### **Interfaces and Dynamic Typing**

This lesson shows how well Go handles dynamic typing as compared to other languages and reveals how Go saves us from a hustle with the implementation of interfaces.

#### WE'LL COVER THE FOLLOWING

- Dynamic typing in Go
- Dynamic method invocation
- Extraction of an interface
- Type implements an interface
- Inheritance of interfaces

# Dynamic typing in Go#

In classical OO languages (like C++, Java, and C#) data and the methods which act upon that data are united in the *class-concept*: a class contains them both, they cannot be separated. In Go, there are no classes: data (structures, or more general types) and methods are treated orthogonally. They are much more loosely coupled.

Interfaces in Go are similar to their Java/C# counterparts; both specify a minimum set of methods that an implementer of an interface must provide. However, they are also more fluid and generic: any type that provides code for the methods of an interface implicitly implements that interface, without having to say explicitly that it does.

Compared to other languages, Go is the only one that combines interface values, static type checking (does a type implement the interface?), dynamic runtime conversion, and no requirement for explicitly declaring that a type satisfies an interface. This property also allows interfaces to be defined and used without having to modify existing code.

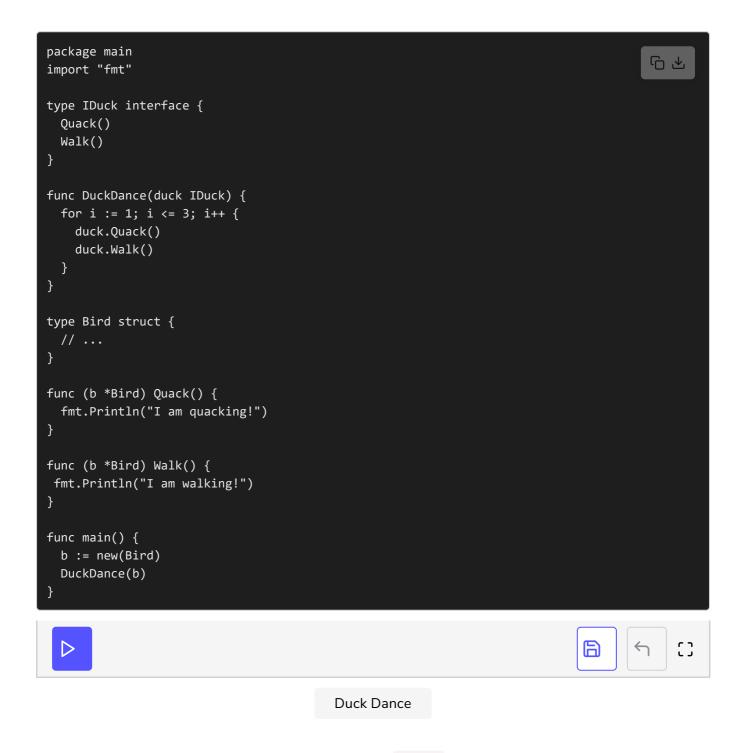
A function that has one or more parameters of an interface type can be called

with a variable whose type implements that interface. Types *implementing an* 

interface can be passed to any function that takes that interface as an argument.

This resembles the *duck typing* in dynamic languages like Python and Ruby; this can be defined to mean that objects can be handled (e.g., passed to functions) based on the methods they provide, regardless of their actual types: what they are is less important than what they can do.

This is illustrated in the following program:



In the code above, at **line 4**, we define an <code>IDuck</code> interface with *two* methods:

<code>Quack()</code> and <code>Walk()</code>. Look at the header of function <code>DuckDance</code>: <code>func</code>

<code>DuckDance(duck IDuck)</code> at **line 9**. It takes the parameter of the <code>IDuck</code> interface

type. It calls the functions Quack() and Walk() 3 times in a row using a for loop.

Type Bird is defined at line 16. We leave out properties here because they are not needed for the example. However, Bird implements the two methods of the interface IDuck, Quack() (from line 20 to line 22) and Walk() (from line 24 to line 26).

We then create a variable **b** of type **Bird** in **main()** at **line 29**. Now, we can call **DuckDance** on **b** (see **line 30**).

## Dynamic method invocation #

However, in Python, Ruby, and the like, duck-typing is performed as late binding (during runtime). The methods are called on those arguments and variables and resolved at runtime (they mostly have methods like responds-to to check whether the object knows this method, but this amounts to more coding and testing).

On the contrary, in Go, the implementation requirements (most often) get statically checked by the compiler. It checks if a type implements all the functions of an interface when there is an assignment of a variable to a variable of that interface. If the method call is on a more general interface type, you can check whether the variable implements the interface by doing a type assertion. As an example, suppose you have different entities represented as types that have to be written out as XML streams. Then, we define an interface for XML, writing with the one method you are looking for (it can even be defined as a private interface):

```
type xmlWriter interface {
   WriteXML(w io.Writer) error
}
```

Now, we can make a function StreamXML for the streaming of any variable, testing with a type assertion if the variable passed implements the interface; if not we call its own function <a href="mailto:encodeToXML">encodeToXML</a> to do the job:

```
// Exported XML streaming function.
func StreamXML(v interface{}, w io.Writer) error {
  if xw, ok := v.(xmlWriter); ok {
    // It's an xmlWriter, use method of asserted type.
```

```
return xw.WriteXML(w)
}

// No implementation, so we have to use our own function (with perhaps r
eflection):
   return encodeToXML(v, w)
}

// Internal XML encoding function.
func encodeToXML(v interface{}, w io.Writer) error {
   // ...
}
```

Go uses the same mechanism in their <code>encoding/gob</code> package. Here, they have defined the two interfaces <code>GobEncoder</code> and <code>GobDecoder</code>. These allow types to define their way to encode and decode their data to and from byte streams; otherwise, the standard way with reflection is used. So, Go provides the advantages of a dynamic language, but it does not have its disadvantages of run-time errors! This alleviates for a part they need for unit-testing, which is very important in dynamic languages but also presents a considerable amount of effort. Go interfaces promote separation of concerns, improve code reuse, and make it easier to build on patterns that emerge as the code develops. With Go-interfaces, the <code>Injection Dependency</code> pattern can also be implemented.

## Extraction of an interface #

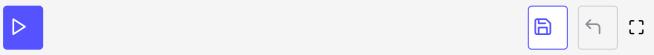
A refactoring pattern that is very useful is extracting interfaces, reducing thereby the number of types and methods needed, without having the need to manage a whole class-hierarchy as in more traditional class-based OO-languages. The way interfaces behave in Go allows developers to discover their programs' types as they write them. If there are several objects that all have the same behavior, and a developer wishes to abstract that behavior, they can create an interface and then use that.

```
package main
import "fmt"

type Shaper interface {
        Area() float32
}

type TopologicalGenus interface {
```

```
Rank() int
}
type Square struct {
        side float32
}
func (sq *Square) Area() float32 {
        return sq.side * sq.side
}
func (sq *Square) Rank() int {
        return 1
}
type Rectangle struct {
        length, width float32
func (r Rectangle) Area() float32 {
        return r.length * r.width
func (r Rectangle) Rank() int {
        return 2
}
func main() {
        r := Rectangle{5, 3} // Area() of Rectangle needs a value
                            // Area() of Square needs a pointer
        q := &Square{5}
        shapes := []Shaper{r, q}
        fmt.Println("Looping through shapes for area ...")
        for n, _ := range shapes {
                fmt.Println("Shape details: ", shapes[n])
                fmt.Println("Area of this shape is: ", shapes[n].Area())
        topgen := []TopologicalGenus{r, q}
        fmt.Println("Looping through topgen for rank ...")
        for n, _ := range topgen {
                fmt.Println("Shape details: ", topgen[n])
                fmt.Println("Topological Genus of this shape is: ", topgen[n].Rank())
        }
```



**Extracting Interface** 

For the program above we found out that, we need a new interface TopologicalGenus, defined at **line 9**, which gives the *rank* of a shape (here, simply implemented as returning an int).

At **line 40**, we create an array **shaper** of type **Shaper** and populate it with **r** and **q**. Then, from **line 42** to **line 44**, we loop through this array and call the **Area()** method on both.

In the exactly same way, we can now use our new TopologicalGenus type: at line 46, we create an array shaper of type TopologicalGenus, and populate it with  $\mathbf{r}$  and  $\mathbf{q}$ . Then, we loop through this array and call the Rank() method on both.

Notice that in order to introduce this new interface, we didn't need to change the Square and Rectangle types themselves. We only needed to make them implement the method Rank() of TopologicalGenus. You may have noticed that for Square the method Rank() is implemented which returns 1 (from line 21 to line23). Similarly for Rectangle, the method Rank() is implemented which returns 2 (from line 33 to line35).

So you don't have to work all your interfaces out ahead of time; the *whole design can evolve without invalidating early decisions*. If a type must implement a new interface, the type itself doesn't have to be changed; you must only make the new method(s) on the type.

## Type implements an interface #

If you wish that the types of an interface explicitly declare that they implement it, you can add a method with a descriptive name to the interface's method set. For example:

```
type Fooer interface {
Foo()
ImplementsFooer()
}
```

A type Bar must then implement the ImplementsFooer method to be a Fooer, clearly documenting the fact.

```
type Bar struct{}
func (b Bar) ImplementsFooer() {}
func (b Bar) Foo() {}
```

Most code doesn't make use of such constraints since they limit the utility of the interface idea. Sometimes, however, they can be necessary to resolve ambiguities among similar interfaces.

#### Inheritance of interfaces #

When a type includes (embeds) another type (which implements one or more interfaces) as a pointer, then the type can use all of the interface's methods. For example:

```
type Task struct {
  Command string
  *log.Logger
}
```

A factory for this type could be:

```
func NewTask(command string, logger *log.Logger) *Task {
  return &Task{command, logger}
}
```

When log.Logger implements a Log() method, then a value task of Task can call it:

```
task.Log()
```

A type can also inherit from multiple interfaces providing something like multiple inheritance:

```
type ReaderWriter struct {
  io.Reader
  io.Writer
}
```

The principles outlined above are applied throughout all Go-packages, thus maximizing the possibility of using polymorphism and minimizing the amount of code. This is considered an important best practice in Go-programming.

Useful interfaces can be detected when the development is already underway. It is easy to add new interfaces because existing types don't have to change (they only have to implement their methods). Current functions can then be generalized from having a parameter(s) of a constrained type to a parameter of the interface type: often, only the signature of the function needs to be shared. Contract this to close based OO languages, where in such a case, the

design of the whole class-hierarchy has to be adapted.

Interfaces providing ease of modifications make Go a popular language. In the next lesson, we'll see how the structs, interfaces, and high-order functions go along so well.