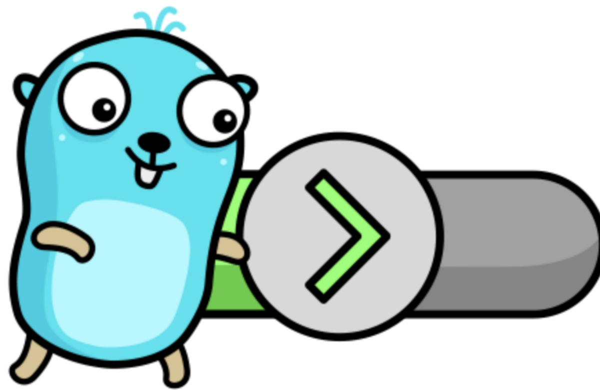
Введение в интерфейсы в Go

В этом уроке мы поверхностно коснёмся понятия интерфейса в Go и его внутреннего устройства, а также ответим на вопрос, зачем вообще нужны интерфейсы?

Невозможно начать рассказывать об ошибках в языке, удовлетворяющих в свою очередь **встроенному интерфейсу** `error` (забегая вперёд), не коснувшись данного понятия в принципе.



Интерфейсы в Go

[Интерфейсы](#) в Go одновременно похожи и не похожи на интерфейсы в других широко распространенных языках программирования.

Пример объявления интерфейса в Kotlin:

```
inteface Stringer {
    fun string(): String

}
```

Пример объявления [интерфейса в Go](#):

```
type Stringer interface {
    String() string

}
```

Как говорится, попробуйте найти 10 отличий в синтаксисе объявления, всё очень похоже.

Сама по себе концепция интерфейсов в Go не отличается какой-то нестандартностью. Можно сказать, что

Интерфейс – это набор методов с определенной сигнатурой.

Но всё же особенности есть. В Kotlin, да и во многих других языках программирования с интерфейсами, мы прямо указываем, что класс или что-то другое реализует интерфейс.

Пример реализации интерфейса в Kotlin:

```
inteface Stringer {
    fun string(): String
}

class MyClass : Stringer {
    override fun string(): String {
        return "MyClass"
    }

}
```

В Go же принят другой подход, который называют **duck-typing** (утиная типизация).

По-русски это звучит как

Если что-то плавает и крякает как утка, то это утка.

(c) Джейсон Стэтхем

Пример реализации интерфейса в Go:

```go
// https://goplay.tools/snippet/LewCu3w4ytF

type Stringer interface {
    String() string
}

type Temperature struct {
    Value float64
}

func (t Temperature) String() string {
    return fmt.Sprintf("%.1f °C", t.Value)

}
```

Тип `Temperature` реализует метод `String`, по сигнатуре идентичный методу, описанному в интерфейсе `Stringer`, и как следствие **неявно** "реализует" этот интерфейс.

Звучит запутанно, поэтому приведем еще один пример:

```go
// https://goplay.tools/snippet/16BTF0WQbNl

type Ducker interface {
    Quack() string // Крякать.
    Swim()         // Плавать.
}

type Duck struct{} // Утка "реализует" интерфейс Ducker.

func (d Duck) Quack() string {
    return "Quack!"
}

func (d Duck) Swim() {
    // Притворимся, что здесь утка плавает.
}

type Otter struct{} // Выдра Otter не "реализует" интерфейс Ducker,
потому что не умеет крякать.

func (o Otter) Squeak() string {
    return "Squaek!"
}
```

```
func main() {
    var d Ducker = Duck{}
    fmt.Println(d.Quack()) // Quack!

    d = Otter{} // Не скомпилируется:
    // cannot use Otter{} (type Otter) as type Ducker in assignment:
    // Otter does not implement Ducker
}
```

Тест "Docker"

Реализует ли тип `Docker` интерфейс `Ducker`?

```
type Ducker interface {
    Quack() string
    Swim()
}
type Docker struct {
}
func (d Docker) RunContainer(c Container) (ContainerID, error) {
    // ...
}
func (d Docker) StopContainer(cid ContainerID) error {
    // ...
}
```

- [ ] Да
- [ ] Нет

Тест "Какое zero value у переменной типа interface?"

https://golang.org/ref/spec#The_zero_value

- [ ] 0
- [ ] nil
- [ ] false
- [ ] ""
- [ ] uintptr(0)
- [ ] interface{}{}

Что лежит внутри интерфейса?

Взглянем на новый пример со старым типом `Duck` и интерфейсом `Ducker`.

Мы хотим принять ванну с чем-то, напоминающим уточку. Для этого берем настоящую уточку и отправляемся в ванную:

```
type Ducker interface {
    Quack() string
    Swim()
}

type Duck struct{}
func (d Duck) Quack() string { return "Quack!" }
func (d Duck) Swim() {}

func TakeBath(duck Ducker) { // Ванну принимаем с уточкой.
     duck.Swim()
}

func main() {
    var duck Duck  // Переменная типа Duck.
    TakeBath(duck) // Переменную подставляем в функцию с аргументом
типа Ducker.

}
```

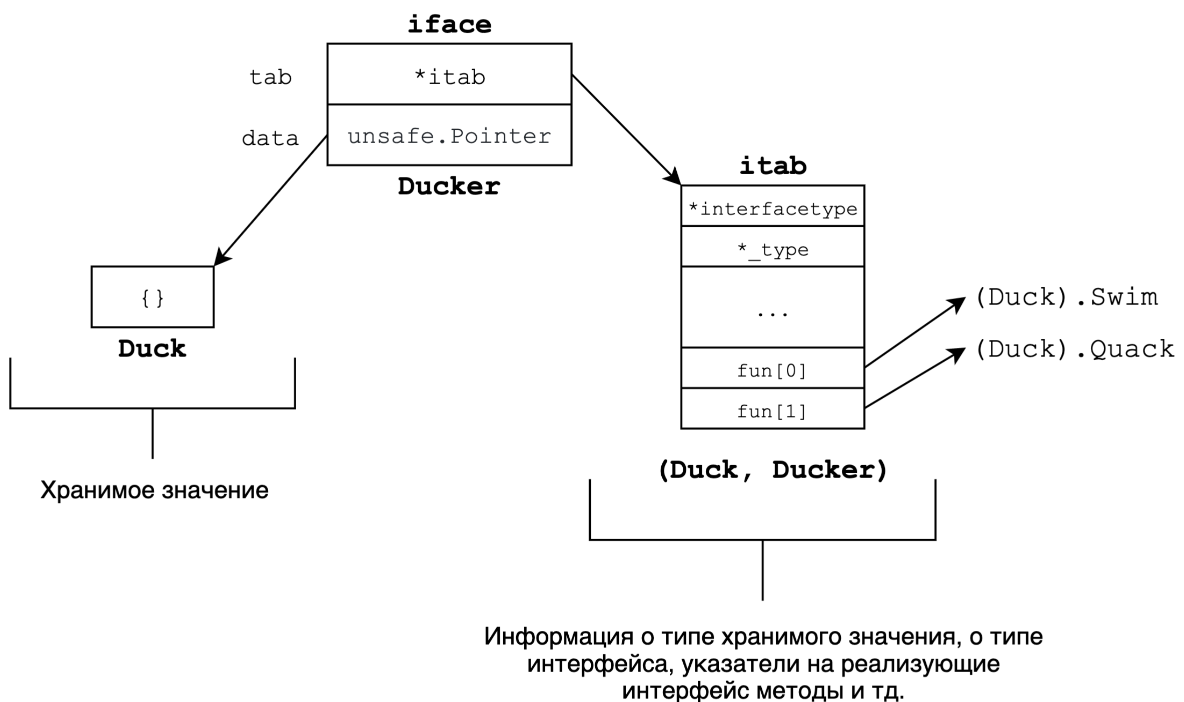
Внимание, вопрос: что будет внутри переменной `duck` типа `Ducker` в функции `TakeBath`?

Внутри переменной будет пара полей (именно так грубо можно рассматривать интерфейс в Go):

- динамический тип хранимого значения (`Duck`);
- собственно хранимое в интерфейсе значение (`Duck{}`).

Динамический тип – это информация о типе, реализующем интерфейс. В нашем случае в этом поле будет `Duck`. Динамический он, потому что в разные моменты времени в интерфейсе могут храниться различные значения различных типов, его реализующих.

Значение – это значение конкретного экземпляра типа, реализующего данный интерфейс (может быть или не быть указателем). В нашем случае это копия переменной `duck` из функции `main`.



Для чего это важно понимать?

Это важно, потому что интерфейс **является `nil` только тогда, когда оба его внутренних поля являются `nil`.**

Тест "nil интерфейс"

Сразу же на практике закрепим только что прочитанную информацию.

Взгляните на код ниже и выберите ответ, который будет на экране в результате выполнения `main`.

```
package main
import "fmt"
type Ducker interface {
    Quack() string
    Swim()
}
type Duck struct{}
func (d Duck) Quack() string { return "Quack!" }
func (d Duck) Swim() {}
func printDuck(d Ducker) {
    if d != nil {
        fmt.Println("hello ducker:", d)
    }
}
func main() {
    var d *Duck
    printDuck(d)

    d = &Duck{}
    printDuck(d)
}
```

```
Обращаем ваше внимание на
func printDuck(d Ducker) {
    if d != nil {
        fmt.Println("hello ducker:", d)
    }
}
```

- [ ] hello ducker: nil
- [ ] hello ducker: nil
  hello ducker: &{}
- [ ] hello ducker: &{}

- [ ] Ничего не будет выведено

Тест "Pointer Receiver"

Скажите, почему код ниже не компилируется?

```
import (
    "fmt"
    "net"
)

type Address struct {
    Protocol string
    Host     string
    Port     string
}

func (a *Address) Network() string {
    if a == nil {
        return ""
    }
    return a.Protocol
}

func (a *Address) String() string {
    if a == nil {
        return ""
    }
    return fmt.Sprintf("%s:%s", a.Host, a.Port)
}

func networkRequest(addr net.Addr) {
    // Делаем сетевой запрос.
}

func main() {
    addr := Address{
        Protocol: "tcp",
        Host:     "www.golang-courses.ru",
        Port:     "80",
    }

    for {
```

```
        networkRequest(addr)
    }
}
```

Интерфейс `net.Addr`:

```
package net

// Addr represents a network end point address.
type Addr interface {
    Network() string // name of the network (for example, "tcp", 
"udp")
    String() string  // string form of address (for example, 
"192.0.2.1:25", "[2001:db8::1]:80")
}
```

- [ ] У for нет условия выхода, он бесконечный
- [ ] Проблем не будет, код скомпилируется
- [ ] Тип Address не реализует интерфейс net.Addr
- [ ] Тело networkRequest пустое, параметр addr не используется

Приведение интерфейса к конкретному типу

Бывают ситуации, когда имеется необходимость узнать, кто на самом деле скрывается за маской красивого и лаконичного интерфейса.

Ниже пара способов, которыми в дальнейшем мы будем пользоваться:

- https://golang.org/ref/spec#Type_assertions

```
// 1. Прямое приведение интерфейса к типу (type assertion).

var w io.Writer

w = MyAwesomeStructWriter{Value: "some value"}

m := w.(MyAwesomeStructWriter) // <- !
fmt.Println(m.Value)           // some value

_, ok := w.(io.Reader) // <- !

fmt.Println(ok)        // false
```

- https://golang.org/ref/spec#Switch_statements

```
// 2. switch по типу интерфейса (type switch).

var w io.Writer

switch m := w.(type) { // <- !
case MyAwesomeStructWriter:
    fmt.Println(m.Value)

case *net.UDPConn:
    fmt.Println(`there was supposed to be a joke, but it won't reach
you`)

default:
    fmt.Printf("have no idea what it is: %T\n", m)

}
```

Задача "GetDeferredFunctionName"

[Ссылка на заготовку.]()

Вам необходимо реализовать функцию

```
func GetDeferredFunctionName(node ast.Node) string
```

которая принимает `ast.Node` – узел [абстрактного синтаксического дерева Go]():

```
package ast

// All node types implement the Node interface.
type Node interface {
    Pos() token.Pos // position of first character belonging to the node
    End() token.Pos // position of first character immediately after the node

}
```

и возвращает имя функции, чей вызов был отложен через `defer` (если узел является `*ast.DeferStmt`). Если откладывается вызов анонимной функции, то необходимо вернуть строковую константу `"anonymous func"`.

Например (псевдокод):

```
GetDeferredFunctionName(`defer foo()`)                 == "foo"
GetDeferredFunctionName(`defer fmt.Println("hello")`) == "fmt.Println"
GetDeferredFunctionName(`defer wg.Done()`)             == "wg.Done"
GetDeferredFunctionName(`defer func() {}()`)           == "anonymous func"
GetDeferredFunctionName(`go foo()`)                    == ""
GetDeferredFunctionName(`func bar {}`)                 == ""

GetDeferredFunctionName(`"something unknown"`)         == ""
```

- Не нужно изобретать космолётов, достаточно, чтобы юнит-тесты в заготовке проходили.

- Для понимания, по каким типам нужно `type switch`'ить или `type assert`'ить, воспользуйтесь дебагером или `fmt.Printf + %T`.

Зачем нужны интерфейсы?

Фундаментальный вопрос, ответ на который в один шаг не уместишь, но мы выделим основные причины использования интерфейсов в Go.

Определение поведения

На примере знакомого нам `net.Addr` – верхнеуровнему коду, работающему с адресом достаточно понимать

- какую сеть представляет адрес: `Network() string`;
- какое строковое представление адрес имеет: `String() string`.

```
package net

// net/net.go
// Addr represents a network end point address.
type Addr interface {
    Network() string // name of the network (for example, "tcp",
"udp")
    String() string  // string form of address (for example,
"192.0.2.1:25", "[2001:db8::1]:80")
}
```

Примеры реализаций интерфейса выше:

```
package net

// net/tcpsock.go
// TCPAddr represents the address of a TCP end point.
type TCPAddr struct {
```

```
    IP   IP
    Port int
    Zone string // IPv6 scoped addressing zone
}

func (a *TCPAddr) Network() string { return "tcp" }
func (a *TCPAddr) String() string { /* ... */ }

// net/udpsock.go
// UDPAddr represents the address of a UDP end point.
type UDPAddr struct {
    IP   IP
    Port int
    Zone string // IPv6 scoped addressing zone
}

func (a *UDPAddr) Network() string { return "udp" }
func (a *UDPAddr) String() string { /* ... */ }

// net/unixsock.go
// UnixAddr represents the address of a Unix domain socket end point.
type UnixAddr struct {
    Name string
    Net  string
}

// Network returns the address's network name, "unix", "unixgram" or
"unixpacket".
func (a *UnixAddr) Network() string { return a.Net }
func (a *UnixAddr) String() string { /* ... */ }
```

Выделение абстракции над адресом, позволяет нам выделить более *крутую* абстракцию — над соединением:

```
package net

// Conn is a generic stream-oriented network connection.
type Conn interface {
    Read(b []byte) (n int, err error)
    Write(b []byte) (n int, err error)
    Close() error

    LocalAddr() Addr
```

```
    RemoteAddr() Addr

    SetDeadline(t time.Time) error
    SetReadDeadline(t time.Time) error
    SetWriteDeadline(t time.Time) error

}
```

Теперь неважно, с каким соединением мы работаем (TCP, UDP и т.д.), независимо от конкретной реализации мы можем читать из него, писать в него, закрыть его и пр.

Более того мы не можем сделать ничего, что не определено в интерфейсе `Conn` — это своеобразный **контракт поведения** для нас, который невозможно нарушить.

Dependency Injection

Без [внедрения зависимостей]() в современной разработке на Go не обойтись:

```
type ILogger interface {
    Debug(msg string, args ...interface{})
    Info(msg string, args ...interface{})
    Warn(msg string, args ...interface{})
}

func NewApp(l ILogger) (*App, error) { // <- Приложение не зависит от конкретной реализации логера.
    return &App{logger: l}, nil
}
```

Дженереки по-домашнему

За отсутствие возможности [обобщённого программирования]() кто только уже не хаял Golang (на момент написания курса Go делает завершающие шаги к поддержке подобного), но интерфейсы помогают нам обойти проблему отсутствия generic'ов через выделение общего поведения.

Например, стандартная функция `Sort` (из пакета `sort`), сортирует всё, что определяет себя как *нечто сортируемое*:

```
package  sort

// An implementation of Interface can be sorted by the routines in
this package.
// The methods refer to elements of the underlying collection by
integer index.
type Interface interface {
    // Len is the number of elements in the collection.
    Len() int

    // Less reports whether the element with index i
    // must sort before the element with index j.
    Less(i, j int) bool

    // Swap swaps the elements with indexes i and j.
    Swap(i, j int)
}

// Sort sorts data.
// It makes one call to data.Len to determine n and O(n*log(n)) calls
to
// data.Less and data.Swap. The sort is not guaranteed to be stable.
func Sort(data Interface)  {
    n := data.Len()
    quickSort(data, 0, n, maxDepth(n))

}
```

Таким образом, `sort.Sort` может сортировать неограниченное множество разнотипных слайсов (и не только), лишь бы они удовлетворяли интерфейсу `sort.Inteface`.

Для примера реализуем обёртку `ByEmail`, позволяющую сортировать слайс пользователей по электронной почте:

```
// https://goplay.tools/snippet/IMC0VCCmTFM
package main

import  (
    "fmt"
    "sort"
```

```
)

type User struct {
    ID    string
    Email string
}

type ByEmail []User

func (s ByEmail) Len() int { return len(s) }
func (s ByEmail) Less(i, j int) bool {  return s[i].Email <
s[j].Email }
func (s ByEmail) Swap(i, j int) { s[i], s[j] = s[j], s[i] }

func main() {
    users := []User{
        {ID: "1", Email: "bob@gmail.com"},
        {ID: "2", Email: "alex@gmail.com"},
        {ID: "2", Email: "alice@gmail.com"},
    }

    sort.Sort(ByEmail(users))

    // [{2 alex@gmail.com} {2 alice@gmail.com} {1 bob@gmail.com}]
    fmt.Println(users)
}
```

(P.S. Когда выделить интерфейсы не представляется возможным, на помощь приходит кодогенерация).

Встроенный интерфейс `error`

И наконец, ошибки в Go реализованы через самый часто используемый в языке интерфейс – интерфейс `error`, о котором мы начнём говорить подробнее в следующем уроке:

```
package builtin

// The error built-in interface type is the conventional interface
for
```

```
// representing an error condition, with the nil value representing
no error.
type error interface {
    Error() string
}
```

Таким образом, интерфейсы в Go помогают нам писать обобщённый код; выделять контракты на основе поведения, а не конкретной реализации; организовывать внедрение зависимостей — тем самым избегая **утечек абстракции**. А также интерфейсы служат основой для некоторых фундаментальных концепций в языке (например, организации ошибок).

Пишите в комментарии или приводите свои примеры, отвечающие на вопрос "Зачем нужны интерфейсы?".

Промежуточные итоги

Мы освежили в памяти устройство интерфейсов и то, как приводить их к конкретному типу.

В следующем уроке мы начнём рассматривать ошибки в Go, которые на самом деле все являются экземплярами типов, реализующих специальный **встроенный интерфейс** `error`.