Composition vs Inheritance

This explains details of Composition in Go and how it can be used as an alternative to Inheritance in Go.

WE'LL COVER THE FOLLOWING

Composition as an Alternative to Inheritance

Coming from an OOP background a lot of us are used to inheritance, something that isn't supported by Go. Instead you have to think in terms of *composition* and *interfaces*. In the previous lessons, we learnt about structs in Go, and that is what we will be using for composition.

The Go team wrote a short but good segment on this topic.

Composition

Composition (or embedding) is a well understood concept for most OOP programmers and Go supports it, here is an example of the problem it's solving:

Environment Variables

Key:	Value:
GOPATH	/go

```
package main

import "fmt"

type User struct {
        Id       int
        Name     string
        Location string
}

//type Player with one additional attribute


type Player struct {
        Id       int
        Name     string
        Location string
        GameId   int
}

func main() {
        p := Player{}
        p.Id = 42
        p.Name = "Matt"
        p.Location = "LA"
        p.GameId = 90404
        fmt.Printf("%+v", p) // the value in a default format when printing structs,
                        // the plus flag (%+v) adds field names
}
```

The above example demonstrates a classic OOP challenge, our `Player` struct has the same fields as the `User` struct but it also has a `GameId` field. Having to duplicate the field names isn't a big deal, but it can be simplified by composing our struct.

Environment Variables

Key:	Value:
GOPATH	/go

```
type User struct {
        Id             int
        Name, Location string
}

type Player struct {
        User //user will contain all the required attributes
        GameId int
}
```

We can initialize a new variable of type `Player` two different ways.

Using the dot notation to set the fields:

Environment Variables

Key: Value:

GOPATH /go

```
package main

import "fmt"

type User struct {
        Id             int
        Name, Location string
}

type Player struct {
        User
        GameId int
}

func main() {
        p := Player{} //initializing
        p.Id = 42
        p.Name = "Matt"
        p.Location = "LA"
        p.GameId = 90404
        fmt.Printf("%+v", p)
}
```

The other option is to use a struct literal:

Environment Variables

Key: Value:

GOPATH /go

```
package main

import "fmt"

type User struct {
        Id             int
        Name, Location string
}

type Player struct {
        User
        GameId int
}

func main() {
        p := Player{
                User{Id: 42, Name: "Matt", Location: "LA"},
                90404,
        }
        fmt.Printf(
                "Id: %d, Name: %s, Location: %s, Game id: %d\n",
                p.Id, p.Name, p.Location, p.GameId)
        // Directly set a field defined on the Player struct

        p.Id = 11
        fmt.Printf("%+v", p)
}
```

When using a struct literal with an implicit composition, we can't just pass the composed fields. We instead need to pass the types composing the struct. Once set, the fields are directly available.

Because our struct is composed of another struct, the methods on the `User` struct is also available to the `Player`. Let's define a method to show that behavior:

Environment Variables

Key:	Value:
GOPATH	/go

```
package main

import "fmt"

type User struct {
        Id             int
        Name, Location string
}

func (u *User) Greetings() string {
        return fmt.Sprintf("Hi %s from %s",
                u.Name, u.Location)
}

type Player struct {
        User
        GameId int
}

func main() {
        p := Player{}
        p.Id = 42
        p.Name = "Matt"
        p.Location = "LA"
        fmt.Println(p.Greetings())
}
```

As you can see this is a very powerful way to build data structures but it’s even more interesting when thinking about it in the context of interfaces. By composing one of your structure with one implementing a given interface, your structure automatically implements the interface.

Here is another example, this time we will look at implementing a `Job` struct that can also behave as a logger.

Here is the explicit way:

Environment Variables

Key:	Value:
GOPATH	/go

```
package main

import (
	"log"
	"os"
)

type Job struct {
	Command string
	Logger  *log.Logger
}

func main() {
	job := &Job{"demo", log.New(os.Stdout, "Job: ", log.Ldate)}
	// same as
	// job := &Job{Command: "demo",
	//            Logger: log.New(os.Stderr, "Job: ", log.Ldate)}
	job.Logger.Print("test")
}
```

Our `Job` struct has a field called `Logger` which is a pointer to another type (log.Logger)

When we initialize our value, we set the logger so we can then call its `Print` function by chaining the calls: `job.Logger.Print()`

But Go lets you go even further and use implicit composition. We can skip defining the field for our logger and now all the methods available on a pointer to `log.Logger` are available from our struct:

Environment Variables

Key:	Value:
GOPATH	/go

```
package main

import (
	"log"
	"os"
)

type Job struct {
	Command string
	*log.Logger
}

func main() {
	job := &Job{"demo", log.New(os.Stdout, "Job: ", log.Ldate)}
	job.Print("starting now...")
}
```

Note that you still need to set the logger and that’s often a good reason to use a constructor (custom constructors are used when you need to set a structure before using a value, see Custom constructors. What is really nice with the implicit composition is that it allows to easily and cheaply make your structs implement interfaces. Imagine that you have a function that takes variables implementing an interface with the `Print` method. By adding `*log.Logger` to your struct (and initializing it properly), your struct is now implementing the interface without you writing any custom methods.

Test Yourself!

Quiz on Composition

Q

True or False: When using a struct literal with an implicit composition, we can pass the composed fields.

Check Answers

That concludes the lesson on how composition works. The following section contains an exercise based on this example to further enhance your understanding of the concept of composition.