blog code talks me

Is Go An Object Oriented Language?

To truly understand what it means to be ‘object-oriented’ you need to look back at the origination of the concept. The first object oriented language, simula, emerged in the 1960s. It introduced objects, classes, inheritance and subclasses, virtual methods, coroutines, and a lot more. Perhaps most importantly, it introduced a paradigm shift of thinking of data and logic as completely independent.

While you many not be familiar with Simula, you are no doubt familiar with languages that refer to it as their inspiration including Java, C++, C# & Smalltalk, which in turn have been the inspiration for Objective C, Python, Ruby, Javascript, Scala, PHP, Perl… a veritable list of nearly all popular languages in use today. This shift in thinking has taken over, so much so that most programmers alive today have never written code any other way.

Since a standard definition doesn't exist, for the purpose of our discussion we will provide one.

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).

Perhaps as a consequence of the initial implementation having inheritance and polymorphism, a feature virtually all of the derivatives also adopted, definitions of object oriented programming typically also include those features as requirements.

We will look at how Go does objects, polymorphism and inheritance and allow you to make your own conclusion.

Objects In Go

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.

Let's use an example to illustrate this:

```
type rect struct {
    width int
    height int
}

func (r *rect) area() int {
    return r.width * r.height
}

func main() {
    r := rect{width: 10, height: 5}
    fmt.Println("area: ", r.area())
}
```

There's a lot we could talk about here. It's probably best to walk through the code line by line and explain what is happening.

The first block is defining a new type called a 'rect'. This is a struct type. The struct has two fields, both of which are type int.

The next block is defining a method bound to this struct. This is accomplished by defining a function and attaching (binding) it to a rect. Technically, in our example it is really attached to a pointer to a rect. While the method is bound to the type, Go requires us to have a value of that type to make the call, even if the value is the zero value for that type (in the case of a struct the zero value is nil).

The final block is our main function. The first line creates a value of type rect. There are other syntaxes we could use to do this, but this is the most idiomatic way. The second line prints to the output the result of calling the area function on our rect 'r'.

To me this feels very much like an object. I am able to create a structured data type and then define methods that interact with that specific data.

What haven't we done? In most object oriented languages we would be using the 'class' keyword to define our objects. When using inheritance it is a good practice to define interfaces for those classes. In doing so we would be defining an inheritance hierarchy tree (in the case of single inheritance).

An additional thing worth noting is that in Go any named type can have methods, not only structs. For example I can define a new type 'Counter' which is of type int and define methods on it. See an example at http://play.golang.org/p/LGB-2j707c

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
- 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.

Inheritance Is Best Left Out

The following really drives home this point. It comes from a JavaWorld article titled why extends is evil :

> *The Gang of Four Design Patterns book discusses at length replacing implementation inheritance (extends) with interface inheritance (implements).*
>
> *I once attended a Java user group meeting where James Gosling (Java's inventor) was the featured speaker. During the memorable Q&A session, someone asked him: "If you could do Java over again, what would you change?" "I'd leave out classes," he replied. After the laughter died down, he explained that the real problem wasn't classes per se, but rather implementation inheritance (the extends relationship). Interface inheritance (the implements relationship) is preferable. You should avoid implementation inheritance whenever possible.*

Polymorphism & Composition In Go

Instead of inheritance Go strictly follows the composition over inheritance principle . Go accomplishes this through both subtyping (is-a) and object composition (has-a) relationships between structs and interfaces.

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.

```
type Person struct {
   Name string
   Address Address
}

type Address struct {
   Number string
   Street string
   City   string
   State  string
   Zip    string
}

func (p *Person) Talk() {
    fmt.Println("Hi, my name is", p.Name)
}

func (p *Person) Location() {
    fmt.Println("I’m at", p.Address.Number, p.Address.Street, p.Address.City, p.Address.State,
p.Address.Zip)
}

func main() {
    p := Person{
        Name: "Steve",
        Address: Address{
            Number: "13",
            Street: "Main",
            City:   "Gotham",
            State:  "NY",
            Zip:    "01313",
        },
    }

    p.Talk()
    p.Location()
}
```

Output

```
> Hi, my name is Steve
> I'm at 13 Main Gotham NY 01313
```

http://play.golang.org/p/LigPIVT2mf

Important things to realize from this example is that Address remains a distinct entity, while existing within the Person. In the main function we demonstrate that you can set the p.Address field to an address, or simply set the fields by accessing them via dot notation.

Pseudo Subtyping In Go

AUTHORS NOTE:

In the first version of this post it made the incorrect claim that Go supports the is-a relationship via Anonymous fields. In reality Anonymous fields appear to be an is-a relationship by exposing embedded methods and properties as if they existed on the outer struct. This falls short of being an is-a relationship for reasons now provided below. Go *does have support for is-a relationships via interfaces, covered below. The current version of this post refers to Anonymous fields as a pseudo is-a relationship because it looks and behaves in some ways like subtyping, but isn't.*

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.

This code depends on and adds to the code in the example above.

```
type Citizen struct {
   Country string
   Person
}

func (c *Citizen) Nationality() {
    fmt.Println(c.Name, "is a citizen of", c.Country)
}

func main() {
    c := Citizen{}
    c.Name = "Steve"
    c.Country = "America"
    c.Talk()
    c.Nationality()
}
```

Output

```
>  Hi, my name is Steve
>  Steve is a citizen of America
```

http://play.golang.org/p/eCEpLkQPR3

We accomplish this pseudo is-a relationship in go using what is called an Anonymous field. In our example Person is an anonymous field of Citizen. Only the type is given, not the field name. It assumes all of the properties and methods of a Person and is free to use them or promote it's own.

Promoting Methods Of Anonymous Fields

An example of this would be that citizens Talk just as People do, but in a different way.

For this we simply define Talk for *Citizen, and run the same main function as defined above. Now instead of calling *Person.Talk(), *Citizen.Talk() will be called instead.

```
func (c *Citizen) Talk() {
    fmt.Println("Hello, my name is", c.Name, "and I'm from", c.Country)
}
```

Output

```
>  Hello, my name is Steve and I'm from America
>  Steve is a citizen of America
```

http://play.golang.org/p/jafbVPv5H9

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.

In our example from above the following code would work:

```
func main() {
    c := Citizen{}
    c.Name = "Steve"
    c.Country = "America"
    c.Talk()         // <- Notice both are accessible
    c.Person.Talk()  // <- Notice both are accessible
    c.Nationality()
}
```

Output

```
>  Hello, my name is Steve and I'm from America
>  Hi, my name is Steve
>  Steve is a citizen of America
```

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:

```
package main
```

```
type A struct{
}

type B struct {
    A  //B is-a A
}

func save(A) {
    //do something
}

func main() {
    b := B
    save(&b);  //OOOPS! b IS NOT A
}
```

Output:

```
> prog.go:17: cannot use b (type *B) as type A in function argument
>  [process exited with non-zero status]
```

http://play.golang.org/p/dt1mTXW-BH

The previous example came directly from a response to this post on Hacker News. Thanks Optymizer!

True Subtyping In Go

Go interfaces are pretty unique in how they work. This section focuses only on how they pertain to subtyping which will not do them proper justice. See the further reading section at the end of the post to learn more.

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.

```
func SpeakTo(p *Person) {
    p.Talk()
}

func main() {
    p := Person{Name: "Dave"}
    c := Citizen{Person: Person{Name: "Steve"}, Country: "America"}

    SpeakTo(&p)
    SpeakTo(&c)
}
```

Output

```
>  Running it will result in
>  prog.go:48: cannot use c (type *Citizen) as type *Person in function argument
>  [process exited with non-zero status]
```

http://play.golang.org/p/lvEjaMQ25D

As expected, this fails. In our code a Citizen isn't a Person, even though they share many of the same properties, they are viewed as distinct types.

However if we add an interface called Human and use that as the input for our SpeakTo function it will all work as intended.

```
type Human interface {
    Talk()
}

func SpeakTo(h Human) {
    h.Talk()
}

func main() {
    p := Person{Name: "Dave"}
    c := Citizen{Person: Person{Name: "Steve"}, Country: "America"}

    SpeakTo(&p)
    SpeakTo(&c)
}
```

Output

```
>   Hi, my name is Dave
>   Hi, my name is Steve
```

http://play.golang.org/p/ifcP2mAOnf

There are two critical points to make about subtyping in Go:

1. 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.

2. 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.

Go, Object-Oriented Programming Without Objects Or Inheritance

As we have demonstrated here, 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.

Welcome to the new 'object'-less OO programming model.

Discussion

Join the discussion on hacker news and Reddit - Golang

Further Reading

- http://nathany.com/good/
- http://www.artima.com/lejava/articles/designprinciples.html
- http://www.goinggo.net/2014/05/methods-interfaces-and-embedded-types.html

Mon Jun 9, 2014
2600 Words

Read In About 12 Min

Development
golang

#go #golang #development #object-oriented

Pointers vs References

9 MongoDB 2.6 Drivers Released

About The Author:

Steve Francia has been responsible for two of the largest commercial open source projects as the current Chief Operator of the Docker Project and the former Chief Developer Advocate of MongoDB . Steve has also run some of the top community-based open source projects: spf13-vim , Hugo , Cobra & Viper

He loves open source and is thrilled to be able to work on it full-time and then some. In writing, Steve tweets as @spf13 , codes on GitHub , blogs at spf13.com , connects on LinkedIn , ocassionally posts on Google+ and has written a few books for O'Reilly.

In person, he enjoys giving talks and workshops and spending time with the developer community around the world. When not coding, he is usually having fun outdoors with his wife and four children.

Comments

10 Comments Hacking Management with spf13 1 Login

Recommend 4 Share Sort by Best

Join the discussion…

sendyHalim • 2 years ago

Great explanation, I'm new to Go and just encountered issues about OOP in Go. Your article really helps :)

3 • Reply • Share ›

rohitio • 9 months ago

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

1 • Reply • Share ›

Roger Welin • a year ago

Great write-up!

1 • Reply • Share ›

kc merrill • a year ago

A really helpful article Steve. Thanks for sharing!

1 • Reply • Share ›

Ryan Cheung • a year ago

I'm using Ruby for the moment and learning Go. This make it clean that Go's OO is different than the typical one. I kinda want to shift to Go asap. Like the less-object OO.

1 • Reply • Share ›

enricoc • 2 years ago

This is probably the best article about OO in Go that I have read so far ;) Great job Steve!

1 • Reply • Share ›

Sahin Habesoglu • 7 months ago

Very nice post, thanks.

Reply • Share ›

essay services reviews • 8 months ago

Putting up those information that explains this GO could help many people in understanding that they should really do something like this. They can then gain some ideas and techniques on how are they going to make use of this and make it as their main tool for some of their projects.

Reply • Share ›

akurkin • a year ago

this is really really good, thank you!

Reply • Share ›

Tredecim • 2 years ago

Excellent explanation of OO in Go, loved it!

Reply • Share ›

ALSO ON **HACKING MANAGEMENT WITH SPF13**

Go is for lovers

1 comment • 10 months ago•

online essay writer — It's actually one of the most useful language in programming that was being used by a lot of people who wanted to aim for some …

I'm joining the Go team at Google

34 comments • 3 months ago•

Anton Holmquist — A good debugger is missing! Also, please never make any breaking changes to the language. It's been such a pleasure to work …

post two

1 comment • 16 days ago•

Tyler — moar comments

Pipelines and oil spills: what data show

2 comments • a day ago•

Saeed Mirshekari — Very nice! The number of available features in the original dataset is tempting to do even more interesting stuff. …

Subscribe **Add Disqus to your site Add Disqus Add** **Privacy**