Methods on Embedded Types

This lesson covers in detail how to embed a functionality in a type and how methods work for embedded structs.

WE'LL COVER THE FOLLOWING

When an anonymous type is embedded in a struct, the *visible* methods of that type are embedded as well. In effect, the outer type *inherits* the methods: methods associated with the anonymous field are promoted to become methods of the enclosing type. *To subtype something, you put the parent type within the subtype*. This mechanism offers a simple way to emulate some of the effects of subclassing and inheritance found in classic OO-languages; it is also very analogous to the *mixins* of Ruby.

Here is an illustrative example. Suppose we have an interface type `Engine`, and a struct type `Car` that contains an anonymous field of type `Engine`.

```
type Engine interface {
  Start()
  Stop()
}
type Car struct {
  Engine
}
```

We could then construct the following code:

```
func (c *Car) GoToWorkIn {
  // get in car
  c.Start();
  // drive to work
  c.Stop();
  // get out of car
}
```

The following complete example shows, how a method on an embedded struct can be called directly on a value of the embedding type.

```
package main
import (
"fmt"
"math"
)

type Point struct {
  x, y float64
}

func (p *Point) Abs() float64 {
  return math.Sqrt(p.x*p.x + p.y*p.y)
}

type NamedPoint struct {
  Point   // anonymous field of Point type
  name string
}

func main() {
  n := &NamedPoint{Point{3, 4}, "Pythagoras"} // making pointer type variable
  fmt.Println(n.Abs()) // prints 5
}
```

Method on Embedded Type

In the above code, at **line** 7, we make a struct of type `Point` with two fields `x` and `y` of type `float64`. We make another struct at **line 15**, of type `NamedPoint` with two fields in it. The first is an anonymous field of type `Point` and the second is `name`, a variable of type *string*. Look at the header of the method `Abs()` at **line 11**: `func (p *Point) Abs() float64`. It shows that this method can only be called by the pointer to the variable of type `Point`, and it returns a value of type *float64*. Following is the formula for calculating the *absolute* value of a point:

$$value = \sqrt{x^2 + y^2}$$

Line 12 of the code implements the above formula. We get the values of `x` and `y` through the selector operator applied on Point `p`. To calculate the square root, we use the package of `math` which is imported at **line 4**. The method calculates the absolute value for a point and returns it. Now, look at `main`. At **line 21**, we make a *pointer* variable `n` of type `NamedPoint` using the struct-literal: `&NamedPoint{Point{3, 4}, "Pythagoras"}`. The part `Point{3,4}` is used to assign value to the anonymous `Point` variable(`x` gets **3** and `y` gets **4**), and **Pythagoras** is assigned to the `name` string. The symbol `&` makes it a pointer variable. At **line 22**, we call the method `Abs` on variable `n` and print the return value, which is **5** ($\sqrt{3^2+4^2} = \sqrt{25}$).

Embedding injects fields and methods of an existing type into another type. Of course, a type can have methods that act only on variables of that type, not on variables of the embedded *parent* type.

Embed functionality in a type

There are two ways of embedding functionality in a type:

- **Aggregation** (or composition): including a named field of the type of the wanted functionality
- **Embedding**: embedding the type of the wanted functionality anonymously

Suppose we have a type `Customer` and we want to include a `Logging` functionality with a type `Log`, which contains an accumulated message (of course, this could be elaborated). If you want to equip all your domain types with a logging capability, you implement such a `Log` and add it as a field to your type. Also, add a method `Log()`, returning a reference to this log.

The first method could be implemented as follows:

```
package main
import (
"fmt"
)

type Log struct {
    msg string
}

type Customer struct {
    Name string
    log *Log
}

func main() {
    c := new(Customer)
    c.Name = "Barack Obama"
    c.log = new(Log)
    c.log.msg = "1 - Yes we can!"
    // shorter:
    c = &Customer{"Barack Obama", &Log{"1 - Yes we can!"}}
    fmt.Println(c.log)
    c.Log().Add("2 - After me, the world will be a better place!")
    fmt.Println(c.Log())
}

func (l *Log) Add(s string) {
    l.msg += "\n" + s
}

func (l *Log) String() string {
    return l.msg
}

func (c *Customer) Log() *Log {
    return c.log
}
```

Aggregation

In the above code, at **line 6**, we make a struct of type `Log` with one field of string named `msg`. At **line 10**, we make a struct of type **Customer** with two fields. The first is a string variable `Name`, and the second is a *pointer* variable `log`, of type `Log`. Before going to `main`, let's see the other three functions of the program:

- See the header of `Add()` method at **line 27**: `func(l *Log) Add(s string)`. This shows that this method can be called by the pointer variable of type `Log`. It takes a string as a parameter. Look at **line 28**. This method appends a new blank line and then adds a string `s` to the internal field `msg` of `l`.

- See the header of `String()` method at **line 31**: `func(l *Log) String()string`. This shows that this method can be called by the pointer variable of type `Log`. This method returns the `msg` of `l`.

- See the header of `String()` method at **line 35**: `func (c *Customer) Log() *Log `. This shows that this method can be called by the pointer variable of type `Customer`. This method returns the `log` of `c`.

Now, look at `main`. At **line 16**, we make a `Customer` type variable `c` using the `new()` function. In the next few lines, we are giving the fields their values. At

line 17, we are giving `Name` (a field of Customer `c`) a value **Barack Obama**. In

the next line, we make a new `Log` type variable and assign it to `c.log`. Every variable of type `Log` has a `msg`. So, at **line 19**, we give the value to `msg` of `c.log` as: `c.log.msg = "1 - Yes we can!"`. It took us *four* lines to make a proper `Customer` object.

How about doing it in a single line? Look at **line 21**. We make a *pointer* variable of type `Customer` using struct-literal and reinitialize `c` as: `&Customer{"Barack Obama", &Log{"1 - Yes we can!"}}`. The part `&Log{"1 - Yes we can!"}` is used to assign a value to an anonymous `Log` pointer variable(`msg` gets **1 - Yes we can!**), and **Barack Obama** is assigned to the `Name` string. The symbol `&` makes it a pointer variable.

See **line 22**. We are printing the `c.log`. This will call the `String()` function, and we'll print the return value. In this case, it's **1 - Yes we can!**. In the next line, we are adding a string to the `msg` of the `log` of `c` as: `c.Log().Add("2 - After me, the world will be a better place!")`. The part `c.Log()` calls the method `Log()`, and the pointer to `log` of `c` is returned, which then calls the `Add()` method along with the string. The string gets appended at the end of `msg` of `c.Log()` (`c.log`). In the next line, we are printing `c.Log()` to see whether the `msg` was changed or not. The `c.Log()` calls the method `Log()`, and the pointer to `log` of `c` is returned, which will call the `String()` function. Then, we'll print the return value. In this case, it's:

1 - Yes we can!

2 - After me, the world will be a better place!

The second method, on the contrary, would be like:

```
package main
import (
"fmt"
)

type Log struct {
  msg string
}

type Customer struct {
  Name string
  Log
}
```

```
func main() {
  c := &Customer{"Barack Obama", Log{"1 - Yes we can!"}}

  c.Add("2 - After me, the world will be a better place!")
  fmt.Println(c)
  }

func (l *Log) Add(s string) {
  l.msg += "\n" + s
}

func (l *Log) String() string {
  return l.msg
}

func (c *Customer) String() string {
  return c.Name + "\nLog:\n" + fmt.Sprintln(c.Log)
}
```

Embedding

In the above code, at **line 6**, we make a struct of type `Log` with one field of string named `msg`. At **line 10**, we make a struct of type **Customer** with two fields. The first is a string variable `Name`, and the second is an anonymous variable of type `Log`. Before going to `main`, let's see the other three functions of the program:

- See the header of `Add()` method at **line 21**: `func(l *Log) Add(s string)`. This shows that this method can be called by the pointer variable of type `Log`. It takes a string as a parameter. Look at **line 22**. This method appends a new blank line and then adds a string `s` to the internal field `msg` of `l`.

- See the header of `String()` method at **line 25**: `func(l *Log) String()string`. This shows that this method can be called by the pointer variable of type `Log`. This method returns the `msg` of `l`.

- See the header of `String()` method at **line 29**: `func (c *Customer) String() string `. This shows that this method can be called by the pointer variable of type `Customer`. This method appends the `Name` of customer `c` and its `log` (anonymous) and returns this final string.

Now, look at `main`. At **line 16**, we make a *pointer* variable of type `Customer` using struct-literal and reinitialize `c` as: `&Customer{"Barack Obama", Log{"1 -`

`Yes we can!"}}`. The part `Log{"1 - Yes we can!"}` is used to assign value to the anonymous `Log` variable(`msg` gets **1 - Yes we can!**), and **Barack Obama** is assigned to the `Name` string. The symbol `&` makes it a pointer variable. In the next line, we are adding a string to the `msg` of the `log` of `c` as: `c.Add("2 - After me, the world will be a better place!")`. The log of `c` will automatically call the `Add()` method. The string gets appended at the end of `msg`. In the next line, we are printing `c` to see whether the `msg` was changed or not. The `c` calls the method `String()`, and we'll print the returned value. In this case, it's:

Barack Obama

Log: {1 - Yes we can!

2 - After me, the world will be a better place!}

Now that you're familiar with embedding, in the next lesson, you'll cover multiple-inheritance in Go.