9 Structs และ Interfaces

ถ้าเราจะเขียนโปรแกรมด้วยโกโดยการใช้แค่ประเภทข้อมูลที่บิลท์อินมา(built-in)อย่างเดียวก็ไม่มีใครว่าอะไร มันเป็นไปได้ แต่ในบางครั้งการทำแบบนั้นมันก็ดูเป็นเรื่องที่ทรมานตัวเองใช่ย่อย ยกตัวอย่างเช่นถ้าเราต้องเขียนโปรแกรมที่ต้องทำงานคำนวณหาพื้นที่ของรูปร่างต่างๆ เราจะเขียนออกมาได้ประมาณนี้

```
package main
import ("fmt"; "math")
func distance(x1, y1, x2, y2 float64) float64 {
     a := x2 – x1
     b := y2 – y1
     return math.Sqrt(a*a + b*b)
}
func rectangleArea(x1, y1, x2, y2 float64) float64 {
     l := distance(x1, y1, x1, y2)
     w := distance(x1, y1, x2, y1)
     return l * w
}
func circleArea(x, y, r float64) float64 {
     return math.Pi * r*r
}
func main() {
     var rx1, ry1 float64 = 0, 0
     var rx2, ry2 float64 = 10, 10
     var cx, cy, cr float64 = 0, 0, 5
     fmt.Println(rectangleArea(rx1, ry1, rx2, ry2))
     fmt.Println(circleArea(cx, cy, cr))
}
```

การไล่ตามเก็บค่า พิกัด หลายๆตัวพร้อมๆกันเป็นเรื่องปวดกระบาลและทำให้โปรแกรมเข้าใจยากมากมายและสิ่งนี้อาจส่งผลให้เราทำอะไรที่ผิดพลาดได้

9.1 Structs

ถ้าเราอยากให้โปรแกรมนี้ดูดีขึ้นเราสามารถนำสตรัคมาใช้ได้โดยที่สตรัคเองเป็นประเภทของข้อมูลที่ใช้เก็บฟิลด์ประเภทต่างไว้ได้ ยกตัวอย่างเช่นถ้าเราต้องการสร้าง struct ของ Circle เราก็สามารถทำได้ดังนี้

```
type Circle struct {
     x float64
     y float64
     r float64
}
```

คีย์เวิร์ด type เป็นตัวที่ใช้บอกว่าเรามี ไทป์ใหม่และจากนั้นเราจะใส่ชื่อไทป์ลงไปโดยตัวอย่างนี้เราใช้ Circle และแน่นอนว่าคีย์เวิร์ดสตรัคท์มีไว้เพื่อบอกว่าไทป์ใหม่นี้เป็นสตรัคท์โดยที่ภายในเครื่องหมายวงเล็บปีกกาจะเป็นบริเวณที่เรามีไว้เพื่อประกาศฟิลด์ที่อยู่ในสตรัคท์นั้นๆ อย่างไรก็ตามถ้าเรามีฟิลด์ที่เป็นประเภทเดียวกันหลายๆตัวเราสามารถรวมเป็นกลุ่มไว้เป็นบรรทัดเดียวกันได้เช่นกัน

```
type Circle struct {
     x, y, r float64
}
```

กำหนดค่าตั้งต้น (Initialization)
หลังจากประกาศ type แล้วต่อไปเราจะนำมาใช้งานเราสามารถกำหนดค่าตั้งต้นได้หลายแบบมากเช่น

```
var c Circle
```

การประกาศแบบนี้จะให้ผลเหมือนการประกาศข้อมูลประเภทอื่นๆคือเราจะได้ตัวแปรโลคอลมาหนึ่งตัวที่มีค่าพื้นฐานเท่ากับศูนย์โดยสำหรับสตรัคส์แล้วการได้ค่าศูนย์แปลว่าทุกๆฟิลด์ภายใต้สตรัคส์นั้นจะมีค่าเริ่มต้นตามประเภทของข้อมูลเช่น 0 ของ int, 0.0 ของ floats, “” สำหรับ string และ nil สำหรับ pointer นอกจากนี้แล้วคำสั่ง new ก็ยังเป็นอีกทางเลือกหนึ่ง

```
 c := new(Circle)
```

คำสั่ง new จะทำการจองหน่วยความจำสำหรับทุกๆฟิลด์และทำการกำหนดค่าศูนย์ให้กับทุกๆฟิลด์จากนั้นก็ทำการส่ง pointer (*Circle) กลับออกมาอย่างไรก็ตามในบางกรณีเราต้องการกำหนดค่าเริ่มต้นให้กับฟิลด์ต่างๆเราก็สามารถทำได้ดังนี้

```
c := Circle{x: 0, y: 0, r: 5}
```

หรือถ้ามันยาวไปเราก็สามารถละชื่อฟิลด์ไว้ในฐานที่เข้าใจในกรณีที่เราเรียงลำดับถูกต้อง

```
 c := Circle{0, 0, 5}
```

ฟิลด์ (Fields)

เราสามารถเข้าถึงฟิวด์ต่างๆในสตรัคส์ได้ด้วยการใช้เครื่องหมาย . ตามตัวอย่าง

```
fmt.Println(c.x, c.y, c.r)
c.x = 10
c.y = 5
```

ดังนั้นเราจึงสามารถปรับแต่งฟังก์ชั่น circleArea เพื่อเปลี่ยนไปใช้งาน Circle ได้แบบนี้

```
func circleArea(c Circle) float64 {
     return math.Pi * c.r*c.r
}
```

และใน main เราจะได้ของหน้าตาแบบนี้

```
c := Circle{0, 0, 5}
fmt.Println(circleArea(c))
```

แต่อย่างไรก็ตามสิ่งหนึ่งที่เราต้องระลึกไว้เสมอคืออาร์กิวเมนท์จะถูกส่งผ่านโดยวิธีคัดลอกค่าเสมอในโก ดังนั้นถ้าเราต้องการเปลี่ยนค่าอะไรก็ตามในฟังก์ชั่น circleArea มันจะไม่กลับไปเปลี่ยนค่าที่ต้นทาง นั่นทำให้เราต้องเปลี่ยนการส่งอาร์กิวเมนท์ใหม่ให้ส่งเป็นพอยท์เตอร์(pointer) ไปแทนในกรณีที่เราต้องการแก้ไขอะไรบางอย่างในฟังก์ชั่น

```
func circleArea(c *Circle) float64 {
     return math.Pi * c.r*c.r
}
```

และใน main เราจะได้ของหน้าตาแบบนี้

```
c := Circle{0, 0, 5}
fmt.Println(circleArea(&c))
```

9.2 เมธอด(Methods)

โค้ดของเราเริ่มดีขึ้นแต่เรายังสามารถทำให้ดีกว่านี้ได้อีกด้วยการใช้ฟังก์ชั่นแบบพิเศษที่เรียกว่าเมธอด

```
func (c *Circle) area() float64 {
     return math.Pi * c.r*c.r
}
```

เราสามารถสร้างเมธอดให้สตรัคได้ด้วยการประกาศฟังก์ชั่นขึ้นมาแล้วเพิ่มรีซีพเวอร์(receiver) เข้าไประหว่างคำว่า func ในที่นี้เราใช้ (c *Circle)และชื่อของฟังก์ชั่นดังนั้นเราจะเห็นรีซีพเวอร์คล้ายๆกับเป็นพารามิเตอร์ - เพราะมันมีทั้งชื่อและประเภท - แต่สิ่งที่ต่างจากการประกาศฟังก์ชั่นทั่วไปคือการประกาศแบบนี้เราสามารถเรียกฟังก์ชั่นได้ด้วยการใช้เครื่องหมาย .

```
fmt.Println(c.area())
```

จะเห็นว่าโค้ดแบบนี้อ่านง่ายกว่าเดิมเยอะและเรายังไม่ต้องใช้เครื่องหมาย & อีกต่อไป (เพราะโกจะรู้เองโดยอัตโนมัติว่ามันจะต้องส่งพอยท์เตอร์ของ circle สำหรับเมธอดนี้)และฟังก์ชั่นนี้เองสามารถถูกใช้งานได้ผ่าน Circle เท่านั้นดังนั้นเราจึงควรเปลี่ยนชื่อฟังก์ชั่นให้เหลือแค่ area และเราก็ควรทำสิ่งนี้ที่ Rectangle เช่นกัน

```
type Rectangle struct {
     x1, y1, x2, y2 float64
}

func (r *Rectangle) area() float64 {
     l := distance(r.x1, r.y1, r.x1, r.y2)
     w := distance(r.x1, r.y1, r.x2, r.y1)
     return l * w
}
```

โดยใน main จะเปลี่ยนเป็นแบบนี้

```
r := Rectangle{0, 0, 10, 10}
fmt.Println(r.area())
```

ไทป์แบบฝังตัว (Embedded Types)
โดยปกติแล้ว struct จะใช้สื่อความสัมพันธ์แบบ has-a ยกตัวอย่างเช่น "Circle has-a radius” ดังนั้น ถ้าเรามี Person สตรัคที่มีหน้าตาแบบนี้

```
type Person struct {
    Name string
}
func (p *Person) Talk() {
    fmt.Println("Hi, my name is", p.Name)
}
```

และเราต้องการสร้าง Android struct ที่มีความสัมพันธ์กับ Person เราสามารถทำได้แบบนี้

```
type Android struct {
    Person Person
    Model string
}
```

โค้ดชุดนี้สามารถทำงานได้ดีแต่สิ่งที่เปลี่ยนไปคือวิธีอ่านเป็น “Android is a Person” แทนที่จะเป็น “Android has a Person” ดังนั้นไทป์ฝังตัวจึงเหมาะสมสำหรับการสร้างความสัมพันธ์แบบนี้อย่างไรก็ตามการเขียนไทป์ฝังตัวยังสามารถเขียนได้อีกแบบคือ

```
type Android struct {
    Person
    Model string
}
```

เราใช้ไทป์ Person โดยที่เราไม่ได้ประกาศชื่อเลย ซึ่งการประกาศแบบนี้เราสามารถเรียก Person struct ได้ด้วยการเรียกชื่อไทป์ได้เลย

```
a := new(Android)
a.Person.Talk()
```

แต่ถ้าการเรียกผ่านไทป์ดูเวิ่นเว้อเราสามารถเรียกใช้เมธอดของ Person ตรงๆได้เลยเช่นกัน

```
a := new(Android)
a.Talk()
```

ดังนั้นเราจึงสามารถอ่านความสัมพันธ์แบบ is-a ได้ในลักษณะนี้ “People can talk, an android is a person, therefore an android can talk”

9.3 Interfaces

จากตัวอย่างด้านบนเราจะเห็นว่าเราสามารถตั้งชื่อเมธอด area ให้ Rectangle และมีเมธอด area สำหรับ Circle อีกเช่นกันเราจะเห็นว่ามันซ้ำกัน และเรื่องแบบนี้เกิดขึ้นได้เสมอในการทำงาน เราจะเห็นว่าทั้ง Rectangle และ Circle มีของที่เหมือนกันอยู่ ดังนั้นสำหรับโกเองเมื่อเกิดเรื่องแบบนี้ขึ้นเราสามารถใช้ Interface เข้ามาช่วยแก้ปัญหาได้ ตัวอย่างด้านล่างเป็นตัวอย่างการสร้าง Shape Interface

```
type Shape interface {
     area() float64
}
```

เราจะเห็นว่าการสร้าง interface นั้นจะละม้ายคล้าย struct มากโดยเราจะใช้คำสั่ง type ก่อนจากนั้นตามด้วยชื่อของ interface และจบด้วยคำว่า interface แต่สิ่งที่ต่างกันระหว่างสตรัคกับอินเทอร์เฟสคือ สำหรับอินเทอร์เฟสเราจะประกาศชุดของเมธอดแทนชุดของฟิวด์ ชุดของเมธอดเหล่านี้จะเป็นตัวบังคับว่าไทป์ใดๆก็ตามที่ต้องการอิมพลีเมนท์ interfaceนี้จะต้องเขียนรายละเอียดให้เมธอดเหล่านั้น
สำหรับกรณีของเราจะเห็นได้ว่าทั้ง Rectangle และ Circle มีเมธอด area ที่คืนค่า float64s ดังนั้นเราจึงสามารถบอกได้ว่าทั้งคู่เป็น implementation ของ Shape
ดังนั้นเราจึงสามารถใช้ความเหมือนของทั้งสองสตรัคให้เป็นประโยชน์ได้ด้วยการ ส่งอินเทอร์เฟสเข้าไปเป็นอาร์กิวเมนท์ของฟังก์ชั่นดังตัวอย่างได้

```
func totalArea(shapes ...Shape) float64 {
     var area float64
     for _, s := range shapes {
           area += s.area()
     }
```

```
	return area
}
```

และเราสามารถเรียกใช้เมธอดได้แบบนี้

```
fmt.Println(totalArea(&c, &r))
```

และ interface เองยังสามารถถูกใช้เป็นฟิวด์ได้

```
type MultiShape struct {
	shapes []Shape
}
```

นอกจากนี้เรายังสามารถเปลี่ยน MultiShape ให้เป็น Shape ได้เพื่อดึงเอา area ออกมาด้วยการเรียกเมธอดได้ดังนี้

```
func (m *MultiShape) area() float64 {
	var area float64
	for _, s := range m.shapes {
		area += s.area()
	}
	return area
}
```

ดังนั้นตอนนี้เราจะเห็นว่า MultiShape สามารถบรรจุได้ทั้ง Circle, Rectangle หรือแม้กระทั่ง MultiShape