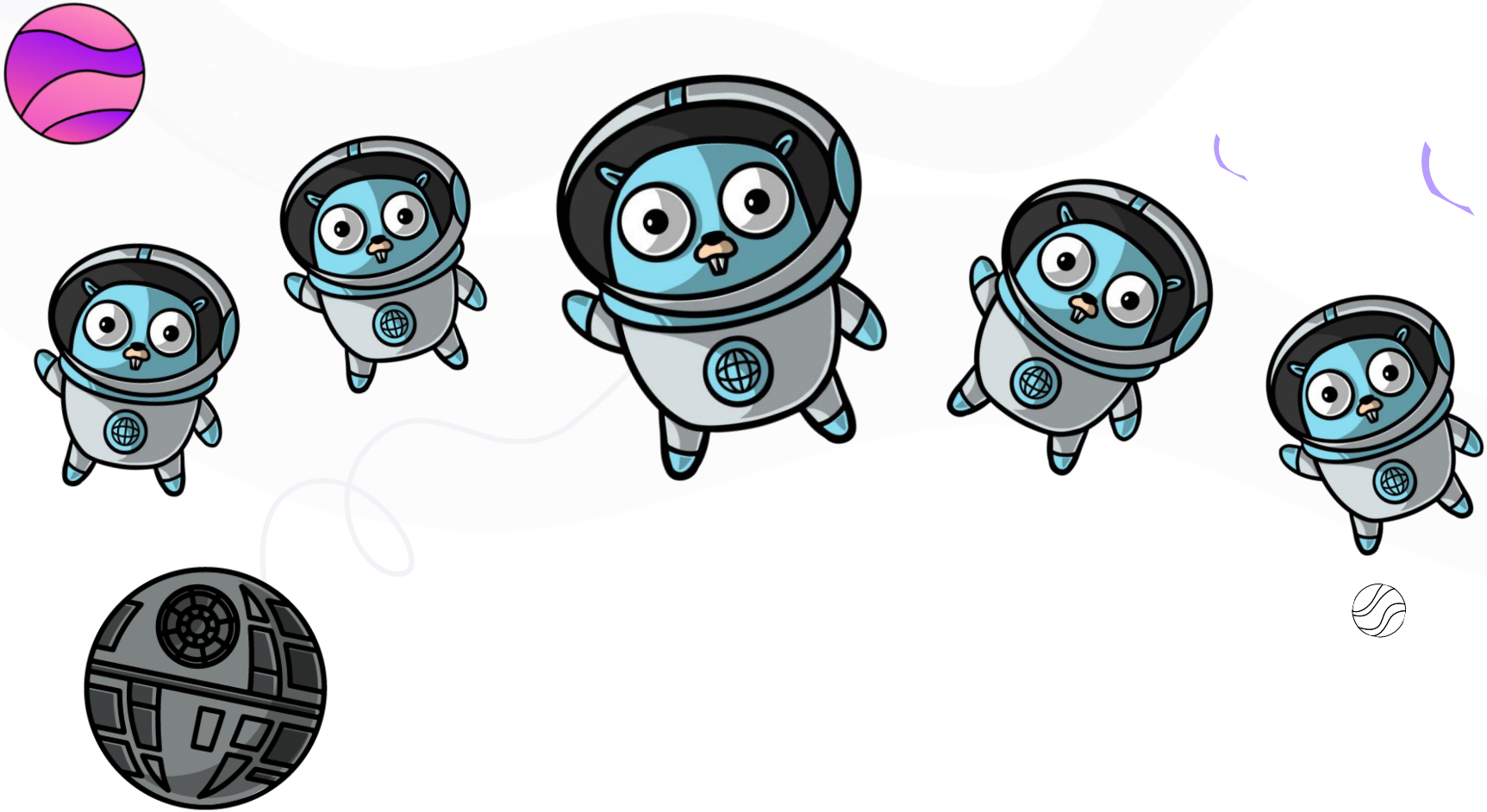


Agenda

Structs

- Structures fields

Methods

- Value vs Pointer receivers
- Method set

OOP in Golang

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

STRUCTURES

Structures - basics

```go
package main

import "fmt"

type Point [2]int

func main() {
    var p Point = [2]int{1, 2}

    fmt.Println(p)
}
```

On earlier lessons we already spoke about ***type*** keyword, this keyword allow to assign custom name for built-in type.

https://go.dev/play/p/txb14VKzlOP

[1 2]

Structures - basics

```go
package main

import "fmt"

type Point struct {
    x, y int
}

func main() {
    var p Point = Point{x: 1, y: 2}

    fmt.Println(p)
}

```

In Golang ***structures*** used to group data that semantically close.

https://go.dev/play/p/k32Q-OjbjWc

{1 2}

Structures - basics

```go
package main

import "fmt"

type Point struct {
    x1, x2 int
}

func main() {
    var p Point = Point{x1: 1, x2: 2}

    fmt.Println(p)
}

```

The ***type*** keyword allow to create structure with custom name.

https://go.dev/play/p/g6Fh6TjRL2P

{1 2}

Structures – anonymous

```go
package main

import "fmt"

type User struct {
    Username     string
    UserContacts struct {
        Phone string
        Email string
    }
}

func main() {
    user := User{
        Username: "Alex",
        UserContacts: struct {
            Phone string
            Email string
        }{
            Phone: "+0000000000000",
            Email: "test@gmail.com",
        },
    }

    fmt.Println(user)
}

```

Structures defined without ***type*** keyword called anonymous, such structures might define structure for some structure field or define structure for variable value.

https://go.dev/play/p/TzlfYLd_rmm

{Alex {+0000000000000 test@gmail.com}}

Structures - anonymous

```go
package main

import "fmt"

type User struct {
    Username string `json:"username"`
}

func main() {
    users := struct {
        Users []User `json:"users"`
    }{
        Users: []User{
            {Username: "Alex"},
            {Username: "Piotr"},
            {Username: "Pavel"},
        },
    }

    fmt.Println(users)
}

```

Structures defined without ***type*** keyword called anonymous, such structures might define structure for some structure field or define structure for variable value.

https://go.dev/play/p/s1DwYOK_8rl

{[{Alex} {Piotr} {Pavel}]}

Structures - initialization

```
1  package main
2
3  import "fmt"
4
5  type User struct {
6      Username string
7      Age      int
8  }
9
10 func NewUser(username string, age int) User {
11     return User{Username: username, Age: age}
12 }
13
14 func main() {
15     user1 := User{Username: "Alex", Age: 25}
16
17     user2 := User{"Sergey", 30}
18
19     user3 := NewUser("Pavel", 21)
20
21     fmt.Println(user1, user2, user3)
22 }
23
```

We have a few ways how to declare structure:

Line 15 – basic example

Line 17 – without field names

Line 19 – constructor function

https://go.dev/play/p/Ig013SlJq5Z

{Alex 25} {Sergey 30} {Pavel 21}

Structures - Pointers

```go
package main

import "fmt"

type Point struct {
    x, y int
}

func main() {
    var p Point = Point{x: 1, y: 2}

    fmt.Println(p)
    moveUp(p)
    fmt.Println(p)
}

func moveUp(p Point) {
    p.y = p.y + 1
}

```

So, as we see by default structures passed by value, not by reference. To fix that we can get a pointer to the variable using ***"&"***.

https://go.dev/play/p/er1JzG8gAQj

{1 2}

{1 2}

Structures - Pointers

```go
package main

import "fmt"

type Point struct {
    x, y int
}

func main() {
    var p Point = Point{x: 1, y: 2}

    fmt.Println(p)
    moveUp(&p)
    fmt.Println(p)
}

func moveUp(p *Point) {
    p.y = p.y + 1
}

```

Point type and reference to Point has different types, if variable is a pointer we should add ***”*”*** before type declaration to explicitly mention it.

https://go.dev/play/p/Mr744Or1WLp

{1 2}

{1 3}

Structures - Pointers

- Structs is a way how we group data that semantically close to each other or used in the same context.
- With help from ***type*** keyword, we can create structure with custom name.
- Structs might be anonymous, that is rarely used approach, but sometimes it might be very useful.
- There a few ways of struct initialization, the most popular way is to define constructor functions with preferred fields, or use full syntax, short syntax for type declaration is very fragile, please avoid it in most cases.

METHODS

Methods - basics

```go
package main

import (
    "fmt"
)

type Point struct {
    x, y int
    link *Point
}

func Link(p1, p2 *Point) {
    p1.link = p2
    p2.link = p1
}

func main() {
    p1 := Point{x: 5, y: 6}
    p2 := Point{x: 6, y: 5}

    Link(&p1, &p2)

    fmt.Println(p1, *p1.link)
    fmt.Println(p2, *p2.link)
}
```

We already know functions, and as we remember functions help us to group some instructions together and alias this group a name.

https://go.dev/play/p/W2E3dZGAAPS

```
{5 6 0xc00000c030} {6 5 0xc00000c018}
{6 5 0xc00000c018} {5 6 0xc00000c030}
```

Methods - basics

```
package main

type Point struct {
    x, y int
    link *Point
}

func Link(p1, p2 *Point) {
    p1.link = p2
    p2.link = p1
}

func main() {
    p1 := Point{x: 5, y: 6}
    p2 := Point{x: 6, y: 5}

    Link(&p1, &p2)

    fmt.Println(p1, *p1.link)
    fmt.Println(p2, *p2.link)
}
```

Function Link accept two pointers to pointer and link them, between them selves.

Methods - basics

```
package main

type Point struct {
    x, y int
    link *Point
}

func (p1 *Point) Link(p2 *Point) {
    p1.link = p2
    p2.link = p1
}

func main() {
    p1 := Point{x: 5, y: 6}
    p2 := Point{x: 6, y: 5}

    p1.Link(&p2)

    fmt.Println(p1, *p1.link)
    fmt.Println(p2, *p2.link)
}
```

Method is the same as the function but with a special part, receiver.

(p1 *Point)

There is two different types of methods, with value and with pointer receiver.

Receiver is context variable for the method, kind of implicit parameter for the function.

To call method as well as with fields, dot notation must be used.

Methods - basics

```
package main

type Point struct {
    x, y int
    link *Point
}

func Link(p1, p2 *Point) {
    p1.link = p2
    p2.link = p1
}

func (p1 *Point) Link(p2 *Point) {
    p1.link = p2
    p2.link = p1
}
```

Let's look at this two examples.

`func Link(p1, p2 *Point)`

In first example Link is a function that accept two parameters, both parameters is the pointer to instances of the Pointer type.

`func (p1 *Point) Link(p2 *Point)`

In second example instead of using two parameters of the Point type, Link become method of the type Point and on of the parameters, just become method receiver.

Methods - basics

```go
package main

import (
    "fmt"
)

type Point struct {
    x, y int
    link *Point
}

func (p1 *Point) Link(p2 *Point) {
    p1.link = p2
    p2.link = p1
}

func main() {
    p1 := Point{x: 5, y: 6}
    p2 := Point{x: 6, y: 5}

    p1.Link(&p2)

    fmt.Println(p1, *p1.link)
    fmt.Println(p2, *p2.link)
}
```

https://play.golang.org/p/xYz1y_I3cdC

```
{5 6 0xc00000c030} {6 5 0xc00000c018}
{6 5 0xc00000c018} {5 6 0xc00000c030}
```

Methods - basics

```go
package main

import (
    "fmt"
    "math"
)

type Point struct {
    x, y int
    link *Point
}

func (p1 Point) Distance(p2 Point) float64 {
    xDiff := math.Pow(float64(p1.x - p2.x), 2)
    if p2.x > p1.x {
        xDiff = math.Pow(float64(p2.x - p1.x), 2)
    }

    yDiff := math.Pow(float64(p1.y - p2.y), 2)
    if p2.y > p1.y {
        yDiff = math.Pow(float64(p2.y - p1.y), 2)
    }

    return math.Sqrt(xDiff + yDiff)
}

func main() {
    p1 := Point{x: 3, y: 2}
    p2 := Point{x: 9, y: 7}

    fmt.Printf("%.3f\n", p1.Distance(p2))
}
```

Methods have pointer or value receiver, in code on the left side, used value receiver, in previous example pointer receiver.

Value receiver – work in the same way as any type, before passing receiver to the method, structure will be copied.

(p1 Point) – *receiver example*

https://go.dev/play/p/RaIkXQjRbL0

7.810

Methods - basics

Value receiver

- Make a shallow copy of the object each time method is called
- All mutates of the object done inside of the method will disappear after method exit
- Used for methods that do some computations or just return the data.

Pointer receiver

- Pass pointer for each method call
- All mutates of the object done inside of the method will there after method exit
- Used for methods that mutate internal state of the object, in case make a copy of the data isn’t an option, in case make a copy of the data is to heavy.

INTERFACES

Interface

```
type Serializable interface {
    Marshal() ([]byte, error)
    Unmarshal([]byte) error
}
```

Interface is just a set of methods, all methods inside of interface should have unique name.

https://go.dev/play/p/SRPx51gvF9f

Interface

```go
type Serializable interface {
    Marshal() ([]byte, error)
    Unmarshal([]byte) error
}

type User struct {
    Username string `json:"username"`
}

func (u User) Marshal() ([]byte, error) {
    return json.Marshal(u)
}

func (u *User) Unmarshal(raw []byte) error {
    return json.Unmarshal(raw, u)
}
```

To implement interface receiver must have all the methods defined in interface, and it all.

https://go.dev/play/p/SRPx51gvF9f

Interface

```go
type Serializable interface {
    Marshal() ([]byte, error)
    Unmarshal([]byte) error
}

type User struct {
    Username string `json:"username"`
}

func (u User) Marshal() ([]byte, error) {
    return json.Marshal(u)
}

func (u *User) Unmarshal(raw []byte) error {
    return json.Unmarshal(raw, u)
}
```

To implement interface receiver must have all the methods defined in interface, and it all.

https://go.dev/play/p/SRPx51gvF9f

Interface

```go
package main

import (
    "encoding/json"
    "fmt"
)

type Serializable interface {
    Marshal() ([]byte, error)
    Unmarshal([]byte) error
}

type User struct {
    Username string `json:"username"`
}

func (u User) Marshal() ([]byte, error) {
    return json.Marshal(u)
}

func (u *User) Unmarshal(raw []byte) error {
    return json.Unmarshal(raw, u)
}

func main() {
    user1 := User{Username: "Aliaksei"}
    raw, err := user1.Marshal()
    if err != nil {
        panic(err)
    }
    user2 := User{}
    if err := user2.Unmarshal(raw); err != nil {
        panic(err)
    }

    fmt.Println(user1, user2)
}
```

https://go.dev/play/p/SRPx51gvF9f

{Aliaksei} {Aliaksei}

Methods set

```go
package main

import (
    "fmt"
)

type Stringer interface {
    ToString() string
}

type Point struct {
    x, y int
}

func (p1 Point) ToString() string {
    return fmt.Sprintf("X is %d, Y is %d", p1.x, p1.y)
}

func print(s Stringer) {
    fmt.Println(s.ToString())
}

func main() {
    p1 := Point{}
    p2 := &Point{}

    print(p1)
    print(p2)
}

```

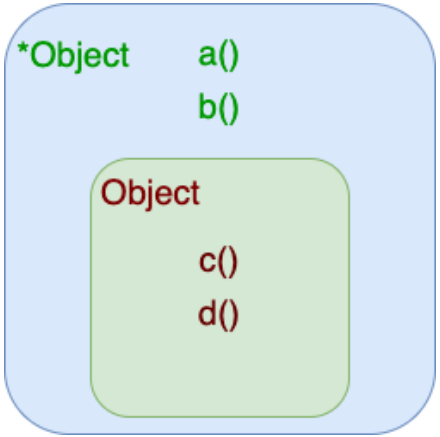
But safely can call value receiver methods on pointers.

https://go.dev/play/p/gMAkQYbHA5C

```
X is 0, Y is 0
X is 0, Y is 0
```

Methods set

```
type Object struct{}
func (o *Object) a() {}
func (o *Object) b() {}
func (o Object) c() {}
func (o Object) d() {}
```



Every type in Go has method set, method set is sequence of methods defined for the type.

Every method for the type belongs to value type method set or pointer type method set.

There one important thing, pointer receiver embed all methods from value method set.

In a few words, on a variable with type `*Object`, methods `a`, `b`, `c`, `d` might be called. On a variable with type `Object`, only methods `c`, `d` might be called.

Methods set

```go
package main

import (
    "fmt"
)

type Stringer interface {
    ToString() string
}

type Point struct {
    x, y int
}

func (p1 *Point) ToString() string {
    return fmt.Sprintf("X is %d, Y is %d", p1.x, p1.y)
}

func print(s Stringer) {
    fmt.Println(s.ToString())
}

func main() {
    p1 := Point{}
    p2 := &Point{}

    print(p1)
    print(p2)
}
```

./prog.go:27:10: cannot use p1 (type Point) as type Stringer in argument to print:

Point does not implement Stringer (ToString method has pointer receiver)%

https://go.dev/play/p/nLC5r1pd887

You cannot call pointer methods on value receiver

Methods set

```go
package main

import (
    "fmt"
)

type Stringer interface {
    ToString() string
}

type Point struct {
    x, y int
}

func (p1 Point) ToString() string {
    return fmt.Sprintf("X is %d, Y is %d", p1.x, p1.y)
}

func print(s Stringer) {
    fmt.Println(s.ToString())
}

func main() {
    p1 := Point{}
    p2 := &Point{}

    print(p1)
    print(p2)
}

```

But safely can call value receiver methods on pointers.

https://go.dev/play/p/gMAkQYbHA5C

```
X is 0, Y is 0
X is 0, Y is 0
```

OOP in Golang

Encapsulation

In Go we have simplified model of encapsulation, main difference.

- Encapsulation works only on a package level, not a structure.
- There only two access levels, ***public*** and ***private.***
- *Public and private modifiers configured via **first letter** of the **name**.*

Encapsulation

Public

```go
package main

var PI = 3.14

type Computator interface {}

type DistributedComputator struct {
    External string
}

func (c DistributedComputator) Compute() (error) {return nil}

```

All that start from upper-case letter available to use by other packages.

Private

```go
package main

var pi = 3.14

type computator interface {}

type distributedComputator struct {
    internal string
}

func (c distributedComputator) compute() (error) {return nil}

```

All that start from lower-case letter not available to use by other package.

Inheritance

In Go we don’t have inheritance in general, but in Go we have composition with some special syntax sugar

Whereas **inheritance** derives one class from another, **composition** defines a class as the sum of its parts

- In Go each structure can be embedded into another structure
- Methods/fields from embedded structure can be used from the parent structure via the same receiver
- In Golang we don’t have neither Methods overloads neither Methods overrides.

Inheritance

```
type Cat struct{}


type OktoCat struct {
    Cat
}
```

To embed structure into another we just put structure without name, like an anonymous type.

https://go.dev/play/p/K6cbpgjp2_D

Inheritance

```go
package main

import (
    "fmt"
)

type Cat struct{}

func (Cat) Legs() int {
    return 4
}

func (c Cat) PrintLegs() {
    fmt.Printf("I'm a cat with %d legs\n", c.Legs())
}

type OktoCat struct {
    Cat
}

func main() {
    c := Cat{}
    c.PrintLegs()

    oc := OktoCat{}
    oc.PrintLegs()

}

```

Parent struct inherit all the fields and Methods from the embedded receiver.

https://go.dev/play/p/K6cbpgjp2_D

I'm a cat with 4 legs

I'm a cat with 4 legs

Inheritance

```go
package main

import (
    "fmt"
)

type Cat struct{}

func (Cat) Legs() int {
    return 4
}

func (c Cat) PrintLegs() {
    fmt.Printf("I'm a cat with %d legs\n", c.Legs())
}

type OktoCat struct {
    Cat
}

func (OktoCat) Legs() int {
    return 8
}

func main() {
    c := Cat{}
    c.PrintLegs()

    oc := OktoCat{}
    oc.PrintLegs()

}
```

Methods cannot be overrided

https://go.dev/play/p/i7NkoQy_LU0

I'm a cat with 4 legs

I'm a cat with 4 legs

Polymorphism

In Go Polymorphism represented by interfaces.

In Go you will often listen such theory as Duck typing.

Duck typing – **If it walks like a duck and it quacks like a duck, then it must be a duck.**

- Structure implement interface, if it has all of the methods defined in interface
- If we dive deeper, we will see that not structs implement interfaces, but struct receivers (pointer or value one)
- All interfaces implemented by value receiver, can be used via pointer receiver.

Polymorphism

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

type MyInt int

func (i MyInt) ToString() string {
    return strconv.Itoa(int(i))
}

type MyBool bool

func (i MyBool) ToString() string {
    if i == MyBool(true) {
        return "true"
    }

    return "false"
}

type User struct {
    Username string
    Age      int
}

func (u User) ToString() string {
    return u.Username + " " + strconv.Itoa(u.Age)
}
```

https://go.dev/play/p/bMt8FSPhs4Z

Let's create interface ***Stringer*** with one method ***ToString***.

*Structures **MyInt**, **MyBool** and **User** are* implementing interface Stringer.

Polymorphism

https://go.dev/play/p/bMt8FSPhs4Z

Function ***println*** don't care about type, if type passed have method ***ToString*** It is enough.

```
func println(elems ...Stringer) {
    for _, elem := range elems {
        fmt.Println(elem)
    }
}
```

Polymorphism

https://go.dev/play/p/bMt8FSPhs4Z

In ***main*** function let's create a bunch of ***Stringers***, and call ***println*** function.

```
func main() {
    printable := []Stringer{
        MyInt(5),
        MyBool(false),
        User{Username: "Aliaksei", Age: 25},
    }

    for _, p := range printable {
        println(p)
    }
}
```

5

False

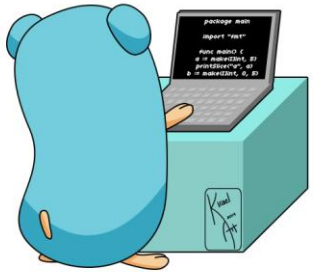
{Aliaksei 25}

QA

HOMEWORK

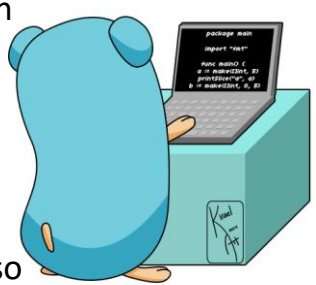
Homework

For the given structure of a Square, create the methods to return end point, area and perimeter using given signatures.
NOTE: receiver is a placeholder, replace appropriately



Homework

Implement a simple cache, that holds values with type string and allows to retrieve them using keys of time stings. Key/value pairs can expire if given a deadline using an appropriate method.
The initial structure is given. You should add appropriate fields, implement the constructor function and methods with given signatures.
k, ok := Get(key string) - returns the value associated with the key and the boolean ok (true if exists, false if not), if the deadline of the key/value pair has not been exceeded yet
Put(key string, value string) places a value with an associated key into cache. Value put with this method never expired(have infinite deadline). Putting into the existing key should overwrite the value
 * PutTill(key string, value string, deadline time.Time) Should do the same as Put, but the key/value pair should expire by given deadline
Keys() []string should return the slice of existing (non-expired keys)
The expiration should, and probably cannot in our case, be implemented asynchronously, so think of a way to clean up the records when needed
NOTE: receiver is a placeholder, replace appropriately Also, the placeholder structures can be found in solution branch



THANK YOU