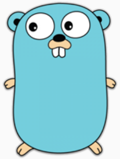
Programming in Go

Matt Holiday
Christmas 2020



Object-Oriented Programming

Object-oriented programming

For many people, the essential elements of object-oriented programming have been:

- abstraction
- encapsulation
- polymorphism
- inheritance

Sometimes those last two items are combined or confused

Go’s approach to OO programming is similar but different

Abstraction

Abstraction: decoupling behavior from the implementation details

The Unix file system API is a great example of effective abstraction

Roughly five basic functions hide all the messy details:

- open
- close
- read
- write
- ioctl

Many different operating system things can be treated like files

Encapsulation

Encapsulation: hiding implementation details from misuse

It’s hard to maintain an abstraction if the details are exposed:

- the internals may be manipulated in ways contrary to the concept behind the abstraction
- users of the abstraction may come to depend on the internal details — but those might change

Encapsulation usually means controlling the visibility of names (“private” variables)

Polymorphism

Polymorphism literally means “many shapes” — multiple types behind a single interface

Three main types are recognized:

- ad-hoc: typically found in function/operator overloading
- parametric: commonly known as “generic programming”
- subtype: subclasses substituting for superclasses

“Protocol-oriented” programming uses explicit interface types, now supported in many popular languages (an ad-hoc method)

In this case, *behavior is completely separate from implementation*, which is good for abstraction

Inheritance

Inheritance has conflicting meanings:

- substitution (subtype) polymorphism
- structural sharing of implementation details

In theory, inheritance should always imply subtyping:
the subclass should be a “kind of” the superclass

See the Liskov substitution principle

Theories about substitution can be pretty messy

Why would inheritance be bad?

It injects a dependence on the superclass into the subclass:

- what if the superclass changes behavior?
- what if the abstract concept is leaky?

Not having inheritance means better encapsulation & isolation

"Interfaces will force you to think in term of communication between objects"
— Nicolò Pignatelli in Inheritance is evil

See also Composition over inheritance and Inheritance tax (Pragmatic)

One more view

“Object-oriented programming to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.” — Alan Kay

He wrote this to

- de-emphasize inheritance hierarchies as a key part of OOP
- emphasize the idea of self-contained objects sending messages to each other
- emphasize polymorphism in behavior

OO in Go

Go offers four main supports for OO programming:

- encapsulation using the package for visibility control
- abstraction & polymorphism using interface types
- enhanced *composition* to provide structure sharing

Go does not offer inheritance or substitutability based on types

Substitutability is based only on **interfaces**: purely a function of abstract **behavior**

See also Go for Gophers

Classes in Go

Classes in Go

Not having classes can be liberating!

Go allows defining methods on any user-defined type, rather than only a “class”

Go allows any object to implemement the method(s) of an interface,
not just a “subclass”

Let’s get away from defining OOP in terms of a particular language’s features!