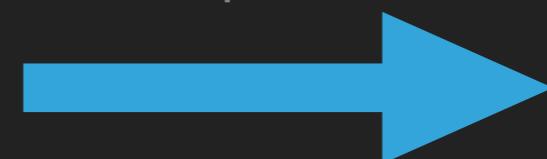
---

# FOR LOOP WITH RANGE

- When looping through **slices/arrays**, **range** has the following syntax:
  - for <index>, <value> := range <slice variable> {}**

```
slice := []string{"a", "b", "c"}  
for index, value := range slice {  
    fmt.Printf("%d: %s\n", index, value)  
}
```

Output



```
0: a  
1: b  
2: c
```

## SECTION 4

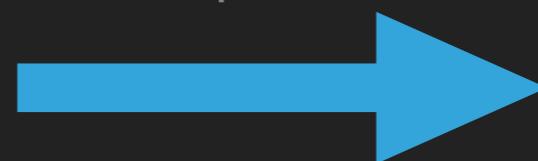
---

# FOR LOOP WITH RANGE

- When looping through **maps**, **range** has the following syntax:
  - for <key>, <value> := range <map variable> {}**

```
m := map[string]string{
    "a": "apple",
    "b": "banana",
}
for key, value := range m{
    fmt.Printf("%s: %s\n", key, value)
}
```

Output



```
b: banana
a: apple
```

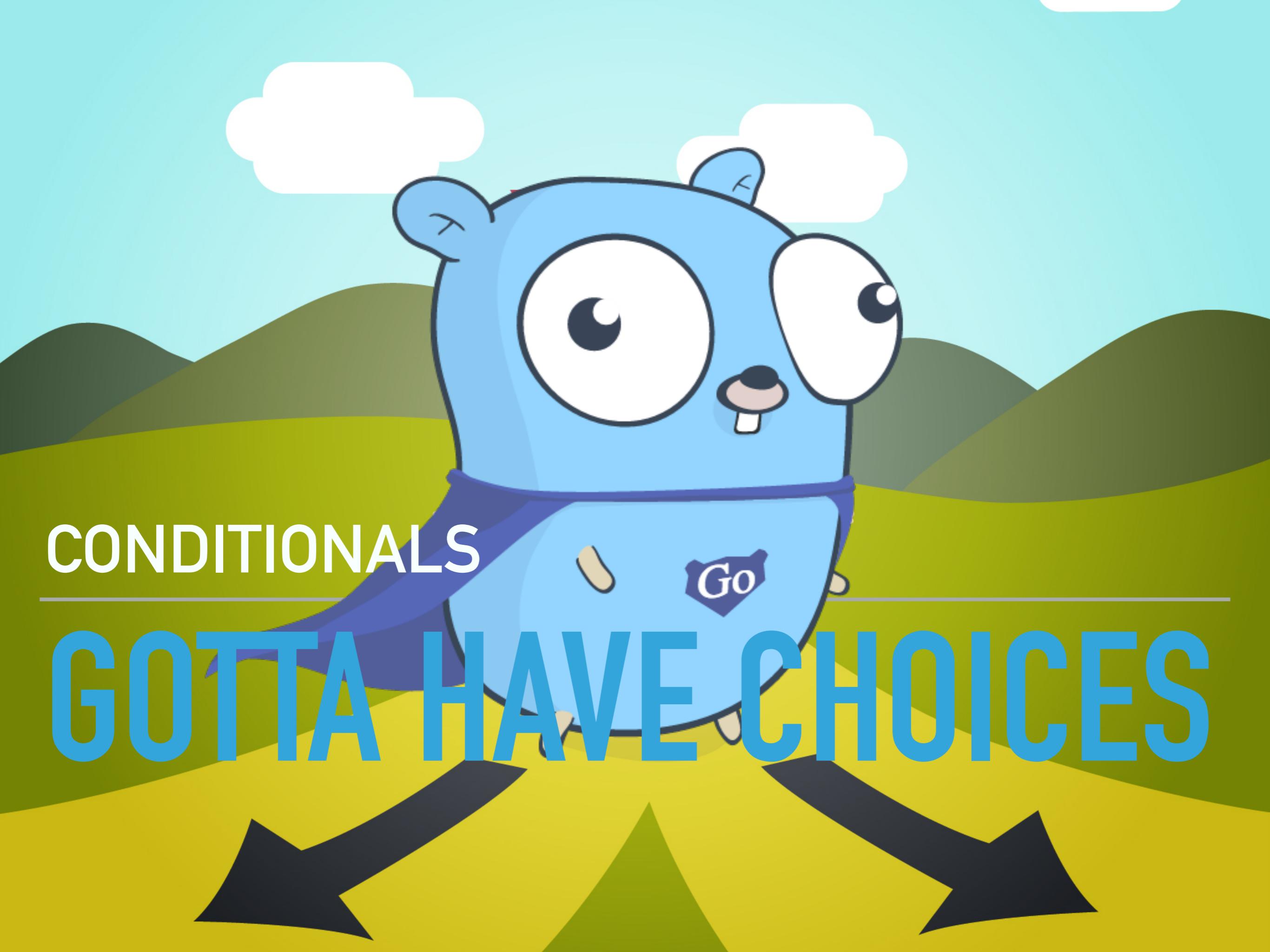
Order is non-deterministic

# FOR LOOP GOTCHA

- When doing loops, you may not want to use the index or the value.
- Go requires that if you declare a variable, you must use it or you get a compiler error.
- To solve this, you may use the `_` character in place of a variable name to indicate you wish to ignore the returned value.
  - `for _, <value> := range <map/slice variable> {}`
    - This will ignore the key/index of the slice or map
  - If you only want the keys or indexes in a loop, it is customary to use another loop format:
    - `for <key/index variable> := range <map/slice variable> {}`
      - Omitting the second variable will produce just the key or index.

# EXERCISE

- ▶ Create a **for** loop that prints out the following **slice** of names: “bob”, “ann”, “marie”, “steve”.
- ▶ Create a **for** loop that prints out the following **map** key/values: 0: “john”, 1: “mary”, 2: “stevie”.
- ▶ Answer at: <https://play.golang.org/p/CgbJ8meBd2>

A cartoon illustration of a blue bear with large white eyes and a small pink nose. It has a purple collar with a gold tag that says "Go". The bear is surrounded by three white speech bubbles. The background consists of green rolling hills under a light blue sky.

CONDITIONALS

---

GOTTA HAVE CHOICES

## IF/ELSE

- ▶ The **if** statement is your basic conditional statement.
- ▶ **if** comes in two forms:
  - ▶ **if [conditional] {}**

```
if x < 10 {  
    fmt.Println("x is less than 10")  
}
```

- ▶ **if [statement]; [conditional] {}**

```
if x := y + z; x < 10 {  
    fmt.Println("x is less than 10")  
}
```

## IF/ELSE

- ▶ The **if** can be combined with **else if** and **else** to handle multiple choices.
- ▶ **else** and **else if** MUST be on the same line as a closing bracket **}**. It cannot be on a separate line as in some languages.

```
if x < 10 {  
    fmt.Println("x is < 10")  
}else if x == 11 {  
    fmt.Println("x is 11")  
}else {  
    fmt.Println("x is > 11")  
}
```

# SWITCH

- ▶ The **switch** statement is a more advanced **if/else** statement.
- ▶ **switch** comes in a number of forms:

```
switch [value] {  
    case [match]:  
        [statement]  
    case [match], [match]:  
        [statement]  
        [statement]  
    default:  
        [statement]  
}
```

Example

```
switch x {  
    case 10:  
        fmt.Println("x == 10")  
    case 20:  
        fmt.Println("x == 20")  
    case 30, 40:  
        fmt.Println("x == 30 or 40")  
    default:  
        fmt.Println("don't know what x is")  
}
```

- ▶ In this form, a **case** statement is executed if any of the matches are equal to the value. The value and the matches must be the same type.
- ▶ **Default** executes if no match == value. **default** may be omitted.
- ▶ The first **case** to match executes. Any following cases do not.
- ▶ The **case** executes all statements inside the **case**.

# SWITCH

- Another form simply adds an [init statement]:

```
switch [init statement]; [conditional] {  
    case [match]:  
        [statement]  
    case [match], [match]:  
        [statement]  
        [statement]  
    default:  
        [statement]  
}
```

Example



```
switch x := y % 2; x {  
    case 0:  
        fmt.Println("x == 0")  
    case 1:  
        fmt.Println("x == 1")  
}
```

- The [init statement] in the example is not very practical, it is just an example. This might be the output of a function or the combination of multiple variables in the real world.
- There is no **default** statement. If **x** was anything but **0** or **1** (it can't be in the example), nothing would execute and the code would continue.

## SWITCH

- Another form uses each case statement as a conditional instead of a match:

```
switch {  
    case [conditional]:  
        [statement]  
    case [conditional]:  
        [statement]  
    default:  
        [statement]  
}
```

Example

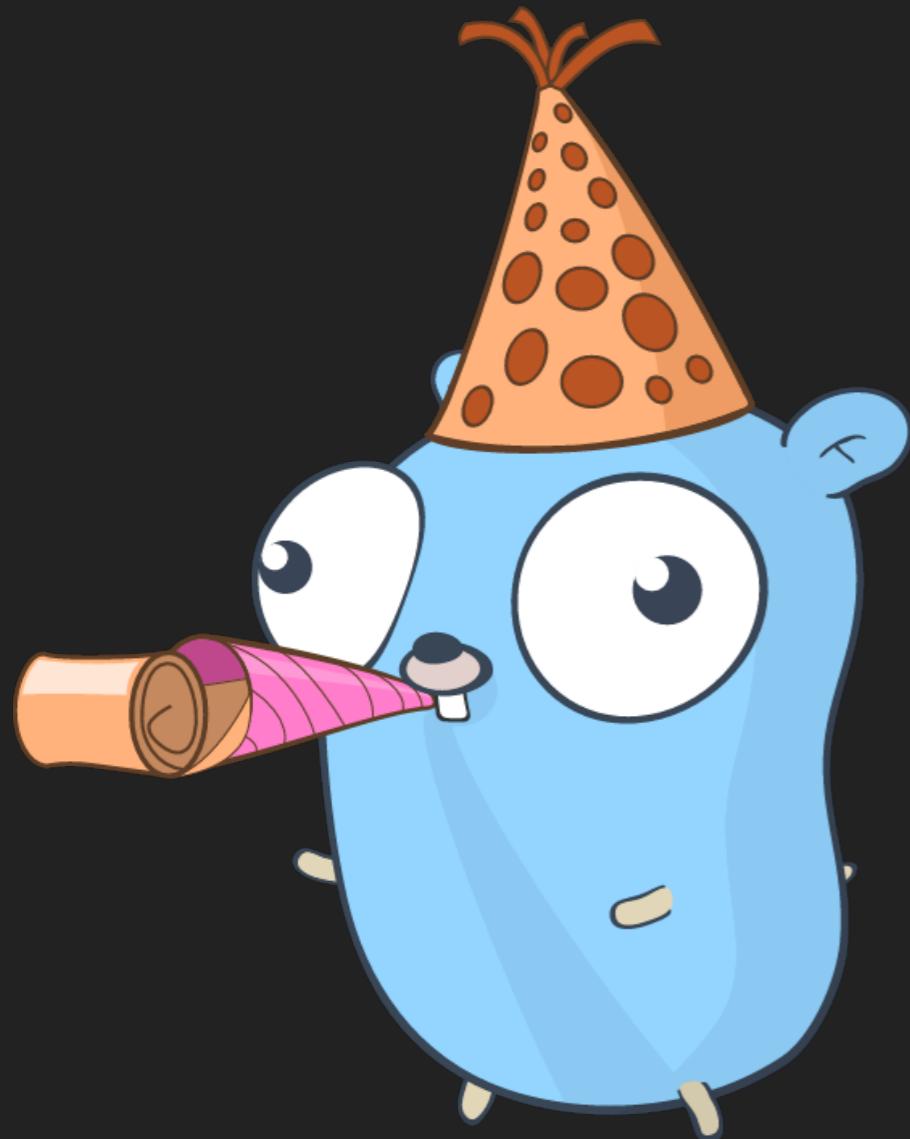


```
switch {  
    case x < 10:  
        fmt.Println("x < 10")  
    case x > 10:  
        fmt.Println("x > 10")  
}
```

- In this form, each conditional is matched against **true/false**. If the case conditional == **true**, the case statements are executed.

# EXERCISE

- ▶ Write an **if/else** statement that:
  - ▶ Prints “**even**” if an integer variable **x** is an even number
  - ▶ Print “**odd**” if it is not.
  - ▶ Hint: use the modulus **%** operator
- ▶ Write a **switch** statement that:
  - ▶ Prints “**unix**” if a string variable **os** is “**linux**”, “**bsd**”, “**osx**”
  - ▶ Print “**microsoft os**” if os is “**windows10**”
  - ▶ Prints “**unknown os**” if **os** is none of the above.
- ▶ Answer at: <https://play.golang.org/p/dhgFm6P-3p>



# FUNCTIONS

---

## ALL THE GOOD LANGUAGES HAVE THEM

## SECTION 6

---

# FUNCTIONS

- Functions are defined using the **func** keyword:

```
func <function name>([var name] [var type], [var name] [var type], ...) ([return value], [return value], ...) {}
```

- A simple example of a function that adds two numbers together called **add()**:

Example use of function **add()**

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

```
func main() {  
    fmt.Println(add(2, 2))  
}
```

- add()** is defined as a function that takes in two integers, **x** and **y**. It returns an integer.
- Inside **add()**, we use the **return** keyword to return the results of adding **x** and **y** together.
- The function's code is enclosed in curly brackets {}

## SECTION 6

---

# FUNCTIONS

- ▶ Functions can also save space by omitting the some of the **types** when multiple variable share the same type.
- ▶ Functions can return multiple values

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

Example use of function **addMultiply()**

```
func main() {  
    a, m := addMultiply(3, 3)  
    fmt.Printf("3 + 3 is %d\n", a)  
    fmt.Println("3 * 3 is ", m)  
}
```

- ▶ **x** and **y** are both type **int**.
- ▶ We return two values, both are **int**. If you return multiple values, they must be in parenthesis **()**. This can be omitted for single value returns.

## SECTION 6

---

# FUNCTIONS

- ▶ Functions can also have named returns.

```
func addMultiply(x, y int) (add, multiply int) {  
    add = x+y  
    multiply = x *y  
    return  
}
```

- ▶ Notice that in the function, we do not create `x` or `y` with `:=`? That is because they are already declared.
- ▶ Your return does not have to contain anything, because we will return the values in `add` and `multiple`.
- ▶ If you do not set a value for a named return, it returns the Zero Value. In this case, it would be `0`. This is a common gotcha.

```
func addMultiply(x, y int) (add, multiply int) {  
    return x+y, x*y  
}
```

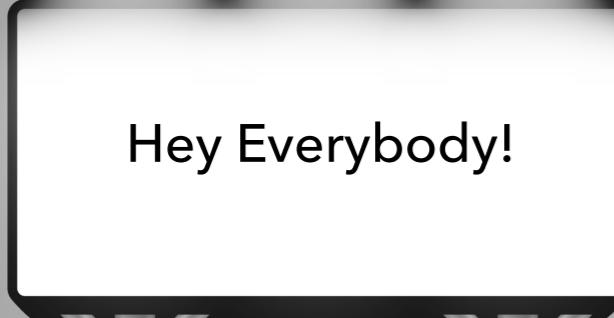
- ▶ This also works. In this case, `add` and `multiply` are there only to give additional information to the user.

# EXERCISE

- ▶ Create a function that:
  - ▶ Is named **divide**.
  - ▶ Takes in two **ints**.
    - ▶ The first is the number to divide (dividend).
    - ▶ The second is the number to divide by (divisor).
  - ▶ Returns the result of the division.
  - ▶ Call the **divide()** function from **main()** and print the result.
- ▶ Answer at: <https://play.golang.org/p/WW6Hop1EQn>

# EXERCISE

- ▶ Change **divide()** so that it:
  - ▶ Returns both the result of the division and the remainder.
  - ▶ Uses named returns.
  - ▶ Hint: use the modulus **%** operator
- ▶ Answer at: <https://play.golang.org/p/-6QKd5pRyD>



Hey Everybody!

VS.



PUBLIC VS. PRIVATE

---

WHAT TO SHOW AND  
WHEN TO SHOW IT

# PUBLIC AND PRIVATE

- Go provides both **Public** and **private** variables, functions, and methods.
- **private** variables, functions, custom types and methods may only be accessed inside a package.
- **Public** variables, functions , custom types and methods may be accessed by other packages.
- To be a **Public** variable, function or custom type, it must be at the root of the package (not inside a function or method).
- Declaring a Public or private variable/function/... is easy:
  - **Public** variables/functions/... starts with a CAPITAL LETTER.
  - **private** variables/functions/... start with a lowercase letter.

## SECTION 7

---

# PUBLIC AND PRIVATE

- ▶ Example:

```
package stuff

import "fmt"

func PrintHello() {
    fmt.Println("Hello")
}

func printWorld() {
    fmt.Println("World")
}

func PrintHelloWorld() {
    PrintHello()
    // printWorld is private, but accessible because
    // it is in the same package.
    printWorld()
}
```

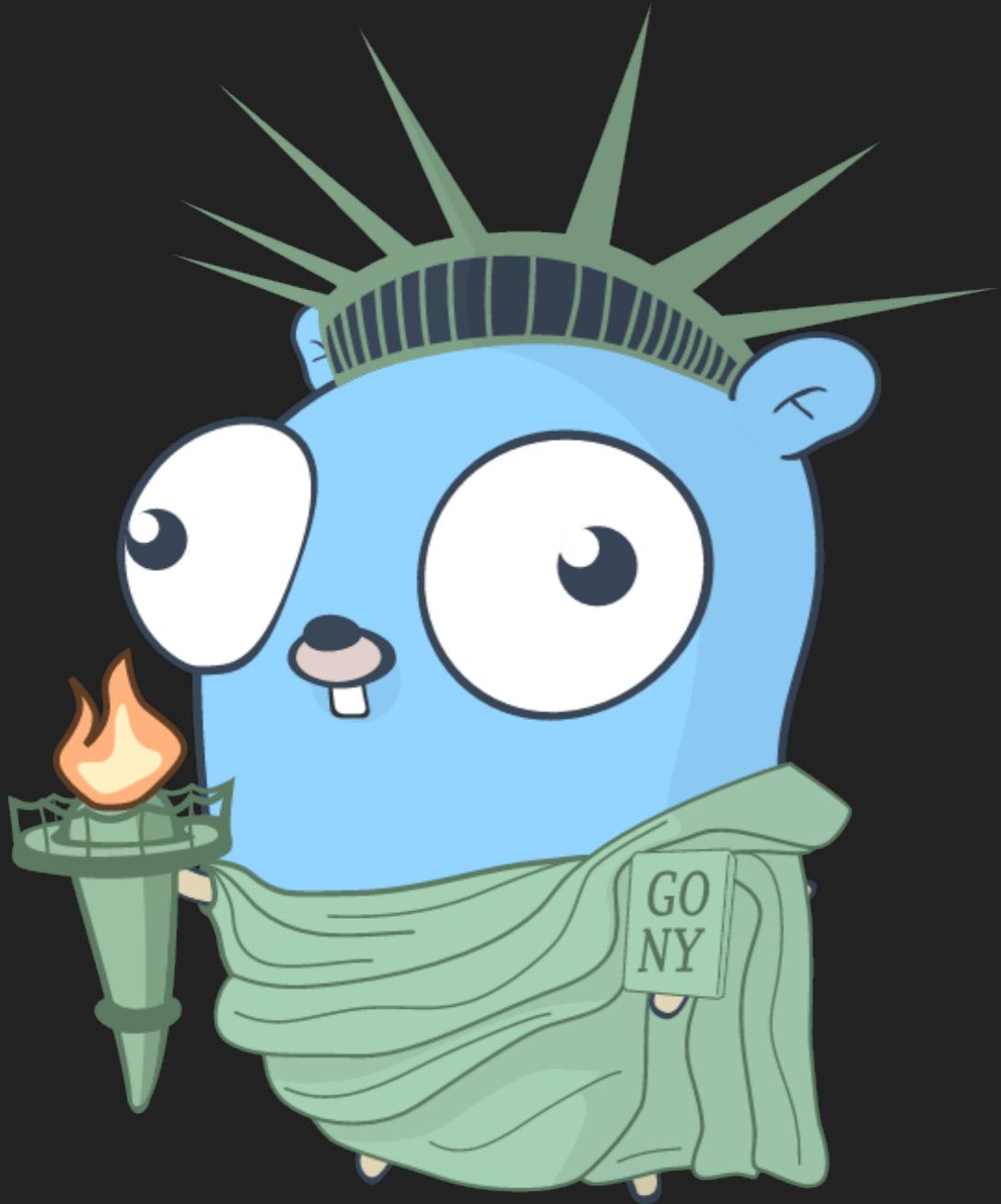
```
package main

import ".../stuff" // ... just means some path.

func main() {
    // This will work.
    stuff.PrintHello()

    // This will work.
    stuff.PrintHelloWorld()

    // Will get a compile error.
    stuff.printWorld()
}
```



## VARIABLE SCOPING

---

**EVERYTHING THAT HAS A  
BEGINNING, HAS AN ENDING  
- FAKE BUDDHA QUOTE**

# VARIABLE SCOPING

- ▶ Variable scope - the range in the code where a variable may be referenced.
- ▶ In Go, variables can be two types:
  - ▶ Package Global - they are defined outside any functions.
  - ▶ Package Local - they are defined inside a function.
- ▶ Global variables can be referred inside any function.
- ▶ If the Global is public, it can be referred to by other packages.

## SECTION 8

---

# VARIABLE SCOPING

- ▶ Examples of global scope variables.

```
package blah

var (
    // Multiplier can be accessed by outside packages by referencing
    // blah.Multipplier
    Multiplier = 3

    // divider can only be accessed within the package.
    divider = 2
)

func IncreaseByMultiplier(x int) int {
    return x * Multiplier
}

func DivideByTwo(x int) int {
    return x/divider
}
```

```
package main

import ".../blah" // ... stands for a directory path.

func main() {
    blah.Multipplier = 10

    // blah.divider = 2 // Won't work, divider isn't public.

    fmt.Println(blah.IncreaseByMultiplier(4))
}
```

## VARIABLE SCOPING

- ▶ Local variables have limited scope. They exist only between the closest braces { }, with one exception.
- ▶ The exception is if the variable is defined within an **init statement** of a **for** loop, **if/else** statement, or **switch statement**.
- ▶ If defined in an **if init statement**, the the starting brace { of the **if** to the final } of an **else** or **if/else** is the scope.

## SECTION 8

---

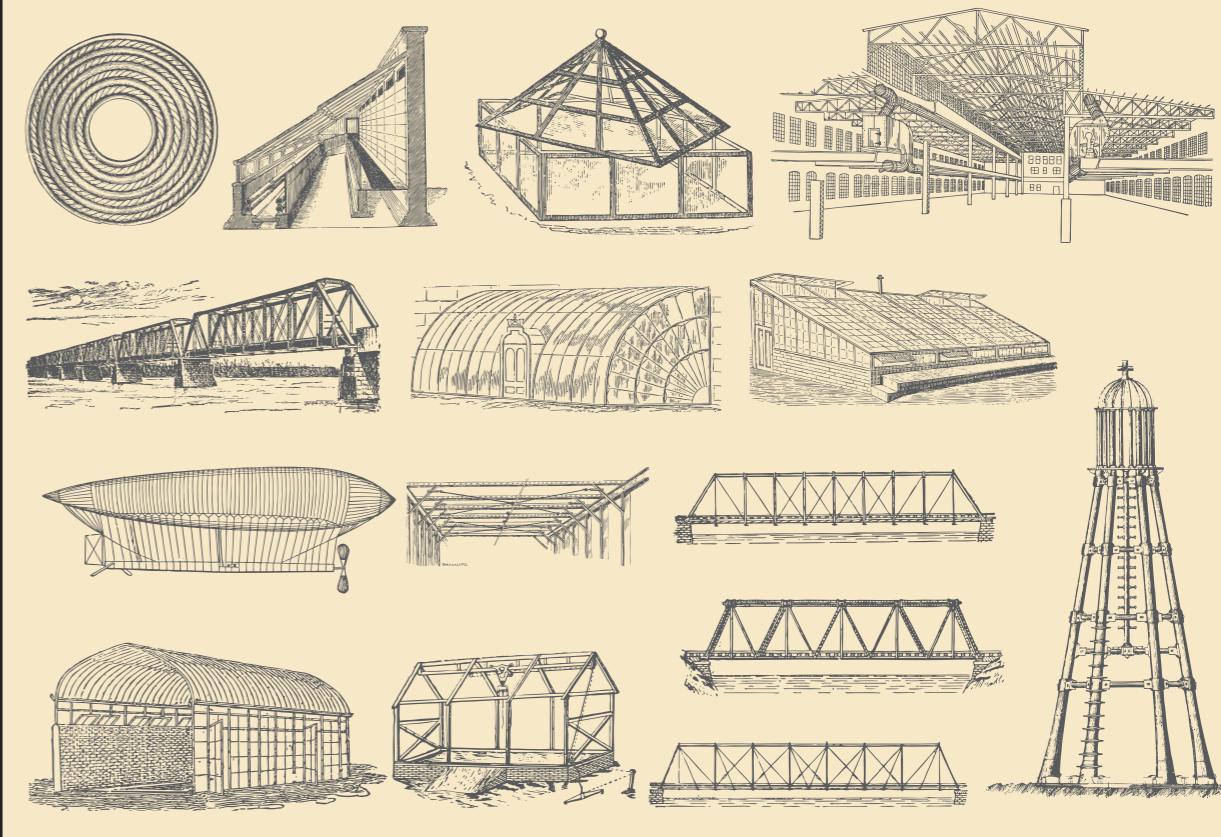
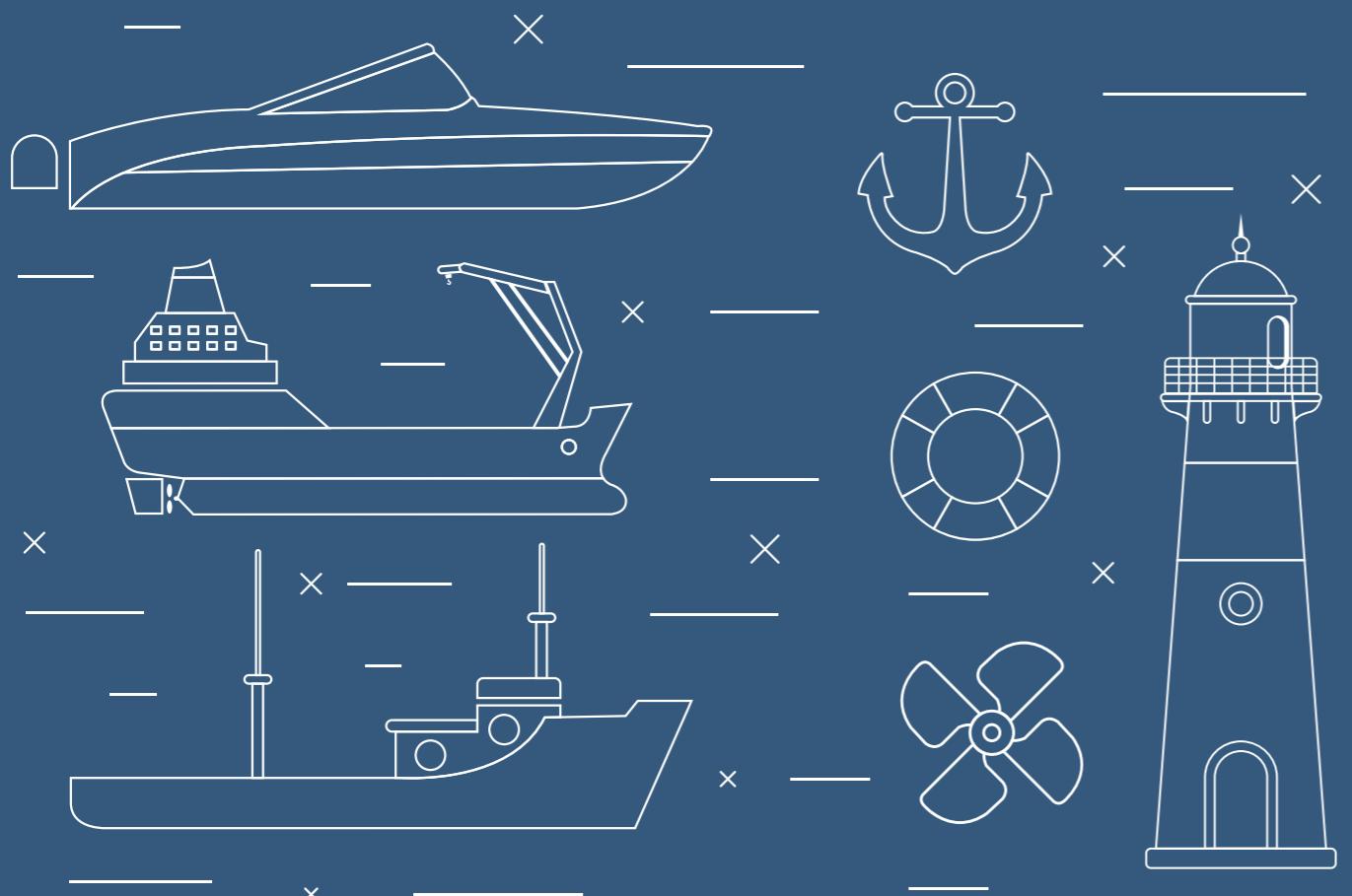
# VARIABLE SCOPING

```
func someFunc() {
    // str can be accessed by anything inside someFunc().
    // It cannot be accessed outside someFunc().
    str := "Hello World"
    fmt.Println(str)

    // i is accessible only inside the for loop.
    // It cannot be accessed outside the for loop.
    for i := 0; i < 10; i++ {
        fmt.Println(i)
    }
    // fmt.Println(i) – uncommenting this will be a compile error.

    // If you need for a loop variable to exist outside the loop, do this:
    var x int
    for x = 0; x < 10; x++ {
        fmt.Println(x)
    }
    // This works because x is declared outside the for loop.
    fmt.Println(x)

    // We create a variable z that is a random integer
    // using the standard library "math/rand".
    if z := rand.Int(); z == 0 {
        fmt.Printf("we randomly got 0 back")
    }else if z < math.MaxInt32 {
        // We can see z here because we are still within the entire if/else if/else.
        fmt.Printf("our random number %d fits in an int32", z)
    }else{
        // Same as above.
        fmt.Printf("our random number %d fits in an int64", z)
    }
    // fmt.Println(z) – uncommenting this will be a compile error.
}
```



# STRUCTS

---

**WHAT IS IN A NAME? THAT WHICH WE CALL A  
CLASS BY ANY OTHER NAME WILL SMELL AS  
SWEET - WILLIAM SHAKESPEARE**

# STRUCTS

- ▶ Go inherits the struct type from the C language.
- ▶ Structs provide groupings of named attributes.
- ▶ This is the closest language primitive to classes in other languages.
- ▶ Structs are normally used by creating customized structs using the **type** keyword.

# STRUCTS

- ▶ An example of a custom struct that represents an employee record:

```
// EmployeeRecord holds information about a company's employee.  
type EmployeeRecord struct {  
    // FirstName is the first name of the employee.  
    FirstName string  
    // LastName is the last name of the employee.  
    LastName string  
    // ID is the employee's unique employee ID.  
    ID int  
}
```

- ▶ Creating an instance of the struct:

```
rec := EmployeeRecord{  
    FirstName: "John",  
    LastName: "Doe",  
    ID: 0,  
}  
  
fmt.Println("The employee's first name is: %s", rec.FirstName)
```

The `.` operator  
allows access to the  
struct's fields.

# STRUCTS

- ▶ Structs can have methods attached to them, similar to classes of other languages.
- ▶ Methods are like functions with two key exceptions:
  - ▶ Methods can receive the struct without passing it as an argument.
  - ▶ Methods can only be called on an instance of a struct.

## SECTION 9

---

# STRUCTS

- Let's add a method that returns a string for printing out a record:

```
// EmployeeRecord holds information about a company's employee.  
type EmployeeRecord struct {  
    // FirstName is the first name of the employee.  
    FirstName string  
    // Lastname is the last name of the employee.  
    LastName string  
    // ID is the employee's unique employee ID.  
    ID int  
}  
  
func(e EmployeeRecord) String() string {  
    // Sprintf is like Printf except that instead of writing to the screen,  
    // it returns a string.  
    return fmt.Sprintf("First: %s;Last: %s;ID: %d", e.FirstName, e.LastName, e.ID)  
}
```

This provides access to the fields in Employee record by using the variable named `e` and attaches it to the struct.

We access the fields here via `e.<field name>`

## STRUCTS

- Let's use our method to print the record in our preferred form:

```
rec := EmployeeRecord{  
    FirstName: "John",  
    LastName: "Doe",  
    ID: 0,  
}  
  
fmt.Println(rec.String())
```

- Will print: First: John;Last: Doe;ID: 0

# EXERCISE

- ▶ We are going to simply recreate something similar to what we did in the previous slides. Note the differences.
- ▶ Create a struct called Record
  - ▶ Create field **First** of type **string**
  - ▶ Create field **Last** of type **string**
  - ▶ Create field **ID** of type **int**
- ▶ Add method **String()** onto **Record**
  - ▶ It should return a string that represents the **Record** in CSV:
    - ▶ **<ID>,<First>,<Last>**
- ▶ Answer at: <https://play.golang.org/p/bvPSxtwVdk>

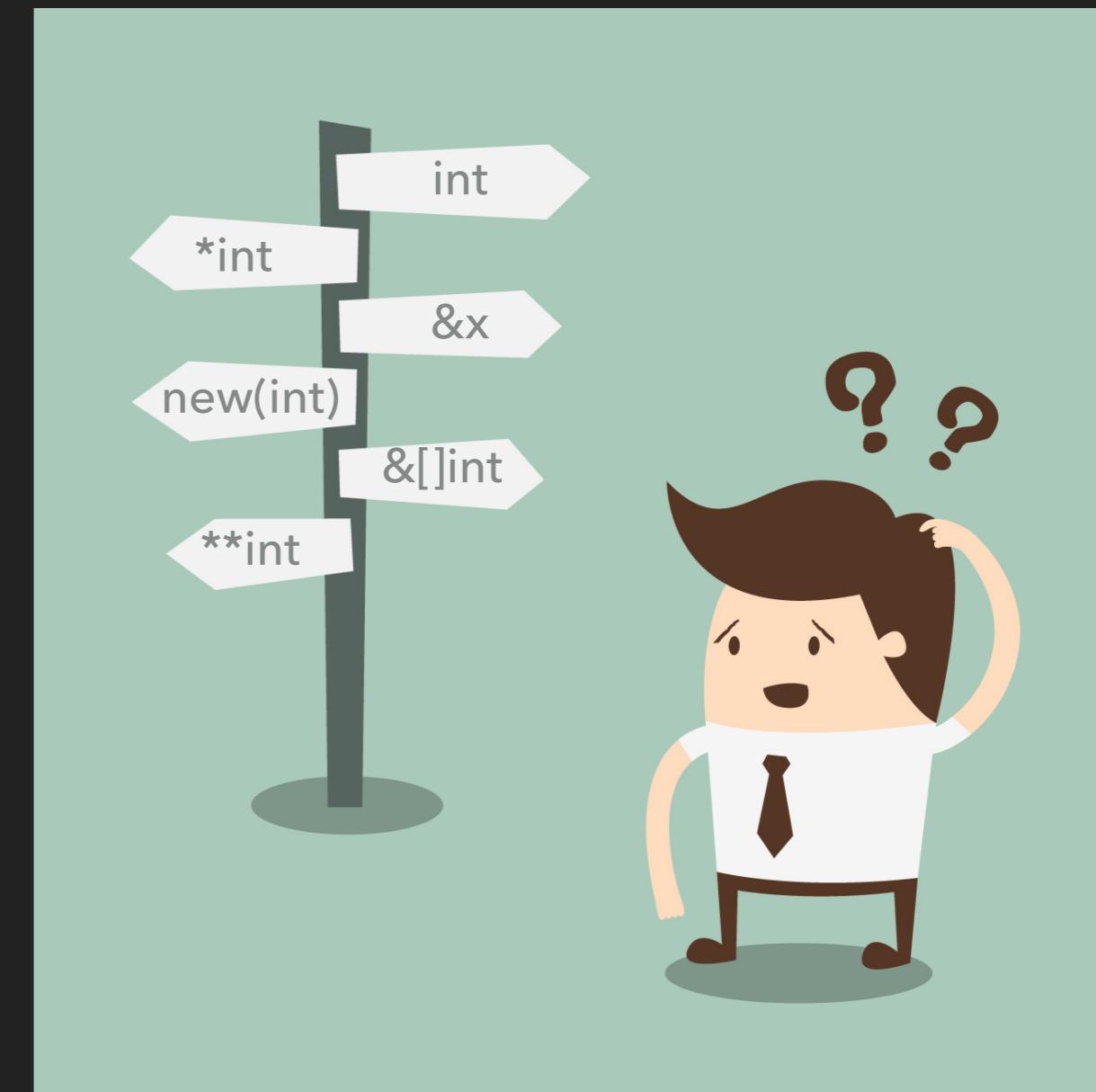


SAVE THIS EXERCISE  
FOR USE LATER

## WHAT'S THE POINTER

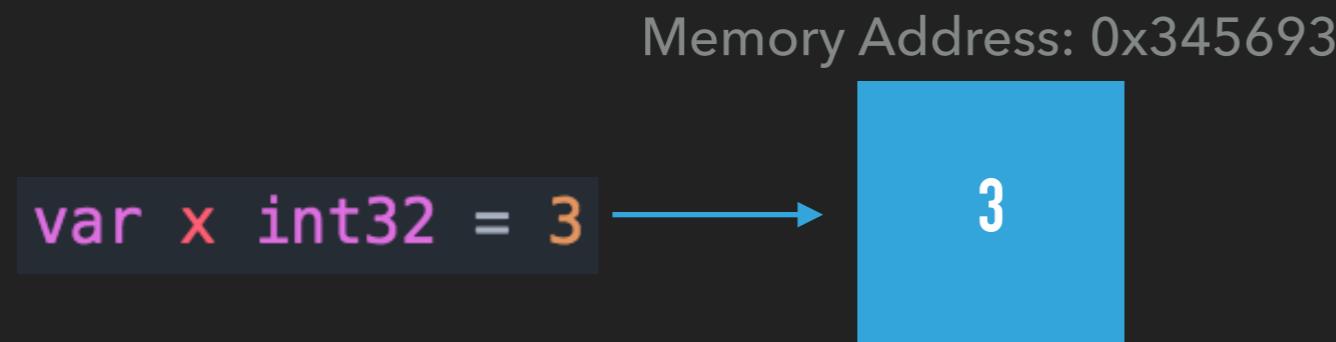
---

POINTER HUMOR, ALWAYS A  
BIG HIT AT THE PARTIES



# UNDERSTANDING POINTERS

- ▶ To understand pointers, it's important to understand what happens when you create a variable, in simple terms.



- ▶ When a variable is created, such as the `int x` represents, memory must be allocated to store that value.
- ▶ In the instance above, a place in memory is created, addressed with a fictional address of `0x345693`.
- ▶ That memory address must be 32 bits in size to store the 32 bit number.

# UNDERSTANDING POINTERS

```
func changeInt32(x int32) {  
    x = 10  
    fmt.Printf("Inside changeInt32: %d\n", x)  
}  
  
func main() {  
    var x int32 = 100  
    changeInt32(x)  
    fmt.Printf("After changeInt32: %d\n", x)  
}
```

Outputs



Inside changeInt32: 10

After changeInt32: 100

- Not what you were expecting? Why did it not change?
- The value didn't change because values are copied when going into a function. It is **ALWAYS** copied.

# UNDERSTANDING POINTERS

- It is also important to understand how passing variables to functions work.

1. `var x int32 = 100`

Memory Address: 0x345693



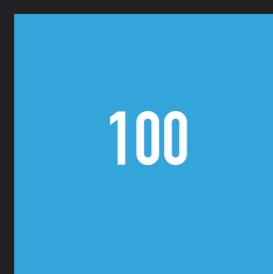
3. `func changeInt32(x int32) {`

Memory Address: 0x987345 (new memory address)



2. `changeInt32(x)`

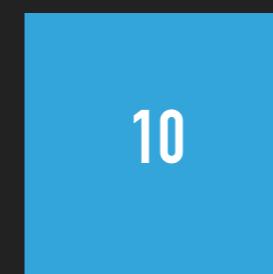
Memory Address: 0x345693



4.

`x = 10`

Memory Address: 0x987345



5.

`}`

Function ends

Memory Address: 0x987345



Memory  
reclaimed

# UNDERSTANDING POINTERS

- ▶ Pointers are types whose value is a memory address.
- ▶ This is similar to an int whose value is an integer.
- ▶ By using the Pointer's memory address, you can access the value that is stored at the address.
- ▶ By passing pointers to variables instead of the value of a variable, you can change values inside a function.

# UNDERSTANDING POINTERS

- ▶ A pointer type can be created several ways:
  - ▶ `var intptr *int`
    - ▶ Creates a pointer to a `int`. Because it doesn't point at anything, it gets the Zero Value, which is `nil`.
  - ▶ `var intptr = new(int)` or `intptr := new(int)`
    - ▶ `new` creates an address in memory to store an `int` and then returns a pointer which points to that address. The value in the memory address is the Zero Value of the type, which in this case is `0`.
  - ▶ `intptr := &x`
    - ▶ `x` in this case would hold an `int`.
    - ▶ `&` indicates to return the address where variable `x` is stored.

# UNDERSTANDING POINTERS

- You can access a value stored by a pointer by dereferencing the pointer.
- Dereferencing is done by putting a \* in front of the pointer name.

```
intptr := new(int)

// This will print 0, the Zero Value stored at the place in memory
// intptr points to.
fmt.Println(*intptr)

// This changes the value stored in intptr to 25.
*intptr = 25

// This will print 25.
fmt.Println(*intptr)

// This will print the memory address that intptr stores.
fmt.Println(intptr)

// intptr = 10 - Uncommenting this will prevent this from compiling.
```

- Try it at: <https://play.golang.org/p/4pOyueVEWD>

## UNDERSTANDING POINTERS

- Let's try the `changeInt32` example again, using pointers.

```
func changeInt32(x *int32) {  
    *x = 10  
    fmt.Printf("Inside changeInt32: %d\n", *x)  
}  
  
func main() {  
    var x int32 = 100  
    changeInt32(&x)  
    fmt.Printf("After changeInt32: %d\n", x)  
}
```

- Try it at: <https://play.golang.org/p/9k7Kp5BJBm>

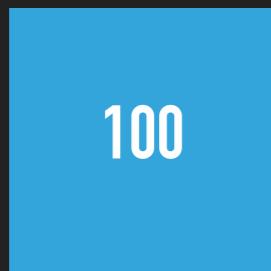
# UNDERSTANDING POINTERS

- ▶ BUT WAIT! You said that you always copy values when you are going into a function. So why do pointers let us change the value?
- ▶ **Answer:** We are still copying the values, but the value has changed from being an **int** to a memory address.
- ▶ When we dereference the copy of the memory address, it is still the same address.

# UNDERSTANDING POINTERS

1. `var x int32 = 100`

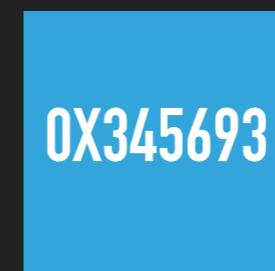
Memory Address: 0x345693



Creates new variable in memory

3. `func changeInt32(x *int32) {`

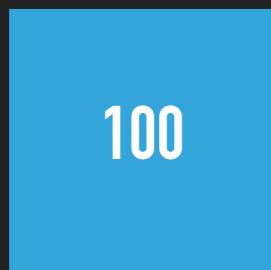
Memory Address: 0x987345 (new memory address)



Copies value into new variable. Value is the memory address.

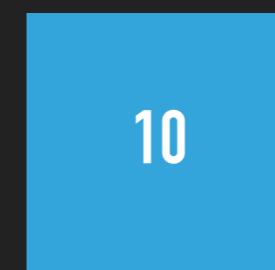
2. `changeInt32(&x)`

Memory Address: 0x345693



4. `*x = 10`

Memory Address: 0x345693



Value Changed



Memory Address: 0x987345

Memory reclaimed

5. `}` Function ends

# EXERCISE

- ▶ Create a function called **changeStr()**
  - ▶ Takes in a pointer to a **string**
  - ▶ Changes the string to “Hello World”
- ▶ In **main()**:
  - ▶ Create a **string** variable that stores “Howdy Partner”
  - ▶ Print the **string** variable.
  - ▶ Call **changeStr()** passing the **string** pointer.
  - ▶ Print the **string** variable.
- ▶ Answer at: <https://play.golang.org/p/b1gueGy6ul>

# UNDERSTANDING POINTERS

- ▶ A note on reference types
  - ▶ Reference type act as if they have a built in pointer.
  - ▶ If you pass a reference type to a function and change it in a function, you change it outside too.
  - ▶ Reference types are **maps**, **slices** and **channels**.

# UNDERSTANDING POINTERS

- ▶ A **slice** gotcha:
  - ▶ Changing a **slice** in a function indeed changes the **slice** outside the function.
  - ▶ However, altering the length of the **slice** does not change the length outside. To do that you must either return the **slice** or pass a pointer to the **slice**.

```
func changeSlice(s []string) {  
    s[0] = "blueberry"  
    s = append(s, "watermelon")  
}  
  
func main() {  
    s := []string{"apples"}  
    changeSlice(s)  
    fmt.Printf("%#v\n", s)  
}
```

Outputs



[]string{"blueberry"}

- ▶ Try it at: <https://play.golang.org/p/1NR9sZVl7y>

# UNDERSTANDING POINTERS

- ▶ Solutions to the gotcha:

```
func changeSlice(s *[]string) {  
    *s = append(*s, "watermelon")  
}  
  
func returnSlice(s []string) []string {  
    return append(s, "bananas")  
}  
  
func main() {  
    s := []string{"apples"}  
  
    changeSlice(&s)  
    fmt.Printf("%#v\n", s)  
  
    s = returnSlice(s)  
    fmt.Printf("%#v\n", s)  
}
```



## Outputs

`[]string{"apples", "watermelon"}`

`[]string{"apples", "watermelon", "bananas"}`

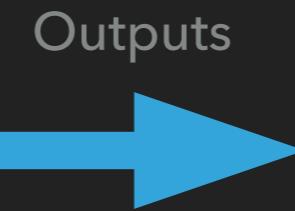
- ▶ Try it at: <https://play.golang.org/p/-Rlm4M9KVdJ>

# UNDERSTANDING POINTERS

- Struct gotcha's to look out for:
  - Many people forget that a struct is not a reference type. You often need to use a pointer to a struct.
  - Methods need to use pointers to change fields, otherwise they are changing copies of the struct, not the struct.

```
type someStruct struct{
    field string
}
func (s someStruct) change() {
    s.field = "hello"
}

func main() {
    s := someStruct{field: "world"}
    s.change()
    fmt.Println(s.field)
}
```



world

We wanted "hello"

- Try this at: <https://play.golang.org/p/crTa2KrgZ0>

# UNDERSTANDING POINTERS

- ▶ Gotcha's fix:

```
type someStruct struct{
    field string
}
func (s *someStruct) change() {
    s.field = "hello"
}

func main() {
    s := &someStruct{field: "world"}
    s.change()
    fmt.Println(s.field)
}
```

Outputs  
→

hello

- ▶ Try this at: <https://play.golang.org/p/36hivvD3Tz>

# UNDERSTANDING POINTERS

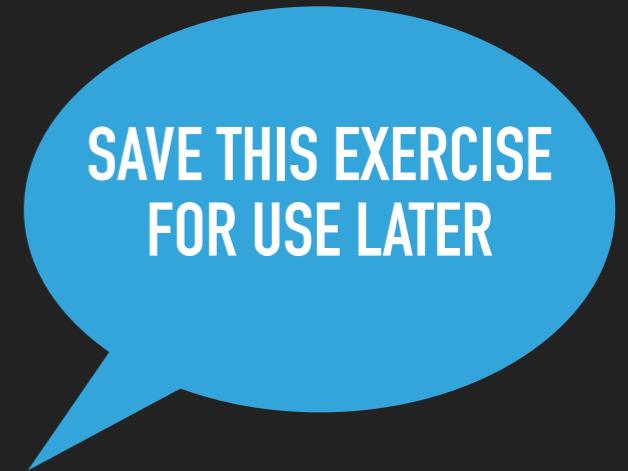
- ▶ Final Notes on Pointers
  - ▶ A pointer to a pointer is as deep as you should go. This isn't Inception.
  - ▶ Don't overuse pointers. Use them where copying data is expensive. While we used them on `int` and `string` in our examples, generally you do this on structs and not basic types.
  - ▶ Copying data between functions (which happens on the stack) is usually much faster than allocating pointers and then copying them between functions (which requires heap allocation).
  - ▶ The default `nil` value in references and pointers is useful for detecting empty values.

# EXERCISE

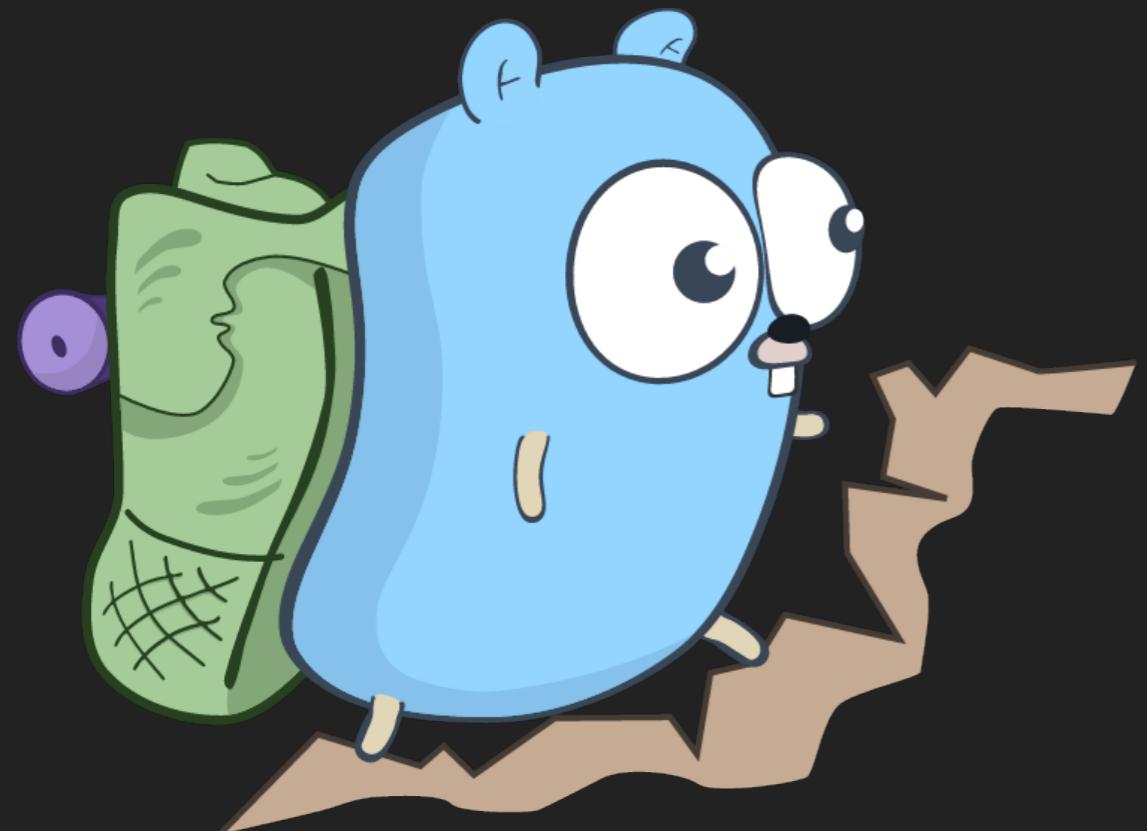
- ▶ Exercise concept:
  - ▶ We want to create a digital file cabinet storing employee records.
  - ▶ We want to create a record for an employee.
  - ▶ We want the record to have a unique ID.
  - ▶ We want to store the record in the file cabinet.
  - ▶ We want to be able to print all records in the cabinet in the order they were added.

# EXERCISE

- Load the “Record” exercise you were asked to save.
- Create a new struct call `FileCabinet`.
  - Add a field called `Files` of type slice of `Record`.
  - Add a field called `nextID` of type `int`.
  - Create a method called `AddRecord` on `FileCabinet`.
    - Should have arguments named `first` and `last` which are of type `string`.
    - Method should create a new `Record` with an ID that is equal to `nextID`.
    - Method should increment `nextID` by 1.
    - Method should store the new `Record` in `FileCabinet.Files`
  - In `main()`:
    - Create a variable called `fc` of type `*FileCabinet`.
    - Call `fc.AddRecord()` 3 times with different names.
    - Use a `for` loop with `range` to print out all the entries in `fc.Files`
  - Answer is at: <https://play.golang.org/p/Hdy4Mi3eyt>



SAVE THIS EXERCISE  
FOR USE LATER



## DETECTING KEYS IN MAPS

---

**LOST VARIABLES ARE  
NEVER FOUND AGAIN**

**-BENJAMIN FRANKLIN**

# FINDING KEYS IN MAPS

- ▶ Map lookups have two different syntaxes:

- ▶ Retrieve a value at key:

```
<value> := mapVar[<key>]
```

- ▶ Retrieve a value at key, determine if key exists:

```
<value>, <found bool> := mapVar[<key>]
```

- ▶ Example of key lookup:

```
m := map[string]int{"apples": 1}

// Does "apples" exist as a key in "m". If so,
// return the value stored and boolean true.
if v, ok := m["apples"]; ok {
    fmt.Printf("Found key %q with value %d", "apples", v)
} else{
    fmt.Printf ("Key %q not found", "apples")
}
```

# EXERCISE

- ▶ Create a **map** called **authorToBooks**
  - ▶ Keys are **strings**
  - ▶ Values are a **slice** of **strings**
  - ▶ Use a composite literal to add the following:
    - ▶ "joseph heller": []**string**{"catch 22", "something happened"}
    - ▶ "nate silver": []**string**{"the signal and the noise"}
  - ▶ In **main()**:
    - ▶ Create a **slice** of authors who are and are not in the map.
    - ▶ Loop through the **slice** looking to see if the author exists. If so, print their books one at a time in a **for** loop.
    - ▶ If the author is not in the **map**, print that you cannot find the author.
- ▶ Answer at: <https://play.golang.org/p/Ic5u5pHxbx>

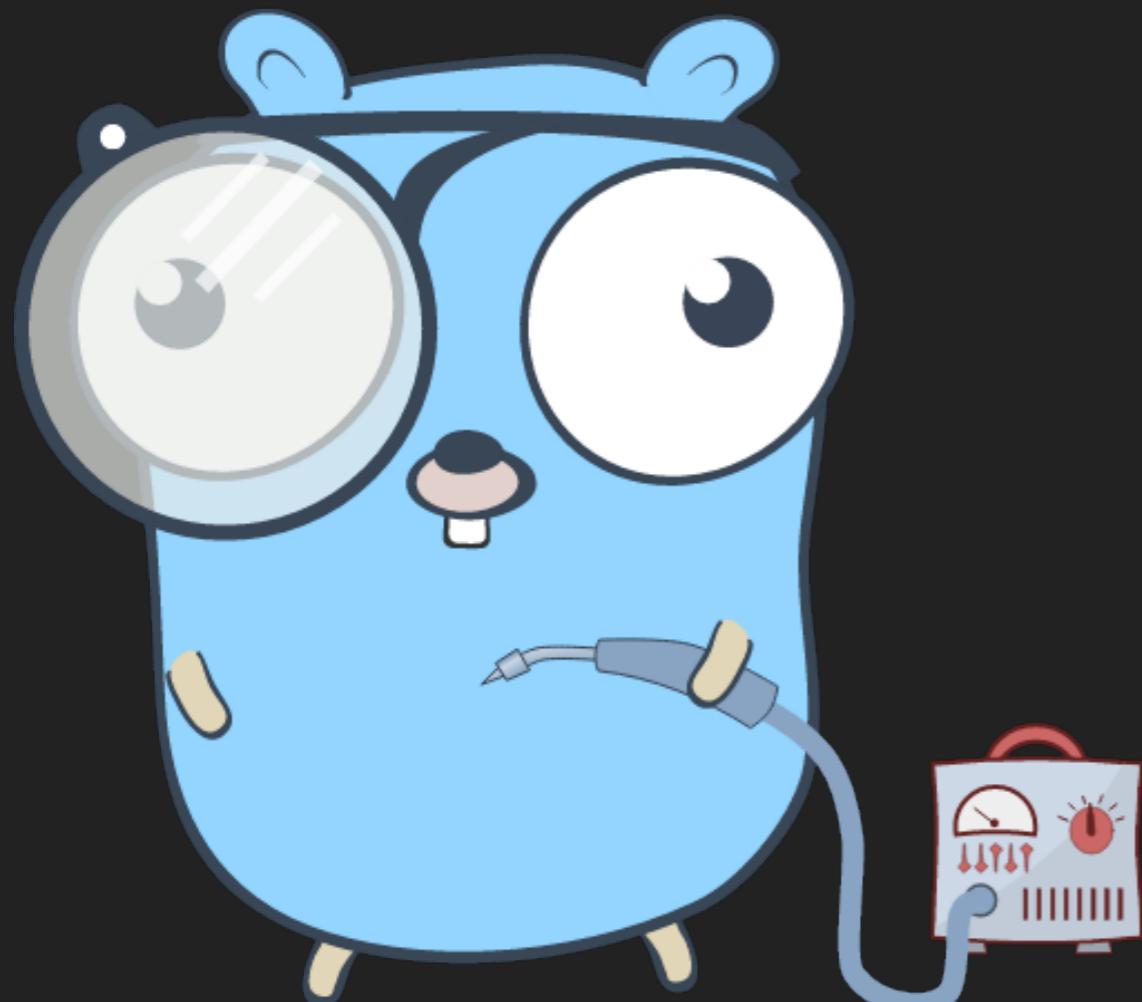
## FINDING KEYS IN MAPS

- ▶ When using a **map** to only detect if a key exists, not to retrieve a value, we can use a **map** with value type of **bool**.
- ▶ This makes detection quicker:

```
// foodsILike is a map of foods like. I'm very selective.  
var foodsILike = map[string]bool{  
    "bbq": true,  
    "milkshakes": true,  
}  
  
// willEat determines if I will eat a meal that contains various foods.  
// I'm picky, so if I don't like one food, I don't eat the meal.  
func willEat(meal []string) bool {  
    for _, f := range meal {  
        if !foodILike[f]{  
            return false  
        }  
    }  
    return true  
}
```

# EXERCISE

- Create a **map** called `goodMovies`
  - Keys are **strings**
  - Values are **bools**
  - Use a composite literal to add the following:
    - `"princess bride": true`
    - `"blade runner": true`
    - `"memento": true`
    - `"i'm gonna get you sucker": true`
  - In `main()`:
    - Loop through a **slice** containing a good movie and `"prometheus"`.
    - Print if its a good movie or not.
- Answer at: <https://play.golang.org/p/GiXj2dizdz>



## VARIADIC FUNCTIONS

---

ALL YOU NEED IN LIFE IS  
VARIADIC FUNCTIONS AND  
CONFIDENCE, AND SUCCESS IS  
SURE - MARK TWAIN

## VARIADIC FUNCTIONS

- › Variadic functions are simply functions that can take a variable number of arguments.
- › In Go, you can specify a variable number of arguments using the `...` operator.
- › `...<type>` can only be the last variable in the function call.
- › Inside the function call, the variable will be a `[]<type>`.

```
func Sum(nums ...int) int {  
    s := 0  
    for _, n := range nums {  
        s +=n  
    }  
    return s  
}
```

## VARIADIC FUNCTIONS

- ▶ Why not just use `[]int` in the previous example?
- ▶ Simply a nicety, you can use `[]int`:

```
func Sum(nums []int) int {  
    s := 0  
    for _, n := range nums {  
        s += n  
    }  
    return s  
}
```

- ▶ But doesn't this look nicer?

```
x := Sum(1, 5, 7, 9)
```

- ▶ Than this?

```
x := Sum([]int{1, 5, 7, 9})
```

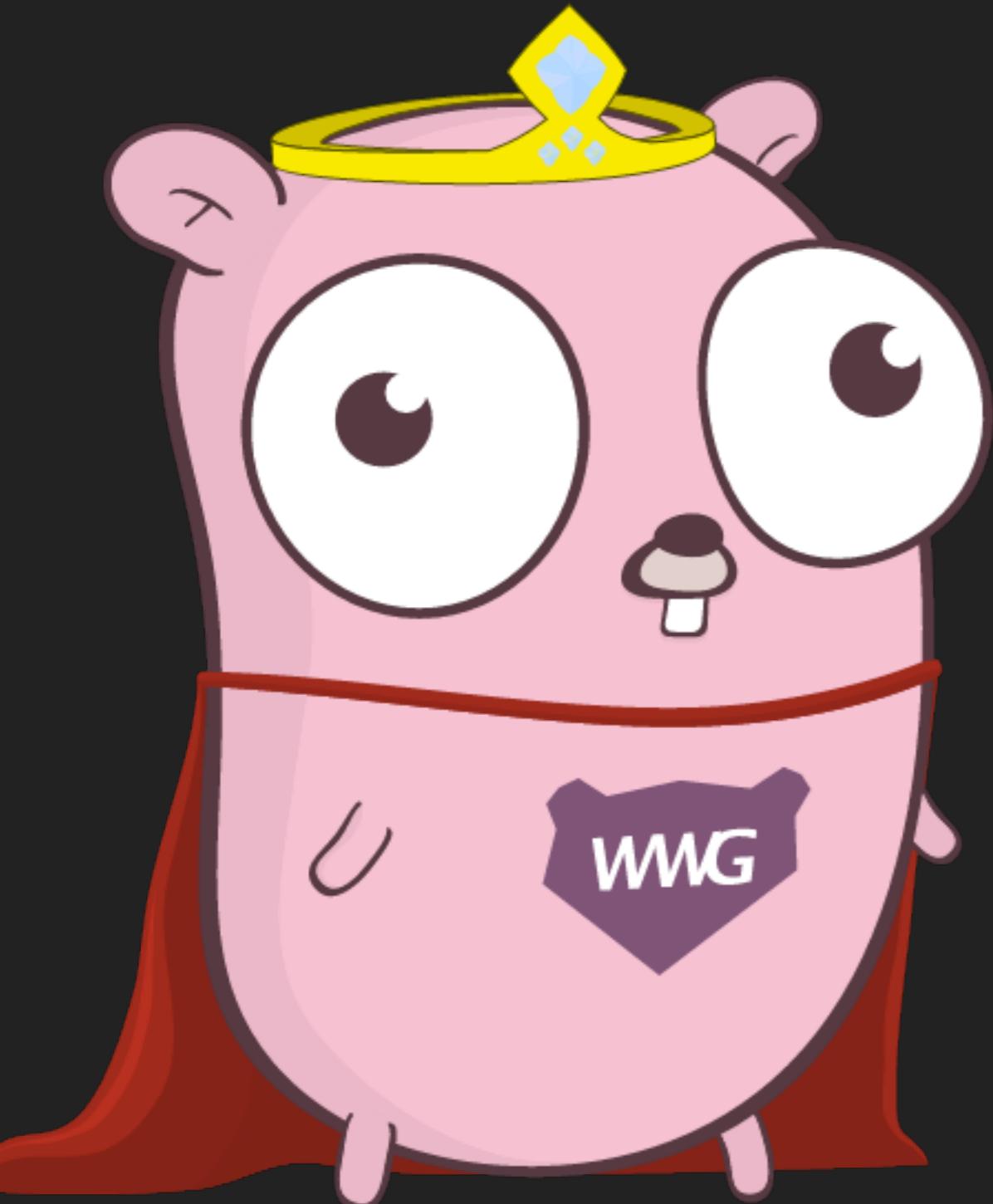
# EXERCISE

- ▶ Create a function called **PrintStrs**.
- ▶ Has an argument called **strs** that is a variable number of **string** arguments.
- ▶ It prints each argument on its own line.
- ▶ In **main()**:
- ▶ Use **PrintStrs()** to print out a list of strings.
- ▶ Answer at: <https://play.golang.org/p/pMeaXsGmTk>

## VARIADIC FUNCTIONS

- ▶ The ... operator can also be used to unpack a []type into a function's variable arguments.
- ▶ For example, append() takes variable arguments of a type, but doesn't take a slice of a type.
- ▶ By putting the ... after the a slice type, it unpacks the slice into individual variable arguments.
- ▶ Example:

```
fruits := []string{"mango", "pears"}  
californiaFruits := []string{"oranges", "apples"}  
  
// Lets append californiaFruits onto fruits.  
fruits = append(fruits, californiaFruits...)  
  
// fruits = append(fruits, californiaFruits) - will not work
```



## ERROR HANDLING

---

BAD CODE BREAKS SOFTWARE  
AND AFTERWARD, SOME CODE IS  
STRONG AT THE BROKEN PLACES  
-ERNEST HEMINGWAY

# ERROR HANDLING

- Go does not attempt to handle error conditions with exceptions that many other languages use.
- Instead, Go couples the ability to have multiple return variables and an error type called `error`.
- The `error` type has a Zero Value of `nil`, which can be checked with a simple `if` statement.
- Errors are generally created in two ways:
  - For simple text `errors` with no variable substitution:
    - `import "errors"`
    - `errors.New("my error message")`
  - For `errors` with variable substitution:
    - `import "fmt"`
    - `fmt.Errorf("i didn't like input: %s", str)`

# ERROR HANDLING

- A common type of error to deal with is the inability to open a file.
- This example tries to print a file's content using a standard library called **ioutil** to read a file.
- If it can't, we return an **error**. The caller can then determine how to deal with that **error**.

```
// PrintFile prints the content of file fn to the screen.
func PrintFile(fn string) error {
    content, err := ioutil.ReadFile(fn)
    if err != nil {
        // We could just return the error, but this allows us to add more
        // context to the error returned by ioutil.ReadFile().
        return fmt.Errorf("could not read file %s: %s", fn, err)
    }

    fmt.Println(string(content))

    return nil
}

func main() {
    files := []string{"exists", "does not exist"}

    for _, fn := range files {
        if err := PrintFile(fn); err != nil {
            fmt.Println("Error: %s", err)
        }
    }
}
```

# EXERCISE

- ▶ Create a function called **Divide()**
  - ▶ Takes in two arguments:
    - ▶ **Dividend** of type **int**
    - ▶ **Divisor** of type **int**
  - ▶ Returns an **int** and an **error**
  - ▶ Detects if the **Divisor** is **0**. If so, it returns the **int Zero Value** and an **error** stating you cannot divide by **0**.
- ▶ In **main()**:
  - ▶ Call **Divide()** for a number divided by a positive number
  - ▶ Call **Divide()** for a number divided by **0**.
  - ▶ Print each result or **error**.
- ▶ Answer at: <https://play.golang.org/p/7gn-d1Kai7>



## ANONYMOUS FUNCTIONS

---

THE PROBLEM WITH LOSING  
YOUR ANONYMOUS FUNCTIONS  
IS THAT YOU CAN NEVER GO BACK  
-MARLA MAPLES

# ANONYMOUS FUNCTIONS

- ▶ Anonymous functions are simply functions that are defined without a name.
- ▶ You can define anonymous functions and assign them to a variable (variables can be a function type).
- ▶ You can define an anonymous function inside another function.
- ▶ Anonymous functions are useful for features such as **defer** and **goroutines**, which we will use later.
- ▶ Example:

```
func helloWorld() {  
    func(str string){  
        fmt.Println(str)  
    }("hello world") // Notice the ("hello world") here? This is where we call the function.  
}
```

## ANONYMOUS FUNCTIONS

- ▶ Assign to a variable example:

```
printer := func(str string) {
    fmt.Println(str)
}

printer("I don't think the quotes in the class are correct")
```

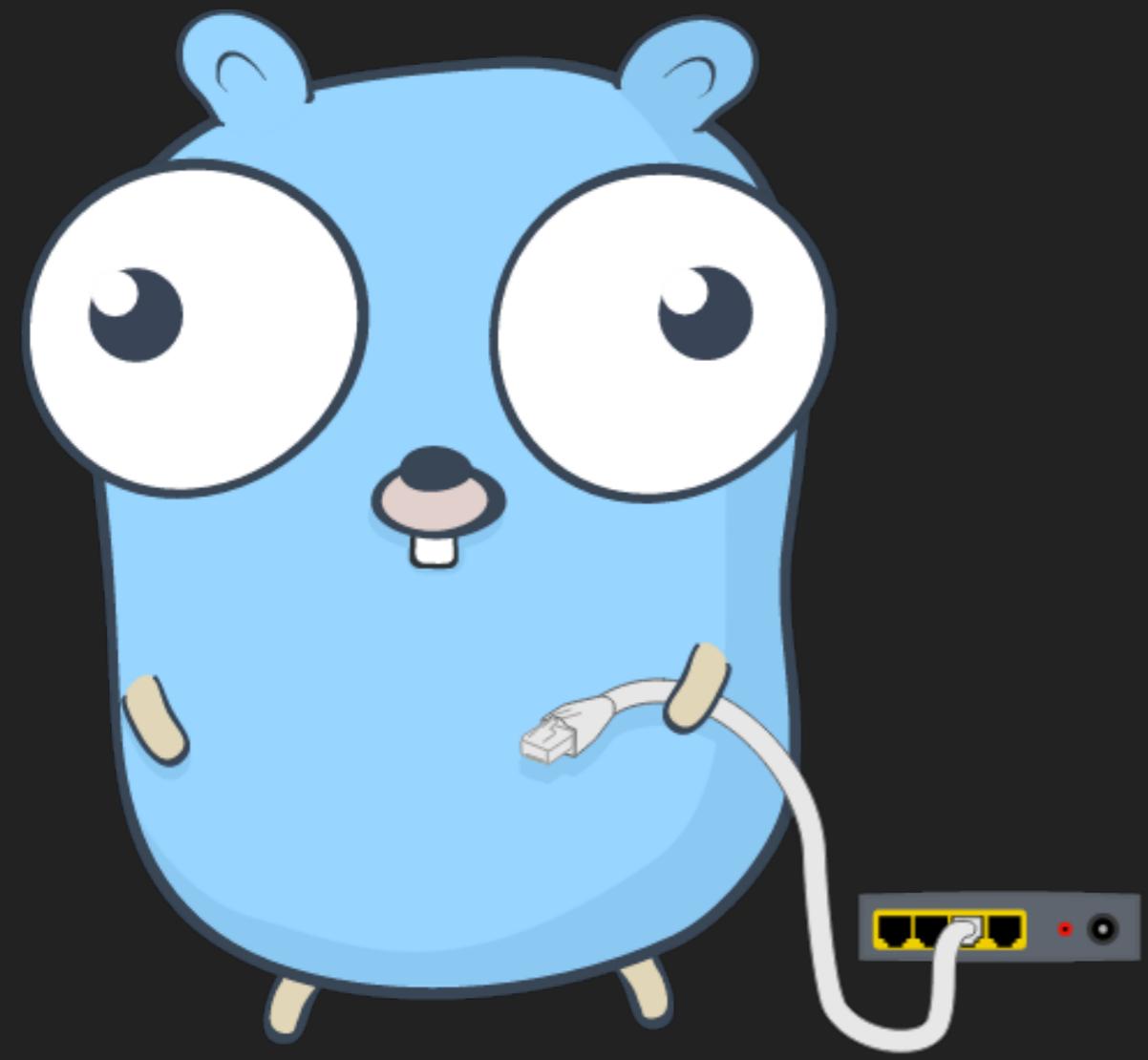
- ▶ Example with a return value:

```
var addition = func(x, y int) int{
    return x + y
}

z := addition(2, 2)
fmt.Println(z)
```

# EXERCISE

- ▶ Create an anonymous function inside main that:
  - ▶ Takes in variadic arguments of type **string**.
  - ▶ Returns a **string** that is the concatenation of the arguments with a space in-between them.
  - ▶ Call the anonymous function with strings: “cat”, “dog”, “bear”.
  - ▶ Print the return value.
- ▶ Answer at: <https://play.golang.org/p/mrCzIY0nrG>



## DEFER/PANIC/ RECOVER

---

IF MY CODE PANICS,  
EVERYONE ELSE'S PANICS.  
-KOBE BRYANT

## DEFER

- ▶ The keyword **defer**, followed by a function call will delay the call until the end of the containing function call.
- ▶ If there is more than one **defer**, the last **defer** statement goes first.

```
func printing() {  
    defer fmt.Println("exiting")  
    defer fmt.Println("I must be going")  
  
    fmt.Println("Hello,")  
}
```

Outputs  
→

Hello,  
I must be going  
exiting

- ▶ Try it out at: <https://play.golang.org/p/DCyvcVxdFo>

## SECTION 15

---

# DEFER

- **Defer** is useful to have a statement called once instead of at each exit point from the function.
- This is often used to unlock mutexes or close files.
- Here's an example using **defer** and without **defer** for closing a file.

```
// filePrinter prints a file to the screen if it doesn't
// contain TOP SECRET in the file. This is an example,
// its is not the most efficient way to do this.
func filePrinter(filePath string) error {
    f, err := os.Open(filePath)
    if err != nil {
        return err
    }
    defer f.Close()

    r := bufio.NewReader(f)
    output := ""
    for {
        out, err := r.ReadString('\n')
        if strings.Contains(out, "TOP SECRET") {
            return fmt.Errorf("Top Secret file")
        }
        output += out
        if err != nil {
            break
        }
    }
    fmt.Println(output)
    return nil
}
```

```
func filePrinter(filePath string) error {
    f, err := os.Open(filePath)
    if err != nil {
        return err
    }

    r := bufio.NewReader(f)
    output := ""
    for {
        out, err := r.ReadString('\n')
        if strings.Contains(out, "TOP SECRET") {
            f.Close() // Now I have to close it here.
            return fmt.Errorf("Top Secret file")
        }
        output += out
        if err != nil {
            break
        }
    }
    fmt.Println(output)
    f.Close() // And I have to here as well.
    return nil
}
```

# EXERCISE

- Write a function called Sum that prints a sum of all numbers 1 to 100.
- The function will also print to the screen how long the function took to execute.
- To make sure the time exceeds 1 second, we want to sleep execution for 1 second.
- To do this, inside the function:
  - Record the start time of the function call to a variable called `start` using `time.Now()`.
  - Uses `defer` to print out the total runtime of the function, using `time.Now().Sub(start)`
  - Sums the numbers 1 to 100 via a loop
  - Call `time.Sleep(1 * time.Second)` to pause execution for 1 second.
  - Prints the sum.
- The way you will try to make it work: [https://play.golang.org/p/oBJtW1I\\_RnV](https://play.golang.org/p/oBJtW1I_RnV)
- Why you have a gotcha: <https://play.golang.org/p/ASTiy0SHV8V>
- Answer at: <https://play.golang.org/p/9EQNhayePr0>

# PANIC

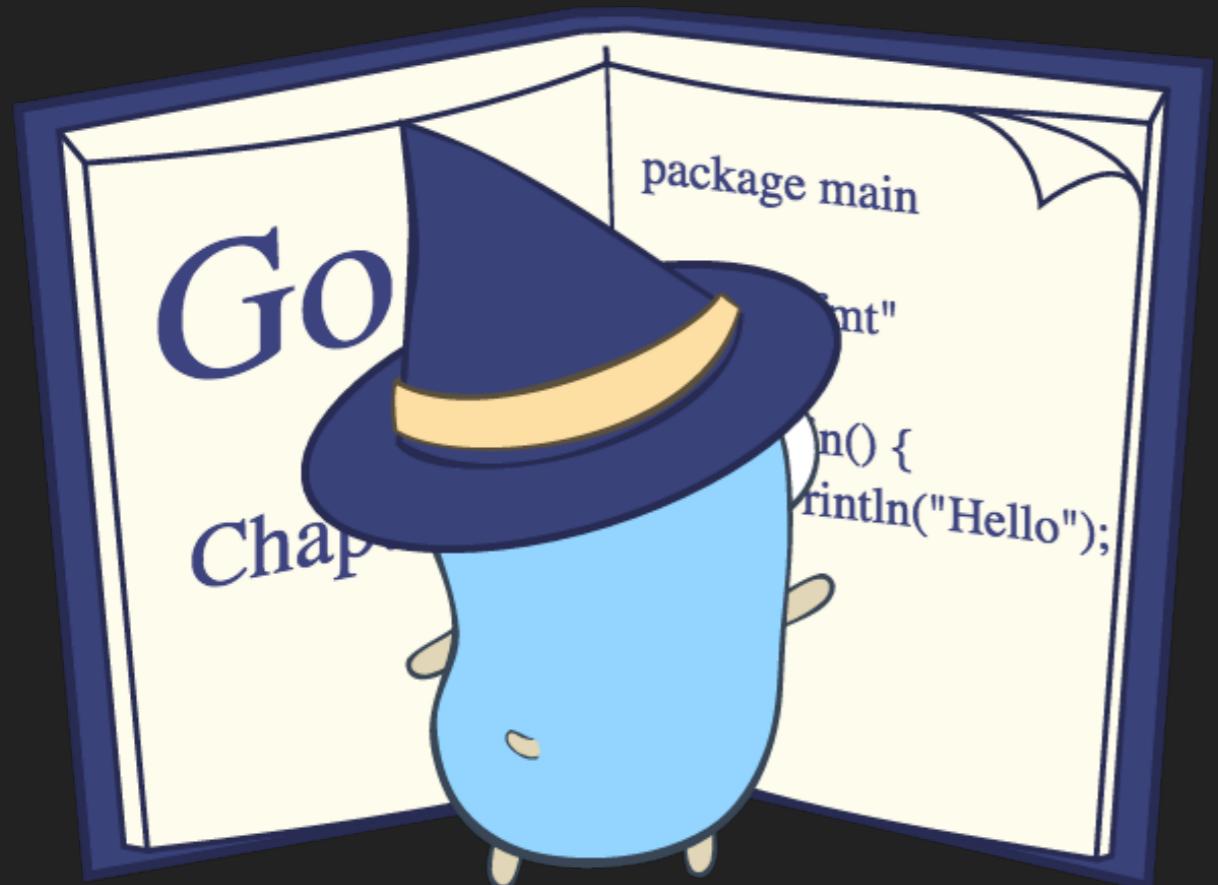
- **panic** simply causes your program to crash with a stack trace.
- Use only if the error is unrecoverable.
- Some general rules are:
  - Do this only on server side apps. Users of client applications have a bad experience when receiving stack traces.
  - Do this only in package **main**. Do not do this in other packages.

```
func main() {
    if !secureConnection() {
        panic("Anyone could be listening!")
    }
}
```

# RECOVER

- ▶ The **recover** function can be used to **recover** from a **panic** in the stack.
- ▶ In most circumstances, if you’re doing **recover**, something is wrong.
- ▶ **recover** is paired with a **defer**, here’s an example:

```
func main() {  
    defer func() {  
        if r := recover(); r != nil {  
            fmt.Printf("Recovered from a panic, panic was: %q\n", r)  
        }  
    }()  
    panic("hello")  
}
```



INTERFACES

---

INTERFACES? WELCOME  
TO THE JUNGLE!

- AXL ROSE

# INTERFACES

- ▶ An **interface** is a **type**, just like an **int**, **string**, etc...
- ▶ An **interface** requires that data stored in a variable have the following characteristics:
  - ▶ Must have the same methods as defined on the **interface**
  - ▶ Must have the same arguments for each method
  - ▶ Must have the same return values for each method
- ▶ Example:

```
var sample interface{Add(x, y int) int}
```

- ▶ This creates a variable, **sample**, of type **interface{Add(x, y int) int}**
- ▶ Only data that has a method called **Add** which takes two **int** arguments and returns an **int** can be stored here.

## SECTION 16

---

# INTERFACES

```
var sample interface{Add(x, y int) int}

type notRight struct{}

func (n notRight)Hello() {
    fmt.Println("hello")
}

// The following line won't work as notRight does not have the right method
// sample = notRight{}


type math struct {}

func (m math) Add(x, y int) int {
    return x+y
}

func (m math) Subtract(x, y int) int {
    return x-y
}

// This will work, because it does have the right method.
sample = math{}


// I can call this, because it is defined on the interface.
fmt.Println(sample.Add(3, 2))

// This will not work, because the interface does not define this method.
// fmt.Println(sample.Subtract(3, 2))
```

# INTERFACES

- ▶ It is more common to have a custom **interface** type than to define it inline, as we did in the previous slide.
- ▶ An **interface** with no value assigned has a Zero Value of **nil**.

```
type Adder interface {  
    Add(x, y int) int  
}  
  
var sample Adder
```

# INTERFACES

```
type MathStuff interface {
    Add(x, y int) int
    Multiply(x, y int) int
}
```

```
type MyMath1 struct {}

func (m MyMath1) Add(x, y int) int {
    return x+y
}
```

```
func (m MyMath1) Multiply(x, y int) int {
    return x * y
}
```

```
var m MathStuff = MyMath1{}
```

```
type MyMath2 struct {}

func (m MyMath2) Add(x, y int) int {
    return x+y
}
```

X var m MathStuff = MyMath2{}

MyMath2 is missing Multiply()

# INTERFACES

- So, why would you want an **interface**?
  - **Interfaces** can provide a simple contract that can be satisfied by multiple implementations.
    - File systems are a good example. Functions often want to access a file system and perform operations. You don't want to write the same function for each file system type. By providing an **interface** for file systems, you can implement support for the local file system, remote file systems (SFTP), specialized file systems (Hadoop HDFS), etc. Your function simply manipulates an interface representing a file system and becomes agnostic to what file system it is.
    - Abstracted storage is another. Often it is required to read and write records to some type of storage. Interfaces can often be useful to abstract these into a storage interface with different implementations. Customers can then store data in MySQL, in memory, or in various disk storage schemes. Your software never changes, you just plug in the new implementation.
    - Tests are another example. You may not want a test to make an expensive call or talk to an external system. By using an **interface**, you can use a fake implementation for tests and use the real implementation when in production.

# INTERFACES

- ▶ Interesting note:
  - ▶ The `error` type is simply an `interface`.
  - ▶ You can satisfy that type with your own custom type.

```
// This is Go's built in error type.  
type error interface {  
    Error() string  
}
```

```
type CustomError int  
  
func (c CustomError) Error() string {  
    return fmt.Sprintf("had a CustomError with error code %d", c)  
}  
  
func blah() error {  
    return CustomError(1)  
}
```

- ▶ The custom type above is built off the `int` type.
- ▶ Yes, a custom `int` can have a method! Its not just for the struct type.
- ▶ This one returns an error code and can store that error code for other uses.

## SECTION 16

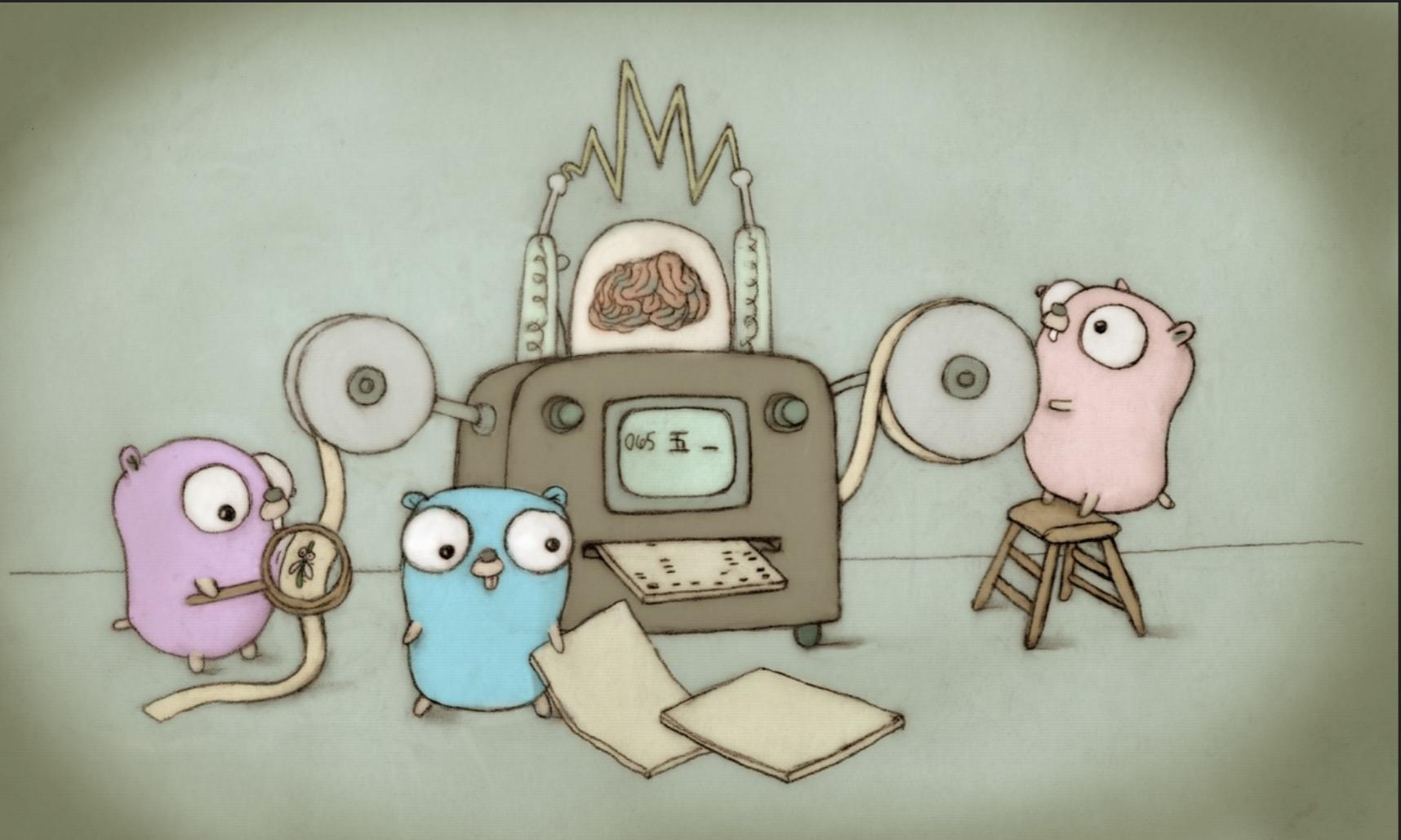
---

# EXERCISE

- Create an interface type called `FileSystem`
  - Has method `Read(f string) ([]byte, error)`
  - Has method `Write(f string, b []byte) error`
- Create a struct called `inmemory`
  - Has attribute `map[string][]byte`
  - Implement `FileSystem` reading and writing from `map`.
- Create a function called `WriteReadme(fs Filesystem, pathStr string) error`
  - This should write a README file to path/README
  - Can contain anything you want. You can do `[]byte("README")` to convert a string to `[]byte`.
  - Use the "path/filepath" package to join `pathStr` and "README".
- In `main()`
  - Create a var called `fs` of type `FileSystem`
  - Call `WriteReadme(fs, "/some/path/")`
  - Print out `fs.Read("/some/path/README")`
- Answer at: [https://play.golang.org/p/r-fQuIWHD\\_b](https://play.golang.org/p/r-fQuIWHD_b)

## EXERCISE NOTES

- ▶ In general, you would not create your own `FileSystem` interface. This example is to make the exercise simple.
- ▶ The `io` package contains interfaces such as `Reader`, `Writer`, `ReadWriter`, ... that are normally implemented for file systems.
- ▶ These filesystems might be local disk (accessed via `os.OpenFile()` or `os.Open()`), or HTTP requests from HTTP clients. Read/Write streams for file systems are almost always implemented using interfaces in the `io` package.
- ▶ If you want to simply open a file and read it into memory (which is great when your files are small), use the `ioutil` package.



## GO ROUTINES

---

**WAKE ME UP BEFORE  
YOU GO GO - WHAM!**

# GOROUTINES

- ▶ A **goroutine** is Go's method of starting a concurrent function
- ▶ A **goroutine** is similar to the idea of a thread, but:
  - ▶ Much lighter weight. You can create millions of **goroutines** on a laptop.
  - ▶ **Goroutines** run on top of threads, scheduled by the runtime, not the OS.
  - ▶ If you come from a C/C++ background, you don't need to be as judicious with **goroutines**.
- ▶ Creating a **goroutine** is as simple as:
  - ▶ `go <function name>()`

# GOROUTINES

```
func main() {
    for i := 0; i < 10; i++ {
        go fmt.Println("hello world", i)
    }

    // We haven't talked about select yet. Here it is used to keep the program
    // from ending. An empty select blocks forever. Without this, we might
    // not see anything print, because then main() function might end before
    // any of the goroutines are able to print.
    // If you try this, everything will print, but the program will crash.
    // We will talk about better ways to handle this later.
    select{}
}
```

- ▶ This simply spins off 10 **goroutines** that will print “hello #”, where # is 0-9 in random order.
- ▶ When you use a **goroutine**, you no longer have control of sequencing, which is why 0-9 will print in random order.

## SYNCHRONIZATION

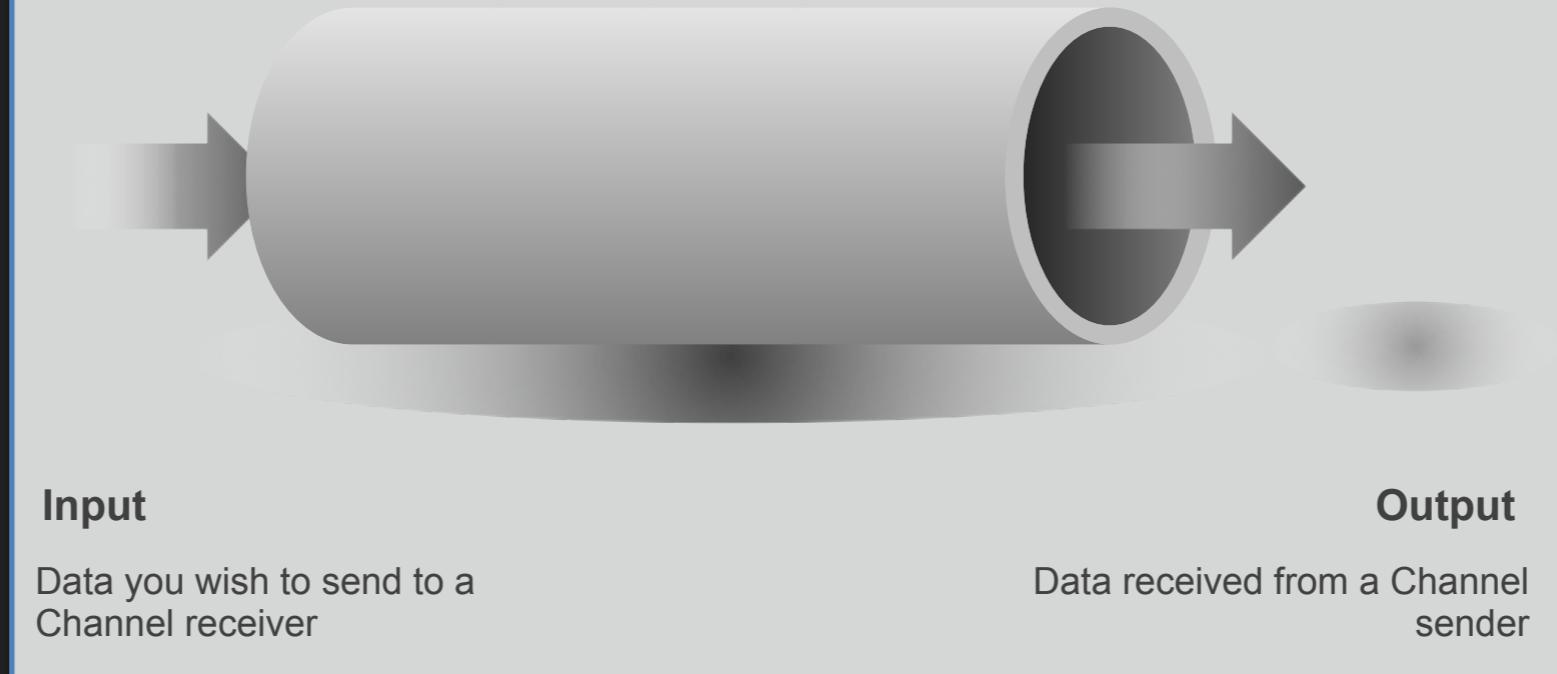
- ▶ Synchronization is the process of preventing read and write access to a variable in two **goroutines** at the same time.
- ▶ So, given a variable **x**:
  - ▶ Any number of **goroutines** can read the data in **x**
  - ▶ But if one of the **goroutines** must write to **x**, then you must use some form of synchronization.
  - ▶ It is undefined what will be in **x** if you do not use a synchronization method.
  - ▶ When two **goroutines** try to read and write to the same variable, this is called a data race. Go has a data race detector not covered here.
- ▶ To prevent data races, Go has **channels** and **Mutexes**.

## SECTION 17

# CHANNELS

### Channel

A synchronization primitive that allows sending data from senders to receivers. The amount that the Channel can hold is called the buffer.



- Channels are a synchronization primitive in Go, normally used to communicate between **goroutines**.
- Variables can be put into a channel and then pulled out in “first in, first out” order (FIFO).
- The number of variables that can be stored in a channel is called the buffer size.
- When doing an insert, if the channel is full execution blocks until room is made.
- When receiving a variable, execution will block until a variable can be received.
- **for/range** can be used with a channel. This allows pulling of data until the channel is closed.
- A closed channel will return the Zero Value of the type stored in the channel.

# CHANNELS

- Creates a channel “ch”

```
ch := make(chan string, 1)
```

CHANNEL TYPE

BUFFER SIZE

- Puts “hello” into the channel

```
ch <- "hello"
```

- Pulls the next variable from the channel into x

```
x := <-ch
```

- Pulls a variable from the channel and stores it in item until the channel is closed.

```
for item := range ch {  
    fmt.Println(item)  
}
```

- Closes a channel

```
close(ch)
```

# CHANNELS

```
// printer reads ints off of input and prints it to the screen. When the
// input channel closes and all data has been read off of input, the exit
// channel will be closed to signal that printer is done processing.
func printer(input chan int, exit chan bool) {
    defer close(exit)
    for i := range input {
        fmt.Println(i)
    }
}

func main() {
    input := make(chan int, 10)
    exit := make(chan bool)

    // Start our printer in a goroutine.
    go printer(input, exit)

    // Send 100 integers to our channel for printing.
    for i := 0; i < 100; i++ {
        input <- i
    }

    // Let the print know that we are no longer going to send data.
    close(input)

    // Wait for the printer to exit.
    <-exit
}
```

# EXERCISE

- ▶ Create a function called **doubler**
  - ▶ Takes in two arguments:
    - ▶ “**input**” of type **chan int**
    - ▶ “**output**” of type **chan int**
  - ▶ It should read off of channel **input**, multiply the **int** by 2 and write the new number to **output**.
  - ▶ It should close **output** when the function exits.
- ▶ In **main()**:
  - ▶ Create **input** and **output** channel of type **chan int** with a buffer of 10.
  - ▶ Use a **goroutine** to start a **doubler(input, output)**
  - ▶ Use a **goroutine** on an anonymous function to
    - ▶ Send the integers 1 to 100 to **doubler** via **input**
    - ▶ Close the **input** channel when done sending
  - ▶ Loop through the **output** channel and print the results.
- ▶ Answer at: <https://play.golang.org/p/48cVucXiB3>

## WAIT GROUPS

- A `WaitGroup` is a counter that can only represent positive values. It is a synchronization primitive, so it can be used by multiple `goroutines`.
- `WaitGroups` are used to count the number of operations that are outstanding. For example, starting 20 `goroutines` and knowing when they are all done.
- You can add to a `WaitGroup` by using the `.Add(n int)` method.
- You can remove from a `WaitGroup` by using the `.Done()`.
- If you call `.Done()` and the internal counter is 0, your program will panic.
- If you call `.Wait()`, your code will block until the `WaitGroup's` internal counter is 0.
- You cannot use an `int` for this, as this would cause a data race.

# WAIT GROUPS

```
func main() {
    wg := &sync.WaitGroup{}

    // Sping off 1000 goroutines, each one printing a number.
    for i := 0; i < 1000; i++ {
        wg.Add(1) // Add 1 to the counter. You MUST do this before the func().
        go func(i int) {
            defer wg.Done() // Remove 1 from the counter after the func() is finished.
            fmt.Println(i)
        }(i)
    }

    wg.Wait() // Wait until all of the goroutines are finished.
}
```

- This lets us spin off any number of **goroutines** and wait for them to finish.
- In normal use, we might continue doing other things and then use **.Wait()** when we need to stop to retrieve data or require all processing has completed before moving on.

# MUTEXES

- A **Mutex** provides a simple locking mechanism.
  - If a function calls `.Lock()`, then any other function calling `.Lock()` on the same **Mutex** blocks.
  - Calling `.Unlock()` allows another function grab the lock.
- This can be used to prevent multiple **goroutines** from accessing the same variable at the same time.

```
var counter = 0

func increment(wg *sync.WaitGroup, mu *sync.Mutex) {
    mu.Lock()
    defer mu.Unlock()
    defer wg.Done()
    counter++
}

func main() {
    wg := &sync.WaitGroup{}
    mu := &sync.Mutex{}

    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go increment(wg, mu)
    }

    wg.Wait()
    fmt.Println(counter)
}
```

# EXERCISE

- ▶ Create a struct called `Sum`
  - ▶ Give it an attribute called `sum` of type `int`
  - ▶ Give it an attribute called `mu` of type `sync.Mutex`
  - ▶ Give it a method called `Add(n int)` that adds to `n` to `sum`, protected by `mu`.
- ▶ In `main()`:
  - ▶ Create a variable called `s` of type `*Sum`
  - ▶ Create a variable called `wg` of type `*sync.WaitGroup`
  - ▶ Create a for loop that counts from `1` to `4`
    - ▶ In each loop, use a `goroutine` to an anonymous function that:
      - ▶ Calls `Add(i)`, where `i` is the number of the loop.
      - ▶ Defer's a `wg.Done()` call.
    - ▶ Use `wg.Wait()` to wait for the `goroutines` to finish.
    - ▶ Print out the `sum`, which should be `10`.
  - ▶ Answer at: <https://play.golang.org/p/yZjiEU1b5P>
  - ▶ Note: Removing the `mutex` locks will do nothing in the playground, but will in real programs.

# SELECT STATEMENT

- **select** is similar to **switch**, but is used to deal with input from multiple channels.
- With **select**, you can listen to multiple channels at the same time and operate on the one that has data on it.
- **select** will randomly choose a case statement if multiple channels have data.

## SELECT STATEMENT

```
select {  
    case <- <channel var>:  
        <statements to execute>  
  
    case <var> := <- <channel var>:  
        <statements to execute>  
  
    default:  
        <statements to execute>  
}  
Take data off  
channel, but do  
nothing with it  
Take data off  
channel, assign it to  
<var>  
If no channel has  
data, this executes
```

# SELECT STATEMENT

```
// printer prints a number divided by 2 if it comes on the half channel.  
// It prints a number multiplied by 2 if it comes on the double channel.  
// It terminates if it receives anything on exit.  
func printer(half, double chan int, wg *sync.WaitGroup, exit chan bool) {  
    for {  
        select {  
            case i := <-half:  
                fmt.Printf("%d/2 = %d\n", i, i/2)  
            case i := <-double:  
                fmt.Printf("%d*2 = %d\n", i, i*2)  
            case <-exit:  
                fmt.Println("Quitting...")  
                return  
        }  
        wg.Done()  
    }  
}
```

```
func main() {  
    half := make(chan int, 1)  
    double := make(chan int, 1)  
    exit := make(chan bool)  
    wg := &sync.WaitGroup{}  
  
    for i := 1; i < 11; i++ {  
        i := i // Make i scoped inside the for loop.  
        wg.Add(1)  
        go func(){  
            half <- i  
        }()  
  
        wg.Add(1)  
        go func() {  
            double <- i  
        }()  
    }  
    // This just prevents everything from being in order. Ignore.  
    time.Sleep(2 * time.Second)  
  
    go printer(half, double, wg, exit)  
  
    wg.Wait()  
    close(exit)  
  
    // Sleep enough time to let "Quitting" print.  
    time.Sleep (2 * time.Second)  
}
```

- Try it at: <https://play.golang.org/p/GDzGHxiEKn>

# SELECT STATEMENT

- › Sometimes you want to only wait for a certain amount of time for something to happen on a channel.
- › The time library provides a nice method to handle this.

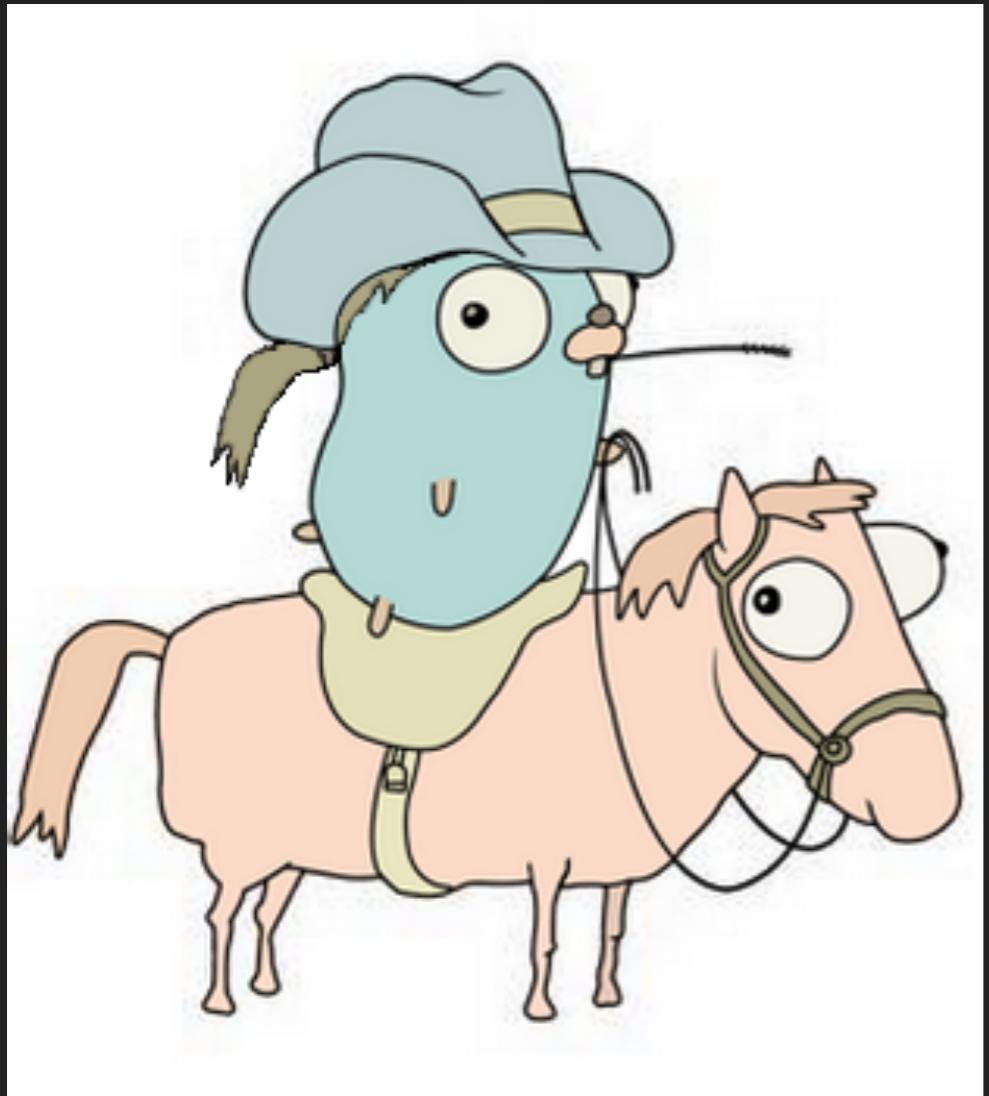
```
select {  
    case <- <channel var>:  
        <statements to execute>  
    case <-time.After(10 * time.Second):  
        <statements to execute>  
}
```



If something doesn't happen after 10 seconds, this executes

# EXERCISE

- ▶ Write a function call `printer(in chan string) (done chan bool)`
  - ▶ Use `make()` to initialize the `done` channel.
  - ▶ Spin off a `goroutine` that:
    - ▶ defers closing the `done` channel.
    - ▶ Takes input off the `in` channel and prints it.
    - ▶ If `printer` does not receive input in 5 seconds:
      - ▶ The function should print “quitting...”
      - ▶ Return.
  - ▶ Return the `done` channel. Remember, the `goroutine` will keep running.
- ▶ In `main()`:
  - ▶ Create a channel called `in` of type `chan string` with a buffer of 1.
  - ▶ Call `done := printer(in)`
  - ▶ Send “hello” to channel `in`.
  - ▶ Send “world” to channel `in`.
  - ▶ Wait for the printer to die after 5 seconds by doing: `<-done`
- ▶ Answer at: <https://play.golang.org/p/-80tpVoYlo>



## CONSTANTS

---

CONSTANTS ARE PROOF THAT  
GOD LOVES US AND WANTS US TO  
BE HAPPY.  
- BENJAMIN FRANKLIN

## CONSTANTS

- ▶ Constants store values that are set at compile time.
- ▶ Constants can represent **boolean**, **rune**, **integer** types, **floating-point** types, **complex** and **string**.
- ▶ Constants are declared using the **const** keyword.
  - ▶ `const str = "hello world"`
  - ▶ `const num = 3`
  - ▶ `const num64 int64 = 3`
- ▶ Constants can be typed or untyped. An untyped integer constant can be used with any integer variable. If typed, it can only be used with another of the same type.
- ▶ In many languages, it is common to see a constant name in all upper case. We do not do this in Go.

## ENUMERATION VIA CONSTANTS

- ▶ The keyword **iota** can be used to create enumerators.

```
// Enumeration using constants.  
const (  
    apple = iota // apple == 0  
    orange // orange == 1  
    pear // pear == 2  
)
```



## BLANK INTERFACE

---

AN INTERFACE DOES NOT BECOME CONCRETE THROUGH MAGIC. IT TAKES SWEAT, DETERMINATION AND TYPE ASSERTION  
- COLIN POWELL

## BLANK INTERFACE

- ▶ A blank **interface**, designated as **interface{}**, can hold any value
- ▶ This allows Go to accept multiple different types as a single argument
- ▶ **fmt.Println()** is an example, it has the following signature:

```
func Println(a ...interface{}) (n int, err error)
```

# BLANK INTERFACE

- ▶ However a blank `interface{}` has no methods and does not expose its value
- ▶ The `interface` must be converted into its concrete type or an `interface` type it satisfies via “type assertion”
- ▶ A type assertion can be done either via a type assertion statement or using a `switch` conditional

## SECTION 19

---

# EXAMPLE

```
var i interface{}  
    // i = "hello world"  
    i = 32  
    // i = 0.0  
  
    switch v := i.(type) {  
    case string:  
        fmt.Printf("%q was a string value\n", v)  
    case int:  
        fmt.Printf("%d was an int value\n", v)  
    default:  
        fmt.Printf("was not a supported type, was %T", v)  
    }  
  
    if x, ok := i.(int); ok {  
        fmt.Printf("the result of %d + 10 = %d\n", x, x+10)  
    }
```

Try it out:

<https://play.golang.org/p/dJAHcDBzjls>

# EXERCISE

- ▶ Write a function called **PrintMapSlice()**
  - ▶ Argument is an **interface{}**
  - ▶ Returns an **error**
  - ▶ Prints the key/values of a **map[string]string**, one per line
  - ▶ Prints the index/values of a **[]string** slice, one per line
  - ▶ Prefixes the printed line with either “map: ” or “slice: ”
  - ▶ Returns an **error** if any type other than a **map[string]string** or **[]string** is passed.
  - ▶ Pass a **map[string]string**, **[]string**, and any one other type to **PrintMapSlice()**
  - ▶ Answer at: <https://play.golang.org/p/raqrFAghwsQ>



SAVE THIS EXERCISE  
FOR USE LATER



## EMBEDDING/ COMPOSITION

---

AVOID INHERITANCE;  
IT HAS MANY SNARES AND NO  
REAL BENEFIT  
- WILLIAM PEN

# EMBEDDING

- ▶ Go does not have the notion of subclassing that many Object Oriented languages provide
- ▶ Go does provide the ability to “borrow” from another type via embedding one **struct/interface** within another **struct/interface**

Embedded Mutex

```
type Records struct {
    recs []Record
    nextID int
    sync.Mutex // Embedded Mutex
}
```

```
recs.Lock()
recs.Add(Record{})
recs.Unlock()
```

Non-Embedded Mutex

```
type RecordsAlt struct {
    recs []Record
    nextID int
    mu sync.Mutex
}
```

```
recs.mu.Lock()
recs.Add(Record{})
recs.mu.Unlock()
```

Embedding To Satisfy An Interface

```
type LockRec interface {
    Lock()
    Unlock()
    Add(Record)
}
```

```
var recs LockRec
recs = Records{}
recs = RecordsAlt{}
```

# EMBEDDING

- ▶ This differs from subclassing in other languages in the following way:
  - ▶ While an embedded method is available on the outer object, when invoked the receiver is still the inner object, not the parent

## EMBEDDING

- ▶ Initializing an embedded type in a `struct` is similar to other composite literal `struct` initializations

```
type Records struct {  
    recs    []Record  
    nextID int  
    sync.Mutex // Embedded Mutex  
}
```

```
// Allows assignment via named field for embedded type.  
mu := sync.Mutex{}  
recs := &Records{nextID: 1, Mutex: mu}  
  
// Uses field order for naked assignment.  
recs := &Records{[]Record{}, 0, sync.Mutex{}}  
  
// Just uses the zero values.  
recs := &Records{}
```

# EMBEDDING

- › Accessing fields within an embedded type with the same name a field in the parent is still possible
- › Outer method/field names have precedence over inner names

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

// Printf is overriding our embedded log.Logger.Printf().
func (job *Job) Printf(format string, args ...interface{}) {
    // We can still access it by using the type name as the field name.
    job.Logger.Printf("%q: %s", job.Command, fmt.Sprintf(format, args...))
}

func New(cmd string) *Job {
    return &Job{Command: cmd, Logger: log.New(os.Stderr, "Job: ", log.Ldate)}
}

func main() {
    j := New("ls -al /home/johndoak")
    j.Println("hello") // Uses the embedded println()
    j.Printf("hello") // Uses the outer defined Printf(), not Logger.Printf()
    j.Logger.Printf("hello") // We can still access Logger.Printf() if required
}
```

Try it at:

<https://play.golang.org/p/9LjnJBEB6LU>

## SECTION 20

---

# EXAMPLE

```
type Records struct {
    recs      []Record
    nextID   int
    sync.Mutex // Embedded Mutex
}

func (r *Records) Add(rec Record) {
    rec.ID = r.nextID
    r.nextID++
    r.recs = append(r.recs, rec)
}

func (r *Records) Print() {
    for _, rec := range r.recs {
        fmt.Println(rec)
    }
}

func main() {
    recs := &Records{}

    // No need for locking.
    for i := 0; i < 1000; i++ {
        recs.Add(Record{})
    }

    // Need locking.
    wg := sync.WaitGroup{}
    wg.Add(1000)
    for i := 0; i < 1000; i++ {
        go func() {
            defer wg.Done()
            recs.Lock()
            defer recs.Unlock()
            recs.Add(Record{})
        }()
    }
    wg.Wait()
    recs.Print()
}
```

Try it out:

<https://play.golang.org/p/hfB5qp204Kw>

# COMPOSITION

- ▶ Interfaces like structs can embed other interfaces
- ▶ This allows for composite interfaces, that is an interface made of the union of multiple interfaces
- ▶ Only an interface can be embedded within an interface

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

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

// ReadWriter is the composition of the Reader and Writer interfaces.
type ReadWriter interface {
    Reader
    Writer
}
```

## SECTION 20

---

# EXERCISE

- Create an **interface** called Reader
  - Has a single method called **Read()**
  - Returns a `[]byte` and an `error`
  - Should return all bytes in an internal buffer and clear the buffer
- Create an **interface** called Writer
  - Has a single method called **Write()**
  - Takes a single argument of type `[]byte`
  - Returns a single type, `error`
  - Appends to an internal buffer
- Create an **interface** called Locker
  - Has a method called **Lock()**
  - Had a method called **Unlock()**
- Create a composite **interface** called ReadWrite
  - Is a composite of Reader and Writer
- Create a composite **interface** called ReadWriteLock
  - Is a composite of ReadWrite and Locker
- Write an implementation of that can satisfy each off these:
  - Should embed a `sync.Mutex` or `*sync.Mutex`
  - Use the `bytes.Buffer` type internally
- Assign the implementation to a ReadWriteLock and try using it
- Answer at: <https://play.golang.org/p/KS1oRNbJlmo>



## WRITING TESTS

---

TESTS ARE PROOF THAT  
HELL DOES EXIST  
- EVERY DEVELOPER

# TESTING PACKAGES

- ▶ Tests are located in `*_test.go` files
- ▶ Tests are normally part of the package they want to test. So if you want to test package ioutil, the `ioutil_test.go` file will have a package declaration of "ioutil". This allows a test to test private functions/methods
- ▶ Each test is a function that starts with the name `Test`. So a test that tested a function called `Add()` might be called `TestAdd()`
- ▶ Each test function has the same signature:
  - ▶ `func Test(t *testing.T) {}`
- ▶ A test passes if `t.Errorf()` or `t.Fatalf()` is not called and the test does not time out
- ▶ `t.Errorf()` will cause the test to fail, but will not stop the test from running
- ▶ `t.Fatalf()` will cause the test to fail and exits that test function immediately

## DO NOT USE MOCK LIBRARIES, USE FAKES

- ▶ Mocks are defined to be automatically generated code that is used to record expected method calls and return controlled output
- ▶ Mocks are generally just bad in any language
  - ▶ Usually brittle (change call order during refactor or a multitude of other factors)
  - ▶ Use of reflection can fail in unexpected and hard to understand ways
  - ▶ Packages like gomock have been deprecated at the places that they were developed at

# DO NOT USE MOCK LIBRARIES, USE INTERFACES

- If you need to test interactions with external systems, hide them behind private interfaces
- Create fakes that only include the methods that you use, not all the methods of the concrete type
- Provide fake implementations that simply test your function by either returning an error or test output for the function to process
- If providing a fake for a 3rd party interface, use composition to protect your fake by including the existing interface in your fake as an anonymous attribute
- Storage systems, the most common complex fake, should be abstracted behind an interface
  - Allows simple file implementations or in-memory implementations to be used in tests
  - Testing concrete implementations almost always requires either:
    - Using containers to setup a test environment to test the concrete methods
    - Test instances with the cloud provider that can be brought up and down via scripts (Azure CosmosDB for example)
  - Tests that do not interact with external systems are “**hermetic**”. All tests not testing a concrete implementation should be hermetic. Test the bare essentials in non-hermetic tests, they are expensive!
- RPC implementations are easily faked, but this is beyond the scope of this tutorial

## GO USES TABLE DRIVEN TESTS

- ▶ The most common test methodology used in Go is the “table driven test”
- ▶ A “table” is defined as a list of structs
- ▶ Each struct defines the test description, the test input to the function and the expected result
- ▶ The test loops through each entry in the table passing the input and retrieving the output. The test compares the received output to the expected output
- ▶ This makes for readable tests that separate the test data from the test method

## SECTION 21

---

# EXAMPLE

```
func Divide(n int, d int) (float64, error) {
    if d == 0 {
        return 0.0, fmt.Errorf("cannot divide by 0")
    }
    return float64(n)/float64(d), nil
}

func TestDivide(t *testing.T) {
    tests := []struct{
        desc string
        n int
        d int
        err bool
        want float64
    }{
        {"divide by 0 error", 10, 0, true, 0.0},
        {"success", 10, 3, false, float64(10)/float64(3)},
    }

    for _, test := range tests {
        got, err := Divide(test.n, test.d)
        switch {
        case err == nil && test.err:
            t.Errorf("TestDivide(%s): got err == nil, want err != nil", test.desc)
            continue
        case err != nil && !test.err:
            t.Errorf("TestDivide(%s): got err == %s, want err == nil", test.desc, err)
            continue
        case err != nil:
            continue
        }
        if got != test.want {
            t.Errorf("TestDivide(%s): got %v, want %v", test.desc, got, test.want)
        }
    }
}
```

Try it out:

<https://play.golang.org/p/VQ9ggHVvXsQ>

## COMPLEX EXAMPLE

- ▶ Has `Storage` interface to define access to storage (could be a database, file system, or in-memory representation)
- ▶ A concrete `fakeStorage` for use in tests
- ▶ A `Registrar` type that accesses `Records` in `Storage`, supporting concurrent access
- ▶ Tests the `Read()` method via a Table Driven Test

See it here:

<https://play.golang.org/p/tAk8FqPDtgq>

## EXERCISE

- ▶ Load the exercise from section 20, the **PrintMapSlice** exercise
- ▶ Write a table driven test that tests various inputs to expected outputs
- ▶ You will need to change to using `fmt.Fprintln()/Fprintf()` instead of `fmt.Println()/Printf()`
  - ▶ You may use a `*bytes.Buffer` for the `io.Writer` type
  - ▶ Use a private global to store the `*bytes.Buffer` as an `io.Writer` type
- ▶ Answer at: <https://play.golang.org/p/pvsmGibaGYQ>



# GENERICS

---

## EVERYTHING AND NOTHING ARE THE SAME IN THE ABSOLUTE

# WHAT ARE GENERICS

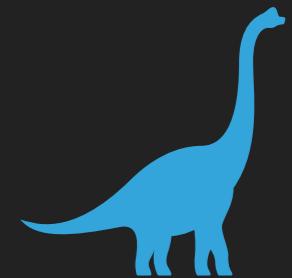
- ▶ Generics allow functions to be written that can handle multiple types without using the `interface{}` or `any` type.
- ▶ This allows generics to compile to a type specific version during compile time, vs runtime type assertions using `interface{}`
- ▶ This provides compiler type protections that runtime type assertions lack and a general increase in speed

## SECTION 22

# SIMPLE EXAMPLE

Represents the generic type

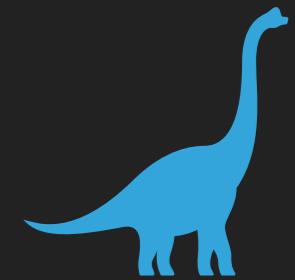
The types "T" can stand for, called a **constraint**



```
func SumInts[T int8 | int16 | int32 | int64 | int](ints ...T) T {  
    var s T  
    for _, n := range ints {  
        s += n  
    }  
    return s  
}
```

- ▶ This will return the sum of any integer type provided
- ▶ This will **NOT** work for custom types based on these integer types

# EXAMPLE THAT CAN HANDLE CUSTOM TYPES

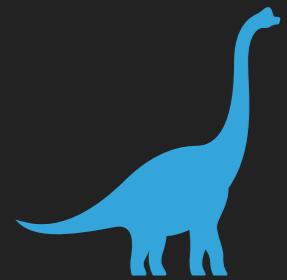


“~” allows for custom types based on these types

```
func SumInts[T ~int8 | ~int16 | ~int32 | ~int64 | ~int](ints ...T) T {  
    var s T  
    for _, n := range ints {  
        s += n  
    }  
    return s  
}
```

- ▶ A variable of “type MyIntType int8” could be used here

# DEFINE A CUSTOM GENERIC CONSTRAINT

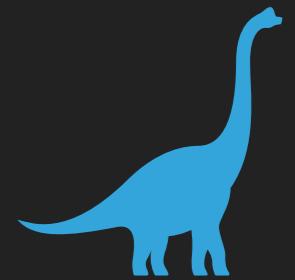


```
type Numbers interface {
    ~int8      | ~int16 | ~int32 | ~int64 | ~int |
    ~uint8     | ~uint16 | ~uint32 | ~uint64 | ~uint |
    ~float32   | ~float64
}
```

```
func SumNumbers[T Numbers](numbers ...T) T {
    var s T
    for _, n := range numbers {
        s += n
    }
    return s
}
```

- ▶ This supports all number types and custom types based on these types
- ▶ While using similar semantics to standard interfaces, you cannot use **Numbers** as an **interface** type, only as a generic **constraint**

# USING MULTIPLE GENERIC TYPES



```
type Numbers interface {
    ~int8      | ~int16 | ~int32 | ~int64 | ~int |
    ~uint8     | ~uint16 | ~uint32 | ~uint64 | ~uint |
    ~float32   | ~float64
}

func SumNumbers2[T Numbers, R Numbers](numbers ...T) R {
    var s R
    for _, n := range numbers {
        s += R(n)
    }
    return s
}
```

- ▶ This allows us to take in numbers of any type but return the sum as a different type
- ▶ “T” represents the types we accept as arguments
- ▶ “R” represents the types we will return
- ▶ You can then sum a bunch of `int8` types, but return the sum as an `int64`

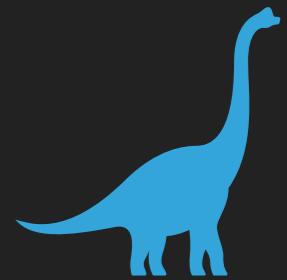
# THE CONSTRAINTS PACKAGE

- ▶ Package: <https://pkg.go.dev/golang.org/x/exp/constraints>
- ▶ Contains generic constraints for:
  - ▶ Complex numbers
  - ▶ Floats
  - ▶ Integers
  - ▶ Ordered numbers
  - ▶ Signed numbers
  - ▶ Unsigned numbers

# EXERCISE

- ▶ Using the constraints package
- ▶ Write a generic function that will take any integer or float type and return a sum with the same type
- ▶ Answer here: <https://go.dev/play/p/6Xhau5dLuc4>

# THINGS THAT DON'T WORK



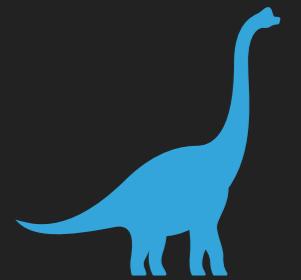
```
type MyStruct struct {
    AField int
}

type MyTypes interface {
    ~int8 | MyStruct
}

func SumMyTypes[T MyTypes](types ...T) T {
    var s T
    for _, n := range types {
        s += n
    }
    return s
}
```

- ▶ This won't work, as **MyStruct** doesn't support the **+** operator
- ▶ You can only do actions on types within a **constraint** that are common to all types

# CONSTRAINTS CAN INCLUDE METHODS



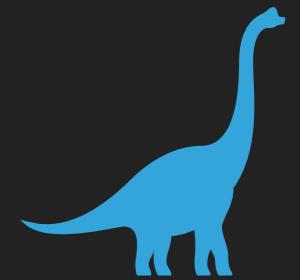
```
type MyConstraint interface {
    ~int8 | ~int16

    DoSomething()
}

func DoThings[T MyConstraint](types ...T) {
    for _, a := range types {
        a.DoSomething()
    }
}
```

- ▶ This will work on any custom type based on `int8` or `int16` that has `DoSomething()` defined

# CALLING GENERIC FUNCTIONS



```
sum := SumNumbers[int8](1, 2, 3, 4, 5, 6)
```

This will be an int8

The type that will stand in for "T"

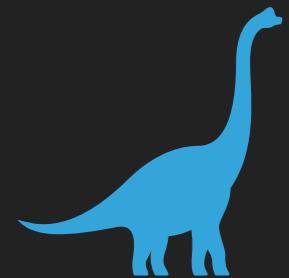
```
sum := SumNumbers2[int8, int64](128, 255, 32, 64, 4)
```

This will be an int64

The type that will stand in for "T"

The type that will stand in for "R"

# CALLING GENERIC FUNCTIONS WITH INFERENCE



```
sum := SumNumbers(int8(1), int8(2))
```

Brackets are not required

Can infer "T" is an `int8` from the arguments

```
sum := SumNumbers(1, 2)
```

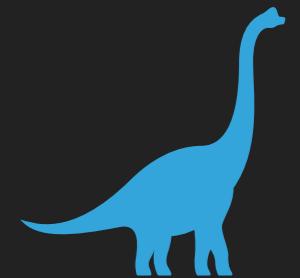
Will work, but will assume the `int` type

```
sum := SumNumbers2(int8(200), int8(100))
```

Result "R" cannot be inferred

Arguments "T"  
can be inferred

# GENERIC TYPES ON STRUCTS: SIMPLE CASE



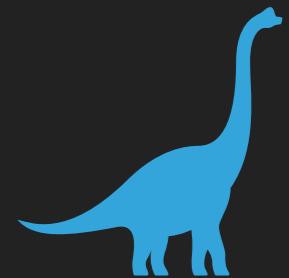
"T" represents any data type

```
// StreamData represents data that will be streamed on a channel.  
type StreamData[T any] struct {  
    // Data is the data in the stream  
    Data T  
    // Error is an error in the stream. If set, the stream ends.  
    Error error  
}
```

# EXERCISE

- ▶ Using the **StreamData** in the previous slide:
  - ▶ Use in a **goroutine** to return movies names that you enjoy
  - ▶ Use in a **goroutine** to return a range of numbers
- ▶ Answer here: <https://go.dev/play/p/DvKMQoLmiGM>

# GENERIC TYPES ON STRUCTS (COMPLEX)



```
type Writer[T string | []byte] struct {
    output io.Writer
}

func (w Writer[T]) Append(data T) error {
    switch t := any(data).(type) {
    case string:
        b := *(*[]byte)(unsafe.Pointer(&t))
        if _, err := w.output.Write(b); err != nil {
            return err
        }
    case []byte:
        if _, err := w.output.Write(t); err != nil {
            return err
        }
    }
    return nil
}
```

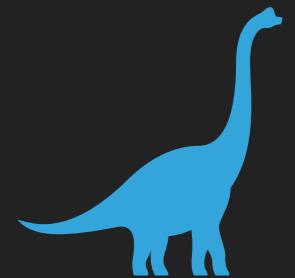
The type that will stand in for "T"

Indicates that this method is tied to Writer with generic type "T"

To use this in a type specific way requires introspection

This converts a **string** to a **[]byte** without an allocation

# USING A GENERIC STRUCT



Writer that handles the `string` type

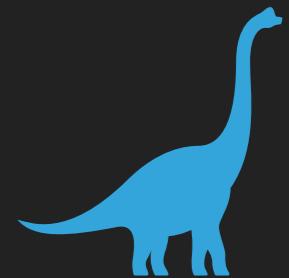
```
b := &bytes.Buffer{  
  
stringWriter := Writer[string]{output: b}  
if err := stringWriter.Append("Hello"); err != nil {  
    panic(err)  
}
```

Writer that handles the `[]byte` type

```
bytesWriter := Writer[[]byte]{output: b}  
if err := bytesWriter.Append([]byte(" world")); err != nil {  
    panic(err)  
}
```

- Try it at: <https://go.dev/play/p/MlpfdtCbMzc>

# CREATING A CONSTRUCTOR ON A GENERIC TYPE



Define the same **constraints** on the function

```
func New[T string | []byte](w io.Writer) Writer[T] {  
    return Writer[T]{output: w}  
}
```



Return the type with the **constraint**

- ▶ See it at: <https://go.dev/play/p/lss893npAkq>

# EXERCISE

- ▶ Using the `constraints.Ordered` type
- ▶ Write a generic type based on `[]T`
  - ▶ Has method `.Sort()` that will sort itself
- ▶ Answer here: <https://go.dev/play/p/b9iN29c5CLQ>

# EXERCISE

- ▶ Write the same thing, but as a generic function
- ▶ Which is better and why?

# WHEN NOT TO USE GENERICS

- ▶ If just calling a method
- ▶ When the implementation of a common method is different for each type
- ▶ When the operation is different for each type, even without a method

## BASIC GUIDELINE FOR USE

**IF THE ONLY DIFFERENCE IN A FUNCTION'S OPERATION IS THE TYPE, LOOK AT USING A TYPE PARAMETER**

# DON'T DO THIS

```
func Readfour [T io.Reader](r T) ([]byte, error) {
    buf := make([]byte, 4)
    _, err := io.ReadFull(r, buf)
    if err != nil {
        return nil, err
    }
    return buf, nil
}
```

THANKS FOR WATCHING

---

## AUTHOR: JOHN DOAK

- ▶ Currently: Principal Engineer @Microsoft
- ▶ Previously: Staff Site Reliability Engineer @Google
- ▶ In ancient time: Network Engineer @LucasFilm
- ▶ Writings on Go: <http://gophersre.com/>
- ▶ Nice Photos: <http://www.obsuredworld.com/>
- ▶ LinkedIn: [here](#)

