

Go Part 3 From pointers to Methods

Admin

- Questions?
- Read chapters 6-7 for this set.
- Let's look <https://medium.com/codefarm-java-ecosystem/engineering-interview-best-practices-what-actually-gets-you-hired-in-the-age-of-ai-087cf54ce037?sk=187dc65991d64dcc9544eac20ff87bb0>
- And for next time (or one week for undergrads) Please listen to
- Please listen to **episode 78 (Aug 22 2026)** of "the programming podcast"
 - <https://www.dthompsondev.com/podcast>

Go Projects and Dependencies

- **The original way that go did dependencies**
 - `go get <project git location>`
 - Then the project would be downloaded, compiled and ready for inclusion in any of your go projects as a library.
 - Eg
 - `go get -d -u gobot.io/x/gobot/...`
- **What could possibly go wrong?**

Go Projects and Dependencies

- **What could possibly go wrong?**
 - Student List?

Go Projects and Dependencies

- **What could possibly go wrong?**
 - Student List?
 - My list: at least:
 - Since often pulling latest version of project, someone can slip malicious code into a project
 - Especially in the era of LLMs and AI slop submissions.
 - Shared dependencies mean you can't easily pin a project to a version of a dependency
 - This one uses v1 of the web framework that one uses v2
 - Shared dependencies makes it hard to tell exactly what dependencies there are for multiple large projects.

Go Modules

- Introduced incrementally in go 1.11 and 1.12, and made default in 1.13
- Go modules uses a go.mod file to determine just the dependencies for this project
 - Can pin a version.
- Goland will help keep these dependencies in sync.
- Old way not really easy any more
 - But useful code lives a long time - discuss

The Plan

- **We want to get to interfaces,**
 - But can't really do that without understanding how Go does 'objects' and methods (and pointers)
 - So a few slides on objects and methods

Pointers

- Quickly remind me – who has had operating systems or something like it?

–

Pointers

- **Quickly remind me – who has had operating systems or something like it?**
 - What is a pointer? (in programming)
 -

Pointers

- **Quickly remind me – who has had operating systems or something like it?**
 - What is a pointer? (in programming)
 - A variable containing a memory address of a value we care about.
 - Example from book: (pretending these are the only two things in memory)
 - `var x int32 = 10`
 - `var y bool = true`

Value	0	0	0	10	1
Address	1	2	3	4	5
Variable	x				y

Pointers and their operators: * and &

- If you have a pointer and want the value it points to, use *
- If you have a value variable and want a pointer use &
- If you want a pointer variable to a type
 - use *type as the var type
- **Book page 120**

```
var x int32 = 10
var y bool = true
pointerX := &x
pointerY := &y
var pointerZ *string
```

Value	0	0	0	10	1	0	0	0	1	0	0	0	5	0	0	0	0
Address	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Variable	x				y	pointerX				pointerY				pointerZ			

Figure 6-2. Storing pointers in memory

Methods

- **How is a method different than a function?**
 - Lucky volunteer?

Methods

- **In Object Oriented Programming**
 - Methods are functions attached to objects
 - In python and java
 - Part of the class definition
 - In C++
 - Sort of – in the header file

Methods

- **In Object Oriented Programming**
 - Methods are functions attached to objects
 - In python and java
 - Part of the class definition
 - In C++
 - Sort of – in the header file
- **In Go**
 - Methods are called on types (usually structs)
 - But declared where ever (in same package)

An Example Struct to work with

- `type Sprite struct {`
- **Pic BitMap**
- **Location Point**
- `}`
- Let this be a simple 2D game struct
- Suppose it is in a game library and we are using it

Adding a method

- **Methods can be added to a type anywhere (in same package)**
 - Look a lot like functions except you call them on an 'object'
 - (but should declare all methods from same *interface* together)
- **Method:**
 - `func (<object called on>) <name>(<params>) <return type>{`
- **Example**
 - `func (spr *Sprite)move(dX, dY int){`
 - `//do stuff here`
 - `}`

Methods mutating vs non-mutating

- **What do I mean by mutating vs non-mutating methods?**
 - Term mostly from C++
 - Java mostly just has non-mutable objects rather than individual methods
 - (and python is wild wild west – you can add a new method or field at runtime)

Methods mutating vs non-mutating

- **What do I mean by mutating vs non-mutating methods?**
 - Methods that can't change the value of the object
 - Or changes are not seen by the calling function

Methods mutating vs non-mutating

- **Methods that don't mutate their objects**
 - const methods in C++ terms
 - Usually called on the object itself (objects also passed by value)
 - Eg:
 - `func (spr Sprite) GetSize()(width int, height int){`
 - `//fill in here`
 - `}`
 - `character Sprite := LoadCharacter()`
 - `h, w := character.GetSize()`

Methods mutating vs non-mutating

- If you want the method to mutate the object you call it on
 - Call it on a pointer to the object
 - Copy of pointer still points at same object.
 - From before:
 - `func (spr *Sprite)move(dX, dY int){`
 - `//do stuff here`
 - `}`
 - Call:
 - `character Sprite := <get result here>`
 - `(&character).move(30, 20)`

Syntactic Sugar

- **But 'programmers are lazy'**
 - And that was ugly
 - Sure there are ways to do it in multiple lines to make it slightly better
 - But go helps out
 - **Iff** you are calling the method on an object held in a variable, then go will take the address of the variable
 - So
 - `(&character).move(30, 20)`
 - Becomes just
 - `character.move(30,20)`
 - And the pointer is passed to move automatically

- **You can call a method on nil**
 - Calling on nil will not cause an error
 - Fixing the ‘billion dollar mistake’?
- **Still need to make sure your method doesn’t cause a panic if you do.**
 - Eg:
 - `var badGuy *Sprite //badGuy is nil`
 - `h, w, := badGuy.GetSize()`
 - Since nil is the zero value for sprites, h and w ***should*** be zero.

Questions?

- **So methods in go**
 - Kinda objective-c like
 - Neat: you can add a method to any struct in the package in any file in the package
 - Scary:
 - you can add a method to any struct in the package in *any file in the package*,
 - you better look through the whole folder
 - With your mark 1 eyeballs, grep, or LLMs (double check the last for now)
 - Keep your private data lower cased :-)
 - More on packages later in semester.

Interfaces

- What are interfaces in java?

Interfaces

- **What are interfaces in java?**
 - A way to promise some methods will be available

Interfaces in Go

- **Interfaces in go are also contracts that types will provide some functionality**
 - So far java-esque
 - Even the interface declaration looks right
 - From go by example:
 - `type geometry interface {`
 - `area() float64`
 - `perim() float64`
 - `}`

Interfaces in go are implicit

- **To implement an interface for a type T,**
 - Just write all the methods that are called on T that are required by the interface
 - Continuing the example
 - `type Rect struct {`
 - `Width, Height float64`
 - `}`
 - `type circle struct {`
 - `radius float64`
 - `}`

Continuing the gobyexample.com example

- **With these methods (in same package), both rect and circle implement the geometry interface**

```
- Don't need to explicitly say so like in java
- func (r rect) area() float64 {
-     return r.width * r.height
- }
- func (r rect) perim() float64 {
-     return 2*r.width + 2*r.height
- }
- func (c circle) area() float64 {
-     return math.Pi * c.radius * c.radius
- }
- func (c circle) perim() float64 {
-     return 2 * math.Pi * c.radius
- }
```

Interfaces can be variables

- `var shape geometry = rect{width: 5, height 4}`
- `a:= shape.area()`
- **Real power**
 - You make a function that takes an interface
 - Can pass anything that satisfies that interface.
 - Idiomatic Go rule-of-thumb: prefer to take interfaces as parameters and return structs
 - To help future proof your code

Interface{}

- **interface{} is empty interface**
 - Matches any type that implements at least zero methods
 - What does this buy us?

Interface{}

- **interface{} is empty interface**
 - Matches any type that implements at least zero methods
 - What does this buy us?
 - What does this buy us?
 - It matches anything
 - Make this a param type and your function can handle any value of any type
 - Use **very sparingly**!!!!!!!
 - In recent go versions **any** is a predefined alias for interface{}
 - And generics (coming later) adds to it.

Dynamic Dispatch

- **Lets talk dynamic dispatch**
 - Since we have some people with a non-cs background – what do I mean by “dynamic dispatch” (sometimes called late-binding)
 - How does it differ from static dispatch (sometimes called early binding)?

Dynamic Dispatch

- **Lets talk dynamic dispatch**
 - How does it work?
 - C++, when does it happen? When does it not?
 - How about Java?
 - (since python is interpreted not an issue here)

Dynamic Dispatch

- **Lets talk dynamic dispatch**
 - How does it work?
 - C++, when does it happen? When does it not?
- **In Go – interface methods use dynamic dispatch.**

Type Assertions

- **Type Assertions**
 - Used to check/cast interface type to
 - Actual concrete type
 - Another interface type
 - Interface var x and Type T
 - `x.(T)` will assert the type and return x as a T or panic
 - What if you just wanted to check?

Type Assertions

- **Type Assertions**

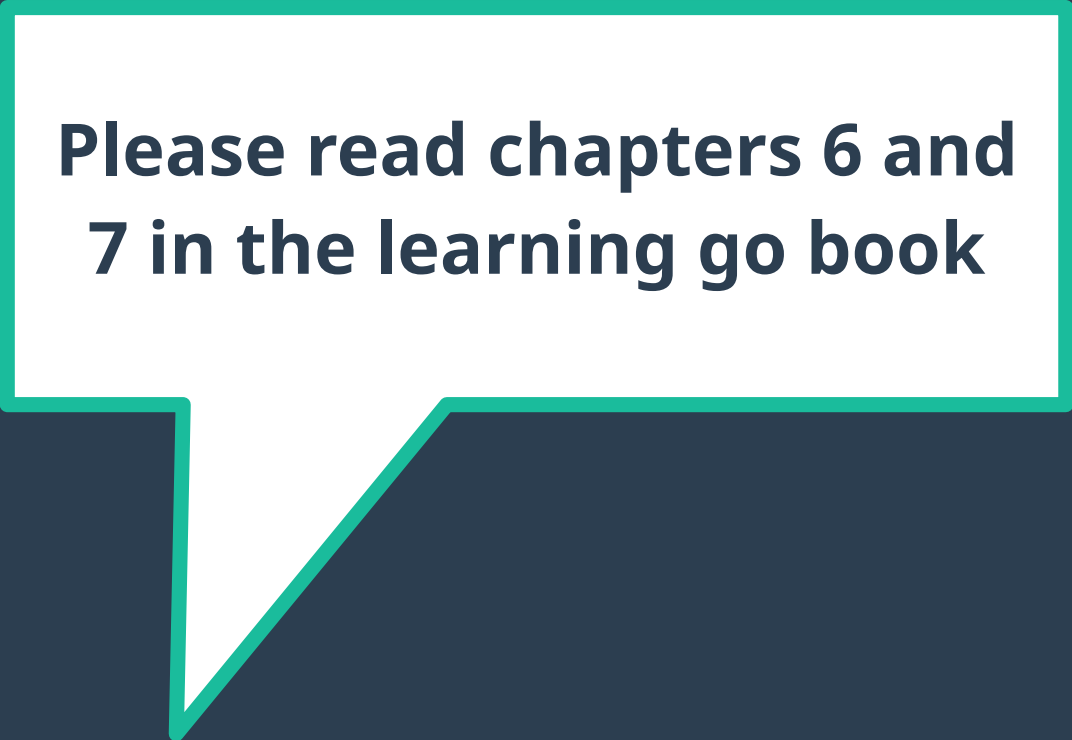
- Used to check/cast interface type to
 - Actual concrete type
 - Another interface type
- Interface var x and Type T
- `x.(T)` will assert the type and return x as a T or panic
 - What if you just wanted to check?
 - `realVal, ok := x.(realType)`
 - ok will be true and realvalue will be a variable of type realType if x is a struct of type realType. Otherwise realVal is nil and ok is false

An idomatic way to use type assertions

- **If file, ok := w.(*os.File); ok{**
 - //Use file here
 - }
- **Recall that first statement does assignment, second statement (just checking Boolean value of 'ok') is used for the if statement.**

Interfaces are comparable

- **What about comparing interfaces**
 - Two interface variables are equal (==) if
 - Both are nil
 - or
 - The underlying concrete type for both is same
 - The concrete objects are comparable and == returns true



**Please read chapters 6 and
7 in the learning go book**