Go Programming - OOP Part II

Concepts of Programming Languages
19 October 2020

Bernhard Saumweber
Rosenheim Technical University

Last lecture

- Types (string, int, bool, float64, ...)
- Functions and Control Structures
- Unit Tests
- Arrays, Slices and Maps
- Comparison
- Pointer
- Type Conversion and Downcast
- Flag API
- Stringer Interface
- OOP: Types, Functions and Interfaces
- OOP: Receiver

OOP

- Classes: data and procedures
- Objects: instances of classes
- Class-based vs. Prototype-based
- Encapsulation
- Polymorphism
- Composition, inheritance, and delegation
- Delegation vs. Consultation or Forwarding

Embedding

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

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

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

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

- In Java this can be done with delegation.
- Syntactically it is similar to inheritance in Java
- Access to embedded field is identical to a normal field inside a struct
- Overriding of methods is supported, overloading is not!

Polymorphism

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

// ColorPoint implements the fmt.Stringer interface.
func (p ColorPoint) String() string {
    return fmt.Sprintf("x=%v,y=%v,c=%v", p.x, p.y, p.c)
    // OR: return fmt.Sprintf("%v,c=%v", p.Point, p.c)  // Delegate to Point.String()
}
```

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

- Interfaces are implemented implicitly
- Go supports polymorphism only via interfaces, not through classes

Interfaces and Polymorphism

```go
func main() {
    var p = Point{1, 2}
    var cp = ColorPoint{Point{1, 2}, 3}

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

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

    // s is a interface and supports Polymorphism
    var s fmt.Stringer
    s = p
    fmt.Println(s.String())
    s = cp
    fmt.Println(s.String())
}
```

Run

Send Mail with Go: A minimal Interface

```go
type Message struct {
    To      string
    Subject string
    Text    string
}

// Sender is a interface to send mails.
type Sender interface {

    // Send a mail to a given address with a subject and text.
    Send(message Message) error
}
```

- A example interface for a service-oriented component

A type implements an interface when providing the required methods

```go
// Package smtp sends mails over the smtp protocol.
package smtp

import (
    "github.com/0xqab/concepts-of-programming-languages/oop/mail"
    "log"
)

// MailSenderImpl is a sender object.
type MailSenderImpl struct {
}

// SendMail sends a mail to a receiver.
func (m *MailSenderImpl) Send(message mail.Message) error {
    log.Printf("Sending message with SMTP:\n  To: %v\n  Subject: %v\n  Text: %v\n",
        message.To, message.Subject, message.Text)
    return nil
}
```

- Import references fully qualified VC directories in $GOPATH/src

The Go interface can be used as in Java

```go
// Package client contains sample code for the mail components.
package client

import (
    "github.com/0xqab/concepts-of-programming-languages/oop/mail"
    "github.com/0xqab/concepts-of-programming-languages/oop/mail/util"
)

// Registry is the central configuration for the service locator
var Registry = util.NewRegistry()

// SendMail sends a mail to a receiver.
func SendMail(to, subject, text string) error {

    // Create an implementation for the mail.Sender interface.
    var sender = Registry.Get("mail.Sender").(mail.Sender)

    email := mail.Message{To: to, Subject: subject, Text: text}
    return sender.Send(email)
}
```

Delegation vs. Consultation or Forwarding

```go
type A struct {
}

func (a A) Foo() {
    a.Bar()
}

func (a A) Bar() {
    fmt.Print("a.bar")
}

type B struct {
    A
}

func (b B) Bar() {
    fmt.Print("b.bar")
}

func main() {
    b := B{}
    b.Foo() // "a.bar" or "b.bar"?
}
```

Run

Summary

- Several interfaces can be put together to form an interface
- Go does not support inheritance but type embedding (delegation without syntactic ballast)
- Go supports polymorphism only via interfaces, not through classes
- Interfaces with one method end with the ending "er" (Stringer, Writer, Reader...)

Video

youtu.be/Ng8m5VXsn8Q?t=414 (https://youtu.be/Ng8m5VXsn8Q?t=414)

Embedding

```go
type Introducer interface { Introduce() }
type Worker interface { Work() }

type Person struct {
    Name string
}

// Person now implicitly satisfies Introducer
func (p Person) Introduce() {
    fmt.Printf("Hello, my name is %s\n", p.Name)
}

type Employee struct {
    Person
    Worker
    EmployeeID int
}
```

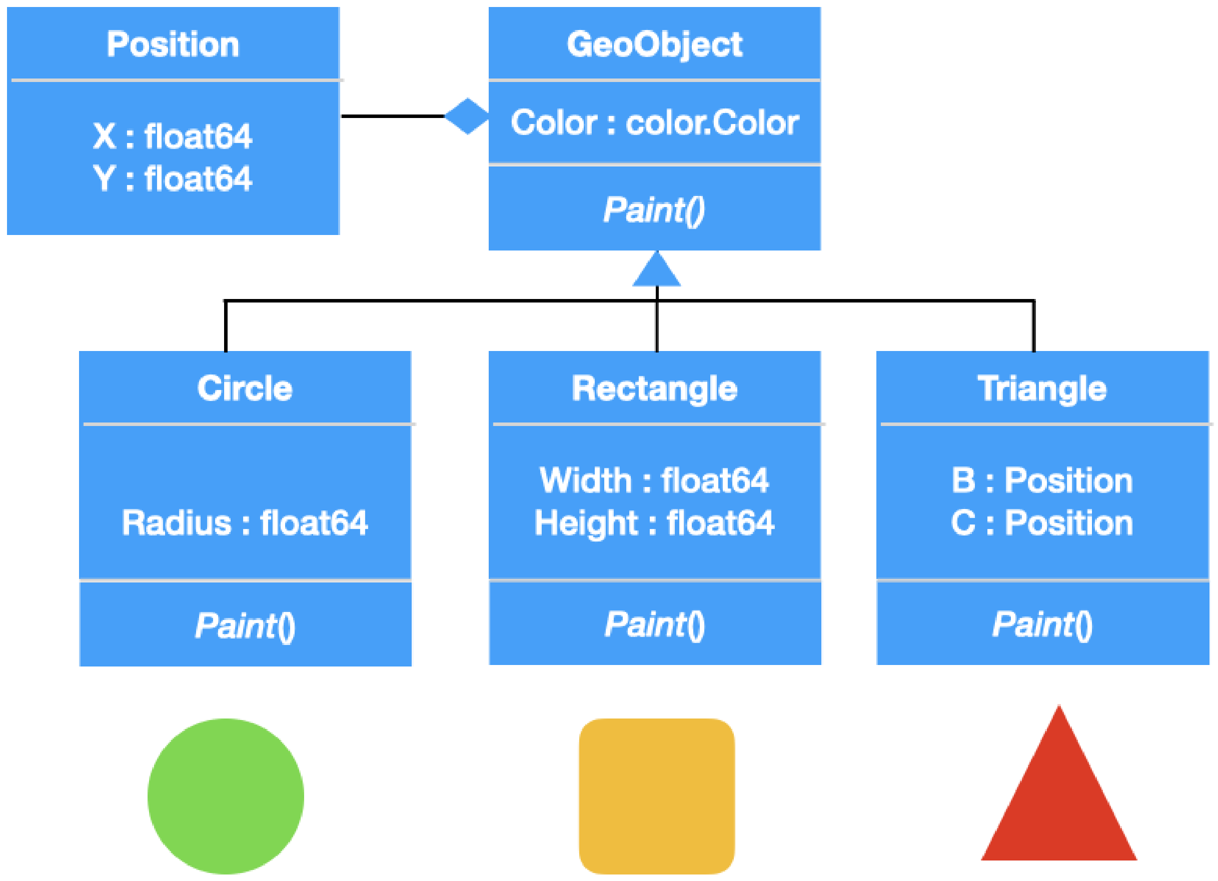
Embedding (cont.)

```go
type Employee struct {
    Person
    Worker
    EmployeeID int
}

// Employee will satisfy Worker even without this declaration
func (e Employee) Work() {
    fmt.Printf("<%s is working>\n", e.Name)
}

func main() {
    e := Employee{
        Person:     Person{"John"},
        EmployeeID: 1,
    }
    e.Introduce() // prints "Hello, my name is John"
    e.Work() // does not require an implementation (but throws exception if not implemented)
}
```

Takeaways

- Implicit polymorphism means fewer dependencies and no type hierarchy
- Inheritance can cause weak encapsulation, tight coupling and surprising bugs
- Struct embedding is still composition, but makes life easier
- Interface embedding makes mocking easy

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/0xqab/concepts-of-programming-languages/blob/master/docs/exercises/Exercise3.md (https://github.com/0xqab/concepts-of-programming-languages/blob/master/docs/exercises/Exercise3.md)

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 inheritance

```go
type Fooer interface {
    Foo()
}

type Barer interface {
    Bar()
}

type X struct {}

func (x X) Foo() {}
func (x X) Bar() {}
```

Multiple inheritance (cont.)

```go
type Foo struct {
    Name string
}

type Bar struct {
    Name string
}

type Y struct {
    Foo
    Bar
}

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

Run

Thank you

Bernhard Saumweber
Rosenheim Technical University
bernhard.saumweber@qaware.de (mailto:bernhard.saumweber@qaware.de)
http://www.qaware.de (http://www.qaware.de)