Object-oriented programming without inheritance

yourbasic.org/golang

Go doesn't have inheritance – instead composition, embedding and interfaces support code reuse and polymorphism.

» Object-oriented programming with inheritance
» Code reuse by composition
» Code reuse by embedding
» Polymorphism and dynamic dispatch with interfaces

Object-oriented programming with inheritance

Inheritance in traditional object-oriented languages offers three features in one. When a Dog inherits from an `Animal`

1. the Dog class reuses code from the `Animal` class,
2. a variable x of type `Animal` can refer to either a Dog or an `Animal`,
3. `x.Eat()` will choose an `Eat` method based on what type of object x refers to.

In object-oriented lingo, these features are known as **code reuse**, **polymorphism** and **dynamic dispatch**.

All of these are available in Go, using separate constructs:

- **composition** and **embedding** provide code reuse,
- **interfaces** take care of polymorphism and dynamic dispatch.

Code reuse by composition

> Don't worry about type hierarchies when starting a new Go project – it's easy to introduce polymorphism and dynamic dispatch later on.

Related

Constructors deconstructed [best practice]
Best practice in Go is to use suitable zero values and factories instead of constructors.
yourbasic.org

Methods explained
Go doesn't have classes, but you can define methods on types. A method is a function with an extra receiver argument.
yourbasic.org

Type, value and equality of interfaces
An interface type consists of a set of method signatures. A variable of interface type can hold any value that implements these methods.
yourbasic.org

Optional parameters, default parameter values and method overloading
Go does not have optional parameters or default parameter values, nor does it support method overloading. Method overloading is emulated by writing several methods with different names.
yourbasic.org

Public vs. private
All identifiers are visible throughout the package where they are defined; an identifier that begins with a capital letter is globally visible.
yourbasic.org

Embedding
Effective Go

Follow on Twitter

Algorithms to Go

Most Read

Do you make these Go coding mistakes?
Why Go? – Key advantages you may have overlooked
Go string handling overview [cheat sheet]
Type, value and equality of interfaces
Concurrent programming

See all 178 Go articles

If a Dog needs some or all of the functionality of an `Animal`, simply use **composition**.

```
type Animal struct {
    // …
}

type Dog struct {
    beast Animal
    // …
}
```

This gives you full freedom to use the `Animal` part of your Dog as needed. Yes, it's that simple.

Code reuse by embedding

If the Dog class inherits **the exact behavior** of an `Animal`, this approach can result in some tedious coding.

```
type Animal struct {
    // …
}

func (a *Animal) Eat()   { … }
func (a *Animal) Sleep() { … }
func (a *Animal) Breed() { … }

type Dog struct {
    beast Animal
    // …
}

func (a *Dog) Eat()   { a.beast.Eat() }
func (a *Dog) Sleep() { a.beast.Sleep() }
func (a *Dog) Breed() { a.beast.Breed() }
```

This code pattern is known as **delegation**.

Go uses **embedding** for situations like this. The declaration of the Dog struct and it's three methods can be reduced to:

```
type Dog struct {
    Animal
    // …
}
```

Polymorphism and dynamic dispatch with interfaces

> Keep your interfaces short, and introduce them only when needed.

Further down the road your project might have grown to include more animals. At this point you can introduce polymorphism and dynamic dispatch using **interfaces**.

If you need to put all your pets to sleep, you can define a `Sleeper` interface.

```
type Sleeper interface {
    Sleep()
}

func main() {
    pets := []Sleeper{new(Cat), new(Dog)}
    for _, x := range pets {
        x.Sleep()
    }
}
```

No explicit declaration is required by the `Cat` and `Dog` types. Any type that provides the methods named in an interface may be treated as an implementation.

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.
–James Whitcomb Riley

What about constructors?

See Constructors deconstructed for best practices on how to set up new data structures in Go.

Share this page: