OOP in Go

Miti Bhat

Intro & Objects

- Rather than structure programs as code and data, an object-oriented system integrates the two using the concept of an “object”. An object is an abstract data type that has state (data) and behavior (code).
- Go doesn’t have a something called ‘object’, but ‘object’ is only a word that connotes a meaning. It’s the meaning that matters, not the term itself.
- While Go doesn’t have a type called ‘object’ it does have a type that matches the same definition of a data structure that integrates both code and behavior. In Go this is called a ‘struct’.
- A ‘struct’ is a kind of type which contains named fields and methods.
- Demo: oop1.go

Inheritance And Polymorphism

- There are a few different approaches to defining the relationships between objects. While they differ quite a bit from each other, all share a common purpose as a mechanism for code reuse.
- Inheritance
- Multiple Inheritance
- Subtyping(Polymorphism)
- Object composition

Single & Multiple Inheritance

- Inheritance is when an object is based on another object, using the same implementation.
- Two different implementations of inheritance exist.
- The fundamental distinction between them is whether an object can inherit from a single object or from multiple objects. This is a seemingly small distinction, but with large implications.
- The hierarchy in single inheritance is a tree, while in multiple inheritance it is a lattice.
- Single inheritance languages include PHP, C#, Java and Ruby. Multiple inheritance languages include Perl, Python and C++.

Subtyping (Polymorphism)

- In some languages subtyping and inheritance are so interwoven that this may seem redundant to the previous section if your particular perspective comes from a language where they are tightly coupled.
- Subtyping establishes an is-a relationship, while inheritance only reuses implementation. Subtyping defines a semantic relationship between two (or more) objects. Inheritance only defines a syntactic relationship.

Object Composition

- Object composition is where one object is defined by including other objects. Rather than inheriting from them, the object contains them. Unlike the is-a relationship of subtyping, object composition defines **a has-a relationship**.

Inheritance In Go

- Go is intentionally designed without any inheritance at all. This does not mean that objects (struct values) do not have relationships, instead the Go authors have chosen to use a alternative mechanism to connote relationships. To many encountering Go for the first time this decision may appear as if it cripples Go. In reality it is one of the nicest properties of Go and it resolves decade old issues and arguments around inheritance.

Object Composition In Go

- The mechanism Go uses to implement the principle of object composition is called embedded types. Go permits you to embed a struct within a struct giving them a has-a relationship.
- An good example of this would be the relationship between a Person and an Address.
- Demo: oop2.go

Pseudo Subtyping In Go

- The pseudo is-a relationship works in a similar and intuitive way. To extend our example above. Let's use the following statement. A Person can talk. A Citizen is a Person therefore a citizen can Talk.
- Demo: oop3.go

Why Anonymous Fields Are Not Proper Subtyping

- There are two distinct reasons why this cannot be considered proper subtyping.
- 1. The Anonymous Fields Are Still Accessible As If They Were Embedded.
- This isn't necessarily a bad thing. One of the issues with multiple inheritance is that languages are often non obvious and sometimes even ambiguous as to which methods are used when identical methods exist on more than one parent class.
- With Go you can always access the individual methods through a property which has the same name as the type.
- In reality when you are using Anonymous fields, Go is creating an accessor with the same name as your type.
- Demo: oop4.go
- 2. True Subtyping Becomes The Parent
- If this was truly subtyping then the anonymous field would cause the outer type to become the inner type. In Go this is simply not the case. The two types remain distinct. The following example illustrates this:
- Demo: oop5.go

True Subtyping In Go

- As we wrote above, subtyping is the is-a relationship. In Go each type is distinct and nothing can act as another type, but both can adhere to the same interface. Interfaces can be used as input and output of functions (and methods) and consequently establish an is-a relationship between types.
- Adherence to an interface in Go is defined not through a keyword like ‘using’ but through the actual methods declared on a type. In Efficient Go it refers to this relationship as ‘if something can do this, then it can be used here.’ This is very important because it enables one to create an interface that types defined in external packages can adhere to.
- Continuing with the example from above, we add a new function, SpeakTo and modify the main function to try to SpeakTo a Citizen and a Person.
- Demo: oop6.go//cannot use &c (type *Citizen) as type *Person in argument to SpeakTo

Subtyping in Go

- There are two critical points to make about subtyping in Go:
- We can use Anonymous fields to adhere to an interface. We can also adhere to many interfaces. By using Anonymous fields along with interfaces we are very close to true subtyping.
- Go does provide proper subtyping capabilities, but only in the using of a type. Interfaces can be used to ensure that a variety of different types can all be accepted as input into a function, or even as a return value from a function, but in reality they retain their distinct types. This is clearly displayed in the main function where we cannot set Name on Citizen directly because Name isn't actually a property of Citizen, it's a property of Person and consequently not yet present during the initialization of a Citizen.

Welcome to the new ‘object’-less OO programming model.

- The fundamental concepts of object orientation are alive and well in Go in spite of some terminology differences. The terminology differences are essential as the mechanisms used are in fact different from most object oriented languages.
- Go utilizes structs as the union of data and logic. Through composition, has-a relationships can be established between Structs to minimize code repetition while staying clear of the brittle mess that is inheritance. Go uses interfaces to establish is-a relationships between types without unnecessary and counteractive declarations.

Interfaces in Go

- Polymorphism in Go is achieved with the help of interfaces. As we have already discussed, interfaces can be implicitly implemented in Go. A type implements an interface if it provides definitions for all the methods declared in the interface.
- Any type which defines all the methods of an interface is said to implicitly implement that interface.
- A variable of type interface can hold any value which implements the interface. This property of interfaces is used to achieve polymorphism in Go.
- For example lets assume that this imaginary organisation has income from two kinds of projects viz. *fixed billing* and *time and material*. The net income of the organization is calculated by the sum of the incomes from these projects.

Interfaces in Go

```
type Income interface { calculate() int source() string }
```

The Income interface defined above contains two methods calculate() which calculates and returns the income from the source and source() which returns the name of the source.

Next let's define a struct for FixedBilling project type.

```
type FixedBilling struct {
 projectName string
 biddedAmount int
 }
```

Interfaces in Go

- The FixedBilling project has two fields projectName which represents the name of the project and biddedAmount which is the amount that the organisation has bid for the project.

- The TimeAndMaterial struct will represent projects of Time and Material type.

```
type TimeAndMaterial struct {
 projectName string
 noOfHours int
 hourlyRate int
 }
```

Interfaces in Go

- The TimeAndMaterial struct has three fields names projectName, noOfHours and hourlyRate.
- Next step would be to define methods on these struct types which calculate and return the actual income and source of income.

```
func (fb FixedBilling) calculate() int {
return fb.biddedAmount
}
func (fb FixedBilling) source() string {
return fb.projectName
}
 func (tm TimeAndMaterial) calculate() int {
return tm.noOfHours * tm.hourlyRate
}
func (tm TimeAndMaterial) source() string {
 return tm.projectName
 }
```

In the case of FixedBilling projects, the income is the just the amount bid for the project. Hence we return this from the calculate() method of FixedBilling type.
In the case of TimeAndMaterial projects, the income is the product of the noOfHours and hourlyRate. This value is returned from the calculate() method with receiver type TimeAndMaterial.
We return the name of the project as source of income from the source() method.

Since both FixedBilling and TimeAndMaterial structs provide definitions for the calculate() and source() methods of the Income interface, both structs implement the Income interface.

Let's declare the calculateNetIncome function which will calculate and print the total income.

```
func calculateNetIncome(ic []Income) {
 var netincome int = 0
 for _, income := range ic {
 fmt.Printf("Income From %s = $%d\n", income.source(), income.calculate())
 netincome += income.calculate()
}
 fmt.Printf("Net income of organisation = $%d", netincome) }
```

- The calculateNetIncome function above accepts a slice of Income interfaces as argument. It calculates the total income by iterating over the slice and calling calculate() method on each of its items. It also displays the income source by calling source() method. Depending on the concrete type of the Income interface, different calculate() and source() methods will be called. Thus we have achieved polymorphism in the calculateNetIncome function.

- In the future if a new kind of income source is added by the organisation, this function will still calculate the total income correctly without a single line of code change :).

- The only part remaining in the program is the main function.

```
func main() {
 project1 := FixedBilling{
      projectName: "Project 1",
      biddedAmount: 5000
      }
project2 := FixedBilling{
      projectName: "Project 2",
      biddedAmount: 10000
      }
project3 := TimeAndMaterial{
      projectName: "Project 3",
      noOfHours: 160,
      hourlyRate: 25
      }
incomeStreams := []Income{project1, project2, project3}
calculateNetIncome(incomeStreams)
 }
```

In the main function above we have created three projects, two of type FixedBilling and one of type TimeAndMaterial. Next we create a slice of type Income with these 3 projects. Since each of these projects has implemented the Income interface, it is possible to add all the three projects to a slice of type Income. Finally we call calculateNetIncome function with this slice and it will display the various income sources and the income from them.

- Demo: polymorph.go

Adding a new income stream to the above program

- Let's say the organisation has found a new income stream through advertisements. Let's see how simple it is to add this new income stream and calculate the total income without making any changes to the calculateNetIncome function. This becomes possible because of polymorphism.

- Lets first define the Advertisement type and the calculate() and source() methods on the Advertisement type.

```
type Advertisement struct {
   adName string
   CPC int
   noOfClicks int
}
func (a Advertisement) calculate() int {
 return a.CPC * a.noOfClicks
 }
func (a Advertisement) source() string {
 return a.adName
 }
```

The Advertisement type has three fields adName, CPC (cost per click) and noOfClicks (number of clicks). The total income from ads is the product of CPC and noOfClicks.

Let's modify the main function a little to include this new income stream.
We have created two ads namely bannerAd and popupAd. The incomeStreams slice includes the two ads we just created.

```
func main() {
project1 := FixedBilling{
projectName: "Project 1",
 biddedAmount: 5000
}
project2 := FixedBilling{
projectName: "Project 2",
 biddedAmount: 10000
}
project3 := TimeAndMaterial{
projectName: "Project 3",
 noOfHours: 160,
 hourlyRate: 25
}
bannerAd := Advertisement{adName: "Banner Ad", CPC: 2, noOfClicks: 500}
popupAd := Advertisement{adName: "Popup Ad", CPC: 5, noOfClicks: 750}
incomeStreams := []Income{project1, project2, project3, bannerAd, popupAd}
 calculateNetIncome(incomeStreams) }
```

- Demo: polymorph2.go

- We did not make any changes to the calculateNetIncome function though we added a new income stream. It just worked because of polymorphism. Since the new Advertisement type also implemented the Income interface, we were able to add it to the incomeStreams slice. The calculateNetIncome function also worked without any changes as it was able to call the calculate() and source() methods of the Advertisement type.

Composition Instead of Inheritance - OOP in Go

- Go does not support inheritance, however it does support composition. The generic definition of composition is "put together". One example of composition is a car. A car is composed of wheels, engine and various other parts.

Composition by embedding structs

- Composition can be achieved in Go is by embedding one struct type into another.
- A blog post is a perfect example of composition. Each blog post has a title, content and author information. This can be perfectly represented using composition. In the next steps of this tutorial we will learn how this is done.

```
type author struct {
 firstName string
 lastName string
 bio string
 }
 func (a author) fullName() string {
 return fmt.Sprintf("%s %s", a.firstName, a.lastName)
 }
```

- In the above code snippet, we have created a author struct with fields firstName, lastName and bio. We have also added a method fullName() with the author as receiver type and this returns the full name of the author.
- The next step would be to create the post struct.

```
type post struct {
 title string
 content string
 author
}
func (p post) details() {
 fmt.Println("Title: ", p.title)
 fmt.Println("Content: ", p.content)
 fmt.Println("Author: ", p.author.fullName())
 fmt.Println("Bio: ", p.author.bio)
 }
```

The post struct has fields title, content. It also has an embedded anonymous field author. This field denotes that post struct is composed of author. Now post struct has access to all the fields and methods of the author struct. We have also added details() method to the post struct which prints the title, content, fullName and bio of the author.
Whenever one struct field is embedded in another, Go gives us the option to access the embedded fields as if they were part of the outer struct. This means that p.author.fullName() in line no. 11 of the above code can be replaced with p.fullName(). Hence the details() method can be rewritten as below,

```
func (p post) details() {
 fmt.Println("Title: ", p.title)
 fmt.Println("Content: ", p.content)
 fmt.Println("Author: ", p.fullName())
 fmt.Println("Bio: ", p.bio)
 }
```

- Now that we have the author and the post structs ready, let's finish this program by creating a blog post.

Embedding slice of structs

- We can take this example one step further and create a website using a slice of blog posts :).

- Let's define the website struct first. Please add the following code above the main function of the existing program and run it.

```
type website struct {
 []post
 }
 func (w website) contents() {
 fmt.Println("Contents of Website\n")
 for _, v := range w.posts {
 v.details() fmt.Println()
 }
 }
```

```
main.go:31:9: syntax error: unexpected [, expecting field name or embedded type
```

- This error points to the embedded slice of structs []post . The reason is that it is not possible to anonymously embed a slice. A field name is required. So let's fix this error and make the compiler happy.

```
type website struct { posts []post }
```

Added the field name posts to the slice of post []post.

Now let's modify the main function and create a few posts for our new website.

Demo: composition3.go

In the main function above, we have created an author author1 and three posts post1, post2 and post3. Finally we have created the website w by embedding these 3 posts and displayed the contents in the next line.

Thank You