

Interfaces in Go

An interface is a great and only way to achieve Polymorphism in Go



Uday Hiwarale

[Follow](#)

Oct 21, 2018 · 11 min read



(source: pexels.com)

▀▀ What is an interface?

We talked a lot about **object** and **behavior** in the **structs** and **methods** lessons. We saw structs (*and other non-struct types too*) implementing methods. An **interface** is a collection of **method signatures** that an **Object** can implement. Hence interface defines the behavior of the object.

For example, a `Dog` can `walk` and `bark`. If an interface declares method signatures of `walk` and `bark` while `Dog` implements `walk` and `bark` methods, then `Dog` is said to implement that interface.

The primary job of an interface is to provide only method signatures consisting of the method name, input arguments and return types. It is up to a Type (*e.g. struct type*) to declare methods and implement them.

If you are an **OOP** programmer, you might have used `implement` keyword a lot while implementing an interface. But in Go, **you do not explicitly mention if a type implements an interface**. If a type implements a method of signature that is defined in an interface, then that type is said to implement that interface. Like saying **if it walks like a duck, swims like a duck and quacks like a duck, then it's a duck.**

➡ Declaring interface

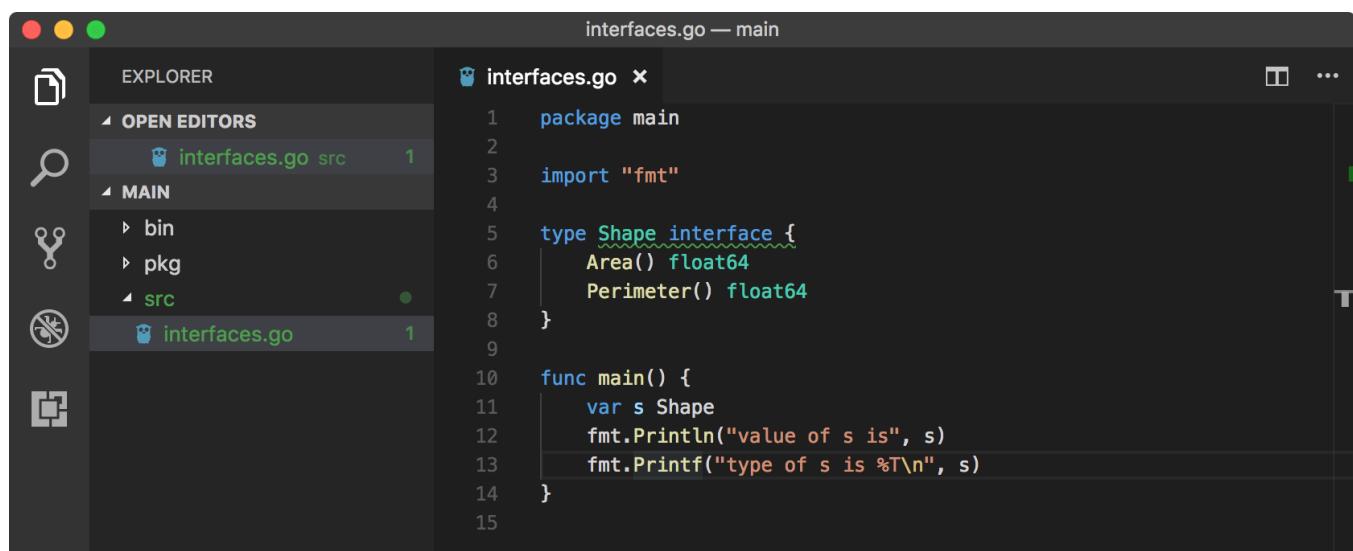
Like `struct`, we need to use type alias to simplify interface declaration along with keyword `interface`.

```
type Shape interface {
    Area() float64
    Perimeter() float64
}
```

The naming convention of interfaces in Go is a little bit tricky, you can follow this [Medium article](#) for more information.

Above, we have defined the `Shape` interface which has two methods `Area` and `Perimeter`, which accepts no arguments and return `float64`. Any type that implements these methods said to be implementing `Shape` interface.

Since `interface` is a type, we can create a variable of its type. In the above case, we can create a variable `s` of type interface `Shape`.



```
interfaces.go — main
1 package main
2
3 import "fmt"
4
5 type Shape interface {
6     Area() float64
7     Perimeter() float64
8 }
9
10 func main() {
11     var s Shape
12     fmt.Println("value of s is", s)
13     fmt.Printf("type of s is %T\n", s)
14 }
```

The screenshot shows a terminal window within the RunGo interface. The terminal output is:

```
main > go run src/interfaces.go
value of s is <nil>
type of s is <nil>
main >
```

The status bar at the bottom indicates: Ln 13, Col 13 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing.

<https://play.golang.org/p/oGRDKhrFJYh>

Before we explode our heads with the above result, let me explain something.

The **interface** has two types. A `static type` of the interface is interface itself, for example `Shape` in the above program. An interface does not have a `static value`, rather it points to a `dynamic value`. A variable of an interface type can **hold a value of the Type that implements the interface**. The value of that **Type** becomes `dynamic value` of the interface and that type becomes the `dynamic type` of the interface.

From the above result, we can see that **zero value and type of the interface is `nil`**. This is because, at this moment, the interface has no idea who is implementing it. When we use `Println` function from `fmt` package with **interface argument**, it points to the **dynamic value of the interface** and `%T` syntax in `Printf` function refers to the **dynamic type of interface**. But in reality, type of interface `s` is `Shape`.

☛ Implementing interface

Let's define `Area` and `Perimeter` methods with signatures provided by `Shape` interface. Also, let's create `Shape` struct and make it implement `Shape` interface.

The screenshot shows the VS Code interface with the file `interfaces.go` open in the editor. The code defines the `Shape` interface and the `Rect` struct.

```
package main
import "fmt"
type Shape interface {
    Area() float64
    Perimeter() float64
}
type Rect struct {
    width float64
    height float64
}
```

```

12 }
13 }
14
15 func (r Rect) Area() float64 {
16     return r.width * r.height
17 }
18
19 func (r Rect) Perimeter() float64 {
20     return 2 * (r.width + r.height)
21 }
22
23 func main() {
24     var s Shape
25     s = Rect{5.0, 4.0}
26     r := Rect{5.0, 4.0}
27     fmt.Printf("type of s is %T\n", s)
28     fmt.Printf("value of s is %v\n", s)
29     fmt.Println("area of rectangle s", s.Area())
30     fmt.Println("s == r is", s == r)
31 }
32

```

TERMINAL

```

main > go run src/interfaces.go
type of s is main.Rect
value of s is {5 4}
area of rectangle s 20
s == r is true
main >

```

JSONPath: Ln 22, Col 1 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing

https://play.golang.org/p/Hb_pA7Xp5V

So in the above program, we created our `Shape` interface and rectangle struct type `Rect`. Then we defined methods like `Area` and `Perimeter` with `Rect` receiver type. Hence `Rect` implemented those methods. Since these methods are defined by `Shape` interface, `Rect` implements `Shape` interface.

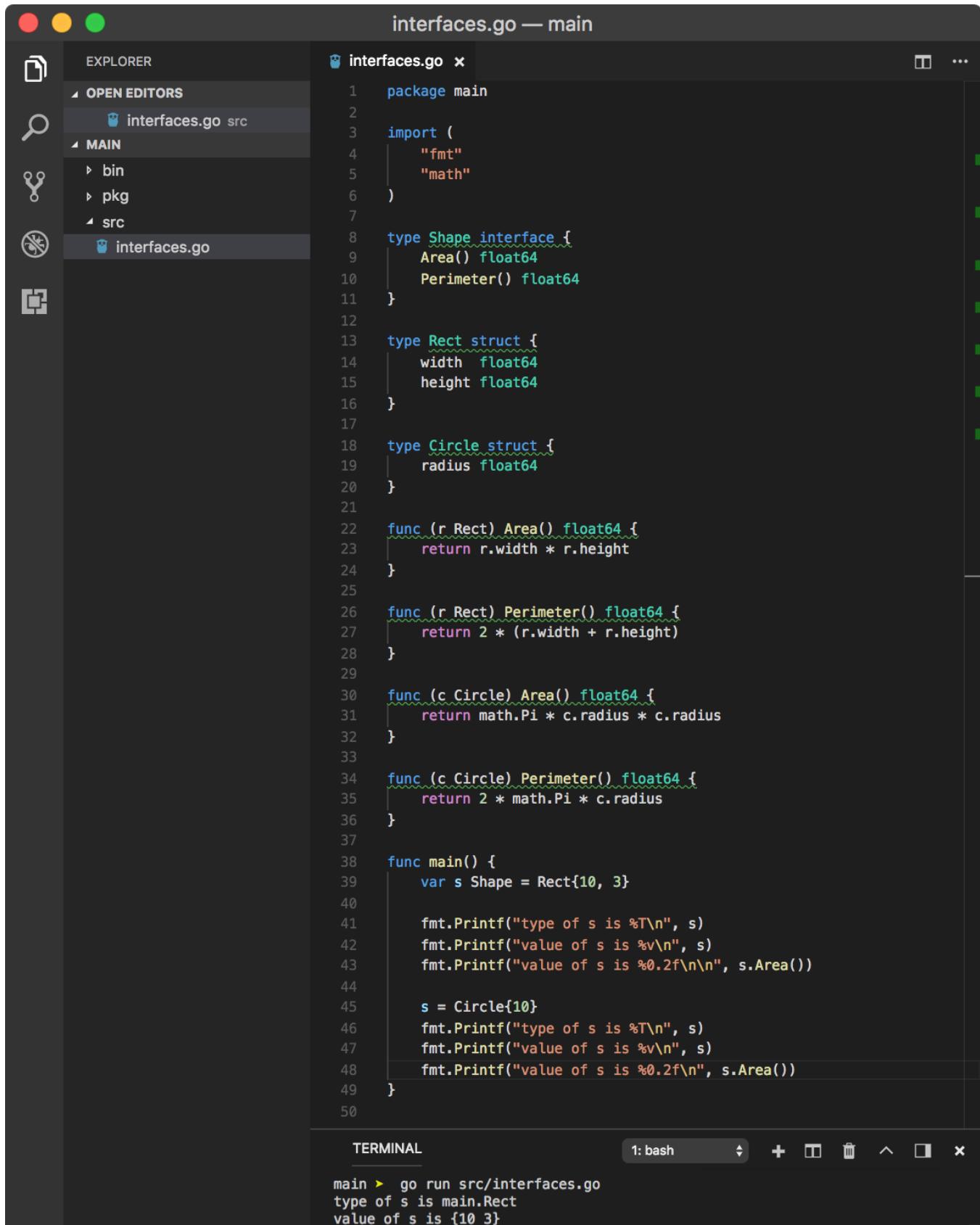
We can confirm that by creating `nil` interface `s` and assigning struct of type `Rect` to it that. Since `Rect` implements `Shape` interface, this is perfectly valid. From the above result, we can see that, dynamic type of `s` is now `Rect` and dynamic value of `s` is the value of the struct `Rect` which is `{5 4}`. Dynamic because we can assign new struct to it of a different type which also implements the interface `Shape`.

Sometimes, the dynamic type of interface also called as `concrete type` as when we access type of interface, it returns type of it's underlying dynamic value and it's static type remains hidden.

We can call `Area` method on `s` because the concrete type of `s` is `Rect` and `Rect` implements `Area` method. Also, we can see that we can

compare `s` with `r` struct of type `Rect` as both are of same `Rect` concrete type and has the same value.

Let's change the **dynamic type and value** of `s`.



```
interfaces.go — main
1 package main
2
3 import (
4     "fmt"
5     "math"
6 )
7
8 type Shape interface {
9     Area() float64
10    Perimeter() float64
11 }
12
13 type Rect struct {
14     width float64
15     height float64
16 }
17
18 type Circle struct {
19     radius float64
20 }
21
22 func (r Rect) Area() float64 {
23     return r.width * r.height
24 }
25
26 func (r Rect) Perimeter() float64 {
27     return 2 * (r.width + r.height)
28 }
29
30 func (c Circle) Area() float64 {
31     return math.Pi * c.radius * c.radius
32 }
33
34 func (c Circle) Perimeter() float64 {
35     return 2 * math.Pi * c.radius
36 }
37
38 func main() {
39     var s Shape = Rect{10, 3}
40
41     fmt.Printf("type of s is %T\n", s)
42     fmt.Printf("value of s is %v\n", s)
43     fmt.Printf("value of s is %.0f\n\n", s.Area())
44
45     s = Circle{10}
46     fmt.Printf("type of s is %T\n", s)
47     fmt.Printf("value of s is %v\n", s)
48     fmt.Printf("value of s is %.0f\n", s.Area())
49 }
50
```

TERMINAL 1: bash

```
main > go run src/interfaces.go
type of s is main.Rect
value of s is {10 3}
```

```
value of s is 30.00
type of s is main.Circle
value of s is {10}
value of s is 314.16
main > [REDACTED]
```

JSONPath: ["value of s is %0.2f\n"] Ln 48, Col 38 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing ☺ 🔔

<https://play.golang.org/p/K8mRGpfApJl>

If you read `structs` and `methods` lessons, then above program shouldn't surprise you. As new struct type `Circle` also implements `Shape` interface, we can assign `s` a struct of type `Circle`.

Now I guess you can understand why the **type** and **value** of the interface are dynamic. Like what we saw in `slices` lesson that a slice hold reference to an `array`, we can say that `interface` is also works in a similar way by **dynamically holding a reference to the underlying type**.

Can you guess what will happen with following program

```
interfaces.go — main
EXPLORER
OPEN EDITORS
MAIN
src
interfaces.go
interfaces.go

package main
import "fmt"
type Shape interface {
    Area() float64
    Perimeter() float64
}
type Rect struct {
    width float64
    height float64
}
func (r Rect) Area() float64 {
    return r.width * r.height
}
func main() {
    var s Shape = Rect{10, 3}
    fmt.Println(s)
}

TERMINAL
1: bash
main > go run src/interfaces.go
# command-line-arguments
src/interfaces.go:20:6: cannot use Rect literal (type Rect)
as type Shape in assignment:
    Rect does not implement Shape (missing Perimeter met
```

The screenshot shows the RunGo IDE interface. The code editor has tabs for 'hod' and 'main'. The status bar at the bottom shows 'JSONPath: Ln 22, Col 2 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing'. There are also icons for settings, Docker, and notifications.

```
hod)
main > [ ]
```

<https://play.golang.org/p/pwhlwfHFzF9>

In the above program, we removed `Perimeter` method. Above program won't compile and the compiler will throw an error

```
program.go:22: cannot use Rect literal (type Rect) as type Shape in
assignment:
    Rect does not implement Shape (missing Perimeter method)
```

It should be obvious from the above error that in order to successfully implement an interface, you need to implement all methods declared by the interface.

Empty interface

When an interface has zero methods, it is called an **empty interface**. This is represented by `interface{}`. Since empty interface has zero methods, all types implement this interface.

Have you wondered how `Println` function from `fmt` package can accept different types of data to print on console. This is possible because of an empty interface. Let's see how it works.

Let's create a function `explain` which accepts an empty interface and explains **dynamic value & type** of the interface.

The screenshot shows the VS Code interface with the file 'interfaces.go' open. The code defines two types: `MyString` (a string) and `Rect` (a struct with width and height). It then defines an `explain` function that takes an empty interface and prints its type and value using `fmt.Printf`. Finally, it calls `main.main`.

```
EXPLORER      interfaces.go — main
OPEN EDITORS  interfaces.go src 2
MAIN          bin
              pkg
              src
                interfaces.go 2
```

```
1 package main
2
3 import "fmt"
4
5 type MyString string
6
7 type Rect struct {
8     width float64
9     height float64
10 }
11
12 func explain(i interface{}) {
13     fmt.Printf("value given to explain function is of type '%T' with value %v\n", i, i)
14 }
15
16 func main() {
```

```

17 ms := MyString("Hello World!")
18 r := Rect{5.5, 4.5}
19 explain(ms)
20 explain(r)
21 }
22 
```

TERMINAL

```

main > go run src/interfaces.go
value given to explain function is of type 'main.MyString' with value Hello World!
value given to explain function is of type 'main.Rect' with value {5.5 4.5}
main > 
```

0 1 2 JSONPath: Ln 15, Col 1 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing

https://play.golang.org/p/NhvO6Qjw_zn

In the above program, we have created a custom string type `MyString` and a struct type `Rect`. Since `explain` function accepts empty interface, we can pass a variable of type `MyString` and `Rect` to it because of argument `i` of type **empty interface** can hold a value of any type as all types implement it.

➡️ Multiple interfaces

A type can implement multiple interfaces. Let's dive into an example.

interfaces.go — main

EXPLORER

- OPEN EDITORS
 - interfaces.go src 5
- MAIN
 - bin
 - pkg
 - src
 - interfaces.go 5

interfaces.go x

```

1 package main
2
3 import "fmt"
4
5 type Shape interface {
6     Area() float64
7 }
8
9 type Object interface {
10    Volume() float64
11 }
12
13 type Cube struct {
14     side float64
15 }
16
17 func (c Cube) Area() float64 {
18     return 6 * [c.side * c.side]
19 }
20
21 func (c Cube) Volume() float64 {
22     return c.side * c.side * c.side
23 }
24
25 func main() {
26     c := Cube{3}
27     var s Shape = c
28     var o Object = c
29     fmt.Println("volume of s of interface type Shape is", s.Area())
30     fmt.Println("area of o of interface type Object is", o.Volume())
31 }
32 
```

TERMINAL

```

main > go run src/interfaces.go
volume of s of interface type Shape is 54
area of o of interface type Object is 54
main > 
```

The screenshot shows the RunGo interface with a Go code editor. The code defines an interface `Shape` with `Area` and `Object` with `Volume`. A struct `Cube` implements both. A variable `s` is assigned to a `Cube` and `o` to an `Object`. Both print their respective values. The output window shows an error: "area of o of interface type Object is 27". The status bar at the bottom includes tabs for JSONPath, Line, Tab Size, and Encoding.

<https://play.golang.org/p/YgW3NBxp8Fh>

In the above program, we created `Shape` interface with `Area` method and `Object` interface with `Volume` method. Since struct type `Cube` implements both these methods, it implements both these interfaces. Hence we can assign a value of struct type `Cube` to the variable of type `Shape` or `Object`.

We expect `s` to have a dynamic value of `c` and `o` to also have a dynamic value of `c`. We used `Area` method on `s` of type `Shape` interface because it defines `Area` method and `Volume` method on `o` of type `Object` interface because it defines `Volume` method. But what will happen if we used `Volume` method on `s` and `Area` method on `o`.

Let's make changes to the above program to see what happens

```
fmt.Println("area of s of interface type Shape is", s.Area())
fmt.Println("volume of o of interface type Object is", o.Volume())
```

Above changes yield the following result

```
program.go:31: s.Volume undefined (type Shape has no field or method Volume)
program.go:32: o.Area undefined (type Object has no field or method Area)
```

The program won't compile because of **the static type** of `s` is `Shape` and of `o` is `Object`. To make it work, we need to somehow extract the underlying value of these interfaces. This can be done using **type assertion**.

☛ Type assertion

We can find out the underlying dynamic value of an interface using the syntax `i.(Type)` where `i` is an interface and `Type` is a type that implements the interface `i`. Go will check if dynamic type of `i` is identical to `Type`.

Hence let's re-write the previous example and extract the dynamic value of interface.

The screenshot shows the VS Code interface with the following details:

- EXPLORER:** Shows the project structure with files like 'interfaces.go' in the 'src' directory.
- EDITOR:** The code for 'interfaces.go' is displayed:


```

1 package main
2
3 import "fmt"
4
5 type Shape interface {
6     Area() float64
7 }
8
9 type Object interface {
10    Volume() float64
11 }
12
13 type Cube struct {
14     side float64
15 }
16
17 func (c Cube) Area() float64 {
18     return 6 * (c.side * c.side)
19 }
20
21 func (c Cube) Volume() float64 {
22     return c.side * c.side * c.side
23 }
24
25 func main() {
26     var s Shape = Cube{3}
27     c := s.(Cube)
28     fmt.Println("area of c of type Cube is", c.Area())
29     fmt.Println("volume of c of type Cube is", c.Volume())
30 }
31 
```
- TERMINAL:** Shows the output of running the program:


```

main > go run src/interfaces.go
area of c of type Cube is 54
volume of c of type Cube is 27
main > 
```
- STATUS BAR:** Shows '0 5' notifications, JSONPath: Ln 31, Col 1, Tab Size: 4, UTF-8, LF, Go, Analysis Tools Missing, and icons for smiley face and bell.

https://play.golang.org/p/0e1XTpjuXJ_e

From the above program, we now have access to the underlying value of the interface `s` in variable `c` which is a structure of type `Cube`. Now, we can use `Area` and `Volume` methods on `c`.

In type assertion syntax `i.(Type)`, if `Type` does not implement the interface (*the type of*) `i` then Go compiler will throw an error. But if `Type` implements the interface but `i` does not have a concrete value of `Type` then Go will panic in runtime. Luckily, there is another variant of type assertion syntax, which is

```
value, ok := i.(Type)
```

In the above syntax, we can check using `ok` variable if `Type` implements interface (*the type of*) `i` and `i` has concrete type `Type`. If it does, then `ok` will be `true`, else `false` with the `value` being **zero value of struct**.

I have one more question. **How would we know if the underlying value of an interface implements any other interfaces?** This is also possible with type assertion. If `Type` in type assertion syntax is `interface`, then **Go will check if dynamic type of i implements interface Type**.

The screenshot shows the VS Code interface with the following details:

- EXPLORER:** Shows the project structure with files: `interfaces.go`, `bin`, `pkg`, and `src/interfaces.go`.
- EDITOR:** The `interfaces.go` file is open, displaying the following Go code:

```
package main
import "fmt"

type Shape interface {
    Area() float64
}

type Object interface {
    Volume() float64
}

type Skin interface {
    Color() float64
}

type Cube struct {
    side float64
}

func (c Cube) Area() float64 {
    return 6 * [c.side * c.side]
}

func (c Cube) Volume() float64 {
    return c.side * c.side * c.side
}

func main() {
    var s Shape = Cube{3}
    value1, ok1 := s.(Object)
    fmt.Printf("dynamic value of Shape 's' with value %v implements interface Object? %v\n", value1, ok1)
    value2, ok2 := s.(Skin)
    fmt.Printf("dynamic value of Shape 's' with value %v implements interface Skin? %v\n", value2, ok2)
}
```
- TERMINAL:** Shows the output of the command `go run src/interfaces.go`.

```
dynamic value of Shape 's' with value {3} implements interface Object? true
dynamic value of Shape 's' with value <nil> implements interface Skin? false
```
- STATUS BAR:** Shows the status bar with tabs, file count (0), and other system information.

<https://play.golang.org/p/Iu84WAzDFwx>

As `Cube` struct does not implement `Skin` interface, we go `ok2` as `false` and `value2` as `nil`. If we would have used the simpler syntax of `v := i.(type)` syntax, then our program would have panicked with error

```
panic: interface conversion: main.Cube is not main.Skin: missing
method Color
```

 Take note here that, we also need to use type assertion for accessing a field of a struct which is represented by a variable of the type interface. In nutshell, accessing anything that is not represented by the interface type will cause in a runtime panic. So make sure to use type assertion when needed.

 Type assertion is not only used to check if an interface has a concrete value of some given type but to **convert a given variable of an interface type to a different interface type** as well. For example, if a struct implements an interface, then that struct can be converted to a type of interface by using type assertion, [check the example here](#). If the struct does not implement the interface passed to type assertion, it returns `nil` and `ok` will be false.

☛ Type switch

We have seen `empty interface` and its use. Let's think of an example using `explain` function as we saw earlier. As argument type of `explain` function is an empty interface, we can pass any argument to it. But if the argument passed is a string, we want to **explain function** to print it in uppercase. We can use `ToUpper` function from `strings` package but since it only accepts string argument, we need to make sure inside `explain` function that concrete type of empty interface `i` is `string`.

This can be done using **Type switch**. The syntax for type switch is similar to type assertion and it is `i.(type)` where `i` is interface and `type` is a fixed keyword. Using this we can get the concrete type of the interface instead of value. **But this syntax will only work in switch statement.**

Let's see an example



```

1 package main
2
3 import (
4     "fmt"
5     "strings"
6 )
7
8 func explain(i interface{}) {
9     switch i.(type) {
10    case string:
11        fmt.Println("i stored string ", strings.ToUpper(i.(string)))
12    case int:
13        fmt.Println("i stored int", i)
14    default:
15        fmt.Println("i stored something else", i)
16    }
17 }
18
19 func main() {
20     explain("Hello World")
21     explain(52)
22     explain(true)
23 }
24

```

TERMINAL

```

main > go run src/interfaces.go
i stored string HELLO WORLD
i stored int 52
i stored something else true
main >

```

JSONPath: Error. Ln 10, Col 9 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing

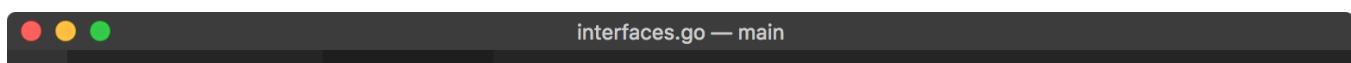
<https://play.golang.org/p/lSSq3VDMbR>

In the above program, we modified `explain` function to use `type switch`. When an `explain` function is called with any type, `i` receives its **value and type as the dynamic type**. Using `i.(type)` statement inside switch, we are getting access to that dynamic type. Using cases inside the switch statement, we can do conditional operations. In case of `string` case, we used `strings.ToUpper` function to convert string to uppercase. But as it accepts only `string` type, we needed the underlying value of `i` which is of string type, hence we used `type assertion`.

A `nil` interface has a `nil` type.

➡ Embedding interfaces

In Go, an interface cannot implement other interfaces or extend them, but we can create new interface by merging two or more interfaces. Let's rewrite our `Shape-Cube` program.



The screenshot shows the RunGo IDE interface. The left sidebar has sections for EXPLORER, OPEN EDITORS, and MAIN. In the OPEN EDITORS section, there are two files: 'interfaces.go' (src) and 'interfaces.go' (MAIN). The main area displays the code for 'interfaces.go' (src), which defines three interfaces: Shape, Object, and Material. It also defines a struct Cube that implements Area and Volume, and shows how Material is an embedded interface of Shape and Object. The terminal at the bottom shows the output of running the program, which prints the dynamic type and value of each interface.

```

1 package main
2
3 import "fmt"
4
5 type Shape interface {
6     Area() float64
7 }
8
9 type Object interface {
10    Volume() float64
11 }
12
13 type Material interface {
14     Shape
15     Object
16 }
17
18 type Cube struct {
19     side float64
20 }
21
22 func (c Cube) Area() float64 {
23     return 6 * (c.side * c.side)
24 }
25
26 func (c Cube) Volume() float64 {
27     return c.side * c.side * c.side
28 }
29
30 func main() {
31     c := Cube{3}
32     var m Material = c
33     var s Shape = c
34     var o Object = c
35     fmt.Printf("dynamic type and value of interface m of static type Material is '%T' and '%v'\n", m, m)
36     fmt.Printf("dynamic type and value of interface s of static type Shape is '%T' and '%v'\n", s, s)
37     fmt.Printf("dynamic type and value of interface o of static type Object is '%T' and '%v'\n", o, o)
38 }
39

```

TERMINAL

```

main > go run src/interfaces.go
dynamic type and value of interface m of static type Material is 'main.Cube' and '{3}'
dynamic type and value of interface s of static type Shape is 'main.Cube' and '{3}'
dynamic type and value of interface o of static type Object is 'main.Cube' and '{3}'
main >

```

JSONPath: Ln 24, Col 2 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing

<https://play.golang.org/p/s2U79IDaKqE>

In the above program, since `Cube` implements method `Area` and `Volume`, it implements interfaces `Shape` and `Object`. But since the interface `Material` is an embedded interface of these interfaces, `Cube` must also implement it. This happens because like `anonymous nested struct`, all methods of nested interfaces get promoted to parent interfaces.

pointer VS value receiver

So far we have seen methods with value receiver. Will interface be ok with method accepting pointer receiver. Let's it check out.

The screenshot shows the RunGo IDE interface with a modified version of the code. The 'interfaces.go' file now includes a line '5 type Shape interface {'. This change makes the `Area` method a pointer receiver, which typically causes a compile error for interfaces. The rest of the code remains the same, defining the `Object` and `Material` interfaces and the `Cube` struct.

```

1 package main
2
3 import "fmt"
4
5 type Shape interface {
6     Area() float64
7 }
8
9 type Object interface {
10    Volume() float64
11 }
12
13 type Material interface {
14     Shape
15     Object
16 }
17
18 type Cube struct {
19     side float64
20 }
21
22 func (c Cube) Area() float64 {
23     return 6 * (c.side * c.side)
24 }
25
26 func (c Cube) Volume() float64 {
27     return c.side * c.side * c.side
28 }
29
30 func main() {
31     c := Cube{3}
32     var m Material = c
33     var s Shape = c
34     var o Object = c
35     fmt.Printf("dynamic type and value of interface m of static type Material is '%T' and '%v'\n", m, m)
36     fmt.Printf("dynamic type and value of interface s of static type Shape is '%T' and '%v'\n", s, s)
37     fmt.Printf("dynamic type and value of interface o of static type Object is '%T' and '%v'\n", o, o)
38 }
39

```

The screenshot shows the RunGo IDE interface. On the left, there's a file tree with a package named 'pkg' containing a 'src' directory with a file 'interfaces.go'. The code editor on the right contains the following Go code:

```

6     Area() float64
7     Perimeter() float64
8 }
9
10 type Rect struct {
11     width float64
12     height float64
13 }
14
15 func (r *Rect) Area() float64 {
16     return r.width * r.height
17 }
18
19 func (r Rect) Perimeter() float64 {
20     return 2 * (r.width + r.height)
21 }
22
23 func main() {
24     r := Rect{5.0, 4.0}
25     var s Shape = r
26     area := s.Area()
27     fmt.Println("area of rectangle is", area)
28 }
29

```

Below the code editor is a terminal window titled 'TERMINAL' showing the command 'go run src/interfaces.go' being run. The output indicates a compilation error:

```

main > go run src/interfaces.go
# command-line-arguments
src/interfaces.go:25:6: cannot use r (type Rect) as type Shape in assignment:
    Rect does not implement Shape (Area method has pointer receiver)
main >

```

The status bar at the bottom shows 'JSONPath: Error.' and other tabs.

<https://play.golang.org/p/vEkRuYo1JKu>

Above program will not compiler and Go will throw compilation error

program.go:27: cannot use Rect literal (type Rect) as type Shape in assignment: Rect does not implement Shape (Area method has pointer receiver)

What the hell? We can clearly see that struct type `Rect` is implementing all methods stated by interface `Shape`, then why we are getting `Rect does not implement Shape` error. If you read error carefully, it says `Area method has pointer receiver`. So what if `Area method has pointer receiver`.

Well, we have seen in `structs` lesson that a method with pointer receiver will work on both pointer or value and if we would have used `r.Area()` in the above program, it would have compiled just fine.

But in case of interfaces, if a method has a pointer receiver, then the **interface will have a pointer of dynamic type rather than the value of dynamic type**.

Hence, instead of assigning a value to an interface variable, we need to assign a pointer. Let's rewrite the above program with this concept.

```

EXPLORER          interfaces.go — main
OPEN EDITORS      interfaces.go src 4
MAIN
  bin
  pkg
  src
    interfaces.go 4
INTERACTOR
TERMINAL          1: bash
JSONPath: Error. Ln 27, Col 31 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing
https://play.golang.org/p/3OY4dBOsXdl

```

```

1 package main
2
3 import "fmt"
4
5 type Shape interface {
6     Area() float64
7     Perimeter() float64
8 }
9
10 type Rect struct {
11     width float64
12     height float64
13 }
14
15 func (r *Rect) Area() float64 {
16     return r.width * r.height
17 }
18
19 func (r Rect) Perimeter() float64 {
20     return 2 * (r.width + r.height)
21 }
22
23 func main() {
24     r := Rect{5.0, 4.0}
25     var s Shape = &r // assigned pointer
26     area := s.Area()
27     perimeter := s.Perimeter()
28     fmt.Println("area of rectangle is", area)
29     fmt.Println("perimeter of rectangle is", perimeter)
30 }
31

```

The only change we made is in line no. 25 where instead of value of `r`, we used the pointer to `r`. Hence the concrete value of `s` is now a pointer. The above program will compile fine.

Use of interfaces

We have learned interfaces and we saw they can take different forms. That's the definition of `polymorphism`. Interfaces are very useful in case of `functions` and `methods` where you need argument of many types to be passed to them, like `Println` function which accepts all types of values. If you see the syntax of `Println` function, it is like

```
func Println(a ...interface{}) (n int, err error)
```

which is also a variadic function.

When multiple types implement the same interface, it becomes easy to work with them using the same code. Hence whenever we can use interfaces, we should.