Golang Programming

Methods. Composing Structs by Type Embedding

Where to Find The Code and Materials?

https://github.com/iproduct/coursegopro

Methods

```go
type Vertex struct {
	X, Y float64
}

func (v Vertex) Abs() float64 {
	return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
	v := Vertex{3, 4}
	fmt.Println(v.Abs())
	abs := Vertex.Abs
	fmt.Println(abs(v))
}
```

Method Sets

Every type has a (possibly empty) method set associated with it:

- The method set of an interface type is its interface.
- The method set of any other type T consists of all methods declared with receiver type T. The method set of the corresponding pointer type *T is the set of all methods declared with receiver *T or T (that is, it also contains the method set of T).
- Further rules apply to structs containing embedded fields. Any other type has an empty method set.
- In a method set, each method must have a unique non-blank name.
- The method set of a type determines the interfaces the type implements and the methods that can be called using a receiver of that type.

Methods on Non Structs

```go
type MyFloat float64

func (f MyFloat) Abs() float64 {
	if f < 0 {
		return float64(-f)
	}
	return float64(f)
}

func main() {
	f := MyFloat(-math.Sqrt2)
	fmt.Println(f.Abs())
}
```

Methods on Non Structs

```go
type Role int

const (
    User Role = 1 << iota
    Manager
    Admin
    RoleMask = (1 << (iota)) - 1
)

func (r Role) String() string {
    switch r {
    case User:
        return "User"
    case Manager:
        return "Manager"
    case Admin:
        return "Admin"
    default:
        return "Invalid role"
    }
}
```

```go
// Status type
type Status int

// User statuses enum
const (
    Registered Status = iota
    Active
    Disabled
)

// Returns string representation of the Role
func (r Status) String() string {
    switch r {
    case Registered:
        return "Registered"
    case Active:
        return "Active"
    case Disabled:
        return "Disabled"
    default:
        return "Invalid status"
    }
}
```

Value and Pointer Receivers

```
type Vertex struct {
	X, Y float64
}

func (v Vertex) Abs() float64 {
	return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func (v *Vertex) Scale(f float64) {
	v.X = v.X * f
	v.Y = v.Y * f
}

func main() {
	v := Vertex{3, 4}
	v.Scale(10)
	fmt.Println(v.Abs())
}
```

Methods Are Just Like Functions

```go
type Vertex struct {
	X, Y float64
}

func Abs(v Vertex) float64 {
	return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func Scale(v *Vertex, f float64) {
	v.X = v.X * f
	v.Y = v.Y * f
}

func main() {
	v := Vertex{3, 4}
	Scale(&v, 10)
	fmt.Println(Abs(v))
}
```

Methods and Pointer Indirection

```go
type Vertex struct {
	X, Y float64
}
func (v *Vertex) Scale(f float64) {
	v.X = v.X * f
	v.Y = v.Y * f
}
func ScaleFunc(v *Vertex, f float64) {
	v.X = v.X * f
	v.Y = v.Y * f
}
func (v Vertex) Abs() float64 {
	return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
func AbsFunc(v Vertex) float64 {
	return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

```go
func main() {
	// Pointer receiver methods
	v := Vertex{3, 4}
	v.Scale(2)
	ScaleFunc(&v, 5)

	p := &Vertex{4, 3}
	p.Scale(5)
	ScaleFunc(p, 2)
	fmt.Println(v, p)

	// Value receiver methods
	fmt.Println(v.Abs())
	fmt.Println(AbsFunc(v))

	fmt.Println(p.Abs())
	fmt.Println(AbsFunc(*p))
}
```

Methods: Value and Pointer Receivers

```go
type ByteSlice []byte

func (slice ByteSlice) Append(data []byte) []byte {
    return append([]byte(slice), data...)
}

func (slice *ByteSlice) AppendPointer(data []byte) {
    *slice = append([]byte(*slice), data...)
}

func (slice *ByteSlice) Write(data []byte) (n int, err error) {
    *slice = append([]byte(*slice), data...)
    return len(data), nil
}

func main() {
        var b ByteSlice
        fmt.Fprintf(&b, "This hour has %d days\n", 7)
        fmt.Printf("%v", b)
}
```

Choosing Value or Pointer Receiver

There are two reasons to use a pointer receiver:

- The first is so that the method can modify the value that its receiver points to.
- The second is to avoid copying the value on each method call. This can be more efficient if the receiver is a large struct, for example.
- In general, all methods on a given type should have either value or pointer receivers, but not a mixture of both.
- More about selectors and method expressions: https://golang.org/ref/spec#Selectors

Method Receivers and Interfaces

```go
type Abser interface {
	Abs() float64
}

func main() {
	var a Abser
	f := MyFloat(-math.Sqrt2)
	v := Vertex{3, 4}

	a = f  // MyFloat implements Abser
	fmt.Println(a.Abs())
	a = &v // *Vertex implements Abser

	// Vertex do not implement Abser
	//a = v

	fmt.Println(a.Abs())
}
```

```go
type MyFloat float64

func (f MyFloat) Abs() float64 {
    if f < 0 {
        return float64(-f)
    }
    return float64(f)
}

type Vertex struct {
    X, Y float64
}

func (v *Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

Methods with Nil Receivers

```go
// Path represents a sequence of Vertices. A nil Path represents empty sequence.
type Path []Vertex
func (p *Path) Distance() (dist float64) {
	dist = 0
	if p == nil || len(*p) == 0 {
		return 0
	}
	v1 := (*p)[0]
	var v2 Vertex
	for i := 1; i < len(*p); i++ {
		v2 = (*p)[i]
		dist += v1.Distance(v2)
		v1 = v2
	}
	return
}
func main() {
	var path Path
	path = Path{{1, 1}, {4, 5}, {4, 1}, {1, 1}}
	fmt.Println("Perimeter = ", path.Distance())
}
```

Composing structs by Type Embedding

```go
type ColorVertex struct {
    Vertex
    Color color.RGBA
}

func main() {
    green := color.RGBA{0, 255, 0, 255}
    yellow := color.RGBA{255, 255, 0, 255}
    cv1 := ColorVertex{Vertex{2, 3}, green}
    cv2 := ColorVertex{Vertex{6, 6}, yellow}
    fmt.Println(cv1.Distance(cv2.Vertex)) // 5
    cv1.Scale(4)
    cv2.Scale(4)
    fmt.Println(cv1.Distance(cv2.Vertex)) // 20
    // cv1.Distance(cv2) // no cv1.Distance(ColorVertex)

}
```

```go
type Vertex struct {
    X, Y float64
}

func (v Vertex) Distance(o Vertex) float64 {
    return math.Hypot(o.X-v.X, o.Y-v.Y)
}

func (v *Vertex) Scale(f float64) {
      v.X = v.X * f
      v.Y = v.Y * f
}
```

Composing structs by Pointer Type Embedding

```go
type ColorVertexP struct {
        *Vertex
        Color color.RGBA
}

cvp1 := ColorVertexP{&Vertex{2, 7}, green}
cvp2 := ColorVertexP{&Vertex{5, 3}, yellow}
fmt.Println(cvp1.Distance(*cvp2.Vertex)) // "5"
cvp1.Vertex = cvp2.Vertex
cvp1.Scale(3)
fmt.Println(*cvp1.Vertex, *cvp2.Vertex) // {15 9} {15 9}
```

Rules for Method Promotion – Value vs. Pointer Fields

Given a struct type S and a defined type T, promoted methods are included in the method set of the struct as follows:

- If S contains an embedded field T, the method sets of S and *S both include promoted methods with receiver T. The method set of *S also includes promoted methods with receiver *T.
- If S contains an embedded field *T, the method sets of S and *S both include promoted methods with receiver T or *T.

Field and Method Selectors

- The selector expression **x.f** denotes the field or method f of the value **x** (or sometimes ***x**).
- A selector **f** may denote a field or method **f** of a type **T**, or it may refer to a field or method **f** of a nested embedded field of **T**.
- The number of embedded fields traversed to reach **f** is called its depth in **T**.
- The depth of a field or method f declared in **T** is zero.
- The depth of a field or method f declared in an embedded field **A** in **T** is the depth of f in **A** plus one.

Rules of Selectors - I

- For a value **x** of type **T** or ***T** where **T** is not a pointer or interface type, **x.f** denotes the field or method at the shallowest depth in **T** where there is such an **f**. If there is not exactly one **f** with shallowest depth, the selector expression is illegal.
- For a value x of type I where I is an interface type, x.f denotes the actual method with name f of the dynamic value of x. If there is no method with name f in the method set of I, the selector expression is illegal.
- As an exception, if the type of x is a defined pointer type and (*x).f is a valid selector expression denoting a field (but not a method), x.f is shorthand for (*x).f.

Rules of Selectors - II

- In all other cases, x.f is illegal.
- If x is of pointer type and has the value nil and x.f denotes a struct field, assigning to or evaluating x.f causes a run-time panic.
- If x is of interface type and has the value nil, calling or evaluating the method x.f causes a run-time panic.

Method Values and Expressions

```
a := Vertex{2, 7}
b := Vertex{5, 3}

distance := Vertex.Distance // method expression
fmt.Println(distance(a, b)) // 5
fmt.Printf("%T\n", distance) // func(main.Vertex, main.Vertex) float64

scale := (*Vertex).Scale // method expression
scale(&a, 2)
fmt.Println(a)             // {4 14}
fmt.Printf("%T\n", scale) // func(*main.Vertex, float64)

scaleB := (&b).Scale  // method value
fmt.Printf("%T\n", scaleB) // func(float64)
scaleB(2)
fmt.Printf("Sacling b with factor 2: b now is %f\n", b) //{10 6}

distanceFromA := a.Distance // method value
fmt.Printf("%T\n", distanceFromA) // func(*Vertex, float64)
fmt.Printf("Distance from A of B is %f\n", distanceFromA(b)) //10
```

Encapsulation

- State encapsulation:

```
type IntSet struct {
    words []uint64
}
```

- No state encapsulation:

```
type IntSet []uint64
```

Examples

- IntBitSet
- PriorityQueue
- HttpServer

JSON Marshalling and Unmarshalling

```go
// Structs --> JSON
data, err := json.Marshal(goBooks)
if err != nil {
	log.Fatalf("JSON marshaling failed: %s", err)
}
fmt.Printf("%s\n", data)

// Prettier formatting
data, err = json.MarshalIndent(goBooks, "", "     ")
if err != nil {
	log.Fatalf("JSON marshaling failed: %s", err)
}
fmt.Printf("%s\n", data)

// JSON -> structs
var books []Book
if err := json.Unmarshal(data, &books); err != nil {
	log.Fatalf("JSON unmarshaling failed: %s", err)
}
fmt.Println("AFTER UNMARSHAL:\n", books)
```

HTTP Client Example

```
import ("bufio"; "fmt"; "log"; "net/http")

func main() {
		//resp, err := http.Get("http://localhost:8080/headers")
		//resp, err := http.Get("http://google.com")
		req, err := http.NewRequest("GET", "http://localhost:8080/headers", nil)
		req.Header.Add("Accept", `Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8`)
		req.Header.Add("Custom-Header", `Custom Value`)
		resp, err := http.DefaultClient.Do(req)
		if err != nil {
				log.Fatal(err)
		}

		fmt.Println("Response status:", resp.Status)
		scanner := bufio.NewScanner(resp.Body)
		for i := 0; scanner.Scan() && i < 10; i++ {
				fmt.Println(i+1, ": ", scanner.Text())
		}
}
```

Homework 2 (GitHub API Client)

Implement GItHub API HTTP client that will:

- Read a text file given as command line argument to the program and parse different Github usernames – each username on separate line in the file
- Fetch GitHub users data in JSON format using public GitHub API: https://api.github.com/users/${username}
- Fetch GitHub user repositories data in JSON format from: https://api.github.com/users/${username}/repos
- Fetch information about programming languages in each repo from: https://api.github.com/repos/${username}/${repo-name}/languages
- Parse the JSON data using json.Unmarshal into appropriate data structures in Go (you could define only fields that are interesting, all fields exported = starting with capital letter).
- Print a statistics report containing the information about the user, the number of user repositories, the distribution of programming languages according to their usage numbers (third URL), the total number of followers, number of forks for all repositories, by year distribution of user activity calculated using repositories creation and last update dates. Format the report and print it to the console as a table with all users to allow comparison.

Recommended Literature

- The Go Documentation - https://golang.org/doc/
- The Go Bible: Effective Go - https://golang.org/doc/effective_go.html
- David Chisnall, *The Go Programming Language Phrasebook*, Addison Wesley, 2012
- Alan A. A. Donovan, Brian W. Kernighan, *The Go Programming Language*, Addison Wesley, 2016
- Nathan Youngman, Roger Peppé, *Get Programming with Go*, Manning, 2018
- Naren Yellavula, *Building RESTful Web Services with Go*, Packt, 2017

Thank’s for Your Attention!

Trayan Iliev

IPT – Intellectual Products & Technologies

http://iproduct.org/

http://robolearn.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD