Module Nine

Methods

Go is like a very delicious trifle - the further into it you go, the more delicious things you find. The quality is clear throughout. Go is very unassuming. You start by wondering what the big deal is + getting annoyed with the minor differences from other C-syntax languages before slowly progressing towards quite liking it, then eventually once you grok how simple + elegant and well engineered it is you come to love it

Lorenzo Stoakes

The way errors are handled in Go as opposed to the old exception model is huge. I'm talking the next evolution of programming languages huge. Knowing where nearly all your points of failure are and deciding what to do with them before you role a product out to a client is priceless.

Michael Schneider

Introduction to Programming in Go

9. Methods in Go

A method is a Go function that is defined in a way so that it operates on objects of a particular type. The type that the method works on is called the receiver of the method, and the set of all methods that work on a type T is called the method set of T. We can think of a method for a type as a function that sends a message, in a generic sense, to an object of that type.

In a language like Java or C++, a class definition is conceptually a type, and the class definitions contains the methods that are associated with that class. That is how we know they are methods for that class, they appear inside the class definition. The methods of a class make up the set of messages that we can send to objects of that type.

Where Go differs from these other languages is how this type and method relationship is specified, although it also is a bit more general in its application than in Java or C++. To associate a function with a type in Go, the function just needs to declare that a particular type is a receiver of the method. The effective result is the same – a type and its associated method set are functionally equivalent to a class with its instance variables and methods.

The reason why Go does it this way is the same reason that Go does everything else – it is leaner and faster and makes for a smoother compile and build process. Go throws away a lot of the features used in other OO languages because they make the language and build process overly complex, and the capabilities that they add for the programmer are, in the judgment of tho language designers, minimal.

9.1 Method definition

While a Go method is conceptually a lot like a method in other languages, the concept is more general since the receiver type of a method can be almost anything, not just a struct. There are only a few exceptions to what can have methods: interfaces being the one Go construct that cannot that we will encounter in this course.

The general syntax of a method is

```
func (receiver type) func_name(parameters)( return_values) {}
```

The receiver variable can be used exactly like a parameter in the body of the method.

9.1.1 A non-struct example

In this example, we are going to define a method on the type int32. However, one of the rules of Go methods is that we can only define methods on types in the same package, which means that the int32 type is out of scope.

However, we can define an alias for int32 – call it myint – in the current package that we can define a method on. This is one of the tricky uses of aliases, to get around a scoping rule.

```go
// Example 09-01 Method for int32

package main

import "fmt"

type myint int32

func (x myint) negate() {
  x = - x
}

func main() {
  z := myint(89)
  fmt.Println(z)
  z.negate()
  fmt.Println(z)
}
```

```
[Module09]$ go run ex09-01.go
89
89
```

The receiver variable x works exactly like a parameter. In this case, the argument z has been passed by value so x in the method is a local copy.

The example 09-01, the method could not change the receiver value.

9.1.2 Pointers as receivers

If we want a method to alter something about the receiver, which we often do with structs, then we define the method on a pointer to the receiver.

It would be awkward to have to use addresses like this in the calling code:

```
(&z).negate()
```

When Go sees a method called on a type T and there is a method with a receiver of type *T or pointer to T, then Go handles the messiness of taking the address for you.

If negate() is defined as:

```
func(x *myint) negate() {}
```

then Go does the following conversion automatically

```
z.negate()  —> (&z).negate
```

```go
// Example 09-02 Method for *int32

package main

import "fmt"

type myint int32

func (x *myint) negate() {
  *x = -*x
}

func main() {

  z := myint(89)
  fmt.Println("Before call", z)
  z.negate()
  fmt.Println("After call", z)
}
```

```
[Module09]$ go run ex08-01.go
Before call 89
After call -89
```

The method in example 09-01 was not able to change the value of what was passed into the receiver variable beause "x" was a copy of the receiver variable "z".

In example 09-02 we have changed the receiver to a pointer so that we can mutate the receiver of the method. We have also been careful to negate what the receiver points by de-referencing the points (ie. using *x).

Because of this automatic conversion that takes place, we cannot define two functions with the same name, one of which operates on a type and the other which operates on a pointer to the type. This produces an ambiguity that Go cannot resolve, the price of having Go do that bit of re-writing mentioned on the previous page.

9.2 Combining Method Sets

In Go functions cannot be overloaded as they are in languages like C++ or Java. However methods can be overloaded in a way. Two methods can have identical signatures as long as they operate on different receiver types. Remember that we are treating a variable type T and a pointer *T as the same for the purpose of method sets.

This makes sense since functions are more or less defined at the package level while methods are grouped into method sets based on their receivers. This means that we are refining the rule about function names being unique a bit by saying that a function name has to be unique within a method set.

9.2.1 Method overloading

The use of method sets partitions all our function and method definitions into disjoint sets.

Suppose we have the following three methods and functions

1. `func (x *myint) negate() {...}`
2. `func (p * point) negate() {...}`
3. `func negate(x float32) float32 {..}`

There will never be any ambiguity about which one of these functions will be called. The first two are distinguished based on the type of the variable that is receiving the message. The third can only be called when there is not a receiving variable.

The rule in Go is actually that the name of a function has to be unique within a method set.

```go
// Example 09-03 Method overloading

package main

import "fmt"

type myint int32
type myfloat float64

func (x *myint) negate() {
  *x = -(*x)
}
func (x *myfloat) negate() {
  *x = 0.0
}

func main() {
  i := myint(89)
  f := myfloat(189.9)
  i.negate()
  f.negate()
  fmt.Println(i, f)
}
```

```
[Module09]$ go run ex09-03.go
-89 0
```

In example 09-03 two functions called negate are defined that are in different methods sets. We are cheating and having the myfloat version just set the receiver variable value to 0 so that we can see which method is called. Notice that the appropriate method is called based on the receiver type.

9.3 Methods on inner structs

Suppose we have a struct which we will call it the outer struct. One of the fields of the outer struct is also a struct which we will call the inner struct. Consider the circle and point for example.

```
type point struct{ x, y int }
type circle struct {
  center point
  radius float32
}
```

If both point and circle have methods defined on them then both method sets apply to circle. Getting this sorted out is a bit confusing the first time you encounter it so an analogy and an example both help.

This is how Go emulates inheritance so we can use that as an analogy. The inner struct is like a super class or parent class in an inhertance hierarchy. The inner point struct has a method set that is now acessible through the circle struct, which is analogous to the subclass or child class in the inheritance heirarchy. Because a circle contains a point, the circle gets to use all of the methods in the method set for point.

Lets start with the first file in our example which contains the two structs and their method sets.

```go
// Example 09-04 Supporting code

package main

import "math"

type point struct{ x, y int }

func (p *point) swap() {
  p.x, p.y = p.y, p.x
}

type circle struct {
  center point
  radius float32
}

func (c *circle) area() float32 {
  return c.radius * c.radius * math.Pi
}
```

Because the point is a named field (center), we have to use the selector that we see in the example to get swap() to be sent to the right receiver. If we just try to send the message to the circle, the circle will reject it because it doesn't recognize it. Of course we can send the circle an area() method because it is in the method set of circle.

This doesn't seem a lot like inheritance until we make the point an anonymous field and then we see a change in behaviour that looks a lot more like inheritance.

```go
// Example 09-04 Inner struct methods

package main

import "fmt"

func main() {

  c := circle{point{1, 0}, 3.0}
  fmt.Println(c)
  c.center.swap()
  fmt.Println(c, c.area())
}
```

```
[ex04]$ ./ex09-04
{{1 0} 3}
{{0 1} 3} 28.274334
```

If we change the struct definition of circle to example below, then we can rewrite the main function code as well,

```
// Example 09-05 modifications

type circle struct{
  point
  radius float32
}

...

func main() {
  c := circle{point{1, 0}, 3.0}
  fmt.Println(c)
  c.swap()
  fmt.Println(c, c.area())
}
```

```
[ex05]$ ./ex09-05
{{1 0} 3}
{{0 1} 3} 28.274334
```

Considering how accessing fields worked in the last module, this behavior is quite consistent with what we saw earlier.

But what if there is a swap function defined for the circle itself? We can speculate what happens by analogy to a situation like Java or C++. If the derived class has a method with the same name as the parent class the derived or child class version overrides the parent class version, which is exactly what we see happening here in example 6

```go
// Example 09-06  Modifications to circle

type circle struct{
  point
  radius float32
}

func (c *circle) swap() {
}

func (c *circle) area() float32 {
  return c.radius * c.radius * math.Pi
}
```

In the above code, a swap function has been added to circle's method set, but it doesn't actually do anything. If we call swap() on a circle, there are three possibilities:

1. The swap() method for point will be called and we will see the x and y co-ordinates of the point swapped.
2. The swap() method for circle will be called and the x and y co-ordinates will remain unchanged.
3. We will get an error.

```go
// Example 09-06 Method overloading

package main

import "fmt"

func main() {

  c := circle{point{1, 0}, 3.0}
  fmt.Println(c)
  c.swap()
  fmt.Println(c, c.area())
}
```

```
[ex06]$ ./ex09-06
{{1 0} 3}
{{1 0} 3} 28.274334
```

9.4 Combining objects

These two ways of nesting structs correspond to two different techniques commonly used in OO programming for combining objects.

1. *Aggregation.* Two objects can be combined by aggregation when one object delegates incoming messages to another object that it holds a reference to. In this case the circle object holds a copy to the point object. When we have a named field (center point), then the swap() method is delegated to the point object.

   The downside to this way of combining functionality is that we have to explicitly delegate using the selector notation c.center.swap(). On the upside, we can have as many point objects as we want associated with the circle because the selector notation ensures the right point object is being swapped.

2. *Inheritance.* Two objects are combined so that the parent objects' methods become available to the child. This is what happened when we embedded point anonymously in circle, the inner struct's (point) method set became part of the outer struct's (circle) method set. This Go form emulates inheritance quite well.

9.4.1 Multiple inheritance

It may seem that the restriction in Go on having only one anonymous field would preclude an emulation of multiple inheritance but in fact it doesn't. In multiple inheritance, a child class inherits from two different parent classes, which would be the same as having two anonymous fields of different types in a struct in Go.

In languages like C++ which allow multiple inheritance, there are usually mechanisms like virtual inheritance to prevent two copies of the same parent class form being present in the inheritance hierarchy. Go enforces the analogous restriction by allowing only one anonymous field of a given type in a struct.