Introduction to OOP in Golang?

Go is not an object oriented language per se.

developers need to force create object-oriented style with the minimum language support that it provides.

**Go has no support of inheritance**

Similarly, a kind of polymorphism *can* be created using interfaces.

## How to Create Objects in Go and Golang

The key idea behind object-orientation is to separate the code into several manageable parts or *objects*.

***Encapsulation****:* Each object has its own identity that is determined by the *data* (*attributes*) and the *methods* (*behavior*) that operate on the data, The attributes are typically hidden and can only be accessed through the methods defined within the object. This is called *encapsulation*.

Go can encapsulate its attributes using **struct**.

package payroll

type Employee struct {

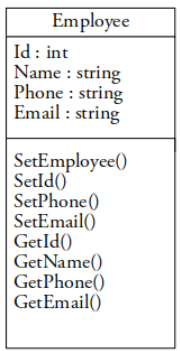
Id int

name string

phone string

email string

}



The above Golang code defines the abstract data type with attributes. Once it has been created we can instantiate it as follows:

e1 := payroll.Employee{Id: 101}

**OR**

e1 := payroll.Employee{}

**OR**

var ep1 \*payroll.Employee

ep1 = &payroll.Employee{Id: 102}

ep1.SetEmployee("Anita", "4787753", "anita@gmail.com")

### **Data Hiding in Go**

e1 := payroll.Employee{Id: 101, name:"Anita",phone:"2783648",email:"anita@gmail.com" } // Error!

But what if we write the code as follows:

Go doesn’t have *private* or *public concepts* to restrict the accessibility of attribute.

Instead, what it provides is a means for package-level accessibility.

We have written **Id** in the **Employee** structure, with the first letter in *uppercase*. This means that the attribute **Id** can be accessed directly outside the package when the object is created.

While the attributes such as **name**, **phone**, and **email** are written in *lowercase* and can only be accessed through member methods.

Which, in Go, are defined as follows:

The below code provides a means to implement a sort of encapsulation in Go. To quickly test the idea,

func (e \*Employee) GetName() string {

return e.name

}

func (e \*Employee) SetName(name string) {

e.name = name

}

To quickly test the idea, here is the complete code:

package payroll

type Employee struct {

Id int

name string

phone string

email string

}

func (e \*Employee) SetEmployee(name, phone, email string) {

e.SetName(name)

e.SetPhone(phone)

e.SetEmail(email)

}

func (e \*Employee) GetId() int {

return e.Id

}

func (e \*Employee) GetName() string {

return e.name

}

func (e \*Employee) GetPhone() string {

return e.phone

}

func (e \*Employee) GetEmail() string {

return e.email

}

func (e \*Employee) SetId(id int) {

e.Id = id

}

func (e \*Employee) SetName(name string) {

e.name = name

}

func (e \*Employee) SetPhone(phone string) {

e.phone = phone

}

func (e \*Employee) SetEmail(email string) {

e.email = email

}

//-------------------------------------------------------------

package main

import (

"fmt"

"oop\_prog/payroll"

)

func main() {

e1 := payroll.Employee{Id: 101}

e1.SetEmployee("Bob", "82347783", "bob@gmail.com")

fmt.Println(e1.GetId(), ":", e1.GetName(), ":", e1.GetPhone(), ":", e1.GetEmail())

}

## Inheritance by Composition in Go

As inheritance is not supported directly in Go, what programmers can do is use *composition* to get some of the effect of it.

Composition basically means putting one or more objects into another object like an attribute.

we can create a **struct** type that embeds other **struct** types.

For example, a computer is a composition of components like CPU, RAM, HDD, a Motherboard and other stuff.

Here is some example code demonstrating inheritance by composition in Go:

package computer

import "fmt"

type CPU struct {

architecture string

}

func (cpu \*CPU) SetArchitecture(arch string) {

cpu.architecture = arch

}

func (cpu \*CPU) GetArchitecture() string {

return cpu.architecture

}

type RAM struct {

size int

}

func (ram \*RAM) SetSize(size int) {

ram.size = size

}

func (ram \*RAM) GetSize() int {

return ram.size

}

type Motherboard struct {

category string

}

func (m \*Motherboard) SetCategory(cat string) {

m.category = cat

}

func (m \*Motherboard) GetCategory() string {

return m.category

}

type Computer struct {

cpu CPU

ram RAM

mboard Motherboard

}

func (c \*Computer) SetSpecification(cpu CPU, ram RAM, mboard Motherboard) {

c.cpu.SetArchitecture(cpu.GetArchitecture())

c.ram.SetSize(ram.GetSize())

c.mboard.SetCategory(mboard.GetCategory())

}

func (c \*Computer) ShowSpecification() {

fmt.Println("CPU: ", c.cpu.GetArchitecture(), ", RAM: ", c.ram.GetSize(), "GB, Motherboard: ", c.mboard.GetCategory())

}

//-------------------------------------

package main

import (

"fmt"

"oop\_prog/computer"

)

func main() {

cpu := computer.CPU{}

cpu.SetArchitecture("64-bit")

ram := computer.RAM{}

ram.SetSize(8)

mboard := computer.Motherboard{}

mboard.SetCategory("Micro ATX")

c1 := computer.Computer{}

c1.SetSpecification(cpu, ram, mboard)

c1.ShowSpecification()

}

The beauty of composition is that many simple object constructs can be combined to create a more complex object only when needed.

It means that developers can *lazy-create* objects only when needed. This keeps memory allocation to a minimum, leading to a lesser memory footprint of the program.

In our example above, we have created objects like CPU, RAM and Motherboard and defined their attributes and methods.

As we create a *Computer*, we can combine the individual objects into a more complex object.

## Polymorphism in Go

[Polymorphism](https://www.webopedia.com/definitions/polymorphism/) is another key feature of object oriented programming.

It provides the ability to write code through the implementation of types that can take on different behavior at runtime.

In Go, polymorphism is achieved through interfaces and interfaces only.

Once an interface implements a type, the functionality defined within it is open to any values of that type.

For example, the **io** package has an extensive set of interfaces and functions to stream data efficiently in our code.

Here is a quick example to illustrate how polymorphic behaviour is achieved through interfaces in Go:

package polymorph

import (

"fmt"

"math"

)

type Shape interface {

area()

}

type Rectangle struct {

X1, Y1, X2, Y2 float64

}

type Circle struct {

Xc, Yc, Radius float64

}

func (r \*Rectangle) area() {

fmt.Println("Rectangle Area : ", (r.X2-r.X1)\*(r.Y2-r.Y1))

}

func (c \*Circle) area() {

fmt.Println("Circle Area: ", math.Pi\*math.Pow(c.Radius, 2))

}

func GetArea(s Shape) {

s.area()

}

//--------------------------------------------------

package main

import "oop\_prog/polymorph"

func main() {

r := polymorph.Rectangle{10, 10, 20, 20}

polymorph.GetArea(&r)

c := polymorph.Circle{5, 5, 30}

polymorph.GetArea(&c)

}