Module Ten

Go Interfaces

The belief that complex systems require armies of designers and programmers is wrong. A system that is not understood in its entirety, or at least to a significant degree of detail by a single individual, should probably not be built.
Niklaus Wirth

The key to performance is elegance, not battalions of special cases.
Jon Bentley and Doug McIlroy

Introduction to Programming in Go

10.1 Interfaces in Go

In the previous two modules, we have looked at types and methods, which allow us to do very typically object oriented kinds of things. In some ways though, the notion of a type is analogous to a concrete class in Java or C++ but without any sort of abstraction mechanism. Composition is supported, as we have seen, but there is nothing that is like a Java style inheritance mechanism in Go.

Instead, Go has opted for a different kind of abstraction model, similar to the notion of a Java Interface, where a Go interface is used to define an abstract type. The idea of an interface in Go defining a type is very much in keeping with the idea of an interface in general in OO programming – it specifies a contract about what an object or type can do, but says nothing about how that contract is implemented or the internal workings of the object that implements the contract.

An interface in Go does not tell you anything more than what an implementing object can do – it defines what might be called a behavioral type. For example, if I say I am a swimmer, then that tells you something about what I do, but nothing about how I do it or anything else about me. We have an good intuitive notion of what a “swimmer” interface is – things that implement a swimmer interface can swim. But how we implement a swimmer interface for a dog is different than how a salmon or squid implements it – the only common thing all those types share is that because they have all implemented the swimmer interface, they can all swim.

Notice that in this abstraction model there is no abstract swimmer type that salmon, squids and Go programmers inherit from, but we are all of type swimmer only because we implement the swimmer interface.

This is exactly how interfaces work in Go. A Go interface defines a set of methods and any type that has those methods defined in its method set is considered to have implemented the interface. There is no need to have any formal declaration that a type implements an interface – if it has the methods defined in the interface, then it implements the interface. After all, you didn’t have to formally declare that you were implementing a swimmer interface, you just do it by being able to swim

10.1.1 Interface Types

To see how interfaces work, we look at a simple example based on the above. In this case we will define a Swimmer interface that defines one method called swim(). Generally, interfaces in Go tend to be quite small, often just one or two methods.

We also define two types, fish and human each of which has a swim method. Without any further coding, both of these types implement the Swimmer interface. However the human type also implements a function not in the Swimmer interface, but this does not affect the fact that human implements the Swimmer interface. There is nothing to prevent a type from having more methods in its method set than are in the interface which means that a type can implement multiple interfaces.

```go
// Example 10-01  Swimmer Interface

package main

import "fmt"

type Swimmer interface {
  swim()
}

type fish struct{ species string }
type human struct{ name string }

func (f *fish) swim() {
  fmt.Println("Underwater")
}

func (f *human) swim() {
  fmt.Println("Dog Paddle")
}

func (f *human) walk() {
  fmt.Println("Strolling around")
}
func main() {
  tuna := new(fish)
  tuna.swim()
  knuth := new(human)
  knuth.swim()
  knuth.walk()
}
```

```
[Module10]$ go run ex10-01.go
Underwater
Dog Paddle
Strolling around
```

10.2 Interface Variables

The example on the previous page seems trivial given that we are working with a couple of concrete types. However, the power of interfaces comes when we use variables whose type is an interface, just like we do in other languages.

In example 10.2, we create a variable "s" of type Swimmer and in turn, we assign it a fish and a human. When we call swim() on s, the interface variable remembers which implementation of swim() is should call based on the type of implementation it references.

The interface variable contains a set of references, one for each method defined in the interface. When an object is assigned to the variable, those references are all updated to point to the correct methods for that concrete type.

Now what we are getting in the example below is a variable of type Swimmer that can hold a reference to any object that implements the swimmer interface.

In the following examples, the code from ex10-01 that defines the concrete types is in the file swimmer.go so we can focus on the code of interest to us.

```go
// Example 10-02  Interface Variable

package main

import "fmt"

type Swimmer interface {
  swim()
}

func main() {
  var s Swimmer
  s = new(fish)
  s.swim()
  s= new(human)
  s.swim()
}
```

```
[Module10]$ ./ex10-02
Underwater
Dog Paddle
```

10.2.1 Interface types as parameters.

In example 10-03, we see the use of an interface variable as a parameter which allows us to write much more powerful and generic functions than we would be able to do if we were limited to concrete types.

In this case the submerge() function takes any object of type Swimmer and calls the appropriate swim() method on that object.

```go
// Example 10-03  Interface variables as parameters
package main

import "fmt"

type Swimmer interface {
  swim()
}

func submerge(s Swimmer) {
  s.swim()
}

func main() {
    submerge(new(fish))
    submerge(new(human))
}
```

```
[Module10]$ ./ex10-03
Underwater
Dog Paddle
```

10.2.2 Limitation of an interface variable

in example 10.04 we see that if we assign a concrete object to an interface variable, the variable can only “see” the methods in its interface. If we assign a human object to a Swimmer variable s, then for each method in the interface, "s" holds a reference to the concrete implementation of swim() in the human method set.

But if we try to call the walk() method, which is implemented for a human type but is not part of the Swimmer interface, then the method call fails because “s” does not know about the walk() method.

```go
// Example 10-04  Interface variable limitiation

package main

import "fmt"

type Swimmer interface {
  swim()
}

func main() {
  var s Swimmer
  s= new(human)
  s.swim()
  s.walk()
}
```

```
[Module10]$ ./ex10-04
./ex10-04.go:8: s.walk undefined (type Swimmer has no field or
method walk)
```

10.3 Combining Interfaces

Go interfaces are typically small, usually one to three methods, but we can combine them in the same way that we can combine structs. In a sense this is exactly what we are doing because an interface is essentially a struct in terms of how it is implemented "under the hood".

In example 10.5 we have added a new interface called Walker and then combined the Walker and Swimmer interfaces into a new interface call Amphib

```go
// Example 10-05 Amphib Type

package main

import "fmt"

type Swimmer interface {
  swim()
}

type Walker interface {
  walk()
}

type Amphib interface {
  Walker
  Swimmer
}

func main() {
  var a Amphib
  a = new(human)
  a.walk()
  a.swim()
}
```

```
[Module10]$ ./ex10-05
Strolling around
Dog Paddle
```

Combining the interfaces can be done in a variety of ways, all that matters is the aggregate of methods in the final result. All of the following would be equivalent ways to define the Amphib interface.

```go
type Amphib interface {
  Walker
  Swimmer
}

type Amphib interface {
  Walker
  swim()
}

type Amphib interface {
  walk()
  Swimmer
}

type Amphib interface {
  walk()
  swim()
}
```

11.4 Type Testing

At any given time an interface variable can potentially hold any one of a number of types. At points in our program logic, it may be important for us to know exactly what sort of concrete type our interface variable is referencing

We can do this by asking the variable if it contains a particular type using the syntax

```
n, ok := v.(T)
```

If the object referenced by v is of type T, then ok is set to true and n is created as a reference to the object but is a variable of type T. If v is not of type T, then ok is false and v is set to the default zero value for T.

This sounds a little confusing until we walk through a concrete example in 10-06

In example we have created a slice of two swimmers, the first is a human and the second is a fish. In the for loop, we ask if each entry in turn is actually a human. The first time through, ok is true and the human object “Bobby” is assigned to h. Now that we have a reference to the human type, we can call the walk() method which which would not have worked otherwise – specifically swimmer.walk() would have produced an error just like we saw before because the swimmer variable does not know about anything outside the Swimmer interface.

The second time through the loop, testing the fish produces an ok value of false, so the body of the conditional is skipped.

```go
// Example 10-06 Type Testing

package main

type Swimmer interface {
  swim()
}


func main() {
  var x = []Swimmer{&human{"bobby"}, &fish{}}

  for _, swimmer := range x {
      if h, ok := swimmer.(*human); ok {
          h.walk()
      }
  }
}
```

```
[Module10]$ ./ex10-06
Strolling around
```

11.4.1 The Type Switch Statement

A more convenient way to deal with processing multiple possible types is to use the switch statement we saw back in module three. In this case, our test value is the type the object and each case corresponds to each different type our object could be. Again a simple example clarifies how this is done.

In example 10-7, we have added a squid and cat types into the swimmer.go file both of which have swim methods as shown below. This means that all of these new structs are of type Swimmer.

```
   // Exmaple 10-07 Swimmer.go

   package main

   import "fmt"

   type fish struct{ species string }
   type human struct{ name string }
   type squid struct{ inkyness int }
   type cat struct{ breed string }

   func (f *fish) swim() {
     fmt.Println("Underwater")
   }
   func (f *squid) swim() {
     fmt.Println("Jetting along")
   }
   func (f *cat) swim() {
     fmt.Println("If I must...")
   }
   func (f *human) swim() {
     fmt.Println("Dog Paddle")
   }
   func (f *human) walk() {
     fmt.Println("Strolling around")

}
```

We can use the switch statement to now select type specific processing as we iterate through a list of Swimmer objects. Back in module three, I mentioned that the generalization of the Go switch statement from that normally seen in C-style languages is quite useful and this is an example of exactly that usefulness.

```go
// Example 10-07 Switching on types

package main

import "fmt"

type Swimmer interface {
  swim()
}

var x = []Swimmer{&human{"bobby"}, &fish{}, &cat{}, &squid{}}

func main() {
  for index, swimmer := range x {
       switch t := swimmer.(type) {
       case *human:
            fmt.Printf("Item %d is a human and is ", index)
            t.walk()
       case *squid:
            fmt.Println("Item ", index, "is a squid")
       case *fish:
            fmt.Println("Item ", index, "is a fish")
       default:
            fmt.Printf("Item %d is of type %T\n", index, t)
       }
  }

}
```

```
[Module10]$ ./ex10-07
Item 0 is a human and is Strolling around
Item  1 is a fish
Item 2 is of type *main.cat
Item  3 is a squid
```

10.5 The empty Interface

One of the enduring constructs in various OO languages is the idea of a common universal type at the root of all inheritance hierarchies, whether it is the “vanilla” flavor in ZetaLisp or the "Object" class in Java, because having such a construct simplifies a lot of language design issues.

While Go does not have inheritance hierarchies, it does have a useful construct that emulates this kind of construct called the empty interface. The empty interface has no methods and thus by default, every type implements the empty interface, whether it is a built-in type or a user defined type.

From a practical perspective, what this allows us to do is to create collections of arbitrary types of objects. In the following example, we define an interface called “whatever” which allows us to create a slice made up of different things.

From an implementation point of view, what we are doing is creating an underlying array of “whatever” interface objects. Each whatever object has two kinds of information embedded in it, a record of the actual type it refers to and a reference to that object. Remember earlier that the interface knows dynamically how to call the methods of the interface by essentially maintaining pointers to the methods of the concrete type, or at least that is a good way to think about it conceptually. Essentially the same sort of mechanism is in play here.

There is not a predefined empty interface, we can create as many empty interfaces as we want, all named different things. Functionally, they are all the same but we may want more than one name for clarity in our code.

```go
// Example 10-08 The empty Interface

package main

import "fmt"

type Swimmer interface {
  swim()
}
type myint int32
type point struct{ x, y int }

var i myint = 0

type whatever interface{}

// continued on next page...
// Example 10-08 The empty Interface... continued

func main() {

   mylist := []whatever{i, float64(45), &point{2, 3}, true,
          &fish{"tuna"}}
   for index, object := range mylist {
       switch v := object.(type) {
       case bool:
            fmt.Printf("item %d is a bool = %v\n", index, v)
       case myint:
            fmt.Printf("item %d is a myint = %v\n", index, v)
       case float64:
            fmt.Printf("item %d is a float64 = %v\n",
                                                  index, v)
       case *point:
            fmt.Printf("item %d is a point = %v\n"
                                                 index, *v)
       default:
            fmt.Printf("item %d is a %T = %v\n",
                                                index, v, v)
       }
  }

}
```

```
[Module10]$ ./ex10-08
item 0 is a myint = 0
item 1 is a float64 = 45
item 2 is a point = {2 3}
item 3 is a bool = true
item 4 is a *main.fish = &{tuna}
```