What is an Interface?

This lesson gives a brief introduction to interfaces and explains how Go, having no concepts of classes and inheritance, implements the OO behavior.

WE'LL COVER THE FOLLOWING

Introduction to interface

Go is not a *classic* OO language. It doesn't recognize the concept of classes and inheritance. However, it does contain the very flexible concept of interfaces, with which a lot of aspects of object-orientation can be made available. Interfaces in Go provide a way to specify the behavior of an object.

An **interface** defines a set of methods (the method set), but these methods do not contain code: they are not implemented (they are abstract). Also, an interface cannot contain variables. An interface is declared in the format:

```
type Namer interface {
    Method1(param_list) return_type
    Method2(param_list) return_type
    ...
}
```

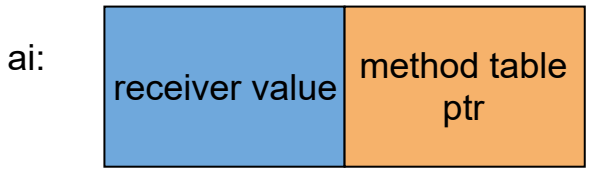
`Namer` is an interface type. The name of an interface is formed by the method name plus the *[er]* suffix, such as Printer, Reader, Writer, Logger, Converter, and so on, thereby giving an active noun as a name. A less-used alternative (when ...er is not so appropriate) is to end it with *able*, like in Recoverable or to start it with an *I* (more like in .NET or Java). Interfaces in Go are short; they usually have one two three methods (except for the empty interface, which

has 0 methods).

Unlike in most OO languages, in Go, interfaces can have values that are a variable of the interface type or an interface value:

```
var ai Namer
```

`ai` is a multiword data structure with an uninitialized value of *nil*. Although not entirely the same thing, `ai` is, in essence, a pointer. So, pointers to interface values are illegal; they would be wholly useless and give rise to errors in code.



Interface Value in Memory

Explanation

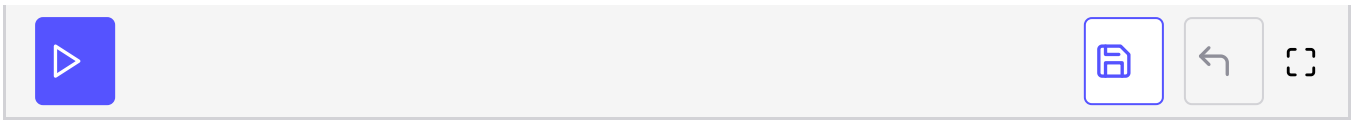
Types (like structs) can have the method `set` of the interface implemented. The implementation contains real code for each method and how to act on a variable of that type: *they implement the interface*. The method `set` forms the interface of that type. A variable of a type that implements the interface can be assigned to `ai` (the receiver value), and the method table then has pointers to the implemented interface methods. Of course, both of these change when a variable of another type (that also implements the interface) is assigned to `ai`. Note the following important points:

- A type doesn't have to state explicitly that it implements an interface; interfaces are satisfied implicitly.
- Multiple types can implement the same interface.
- A type that implements an interface can also have other functions.
- A type can implement many interfaces.
- An interface type can contain a reference to an instance of any of the types that implement the interface.

Even if the interface was defined later than the type, in a different package or compiled separately: if the object implements the methods named in the interface, then it implements the interface. All these properties allow for a lot of flexibility.

As a first example look below:

```
package main
import "fmt"

type Shaper interface {
  Area() float32
}

type Square struct {
  side float32
}

func (sq *Square) Area() float32 {
  return sq.side * sq.side
}

func main() {
  sq1 := new(Square)
  sq1.side = 5
  var areaIntf Shaper
  areaIntf = sq1
  // shorter, without separate declaration:
  // areaIntf := Shaper(sq1)
  fmt.Printf("The square has area: %f\n", areaIntf.Area())
}
```



Interfaces in Go

In the program above, we define an interface `Shaper` at **line 4**, with one method `Area()` returning *float32* value and a struct of type `Square` at **line 8**, with one field `side` of type *float32*. At **line 12**, we define a method `Area()` that can be called by a pointer to the `Square` type object. This method returns the area of a square `sq`. The struct `Square` implements the interface `Shaper`. Now, the interface variable contains a reference to the `Square` variable, and through it, we can call the method `Area()` on `Square`.

Look at the `main`. We make a `Square` type variable `sq1` at **line 17** and give a value of **5** to its field `side` (at **line 18**). As discussed above, you could call the method immediately on the `Square` value `sq1.Area()`, but the novel thing is that we can call it on the interface value (see **line 23**), thereby generalizing the

that we can call it on the interface value (see **line 23**), thereby generalizing the call.

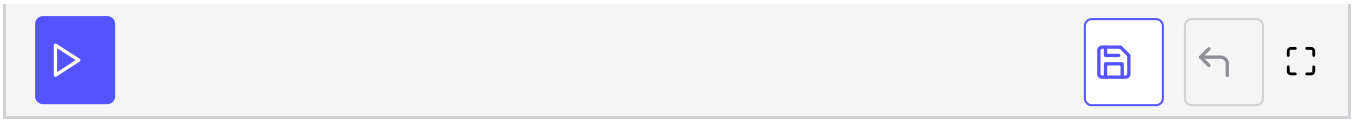
The interface variable both contains the value of the receiver value and a pointer to the appropriate method in a method table.

This is Go's version of *polymorphism*, a well-known concept in OO software. The right method is chosen according to the current type or put otherwise. A type seems to exhibit different behaviors when linked to different values. If `Square` didn't have an implementation of `Area()`, we would receive the clear compiler error: `cannot use sq1 (type *Square) as type Shaper in assignment: *Square does not implement Shaper (missing Area method)`.

The same error would occur if `Shaper` had another method `Perimeter()`, and `Square` would not have an implementation for that. We expand the example with a type `Rectangle` which also implements `Shaper`. Now, we can make an array with elements of type `Shaper` and show polymorphism in action by using a for range on it and calling `Area()` on each item:

```
package main
import "fmt"

type Shaper interface {
  Area() float32
}

type Square struct {
  side float32
}

func (sq *Square) Area() float32 {
  return sq.side * sq.side
}

type Rectangle struct {
  length, width float32
}

func (r Rectangle) Area() float32 {
  return r.length * r.width
}

func main() {
  r := Rectangle{5, 3} // Area() of Rectangle needs a value
  q := &Square{5} // Area() of Square needs a pointer
  // shapes := []Shaper{Shaper(r), Shaper(q)}
  // or shorter:
  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())
```

```
  }
}
```



Interface with 2 Different Structs

In the above program, we define an interface `Shaper` at **line 4** with one method `Area()`, returning a *float32* value and a struct of type `Square` at **line 8** with one field `side` of type *float32*. At **line 12**, we define a method `Area()` that can be called by a pointer to the `Square` type object. Again at **line 16**, we make a struct of type `Rectangle` at **line 16** with one field `side` of type *float32*. At **line 20**, we define a method `Area()` that can be called by the `Rectangle` type object.

Now, look at `main`. We make a value type `Rectangle` object and a pointer type `Square` object. As the interface variable contains the reference to the `Square` and `Rectangle` type object, we create a `Shaper` type array `shapes` and add `r` (a `Rectangle` object) and `q` (a `Square` object) in it.

Our goal is to find the area of all the shapes present in `shapes`. We use a *for* loop at **line 31**. For each iteration, we print the details of a shape in `shapes` (see **line 32**) and print the area (see **line 33**). In this case, our first shape is the rectangle `r`. So, **line 32** will output **{5,3}** on the screen. At **line 33**, `Area()`, having the object of `Rectangle` as a receiver (see **line 20**), will be called, and **15** will be returned from the method and printed on the screen by **line 33**. In the next iteration, our shape is the square `q`. So, **line 32** will output **&{5}** on the screen, as `q` is the pointer variable. At **line 33**, `Area()`, having a pointer to the object of `Square` as a receiver (see **line 12**), will be called, and **25** will be returned from the method and printed on screen by **line 33**.

Perhaps, you can now begin to see how interfaces can produce *cleaner*, *simpler*, and more *scalable* code.

An example from the standard library

The `io` package contains an interface type `Reader`:

```
type `Reader` interface {
  Read(p []byte) (n int, err error)
}
```

If we define a variable r as:

```
var r io.Reader
```

then the following is correct code:

```
r = os.Stdin
r = bufio.NewReader(r)
r = new(bytes.Buffer)
f,_ := os.Open("test.txt")
r = bufio.NewReader(f)
```

because the right-hand side objects each implement a `Read()` method with the exact same signature. The static type of `r` is `io.Reader`.

> **Remark:** Sometimes the word interface is also used in a slightly different way: seen from the standpoint of a particular type, the interface of that type is the set of exported methods defined for that type, without there having to be an explicit interface type defined with these methods. This is better called the API (Application Programming Interface) of the type.

Now that you know the purpose of interfaces in Go, in the next lesson, you have to write a program to solve a problem.