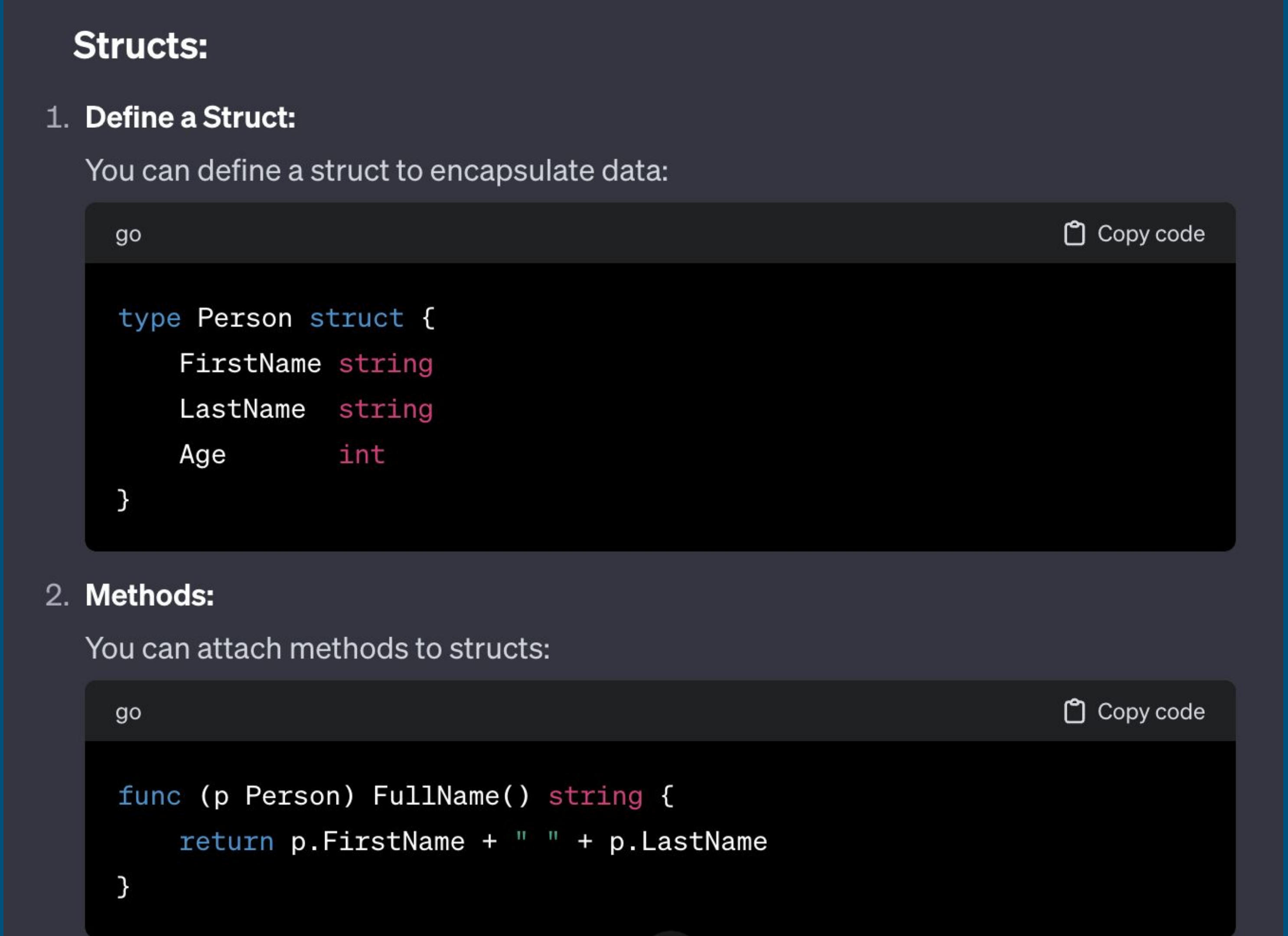
Week 2, Go

Azamat Serek, PhD, Assist.Prof.

Struct



Structs:

1. **Define a Struct:**

   You can define a struct to encapsulate data:

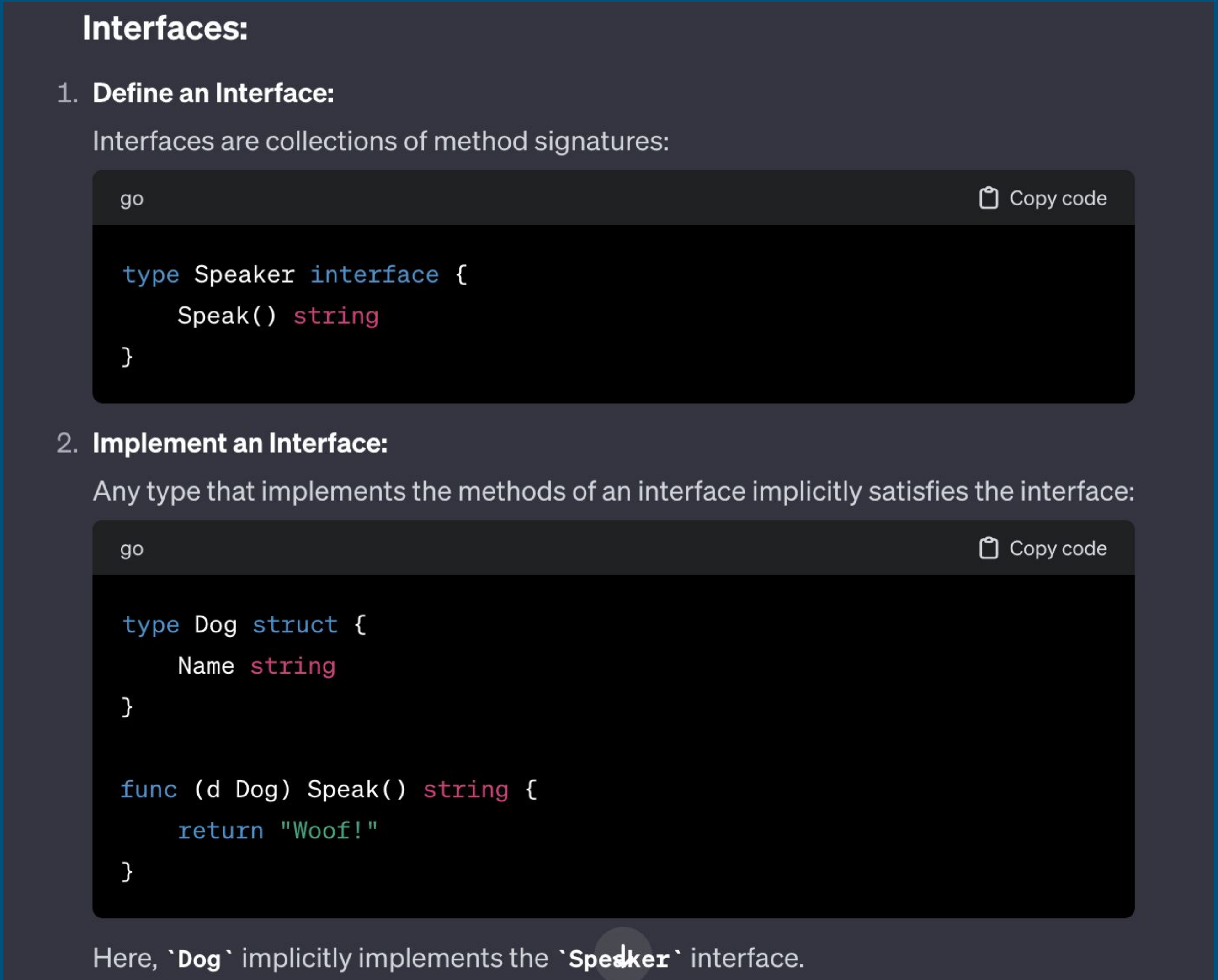
```go
type Person struct {
    FirstName string
    LastName  string
    Age       int
}
```

2. **Methods:**

   You can attach methods to structs:

```go
func (p Person) FullName() string {
    return p.FirstName + " " + p.LastName
}
```

Interfaces



Interfaces:

1. **Define an Interface:**

   Interfaces are collections of method signatures:

```go
type Speaker interface {
    Speak() string
}
```

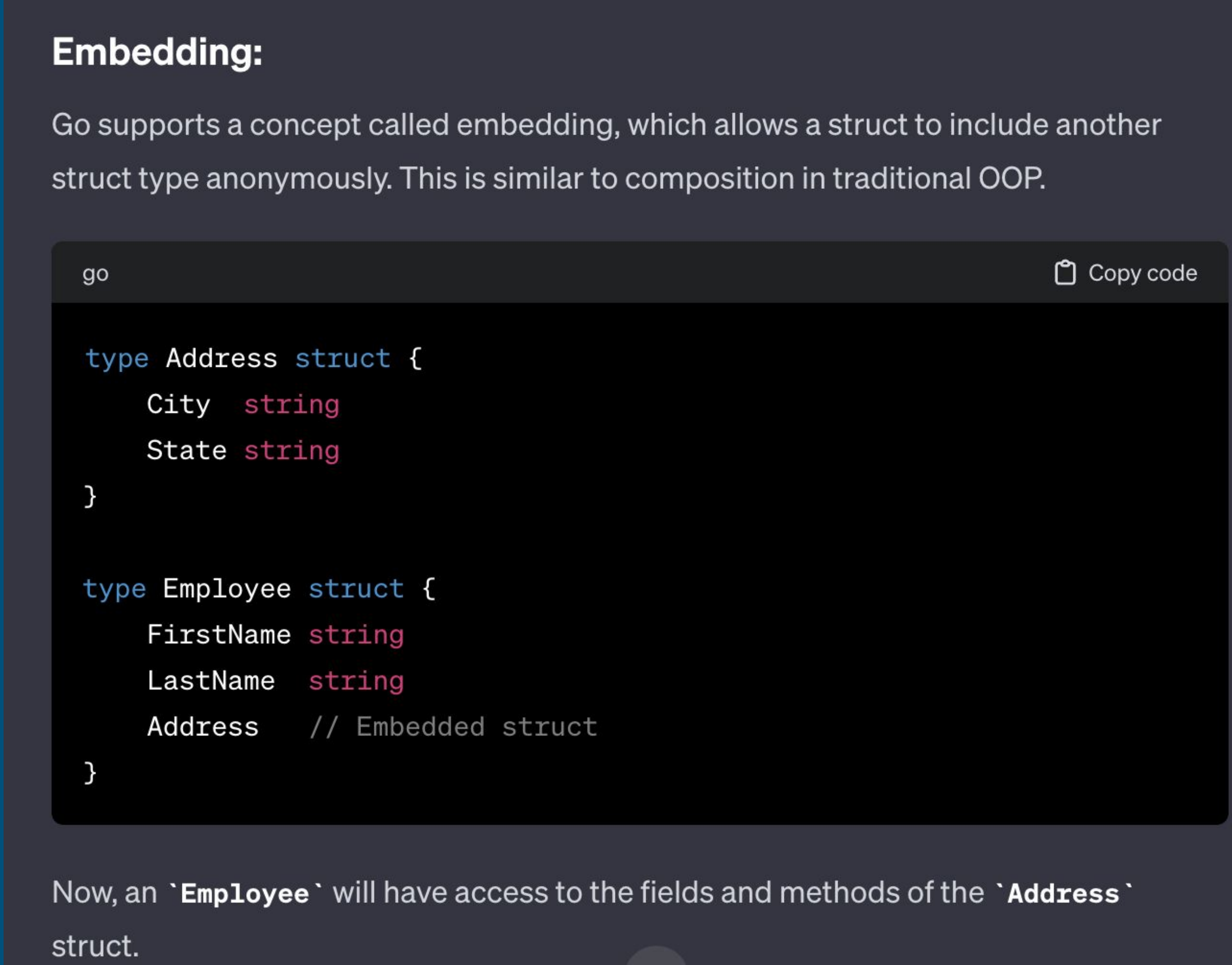
2. **Implement an Interface:**

   Any type that implements the methods of an interface implicitly satisfies the interface:

```go
type Dog struct {
    Name string
}

func (d Dog) Speak() string {
    return "Woof!"
}
```

Here, `Dog` implicitly implements the `Speaker` interface.

Embed



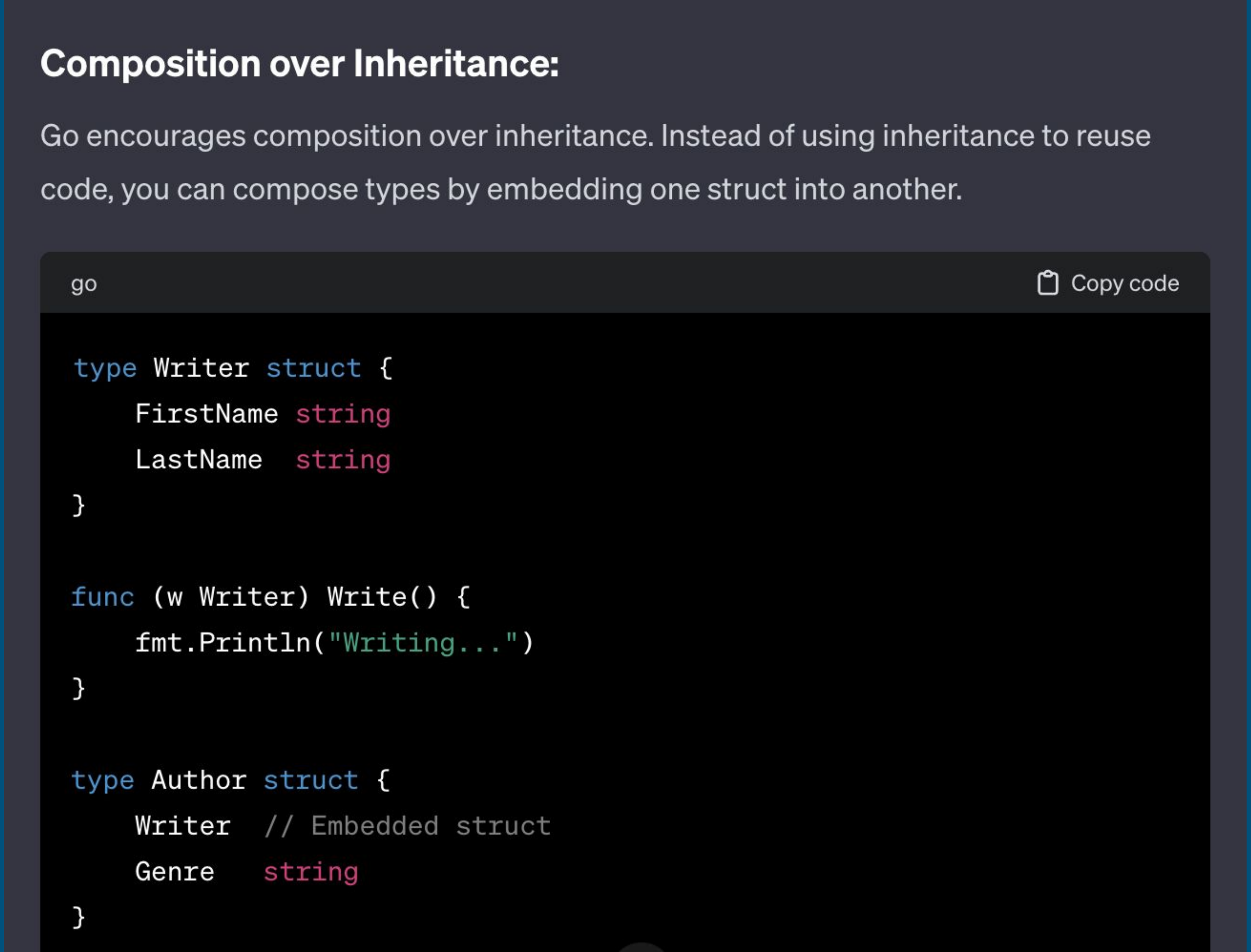
Embedding:

Go supports a concept called embedding, which allows a struct to include another struct type anonymously. This is similar to composition in traditional OOP.

```go
type Address struct {
    City  string
    State string
}

type Employee struct {
    FirstName string
    LastName  string
    Address   // Embedded struct
}
```

Now, an `Employee` will have access to the fields and methods of the `Address` struct.

Composition

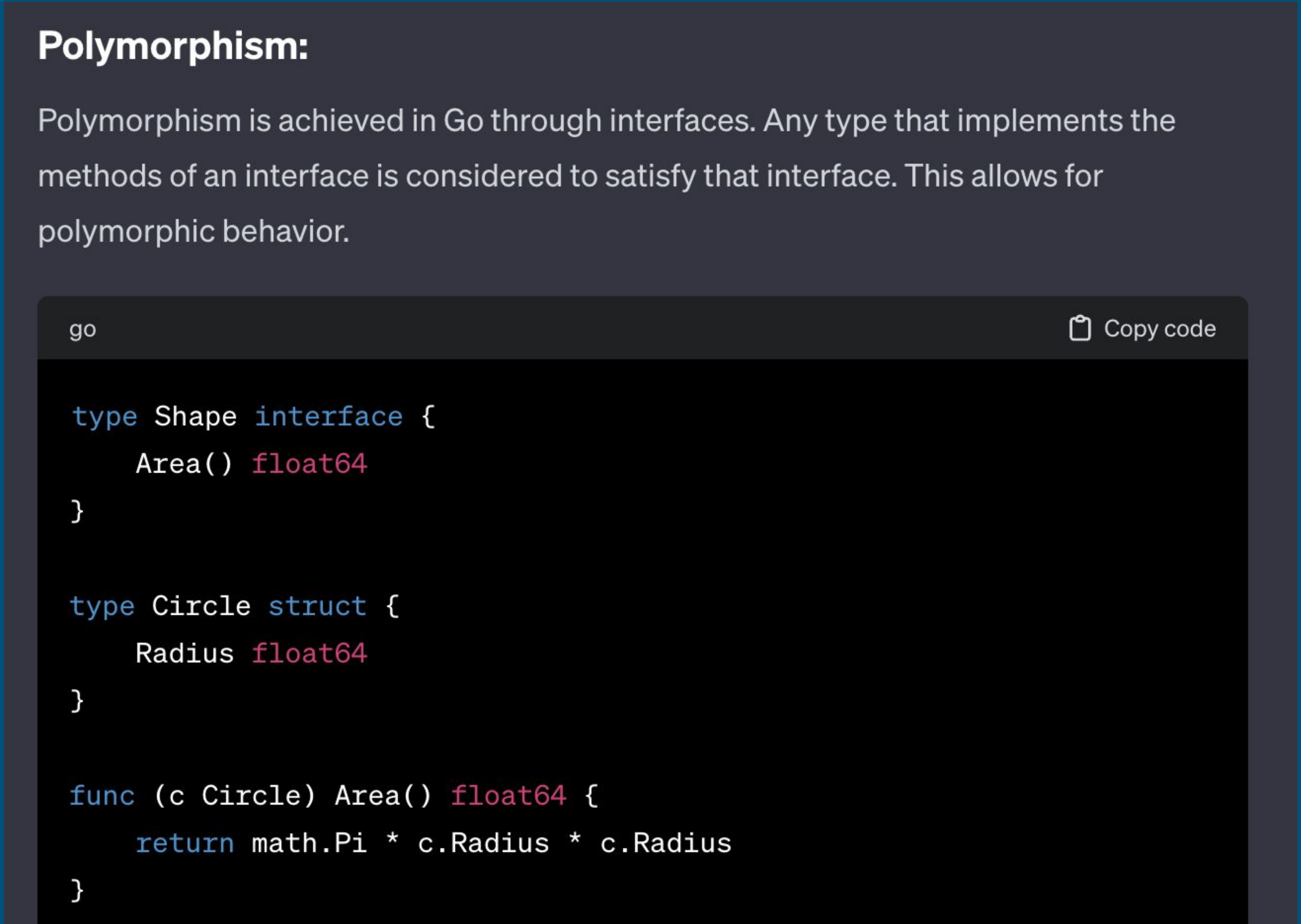


Composition over Inheritance:

Go encourages composition over inheritance. Instead of using inheritance to reuse code, you can compose types by embedding one struct into another.

```go
type Writer struct {
    FirstName string
    LastName  string
}

func (w Writer) Write() {
    fmt.Println("Writing...")
}

type Author struct {
    Writer  // Embedded struct
    Genre   string
}
```

Poly

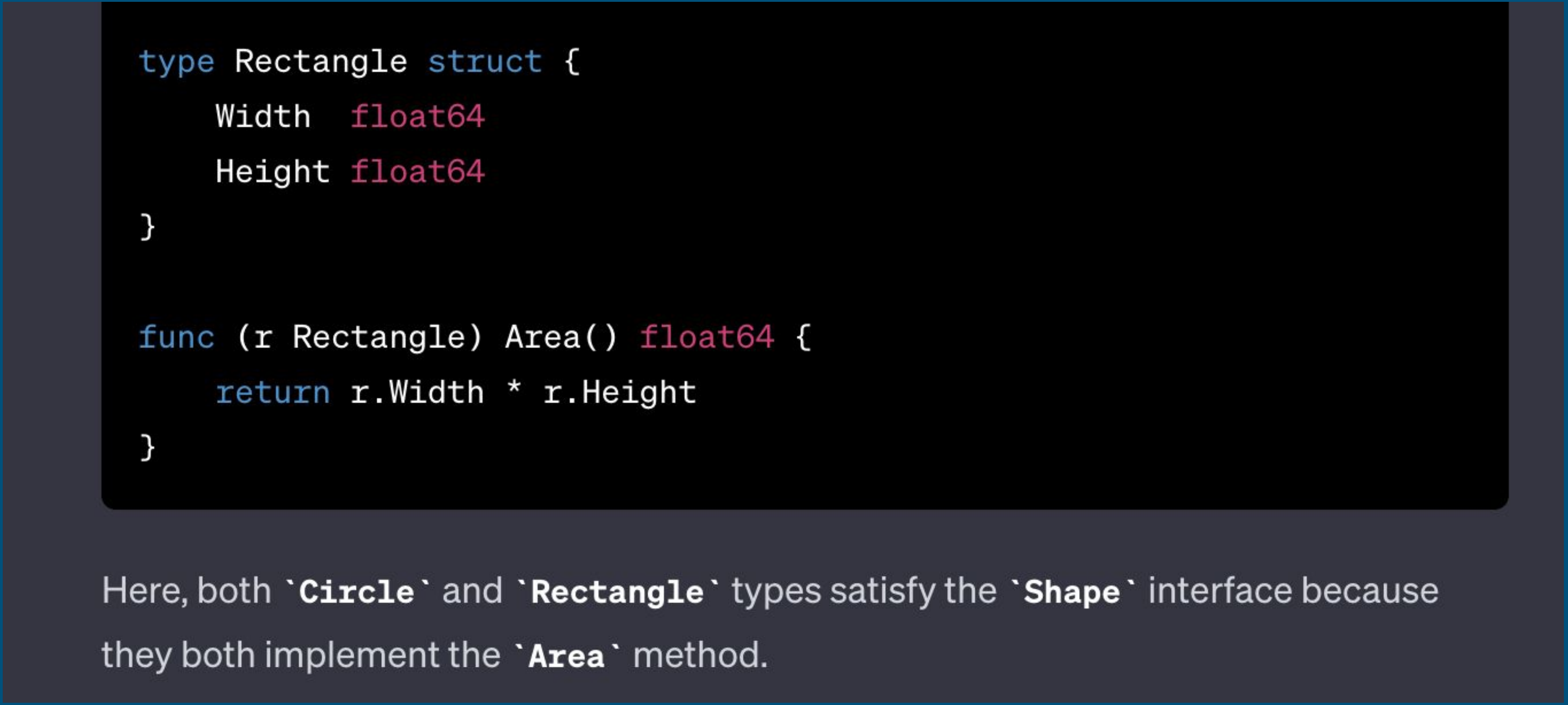


Polymorphism:

Polymorphism is achieved in Go through interfaces. Any type that implements the methods of an interface is considered to satisfy that interface. This allows for polymorphic behavior.

```go
type Shape interface {
    Area() float64
}

type Circle struct {
    Radius float64
}

func (c Circle) Area() float64 {
    return math.Pi * c.Radius * c.Radius
}
```

Implementing interface method



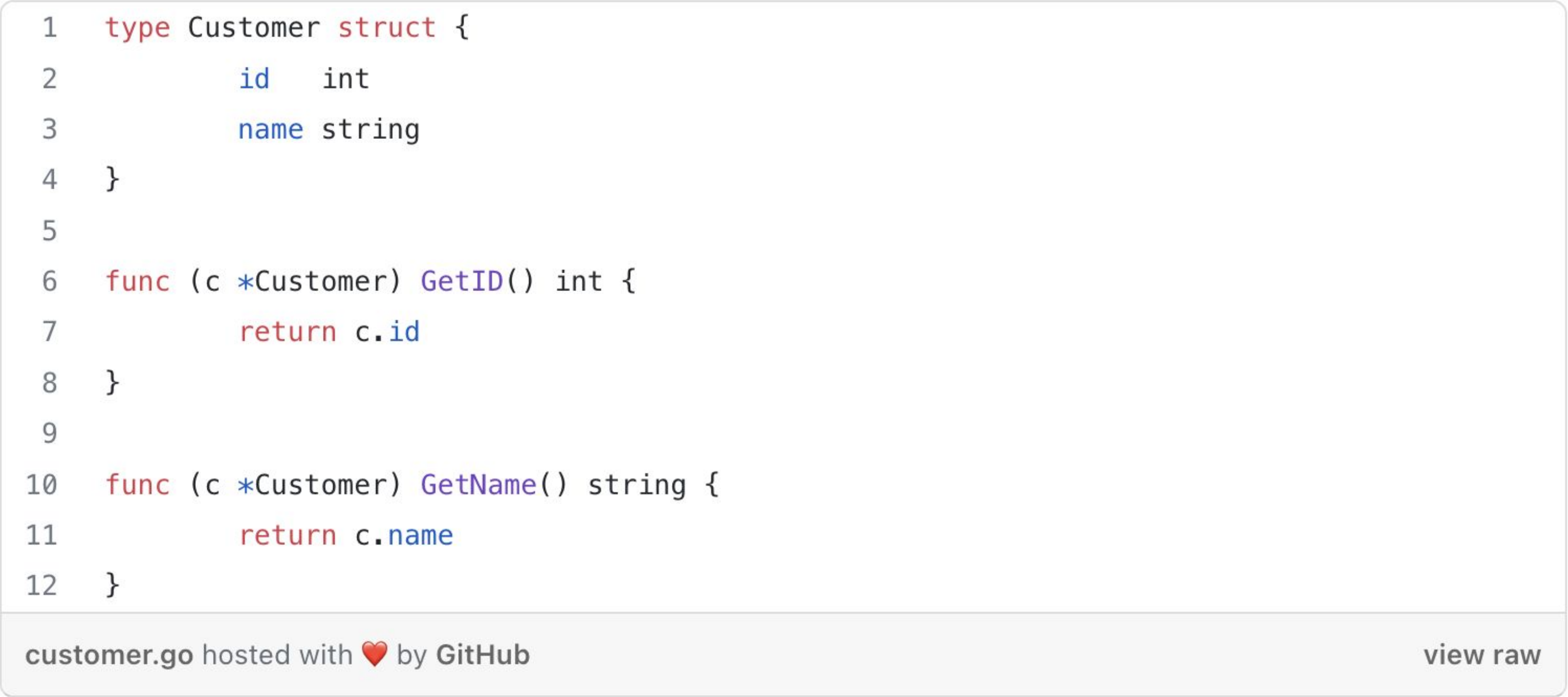
```
type Rectangle struct {
    Width  float64
    Height float64
}

func (r Rectangle) Area() float64 {
    return r.Width * r.Height
}
```

Here, both `Circle` and `Rectangle` types satisfy the `Shape` interface because they both implement the `Area` method.

Encapsulation

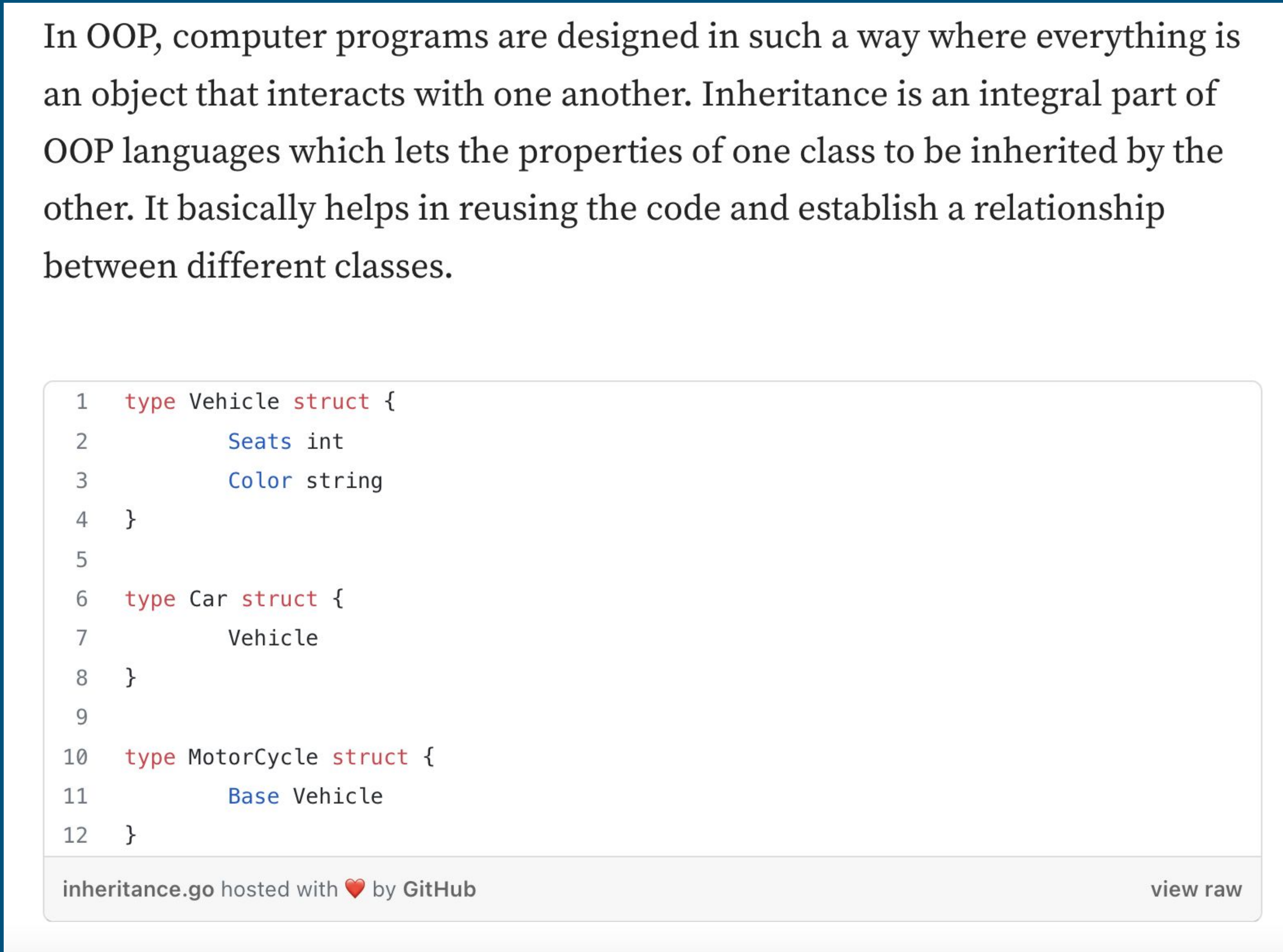
- Encapsulation keeps data safe from external interferences. It is done in package level.
- In go, there exported and unexported fields.
- Lowercase means unexported, capital means exported.

Encapsulation



```go
type Customer struct {
	id   int
	name string
}

func (c *Customer) GetID() int {
	return c.id
}

func (c *Customer) GetName() string {
	return c.name
}
```

Inheritance



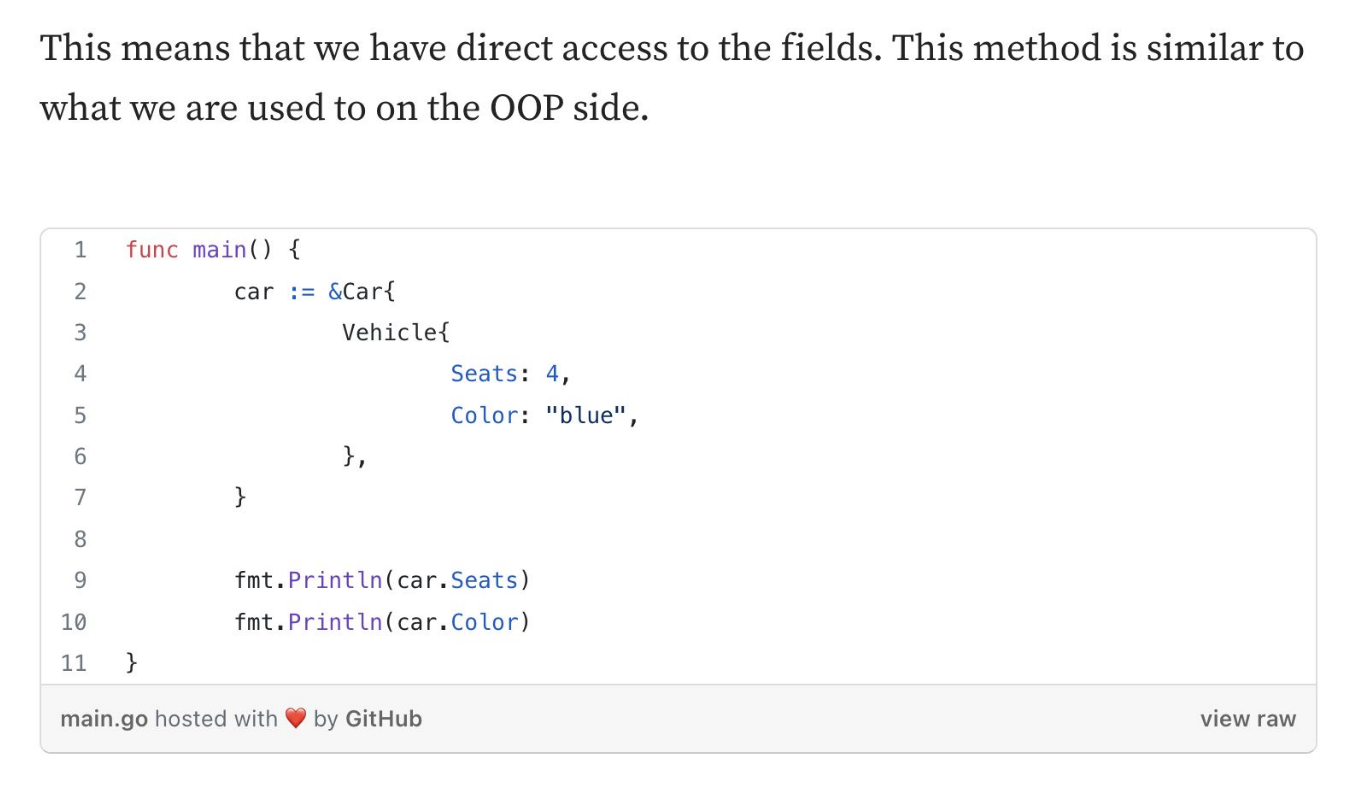
In OOP, computer programs are designed in such a way where everything is an object that interacts with one another. Inheritance is an integral part of OOP languages which lets the properties of one class to be inherited by the other. It basically helps in reusing the code and establish a relationship between different classes.

```go
type Vehicle struct {
        Seats int
        Color string
}

type Car struct {
        Vehicle
}

type MotorCycle struct {
        Base Vehicle
}
```

inheritance.go hosted with ❤️ by GitHub

view raw

Direct access to fields

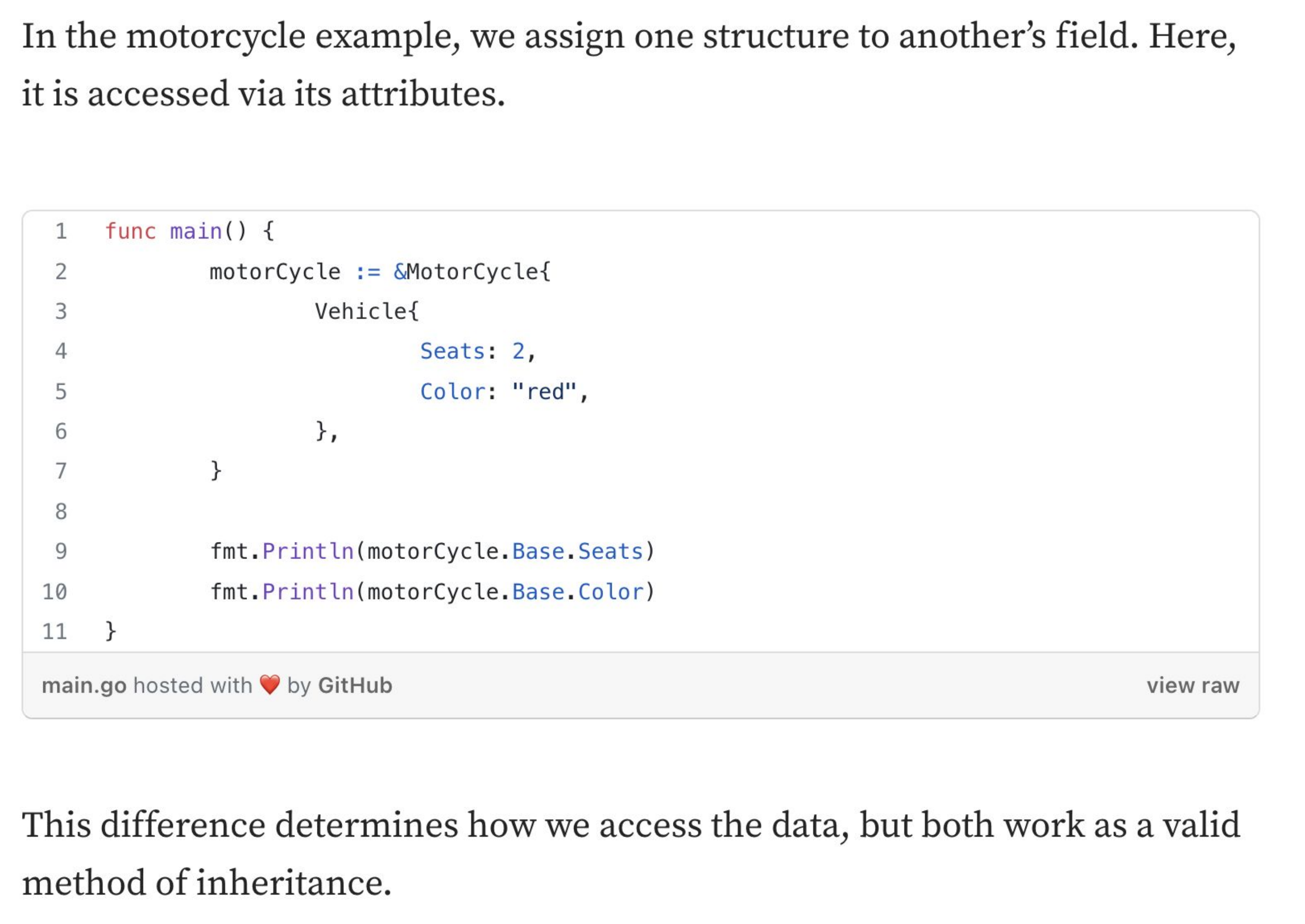


This means that we have direct access to the fields. This method is similar to what we are used to on the OOP side.

```go
func main() {
	car := &Car{
		Vehicle{
			Seats: 4,
			Color: "blue",
		},
	}

	fmt.Println(car.Seats)
	fmt.Println(car.Color)
}
```

main.go hosted with ❤️ by GitHub view raw

Motor Cycle example



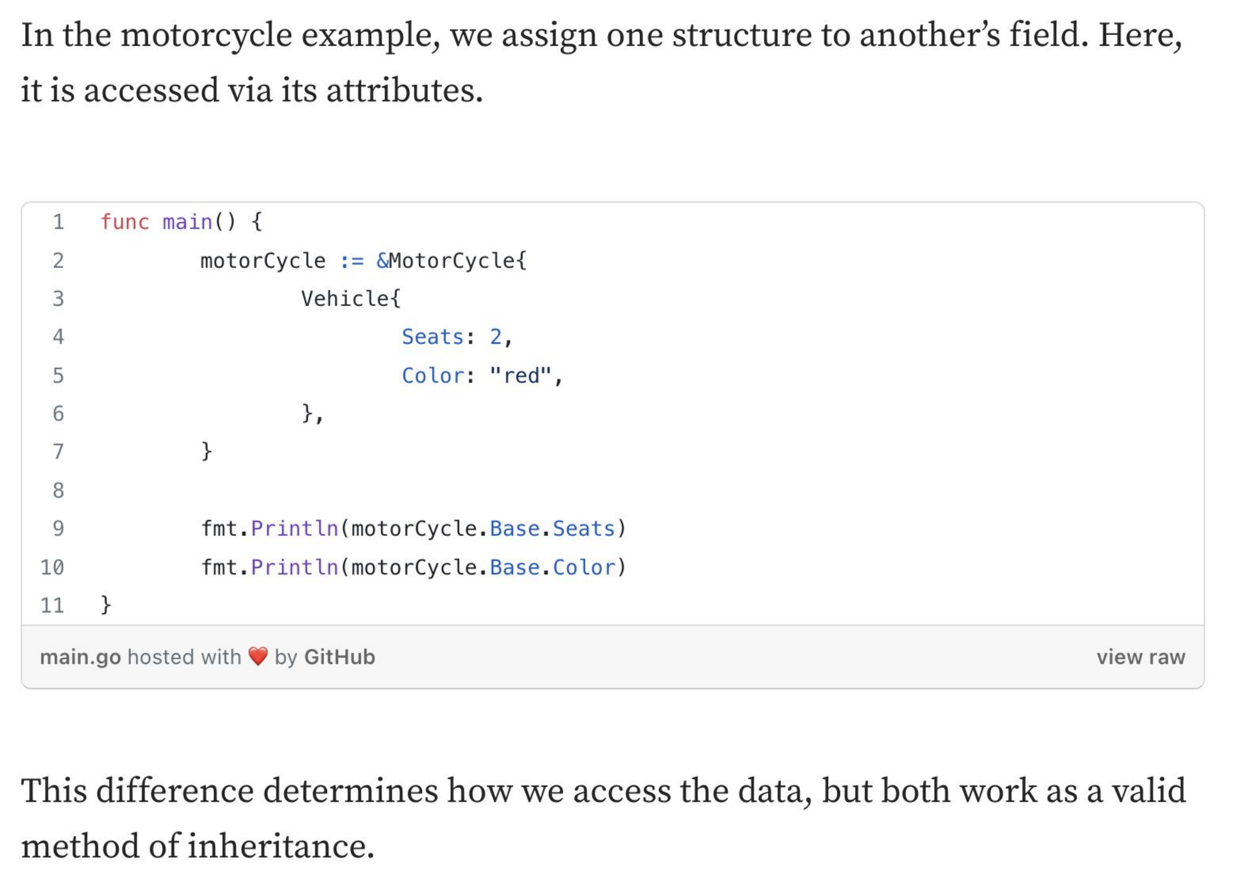
In the motorcycle example, we assign one structure to another's field. Here, it is accessed via its attributes.

```
func main() {
        motorCycle := &MotorCycle{
                Vehicle{
                        Seats: 2,
                        Color: "red",
                },
        }

        fmt.Println(motorCycle.Base.Seats)
        fmt.Println(motorCycle.Base.Color)
}
```

main.go hosted with ❤ by GitHub view raw

This difference determines how we access the data, but both work as a valid method of inheritance.

Constructor

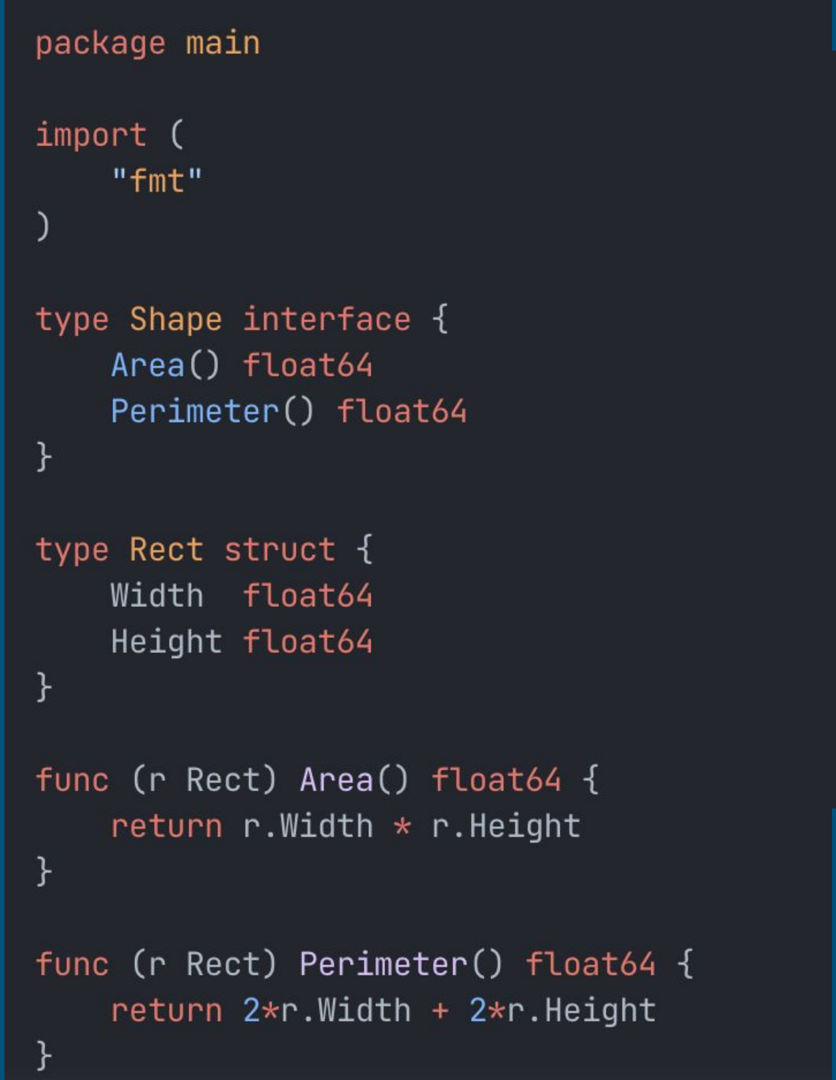


In the motorcycle example, we assign one structure to another’s field. Here, it is accessed via its attributes.

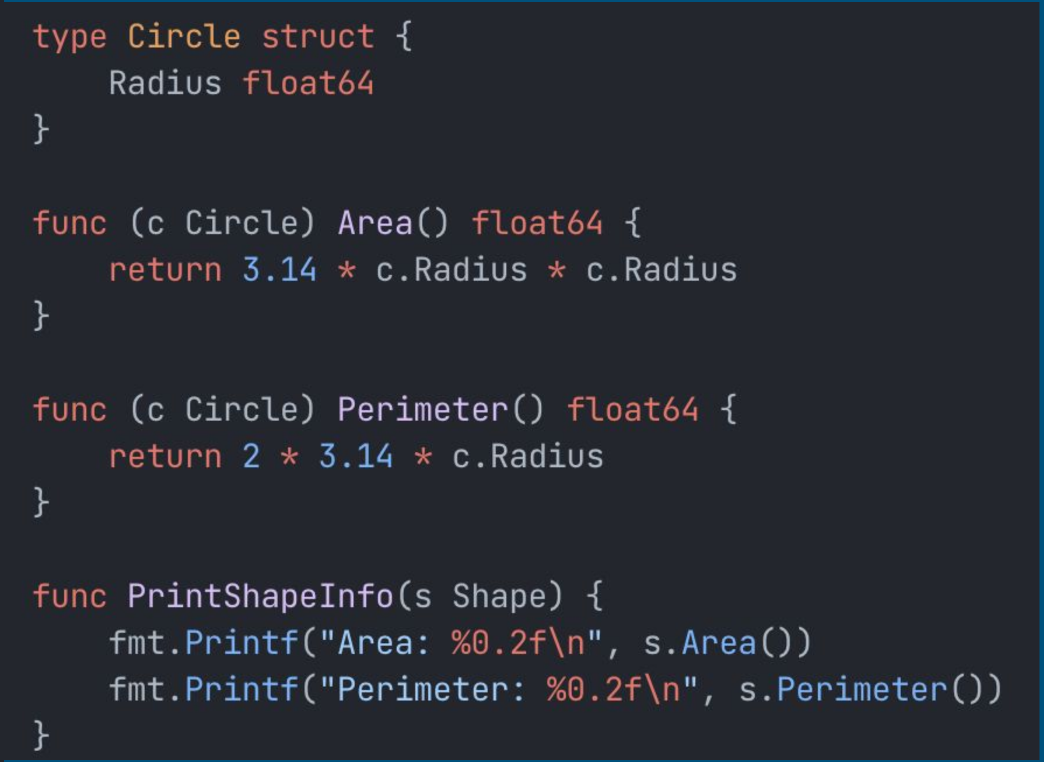
```go
func main() {
        motorCycle := &MotorCycle{
                Vehicle{
                        Seats: 2,
                        Color: "red",
                },
        }

        fmt.Println(motorCycle.Base.Seats)
        fmt.Println(motorCycle.Base.Color)
}
```

main.go hosted with ❤ by GitHub view raw

This difference determines how we access the data, but both work as a valid method of inheritance.

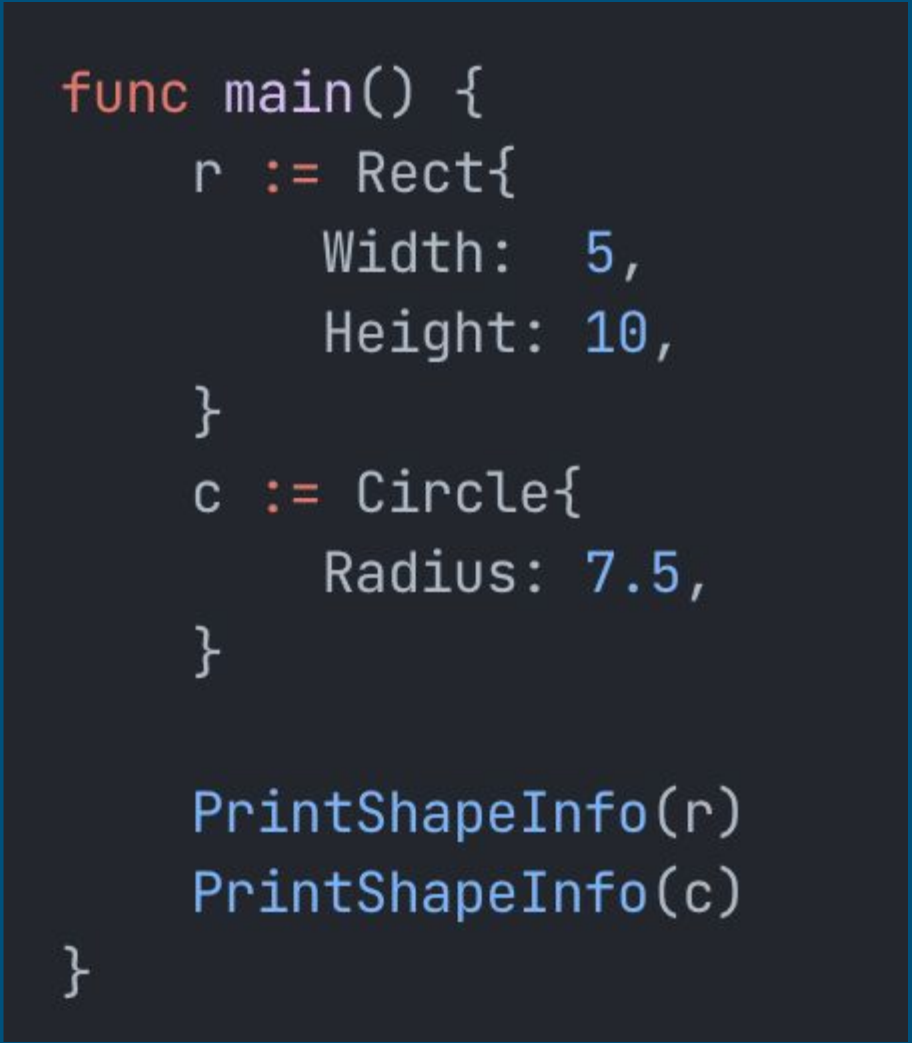


```go
package main

import (
    "fmt"
)

type Shape interface {
    Area() float64
    Perimeter() float64
}

type Rect struct {
    Width  float64
    Height float64
}

func (r Rect) Area() float64 {
    return r.Width * r.Height
}

func (r Rect) Perimeter() float64 {
    return 2*r.Width + 2*r.Height
}
```



```go
type Circle struct {
    Radius float64
}

func (c Circle) Area() float64 {
    return 3.14 * c.Radius * c.Radius
}

func (c Circle) Perimeter() float64 {
    return 2 * 3.14 * c.Radius
}

func PrintShapeInfo(s Shape) {
    fmt.Printf("Area: %0.2f\n", s.Area())
    fmt.Printf("Perimeter: %0.2f\n", s.Perimeter())
}
```

Execution



```go
func main() {
    r := Rect{
        Width:  5,
        Height: 10,
    }
    c := Circle{
        Radius: 7.5,
    }

    PrintShapeInfo(r)
    PrintShapeInfo(c)
}
```

Practice exercise

Design struct system for university. Apply the studied principles.

References

- https://medium.com/@canoguz/object-oriented-programming-in-go-e50f8fe4a620
- https://thegodev.com/is-golang-oop/
- https://www.digitalocean.com/community/tutorials/how-to-write-packages-in-go