Go Programming - OOP

Concepts of Programming Languages

Sebastian Macke
Rosenheim Technical University

Last lecture

- Types (string, int, bool, float64, ...)
- weak vs. strong typing, statically vs. dynamically typed.
- Functions and Control Structures
- Arrays, Slices and Maps
- Pointer
- Unit Tests

Last Exercise

- Let's look at an example implementation

3

Error Handling

Errors in Go

Go doesn't have try-catch exception based error handling.
Go distinguishes between recoverable and unrecoverable errors:

Recoverable: e. g. file not found

- Go returns the error as one of the return values

```
err := ioutil.WriteFile(src.Name(), []byte("hello"), 0644)
if err != nil {
    log.Error(err)
}
```

Unrecoverable: array access outside its boundaries, out of memory

Go defer: run code before function exists

The "defer" statement lets us ensure that code runs before a function exits

```
f, err := os.Open("myfile.txt")
if err != nil {
    return err
}
defer f.Close() // Will be executed on function exit, even in case of an unrecoverable error.
....
// read, write
```

6

unrecoverable error: panic and recover by using defer

```
func getElement(array []int, index int) int {
    if index >= len(array) {
        panic("Out of bounds")
    }
    return array[index]
}

func getElementWithRecover(array []int, index int) (value int) {
    defer func() {
        r := recover()
        if r != nil {
            fmt.Println("Recovered with message '", r, "'")
        }
        value = -1
    }()
    return getElement(array, index)
}

func main() {
    array := []int{3, 4, 5}
    ret := getElementWithRecover(array, 3)
    fmt.Println("return value: ", ret)
}
```

7

Exception Pro and Cons

- What do you think are the pros and cons of exceptions?
- Miro Board ...

8

Object Oriented Programming

9

Structure of object oriented programming

Wikipedia: Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods).

- **Classes**: Data and methods which acts on the data. Blueprint for objects
- **Objects**: instances of classes
- **Fields**, **attributes**, **properties**: State of the object
- **Procedure**, **Methods**: Associated to an object.

Principles of Object Oriented Programming

- **Encapsulation:** Bundling of data with the methods that operate on that data. Select what is public accessible and what is private. Ensure that the consistency of the data is maintained.
- **Abstraction:** Provide abstraction of a class without the implementation. Manage the complexity by hiding.
- **Inheritance:** Based class upon another class. Goal is reusing the parent class properties.
- **Polymorphism:** Objects of different types can be accessed through the same interface or the use of a single variable to represent different types

Go is not a pure object oriented programming language but
allows an object-oriented style of programming

Encapsulation I: No classes, but structs

```
// Rational represents a rational number numerator/denominator.
type Rational struct {
    numerator   int
    denominator int
}

// Constructor
func NewRational(numerator int, denominator int) Rational {
    if denominator == 0 {
        panic("division by zero")
    }
    return Rational{
        numerator:   numerator,
        denominator: denominator,
    }
}
```

- Capitalized fields are public outside of the package. E. g. **Rational** and **NewRational** are public while **Rational.numerator** and **Rational.denominator** are not.

Encapsulation II: Bind together code and data it manipulates

- Classical Procedural syntax (Used often in C)

```
// Multiply method for rational numbers
func Multiply(r Rational, y Rational) Rational {
    return NewRational(r.numerator*y.numerator, r.denominator*y.denominator)
}
```

- Object Oriented style syntax

```
// Multiply method for rational numbers
func (r *Rational) Multiply(y Rational) Rational {
    return NewRational(r.numerator*y.numerator, r.denominator*y.denominator)
}

r1 := NewRational(1, 2)
r2 := NewRational(2, 4)
r3 := r1.Multiply(r2)
```

The variable **r** is in both cases similar to the Java **this**, which reference to the class instance

Orthogonal: Encapsulation can be done with any type

```
package main

import "fmt"

type MyInt int

func (b *MyInt) Inc() {
    *b = *b + 1
}

func main() {
    var b MyInt = 10
    fmt.Println(b)
    b.Inc()
    fmt.Println(b)
}
```

14

Syntax level OOP

```
package main

import "fmt"

type Bar struct{}

func (b *Bar) GetHello() string {
    fmt.Println(b)
    return "Hello"
}

type Foo struct {
    B *Bar
}

func main() {
    var f Foo
    fmt.Println(f.B)            // Output "nil"
    fmt.Println(f.B.GetHello()) // Null Pointer error or "Hello"?
}                               // prints "Hello" bc function is not dependant on struct obj
```

15

Syntax level OOP

Encapsulation of methods is basically is just a syntax element.

```
func (r *Rational) Multiply(y Rational) Rational {
 ....
}
```

is internally transformed into

```
func Multiply(r *Rational, y Rational) Rational {
 ....
}
```

A lot of languages do it this way.

:* Syntax level OOP 16

Composition VS. Inheritance

Composition and inheritance are two ways to achieve code reuse and design modular systems

- Composition

```
class Engine {
    ....
}
class Car {
    private Engine engine
}
```

- Inheritance

```
class Animal {
    ....
}

class Dog extends Animal {
}
```

Composition VS. Inheritance?

Composition and inheritance are two ways to achieve code reuse and design modular systems

- Composition: A design principle where objects are formed by combining multiple smaller objects
- Inheritance: A mechanism where a new class derives properties and behaviors from an existing class
- When to use which? What Pros and Cons exist?

The issue with inheritance

www.youtube.com/watch?v=Ng8m5VXsn8Q&t=414s (https://www.youtube.com/watch?v=Ng8m5VXsn8Q&t=414s)

```
class Runner {
    public void run(Task task) { task.run(); }
    public void runAll(List<Task> tasks) { for (Task task : tasks) { run(task); } }
}

class RunCounter extends Runner {
    private int count = 0;

    @override public void run(Task task) {
        count++;
        super.run(task);
    }

    @override public void runAll(List<Task> tasks) {
        count += tasks.size();
        super.runAll(tasks);
    }
}
```

When I run 3 tasks with RunCounter.runAll, what value does **count** have? 19

// prints 6 bc run() and runAll() now point to the override versions

According to the Go main architects inheritance causes

- Tight Coupling
- Weak Encapsulation
- Surprising Bugs

20

Embedding

- Go does of course support composition.
- But Go does not support inheritance: Go supports embedding of other structs.

```
// Point is a two dimensional point in a cartesian coordinate system.
type Point struct{ x, y int }
```

```
// ColorPoint extends Point by adding a color field.
type ColorPoint struct {
    // Point p // Composition is of cour supported
    Point // Embedding simulates inheritance but it is (sort-of) delegation!
    c     int
}
```

```
    fmt.Println(cp.x)       // access inherited field
```

- Access to embedded field is identical to a normal field inside a struct
- Syntactically it is similar to inheritance in Java

Polymorphism is not possible with embeddings.

```
// Point is a two dimensional point in a cartesian coordinate system.
type Point struct{ x, y int }

// ColorPoint extends Point by adding a color field.
type ColorPoint struct {
    Point // Embedding simulates inheritance but it is (sort-of) delegation!
    c     int
}

var Point p = Point{}
var ColorPoint cp = ColorPoint{}

p = cp // Compile Error
p = cp.Point // Works
```

Delegation of Functions in Go

- Overriding of methods is kind of supported, overloading is not!
- please check ../src/oop/runtask/runtask.go

23

override does not overwrite the baseline implementation, meaning that with the same run task counter, the output would be the expected 3

Polymorphism I

An interface is a set of methods

In Java:

```
interface Switch {
    void open();
    void close();
}
```

In Go:

```
type OpenCloser interface {
    Open()
    Close()
}
```

- Interfaces with very few methods end with the ending "er" (Stringer, Writer, Reader...)

Polymorphism II

- Java interfaces are satisfied explicitly // forced to implement everything

```
class Door implements Switch {
  public void Open() { ... }
  public void Close() { ... }
}
```

- Go interfaces are satisfied implicitly

```
type Door struct {}
func (d *Door) Open() { ... }
func (d *Door) Close() { ... }  // not forced to implement every function from the interface, pro: possibility to implement multiple interfaces
```

Door implicitly satisfies the interface **OpenCloser**

Go supports polymorphism only via interfaces

Polymorphism III: Example The stringer interface

The print functions in the **fmt** package support the following interface

```
type Stringer interface {
    String() string
}
```

- Every type with a String() Method is detected by the print commands and the String function is executed
- Detection if object is type of interface

```
if tmp, ok := object.(Stringer); ok {
    // The object implements stringer
}
```

- Issue: The detection is done at runtime.

Polymorphism III: The stringer interface

```
package main

import "fmt"

type DoorOpen bool

func (d DoorOpen) String() string {
    if d == true {
        return "Door is open"
    } else {
        return "Door is closed"
    }
}

func main() {
    var d DoorOpen = false
    fmt.Println(d)
}
```

An implementation can support multiple interfaces at the same time.

Interfaces and Polymorphism

```
func main() {
    var p = Point{1, 2}
    var cp = ColorPoint{Point{1, 2}, 3} // embeds Point
    fmt.Println(p)
    fmt.Println(cp)
    fmt.Println(cp.x) // access inherited field

    // p = cp       // does not work: No hierarchy, no polymorphism
    // p = cp.Point // works

    // s is an interface and supports Polymorphism
    var s fmt.Stringer
    s = p // check at compile time
    fmt.Println(s)
    s = cp
    fmt.Println(s)
}
```

Recap: Go does support a dynamic type

```
func main() {
    var someValue any
    someValue = 2
    PrintVariableDetails(someValue)

    someValue = "abcd"
    PrintVariableDetails(someValue)

    if tmp, ok := someValue.(string); ok {
        fmt.Println("someValue is a string and has the value", tmp)
    }
}
```

- any is an alias for an empty **interface{}** and hence matches all types

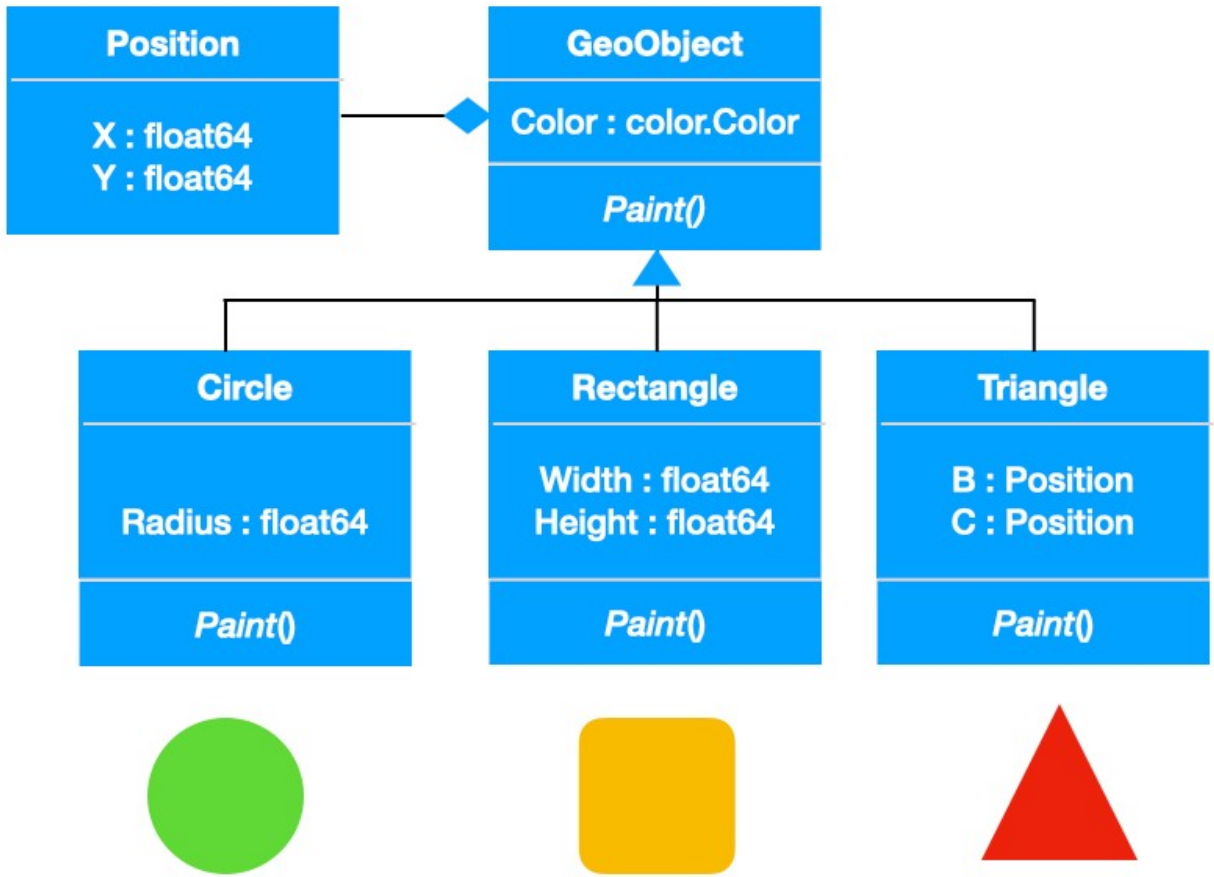
```
type any = interface{}
```

29

Summary

- Go does support Encapsulation via an OOP style syntax
- Go does not support inheritance but type embedding (delegation without syntactic ballast)
- Implicit polymorphism means fewer dependencies and no type hierarchy
- Several interfaces can be put together to form an interface
- Interface embedding makes mocking easy
- Go supports polymorphism only via interfaces, not through classes
- https://youtu.be/Ng8m5VXsn8Q?t=414

Exercise 3


Position
X : float64
Y : float64
GeoObject
Color : color.Color
Paint()
Circle
Radius : float64
Paint()
Rectangle
Width : float64
Height : float64
Paint()
Triangle
B : Position
C : Position
Paint()

Exercise

- Implement the UML diagram with Go
- The Paint() method should print the names and values of the fields to the console
- Allocate an array of polymorph objects and call Paint() in a loop

github.com/s-macke/concepts-of-programming-languages/blob/master/docs/exercises/Exercise3.md (https://github.com/s-macke/concepts-of-programming-languages/blob/master/docs/exercises/Exercise3.md) 32

Questions

- What is the difference between inheritance in Java and embedding in Go?
- How does Go support multiple inheritance? Is is supported for interfaces and types?

Multiple embeddings of same variable or function names

```
type Foo struct {
    Name string
}

type Bar struct {
    Name string
}

type X struct {
    Foo
    Bar
}

func main() {
    y := X{
        Foo: Foo{Name: "Foo!"},
        Bar: Bar{Name: "Bar!"},
    }
    fmt.Print(y.Foo.Name)
    fmt.Print(y.Bar.Name)
    //fmt.Print(y.Name) // compile error, Ambiguous Reference
}
```

Thank you

Sebastian Macke
Rosenheim Technical University
Sebastian.Macke@th-rosenheim.de (mailto:Sebastian.Macke@th-rosenheim.de)
https://www.qaware.de (https://www.qaware.de)