# Module Eight

*User Defined Types and Structs*

*[the Go authors] designed a language that met the needs of the problems they were facing, rather than fulfilling a feature checklist*

ywgdana in reddit

*I like a lot of the design decisions they made in the [Go] language. Basically, I like all of them*

Martin Odersky, creator of Scala

# Introduction to Programming in Go

# 8. 1 User Defined Types

All programming languages have some mechanism for creating data types. Generally a data type is a bundle of data items that conceptually describes some entity in the real world encapsulated (packaged up) so that we can manipulate that bundle as a single object in our code. For example, COBOL has record types, C has struct types (which is a direct ancestor to Go structs) and object oriented languages have class definitions.

For this module, I am assuming that as an experienced programmer, you have worked with user defined types before in other programming languages so we can focus on the Go specific way that user defined data types are used.

User defined types in Go are created by using the keyword type. There are two kinds of new types that we can define in Go, aliases and structs.

### *8.1.1 Aliases*

When we use the type keyword to create an alias, we are not really defining a new type but rather a new name for an existing type. For example, Go has two aliases defined in its list of basic data types

```
type rune int32
type byte uint8
```

There are a number of reasons why we might want to alias a type, often it is just to be able to use a more natural name for a type, as in the case of runes and bytes. In some cases we may not be able to do certain things with the existing type that we could do with an alias. We will see an example of needing to do exactly that in the next module.

### *8.1.2 Defining structs*

Before we can use a struct we have to define its structure and its type. A struct in Go is a user defined type that has a type name and a set of fields, each of which is (usually) named and has a type. The order the fields appear in is significant in determining the struct type. That means that if we have two structs with the identical fields but the order of the fields is different, then the two structs define two different types.

Structs are defined using the following syntax:

```
type name struct {
  field1 type1
  field2 type2
...
}
```

There is also a more compact form of the definition for simple structs:

```
type name struct { field1 type1, field2 type2..}
```

An example of two struct definitions appear on the next page.

```
 type employee struct {
  firstname, lastname string
  id  int
  job string
  salary int }
```

```
type point struct {x,y float }
```

There are very few restrictions on what type a field can be, the primary one of interest to us is that the structure definition cannot be recursive, meaning the struct cannot have itself as the type of one of its fields.

For example, the following is illegal:

```
type employee struct {
  fname, lname string
  id  int
  job string
  salary int
  supervisor employee  // not allowed
}
```

But we can build up structs using other structs, a technique which is referred to as composition. For example

```
  type circle {center point, radius float}
```

Structs are a lot like arrays in terms of how they behave, in fact an array is a struct in which all the the fields are of the same type and can be referenced by position instead of name. A good guess for how a struct would act in a program is to think about how an array would act in the same situation.

Like arrays, all of the memory to hold the fields of a struct is allocated contiguously which means that structs are very efficient in terms of performance.

## *8.2 Creating structs*

Given a struct definition, there are two ways to create objects of that type, which we also call instances of the struct or type,

### *8.2.1 Variable declaration*

Just like arrays, we can define variables of our type like any other type

```
var bob employee
var origin point
```

In these cases, the appropriate object is created and initialized by initializing each field to its default zero value. In the above example, origin is initialized to {0,0}

```
// Example 08-01 Basic struct definiton

package main

import "fmt"

type point struct{ x, y int }
type employee struct {
  fname, lname string
  id           int
  job          string
  salary       int          }

func main() {

  var anil employee
  var p point
  fmt.Println("anil=", anil, "p=", p)
}
```

```
[Module058$ go run ex08-01.go
anil= {  0  0} p= {0 0}
```

### *8.2.2 Initialization*

Just like other variables, we can use literals to initialize structs

```
var anil = employee {“Anil”, “Patel”, 8971,
                     “Developer”, 100000}

p := point{4,2}
```

Selected fields may be initialized

```
var anil = employee{id: 8971, fname: “Anil”, lname: “Patel”}
```

In this last form, because field names are provided, the order does not matter. Any field not named in the initial list is set to its default zero value.

```
// Example 08-02 Struct initialization
package main

import "fmt"

type point struct{ x, y int }
type employee struct {
  fname, lname string
  id           int
  job          string
  salary       int
}

func main() {

  var p = point{2, 3}
  fmt.Println("p =", p)

  anil := employee{"Anil", "Patel", 9891,
                  "Developer", 100000}
  greta := employee{id: 8897, fname: "Greta", lname: "Smith"}

  fmt.Println("anil =", anil)
  fmt.Println("greta =", greta)
}
```

```
[Module08]$ go run ex08-02.go
p = {2 3}
anil = {Anil Patel 9891 Developer 100000}
grea = {Greta Smith 8897  0}
```

### *8.2.3 Using new*

We an also use the new() function to allocate memory for a struct object and return a pointer to the newly allocated object

```
var greta *employee = new(employee)
```

The usual sort of notation for pointers applies. We assign the value returned by new to a pointer, then we can access the underlying object with the de-referencing operator *.

The notation

```
p := &point{4,5}
```

is a short way for calling the new operator and then initializing the fields of the new point object. This is a very common idiom in Go. We will return to pointers shortly since they are incredibly useful when working with structs.

```go
// Example 08-03 Using new

package main

import "fmt"

type point struct{ x, y int }

func main() {

  var pp1 *point = new(point) // pp1 is a pointer
  fmt.Println("pp1 =", pp1, "*pp=", *pp1)

  p := point{3, 4} // p is a variable
  pp := &p         // pp is a pointer
  fmt.Println("pp =", pp, "*pp=", *pp, "p=", p)

  pp2 := &point{5, 6} //pp2 is a pointer
  fmt.Println("pp2 =", pp2, "*pp2=", *pp2)
}
```

```
[Module08]$ go run ex08-03.go
pp1 = &{0 0} *pp= {0 0}
pp = &{3 4} *pp= {3 4} p= {3 4}
pp2 = &{5 6} *pp2= {5 6}
```

To keep it straight, the following might help:

```
bob := employee{0,0} bob is an object
ptr_to_bob := &bob   ptr_to_bob is the address of bob

ptr_to_sue := &employee{fname: "Sue"}
*(ptr_to_sue)  is the sue object pointed to by
ptr_to_sue
```

## 8.3 Working with structs

The fields of a struct are accessed using selectors, which is the usual dot notation used by most C-style programming languages. The dot notation works the same whether or not the variable is a value object or a pointer.

```go
// Example 08-04 Field access

package main

import "fmt"

type point struct{ x, y int }

func main() {
    p := point{2, 3}        // object
    pp := &point{4, 5}    // pointer
    fmt.Println("x coord of p", p.x)
    fmt.Println("x coord of pp", pp.x)
    pp.y = -1
    p.y = 0
    fmt.Println("p=", p, " pp=", *pp)
}
```

```
[Module08]$ go run ex08-04.go
x coord of p 2
x coord of pp 4
p= {2 0}  pp= {4 -1}
```

### *8.3.1 Comparing structs*

Like arrays, the operator == is defined on a struct if all the fields in the struct have the == defined for their type.

Two structs are equivalent if they are the same type and the each corresponding field in the two structs is also equivalent.

Warning! When working with pointers be sure you are comparing the objects they point to and not the pointers themselves. In example 08-05 pp1 and pp2 point to two different point objects, but the point objects themselves are equivalent.

pp1 == pp2  means "Are we both pointing to the same object?"

*pp1 == *pp2 means "Are the objects we are pointing to equivalent?"

```go
   // Example 08-05 Comparisons

   package main

   import "fmt"

   type point struct{ x, y int }

   func main() {

     p1 := point{2, 3}
     p2 := point{4, 5}
     p3 := point{2, 3}

     fmt.Println("p1 == p2? ", p1 == p2)
     fmt.Println("p1 == p3? ", p1 == p3)

     pp1 := &point{1, 1}
     pp2 := &point{1, 1}

     fmt.Println("pp1 == pp2? ", pp1 == pp2)
     fmt.Println("*pp1 == *pp? ", *pp1 == *pp2)
}
```

```
[Module08]$ go run ex08-05.go
p1 == p2?  false
p1 == p3?  true
pp1 == pp2?  false
*pp1 == *pp?  true
```

### *8.3.2 Struct pointers and functions*

What slices do for arrays, pointers do for structs. Because structs are value object, like arrays, when we pass them to a function, the function works with a copy of the struct. This has a couple of disadvantages

1. If our structs start to get large or we pass them around a lot, or a combination of the two, our program efficiency start to take a hit from the constant copying.
2. Functions are working with copies of our structs which means they can't make any changes to the structs they are passed.

Just like we used slices to deal with these issues for arrays, we do the same with pointers for structs

In example 08-06, we have two versions of a function that swaps the x and y co-ordinates of a point. The first one – swap1 – is passed the point as an object. Because it is working on a copy of the point, the changes it made locally are never propagated back to the original argument.

The second version – swap2 – works with a pointer to the original "a", which is why &a (the address of a) is used as an argument. As can be seen by the output, swap2 successfully mutates a.

```go
// Example 08-06 Struct pointers

package main

import "fmt"

type point struct{ x, y int }

func swap1(p point) {
  p.x, p.y = p.y, p.x
  fmt.Println("After executing swap1 p=", p)
}
func swap2(p *point) {
  p.x, p.y = p.y, p.x
}

func main() {
  a := point{1, 2}
  fmt.Println("Original a =", a)
  swap1(a)
  fmt.Println("After swap1 a =", a)
  swap2(&a)
  fmt.Println("After swap2 a =", a)
}
```

```
[Module08]$ go run ex08-06.go
Original a = {1 2}
After executing swap1 p= {2 1}
After swap1 a = {1 2}
After swap2 a = {2 1}
```

## 8.4 Embedded structs

Structs can be embedded in two ways in other structs. The first is where we just use the one struct as a field type in a larger struct.

In example 08-07, we do just that, using a point struct as a field in a circle struct. The selector works as you would expect in setting the x co-ordinate of the center.

```go
// Example 08-07 Embedded structs 1

package main

import "fmt"

type point struct{ x, y int }

type circle struct {
  center point
  radius float32
}

func main() {
  c := circle{point{50, 32}, 13.0}
  fmt.Println("c=", c)
  c.center.x = 0
  fmt.Println("c=", c)
}
```

```
[Module08]$ go run ex08-07.go
c= {{50 32} 13}
c= {{0 32} 13}
```

### *8.4.1 Anonymous fields*

An anonymous field is one that only has a type but no name. By using an anonymous field we can access the fields in the inner struct as if they were fields in the outer struct. This is best illustrated in example 08-08 where we have two anonymous fields.

A point structure is used anonymously within a circle struct which means that we can reference the fields of the point struct as if they were fields of circle. There is also a bool which does not have a name which can also be used anonymously but has a bit of an odd selector syntax.

```go
// Example 08-08 Anonymous fields

package main

import "fmt"

type point struct{ x, y int }

type circle struct {
  point
  bool
  radius float32
}

func main() {
  c := circle{point{50, 32}, false, 3.0}
  fmt.Println("c=", c)
  c.x = 0
  c.bool = true
  fmt.Println("c=", c)
}
```

```
[Module08]$ go run ex08-08.go
c= {{50 32} false 3}
c= {{0 32} true 3}
```

There are a couple of rules about anonymous fields

1. There cannot be two anonymous fields of the same type.
2. Names in the outer struct shadow names in the inner struct
3. Two anonymous fields of different types may have fields with the same name but an error will occur if either of the fields with the same are actually accessed

The last one is a bit odd, but basically the conflict in names between fields of two different anonymous structures are not picked up unless one of them is referenced at which time the ambiguity is discovered.

## 8.5 Pointers as fields

Struct definitions cannot contain themselves recursively, however containing a pointer instead is not recursive. Consider the problem we had before at the start of this module linking an employee to their boss who was also an employee. By converting the boss field to a pointer, the structure works the way we would want it to.

```go
// Example 08-09 Pointers as fields

package main

import "fmt"

type employee struct {
  fname, lname string
  id           int
  job          string
  salary       int
  boss         *employee
}

func main() {
  greta := employee{fname: "Greta", lname: "Smith"}
  anil := employee{fname: "Greta", lname: "Smith",
                                               boss: &greta}
  fmt.Println("Anil's boss is", anil.boss.fname)
}
```

```
[Module08]$ go run ex08-09.go
Anil's boss is Greta
```

## 8.6 Pseudo-Constructors

In most OO languages, one of the roles of the constructors of a class is to ensure that the object instantiated from the class is logically correct. Since we don't have constructors in Go, we can emulate the role of a constructor by applying the factory pattern and define a function that generates well formed structs for us. This is quite straightforward if we use pointers.

Consider for example the point struct. We may, for some reason decide that in our application a point can only be in the upper right quadrant, ie, both x and y are non-negative. If either component is negative, then we reflect the point around the appropriate axes.

```go
// Example 08-10 Struct factory

package main

import "fmt"

type point struct{ x, y int }

func makePoint(x, y int) *point {
  if x < 0 {
       x = -x
  }
  if y < 0 {
       y = -y
  }
  return &point{x, y}
}
func main() {
  p1 := makePoint(1, 1)
  fmt.Println(*p1)
  p1 = makePoint(-4, -9)
  fmt.Println(*p1)
}
```

```
[Module08]$ go run ex08-10.go
{1 1}
{4 9}
```