# Module 1

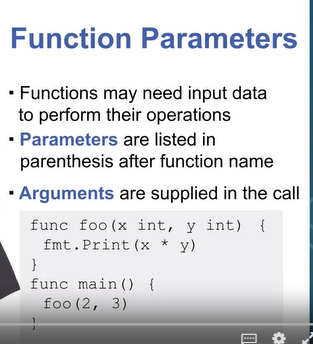
## M1.1.1 - Why Use Functions?

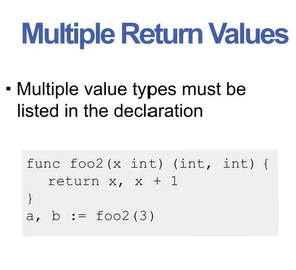
Main function is where the compiler starts



* Abstraction, repeatability

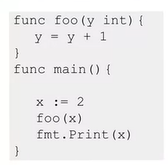
## M1.1.2 - Function Parameters and Return Values

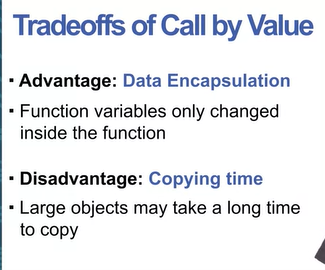




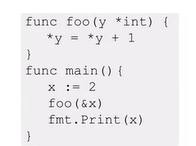
## M1.1.3 - Call by Value, Reference

Call by value is used in go

* Passed arguments are copied to parameters
* Modifying parameters has no effect outside the function
* This will just print 2

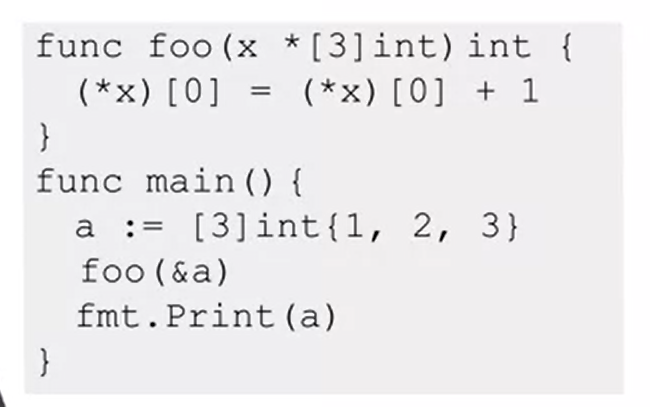


Call by Reference

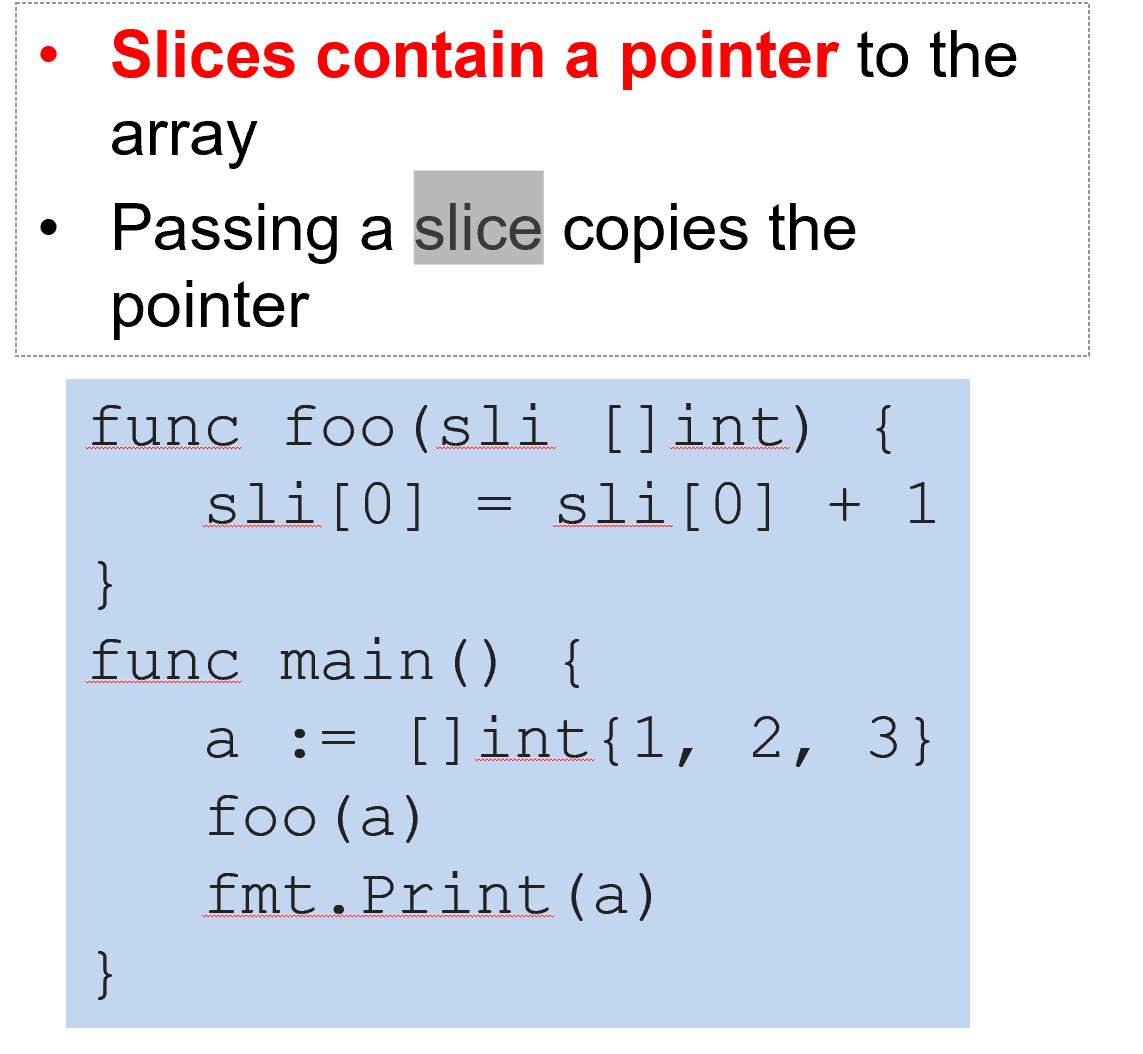
* Not strictly speaking built into this language
* All you need to do to implement this is to pass a pointer
* Called function has direct access to caller variable in memory
* 

## M1.1.4 - Passing Arrays and Slices

Pass as pointer (messy, more challenging – doesn’t scale well ):



* Easier to pass slices
* Passing a slice copies the pointer
* Slice is essentially a structure with the pointer, the length, and the capacity



## M1.2.1 - Well-Written Functions

* Understandability

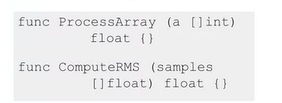
Debugging (high level):

1. Function is written incorrectly
   1. Sorts a slice in the wrong order for example
2. Data that the function uses is incorrect
   1. Sorts slice correctly, but slice has wrong elements in it

* Functions should be understandable, and data should be traceable

## M1.2.2 - Guidelines for Functions

Types for making good functions:

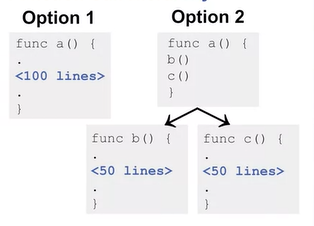
1. Function name
   1. Behaviour should be understood at a glance
   2. Parameter naming counts too
   3. Same function, one is more understandable
2. Function should perform only one ‘operation’
   1. Operation depends on the context
   2. Merging behaviours makes code complicated
3. Fewer parameters is better
   1. More parameters means you might have some bad functional cohesion
   2. Might be worthwhile splitting up the function
4. Reduce Parameter Numbers solutions?
   1. May need to group related arguments into structures
   2. Don’t ‘force’ things, only group them if they’re related

## M1.2.3 - Function Guidelines

Function Complexity:

* Function length is the most obvious measure
* You want functions to be as simple as possible

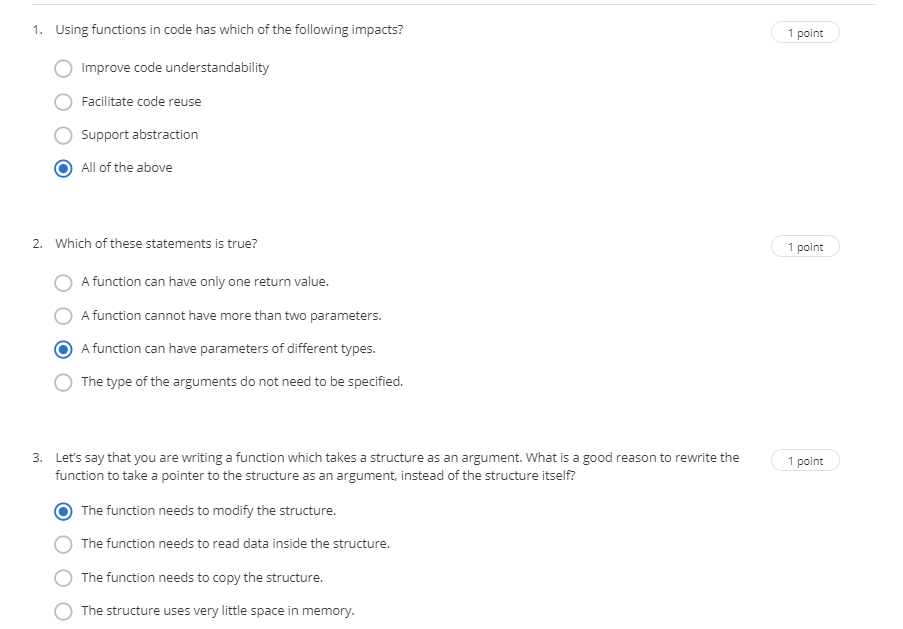
**Function Call Hierarchy:**

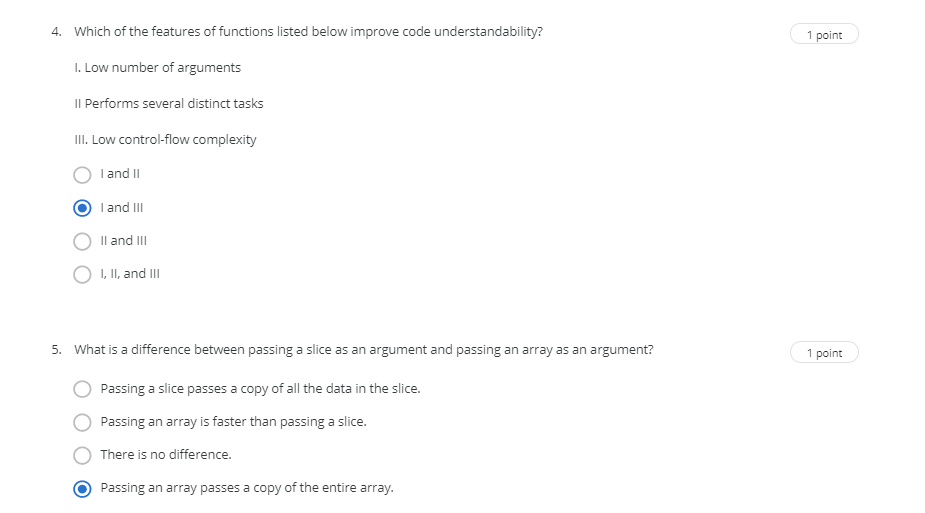
****

**Control-flow Complexity**

* Control-flow describes conditional paths
  + If statements for examples create additional control paths

## Module 1 Quiz





# Module 2

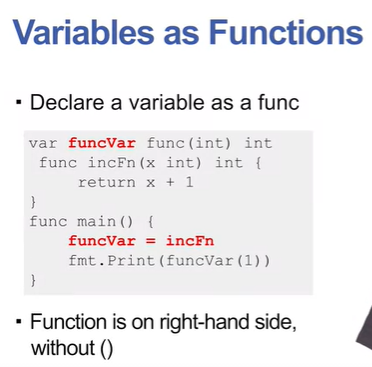
## M2.1.1 - First-Class Values

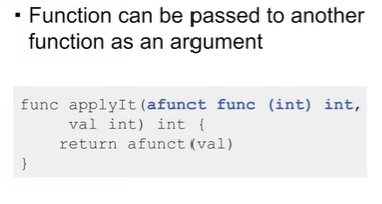
Functions are First-class

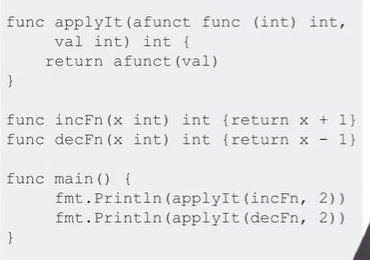
* Functions can be treated like other types
* Variables can be declared with a function type
* Created dynamically
* Can be passed as arguments and returned as values
* Can be stored in data structures

Variables as Functions

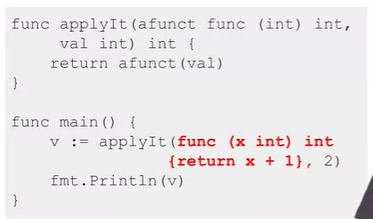
* Declare, then variable acts like an alias.





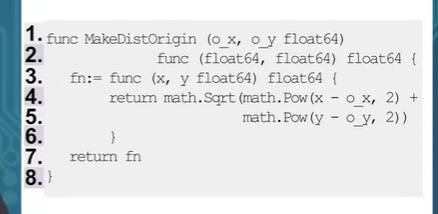


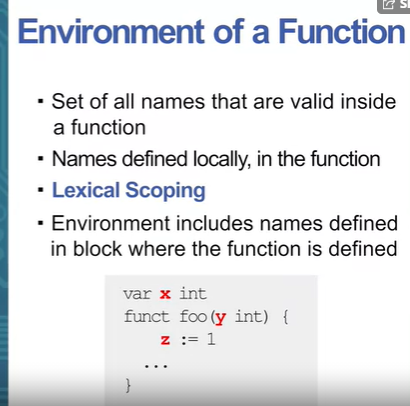
Anonymous Functions 🡪 Essentially the same thing as lambdas

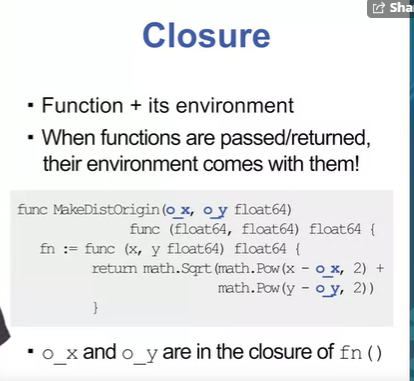


## M2.1.2 - Returning Functions

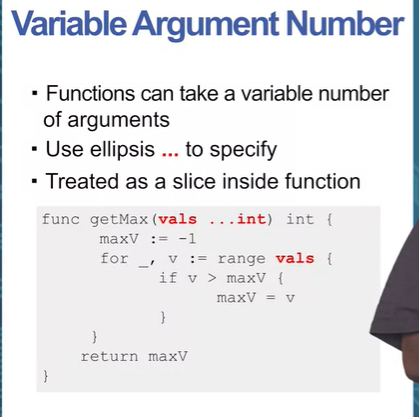
Essentially you’d be looking at a function that creates a new function based on certain input parameters





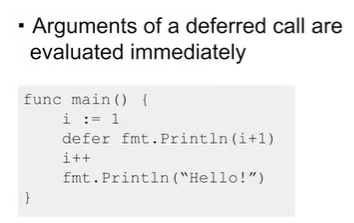


## M2.2.2 - Variadic and Deferred



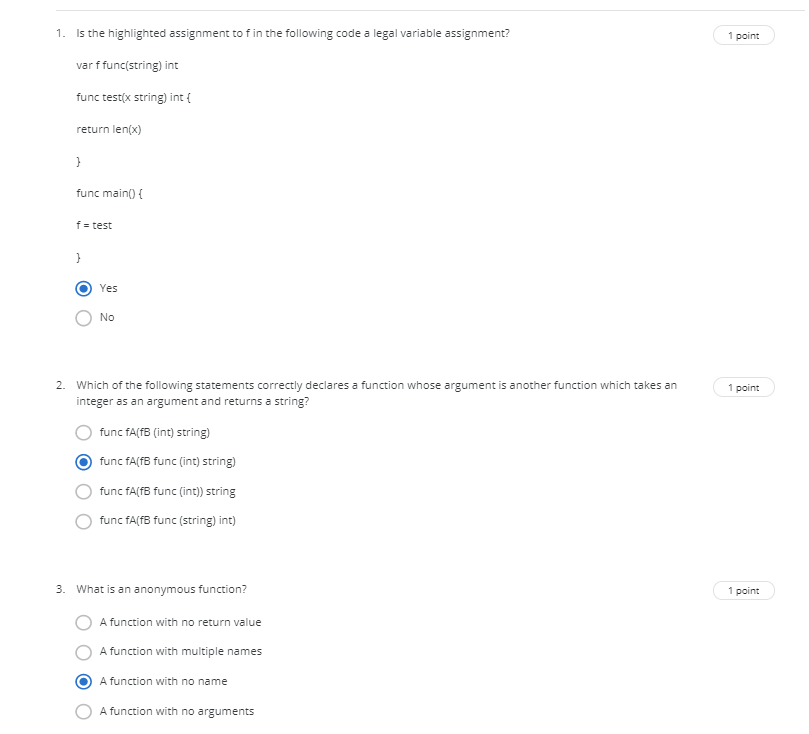


Essentially, you’re saying execute the deferred function when the surrounding function’s (the one it’s within) execution is complete.

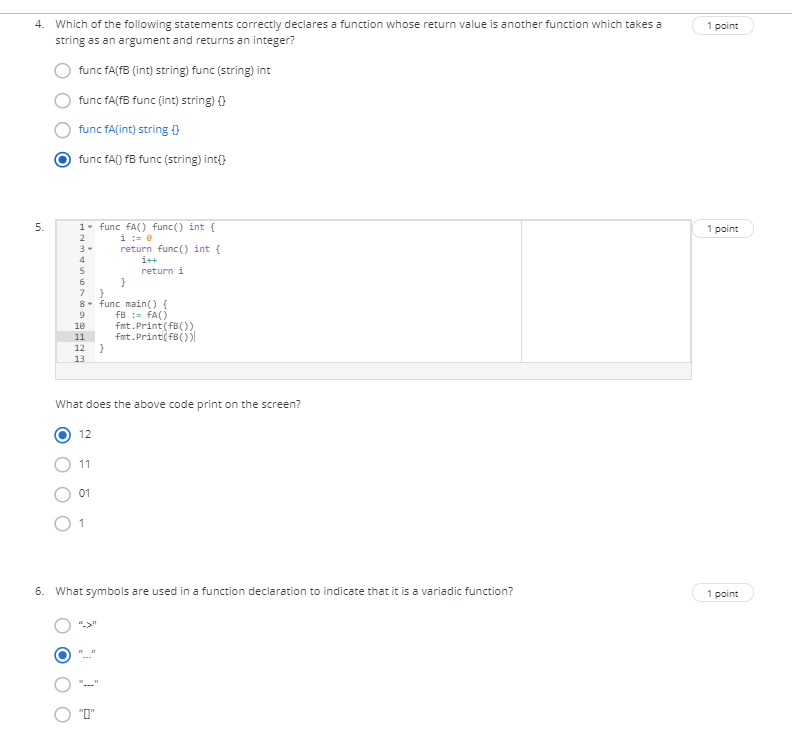


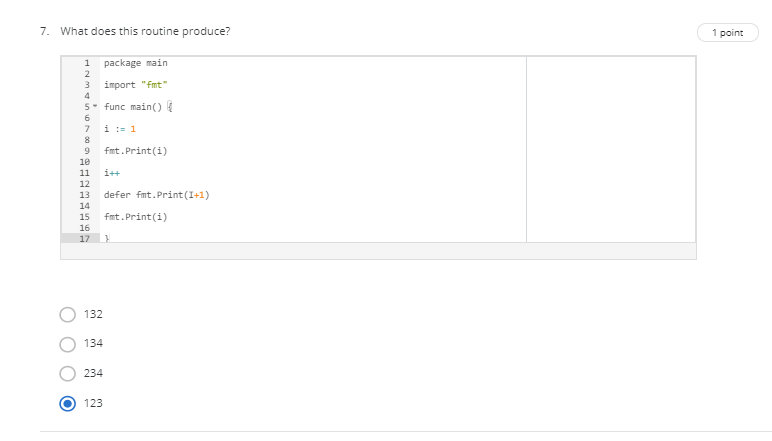
By the time the deferred statement is executed, I = 3, but 2 will be printed since it was 2 when arguments were evaluated.

## Module 2 Quiz



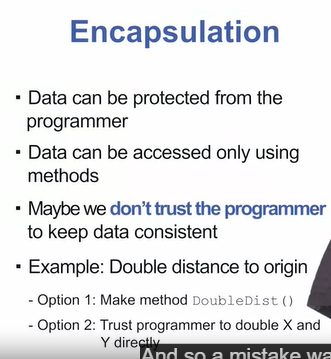
Q4 excluded – no correct answers presented





# Module 3

## M3.1.1 - Classes and Encapsulation



## M3.1.2 - Support for Classes (1)

Go does not have a class keyword

Go uses receiver types and standard dot notation

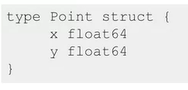


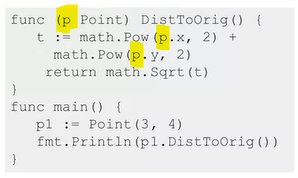
MyInt will be the receiver type

## M3.1.3 - Support for Classes (2)

The important thing to note is that with GO, you don’t have classes, you have receiver types

🡪 You know what the gameplan gotta be – using structs

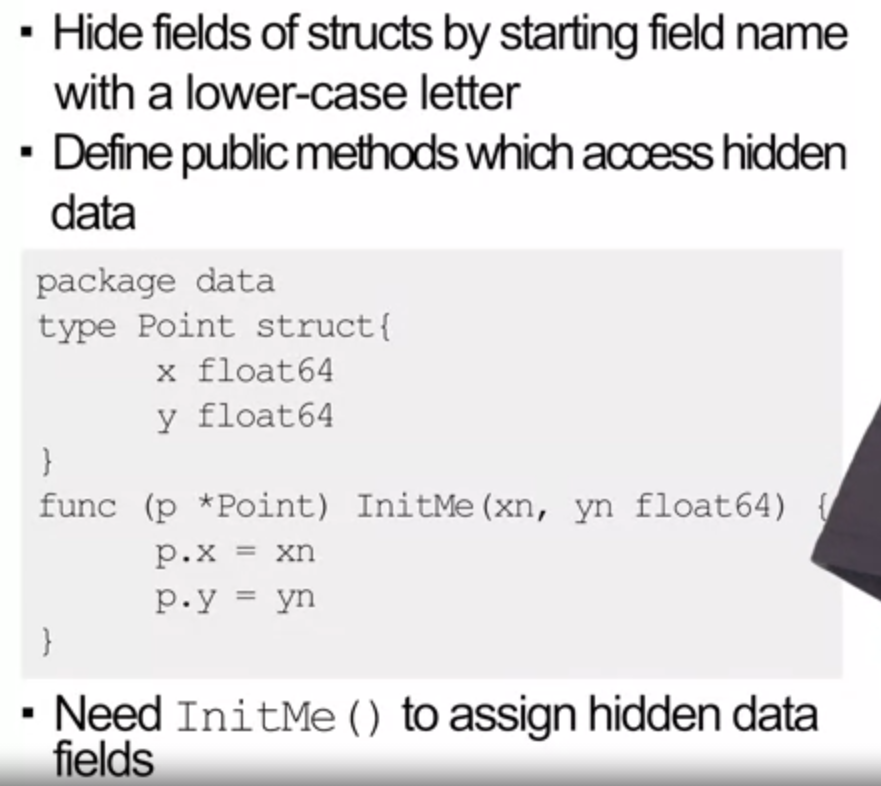
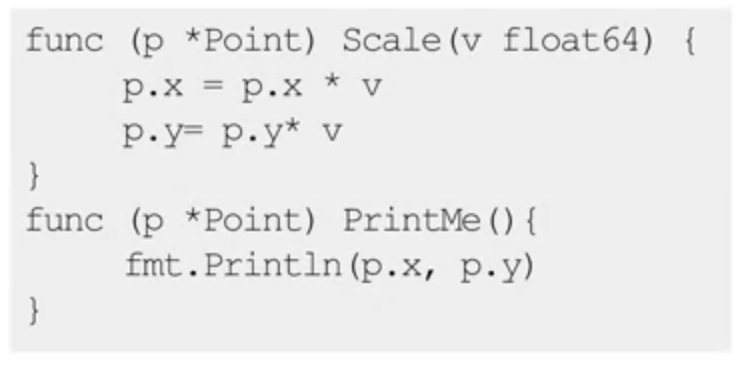
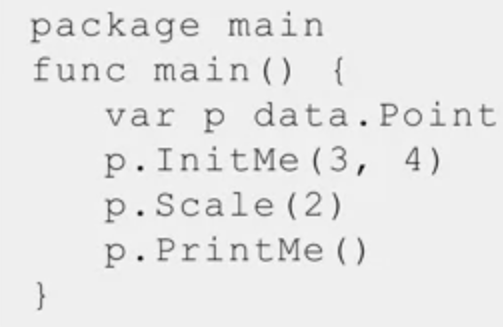




Highlighted is essentially your equivalent to ‘self’ … implicit passing

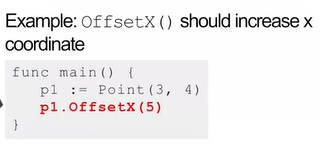
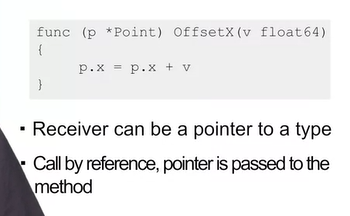
## M3.2.1 - Encapsulation

Controlling Access

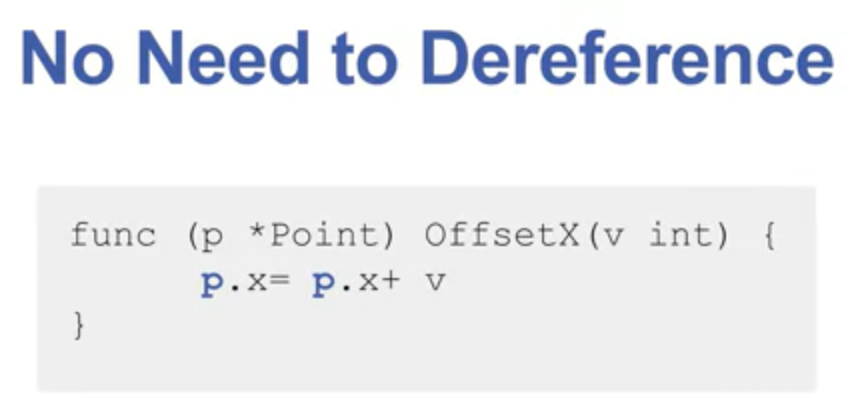
* Define public functions to allow access to hidden data
* Want to give access to data, but only in the way we define
* 
* 
* To make your methods public, you just need to start with a capital letter
  + Look at InitMe, PrintMe, Scale
* 
* 

## M3.2.2 - Point Receivers

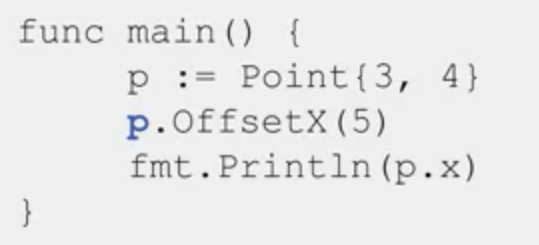
Limitations:

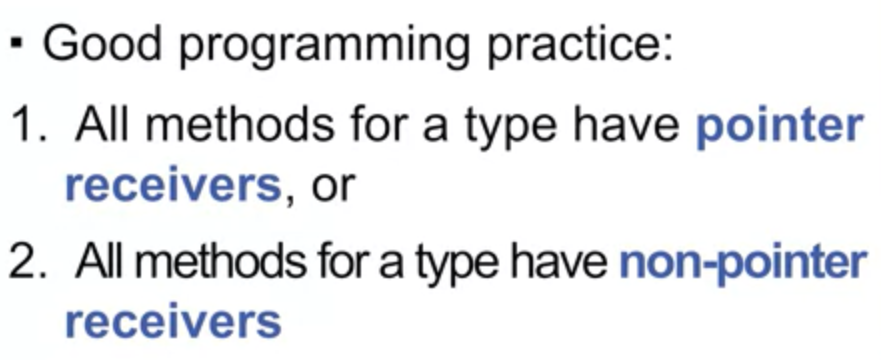
* Receiver type is implicitly passed
* Method cannot modify the data inside the receiver object
* This can’t change the actual value inside p1
* Lots of copying
* 
* Solution is to pass by pointer!
* 

## M3.2.3 - Point Receivers, Referencing, Dereferencing

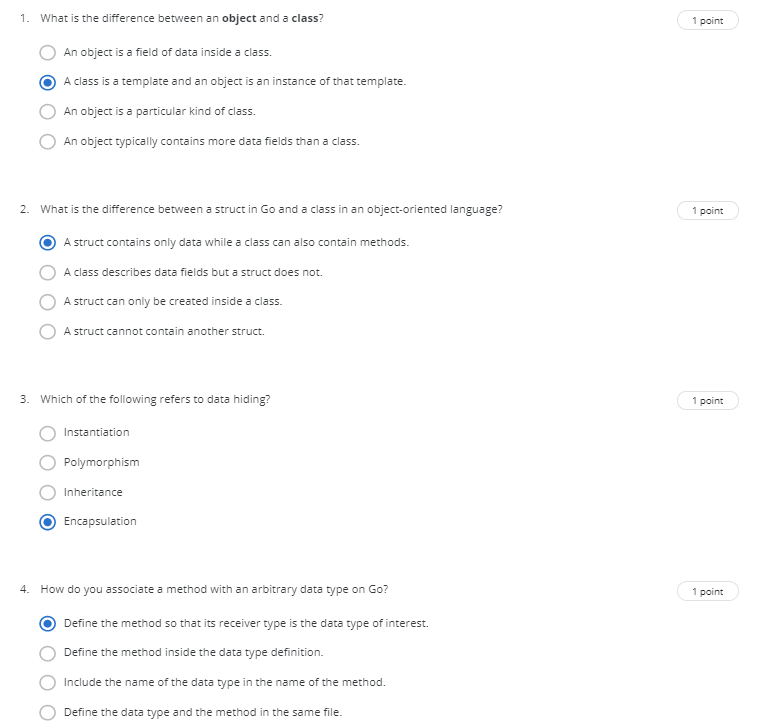


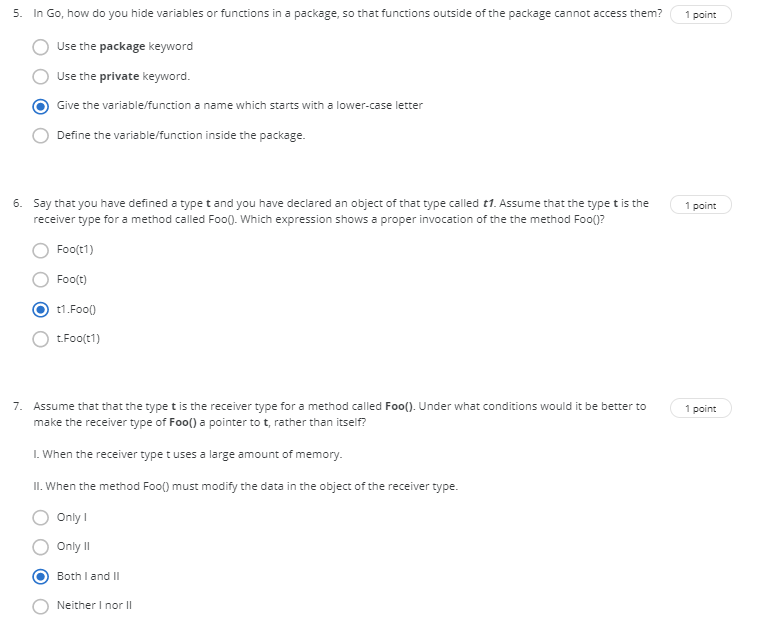
Golang doesn’t want you to dereference in these cases 😊

****



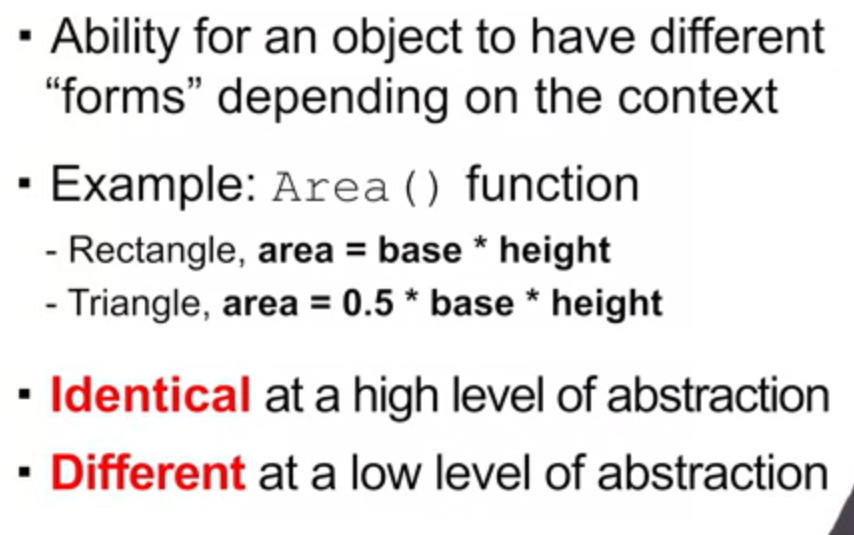
## Module 3 Quiz



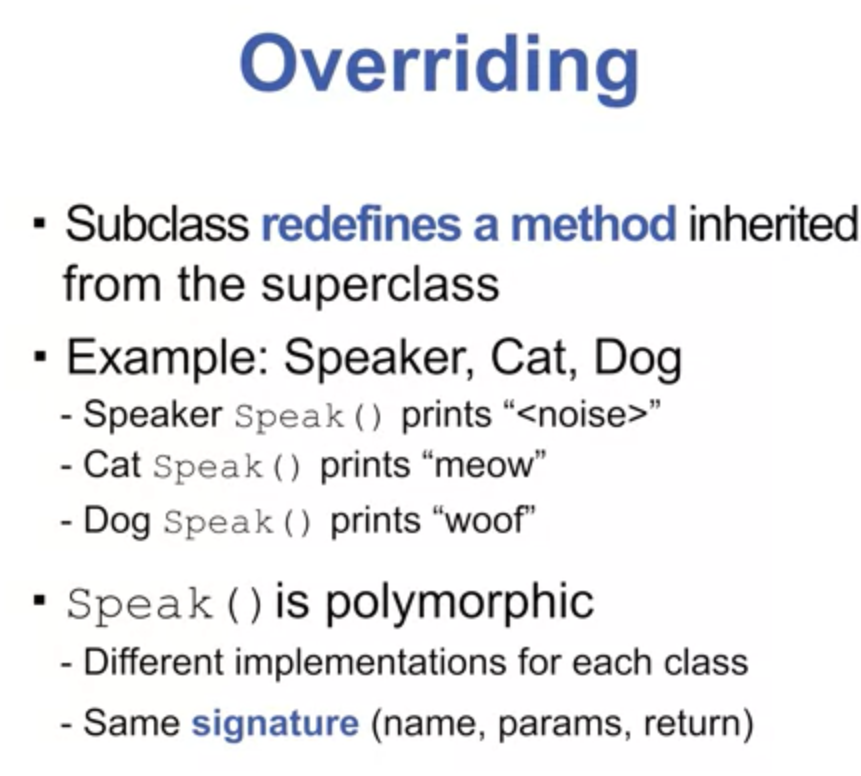


# Module 4

## M4.1.1 - Polymorphism



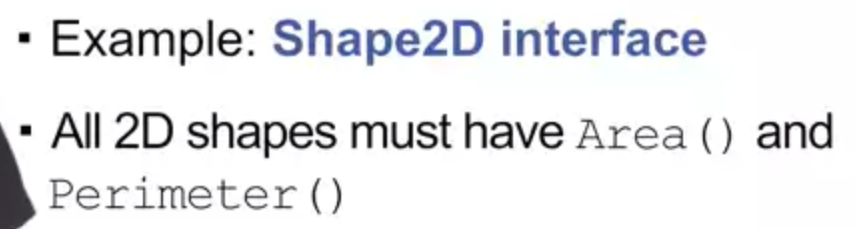
Go doesn’t have inheritance

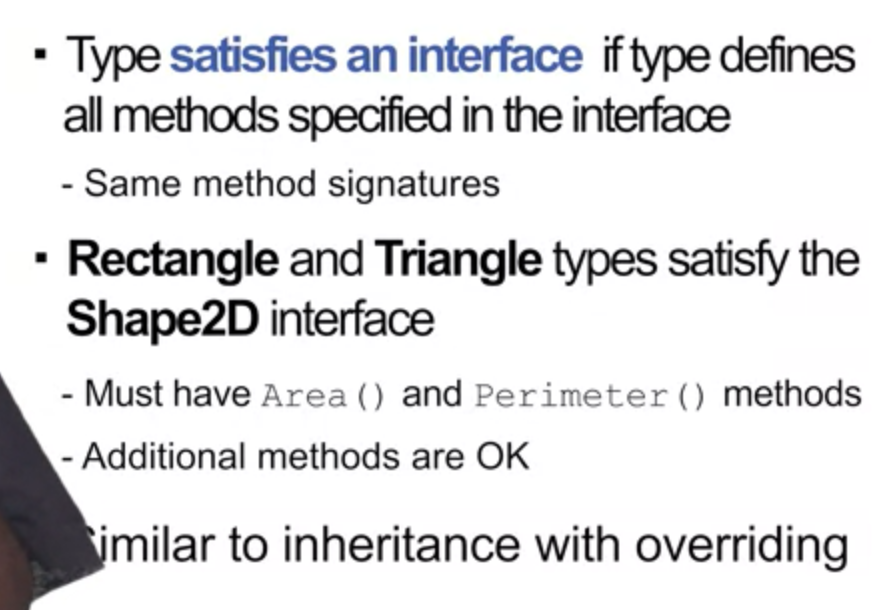


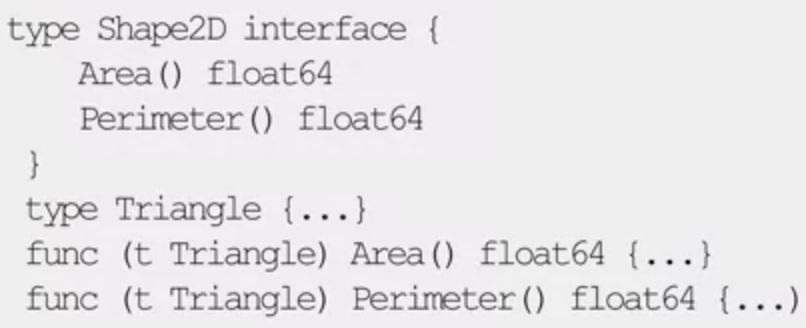
## M4.1.2 - Interfaces

Interfaces help us get polymorphism in go

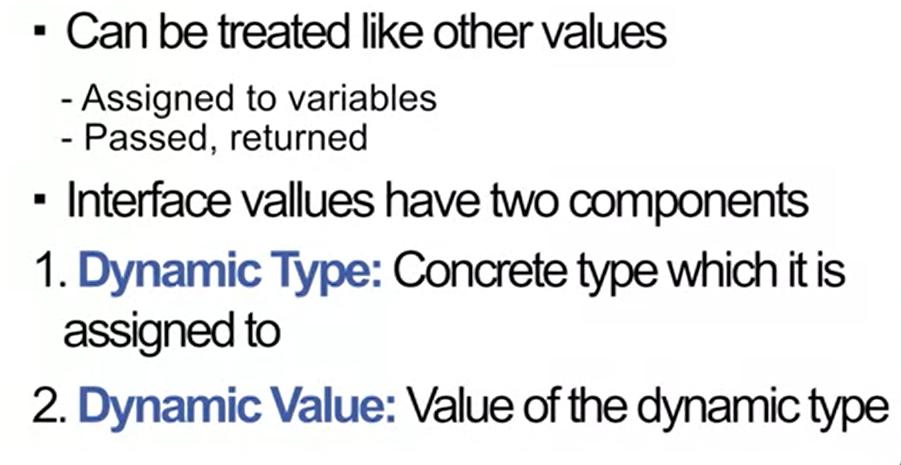
* Set of method signatures
* Implementation is not defined
* Used to express conceptual similarity between types

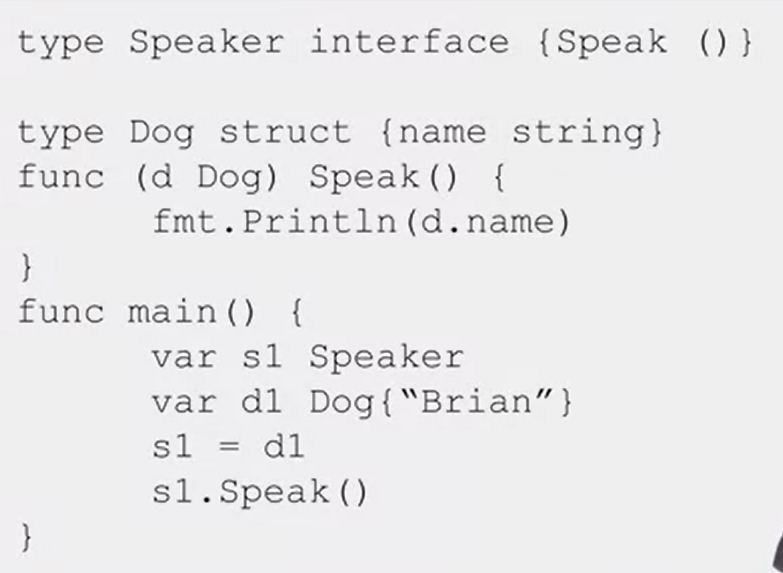


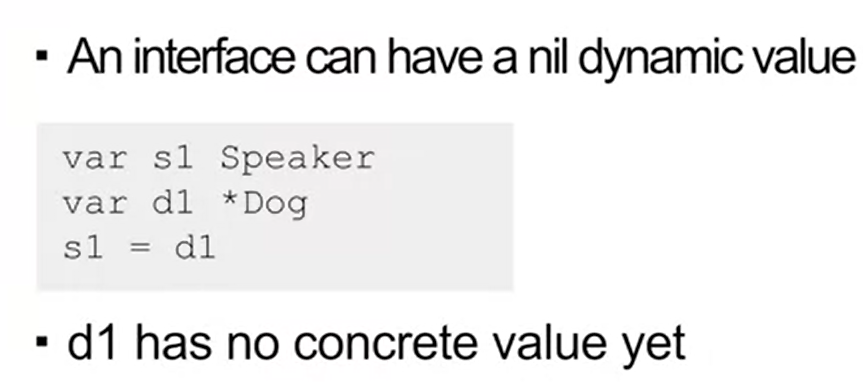




## M4.1.3 - Interface vs. Concrete Types

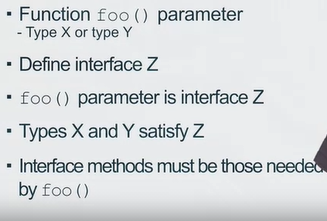


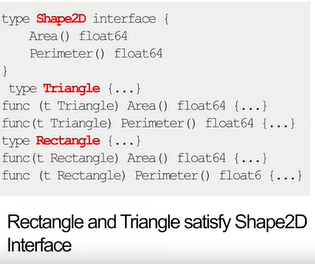


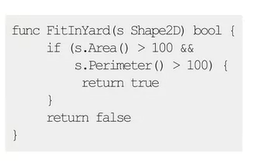
* Key point is at the s1 = d1
* This is legal because Dog object satisfies the speaker interface
* 

## M4.2.1 - Using Interfaces

* When you need a function to take multiple types of parameter

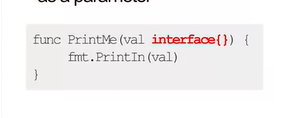
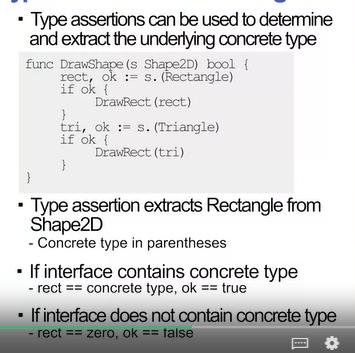






* Ensure that your argument can accept any type that matches the specified interface

Empty Interface

* Specifies no methods
* Just a way to accept any type
* 
* 
* S.(type)