Let's Go: Object-Oriented Programming in Golang

by Gigi Sayfan 19 Aug 2016

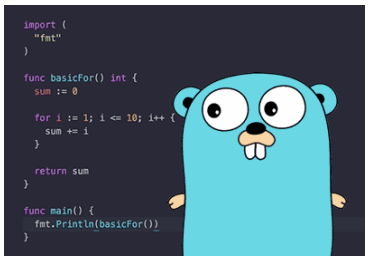
Difficulty: Intermediate Length: Medium Languages: English

Go

Go is a strange mix of old and new ideas. It has a very refreshing approach where it isn't afraid to throw away established notions of "how to do things". Many people are not even sure if Go is an object-oriented language. Let me put that to rest right now. It is!

In this tutorial you'll learn about all the intricacies of object-oriented design in Go, how the pillars of object-oriented programming like encapsulation, inheritance, and polymorphism are expressed in Go, and how Go compares to other languages.

Go is an incredibly powerful programming language, learn everything from writing simple utilities to building scalable, flexible web servers in our full course.



GO LANG

Go Fundamentals for Building Web Servers

Derek Jensen

The Go Design Philosophy

Go's roots are based on C and more broadly on the Algol family. Ken Thompson half-jokingly said that Rob Pike, Robert Granger and himself got together and decided they

hate C++. Whether it's a joke or not, Go is very different from C++. More on that later. Go is about ultimate simplicity. This is explained in detail by Rob Pike in Less is exponentially more.

Go vs. Other Languages

Go has no classes, no objects, no exceptions, and no templates. It has garbage collection and built-in concurrency. The most striking omission as far as object-oriented is concerned is that there is no type hierarchy in Go. This is in contrast to most object-oriented languages like C++, Java, C#, Scala, and even dynamic languages like Python and Ruby.

Go Object-Oriented Language Features

Go has no classes, but it has types. In particular, it has structs. Structs are user-defined types. Struct types (with methods) serve similar purposes to classes in other languages.

Structs

A struct defines state. Here is a Creature struct. It has a Name field and a boolean flag called Real, which tells us if it's a real creature or an imaginary creature. Structs hold only state and no behavior.

```
type Creature struct {

 Name string

 Real bool

}
```

Methods

Methods are functions that operate on particular types. They have a receiver clause that mandates what type they operate on. Here is a `Dump()` method that operates on Creature structs and prints their state:

```
func (c Creature) Dump() {
 fmt.Printf("Name: '%s', Real: %t\n", c.Name, c.Real)
}
```

This is an unusual syntax, but it is very explicit and clear (unlike the implicit "this" or Python's confusing "self").

Embedding

You can embed anonymous types inside each other. If you embed a nameless struct then the embedded struct provides its state (and methods) to the embedding struct directly. For example, the `FlyingCreature` has a nameless `Creature` struct embedded in it, which means a `FlyingCreature` is a `Creature`.

```
type FlyingCreature struct {
 Creature
```

```
  WingSpan int
}
```

Now, if you have an instance of a FlyingCreature, you can access its Name and Real attributes directly.

```
dragon := &FlyingCreature{
   Creature{"Dragon", false, },
   15,
}

fmt.Println(dragon.Name)
fmt.Println(dragon.Real)
fmt.Println(dragon.WingSpan)
```

Interfaces

Interfaces are the hallmark of Go's object-oriented support. Interfaces are types that declare sets of methods. Similarly to interfaces in other languages, they have no implementation.

Objects that implement all the interface methods automatically implement the interface. There is no inheritance or subclassing or "implements" keyword. In the following code snippet, type Foo implements the Fooer interface (by convention, Go interface names end with "er").

```
type Fooer interface {
  Foo1()
  Foo2()
  Foo3()
}

type Foo struct {
}

func (f Foo) Foo1() {
   fmt.Println("Foo1() here")
}

func (f Foo) Foo2() {
   fmt.Println("Foo2() here")
}

func (f Foo) Foo3() {
   fmt.Println("Foo3() here")
}
```

Object-Oriented Design: The Go Way

Let's see how Go measures up against the pillars of object-oriented programming: encapsulation, inheritance, and polymorphism. Those are features of class-based programming languages, which are the most popular object-oriented programming languages.

At the core, objects are language constructs that have state and behavior that operates on the state and selectively exposes it to other parts of the program.

Encapsulation

Go encapsulates things at the package level. Names that start with a lowercase letter are only visible within that package. You can hide anything in a private package and just expose specific types, interfaces, and factory functions.

For example, here to hide the `Foo` type above and expose just the interface you could rename it to lower case `foo` and provide a `NewFoo()` function that returns the public Fooer interface:

```
01  type foo struct {
02  }
03
04  func (f foo) Foo1() {
05     fmt.Println("Foo1() here")
06  }
07
08  func (f foo) Foo2() {
09     fmt.Println("Foo2() here")
10  }
11
12  func (f foo) Foo3() {
13     fmt.Println("Foo3() here")
14  }
15
16  func NewFoo() Fooer {
17     return &Foo{}
18  }
```

Then code from another package can use `NewFoo()` and get access to a `Fooer` interface implemented by the internal `foo` type:

```
1  f := NewFoo()
2
3  f.Foo1()
4
5  f.Foo2()
6
7  f.Foo3()<br>
```

Inheritance

Inheritance or subclassing was always a controversial issue. There are many problems with implementation inheritance (as opposed to interface inheritance). Multiple inheritance as implemented by C++ and Python and other languages suffers from the deadly diamond of death problem, but even single inheritance is no picnic with the fragile base-class problem.

Modern languages and object-oriented thinking now favor composition over inheritance. Go takes it to heart and doesn't have any type hierarchy whatsoever. It allows you to share implementation details via composition. But Go, in a very strange twist (that probably originated from pragmatic concerns), allows anonymous composition via embedding.

For all intents and purposes, composition by embedding an anonymous type is equivalent to implementation inheritance. An embedded struct is just as fragile as a base class. You can also embed an interface, which is equivalent to inheriting from an interface in languages like Java or C++. It can even lead to a runtime error that is not discovered at compile time if the embedding type doesn't implement all the interface methods.

Here SuperFoo embeds the Fooer interface, but doesn't implement its methods. The Go compiler will happily let you create a new SuperFoo and call the Fooer methods, but will obviously fail at runtime. This compiles:

```
type SuperFooer struct {
  Fooer
}

func main() {
  s := SuperFooer{}
  s.Foo2()
```

Running this program results in a panic:

```
panic: runtime error: invalid memory address or nil pointer dereference
[signal 0xb code=0x1 addr=0x28 pc=0x2a78]

goroutine 1 [running]:
panic(0xde180, 0xc82000a0d0)
  /usr/local/Cellar/go/1.6/libexec/src/runtime/panic.go:464 +0x3e6
main.main()
  /Users/gigi/Documents/dev/go/src/github.com/oop_test/main.go:104 +0x48
```

```
exit status 2

Process finished with exit code 1
```

Polymorphism

Polymorphism is the essence of object-oriented programming: the ability to treat objects of different types uniformly as long as they adhere to the same interface. Go interfaces provide this capability in a very direct and intuitive way.

Here is an elaborate example where multiple creatures (and a door!) that implement the Dumper interface are created and stored in a slice and then the `Dump()` method is called for each one. You'll notice different styles of instantiating the objects too.

```
package main

import "fmt"

type Creature struct {
  Name string
  Real bool
}

func Dump(c*Creature) {
  fmt.Printf("Name: '%s', Real: %t\n", c.Name, c.Real)
}

func (c Creature) Dump() {
  fmt.Printf("Name: '%s', Real: %t\n", c.Name, c.Real)
}

type FlyingCreature struct {
  Creature
  WingSpan int
}

func (fc FlyingCreature) Dump() {
  fmt.Printf("Name: '%s', Real: %t, WingSpan: %d\n",
    fc.Name,
    fc.Real,
    fc.WingSpan)
}

type Unicorn struct {
  Creature
}

type Dragon struct {
  FlyingCreature
}

type Pterodactyl struct {
  FlyingCreature
}

func NewPterodactyl(wingSpan int) *Pterodactyl {
  pet := &Pterodactyl{
    FlyingCreature{
```

```
    FlyingCreature{
      Creature{
        "Pterodactyl",
        true,
      },
      wingSpan,
    },
  }
  return pet
}

type Dumper interface {
  Dump()
}

type Door struct {
  Thickness int
  Color     string
}

func (d Door) Dump() {
  fmt.Printf("Door => Thickness: %d, Color: %s", d.Thickness, d.Color)
}

func main() {
  creature := &Creature{
    "some creature",
    false,
  }

  uni := Unicorn{
    Creature{
      "Unicorn",
      false,
    },
  }

  pet1 := &Pterodactyl{
    FlyingCreature{
      Creature{
        "Pterodactyl",
        true,
      },
      5,
    },
  }

  pet2 := NewPterodactyl(8)

  door := &Door{3, "red"}

  Dump(creature)
  creature.Dump()
  uni.Dump()
  pet1.Dump()
  pet2.Dump()

  creatures := []Creature{
    *creature,
    uni.Creature,
    pet1.Creature,
    pet2.Creature}
  fmt.Println("Dump() through Creature embedded type")
  for _, creature := range creatures {
    creature.Dump()
  }
```

```
109   }
110
111   dumpers := []Dumper{creature, uni, pet1, pet2, door}
112   fmt.Println("Dump() through Dumper interface")
113   for _, dumper := range dumpers {
114     dumper.Dump()
115   }
116 }
```

Conclusion

Go is a bona fide object-oriented programming language. It enables object-based modeling and promotes the best practice of using interfaces instead of concrete type hierarchies. Go made some unusual syntactic choices, but overall working with types, methods, and interfaces feels simple, lightweight, and natural.

Embedding is not very pure, but apparently pragmatism was at work, and embedding was provided instead of only composition by name.

Gigi Sayfan
Principal Software Architect at Helix

Gigi Sayfan is a principal software architect at Helix — a bioinformatics and

genomics start-up. Gigi has been developing software professionally for more than 20 years in domains as diverse as instant messaging, morphing, chip fabrication process control, embedded multimedia applications for game consoles, brain-inspired machine learning, custom browser development, web services for 3D distributed game platforms, IoT sensors and virtual reality. He has written production code in many programming languages such as Go, Python, C, C++, C#, Java, Delphi, JavaScript, and even Cobol and PowerBuilder for operating systems such as Windows (3.11 through 7), Linux, Mac OSX, Lynx (embedded), and Sony PlayStation. His technical expertise includes databases, low-level networking, distributed systems, unorthodox user interfaces, and general software development life cycle.

https://ewencp.org/blog/gol... and could not find a general iterator interface for golang.
A language that is object oriented needs to have at least the ability to create a general numerical iterator with a generic sum method. The exact syntax and name isn't important, however it is a serious breach of OO design if it is impossible to design a calculation based on a custom type that produces 100% correct results for all classes of that type, in other words using only interface/class methods to calculate the sum such as "fun(numericIterator N) := sum=0; while N.hasNext() { sum+=N.nextSum()} return sum" (in some hypothetic language).
I was expecting to see functions that have generic interface types as input and not structs, the struct type in golang is not sufficient for writing OO code.
Creating a generic numerical iterator is easy using Javascript, Python, C#, Java, C++. It also possible to create a generic sum iterator in C using callbacks but thats obviously not something considered OO.

• Reply • Share ›

The Gigi ➚ Rolf Stenholm • 2 years ago

@Rolf Stenholm

Hi Rolf, sorry for the late response. I didn't get notified. You're correct that Go currently doesn't have the ability to use generic types in method/function signatures so it's not possible to write a generic sum() function that works for every numeric type. A lot of people think that generics should be added to the language (I'm among them). But, the Go designers so far preferred simplicity as generics add complexity. I hope that in Go 2.0 we will have generics. That said, most definition don't consider generics a requirement for OO.

• Reply • Share ›

Rolf Stenholm ➚ The Gigi • 2 years ago

While there isn't a set definition I have never heard about a language considered OO since the 90s that was considered OO which could not create function/methods of this type.
Java and Python both have this ability using inheritance. The limited multiple inheritance in Java (interfaces) suffices to support these types of functions. Interfaces or object type was used in Java before they introduced generics for this type of fun/method.
Go should have the ability to support this funs/methods despite not supporting generics.

• Reply • Share ›

The Gigi ➚ Rolf Stenholm • 2 years ago

Go supports the empty interface, which is equivalent to Java's Object type (all types in Go implement the empty interface). Anything you can do in Java without generics you can do in Go too. Python is a different story since it's a dynamic language, so generics are not needed as every argument is always of type object, which can be anything and is resolved st runtime. There are multiple alternative approaches to generics in Go and I hope that there will be real generics in Go 2.0. If you're interested check out this article: https://appliedgo.net/gener...

• Reply • Share ›

Rolf Stenholm ➚ The Gigi • 2 years ago

The article is good and I have seen it before but for a language to be OO it should allow OO syntax out of the box where the syntax is easily readable. The article suggest code generation as the best option which implies that GO is not actually an Object Oriented language.
Btw python often uses duck typing or as it is called in the article "reflection".

Marcelo Magallon • 3 years ago

Nice article.

I think it would benefit from a more verbose explanation in the "Inheritance" section, in particular, why the panic.

What you have as an example is _close_ to this:

```
package main

import "fmt"

type AFoo struct{}

func (AFoo) Foo() string {
return "foo"
}

type MyFoo struct {
a *AFoo
}

func main() {
myFoo := MyFoo{}
fmt.Println(myFoo.a.Foo())
}
```

this panics because AFoo's Foo() method is defined with a non-pointer receiver and and you have a pointer member. When calling myFoo.a.Foo(), myFoo.a is dereferenced in order to make the call, and that's where it panics, since "a" is nil.

Modifying the above to match your example:

```
package main

import "fmt"

type Fooer interface {
Foo() string
}

type MyFoo struct {
Fooer
}

func main() {
```

```
myFoo := MyFoo{}
fmt.Println(myFoo.Foo())
}
```

you are embedding an interface, which contains two bits of information: a pointer to the data and a pointer to the type. If you want to call a method thru an interface value, you have to dereference the pointer to the type, which is my this code panics: the Fooer field is not