Go

Начальный курс по Go

РАБОТА С ОШИБКАМИ

error

Интерфейс, объявленный в builtin.go:

```
type error interface {
   Error() string
}
```

errors

errors.New(message) создаёт stringError (см. errors.go)

errors

Ошибки возвращаются функцией как обычное значение (по соглашению – последним значением)

Общий формат (для main)

```
result, err := functionCall()

if err != nil {

   log.Fatal(err)

}
```

Печать ошибки + os.Exit(1)

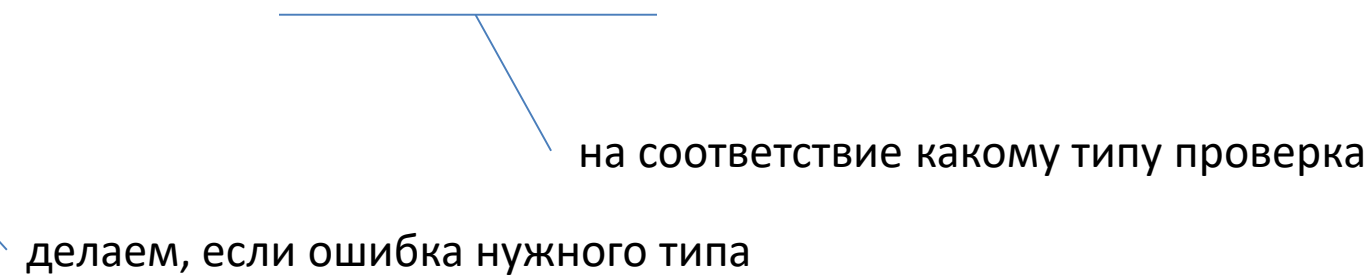
Custom Errors

```
type queryError struct {
  query string
  err error
}
func (e *queryError) Error() string {
  return e.query + ": " + e.err.Error()
}
```

В обработке ошибок

В обработке ошибок:

```
_, err := someCall()
if err != nil {
    if typedError, ok := err.(*queryError); ok {
        ...
    }
}
```



fmt.Errorf

Форматированная ошибка, позволяющая "форматировать" текст, так же, как Printf

Начиная с Go 1.13 появился модификатор %w, который позволяет заворачивать ошибку для дальнейших тестов

Общий формат (для не main)

```
result, err := doStuff(arg)

if err != nil {

   return fmt.Errorf("can't do stuff %v, %w", arg, err)

}
```

вернёт ошибку вызывающей функции
и завернёт исходную ошибку

Общий формат (для не main)

```
result, err := doStuff(arg)

if err != nil {

   return fmt.Errorf("can't do stuff %v, %v", arg, err)

}
```

вернёт ошибку вызывающей функции
и не завернёт исходную ошибку

Но чаще всего делают

```
result, err := doStuff(arg)
if err != nil {
    return
}
```

не делайте так

Надо обработать ошибку, а не просто "молчаливо"

возвращать управление вызывающей функции

Format Strings

- %d, %x, %o, %b – integer
- %f, %g, %e - floating point number
- %t – boolean
- %c – rune
- %s – string
- %v - any value in natural format
- %T - type of value
- %w - error wrapping (only for Errorf)

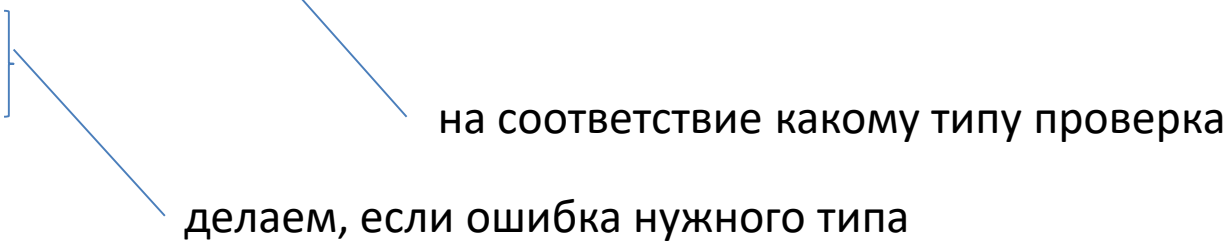
Format Strings

fmt.Printf, log.Printf и fmt.Errorf используют одни и те же (за исключением %w) format strings

go 1.13+

В обработке ошибок:

```
_, err := someCall()
if err != nil {
   var typedError *queryError
   if errors.As(err, &typedError) {
      ...
   }
}
```



go 1.13+

В обработке ошибок:

```
var specificError = errors.New("specific")
_, err := someCall()
if err != nil {
   if errors.Is(err, specificError) {
      ...
   }
}
```

выносят вне функций

на соответствие какой ошибке (где-то из wrapped)

делаем, если ok

Unwrap

Важно: Is и As работают только, если у ошибок (в которые завёрнуты нужные вам ошибки), реализован метод Unwrap

МЕТОДЫ

Классическое ООП

Объект – сущность, обладающая набором свойств и методов

Текущее значение всех свойств – состояние

Методы – операции, которые может выполнять объект

Методы в Go

Функции со специальным параметром, который называется receiver – "объект", на котором будет вызываться функция:

```
func (r Account) deposit(amount int) { ... }
```

receiver

Методы в Go

Теперь можно делать так:

```
account.deposit(1000)
```

receiver

метод

Методы в Go

Для Pointer Receiver'ов:

```
func (r *Account) deposit(amount int) { ... }
```

receiver

Но синтаксис вызова такой же:

```
account.deposit(1000)
```

receiver метод

Методы в Go

На самом деле, go сам сделает:

(&account).deposit(1000)

Правило

Если хотя бы один метод с Pointer Receiver'ом, то все должны быть объявлены с Pointer Receiver'ом

Правило

1. Если нужно менять receiver – то pointer
2. Если большая структура/тип данных – то pointer
3. Если хотя бы один pointer, то все для консистентности – pointer*

Примечание*: *T содержит все методы T, но не наоборот

Правило

```
type stuff int64
type valuer interface {
    value()
}
func (receiver *stuff) value() { }

func main() {
    value := stuff(1)
    i := valuer(&value)
    fmt.Println(i)
}
```

INTERFACES

Interfaces

Интерфейс – контракт на реализацию поведения

Должны быть методы определённого типа (с нужной сигнатурой)

Interfaces

В Go интерфейсы реализуются неявно: т.е. если нужные методы реализованы, то тип уже соответствует интерфейсу

Важно: интерфейс ничего не говорит о внутренней реализации методов

Interfaces

```
type printer interface {
  print()
}
```

Рассмотрение пакета sort

Практика на базе пакета sort

GoLand: Alt + Enter -> Implement Interface

interface{}

Возможность принимать любой тип данных

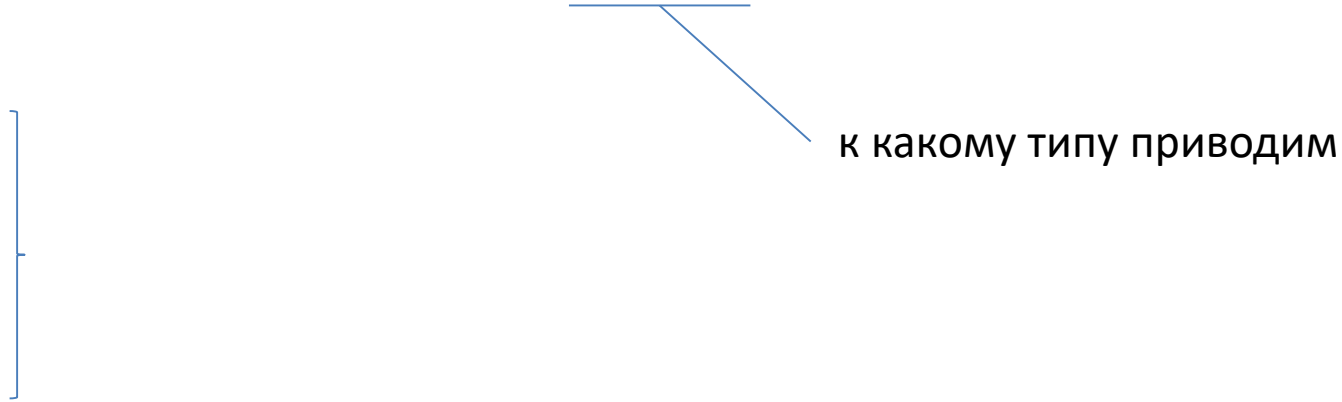
TYPE ASSERTIONS

Type Assertion

Type Assertion: попытка преобразовать "объект" из одного типа в другой:

```
converted, ok := original.(string)

if ok {
    ...
}
```



Type Assertion

Type Assertion: попытка преобразовать

"объект" из одного типа в другой:

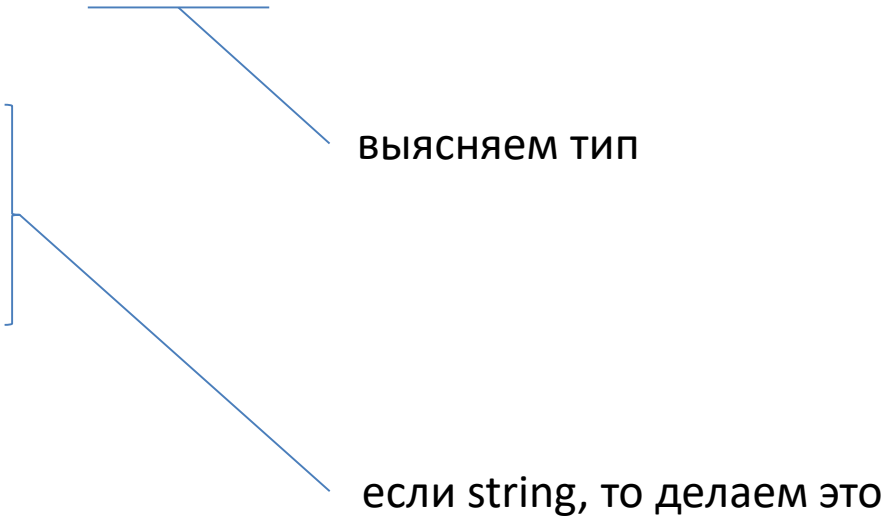
```
converted := original.(string)
```

Без ok может привести к panic

Type Switch

Проверка типа

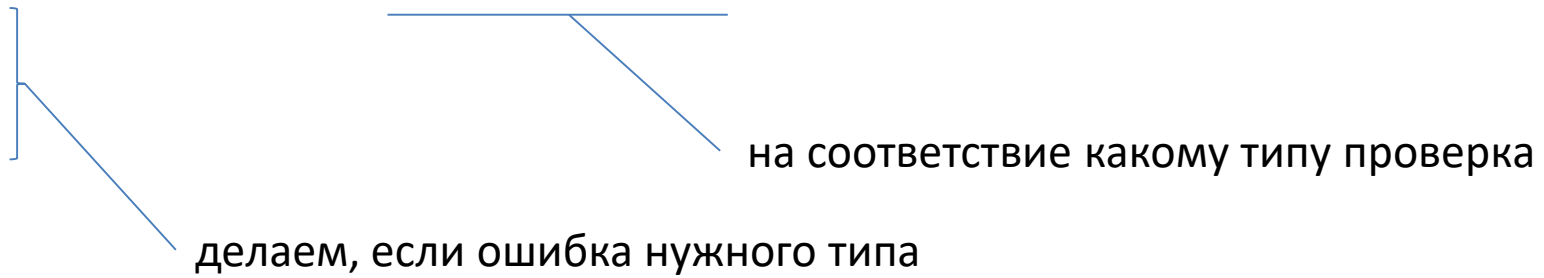
```
switch original.(type) {
case string:
  ...
case int:
  ...
```



В обработке ошибок

В обработке ошибок:

```
_, err := someCall()
if err != nil {
    if typedErr, ok := err.(*pkg.Error); ok {
        ...
    }
}
```



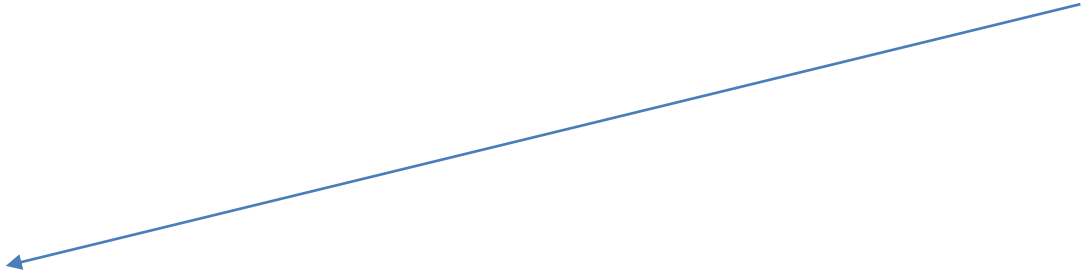
PANIC, DEFER, RECOVER

defer

Выполняет функцию после выхода из метода:

```
func demo() {
    defer func() {}()
}
```

Функция выполнится, когда мы будем выходить **отсюда**



panic

panic() - прерывает нормальный ход выполнения программы и завершает выполнение функции (но defer отрабатывает) и переходит вверх по стеку вызовов

panic

Для вызывающей функции (той, что вызвала функцию, в которой был panic) вместо результата тоже "подставляется panic", таким образом и она завершается

recover

И так, пока не встретится recover (recover обязательно должен быть в defer).

recover() возвращает аргумент, переданный в panic() и на этом "паника" прекращается - дальше можно обрабатывать как обычную ошибку

ДЗ

Queue

Реализовать на базе структур, методов и указателей тип очередь, которая поддерживает следующие операции:

- len() - получение длины (кол-во элементов)
- first() - указатель на первый элемент
- last() - указатель на последний элемент
- equeue() - добавление элемента в конец очереди
- dequeue() - извлечение из начала очереди

PriorityQueue

Добавить в предыдущую очередь приоритезацию (т.е. выстраивание элементов не по времени добавления, а сначала по приоритету, а потом по времени добавления)

Спасибо за внимание

Ильназ Гильязов

2020г.