

Объектная модель в Go



План занятия

- Структуры
- Интерфейсы
- Обработка ошибок
- Работа с окружением



Структуры

Структура – агрегированный тип данных, объединяющий несколько значений произвольных типов

```
1 type Exchange struct {
2     Name      string
3     StartTime time.Time
4     EndTime   time.Time
5     stocks    []string // это поле не может быть использовано в других пакетах
6 }
```

Объявление структуры

- Объявить все поля
- Объявить часть полей
- Без объявления полей
- Все поля, без имен (строгий порядок, не рекомендуется)

```
1 type Candle struct {
2     name      string
3     openPrice float64
4     closePrice float64
5 }
6
7 func main() {
8     c1 := Candle{name: "BABA", openPrice: 1.0, closePrice: 1.5}
9     fmt.Println(c1)
10
11    c2 := Candle{name: "SPCE", openPrice: 0.55}
12    fmt.Println(c2)
13
14    c3 := Candle{}
15    fmt.Println(c3)
16
17    c4 := Candle{"AMZN", 1.0, 2.0}
18    fmt.Println(c4)
19 }
```

```
{BABA 1 1.5}
{SPCE 0.55 0}
{ 0 0}
{AMZN 1 2}
```

Пустая структура

```
1 type a struct{}
```

```
2
```

```
3 empty := struct{}{}
```

```
1 fmt.Println(unsafe.Sizeof(struct{}{}))
```

```
2 fmt.Println(unsafe.Sizeof([1000]struct{}{}))
```

```
3 fmt.Println(unsafe.Sizeof([1000]*int{}))
```

```
0
```

```
0
```

```
8000
```

- Значение в map
- Объект для семафоров

Конструктор

- Скорее не нужен, если zero-value рабочий
- Если очень хочется – можно использовать new(T)
- Явное задание параметров
- Правильная инициализация



Конструктор

```
1 package exchange
2
3 type Exchange struct {
4     Name      string
5     StartTime time.Time
6     EndTime   time.Time
7
8     duration time.Duration
9 }
10
11 func New(name string, sTime, eTime time.Time) (Exchange, error) {
12     if eTime.Before(sTime) {
13         return Exchange{}, errors.New("end time can't be before start
14             time")
15     }
16     return Exchange{
17         Name:      name,
18         StartTime: sTime,
19         EndTime:   eTime,
20         duration:  eTime.Sub(sTime),
21     }, nil
21 }
```

Методы

```
1 func (e Exchange) Duration() time.Duration {
2     return e.duration
3 }
4
5 func Duration(e Exchange) time.Duration {
6     return e.duration
7 }
```

- Выделяют функции для отдельного типа
- Удобно использовать

Встраивание

В Go нет наследования, но есть встраивание (embedding).

Встраивание позволяет использовать поля и методы другой структуры.

```
1 type stock struct {
2     stockType string
3     exchange.Exchange
4 }
5
6 fmt.Printf(s.stockType, s.Name, s.Duration())
```

Value vs Pointer

Используй указатель, если подразумевается изменение внутреннего состояния

```
1 func (e Exchange) Duration() time.Duration {
2     return e.duration
3 }
4
5 func (e *Exchange) UpdateEndTime(t time.Time) {
6     e.EndTime = t
7     e.duration = e.EndTime.Sub(e.StartTime)
8 }
```

Value vs Pointer

Используй **value** для:

- Стандартных типов (int, string, bool, ...)
- Ссылочных типов (slice, chan, map, interface, func) – они уже содержат указатель внутри себя
- Где допустимо копирование данных

Используй **pointer** для:

- Изменение состояния
- Конструктор вернул указатель
- Не уверен, что можно копировать



Полиморфизм

Есть необходимость писать код, который будет менять свое поведение, в зависимости от типа

Go поддерживает концепцию «утиной» типизации – если это выглядит как утка, плавает как утка и крякает как утка, то это, вероятно, и есть утка.



Интерфейс

```
1 type CandleFetcher interface{
2     Fetch() (Candle, error)
3 }
4
5 type MOEXFetcher struct{}
6
7 func (m MOEXFetcher) Fetch() (Candle, error) {
8     fmt.Println("received candle from MOEX")
9     return Candle{}, nil
10 }
11
12 type SPBFetcher struct{}
13
14 func (s SPBFetcher) Fetch() (Candle, error) {
15     fmt.Println("received candle from SPB")
16     return Candle{}, nil
17 }
```

- Чтобы удовлетворять интерфейсу, типу надо реализовать все методы интерфейса
- Не важна реализация, важен результат
- Удобно для «сходных» функционалов, тестов

Конвенция имен

```
1 type Reader interface {
2     Read(p []byte) (n int, err error)
3 }
4
5 type Writer interface {
6     Write(p []byte) (n int, err error)
7 }
8
9 type Stringer interface {
10    String() string
11 }
```

Композиция

Интерфейс должен быть максимально простым и содержать **минимум функционала**.

Для расширения функционала используется композиция.

```
1 type ReadWriter interface {
2     Reader
3     Writer
4 }
```

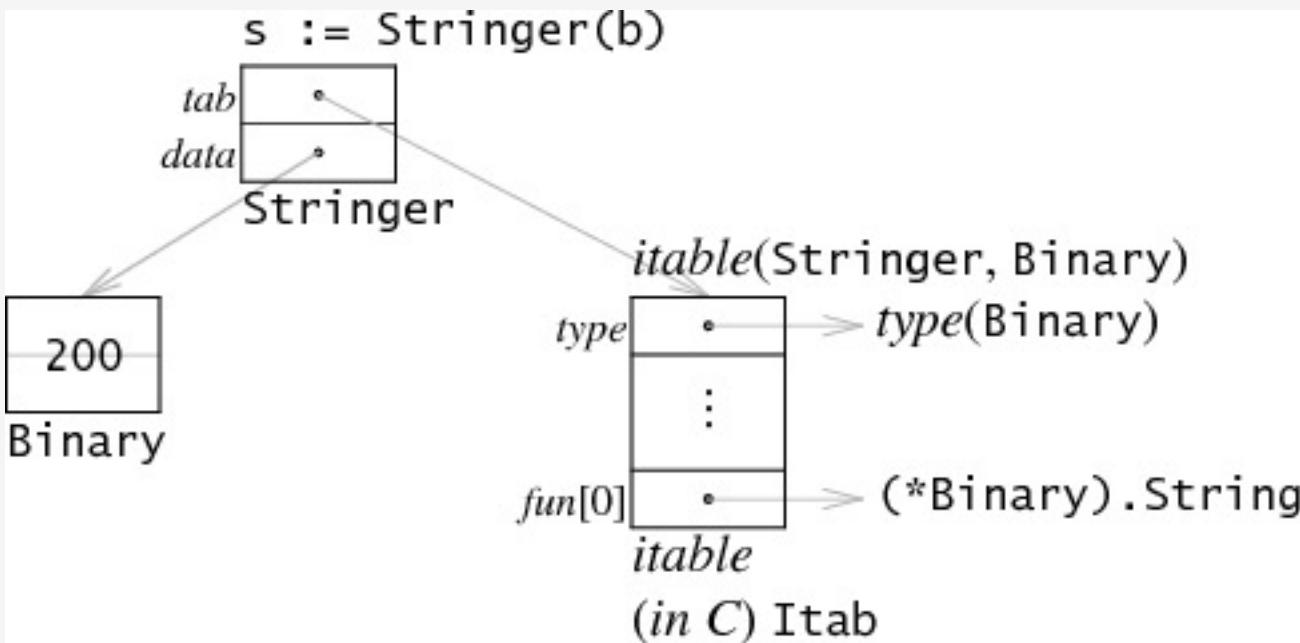
Композиция является более гибким инструментом расширения проекта (чем наследование)

Соответствие интерфейсу

Можно легко проверить на этапе компиляции соответствие типа интерфейсу.

```
1 var _ io.Reader = new(bytes.Buffer)  
2 var _ io.Reader = "abc" // ошибка компиляции
```

Строение интерфейса



Ассоциация типов

Каждый тип ассоциирован с своим множеством методов

Methods	Receivers	Values

(t T)		T and *T
(t *T)		*T

Пустой интерфейс

```
1  var b interface{}
2
3  b = "Hello"
4  _ = interface{}{"Hello"}
5  fmt.Printf("%T, %+v\n", b, b)
6
7  b = []int{1,2,3}
8  _ = interface{}{[]int{1,2,3}}
9  fmt.Printf("%T, %+v\n", b, b)
10
11 b = func(){fmt.Println("Hi, Mark!")}
12 _ = interface{}(func(){fmt.Println("Hi, Mark!")})
13 fmt.Printf("%T, %+v\n", b, b)
```

```
string, Hello
[]int, [1 2 3]
func(), 0x10a4d60
```

- Интерфейс не содержит методов
- Является попыткой реализовать динамическую типизацию в языке со строгого статической типизацией
- Любое значение можно привести к пустому интерфейсу

Приведение типов

```
1 var b interface{}
2
3 b = "Hi, Mark!" // b still has type interface{}
4
5 s, ok := b.(string)
6 if ok {
7     // s is string
8 } else {
9     // b was not string, s has zero value
10 }
```

```
1 b = exchange.Exchange{Name: "MOEX"} // b still has type interface{}
2 if e, ok := b.(exchange.Exchange); ok {
3     fmt.Println(e.Name)
4 }
```

```
1 b = new(bytes.Buffer) // b still has type interface{}
2 if r, ok := b.(io.Reader); ok {
3     _, err = r.Read(...)
4 }
```

Допускается приведение к:

- Стандартным типам
- Кастомным типам
- Другим интерфейсам

Приведение типов

```
1 switch b.(type){  
2     case string: // default types  
3         fmt.Println("b is string")  
4         _ = b.(string)  
5     case io.Reader: // interfaces  
6         fmt.Println("b is Reader")  
7         _ = b.(io.Reader)  
8     case error: // ???  
9         fmt.Println("b is error")  
10        _ = b.(error)  
11    }
```

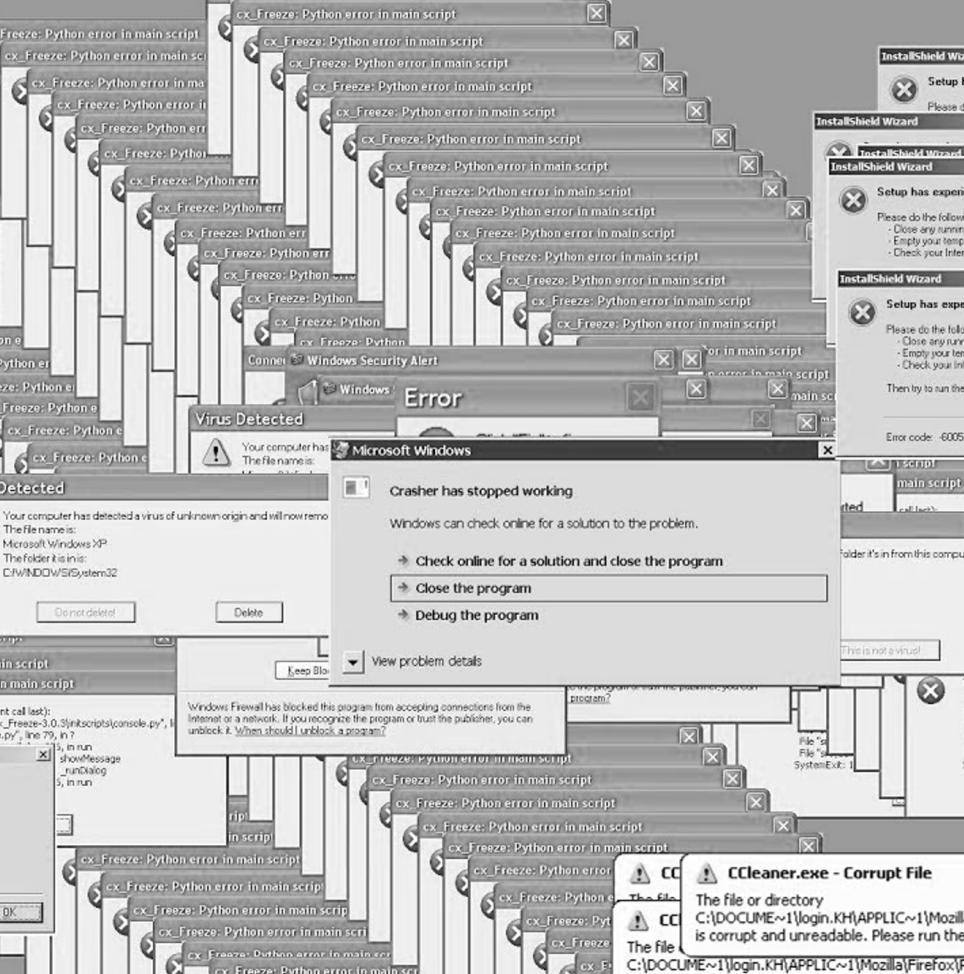
- По возможности избегайте работы с пустым интерфейсом
- Всегда делайте безопасное приведение типов
- ???
- Profit!

error

```
1 type error interface {
2     Error() string
3 }
```

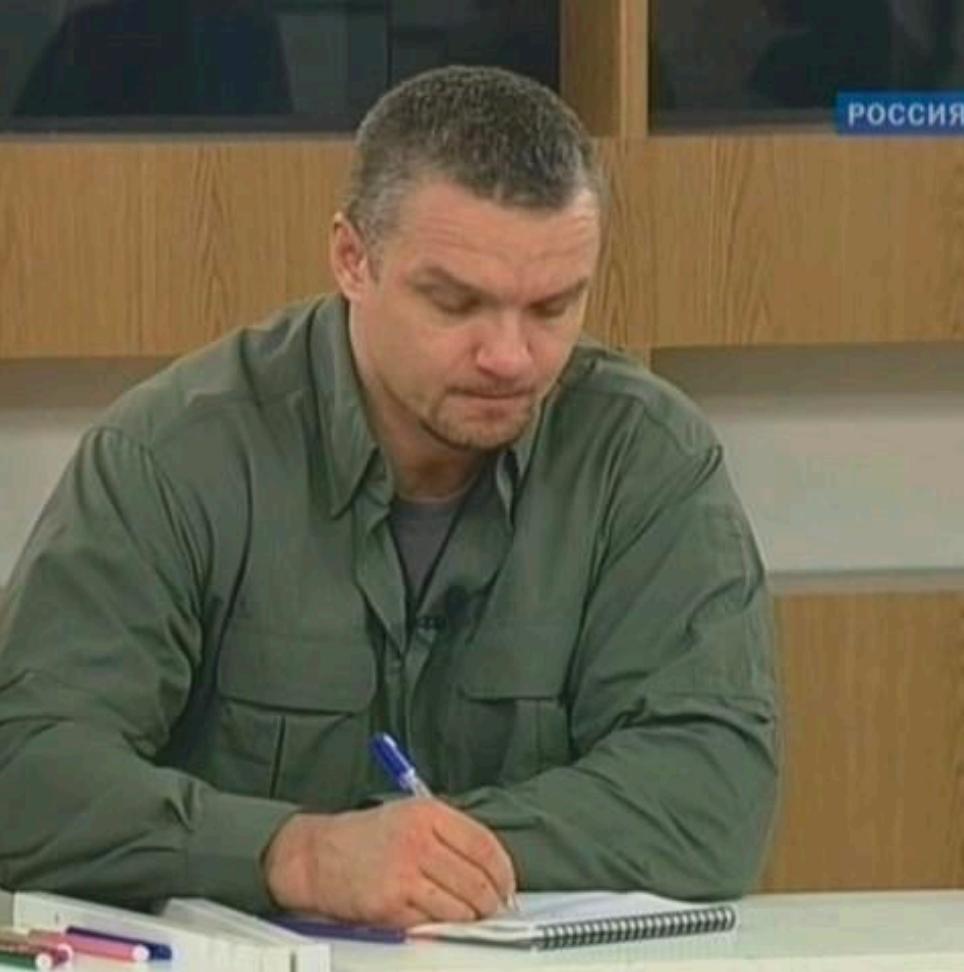
Реализовывать интерфейс может любой тип, у которого объявлен метод Error() string

```
1 // A SyntaxError is a description of a JSON
   syntax error.
2 type SyntaxError struct {
3     msg      string // description of error
4     Offset int64   // error occurred after
   reading Offset bytes
5 }
6
7 func (e *SyntaxError) Error() string {
8     return e.msg
9 }
```



Обработка ошибок

- Если функция возвращает **объект и ошибку**, пусть возвращает *object, nil* в случае успеха и *nil, err* в случае ошибки (для слайса можно *nil, nil*)
- Обрабатывай ошибки явно – никаких `checkErr()`
- Добавляй контекст в ошибки



errors.Is errors.As

```
1 var ErrNotFound = CustomErr{Msg: "not found", Code: 404}
2
3 err := fmt.Errorf("opening file error: %w", ErrNotFound)
4 _, ok := err.(CustomErr) // false
5
6 ok = errors.Is(err, ErrNotFound)
7 fmt.Println(ok) // true
8
9 var ce CustomErr
10 ok = errors.As(err, &ce) // true
11 fmt.Printf("%t, %v, %v", ok, ce.Msg, ce.Code)
```

Конфигурация приложения

- Пользовательский ввод
- Переменные окружения
- Флаги
- Файлы конфигурации



User input

```
Enter login:  
admin  
Enter password:
```

```
fmt.Println("Enter login:")  
var login string  
n, err = fmt.Scan(&login)
```

```
fmt.Println("Enter animal and random integer:")  
var (  
    animal string  
    number int  
)  
n, err := fmt.Scanf("%s %d", &animal, &number)
```

ENV

```
1 var verbose = "VERBOSE"
2
3 s, ok := os.LookupEnv(verbose)
4
5 s = os.Getenv(verbose)
6
7 all := os.Environ()
```

```
// linux, mac
VERBOSE=true ENV2='text' ENV3=33 executable-name

// windows
set VERBOSE=true
executable-name.exe
```

Переменные окружения – всегда строки.

Для числовых, булевых и других типов нужна дополнительная конвертация

Flags

```
// pointer to existing var  
var verbose bool  
flag.BoolVar(&verbose, "v", false, "add expanded logs")  
  
// new variable  
var configFilePath = flag.String("config", "default_value", "path to config  
file")  
  
// don't forget to  
flag.Parse()
```

```
// bool flags can be assigned or just defined  
executable -v -file=true --path='/opt/example/config.json'
```

Files

```
f2, err := os.Open("input2.txt")

// *os.File implements io.Reader
data, err := ioutil.ReadAll(f2)
f2.Close()
```

```
f, err := os.Create("joke.txt")
defer f.Close()

// *os.File implements io.Writer
_, err = f.WriteString("text")
_, err = f.Write([]byte("text"))
_, err = fmt.Fprint(f, "text")
```



Сериализация

Сериализация — процесс перевода структуры данных в последовательность байтов. Обратной к операции сериализации является операция десериализации (структуризации) — создание структуры данных из битовой последовательности.

Популярные человекочитаемые форматы

- JSON
- YAML
- XML

Популярные бинарные форматы

- protobuf
- BSON
- [тысячи их...](#)

JSON (encoding/json)

```
1 type User struct {
2     Name    string
3     Active  bool
4     age     uint
5 }
6
7 data, err := json.Marshal(User{
8     Name: "Shamil",
9     Active: false,
10    age: 30,
11 })
12 fmt.Println(string(data))
```

```
{"Name": "Shamil", "Active": false}
```



YAML (gopkg.in/yaml.v2)

```
1  type DB struct {
2      DBAddress string `yaml:"host"`
3      Username   string `yaml:"login"`
4      Password   string `yaml:"pwd"`
5  }
6
7  type Config struct {
8      Postgres DB `yaml:"postgres"`
9  }
10
11 data, err = yaml.Marshal(Config{
12     Postgres: DB{
13         DBAddress: "localhost:5432",
14         Username: "admin",
15         Password: "dnkroz",
16     },
17 })
18 fmt.Println(string(data))
```

```
postgres:
  host: localhost:5432
  login: admin
  pwd: dnkroz
```

Становление профессионалом

- Читай доки и исходники
- Исследуй мир вокруг себя
- Думай
- ...
- PROFIT



Итоги

Структуры

Основной кирпичик рабочего процесса

Интерфейсы

Реализация полиморфизма

Обработка ошибок

Как правильно

Конфигурация и файлы

Храним данные и настраиваем приложение

