Anonymous Fields and Embedded Structs

In the first part of this lesson, you'll study how to make structs with nameless fields and how to use them. In the second part, you'll see how to embed a struct inside another struct and use it.

Definition

Sometimes, it can be useful to have structs that contain one or more *anonymous* (or *embedded*) fields that are *fields with no explicit name*. Only the type of such a field is mandatory, and the type is then also the field's name. Such an anonymous field can also be itself a struct, which means structs can contain embedded structs. This compares vaguely to the concept of inheritance in OO-languages, and as we will see, it can be used to simulate a behavior very much like inheritance. This is obtained by embedding or composition. Therefore, we can say that in Go *composition is favored over inheritance*.

Consider the following program:

```
package main
import "fmt"

type innerS struct {
  in1 int
  in2 int
}

type outerS struct {
```

```
  b int
  c float32


  int // anonymous field
  innerS // anonymous field
}

func main() {
  outer := new(outerS)
  outer.b = 6
  outer.c = 7.5
  outer.int = 60
  outer.in1 = 5
  outer.in2 = 10
  fmt.Printf("outer.b is: %d\n", outer.b)
  fmt.Printf("outer.c is: %f\n", outer.c)
  fmt.Printf("outer.int is: %d\n", outer.int)
  fmt.Printf("outer.in1 is: %d\n", outer.in1)
  fmt.Printf("outer.in2 is: %d\n", outer.in2)
  // with a struct-literal:
  outer2 := outerS{6, 7.5, 60, innerS{5, 10}}
  fmt.Println("outer2 is: ", outer2)
}
```

Struct with Anonymous Fields

In the above code, at **line 4**, we make a struct of type `innerS` containing two integer fields `in1` and `in2`. At **line 9**, we make another struct of type `outerS` with two fields `b` (an integer) and `c` (a float32 number). That's not all. We also have two more fields that are *anonymous* fields. One is of type *int* (see **line 13**), and the other is of type `innerS` (see **line 14**). They are anonymous fields because they have no explicit names.

Now, look at `main`. We make an `outerS` variable `outer` via the `new` function at **line 18**. In the next few lines, we are giving the values to its fields. The fields `b` and `c` of `outer` are given values at **line 19** and **line 20**, respectively. Now, it's the turn for anonymous fields of `outer`.

To store data in an anonymous field or get access to the data, we use the name of the data type, e.g. `outer.int`. A consequence is that we can **only have one** anonymous field of each data type in a struct. Look at **line 21**. We are assigning the value of **60** to the *int* type anonymous field declared at **line 13**. But how to assign the value to the second anonymous field `innerS` declared at **line 14**? The answer is simple. We also have a struct of type `innerS` in our program, which contains two *int* fields `in1` and `in2`. So, to assign the value to the second anonymous field `innerS`, we'll assign the values to the fields of

`innerS` (see **line 22** and **line 23**). From **line 24** to **line 28**, we are printing the above fields of `outer` to which we assigned the values, to verify the results.

This is one of the methods to make and assign values to a struct type variable that contains anonymous fields, i.e., using the `new` function and then the selector operator. How about making such a struct type variable using *struct-literal*? Look at **line 30**, we are making an `outerS` variable `outer2`, and assigning the same values as the above but in a struct-literal manner as: `outer2 := outerS{6, 7.5, 60, innerS{5, 10}}`. The order is followed. The variable `b` gets **6**, and `c` gets **7.5**. The first anonymous field of `outer2`, which is an *int*, gets **60**. The anonymous field of `innerS` gets the value **5** for `in1` and **10** for `in2`. In the last line, we are printing `outer2` to check the values assigned to its fields.

To store data in an anonymous field or get access to the data, we use the name of the data type, e.g., `outer.int`. A consequence is that we can *only have one* anonymous field of each data type in a struct.

Embedded structs

As a struct is also a data type; it can be used as an anonymous field. See the example above. The outer struct can directly access the fields of the inner struct as in `outer.in1`; this is possible even when the embedded struct comes from another package. The inner struct is inserted or *embedded* into the outer struct. This simple *inheritance* mechanism provides a way to derive some or all of your implementation from another type or types.

Conflicting names

What are the rules when there are two fields with the same name (possibly a type-derived name) in the outer struct and an inner struct?

1. An outer name hides an inner name. This provides a way to override a field or method.
2. If the same name appears twice at the same level, it is an error if the program uses the name. If it's not used, it doesn't matter. There are no rules to resolve the ambiguity; it must be fixed. For example:

```go
type A struct { a int }
type B struct { a, b int }
type C struct { A; B }
var c C
```

According to rule (2), when we use `c.a`, it is an error because there is ambiguity on what is meant: `c.A.a` or `c.B.a`. The compiler error is: `ambiguous DOT reference c.a`. We have to disambiguate with either `c.A.a` or `c.B.a`. Look at another example:

```go
type D struct { B; b float32 }
var d D
```

According to rule (1), using `d.b` is ok. It is the *float32*, not the `b` from `B`. If we want the inner `b`, we can get at it by `d.B.b`.

Anonymous structs

Go also allows you to use structs without a type name, so-called anonymous structs, as in this example:

```go
package main
import "fmt"

func main() {

  var person struct {
    name, surname string
  }

  person.name, person.surname = "Barack", "Obama"
  anotherPerson := struct {   // anonymous struct
    name, surname string
  }{"Barack", "Obama"}
  fmt.Println(person, anotherPerson)
}
```

Anonymous Structs

In the above code, at **line 6**, we make a struct type `person` with two string fields `name` and `surname`. At **line 10**, we set the fields of the `person` struct. The field `name` gets **Barack** and `surname` gets **Obama**. At **line 11**, we make an anonymous struct `anotherPerson`. We do not define any *struct type alias* but anonymous struct `anotherPerson`. We do not define any *struct type alias* but create a struct from the inline struct type and define the initial values in the same syntax. We assign the value to `name` and `surname` as **Barack** and **Obama**. At **line 14**, we are printing `person` and `anotherPerson`. The output shows that they both print the same result.

Comparing structs

Various structs are by default different types, so the values of these structs can't be equal. However, if the fields are identical, structs can be converted into one another. This is illustrated in the following program:

```
package main
import "fmt"

type T1 struct {
  a int
}

type T2 struct {
  b int
}

type T3 struct {
  a int
}

func main() {
  t1 := T1{10}
  t2 := T2{10}
  t3 := T3{10}
  t4 := T3{20}
  fmt.Println(t1, t2, t3 ,t4)
  // fmt.Println("t1 == t2?", t1==t2) // <-- invalid operation: t1 == t2 (mismatched types T1
  // fmt.Println("t1 == t3?", t1==t3) // <-- invalid operation: t1 == t3 (mismatched types T1
  fmt.Println("t1 == t3?", t1==T1(t3)) // true
  fmt.Println("t3 == t4?", t3==t4) // false
}
```

Comparing Structs

In the above code, at **line 4**, we make a struct of type `T1` with one *int* field `a`. Then at **line 8**, we make a struct of type `T2` with one *int* field `b`. Again at **line 12**, we make a struct of type `T3` with one *int* field `a`.

Now, look at `main`. At **line 17**, we make a `T1` type variable `t1` using *struct literal* assigning `a` value of **10**. In the next line, we make a `T2` type *struct_literal* assigning `a` value of **10**. In the next line, we make a `T2` type variable `t2` using *struct_literal* assigning `b` value of **10**. Then, at **line 19**, we

make a `T3` type variable `t3` using *struct_literal*, assigning `a` a value of **10**. Similarly, in the next line, we make a `T3` type variable `t4` using *struct_literal*, assigning `a` a value of **20**. At **line 21**, we are printing `t1`, `t2`, `t3` and `t4` to verify the outputs.

Now, let's perform some comparisons between these structs. See the commented **line 22**. At **line 22**, comparison between `t1` and `t2` was made via the `==` operator. It will give an error because the fields of `t1` and `t2` are not the same. Similarly, see the next commented line, where the comparison between `t1` and `t3` was made. It will give an error not because fields of `t1` and `t3` are not the same. They are the same. However, you have to convert the type first (see **line 24**). The struct `t3` is first type-casted to `T1` and then compared with `t1` as: `t1==T1(t3)`, which will give *true* because both fields (`a`) have a value of **10**. In the last line, we are comparing `t3` with `t4`. This doesn't even type conversion because `t3` and `t4` are of type `T3`. This will give *false* because `a` of `t3` is **10** and `a` of `t4` is **20**.

Now, you are familiar with the use of structs. In the next lesson, you have to write a program to solve a problem.