Introduction to Programming in Go

9. Methods

Module Topics

1. Methods and Receivers
2. Method Sets
3. Methods on Inner Structs
4. Combining Objects

Methods and Receivers

Methods

1. Methods are functions with one modification.
2. Methods have a special "parameter" called a receiver.
3. The type of the receiver binds the method to that type.
4. Works exactly like a class method in Java.

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

5. The binding is only indicated on the method, not in the type.
6. Methods are invoked using message type notation like Java.

Methods

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

Pointers as Receivers

1. Like parameters, receiver variable are copies, which is why the variable "z" in the example was not negated, "x" in the method was a copy of "z".

2. Methods can take pointers as receivers.

3. If a method is applied to an object, Go automatically takes the address and uses that if the method takes a pointer receiver.

4. If the method is:

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

   then

   z.negate() is converted to (&z).negate()

5. Because of the re-writing, we cannot have the same method name with a receiver of type T and a receiver of type *T.

Using a Pointer Receiver

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

Method Sets

Method Sets

1. Functions cannot be overloaded in Go.

2. However functions can be partioned into method sets.

3. Two functions are in the same method set if they have the same receiver.

4. Two methods can have the same name if they are in different method sets.

5. All of the following are ok - they are in different method sets.

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

func (p * point) negate() {...}

func negate(x float32) float32 {..}
```

Method Overloading

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

Method on Inner Structs

Inner Structs

1. If an outer struct has an inner struct as a field then:
2. The methods of the inner struct can be invoked on the outer struct.
3. But a selector has to be used.
4. This is the OO concept of aggegation and delegation.
5. The outer struct is a container for the inner structs.
6. The outer struct forwards or delegates methods to the appropriate inner struct.

Method Delegation

```
// Example 09-04 Support code

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
}
```

Method Delegation

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

...

func (c *circle) area() float32 {
   c := circle{point{1, 0}, 3.0}
   fmt.Println(c)
   c.center.swap()  // delegated using selector
   fmt.Println(c, c.area())
}
```

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

Anonymous Inner Structs

1. If the inner struct is anonymous, no selector is needed.
2. The method set of the inner struct is now part of the method set of the outer struct.
3. This emulates inheritance.
4. The inner struct acts as the parent class or super class.
5. The outer struct acts as the derived or subclass.
6. Allows for emulation of multiple inheritance.
7. Methods in the outer struct with the same name as a method in the inner struct overrides the inner struct method.

Method Incorportion

```
// Example 09-05 Changes to previous example


type circle struct {
   point     //this is now anonymous
   radius float32
}

func (c *circle) area() float32 {
   c := circle{point{1, 0}, 3.0}
   fmt.Println(c)
   c.swap()  // swap is in circle's method set
   fmt.Println(c, c.area())
}
```

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

Overriding Methods

```
// Example 09-06 Changes to previous example

type circle struct {
   point     //this is now anonymous
   radius float32
}

func (c *circle) swap() {
}

func (c *circle) area() float32 {
   c := circle{point{1, 0}, 3.0}
   fmt.Println(c)
   c.swap()  // swap is now overridden
   fmt.Println(c, c.area())
}
```

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

Combining Objects

Implementation of OO Inheritance

1. Aggregation: Using named inner structs.
2. Inheritance: Using anonymous inner structs.
3. Can emulate multiple inheritance.
4. Requirement that only one anonymous inner struct of a given type allows for something like virtual inheritance in C++.
5. Outer structs (subclass) methods can override inner struct (superclass) methods.

Methods and Inner Structs

Combining Methods Sets

1. Given a struct with an inner struct both of which have method sets

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

2. We can apply both sets of methods by referencing the appropriate struct.

Enbedded Structs 1

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

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
}
```

Embedded Structs 2

```
// 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
```

Combining Objects

Combining Methods Sets

1. Changing our last example

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

2. We can apply both sets of methods on the circle struct.

3. This has the effect of the circle "inheriting" the point methods.

4. Outer struct methods can override inner struct methods.

Embedded Structs 3

```
// 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
```

Method Overloading

1. Changing our last example so that we have a swap method in both structs.

2. The cicle swap() method overrides the point swap() method.

3. To illustrate we add a swap() method to the circle that does nothing.

Method Overriding

```
// 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
}
```

Method Overriding

```
// 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
```

Lab 9: Methods