Interfaces in Go

Main source of motivation –

<https://medium.com/codex/go-interfaces-for-new-gophers-3d608db99d46>

What is an interface?

Well after lot of reading online, to explain this to myself, I can say that an interface contains a set of methods (declarations of methods). A struct (or class in case of OOPs) is said to implement an interface, if it contains (implements) all the methods ( of exactly same signature) as declared in the interface definition.

Explaining this with an example –

Interface Duck() {

Func quak() string,

Func wak() int,

}

Above is interface declaration/definition.

Now lets see a struct below with its function

Type Donald struct {

Name string

}

Func (D Donald) quak() {

Return “Quak Quak Quak”

}

Func (D Donald) wak() {

Return 1

}

In the above code, I declared a struct named Donald and I defined two methods quak and wak associated with the struct and these two methods are having exactly same definition as that of the methods declared in the interface Duck, we can say that the struct Donald implements interface Duck aka Donald “IS A” duck since it implements wak and quak!!

And in main function, things would be written as below –

Func main() {

D := new(Donald)

D .Quack()

D.Wak()

}

In the previous section, I have explained what is a Go interface and the necessary syntax to use it. To recap, I created a struct Donald that implements the Duck interface with two methods, Quack and Walk.

To effectively use Go interfaces, we need to understand why and how an interface can be useful in our program. In essence, Go interfaces allow us to achieve what’s known as **duck typing**.

*💡 Duck typing defines an object or type by****WHAT IT CAN DO****instead of what it actually is.*

Now the question is –

What is the use of such interfaces?!!!

Let us imagine you are working on an application that makes objects quack and walk like a duck.

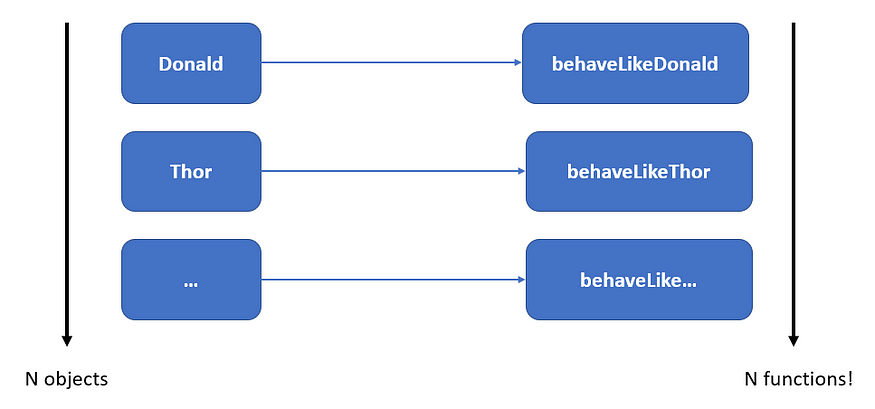
To start simple, you create a struct called Donald like in the previous section with the methods Quack and Walk . Then, you create a normal function that takes in an argument of type Donald and call both the methods.

func behaveLikeDonald(d Donald) {  
 d.Quack()  
 d.Walk()  
}

People love Donald and your application grows in popularity! One day, your users want Thor to behave like a duck. Simple enough, you repeat what you did with Donald .

func behaveLikeThor(t Thor) {  
 t.Quack()  
 t.Walk()  
}

Now, your application gets better! More people start requesting different objects to behave like ducks. Suddenly, you realize your code is not scalable because you need to**create a new function for each object**!



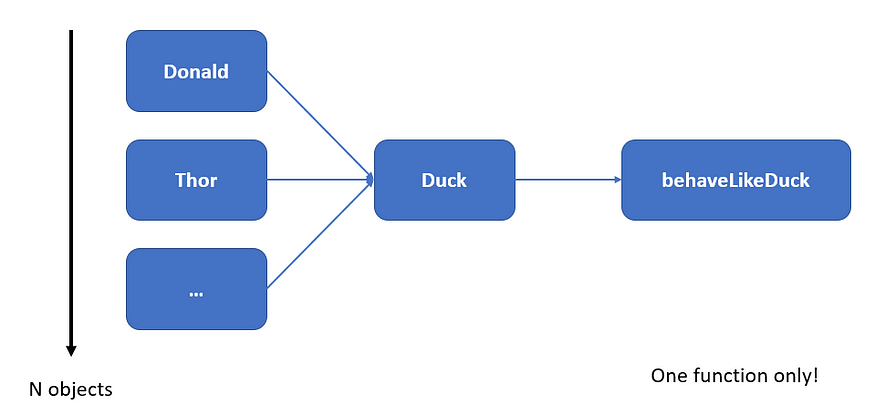
Wouldn’t it be great if you can create **only one function** to handle any objects that have both the Quack and Walk method? You guessed it, *Go interfaces to the rescue*!

In Go, interfaces can be **treated as a type**in function parameters and return values. In the example above, both Donald and Thor are of interface type Duck because **they implement the interface**(i.e. has methods of Duck).

With that, you only need one function that accepts an argument of interface type Duck . This makes your application scales easily with the number of Duck-related objects!

// Define function argument to be of type Duck  
func behaveLikeDuck (d Duck) {  
 d.Quack()  
 d.Walk()  
}

If you are familiar with Object-Oriented Programming (OOP), you can say that the function behaveLikeDuck has achieved [**polymorphism**](https://en.wikipedia.org/wiki/Polymorphism_(computer_science))**.**



N objects with only one function.

In conclusion, Go interfaces allow us to create abstractions based on **the methods that an input type has** instead of its actual concrete type. This allows us to design abstractions that are *flexible, extensible, and scalable*!

If you still cannot see the benefits of Go interfaces, *don’t worry too much about it yet*. When you are starting in Go, there is rarely a need for you to design your own interfaces.

The main takeaway is the idea of **using interfaces as data types**because this is a common design pattern of Go abstractions. Always remember the following statement:

*An interface type is concerned with what an object can do instead of what is actually is. In other words, duck-typing*

**Under the Hood with Interface Values**

In the section above, I have given a simple example of how Go interfaces can be useful in designing powerful and reusable abstractions.

To recap, I created a custom struct Donald that implements a Duck interface with two methods Quack and Walk . There is also a function behaveLikeDuck that takes in an argument of interface type Duck .

func behaveLikeDuck (d Duck) {  
 d.Quack()  
 d.Walk()  
}

Let us think about how Go interfaces work underneath the hood. To be specific, what happens when we call the function behaveLikeDuck ?

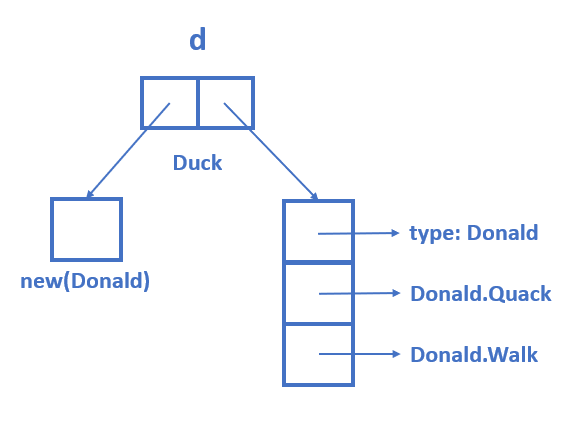
When we pass a variable to behaveLikeDuck , Go performs **static type checking** during compilation to check if the variable satisfies the Duck interface. If it doesn’t, the compilation fails.

During runtime, Go performs **type conversion** and creates a local variable d of type Duck inside behaveLikeDuck . This raises a question: when the methods Quack and Walk are called, *how does Go know which concrete implementation of the methods to execute*?

*❗ There could be many types that implement Duck , each with their own version of Quack and Walk .*

The local variable d is an **interface value**. In the simplest sense, you can think of an interface value as a list with two pieces of information: the **underlying data** and the **concrete** **type of the interface**.

For example, if we provide a variable of type Donald , the underlying data of d points to the instance of Donald while the concrete type is the struct Donald with its method implementations.



Simplified representation of an interface value

An interface value also exposes a method for us to perform [**type assertions**](https://tour.golang.org/methods/15) with its underlying concrete type. We won’t be going into that for now as this deserves a separate article.

When we call a method on the interface value d , Go **looks for the method of the same name** under the concrete type and executes it. In that sense, Go is dynamic enough to work with different but related objects at runtime.

One caveat that you must remember is that the methods you can access on an interface value are **only those defined in the interface**.

Back to the example above, if Donald has another method called Eat , you cannot call it with the interface value d because it is not defined in the Duck interface. Again, if you are familiar with OOP, this is consistent with how **upcasting**works in languages like Java.