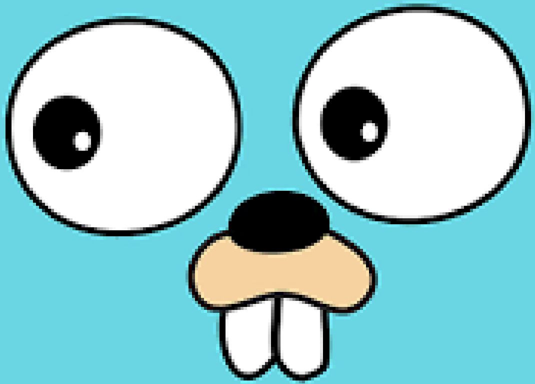


Phase 1: Introduction to Golang (Sessions 1-8)

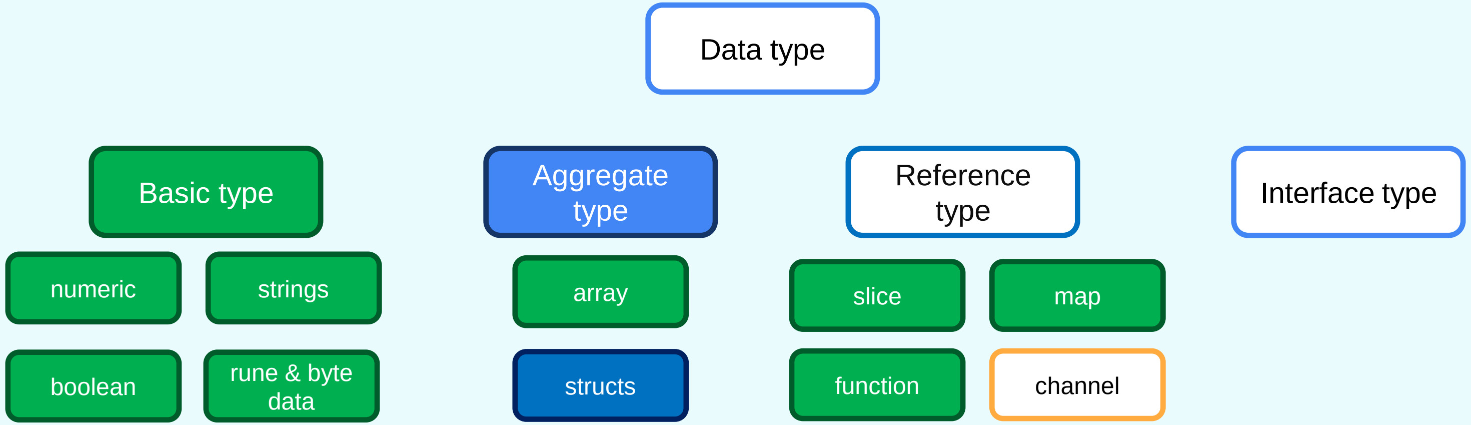
Objective: Build a strong foundation in Golang basics.

Session 6: Structs and Methods

Discussion Points

- Defining and using structs
- Methods on structs
- Embedding structs

What is next with Data Types?



Defining and using structs

What is Struct?

```
var carDoors int
var carModel string
var carBrand string
var carWeight float32
```

Defining and using structs

What is Struct?

Unlike traditional Object-Oriented Programming, Go does not have **class-object architecture**. Go is often categorized as a “**modern**” programming language rather than a purely object-oriented programming (OOP) language.

Go encourages a composition-based approach to code organization and reuse.

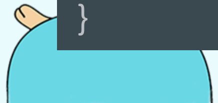
A struct or structure can be compared with the class in the Object-Oriented Programming paradigm. Declaring a struct type

```go
package main

type StructName struct {
   field1 fieldType1
   field2 fieldType2
}
```

```go
package main

type StructName struct {field1 fieldType1 ; field2 fieldType2}
```



Defining and using structs

Initializing a Struct?

We have used the shorthand notation (using := syntax) to create the variable **student** so that, Go can infer type **Student** automatically. The order of the appearance of struct's fields does not matter, as you can see, we have initialized the lastName field before the firstName field.

The comma (,) is absolutely necessary after the value assignment of the last field while creating a struct using the above syntax. This way, Go won't add a semicolon just after the last field while compiling the code.

You can also initialize only some fields of a struct and leave others to their zero values.

```go
type Student struct {
   firstName string
   lastName  string
}

func main() {
   student := Student{
     lastName:  "Aliyev",
     firstName: "Ali",
   }

   fmt.Println(student)
}
```

Defining and using structs

Anonymous Struct?

An anonymous structure is a structure which does not contain a name. It useful when you want to create a one-time usable structure

In the right program, we are creating a struct **student** without defining a derived struct type. By using fmt.Printf function and **%T** format syntax, we get the following result.

{Ali Aliyev}
struct { firstName string; lastName string }

```go
package main

import "fmt"

func main() {
   student := struct {
     firstName string
     lastName  string
   }{
     "Ali",
     "Aliyev",
   }

   fmt.Println(student)

   fmt.Printf("%T", student)
}
```

Defining and using structs

Nested Struct?

A struct field can be of any data type, including another struct. Therefore, it is completely valid for a struct field to hold a struct as its value. When this happens, the field is referred to as a nested struct because it resides within a parent struct.

In the example, we defined a new struct type called **Teacher** to represent a student's teacher. Then, we updated the **teacher** field in the **Teacher** struct to hold a value of type Teacher. When initializing the **student** struct of type **Student**, we provided values for all the fields, including the **teacher**. Since the **teacher** field holds a struct of type **Teacher**, we assigned a struct value to it, using a shorthand method by excluding field names during initialization.

```go
type Teacher struct {
   fullName string
   area     string
}

type Student struct {
   firstName string
   lastName  string
   teacher   Teacher
}

func main() {
   student := Student{
     firstName: "Ali",
     lastName:  "Aliyev",
     teacher: Teacher{
       fullName: "Robert Griesemer",
       area:     "Computer Science",
     },
   }

   fmt.Println(student.teacher.area)
}
```

Defining and using structs

Getting and setting Struct fields?

To access individual fields of a struct, you use the dot (.) operator. To assign a value to the field, the syntax would be:

```go
type Teacher struct {
   fullName string
   area     string
}

type Student struct {
   firstName string
   lastName  string
   teacher   Teacher
}

func main() {
   var student Student

   student.firstName = "Ali"
   student.lastName = "Aliyev"
   student.teacher.fullName = "Robert Griesemer"
   student.teacher.area = "Computer Science"

   fmt.Println(student)
}
```

Defining and using structs

Function fields in Struct?

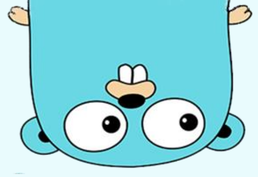
We defined a struct type **Student** with two string fields and one function field. For simplicity, we also created a derived function type called **Teacher**. While initializing the struct **student**, we must ensure the **fullName** field follows the function type syntax. In this case, we assigned it an anonymous function. Since the syntax of this anonymous function matches the declaration of **Teacher**, the assignment is valid and works correctly.

```
type Teacher func(string, string) string

type Student struct {
   firstName string
   lastName  string
   teacher   Teacher
}

func main() {
   student := Student{
      firstName: "Ali",
      lastName:  "Aliyev",
      teacher: func(fullName string, area string) string {
         return fmt.Sprintf("%s is in a %s area", fullName, area)
      },
   }

   fmt.Println(student.teacher("Robert", "Computer Science"))
}
```



Defining and using structs

Struct with Tags?

Structs provide an additional feature that allows you to add metadata to their fields. This is commonly used to specify how a field should be transformed during encoding, decoding, or when interacting with a database. However, you can use this metadata to store any information you need, whether it's intended for another package or for your own internal use.

```go
type Student struct {
   FirstName string  `json:"firstName"`
   LastName string  `json:"lastName"`
   Teacher  Teacher `json:"teacher"`
}
```

Methods on structs

What with Method?

In Go, methods are functions tied to a specific type, allowing us to define behavior for that type, making the code more modular and reusable. By using a receiver argument, the method can access and manipulate the properties of the receiver. The receiver can be either a struct type or a non-struct type.

```
func (receiverName receiverType) methodName(parameters) returnType {
   // method body
}
```

Methods on structs

Method declarations?

The **receiverType** is the type that the method is tied to, which can be a struct or any other user-defined type. It can be a **value** or a **pointer**. Here is an example of a method associated with a type:

```
type Student struct {
    firstName string
    lastName  string
}

func (s *Student) SetFirstName(f string) {
    s.firstName = f
}

func (s Student) Learning() string {
    return fmt.Sprintf("%s is learning %s", "Ali", "Golang")
}
```

Methods on non-struct type?

In the previous example, we defined methods with struct type receivers. In Go, you can define methods for **non-struct types**, but both the type and the method must be in the same package. If the type comes from another package (like int, string, etc.), the compiler will throw an error.

```go
type NewString string

func (n NewString) ToLower() string {
   return strings.ToLower(string(n))
}
```

Pointer Receivers vs Value Receivers in Methods?

The choice between a **pointer receiver** and a **value receiver** depends on factors like whether the method needs to modify the receiver, whether the type has fields that can't or shouldn't be copied, and the size of the object.
If you're unsure, it's generally safer to use a pointer receiver.

Value receiver method	Pointer receiver method
If you only need to read from the value and not modify it	If you need to modify the value or avoid copying large structs
The internal copy process might impact your program’s performance*	Can be used to avoid copying large structs, which can improve performance

Methods on structs

Methods vs Functions?

One key difference between functions and methods is that you can define multiple methods with the same name, as long as their receivers are different, whereas functions with the same name cannot coexist in the same package.

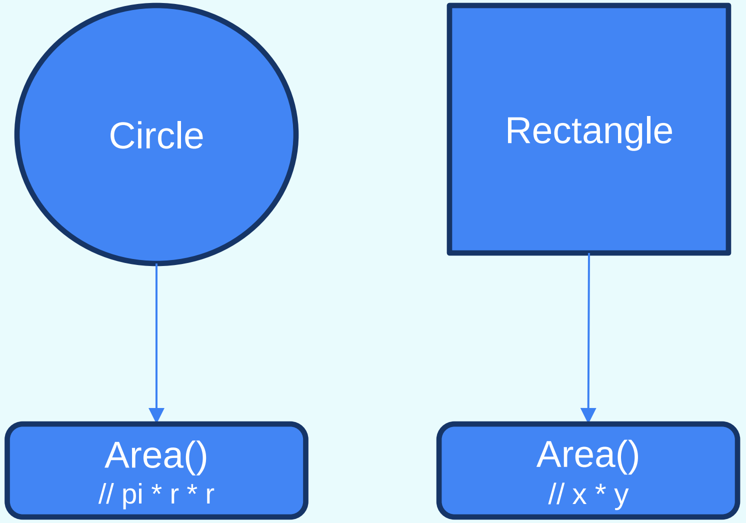
Functions are stateless, meaning for the same input, they will always produce the same output and are not dependent on any internal state.

Method	Function
It contains a receiver	It does not contain a receiver
Methods of the same name but different receivers can be defined in the program	Functions of the same name but different type are not allowed to be defined in the same package

Methods on structs

Methods with the same name?

One key difference between functions and methods is that you can define multiple methods with the same name, while you cannot define two functions with the same name in a package. It’s possible to have methods with the same name as long as their receivers are different.



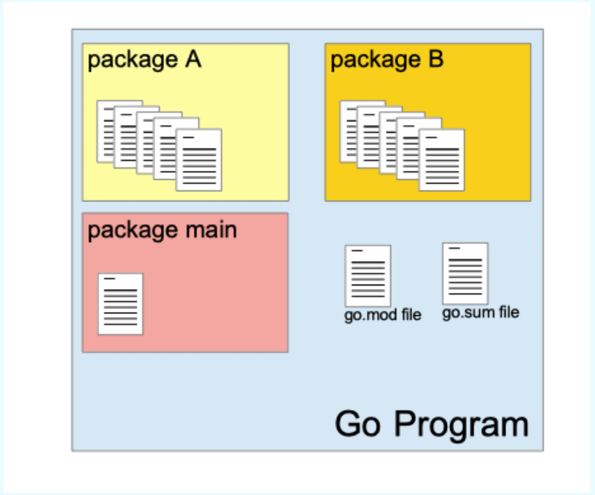
Methods on structs

Methods visibility?

Go does not have any public, private or protected keyword. The only mechanism to control the visibility outside the package is using the capitalized and non-capitalized formats.

- **Capitalized identifiers are exported**. The capital letter indicates that this is an exported identifier and is available outside the package.
- **Non-capitalized identifiers are not exported**. The lowercase indicates that the identifier is not exported and will only be accessed from within the same package.

When creating a package, you must consider method visibility. An exported method can be called from outside the package, while a non-exported method cannot. A method is exported when its name starts with an uppercase letter, indicating it is accessible outside the package. If the method name starts with a lowercase letter, it remains unexported and is only available within the package.



Methods on structs

Promoted Methods?

It's important to note that Go allows defining a struct type **without explicitly naming its fields**. In this case, Go automatically infers field names based on the field types, demonstrating the language's simplicity and flexibility. This approach also applies to nested structs, where you can omit the field name, and Go will use the struct type itself as the field name.

```go
type Teacher struct {
   fullName string
   area     string
}

type Student struct {
   firstName string
   lastName  string
   Teacher
}

func main() {
   s := Student{
     firstName: "Ali",
     lastName:  "Aliyev",
     Teacher: Teacher{
       fullName: "Robert Griesemer",
       area:     "Computer Science",
     },
   }
   fmt.Println(s.area)
}
```

Methods on structs

Methods on nested vs anonymous field struct?

```go
func (s *Student) SetTeacherArea(str string) {
   s.teacher.area = str
}

func main() {
   s := Student{
     firstName: "Ali",
     lastName:  "Aliyev",
     teacher: Teacher{
       fullName: "Robert Griesemer",
     },
   }

   s.SetTeacherArea("Programming")
   fmt.Println(s.teacher.area)

}
```

```go
func (s *Student) SetTeacherArea(str string) {
   s.area = str
}

func main() {
   s := Student{
     firstName: "Ali",
     lastName:  "Aliyev",
     Teacher: Teacher{
       fullName: "Robert Griesemer",
     },
   }

   s.SetTeacherArea("Programming")
   fmt.Println(s.area)

}
```

Defining and using structs

Struct String method?

In Go, the **String()** method is a special method that lets you define a custom string representation for your user-defined types (such as structs). By implementing the String() method for a struct, you control how instances of that struct are formatted when converted to a string using functions like fmt.Printf(), fmt.Println(), or during string interpolation. This makes debugging and logging more intuitive and informative by displaying structured data in a meaningful way.

```go
type Teacher struct {
    fullName string
    area     string
}

type Student struct {
    firstName string
    lastName  string
    teacher   Teacher
}

func (s Student) String() string {
    return fmt.Sprintf("%s %s (Teacher: %v)", s.firstName, s.lastName, s.teacher)
}
```

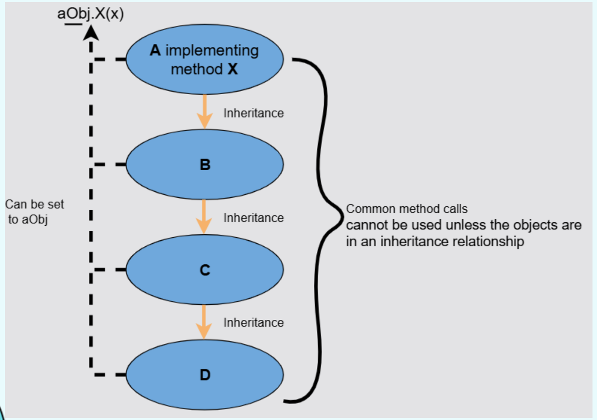
Embedding structs

Struct Composition over Inheritance?

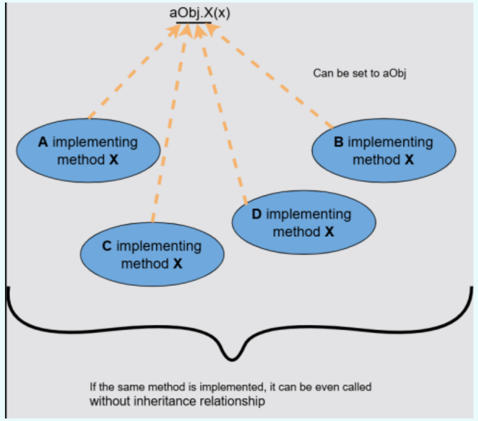
Go emphasizes the use of structs and methods to define data structures and their associated behaviors. While you can define methods on structs, Go does not support traditional classes or inheritance. Instead, Go promotes composition over inheritance as the preferred approach for code reuse and structuring.

Composition allows you to build complex types by combining simpler types, fostering code modularity and flexibility. Let's look at an example that illustrates composition over inheritance:

In C++


aObj.X(x)
A implementing method X
Inheritance
B
Inheritance
C
Inheritance
D
Can be set to aObj
Common method calls cannot be used unless the objects are in an inheritance relationship

In Go



Embedding structs

Struct Composition over Inheritance?

In this example, the **Phone** and **Tablet** struct embeds the **Device** struct. This allows the **Phone** struct to inherit the properties and methods of the **Device** struct, enabling reuse of functionality through composition rather than inheritance.

```go
type Device struct {
    name string
}

type Phone struct {
    Device
}

type Tablet struct {
    Device
}

func (d Device) Screen() {
    fmt.Printf("%s device has screen\n", d.name)
}
```

Session Quiz

Go?

Join at

slido.com

# 4180120

Thank you.

Questions?

- Defining and using structs
- Methods on structs
- Embedding structs

“Learn from yesterday, live for today, hope for tomorrow. The important thing is not to stop questioning.”

- Albert Einstein