Empty Interface

Sometimes we have no idea about the types that can be used by the code. Therefore, Go provides us with 'empty interfaces' that let us deal with these values. This lesson shows us how to work with them.

WE'LL COVER THE FOLLOWING

Introduction

The *empty* or *minimal interface* has no methods, and so it doesn't make any demands at all.

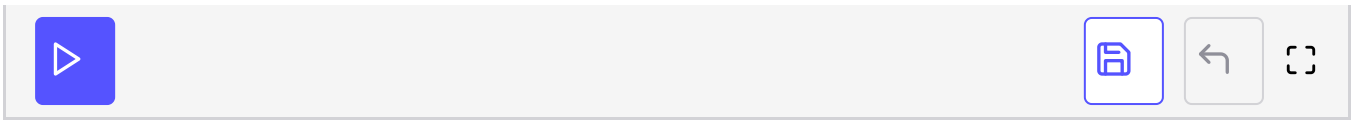
```
type Any interface{}
```

Any variable, any type implements it (not only reference types that inherit from Object in Java/C#), and *any* or *Any* is a good name as an alias and abbreviation! (It is analogous to the class Object in Java and C#, the base class of all classes. So, Obj would also fit.) A variable of that interface type `var val interface{}` can through assignment receive a variable of any type.

This is illustrated in the program:

```
package main
import "fmt"

var i = 5
var str = "ABC"
```

```
type Person struct {
        name string
        age  int

}

type Any interface{}    // empty interface

func main() {
        var val Any
        val = 5 // assigning integer to empty interface
        fmt.Printf("val has the value: %v\n", val)
        val = str // assigning string to empty interface
        fmt.Printf("val has the value: %v\n", val)
        pers1 := new(Person)
        pers1.name = "Rob Pike"
        pers1.age = 55
        val = pers1 // assigning *Person type variable to empty interface
        fmt.Printf("val has the value: %v\n", val)
        switch t := val.(type) {        // cases defined on type of val
        case int:       // if val is int
                fmt.Printf("Type int %T\n", t)
        case string: // if val is string
                fmt.Printf("Type string %T\n", t)
        case bool: // if val is bool
                fmt.Printf("Type boolean %T\n", t)
        case *Person:   // if val is *Person
                fmt.Printf("Type pointer to Person %T\n", *t)
        default:        // None of the above types
                fmt.Printf("Unexpected type %T", t)
        }
}
```



Empty Interface

In the above code, at **line 4** and **line 5**, we make two global variables `i` and `str`, respectively, and set their values. Then, at **line 7**, we define a `Person` type struct with two fields `name` (a string) and `age` (an integer). Next, at **line 12**, we make an empty interface `Any`.

Now, look at `main`. We make `Any` type variable `val` at **line 15**. In the next line, we set `val` to **5**. At **line 18**, `val` gets the value of `str`, so in the next line, **ABC** (value of `str`) will be printed.

At **line 20**, we make `pers1` a `Person` pointer-type variable. In the next two lines, we give a value to the internal fields (**Rob Pike** as `name` and **55** as `age`) of `pers1`. At **line 23**, `val1` is given the value of `pers1`. In the next line, `val` is printed, which prints the `pers1` struct as **&{Rob Pike 55}**.

Now, we have switch cases ahead. Cases are to be judged on *type* of `val`.

- The first case will be true if `val` is of *int* type.
- The second case will be true if `val` is of *string* type.
- The third case will be true if `val` is of *bool* type.
- The fourth case will be true if `val` is of `*Person` type.

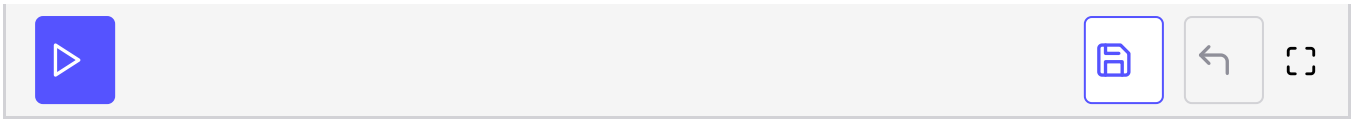
Now, the type will be printed accordingly when one case is found true. In this case, the last updated value of `val` was `pers1` of type `*Person`. So, the third case will become true, and **line 33** will be executed.

What if, no case is found true? In this case, we have a *default* case that prints the message of the *unexpected* type.

Each `interface{}` variable takes up 2 words in memory: one word for the type of what is contained, and the other word for either the contained data or a pointer to it.

The following program is an example of usage of the empty interface in a type switch combined with a lambda function:

```
package main
import "fmt"

type specialString string

var whatIsThis specialString = "hello"

func TypeSwitch() {
        testFunc := func(any interface{}) {     // lambda function in combination with empty
                switch v := any.(type) {
                case bool: // if v is bool
                        fmt.Printf("any %v is a bool type", v)
                case int: // if v is int
                        fmt.Printf("any %v is an int type", v)
                case float32: // if v is float32
                        fmt.Printf("any %v is a float32 type", v)
                case string: // if v is string
                        fmt.Printf("any %v is a string type", v)
                case specialString: // if v is specialString
                        fmt.Printf("any %v is a special String!", v)
                default: // none of types satisfied
                        fmt.Println("unknown type!")
                }
        }
        testFunc(whatIsThis)
}

func main() {
        TypeSwitch()
}
```



Type Switch and Lambda Function

In the above code, we define a new type `specialString` on the basis of `string` type at **line 4**. Then, at **line 6**, we create a variable `whatIsThis` of type `specialString` and give it a value **hello**. Now, we have a `typeSwitch` function at **line 8**. In the next line, we declare a *function* `testFunc`. This function can take an empty interface as its parameter.

At **line 10**, we are making a switch statement with cases dependent on the *type* of `v` passed to `testFunc`.

- The first case will be true if `v` is of *bool* type.
- The second case will be true if `v` is of *int* type.
- The third case will be true if `v` is of *float32* type.
- The fourth case will be true if `val` is of *string* type.
- The fifth case will be true if `v` is of `specialString` type.

Now, the type will be printed, accordingly, when one case is found true.

What if no case is found true? In this case, we have a *default* case that prints the message of an *unknown* type. At **line 25**, we pass `whatIsThis` to `testFunc`, which means that the empty interface is of type `specialString` here.

Now, look at `main`. At **line 29**, we are calling the `TypeSwitch` function, which will call the `testFunc` for `whatIsThis`. **Line 20** will be executed, and **any hello is a special String** will be printed on the screen.

Constructing an array of a general type or with variables of different types

In Chapter 5, we saw how arrays of ints, floats, and strings can be searched and sorted. But what about arrays of other types? Do we have to program that for ourselves? We now know that this is possible by using the empty interface. Let us give it the alias `Element`: `type Element interface {}` Then, define a container struct Vector, which contains a slice of `Element` items:

```
type Vector struct {
```

```
type Vector struct {
  a []Element
}
```

Vectors can contain anything because any type implements the empty interface; every element could be of different types. We can define a method `At()` that returns the *i-th* element:

```
func (p *Vector) At(i int) Element {
  return p.a[i]
}
```

And a function `Set()` that sets the *i-th* element:

```
func (p *Vector) Set(i int, Element e) {
  p.a[i] = e
}
```

Everything in the vector is stored as an `Element` to get the original type back (unboxing); we need to use type-assertions. The compiler rejects assertions guaranteed to fail, but be aware that type assertions always execute at run time and so can produce run-time errors!

Copying a data-slice in a slice of interface{}

Suppose you have a slice of data of `myType` and you want to put them in a slice of empty interface, like in:

```
var dataSlice []myType = FuncReturnSlice()
var interfaceSlice []interface{} = dataSlice
```

This doesn't work; the compiler gives you the error: `cannot use dataSlice (type []myType) as type []interface{} in assignment`. The reason is that the memory layout of both variables is not the same (try to reason this yourself or see this link). The copy must be done explicitly with a for-range statement, like in:

```
var dataSlice []myType = FuncReturnSlice()
var interfaceSlice []interface{} = make([]interface{}, len(dataSlice))
for ix, d := range dataSlice {
  interfaceSlice[ix] = d
}
```

Node structures of general or different types

In Chapter 8, we encountered data-structures like lists and trees; using a recursive struct type called a *node*. The nodes contained a data field of a certain type. Now, with the empty interface at our disposal, data can be of that type, and we can write generic code. Here is some starting code for a binary tree structure:

```
type Node struct {
  le *Node
  data interface{}
  ri *Node
}

func NewNode(left, right *Node) *Node {
  return &Node{left, nil, right}
}

func (n *Node) SetData(data interface{}) {
  n.data = data
}
```

In the method, the `SetData` data type is kept an empty interface. It means the type of `data` can be decided on runtime. It can be *int*, *string*, *float32*, or even any *user-defined* type, as seen in the above examples.

Interface to interface

An interface value can also be assigned to another interface value, as long as the underlying value implements the necessary methods. This conversion is checked at runtime. When it fails, a runtime error occurs: this is one of the dynamic aspects of Go, comparable to dynamic languages like Ruby and Python. Suppose:

```
var ai AbsInterface // declares method Abs()
type SqrInterface interface { Sqr() float }
var si SqrInterface
pp := new(Point) // say *Point implements Abs, Sqr
var empty interface{}
```

Then, the following statements and type assertions are valid:

```
empty = pp; // everything satisfies empty
ai = empty.(AbsInterface); // underlying value pp implements Abs()
// (runtime failure otherwise)

si = ai.(SqrInterface); // *Point has Sqr() even though AbsInterface doesn
't
empty = si; // *Point implements empty set
// Note: statically checkable so type assertion not necessary.
```

Here is an example with a function call:

```
type myPrintInterface interface {
  print()
}
func f3(x myInterface) {
  x.(myPrintInterface).print() // type assertion to myPrintInterface
}
```

The conversion to myPrintInterface is entirely dynamic: it will work as long as the underlying type of `x` (the *dynamic type*) defines a print method.

Type `map[string]` interface{}

By using the empty interface as type, we could store any value, but when using that value, we would first have to do a type assertion.

This type is commonly used to store anything that doesn't fit into a typed-value map. It's practical for building an application rapidly, since you only have to unbox the elements as you need them, and you can mix and match types for its values. This is great for semantically related elements, like configuration structures. It is used in untyped JSON unmarshalling, and the gob and template packages use them internally to store things: types by name, and functions by name.

Now that you are familiar with the uses and advantages of empty interfaces, the next lesson brings you a challenge to solve.