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

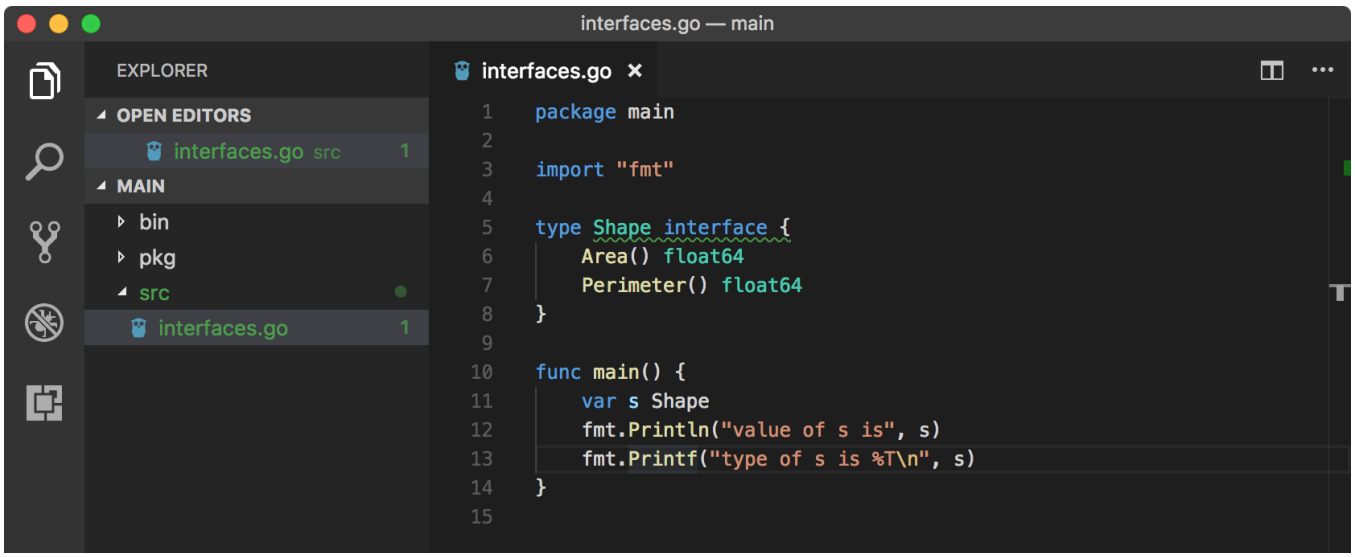
Like `struct`, we need to use type alias to simplify interface declaration along with keyword `interface`.

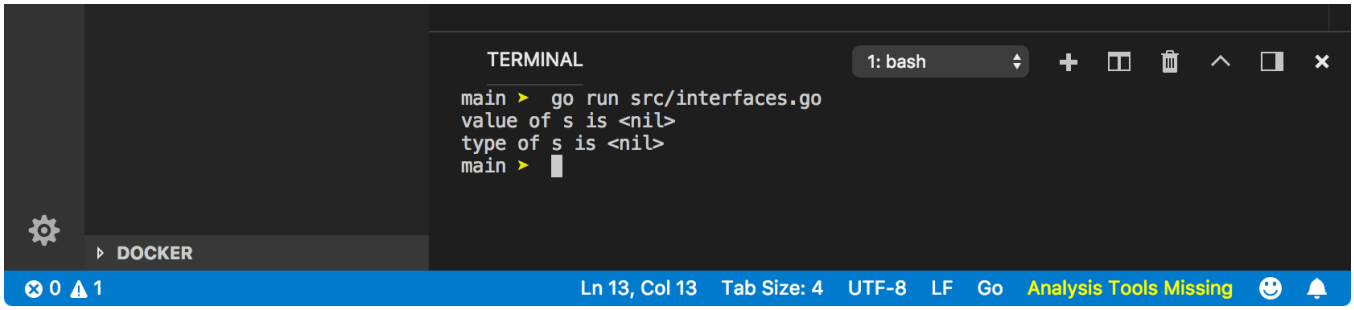
```
type Shape interface {
    Area() float64
    Perimeter() float64
}
```

> *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`.





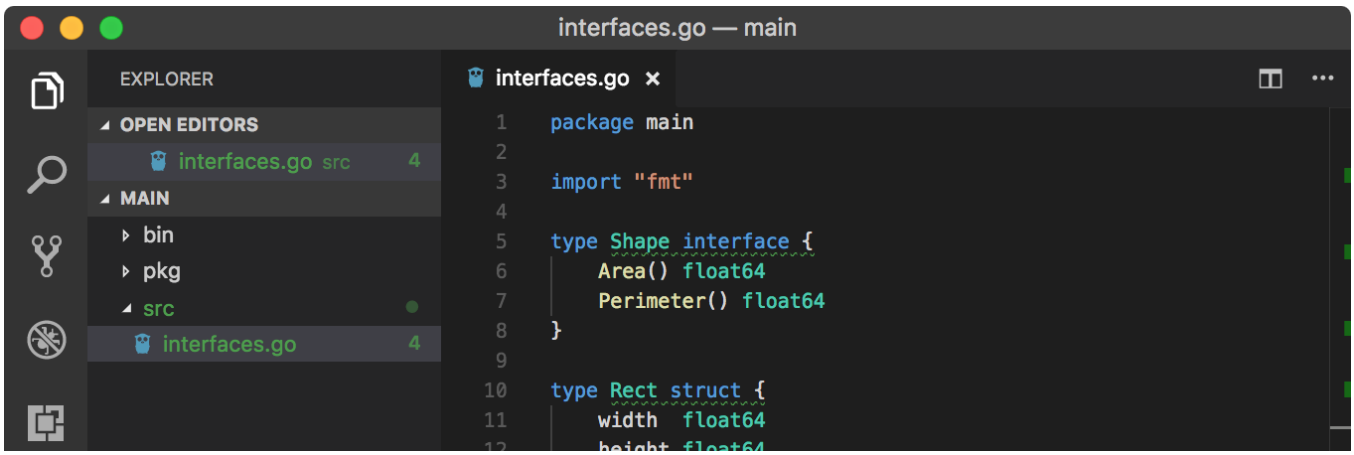
https://play.golang.org/p/oGRDKbrFJYb

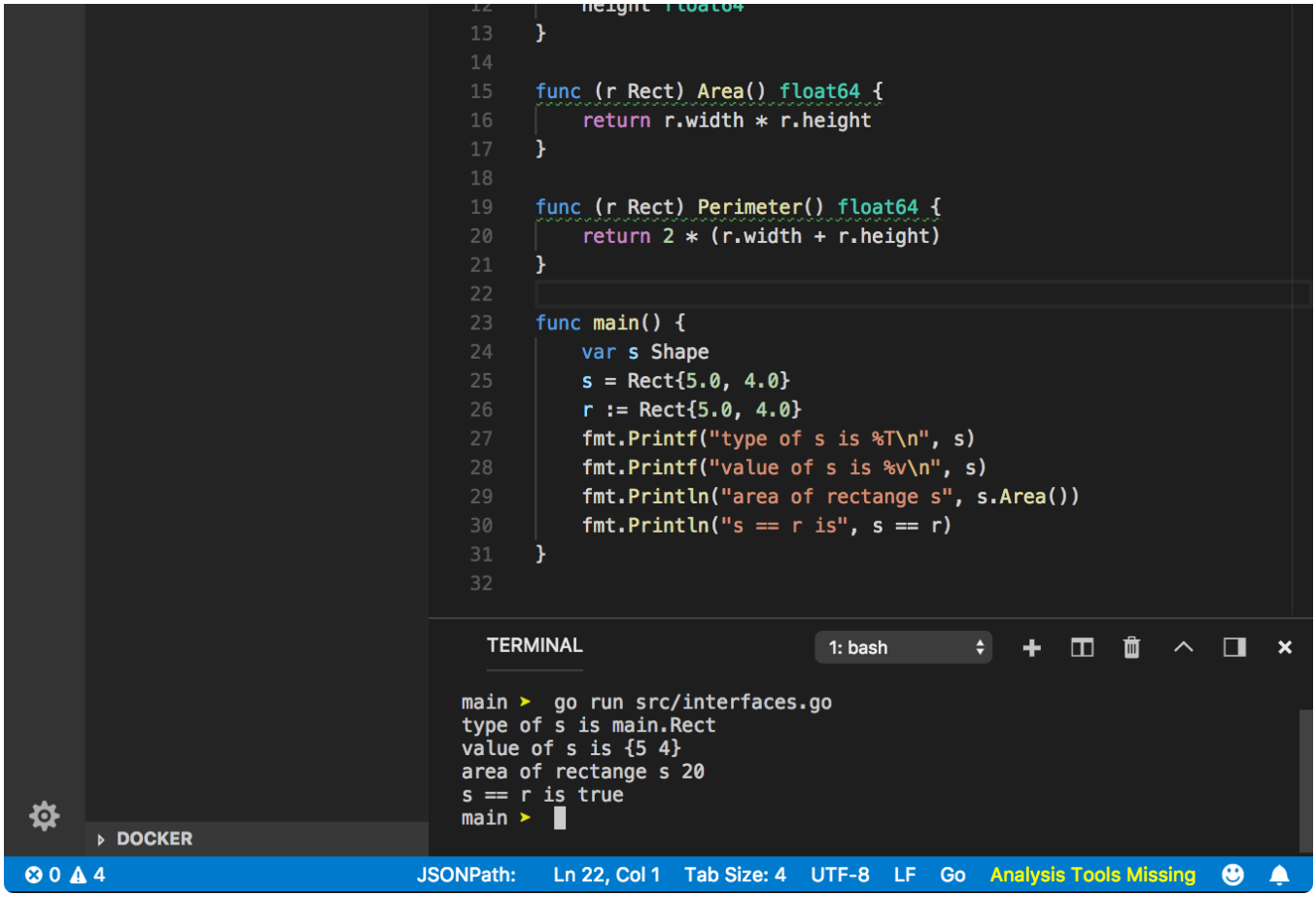
Before we explode our heads with the above result, let me explain something. The **interface** has two types. A `static type` of the interface is interface itself, for example `Shape` in the above program. An interface does not have a `static value`, rather it points to a `dynamic value`. A variable 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.





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.

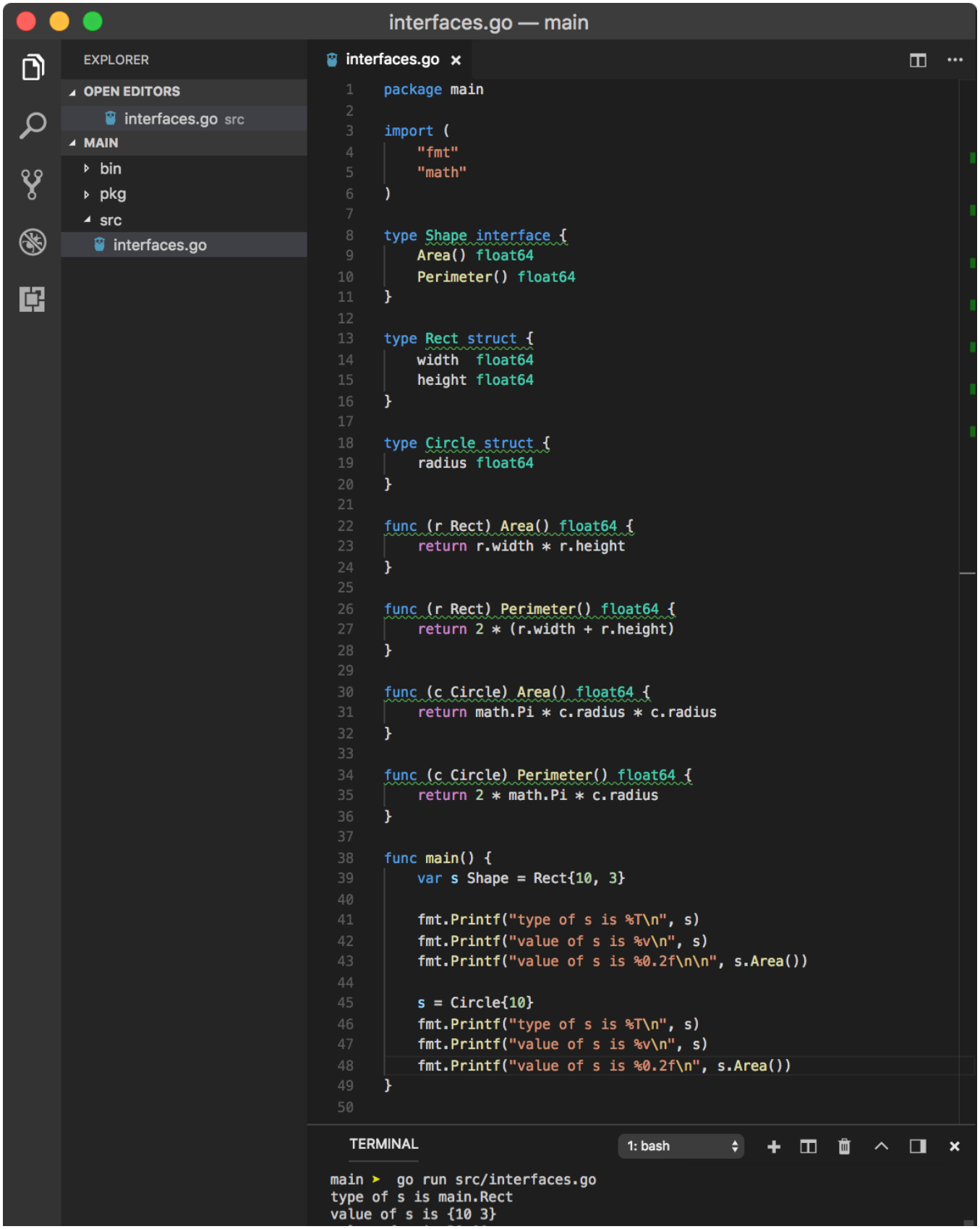
We can confirm that by creating `nil` interface `s` and assigning struct of type `Rect` to it that. Since `Rect` implements `Shape` interface, this is perfectly valid. From the above result, we can see that, dynamic type of `s` is now `Rect` and dynamic value of `s` is the value of the struct `Rect`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

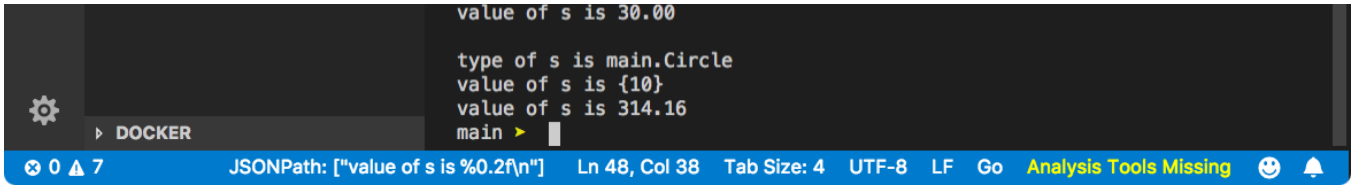
Sometimes, the dynamic type of interface also called as `concrete type` as when we access type of interface, it returns type of it's underlying dynamic value and it's static type remains hidden.

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`.



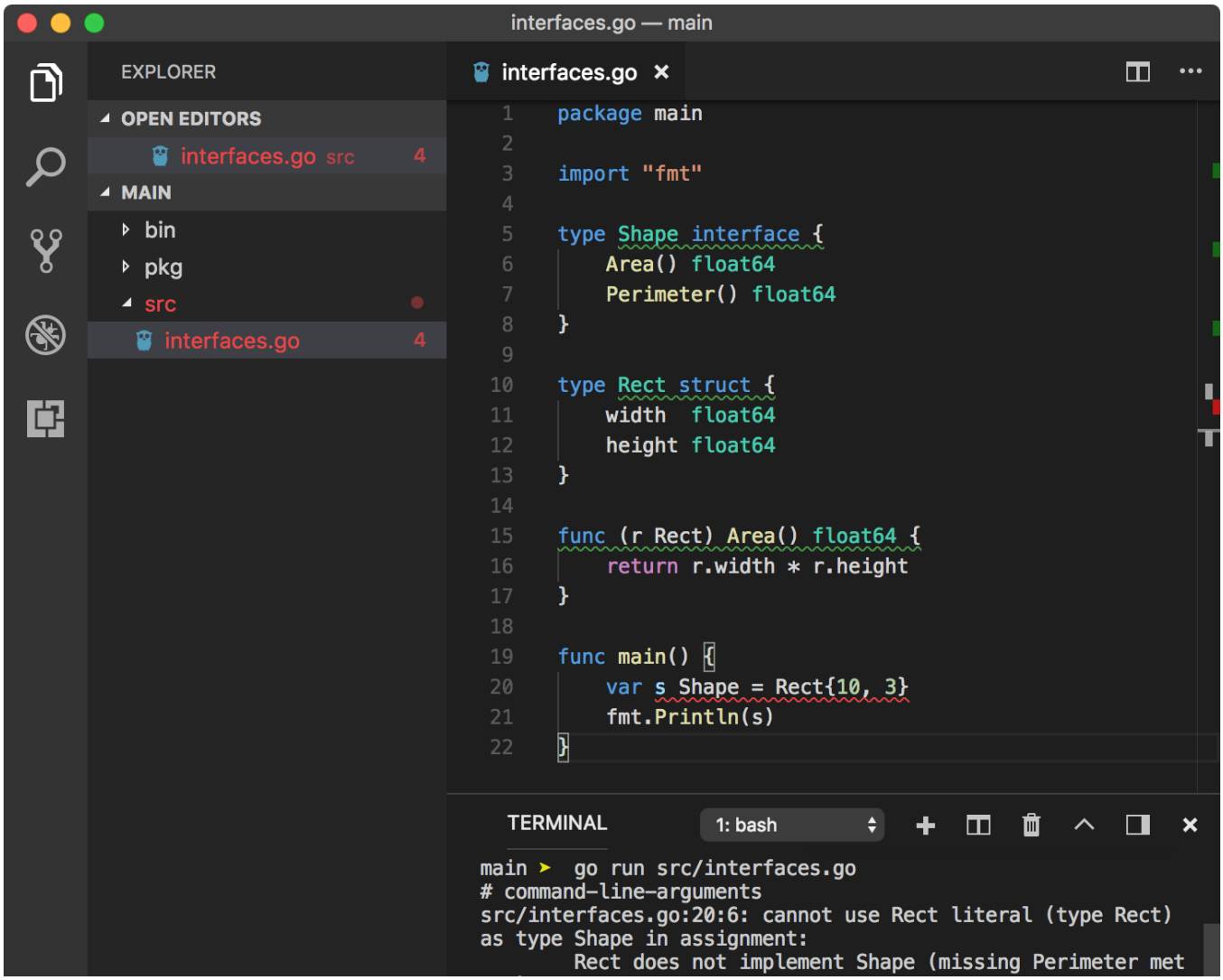


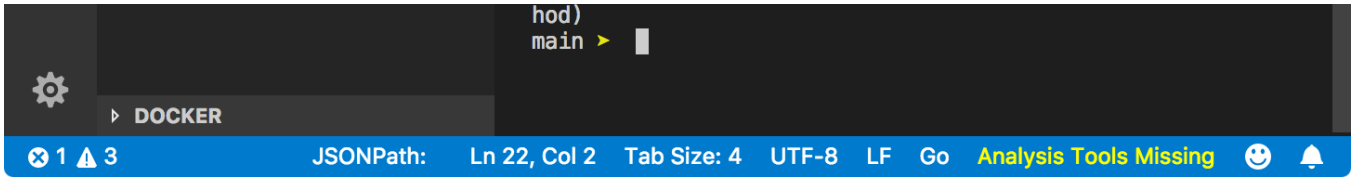
https://play.golang.org/p/K8mBGpfApjJ

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





https://play.golang.org/p/pwhIwfHFzF9

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)
```

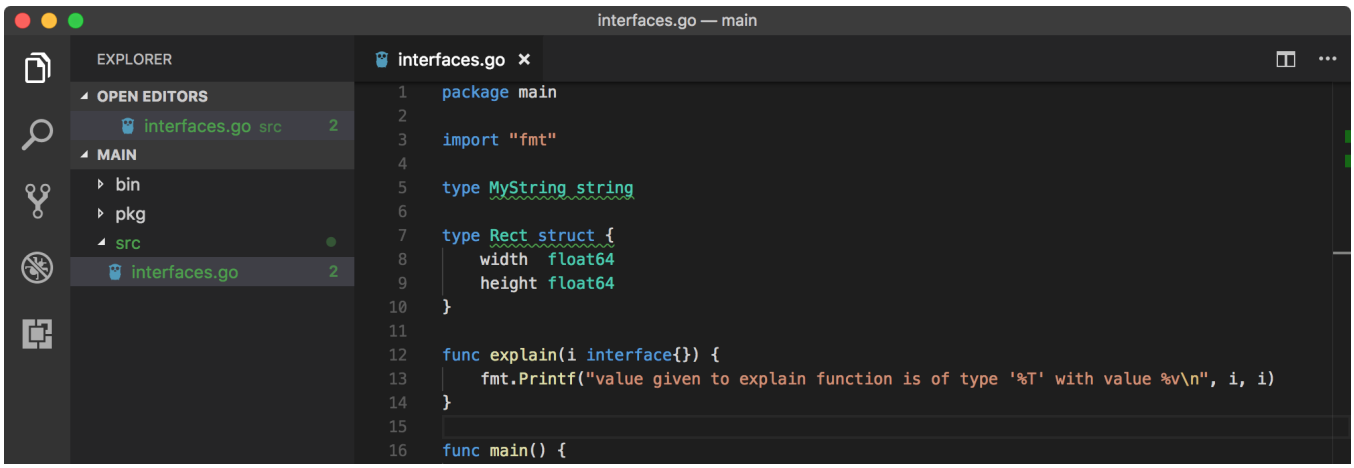
It should be obvious from the above error that in order to successfully implement an interface, you need to implement all methods declared by the interface.

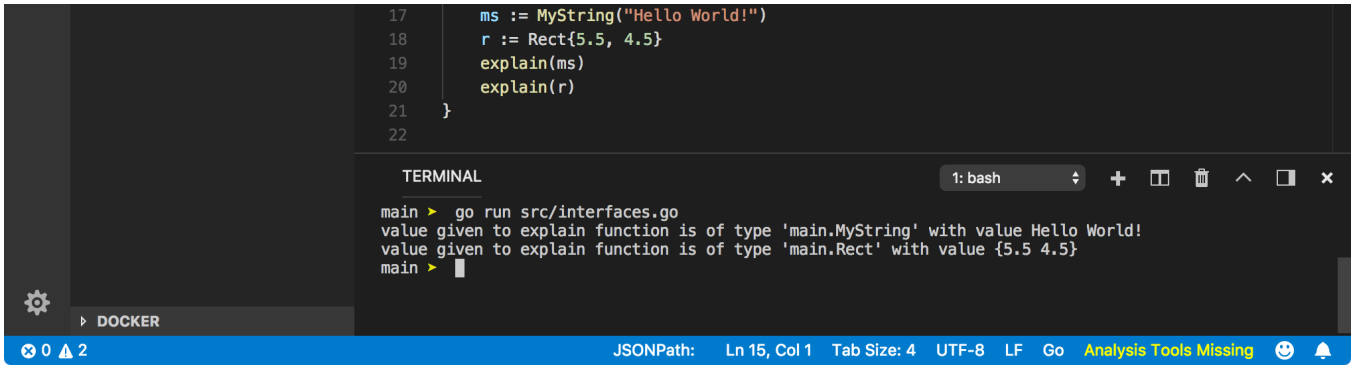
☛ Empty interface

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.



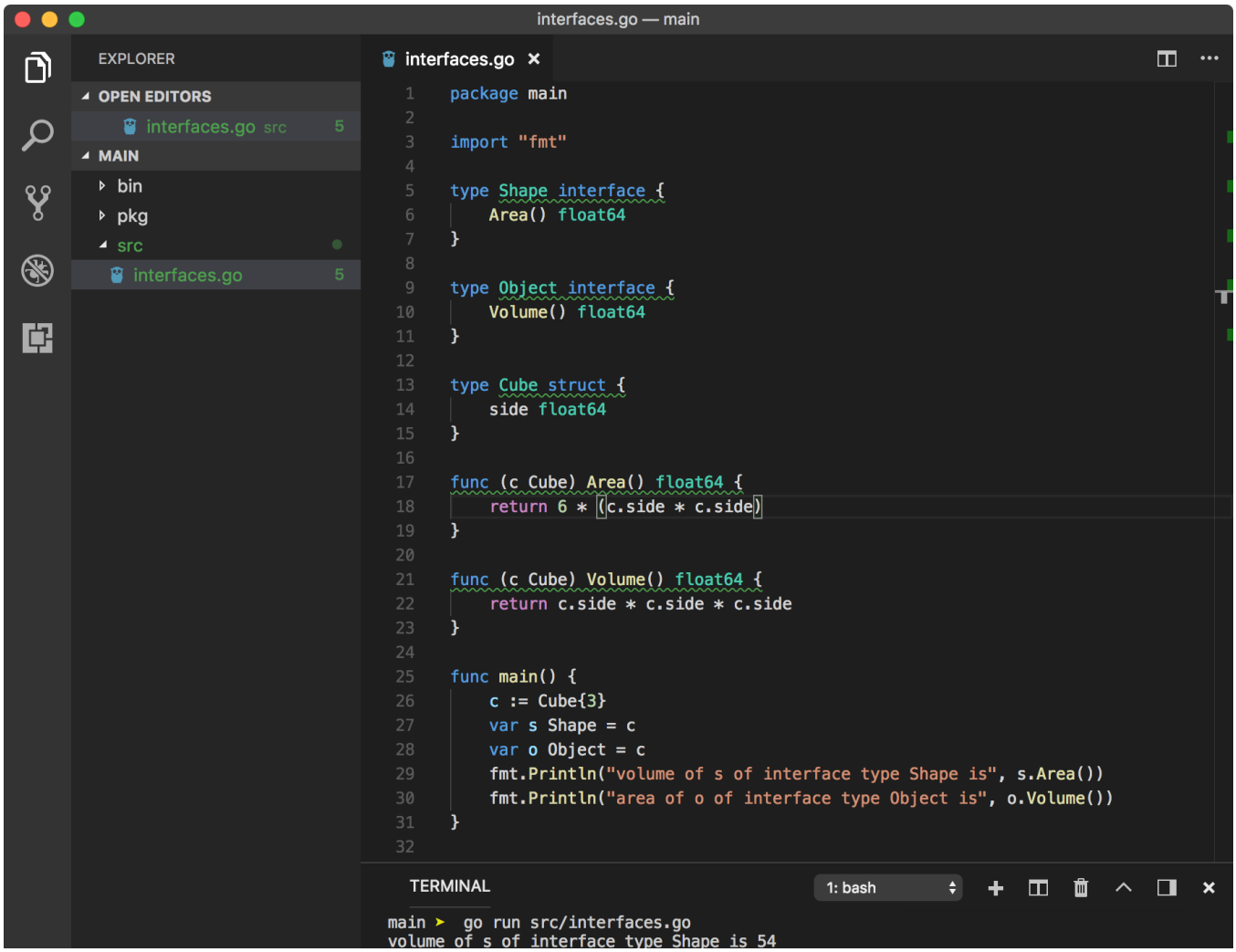


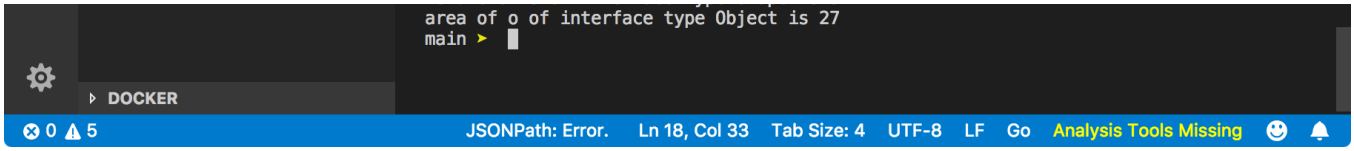
https://play.golang.org/p/NhvO6Qjw_zp

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.





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

Above changes yield the following result

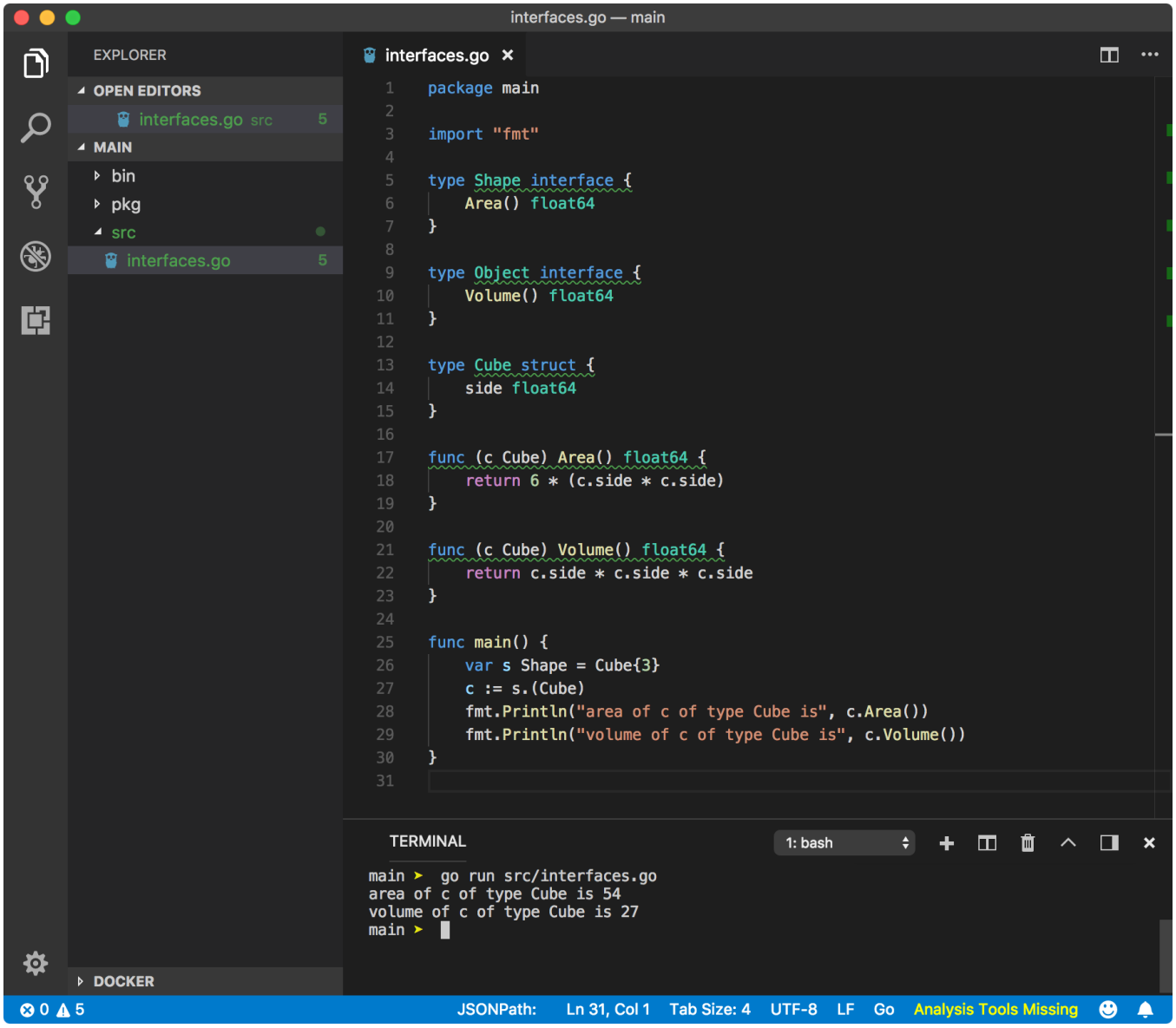
```
program.go:31: s.Volume undefined (type Shape has no field or method
Volume)
program.go:32: o.Area undefined (type Object has no field or method
Area)
```

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.



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

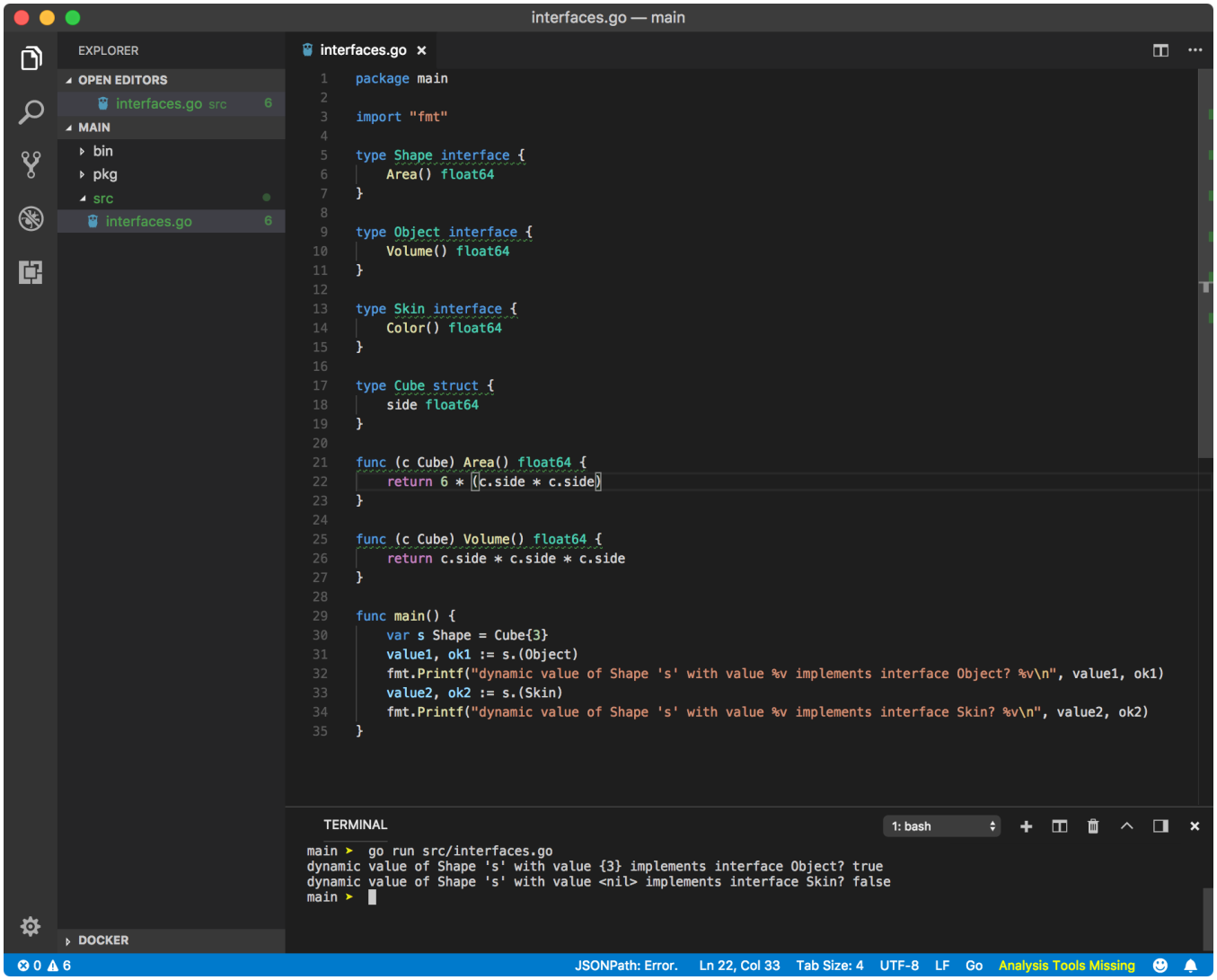
From the above program, we now have access to the underlying value of the interface `s` in variable `c` which is a structure of type `Cube`. Now, we can use `Area` and `Volume` methods on `c`.

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`**.



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.*

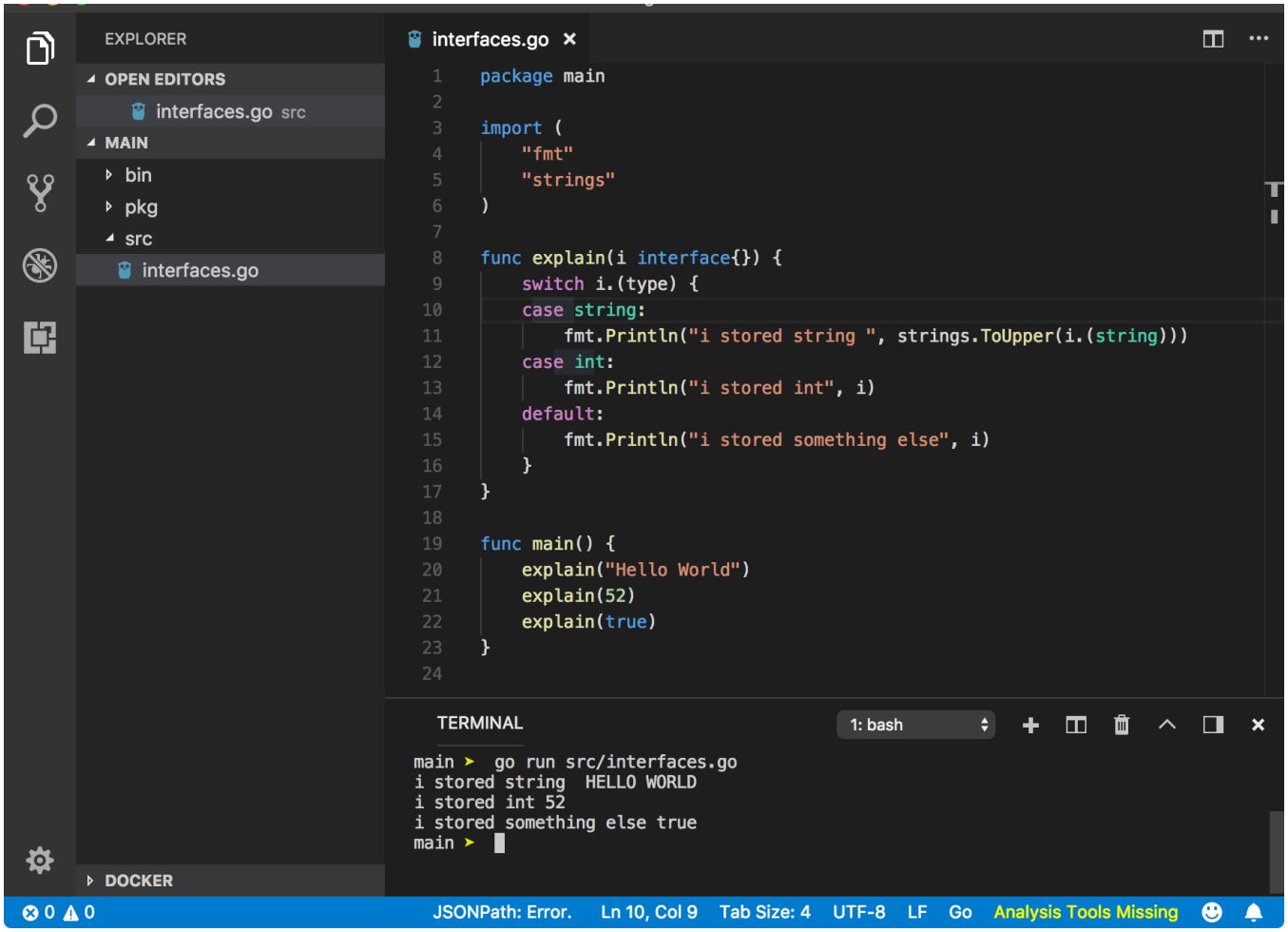
🌞 Type assertion is not only used to check if an interface has a concrete value of some given type but to **convert a given variable of an interface type to a different interface type** as well. For example, if a struct implements an interface, then that struct can be converted to a type of interface 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



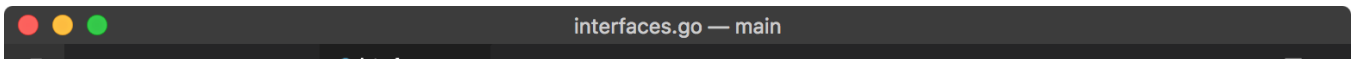
https://play.golang.org/p/ItSSq3VDMbB

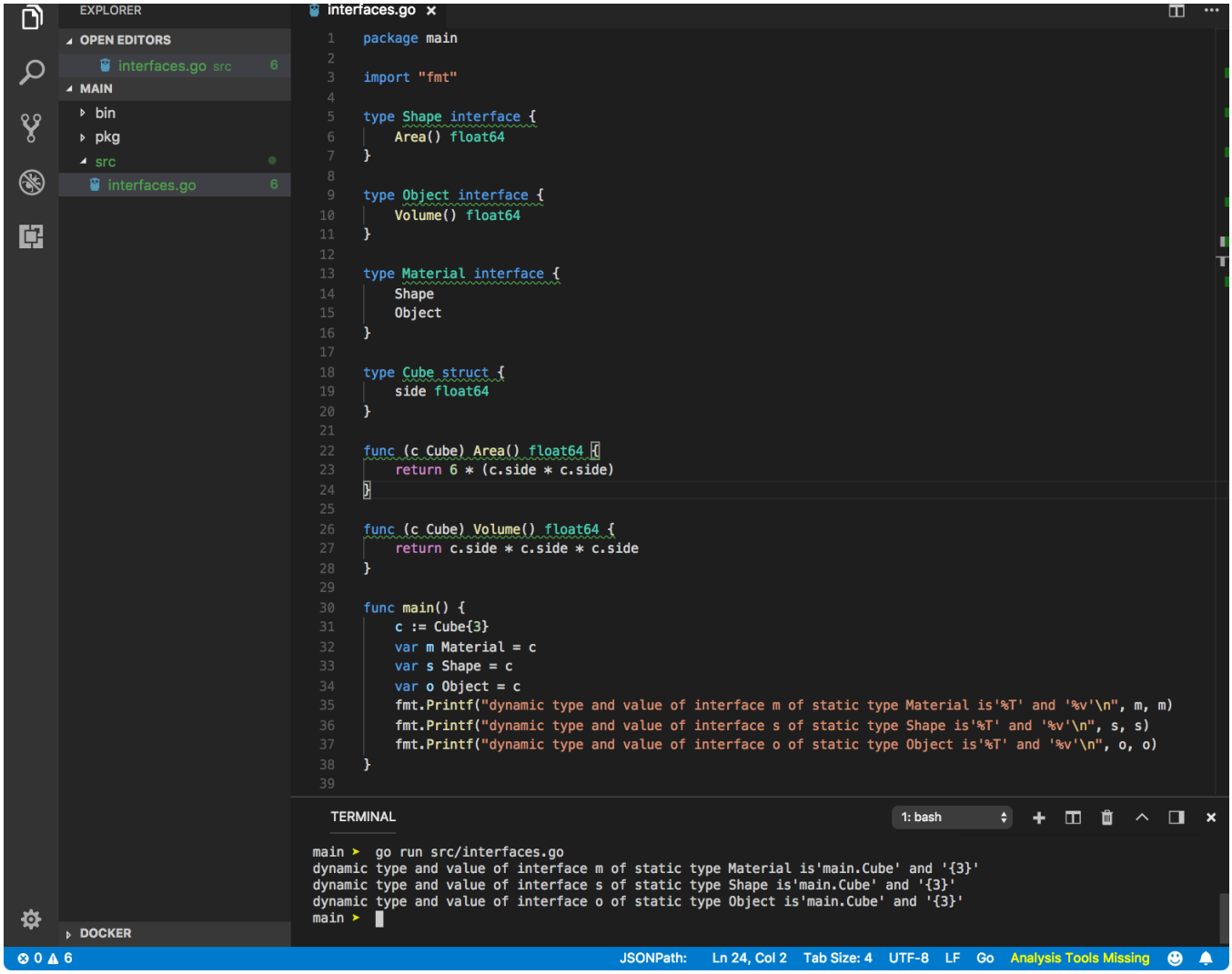
In the above program, we modified `explain` function to use `type switch`. When an explain function is called with any type, `i` receives its **value and type as the dynamic type**. Using `i.(type)` statement inside switch, we are getting access to that dynamic type. Using cases inside the switch statement, 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.



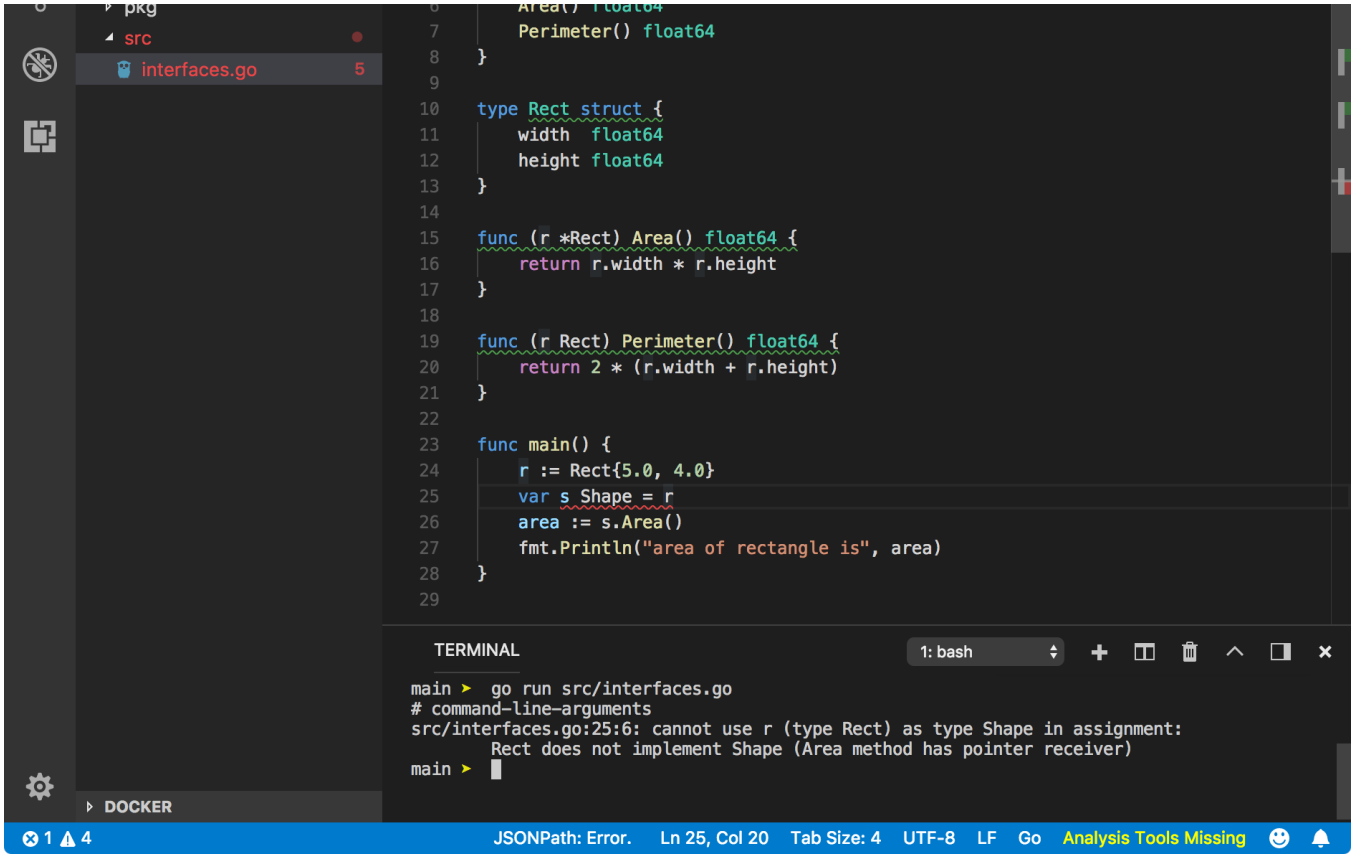


https://play.golang.org/p/s2U79IDaKqE

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.



https://play.golang.org/p/vFkRuYo1JKu

Above program will not compiler and Go will throw compilation error

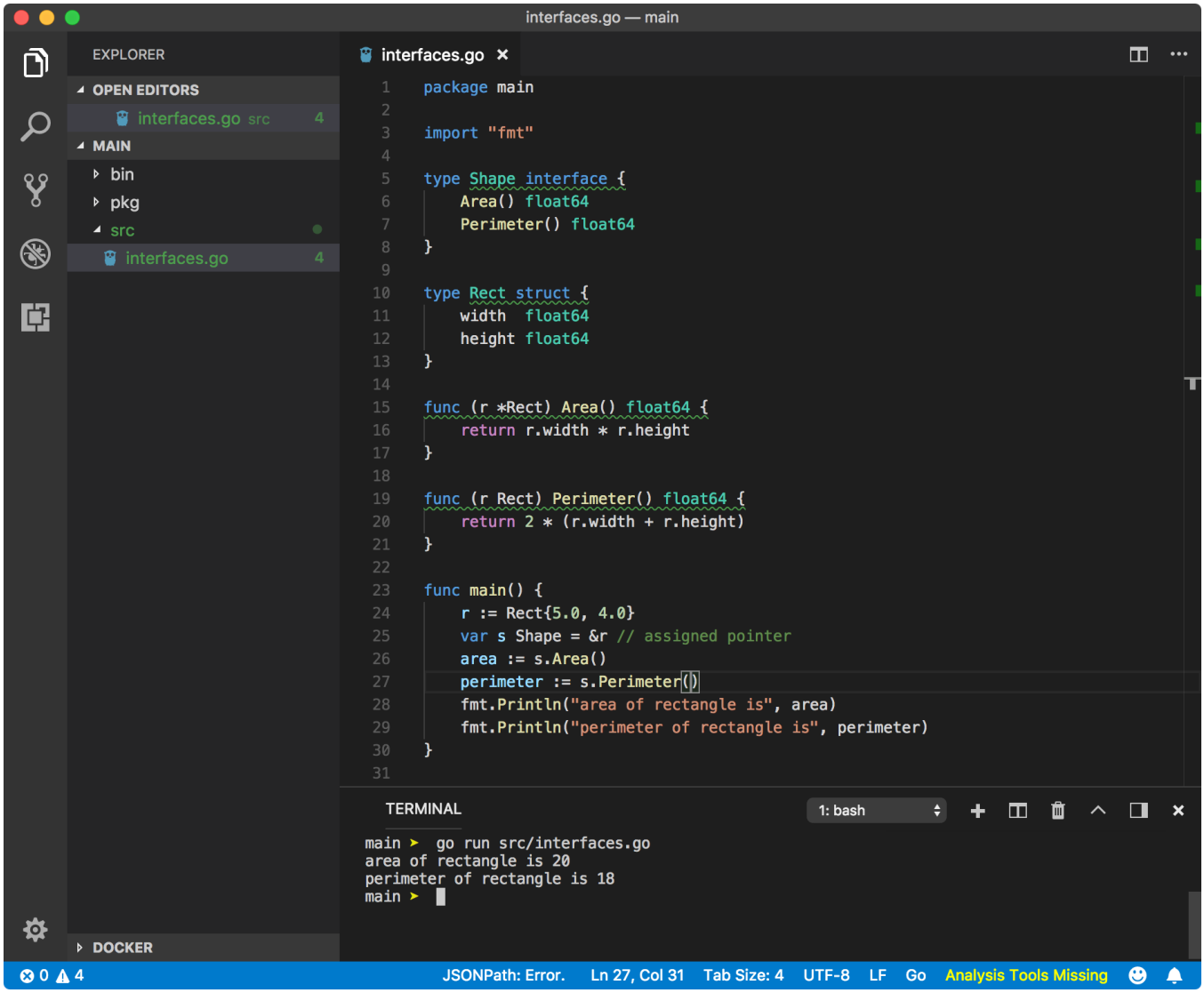
```
program.go:27: cannot use Rect literal (type Rect) as type Shape in
assignment: Rect does not implement Shape (Area method has pointer
receiver)
```

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 ins `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.



https://play.golang.org/p/3OY4dBOSXdL

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.