Part 26: Structs Instead of Classes - OOP in Go

21 AUGUST 2017

Welcome to tutorial no. 26 in Golang tutorial series.

Is Go Object Oriented?

Go is not a pure object oriented programming language. This excerpt taken from Go's FAQs answers the question of whether Go is Object Oriented.

> Yes and no. Although Go has types and methods and allows an object-oriented style of programming, there is no type hierarchy. The concept of "interface" in Go provides a different approach that we believe is easy to use and in some ways more general. There are also ways to embed types in other types to provide something analogous—but not identical—to subclassing. Moreover, methods in Go are more general than in C++ or Java: they can be defined for any sort of data, even built-in types such as plain, "unboxed" integers. They are not restricted to structs (classes).

In the upcoming tutorials we will discuss how object oriented programming concepts can be implemented using Go. Some of them are quite different in implementation compared to other object oriented languages such as Java.

Structs Instead of Classes

Go does not provide classes but it does provide structs. Methods can be added on structs. This provides the behaviour of bundling the data and methods that operate on the data together akin to a class.

Lets start with a example right away for better understanding.

We will create a custom package in this example as it helps to better understand how structs can be a effective replacement for classes.

Create a folder inside your Go workspace and name it `oop`. Create a subfolder `employee` inside `oop`. Inside the `employee` folder, create a file named employee.go

The folder structure would look like,

```
    workspacepath -> oop -> employee -> employee.go
```

Please replace the contents of `employee.go` with the following,

```
package employee

import (
    "fmt"
)

type Employee struct {
    FirstName   string
    LastName    string
    TotalLeaves int
    LeavesTaken int
}

func (e Employee) LeavesRemaining() {
    fmt.Printf("%s %s has %d leaves remaining", e.FirstNa
}
```

In the program above, the first line specifies that this file belongs to the `employee` package. An Employee struct is declared in line no. 7. A method named `LeavesRemaining` is added to the `Employee` struct in line no. 14. This calculates and displays the number of remaining leaves an employee has. Now we have a struct and a method that operates on a struct bundled together akin to a class.

Create a file named `main.go` inside the `oop` folder.

Now the folder structure would look like,

```
workspacepath -> oop -> employee -> employee.go
workspacepath -> oop -> main.go
```

The contents of `main.go` is provided below.

```
package main

import "oop/employee"

func main() {
    e := employee.Employee {
        FirstName: "Sam",
        LastName: "Adolf",
        TotalLeaves: 30,
        LeavesTaken: 20,
    }
    e.LeavesRemaining()
}
```

We import the `employee` package in line no. 3. The `LeavesRemaining()` method of the `Employee` struct is called from line no. 12 in `main()`.

This program cannot be run on the playground as it has a custom package. If you run this program in your local by issuing the commands `go install oop` followed by `workspacepath/bin/oop`, the program will print the output,

```
Sam Adolf has 10 leaves remaining
```

New() function instead of constructors

The program we wrote above looks alright but there is a subtle issue in it. Lets see what happens when we define the employee struct with zero values. Change the contents of `main.go` to the following code,

```
package main

import "oop/employee"

func main() {
    var e employee.Employee
    e.LeavesRemaining()
}
```

The only change we have made is creating a zero value `Employee` in line no.6. This program will output,

```
  has 0 leaves remaining
```

As you can see, the variable created with the zero value of `Employee` is unusable. It doesn't have a valid first name, last name and also doesn't have valid leave details.

In other OOP languages like java, this problem can be solved by using constructors. A valid object can be created by using parameterised constructor.

Go doesn't support constructors. If the zero value of a type is not usable, it is the job of the programmer to unexport the type to prevent access from other packages and also to provide a function named `NewT(parameters)` which initialises the type `T` with the required values. It is a convention in Go to name a function which creates a value of type `T` to `NewT(parameters)`. This will act like a constructor. If the package defines only one type, then its a convention in Go to name this function just `New(parameters)` instead of `NewT(parameters)`.

Let's make changes to the program we wrote so that every time a employee is created, it is usable.

First step is to unexport the `Employee` struct and create a function `New()` which will create a new `Employee`. Replace the code in `employee.go` with the following,

```
package employee

import (
    "fmt"
)

type employee struct {
    firstName   string
    lastName    string
    totalLeaves int
    leavesTaken int
}

func New(firstName string, lastName string, totalLeave i
    e := employee {firstName, lastName, totalLeave, leav
    return e
}

func (e employee) LeavesRemaining() {
    fmt.Printf("%s %s has %d leaves remaining", e.firstNa
}
```

We have made some important changes here. We have made the starting letter `e` of Employee struct to lower case, that is we have changed `type Employee struct` to `type employee struct`. By doing so we have successfully unexported the `employee` struct and prevented access from other packages. It's a good practice to make all fields of a unexported struct to be unexported too unless there is a specific need to export them. Since we don't need the fields of the `employee` struct anywhere outside the package, we have unexported all the fields too.

We have changed the field names accordingly in `LeavesRemaining()` method.

Now since `employee` is unexported, its not possible to create values of type `Employee` from other packages. Hence we are providing a exported `New` function in line no. 14 which takes the required parameters as input and returns a newly created employee.

This program still has changes to be made to make it work but lets run this to understand the effect of the changes so far. If this program is run it will fail with the following compilation error,

```
go/src/constructor/main.go:6: undefined: employee.Employee
```

This is because we have unexported `Employee` and hence the compiler throws error that this type is not defined in `main.go`. Perfect. Just what we wanted. Now no other package will be able to create a zero valued `employee`. We have successfully prevented a unusable employee struct value from being created. The only way to create a employee now is to use the `New` function.

Replace the contents of `main.go` with the following,

```
package main

import "oop/employee"

func main() {
    e := employee.New("Sam", "Adolf", 30, 20)
    e.LeavesRemaining()
}
```

The only change to this file is in line no. 6. We have created a new employee by passing the required parameters to the `New` function.

Here are contents of the two files after making the required changes,

employee.go

```
package employee

import (
    "fmt"
)

type employee struct {
    firstName   string
    lastName    string
    totalLeaves int
    leavesTaken int
}

func New(firstName string, lastName string, totalLeave i
    e := employee {firstName, lastName, totalLeave, leav
    return e
}

func (e employee) LeavesRemaining() {
    fmt.Printf("%s %s has %d leaves remaining", e.firstNa
}
```

main.go

```
package main

import "oop/employee"

func main() {
    e := employee.New("Sam", "Adolf", 30, 20)
    e.LeavesRemaining()
}
```

Running this program will output,

```
Sam Adolf has 10 leaves remaining
```

Thus you can understand that although Go doesn't support classes, structs can effectively be used instead of classes and methods of signature `New(parameters)` can be used in the place of constructors.

Thats it for classes and constructors in Go. Have a good day.

Next tutorial - Composition Instead of Inheritance

Naveen Ramanathan

iOS developer at dietco.de and Golang enthusiast.

Share this post

About

For any queries/suggestions, please contact us at **naveen[at]golangbot[dot]com**

Follow Us

Newsletter

Join Our Newsletter

Signup for our newsletter and get the **Golang tools cheat sheet for free**.

Email*

Subscribe