Введение в Golang. Лекция 1

Let's Go

Язык Go был разработан Кеном Томпсоном, который известен вам по операционной системе UNIX, и является одним из соавторов кодировки UTF-8, Робом Пайком, который тоже принимал участие в разработке UTF-8, а также участвовал в разработке операционных система Plan9 и Inferno. И Роберт Гризмер, который занимался Java HotSpot'ом, языком Swazall и распределёнными системами в Google. Первая версия языка была доступна публике в 2009 году, и в 2012 году вышла версия 1.0, то есть Go — это очень молодой язык.

Зачем нужен Go?

Go разрабатывался с прицелом на эффективность. Во-первых, это эффективность работы непосредственно самой программы — эффективная утилизация многопроцессорных систем. Дело в том, что в начале нулевых годов количество физических ядер в процессорах сильно выросло. При этом очень многие языки, которые проектировались в 90-х годах, были нацелены на работу только с одним ядром. Go же очень хорошо масштабируется по ядрам процессора, что позволяет писать очень эффективные, очень нагруженные сервисы. Также Go изначально проектировался для обработки большого количества конкурентных запросов — большого числа маленьких независимых запросов, которые поступают одновременно в программу.

Во-вторых, важной задачей была эффективность программиста. В Go очень простой синтаксис, очень мало магии и синтаксического сахара. Это позволяет писать очень простые программы. Простые в плане понимания и несколько уменьшающие груз legacy, потому что программы, как правило, живут гораздо дольше, чем мы думаем. Также именно для этих целей в Go задан жёсткий стиль форматирования кода, и создан инструмент для автоматического форматирования. Как правило, этот инструмент встраивается прямо в среду разработки, и при сохранении файла он сразу приводит стиль к нужному виду.

В Go очень быстрая компиляция. Это значит, что для того чтобы проверить, работает программа или нет, программисту не придётся ждать 20 минут, пока она соберётся. Кроме того в Go очень удобная работа с зависимостями. Весь код лежит в одном месте и при сборке собирается в статический бинарник, что позволяет ему не зависеть от каких-то внешних пакетов, и избегать конфликтов с другими программами.

В каких сферах применяется Go? Основными сферами применения Go являются web-разработка, системы мониторинга и разного рода системные утилиты. Go может плохо подойти в тех местах, где вам нужен очень жёсткий контроль над памятью и недопустимы практически никакие паузы, потому что Go — это язык со сборщиком мусора. Конечно, там нет таких пауз, как stop-the-world, которые надолго вешают всё, однако в каком-то виде они все-таки есть, поэтому в системах реального времени Go вам будет тяжело применить. Также если у вас очень мало памяти, например, вы используете какие-то встраиваемые устройства, Internet of Things, то Go, возможно, будет не самым лучшим выбором. Но для web'а Go — это очень хороший выбор.

Как установить Golang

В курсе используется версия Go 1.9. Скорее всего все примеры будут работать на версии 1.8, но не 1.7.

1. Windows - загрузить инсталятор с оф. сайта https://golang.org/dl/
и следовать инструкциям https://golang.org/doc/install

2. Linux - можно воспользоваться стандартным пакетным менеджером,
скриптом https://gist.github.com/jniltinho/8758e15a9ef80a189fce
или статьей https://www.tecmint.com/install-go-in-linux/.
Однако обратите внимание, что в вашем дистрибутиве может быть версия младше 1.8

3. MacOS - следуйте инструкциям с офф. сайта (ссылки выше) или
http://sourabhbajaj.com/mac-setup/Go/README.html.

Не забудьте перелогиниться после установки для обновления GOPATH - это бывает частой проблемой. Так же полезно будет добавить в переменную PATH путь к $GOPATH/bin - здесь стандартный установщик пакетов будет размещать запускаемые файлы.

В курсе используется бесплатный редактор Visual Studio Code со светлой темой оформления. Причина выбора данного редактора - большой красивый терминал, открываемый поверх кода. Ещё альтернативы:

Sublime Text 3 - https://www.sublimetext.com/ + плагин для golang
Vim-go - https://github.com/fatih/vim-go
GoLand - https://www.jetbrains.com/go/ - платный редактор от JetBrains, который
хорошо подойдёт тем, кто активно пользуется их софтом.

Первая программа на Go

Hello world

Любой пакет в go, любая программа начинается с имени пакета. Добавляется он при помощи служебного слова package. Особый пакет —package main, он является основным, и нем должна находиться функция main, которая будет запускаться при старте программы.

```
package main

import "fmt"

func main() {
        fmt.Println("Hello, World!")
}
```

В примере нам понадобится так же функция вывода текста на экран. Она находится в другом пакете, который называется fmt. Импортируем её, то есть скажем нашей программе, что будем использовать еще один пакет, командой import "fmt". Теперь напишем непосредственно функцию main. func — это ключевое слово, которое определяет, что далее будет определение функции, main это название функции. Выведем на экран по традиции "Hello, World".

Форматирование, запуск и компиляция.

Мы можем несколькими образами запустить код. Мы можем использовать команду go run, она нам скомпилирует бинарник во временном месте и запустит уже его, либо использовать команду go build. В процессе курса мы будем в основном пользоваться командой go run. Итак, выполняем go run hello_world.go. Hello, World выведется на экран.

Для форматирования кода в go есть специальная программа, называется gofmt. Запускается она обычно с флагом -w, который значит, что нужно перезаписать изменения в этот же файл. В go считается очень плохим тоном не использовать gofmt. На visual studio с плагином для go он будет запускаться автоматически на каждое ваше сохранение.

Немного про стиль.

В go принят camelCase стиль, under_score стиль не очень приветствуется. Если вы будете использовать какие-то синтаксические анализаторы, то они могут на это ругаться.

```
// camelCase - принятый стиль
userIndex := 10
// under_score - не принято
user_index := 10
fmt.Println(userIndex, user_index)
```

Основы языка

Переменные

Базовые типы данных

В этом разделе мы рассмотрим самые-самые основы языка — переменные. С переменными вы будете работать большую часть своей программистской жизни, потому что программирование в основном состоит в том, чтобы переложить данные из одной переменной в другую.

Переменная в go объявляется при помощи ключевого слова var, после которой следует имя переменной и тип переменной. При этом вы можете не указывать значения по умолчанию, и тогда переменная будет инициализирована значением по умолчанию.

```
// значение по умолчанию
var num0 int
```

В данном случае для *int* это ноль. Это одна из основных особенностей go, потому что нет риска получить какую-то неинициализированную переменную, в которой осталось что-то из предыдущей области памяти, от предыдущего владельца.

Можно объявить переменную и сразу присвоить ей значение.

```
// значение при инициализации
var num1 int = 1
```

Для некоторых типов данных возможно пропустить указание типа, и компилятор определит этот тип автоматически.

```
// пропуск типа
var num2 = 20
```

В данном случае в переменную num2 мы кладем 20, и компилятор определяет, что это тип int.

Иногда бывает так, что вам нужно что-то вернуть из функции и очень не хочется писать имя и тип переменной, которая там будет. Для этих целей в go есть короткое объявление. Оно осуществляется при помощи оператора :=.

```
// короткое объявление переменной
num := 30
```

Однако, используя короткое объявление, вы можете объявить только новую переменную, то есть присвоить значение в уже существующую переменную, используя этот оператор, вы не сможете.

```
num := 30
num := 31
```

При попытке выполнить эту программу, компилятор ругнется, что нет новых переменных.

В go можно объявить сразу несколько переменных через запятую и сразу же проинициализировать их.

```
// объявление нескольких переменных
var weight, height int = 10, 20
```

Можно присвоить им значение после объявления.

```
// присваивание в существующие переменные
weight, height = 11, 21
```

Также вы можете использовать короткое присваивание для объявления нескольких переменных, однако хотя бы одна из этих переменных должна быть новой.

```
// короткое присваивание
// хотя-бы одна переменная должна быть новой!
weight, age := 12, 22
```

Например, если здесь я напишу height, то компилятор будет ругаться, потому что ни одна из этих переменных не новая. Если же одна из них новая, то уже существующим присвоится значение и будет объявлена новая переменная.

Тип int

Во всех примерах мы рассматривали тип int. int платформозависимый тип в go. Он имеет разрядность либо 32 бита, либо 64 бита в зависимости от вашей операционной системы. int выбирается автоматически. Если вас не устраивает разрядность или вы хотите большего контроля, то в go поддержка от int8 до int64, которая позволяет хранить очень большие значения. Есть беззнаковый int — uint, он тоже может быть платформозависимым. Если платформозависимость не устраивает, к языке есть типы uint8 до uint64.

```
// int - платформозависимый тип, 32/64
var i int = 10

// автоматически выбранный int
var autoInt = -10

// int8, int16, int32, int64
var bigInt int64 = 1<<32 - 1

// платформозависимый тип, 32/64
var unsignedInt uint = 100500

// uint8, unit16, uint32, unit64
var unsignedBigInt uint64 = 1<<64 - 1
```

Тип float

Числа с плавающей точкой представлены как одинарной точности float 32, так и двойной точности. Для них тоже есть автоподстановка типа, автоугадывание типа, значение по умолчанию для них тоже ноль.

```
// float32, float64
var pi float32 = 3.141
var e = 2.718
goldenRatio := 1.618
```

Тип bool

В булевых переменных значение по умолчанию ноль, и оно тоже может угадываться автоматически.

```
// bool
var b bool // false по-умолчанию
var isOk bool = true
var success = true
cond := true
```

Для всех выше упомянутых типов данных есть все классические операторы. Конечно, есть оператор + =, есть постфиксный инкремент. Отдельное внимание стоит обратить только на то, что в go нет префиксного инкремента.

Тип complex

Зато в go есть комплексные переменные и полный набор функций для работы с ними. Один из создателей языка очень хотел, чтобы они там были, но на практике ими приходится пользоваться редко.

```
// complex64, complex128
var c complex128 = -1.1 + 7.12i
c2 := -1.1 + 7.12i
```

Тип string

Строки — это очень важный тип данных. Значение по умолчанию для строки — это пустая строка. Строка объявляется в двойных кавычках, и при этом внутри этих двойных кавычек могут быть спецсимволы, такие как перенос строки либо символ табуляции.

```
// пустая строка по-умолчанию
var str string

// со спец символами
var hello string = "Привет\n\t"
```

Если же вы хотите писать как есть, то вы можете использовать backtick'и, обратные кавычки.

```
// без спец символов
var world string = `Мир\n\t`
```

Тогда символ новой строки либо табуляции там будет прямо, как мы описали, $\backslash n$. В go строки из коробки поддерживают UTF-8, поэтому вы можете там писать на практически любых алфавитах, в данном случае на русском (а не на более изощренном китайском, потому что все, даже TEX, поддерживает Unicode сильно хуже, чем go).

```
// UTF-8 из коробки
var helloWorld = "Привет, Мир!"
```

Одинарные кавычки в go используются для символов byte, который является по сути alias'ом для uint8. Либо же для rune, который представляет собой полноценный UTF-8 символ. Внутри это uint32.

```
// одинарные кавычки для байт (uint8)
var rawBinary byte = '\x27'

// rune (uint32) для UTF-8 символов
var someRune rune = '*'
```

Как и в других языках, строки в go можно конкатенировать то есть соединять между собой при помощи оператора плюс.

```
helloWorld := "Привет Мир"
// конкатенация строк
andGoodMorning := helloWorld + " и доброе утро!"
```

Однако строки в go незаменяемы, вы не можете заменить какой-то один символ.

```
// строки неизменяемы
// cannot assign to helloWorld[0]
helloWorld[0] = 72
```

Вот в данном случае программа будет ругаться.
Как получить длину строки? Если вы попробуете получить длину строки, используя встроенную функцию len, то вы получите длину в байтах. Иногда это может запутать, потому что это не символы. Если там лежит внутри utf, то один символ может занимать больше одного байта. Если же вы хотите получить именно количество символов, вам придется воспользоваться специальной функцией RunCountInString из пакета utf8.

```
// получение длины строки
byteLen := len(helloWorld)                    // 19 байт
symbols := utf8.RuneCountInString(helloWorld) // 10 рун

// получение подстроки, в байтах, не символах!
hello := helloWorld[:12] // Привет, 0-11 байты
H := helloWorld[0]       // byte, 72, не "П"
```

Также можно получить часть строки, подстроку, используя оператор срез. Для этого надо указать, начиная с какого байта (включительно) и по какой байт (не включительно) вы хотите получить подстроку.

```
// получение подстроки, в байтах, не символах!
hello := helloWorld[:12] // Привет, 0-11 байты
H := helloWorld[0]       // byte, 72, не "П"
```

Еще одной особенностью строк является то, что их можно конвертировать в слайс байт, и обратно слайс байт вы можете конвертировать в строку. Подробнее про слайсы мы поговорим позднее.

```
// конвертация в слайс байт и обратно
byteString = []byte(helloWorld)
helloWorld = string(byteString)
```

Константы

Константа в go определяется при помощи ключевого слова const, после которой идет имя переменной, имя этой константы и опционально ее тип.

```
const pi = 3.141
```

При этом вы можете объявить сразу блок констант, то есть не указывая ключевое слово перед каждой константой.

```
const (
    hello = "Привет"
    e     = 2.718
)
```

Еще в go есть очень полезная вещь, называется iota. iota — это такой автоинкремент для константы. Довольно часто бывают случаи, когда вам надо объявить пачку констант, которые отличаются между собой на единицу. Например, значения битов в какой-то битовой маске. Автоинкремент тут очень кстати.

```
const (
    zero  = iota
    _     // пустая переменная, пропуск iota
    three // = 3
)
```

Вы указываете iota на первом месте, а дальше вам не нужно указывать ничего, в данном случае константа 3 объявилась через iota. Если вам нужно пропустить вдруг какую-то константу, вы можете использовать символом подчеркивания, это в go пустая переменная, которая говорит, что, да, здесь должно быть значение, но мы его проигнорируем. При этом iota может выступать не только каким-то простым автоинкрементом, но и участвовать в математических выражениях. Например, в данном случае константа мегабайт определяется тоже через iota.

```
const (
    _          = iota             // пропускаем первое значне
    KB uint64 = 1 << (10 * iota) // 1024
    MB                            // 1048576
)
```

Еще одной особенностью go является то, что нетипизированная константа, которой вы не объявили тип сами, она так и остается внутри нетипизированной. И может в нужное место подставиться уже с нужным типом. За счет этого, например, константы в go могут иметь целочисленные значения, значительно превышающие 64 бита.

```
const (
        // нетипизированная константа
        year = 2017
        // типизированная константа
        yearTyped int = 2017
)

func main() {
        var month int32 = 13
```

```
        fmt.Println(month + year)

        // month + yearTyped (mismatched types int32 and int)
        // fmt.Println( month + yearTyped )
}
```

В данном примере переменная «месяц», которая имеет тип int32. К ней можно добавить год, year, потому что это нетипизированная константа. При этом если попробовать добавить типизированную константу, то компилятор заругается, что так нельзя, надо преобразовывать тип.

Определение типов

В go вы можете определять собственные типы, либо основываясь на каких-то базовых примитивах, либо используя более сложные конструкции, такие как структуры, о которых речь пойдет позже. Возможность определять пользовательские типы очень полезна, если необходимо моделировать какие-то сущности. В качстве примера создадим тип UserID.

```
package main

type UserID int

func main() {
        idx := 1
        var uid UserID = 42

        // даже если базовый тип одинаковый, разные типы несовместимы
        // cannot use uid (type UserID) as type int64 in assignment
        // myID := idx

        myID := UserID(idx)

        println(uid, myID)
}
```

Особенностью Go является то, что в нем нет автоматического приведения типов. Никогда! Для того чтобы привести какую-то переменную базового типа к другому типу, который тоже основан на базовом типе, приходится делать конвертацию, в данном случае это типы совместимые, используя объявление типа и скобки, можно привести его к нужному UserID. То есть idx имеет типа int, а myID будет типа UserID. Также можно конвертировать простые инты между собой, или, как мы уже разбирали, строку со слайсом байт.

Указатели

Начать надо с того, что в Go нет адресной арифметики. Вы не можете прибавить какое-то значение к указателю и получить указатель на другую область памяти. В Go указатель — это отдельный тип данных. Что это значит? Это значит, что если вы объявляете какую-то переменную как указатель на другую, то внутри этой переменной будет лежать значение адреса на другую переменную. То есть указатель не является полноценной ссылкой, которая указывает ровно туда же, куда и основная переменная.

```
1   package main
2
3   import "fmt"
4
5   func main() {
6           a := 2
7           b := &a
8           *b = 3  // a = 3
9           c := &a // новый указатель на переменную a
10
11          // получение указателя на переменнут типа int
12          // инициализировано значением по-умолчанию
13          d := new(int)
14          *d = 12
```

```
15              *c = *d // *c = 12 -> a = 12
16              *d = 13 // c и a не изменились
17
18              c = d   // теперь c указывает туда же, куда d
19              *c = 14 // c = 14 -> d = 14, a = 12
20      }
```

Рассмотрим пример. Объявляем переменную a, Создаем переменную b и инициализируем её указателем на a. Теперь мы можем поменять значение в a, используя указатель на a и оператор разыменования *. Указателей на одну и ту же переменную может быть несколько. В строке 9 создаем c - новый указатель на a. Так же возможно создать указатель на какой-то тип данных, что часто используют, например, в структурах. В 13 строке создаем указатель на int, используя ключевое слово int, точнее встроенную функцию int, которая создаст переменную нужного типа, заполнит её значением по умолчанию, (это важно, так как у вас там не будет мусора) и вернёт вам указатель на эту переменную. Посмотрим, как происходит изменение какого-то значения. Сейчас переменная, лежащая за d, проинициализирована нулем. Изменим её значение на 12. Положим в переменную за c значение, которое лежит внутри d. И, таким образом, переменная a, указателем на которую является c, изменится. При этом, если теперь я поменять d, то и c и a уже не меняются, потому что c указывает на a, а не становится сразу же ссылкой на d. Однако, если теперь сделать так, что c — указатель на d, точнее в c теперь находится тот же указатель, который находится внутри d, то при изменении значения c, изменяется значение, которое лежит за c, и значение, которое лежит внутри d, но не значение, которое лежит внутри a.

Составные типы данных

В этом разделе мы проговорим уже про более сложный тип данных. И начнем мы с массива.

Массивы

Массив — это какой-то набор из нескольких данных, одного типа. Особенностью массивов в Go является то, что размер массива — это часть типа данных. Что это значит? Это значит, что массив размерностью 2 и массив размерностью 3 — это два совершенно разных типа данных и они между собой несовместимы. Причем размерность массива задается при компиляции, она не может быть изменена динамически. Как объявляется массив? Массив объявляется, используя квадратные скобки, в которых указан его размер, и потом тип данных, который находится внутри этого массива.

```
// размер массива является частью его типа
// инициализация значениями по-умолчанию
var a1 [3]int // [0,0,0]
fmt.Println("a1 short", a1)
fmt.Printf("a1 short %v\n", a1)
fmt.Printf("a1 full %#v\n", a1)
```

Вот в данном примере создаем массив из трех элементов, под них сразу же будет выделена память, и эти элементы сразу же будут проинициализированы значениями по умолчанию. Для переменной типа int — это 0. Для определения размера массива можно использовать константы, но не переменные.

```
const size = 2
var a2 [2 * size]bool // [false,false,false,false]
fmt.Println("a2", a2)
```

То есть нельзя написать вот так.

```
size := 2
var a2[size] bool
```

компилятор будет ругаться. Еще можно определить массив при инициализации. То есть я могу указать три точки, это значит, что возьми столько элементов, сколько я тебе скажу, в фигурных скобках. И в фигурных скобках могу сразу же инициализировать этот массив.

```
// определение размера при объявлении
a3 := [...]int{1, 2, 3}
fmt.Println("a2", a3)
```

В данном случае в массиве будет три элемента. Если же попробовать обратиться к массиву, выходя за его пределы, то это будет проверено при инициализации. Если это константа, то код просто не скомпилируется. Если же там будет такой id индекс, то это будет проверено уже в run time, и программа завершится паникой.

```
// проверка при компиляции или при выполнении
// invalid array index 4 (out of bounds for 3-element array)
// a3[idx] = 12
```

Итак, поскольку мы не всегда знаем, какого размера точно нам нужен массив и мы никак не можем это определить в run time, то массив — это довольно низкоуровневый тип данных, он используется не очень часто. Гораздо чаще используется другой тип, который основывается на массиве и называется он слайс.

Слайсы

Слайс — это чуть-чуть более сложная структура данных, чем массив, потому что у слайса есть его длина, то есть то количество элементов, которое там уже есть. И capacity, то есть то количество элементов, которое влезет еще в этот слайс без аллоцирования дополнительной памяти. Для начала рассмотрим просто создание слайса.

```
var buf0 []int             // len=0, cap=0
buf1 := []int{}            // len=0, cap=0
buf2 := []int{42}          // len=1, cap=1
buf3 := make([]int, 0)     // len=0, cap=0
buf4 := make([]int, 5)     // len=5, cap=5
buf5 := make([]int, 5, 10) // len=5, cap=10
```

Слайс можно создать, просто не указывая размерность в квадратных скобках. В данном примере мы создаем совершенно пустой слайс, у него не будет ни длины, ни capacity, он не инициализирован, там nil внутри. В слдующей строчке показано, как создать пустой слайс, который уже инициализирован, но у которого значения по-прежнему нет. Длина его 0, и capacity 0. Но слайс можно и сразу инициализировать чем-то. В данном случае кладем туда 42, и под это сразу выделяется массив размерностью 1 элемент, то есть его длина 1 и capacity тоже 1. Но гораздо чаще используется встроенная функция make, которая создает массив нужной размерности и capacity. Например, $make([]int, 0)$ создает вообще совсем пустой слайс. То есть у него нет ни значений, ни какой-нибудь памяти под него не инициализировано. Однако, если указать 5 вторым параметром, то я уже создается слайс, который будет проинициализирован пятью элементами, которые конечно же, будут иметь значения по умолчанию. При этом capacity будет являться также 5. Так же можно сразу указать capacity, то есть сразу размерность нижележащего массива, уже на слайс, на который ссылается этот слайс этой области памяти. Это очень полезно, если вдруг вы знаете, сколько элементов у вас будет в этом слайсе. Например, вы создаете слайс из нуля элементов, но вы сразу аллоцируете память для десяти элементов. Такой подход очень положительно сказывается на быстродействии программы, потому что если вы доходите до конца слайса и упираетесь в capacity, то тогда ваша программа вынуждена выделять новую область памяти, вдвое большего размера, копировать туда все значения из одной области в другую. А старые значения убирать сборщиком мусора.Очевидно, это не самые легкие операции.

Обращаться к элементам слайса можно также через квадратные скобки, как к массивам. При этом если вдруг вы обратитесь к элементу, который выходит за границы этого слайса, то выкинетеся run time паника.

```
// обращение к элементам
someInt := buf2[0]

// ошибка при выполнении
// panic: runtime error: index out of range
// someOtherInt := buf2[1]
```

Для того чтобы добавить элемент в слайс, есть специальная функция append, первым параметром которой передается сам слайс, а далее идут значения, которые вы хотите добавить.

```
// добавление элементов
var buf []int             // len=0, cap=0
buf = append(buf, 9, 10) // len=2, cap=2
buf = append(buf, 12)     // len=3, cap=4
```

В слайсе из примера у нас сначала нет ничего, после в слайс добавлено два элемента — 9 и 10. И на следующей строчке добавляется еще один элемент. Стоит обратить внимание, что при последнем добавлении произошла переаллокация слайса и capacity становится не 3, а 4, потому что при увеличении размерности run time просто делает x2 от предыдущего размера.

Если у вас есть слайс, вы хотите его домержить в текущий слайс, то для этого есть специальный оператор троеточие.

```
// добавление друго слайса
otherBuf := make([]int, 3)      // [0,0,0]
buf = append(buf, otherBuf...) // len=6, cap=8
```

Если бы мы написали без троеточия, то получилось бы, что в первоначальный слайс мы бы захотели добавить уже не инты, а другой слайс, как элемент. В данном случае получились бы несовместимые типы данных.

Информацию о слайсе можно получить, используя встроенную функцию len, которая говорит вам длину массива, то есть сколько там уже элементов, и функцию cap, capacity, которая говорит, сколько памяти аллоцировано.

```
// просмотр информации о слайсе
var bufLen, bufCap int = len(buf), cap(buf)

fmt.Println(bufLen, bufCap)
```

То есть len — это, количество элементов, которые уже есть в слайсе, а capacity — это количество элементов, которое может в слайс влезть, до того как будет аллоцирована еще память, до того как слайс будет расширен.

Еще одной особенностью слайса является то, что можно взять какой-то кусок его, который будет ссылаться ровно на ту же область памяти, на которую ссылается оригинальный слайс.

```
buf := []int{1, 2, 3, 4, 5}
fmt.Println(buf)

// получение среза, указывающего на ту же память
sl1 := buf[1:4] // [2, 3, 4]
sl2 := buf[:2]  // [1, 2]
sl3 := buf[2:]  // [3, 4, 5]
fmt.Println(sl1, sl2, sl3)
```

В данном примере объявлен слайс из пяти элементов от 1 до 5. Для того чтобы получить элементы, сослаться на какую-то его часть, вы можете в квадратных скобках указать, с какого элемента (включительно) по какой (не включительно) вы хотите сослаться. При этом можно пропустить либо первое значение и сослаться от нуля до в данном случае двойки, либо же пропустить второе значение и сказать: вот отсюда и до конца. Есть очень важная особенность при работе со слайсами, которую нужно понимать. Если вы ссылаетесь на ту же область памяти, (в примере создаем новый слайс такой же размерности указав «дай мне все»), и изменяете там какое-либо значение, то изменится в обоих слайсах.

```
newBuf := buf[:] // [1, 2, 3, 4, 5]
newBuf[0] = 9
// buf = [9, 2, 3, 4, 5], т.к. та же память
```

Если же добавить элемент в новый слайс, то тогда произойдет увеличение размерности этого слайса, потому что мы создали слайс, у которого capacity 5, и в нем лежит 5 элементов, и при добавлении 6-ого элемент слайс должен будет расшириться. Это значит, что создастся другая область памяти, и туда скопируются значения. то есть он уже будет ссылаться на другую область памяти. И теперь если поменять какое-то значение, то оно изменится уже в новом слайсе, а в старом не изменится.

```
// newBuf теперь указывает на другие данные
newBuf = append(newBuf, 6)

// buf    = [9, 2, 3, 4, 5], не изменился
// newBuf = [1, 2, 3, 4, 5, 6], изменился
newBuf[0] = 1
fmt.Println("buf", buf)
fmt.Println("newBuf", newBuf)
```

Это очень важная особенность, поиграйтесь с примером, посмотрите, как оно работает.

Осталось обсудить копирования слайса. Иногда бывает нужно скопировать один слайс в другой: не создать ссылку на ту же область памяти, а именно честно скопировать. Такой способ неправильный.

```
// копирование одного слайса в другой
var emptyBuf []int // len=0, cap=0
// неправильно - скопирует меньшее (по len) из 2-х слайсов
copied := copy(emptyBuf, buf) // copied = 0
fmt.Println(copied, emptyBuf)
```

Когда вы создаете пустой слайс и вызываете функцию копии, она копирует элементы в уже существующий слайс, уже в занятые элементы, то есть на место тех элементов, количество которых нам показывает переменная len. В данном случае len для слайса, в который мы копируем, равна 0, и мы на самом деле ничего не скопируем. Для того чтобы скопировать полноценно, нам нужно создать новый слайс, такой же размерности и такой же длины, сразу с данными, и уже скопировать в него.

```
// правильно
newBuf = make([]int, len(buf), len(buf))
copy(newBuf, buf)
fmt.Println(newBuf)
```

Тогда поведение будет ожидаемым. Копировать можно не только в переменную, но, например, в срез, слайс, который вы только что получили, который ссылается на какую-то другую область памяти, то есть на другой слайс, на часть другого слайса.

```
// можно копировать в часть существующего слайса
ints := []int{1, 2, 3, 4}
copy(ints[1:3], []int{5, 6}) // ints = [1, 5, 6, 4]
fmt.Println(ints)
```

Это очень полезный трюк, когда вам нужно хранить сначала записать длину данных, а потом сами данные, при этом размер данных вы еще не знаете (например, при бинарной упаковке). Поэтому вы пишете сначала нулевой размер, потом пишете данные, а потом, используя вот такой трюк, вы пишете длину в нужное место.

Мапа

Еще одним типом данных, про которые мы поговорим — это map. map, он же хеш-таблица, он же ассоциативный массив, он же - сложное слово - «отображение». map позволяет по ключу быстро получить значение. Это очень удобно, если у вас значений довольно много. Если бы они лежали в слайсе, вам бы пришлось все перебирать, а так вы сразу идете в нужное место. Как определяется map?

```
// инициализация при создании
var user map[string]string = map[string]string{
        "name":     "Vasily",
        "lastName": "Romanov",
}
```

При помощи ключевого слова map, потом в квадратных скобках идет тип ключа и тип данных. При этом вы можете, конечно же, сделать, например, мап мапа мапов. Что может выступать в качестве ключа? В качестве ключа может выступать любая сравниваемая структура данных.

В примере мы объявили мапу и сразу же её инициализировали. Чтобы инициализировать, нужно указать ключ, через двоеточие — значение. Запятая нужна для того, чтобы сказать компилятору, что там что-то идет дальше. При этом мапу можно точно так же создать, выделив сразу нужно место под нужное количество элементов, чтобы не приходилось расширять ее в рантайме.

```
// сразу с нужной ёмкостью
profile := make(map[string]string, 10)

// количество элементов
```

```
mapLength := len(user)

fmt.Printf("%d %+v\n", mapLength, profile)
```

Через len можно получить количество элементов в мапе.

Обратится к уже лежащим в мапе элементам можно при помощи квадратных скобок.

```
// если ключа нет - вернёт значение по умолчанию для типа
mName := user["middleName"]
fmt.Println("mName:", mName)
```

Но есть одна особенность: если этого ключа нет в мапе, то вернется значение по умолчанию. Про это обязательно надо помнить! Например, еслиу вас мапа, которая состоит из булов, то значение по умолчанию для була — это false. Если вернется false, то нужно как-то отличать что вам вернулось: значение false, которое действительно лежит по ключу, либо такого ключа вообще нет. Для разрешения этой ситуации можно получить признак существования ключа. Он может быть получен с использованием второй переменной при обращении к мапе.

```
// проверка на существование ключа
mName, mNameExist := user["middleName"]
fmt.Println("mName:", mName, "mNameExist:", mNameExist)
```

В данном случае получаем признак существования ключа в переменную nNameExist. В ней будет лежать булевая переменная, которая будет говорить, был там ключ, либо не было там ключа. Если же нужно только проверить, есть значение или нет, то можно пропустить первую переменную, используя символ подчеркивания — пустую переменную, говоря,что мы знаем, что там должна быть переменная, но она нам не нужна, и мы её не будем использовать.

```
// пустая переменная - только проверяем что ключ есть
_, mNameExist2 := user["middleName"]
fmt.Println("mNameExist2", mNameExist2)
```

И, наконец, удаление ключа. Удаление ключа происходит через встроенную функцию delete.

```
// удаление ключа
delete(user, "lastName")
fmt.Printf("%#v\n", user)
```

В функции delete указываете мапу и ключ. На этом со сложными типами данных мы закончили, теперь давайте поговорим про структуры.

Управляющие конструкции

В этом разделе мы рассмотрим управляющие структуры, благодаря которым задается вся логика программ.

Условный оператор if

Начнем с условного оператора. Условный оператор в if, как и в других языках программирования, представлен конструкцией if с условным выражением. А в качестве условия в Go может быть только булевая переменная.

```
// простое условие
boolVal := true
if boolVal {
        fmt.Println("boolVal is true")
}
```

Поскольку в Go нет приведения типов автоматически, то вы не можете, как, например, в PHP, написать пустую строку или ноль, чтобы ваше выражение прошло. Вы должны четко свести это к условному выражению!

Кроме простых условий, есть условие с блоком инициализации, в котором можно получить результат какой-то функции, либо, например, получить значение существования ключа в map'е, как в примере.

```
mapVal := map[string]string{"name": "rvasily"}
// условие с блоком инициализации
if keyValue, keyExist := mapVal["name"]; keyExist {
        fmt.Println("name =", keyValue)
}
// получаем только признак существования ключа
if _, keyExist := mapVal["name"]; keyExist {
        fmt.Println("key 'name' exist")
}
```

После блока инициализации, нужно поставить точку с запятой и написать логическое выражение, значение которого будет проверяться для выполнения условия. В данном случае мы проверяем, что ключ существует или не существует. Также, если не нужна часть значений, получаемых в блоке инициализации, то можно пропустить само значение, используя пустую переменную, то есть символ подчеркивания. Здесь самое время обратить внимание на то, что компилятор Go ругается на неиспользуемые переменные, поэтому их все следут заменять на пустые.

Оператор switch

В Go также есть множественный if else, когда вы можете комбинировать их в одну длинную цепочку.

```
cond := 1
// множественные if else
if cond == 1 {
        fmt.Println("cond is 1")
} else if cond == 2 {
        fmt.Println("cond is 2")
}
```

Это не всегда бывает удобно, поэтому в Go в дополнение к этому есть оператор switch. Оператор switch может работать по одной переменной, и в блоке case должны быть указаны условия, на соответствие которым проверяется эта переменная. Но есть нюансы. Во-первых, в отличие, например, от C, где по умолчанию в switch case у вас происходит проваливание в следующее условие и вам нужно писать везде break, в Go проваливание по умолчанию отсутствует. Вам, наоборот, нужно писать fallthrough, для того чтобы ваша программа провалилась в следующее условие. Также в условиях, то есть в блоке case, может стоять не одно значение, а несколько, перечисленные через запятые. Ну и конечно же, есть блок default, который будет выполнен тогда, когда ни одно из этих условий не применилось.

```
// switch по 1 переменной
strVal := "name"
switch strVal {
case "name":
        fallthrough
case "test", "lastName":
        // some work
default:
        // some work
}
```

Кроме выполнения switch по одной переменной, можно использовать разные условия. Тогда этот оператор больше похож на замену многим if else. Например, мы можем проверить, что либо одно условие, либо другое выполняется.

```
// switch как замена многим ifelse
var val1, val2 = 2, 2
switch {
case val1 > 1 || val2 < 11:
        fmt.Println("first block")
case val2 > 10:
        fmt.Println("second block")
}
```

При этом в условиях будут использованы совершенно разные переменные, то есть это будут совсем разные условия. Также иногда бывает нужно выйти из case где-то на середине. Для этого вы можете использовать оператор break. Тогда вы не пройдете в следующее условие, а выйдете из switch.

```
        switch {
                case key == "lastName":
                        break
                        println("dont pront this")
                case key == "firstName" && val == "Vasily":
                        println("switch - break loop here")
                        break
                }
```

В данном примере строчка "dont pront this"никогда не будет выведена на экран, потому что оператором break мы как бы сказали "пожалуйста, не ходи дальше". Но иногда бывает нужно, чтобы в операторе switch вы завершили какой-то цикл. В этом случае вам нужно указать метку этому циклу.

```
        // выход из цикла, находясь внутри switch
Loop:
        for key, val := range mapVal {
                println("switch in loop", key, val)
                switch {
                case key == "lastName":
                        break
                        println("dont pront this")
                case key == "firstName" && val == "Vasily":
                        println("switch - break loop here")
                        break Loop
                }
        } // конец for
```

в данном случае она находится на второй строчке, и поставить эту же метку после break.Ну и раз уж мы заговорили про циклы, давайте рассмотрим те циклы, которые есть в Go.

Циклы for

В Go цикл представлен только одной конструкцией for, но она может принимать совершенно разные формы. Во-первых, есть конструкция, когда у вас нет условия — бесконечный цикл. Это аналог while(true) либо for (;;;).

```
        // цикл без условия, while(true) OR for(;;;)
        for {
                fmt.Println("loop iteration")
                break
        }
```

В этом случае вам этот цикл нужно прервать самостоятельно. Для этого используется конструкция break. Следующим видом цикла является цикл с одиночным условием. То есть цикл будет выполняться до тех пор, пока это условие истинно.

```
        // цикл с одиночным условием, while(isRun)
        isRun := true
        for isRun {
                fmt.Println("loop iteration with condition")
                isRun = false
        }
```

Это аналог while переменная. Он будет выполняться до тех пор, пока переменная isRun = true. В данном примере в первой же итерации цикла isRun устанавливается false, и цикл завершает свою работу. Третьим вариантом цикла является классический for из C, когда вы указываете блок инициализации, блок с условием и блок, который выполняется после итерации цикла.

```
        // цикл с условие и блоком инициализации
        for i := 0; i < 2; i++ {
                fmt.Println("loop iteration", i)
                if i == 1 {
                        continue
                }
        }
```

Чаще всего используется автоинкремент индекса, по которому идет цикл. В данном случае мы выполняем всего лишь две итерации. И если у нас i = 1, то мы выполняем переход к следующей итерации цикла. Для этого в Go есть ключевое слово continue. Еще одним примером будет операция по слайсам. Например, объявим слайс и попробуем проитерироваться по нему. Самый простой способ — это использовать одно условие, пока мы не дошли до конца.

```
// операции по slice
sl := []int{1, 2, 3}
idx := 0

for idx < len(sl) {
        fmt.Println("while-stype loop, idx:", idx, "value:", sl[idx])
        idx++
}
```

Это аналог while. Мы будем печатать значение, печатать индекс и сами делать автоинкремент. Такое не очень удобно, поэтому гораздо чаще используется цикл с инициализацией, с условием и операцией после цикла.

```
for i := 0; i < len(sl); i++ {
        fmt.Println("c-style loop", i, sl[i])
}
```

В данном случае мы объявляем в одной строчке индекс, условие и автоинкремент для этого индекса. Четвертым типом цикла, который есть в Go, это цикл с оператором range. Оператор range заменяет всю рутину, как указать первоначальный индекс, указать условие, указать инкремент для этого индекса, вам достаточно просто проитерироваться, используя оператор range. Оператор range может вам вернуть либо только индекс, элемента из этого слайса,и вы уже сами можете вывести у него значение, например, вот так.

```
for idx := range sl {
        fmt.Println("range slice by index", sl[idx])
}
```

Либо же он может вам вернуть сразу значение и индекса и значения, и вам не придется уже обращаться к самому элементу слайса.

```
for idx, val := range sl {
        fmt.Println("range slice by idx-value", idx, val)
}
```

При этом происходит создание копии этого значения. Также вы можете итерироваться в цикле по map. Вы можете итерироваться как только по ключам этой map и сами обращаться к ее значениям.

```
// операции по map
profile := map[int]string{1: "Vasily", 2: "Romanov"}

for key := range profile {
        fmt.Println("range map by key", key)
}
```

Стоит заметить, что что в map порядок ключей не определен. Это значит, что в разных запусках программы ключи в памяти во внутренней структуре map могут располагаться совершенно по-разному. Это обусловлено особенностями реализации. Не ожидайте, что ключи будут в map ровно в том порядке, в котором вы их добавляли. Также вы можете сразу проитерироваться по ключу и по значению из map, также используя оператор range.

```
for key, val := range profile {
        fmt.Println("range map by key-val", key, val)
}
```

Либо если вам ключ не нужен, вы можете использовать, как всегда, пустую переменную — символ подчеркивания — и итерироваться только по значениям map.

```
for _, val := range profile {
        fmt.Println("range map by val", val)
}
```

Отдельно стоит отметить, как происходит итерирование по строке. Строка внутри себя представляет слайс байт, но range для строки определен отдельно. Это значит, что, итерируясь по строке по типу string, вы будете итерироваться не по байтам, а уже по отдельным символам. И в том, что вам возвращает range для строки, будет позиция, то есть номер, на котором находится ваш символ, и непосредственно символ, то есть руна, которую представляет собой utf-ный символ.

```
str := "Привет, Мир!"
for pos, char := range str {
        fmt.Printf("%#U at pos %d\n", char, pos)
}
```

Функции

Основы функций

В этом разделе речь пойдет про функции - основной инструмент для декомпозиции программы на более простые блоки. Функция в Go объявляется через служебное слово func, после которого следует имя функции, перечисление параметров в скобках и тип возвращаемого значения после скобок. Результат из функции возвращается, с помощью оператора return, как и в других языках.

```
// обычное объявление
func singleIn(in int) int {
        return in
}
```

При этом, если у вас несколько параметров одного типа, то вы можете указать тип только один раз, а сами параметры перечислить через запятую. Либо же указывать тип для каждого из параметров.

```
// много параметров
func multIn(a, b int, c int) int {
        return a + b + c
}
```

В Go можно указывать сразу указывать переменную, в которую будет возвращен результат.

```
// именованный результат
func namedReturn() (out int) {
        out = 2
        return
}
```

При этом вы можете записать туда значение и написать пустой return без имени этой переменной. И она вернет вам то, что вы записали. В данном случае это переменная out. Впрочем, можно написать,например, return 3, даже в таком случае функция будет корректно работать. В Go функции могут возвращать несколько результатов. Чаще всего всего это используется для возврата ошибки в качестве второго параметра.

```
// несколько результатов
func multipleReturn(in int) (int, error) {
        if in > 2 {
                return 0, fmt.Errorf("some error happend")
        }
        return in, nil
}
```

При этом параметров не обязательно должно быть два, их может быть три или более. Но очень многие служебные функции Go из стандартной библиотеки возвращают как раз два параметра. При этом при использовании return для возврата значения необходимо каждый раз указывать оба параметра через запятую. В данном случае указываем сначала первый параметр in, потом второй параметр ошибку. На 5-й строчке указан возвращаемый результат in и nil в качестве ошибки. Также можно вернуть несколько именованных результатов. Для этого достаточно объявить у них имена. Стоит обратить внимание, что все именованные возвращаемые результаты сразу инициализируются значениями по умолчанию.

```
// несколько именованных результатов
func multipleNamedReturn(ok bool) (rez int, err error) {
        rez = 1
        if ok {
                err = fmt.Errorf("some error happend")
                //return
                // аналогично return rez, err
                return 3, fmt.Errorf("some error happend")
                }
        rez = 2
        return
}
```

То есть, если в теле функции вы не переприсвоите значение, то будет возвращаться значение по умолчанию. Например, по умолчанию rez = 0, после присвоения rez = 1, функция будет возвращать 1. Если теперь присвоить какое-то значение ошибке на строчке 5, то return будет аналогичен return rez, err. Однако, еще раз обратим внимание,что никто не заставляет использовать именованные результаты. Мы все так же вправе вернуть самостоятельно, например, 3 и ошибку.

В Go функции могут принимать неограниченное количество параметров, но только одно типа. Такие функции называются вариативными, по английски variative. Для того чтобы так сделать, нужно указать перед типом параметра троеточие.

```
// не фиксированное количество параметров
func sum(in ...int) (result int) {
        fmt.Printf("in := %#v \n", in)
        for _, val := range in {
                result += val
        }
        return
}
```

В этом случае на вход, в данном случае в переменную in, поступит slice интов, по которым вы сможете проитерироваться, используя цикл for range. Обратим внимание на одну тонкость привызове функции с вариативным числом параметров.

```
func main() {
        nums := []int{1, 2, 3, 4}
        fmt.Println(nums, sum(nums...))
        return
}
```

У нас есть слайс nums, если передать в сумму просто слайс, то компилятор в этом случае будет ругаться, потому что это отдельный тип. А ожидается какое-то количество повторяющихся одиночных параметров. Поэтому необходимо использовать троеточие после имени слайса. Это распаковывает слайс в одиночные аргументы, которые передаются функции. Кстати, функция Println — как раз-таки пример вариативной функции. Она принимает множество параметров и выводит их в строковое представление.

Функция, как объект первого класса, анонимные функции

А теперь поговорим о функциях, как об объектах первого класса. Слова функция как объект первого класса означают, что вы можете присваивать функцию в какую-то переменную, принимать функцию, как аргумент в другую функцию и возвращать функцию, как результат работы какой-то функции. А также иметь функцию, как поле какой-то структуры. Например, обычная функция, определяется в отдельном блоке.

```
// обычная функция
func doNothing() {
        fmt.Println("i'm regular function")
}
```

То есть она имеет имя, в данном случае doNothing. А анонимная функция не имеет имени и определяется там, где вы захотите. Например, вот так можно объявить анонимную функцию.

```
func main() {
        // анонимная функция
        func(in string) {
                fmt.Println("anon func out:", in)
        }("nobody")
...
```

Мы не указываем имя, а сразу после ключевого слова func указываем входныепараметры. И сразу же вызываем эту функцию на строке 5, передавая в качестве параметра строчку nobody. То есть nobody — это параметр анонимной функции, который выведется в Println. Также анонимную функцию можно присвоить какой-то переменную и потом вызвать ее.

```
// присванивание анонимной функции в переменную
        printer := func(in string) {
                fmt.Println("printer outs:", in)
        }
        printer("as variable")
```

То есть мы будум обращаться к такой переменной, как к функции.

Иногда случается так, что вам приходится много где объявлять функции, передавать их куда-то, поэтому вы можете определить специальный тип функции. Так же, как и другие типы он определяется при помощи ключевого слова type, имени типа и базовой переменной.

```
// определяем тип функции
type strFuncType func(string)
```

В данном определение функции сигнатура функции тоже может выступать базовым типом. При этом можно указать как входящие параметры, так и возвращаемые значения,

Теперь давайте рассмотрим следующий вариант, когда мы хотим передавать функцию, как параметр, в другую функцию. Например, определим анонимную функцию worker, которая будет принимать переменную, которая называется callback типа strFuncType, определенного выше. Присвоим анонимную функцию переменной и будем её вызывать.

```
// функция принимает коллбек
worker := func(callback strFuncType) {
        callback("as callback")
}
worker(printer)
```

Передадим определенную выше функцию printer, которая будет выводить то, что в нее напечатали. На экран будут выведено as callback, то есть функция printer была вызвана, как callback. Callback-и полезны, если вы необходимо выполнить какую-то функцию по завершении какой-то работы или в зависимости от разных условий выполнить разную логику какой-то другой функции.

Теперь рассмотрим замыкание. Замыкание — это такая функция, которая обращается к переменным, которые были объявлены вне ее блока, вне ее объявления. Как это выглядит?

```
// функция возвращает замыкание
prefixer := func(prefix string) strFuncType {
        return func(in string) {
                fmt.Printf("[%s] %s\n", prefix, in)
        }
}
successLogger := prefixer("SUCCESS")
successLogger("expected behaviour")
```

Мы определяем функцию prefixer, которая принимает строчку prefix и возвращает функцию определенного типа. Определяем возвращаемую функцию в строчке return func... У неё есть свой входной параметр in, но она — самое интересное — будет использовать переменную prefix, которая была объявлена не в ее контексте, а в более в контексте более высокого уровня, Это называется замыкание, то есть функция замкнулась на что-то. Конечно, поскольку Go — язык со сборщиком мусора, переменная prefix не удалится сразу, как мы вернем значение, а будет существовать до тех пор, пока кто-то использует эту переменную. Давайте посмотрим, как это работает. Когда мы вызываем функцию prefixer, передавая туда строку

SUCCESS, то в successLogger записывается функция, принимающая на вход строку и печатающая её с префиксом SUCCESS. После вызова $successLogger(\text{"}expectedbehaviour\text{"})$, как и ожидалось, будет выведено $[SUCCES]\, expectedbehaviour$.

Отложенное выполнение и обработка паники

Теперь поговорим про такие понятия, как отложенное выполнение, паника и восстановление после паники. В Go есть возможность отложенного выполнения функции. Это значит, что какая-то работа будет выполнена после завершения функции. Чаще всего этот подход используется, когда вам нужно посчитать, например, время работы функции, либо закрыть какой-то ресурс, например, сетевое соединение либо файловый дескриптор. Причем, если у вас много условий и много returns, в каждом из них руками прописывать это бывает не очень удобно и довольно утомительно. Для этого в Go есть отложенное выполнение — вы можете объявить один раз отложенное выполнение на закрытие какого-то ресурса - это делается сразу же после открытия - и забыть о нем. Давайте посмотрим, как это работает.

```
package main

import "fmt"

func getSomeVars() string {
        fmt.Println("getSomeVars execution")
        return "getSomeVars result"
}

func main() {
        defer fmt.Println("After work")
        defer fmt.Println(getSomeVars())
        fmt.Println("Some userful work")
}
```

У нас есть функция main. Вначале мы определяем, что нужно выполнить какую-то функцию отложенно, то есть не сейчас, а в конце. Определяется это через ключевое слово defer, и непосредственно описание вызова. При запуске строка Some useful work выведется раньше, чем after work. Если используется несколько отложенных функций, то они выполняются в порядке, обратном их объявлению. В данном случае сначала выполнится функция $fmt.Println(getSomeVars())$, а потом $fmt.Println(\text{"}Afterwork\text{"})$. Стоит обратить отдельное внимание на аргументы, которые передаются в отложенные функции. Дело в том, что аргументы, передеваемые в отложенные функции, вычисляются на момент их объявления.Поэтому у нашей программы вывод будет такой:

```
getSomeVars execution
Some userful work
getSomeVars result
After work
```

Потому что сначала выполнилась GetSomeVars, потому что результат её выполнения является аргументом отложенной функции. Потом выполнилась какая-то полезная работа, потом вывелся результат, который вернула нам функция, GetSomeVars, и потом вызвался первый блок defer. Иногда такое поведение бывает не очень удобно, поэтому чаще используется подход, когда мы объявляем анонимную функцию, которая должна быть вызвана в конце.

```
package main

import "fmt"

func getSomeVars() string {
        fmt.Println("getSomeVars execution")
        return "getSomeVars result"
}

func main() {
        defer fmt.Println("After work")
        defer func() {
                fmt.Println(getSomeVars())
```

```
        }()
        fmt.Println("Some userful work")
}
```

У этой программы вывод уже будет другим:

```
Some userful work
getSomeVars execution
getSomeVars result
After work
```

Конструкции с defer очень полезны при восстановлении из паники. Сначала давайте рассмотрим, что такое вообще паника. Представьте, у нас есть какая-то функция, defer test. Она делает какую-то работу и потом паникует.

```
func deferTest() {
        fmt.Println("Some userful work")
        panic("something bad happend")
        return
}

func main() {
        deferTest()
        return
}
```

Паника это служебная функция, которая останавливает выполнение работы программы, то есть вся программа крашится. Как это работает? При выполнении данной функции выполнится какая-то полезная работа, потом вызовется паника и напечатается стек трейс. На самсом деле аника штука не очень хорошая, и завершать работающий демон в продакшене, когда обрабатывается много параллельных запросов, не очень хорошо. Паника — это абсолютно исключительная ситуация. И паники надо как-то отлавливать. Отлавливаются они с помощью блока defer.

Сначала необходимо объявить, собственно, блок, defer, который будет выполнен в любом случае после завершения функции, Внутри блока надо воспользоваться функцией recover, которая возвращает ошибку, выброшенную последней паникой.

```
func deferTest() {
        defer func() {
                if err := recover(); err != nil {
                        fmt.Println("panic happend:", err)
                }
        }()
        fmt.Println("Some userful work")
        panic("something bad happend")
        return
}
```

Получаем ошибку, если она вообще была, если err != nil, и обрабатываем её. Теперь программа будет завершаться корректно. Это может показаться немножко похожим на блок try-catch. Но следует особо отметить, что никогда не следует использовать панику и восстановление после паники как эмуляцию блока try-catch. Они для этого не предназначены. Паника это абсолютная исключительная ситуация, которая обычно приводит к краху программы. Она ловится для того, чтобы другие запросы, которые выполняются этой программой, не затронулись. Естественно, иногда бывает, что функция в defer случайно тоже упали в панику.

```
func deferTest() {
        defer func() {
                if err := recover(); err != nil {
                        fmt.Println("panic happend FIRST:", err)
                }
        }()
        defer func() {
                if err := recover(); err != nil {
```

```
                        fmt.Println("panic happend SECOND:", err)
                        panic("second panic")
                }
        }()
        fmt.Println("Some userful work")
        panic("something bad happend")
        return
}
```

Другие функции, которые выполняются позже, могут поймать эту панику, но вообще бросать панику в восстановлении очень плохая практика. Так работает паника. Пользуйтесь ею с осторожностью!

Структуры и методы

Основы работы со структурами

Теперь рассмотрим такой тип составных данных, как структуры. По мере роста программы бывает неудобно моделировать сущности набором скаляр, и возникает желание как-то объединить их, чтобы они представляли собой единую сущность. Структуры — это как раз такое объединение. Давайте рассмотрим, как они объявляются. Структуры, как и все остальные типы в Go, объявляются через ключевое слово type, потом идёт имя этого типа, ключевое слово struct, и в фигурных скобках объявляются поля этой структуры.

```
type Person struct {
        Id      int
        Name    string
        Address string
}
```

Полями структуры может быть абсолютно любой тип, который есть в Go, например string или даже функция, или другая структура.

```
type Account struct {
        Id int
        // Name    string
        Cleaner func(string) string
        Owner   Person
}
```

Как инициализируются структуры? Для того чтобы инициализировать структуры, можно воспользоваться либо полным форматом объявлений — в фигурных скобках указать нужные поля, при этом какие-то поля можно пропустить, и тогда они примут значения по умолчанию.

```
// полное объявление структуры
        var acc Account = Account{
                Id: 1,
                Name: "rvasily",
        }
```

Либо же мы можем воспользоваться краткой формой объявления, тогда нам не нужно указывать имена всех полей структуры, но при этом нам нужно указать значения абсолютно для всех полей структуры.

```
// короткое объявление структуры
acc.Owner = Person{2, "Romanov Vasily", "Moscow"}
```

Обращение к полям структуры происходит через точку.

```
fmt.Println(acc.Name)
```

Иногда бывает так, что мы хотим больше композиции. В Go нет ООП в классическом понимании, однако вся работа с объектами в Go построена на композиции. Например, мы можем встроить одну структуру в другую. Делается это так.

```
type Account struct {
        Id int
        // Name    string
        Cleaner func(string) string
        Owner   Person
        Person
}
```

В чем отличие Owner от Person? Дело в том, что Owner — это какое-то свойство структуры, а встроенный, заэмбедженный Person — это уже не свойство структуры, а часть самой структуры. И все поля Person являются частью структуры Account. Например, вы можете обращаться к этим полям непосредственно.

```
fmt.Println(acc.Adress)
```

То есть Address является частью поля структуры Person, но, поскольку мы встроили Person в Account, то мы можем обращаться к полям Person напрямую.

Обратим внимание, что при объявлении Person'а встроенной структуры, всё равно нужно объявлять её через префикс.

```
var acc Account = Account{
        Id: 1,
        // Name: "rvasily",
        Person: Person{
                Name:    "Василий",
                Address: "Москва",
        },
}
```

Если, как в данном примере, в структуре есть поля с одинаковыми именами у структуры, в которую встраиваем, и у встраиваемой структуры, то никакого конфликта не будет. Оба поля сохранятся. При этом при обращении к этому полю приоритет будет к наиболее верхнему полю структуры. То есть при выполнении fmt.Println(acc.Name) будет выведено поле Account, в котором лежит rvasily, а не поле Person. Если же мы хотим всё-таки обратиться к полю встроенной структуры Person, мы должны явно это указать.

```
fmt.Println(acc.Person.Name)
```

Теперь выведется Василий.

Методы структур

Теперь рассмотрим методы. Метод — это какая-то функция, которая может быть привязана к определенному типу данных. Для начала рассмотрим методы, которые привязываются к структурам. Определим тип Person.

```
type Person struct {
        Id   int
        Name string
}
```

И несколько методов для него.

```
// не изменит оригинальной структуры, для которой вызван
func (p Person) UpdateName(name string) {
        p.Name = name
}

// изменяет оригинальную структуру
func (p *Person) SetName(name string) {
        p.Name = name
}
```

Метод отличается от обычной функции только тем, что перед определением имени этой функции добавляется имя типа, для которого определен этот метод, с указанием роли получателя. Роль получателя — это то, в каком виде этот метод получит тип, к которому привязан. Это может быть либо передача по

значению, то есть вам в метод передается копия этого типа, либо по адресу. В первом случае в функции UpdateName любые изменения, вызванные в этом методе, оригинальную структуру не затронут. На самом деле функция UpdateName смысла не имеет. Если же мы в качестве роли указываем адрес на тип, то тогда все изменения, которые были внесены в структуру этим методом сохранятся.

Попробуем выполнить.

```
pers := Person{1, "Vasily"}
pers.SetName("Vasily Romanov")
```

Поле Name действительно обновляется. Но обратим внимание, что pers у нас не ссылка на структуру, а сама структура. В то время как «получателем» указан адрес структуры. Дело в том, что Go определяет, в каком виде мы хотим получить получателя, либо по значению, либо по адресу, и автоматически выполняет необходимые преобразования. Точно так же можно было вызвать функцию с помощью строчки

```
(&pers).SetName("Vasily Romanov")
```

или, например, если бы pers было не структурой, а указателем на структуру, то есть создавалось бы при помощи

```
new(Person)
```

или

```
pers := &Person{1, "Vasily"}
```

и вызывалась бы

```
pers.SetName("Vasily Romanov")
```

Возникает закономерный вопрос: что происходит с методами, если встраивать одну структуру в другую?

```
type Account struct {
        Id   int
        Name string
        Person
}
```

Правильно, метод наследуется. То есть структура account в примере может иметь доступ ко всем методам структур, которые в нее встроены. В данном случае она может иметь доступ к методу SetName. Давайте посмотрим, как это работает. Создадим Account, создадим там Person и вызовем функцию SetName

```
var acc Account = Account{
                Id:   1,
                Name: "rvasily",
                Person: Person{
                        Id:   2,
                        Name: "Vasily Romanov",
                },
        }

acc.SetName("romanov.vasily")
```

После вызова функции изменится значение в Person. Как вы помните, сохраняются оба поля name, который был и во встроенной структуре, и в структуре верхнего уровня. А что будет если объявить функцию для Account?

```
func (p *Account) SetName(name string) {
        p.Name = name
}
```

Теперь при вызове

```
acc.SetName("romanov.vasily")
```

изменится уже поле Name аккаунта, потому что его метод имеет больший приоритет, чем метод встроенной структуры. Если же теперь требуется вызвать метод непосредственно у встроенной структуры, то обратиться к нему можно, используя полный селектор.

```
acc.Person.SetName("Test")
```

Используя такой подход, можно создавать очень мощные структуры данных. При этом методы могут быть не только у структур. Например, можно объявить какой-то тип, например

```
type MySlice []int
```

И у этого типа объявлять методы. Например,с помощью методов можно добавлять значения и получать длину.

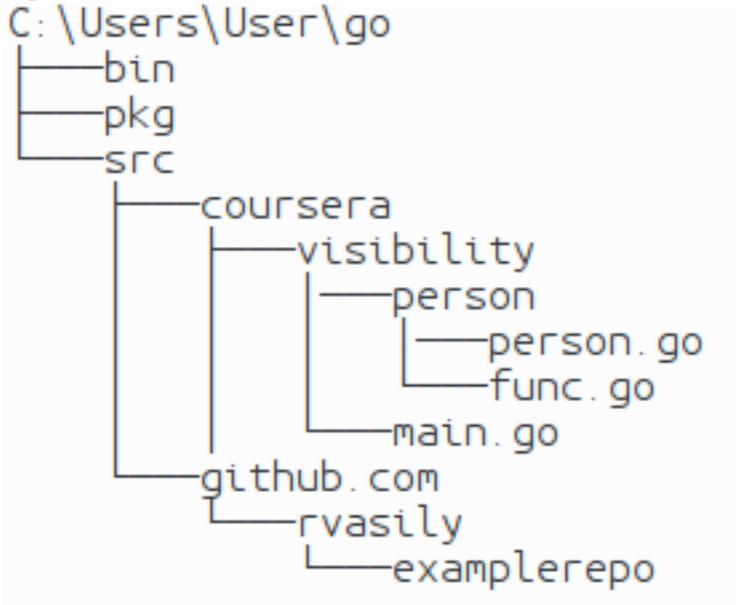
```
func (sl *MySlice) Add(val int) {
        *sl = append(*sl, val)
}

func (sl *MySlice) Count() int {
        return len(*sl)
}

func main() {
        sl := MySlice([]int{1, 2})
        sl.Add(5)
        fmt.Println(sl.Count(), sl)
}
```

Пакеты и область видимости

Теперь поговорим систему пакетов в go, а также про область видимости ваших переменных в этих пакетах. Давайте сначала рассмотрим базовую структуру. Go работает с так называемой системной переменной go pass. Это адрес директории, внутри которой должны находиться некоторые специализированные директории.



Первая из них — это папка bin, в нее кладутся собранные бинарники, то есть либо .exe, либо l-файлы. Вторая - pkg, в pkg кладутся временные объектные файлы, а src непосредственно содержит исходники вашей программы. Например, весь курс может лежать в папке coursera. Также если вы вдруг клонируете какой-то пакет, какую-то библиотеку, например, с github, то она также будет копироваться в src. Например, github, rvasily и examplerepo положатся по этому пути в соответствующую папку.

Теперь поговорим непосредственно про область видимости. Если есть папка coursera, visibility и в ней main.go, который непосредственно собирает программу, и пакет person, там лежит два файла: person.go, func.go.

```
//main.go
package main

import (
```

```
        "coursera/visibility/person"
        "fmt"
)

func main() {
        p := person.NewPerson(1, "rvasily", "secret")

        // p.secret undefined (cannot refer to unexported field or method secret)
        // fmt.Printf("main.PrintPerson: %+v\n", p.secret)

        secret := person.GetSecret(p)
        fmt.Println("GetSecret", secret)
}
```

```
// person.go
package person

var (
        Public  = 1
        private = 1
)

type Person struct {
        ID     int
        Name   string
        secret string
}

func (p Person) UpdateSecret(secret string) {
        p.secret = secret
}
```

```
//func.go
package person

import (
        "fmt"
)

func NewPerson(id int, name, secret string) *Person {
        return &Person{
                ID:     1,
                Name:   "rvasily",
                secret: "secret",
        }
}

func GetSecret(p *Person) string {
        return p.secret
}

func printSecret(p *Person) {
        fmt.Println(p.secret)
}
```

В файле main.go обязательно указывается package main, потому что это тот пакет, который собирается, у него жестко зафиксировано имя. person.go уже находится в пакете person и в нем должно быть прописано package person. Обратите внимание, что не требуется прописывать весь путь, только название пакета. Итак, в go очень простое правило. Если переменная, структура, поле структуры, константа, функция начинается с заглавной буквы, это значит, что эта сущность экспортируемая, то есть она доступна для обращения из других пакетов. Если же она начинается со строчной, это значит, что эта сущность принадлежит только этому пакету. Обратите внимание. Пакет состоит не из файла, а из директории. То есть

одна директория — это и есть пакет. Следовательно, все, что я объявлю в файле person.go, будет доступно и из файла func.go, потому что они составляют один пакет.

Импорт в свою очередь распространяется только на файл. Кроме того нельзя оставлять неиспользуемые импорты. Если, например, написать импорт в person.go, то программа не соберется, потому что есть неиспользуемый импорт fmt. То есть, если функции из пакета fmt используются только в func.go, то и импорт должен быть указан только строго в func.go. При попытке обращения к приватным полям после импорта пакета, программа не скомпилируется. Например, можно получить ошибку

```
visibility\main.go secret unidentified (cannot
refer to unexported field or method).
```

Отдельно хочется отметить, что в go не принято дробить программу на миллион маленьких пакетов, внутри которых лежит всего лишь один файл. Правильным подходом считается, когда у вас большие пакеты. Это не значит, что надо складывать всю-всю-всю вашу программу в пакет main, хотя такое тоже бывает, но бесконечно дробить её не надо,

Также последнее время набирает становится распространенной ситуация, когда вы работаете одновременно не с проектом, и у вас не один go pass. Для таких ситуация используется Vendor — папка, в которой лежат все ваши зависимости, и используется специальный инструмент, например, gb, либо dep, которые управляют этими зависимостями и подтягивают их в соответствующую папку. Однако это будет рассматриваться позже.

Интерфейсы

Основы работы с интерфейсами

Теперь рассмотрим такую важную часть языка Go, как интерфейсы. Через интерфейсы в Go реализован полиморфизм, то есть возможность функции принимать аргументы различных типов. В отличие от таких языков, как C++, Java и PHP, где типизация явная, в Go реализована утиная типизация. Что это значит? В C++, когда вы создаете класс, вы явно указываете, что он реализует такой интерфейс. «Вот мои документы, вот я наследовался от этого интерфейса, я его реализую.Я — утка.Я точно знаю, что я — утка». В Go, когда вы создаете структуру с методами, она не знает, какому интерфейсу она соответствует, то есть типизация неявная. Там применяется другой подход: Если что-то крякает, как утка, плавает, как утка и летает, как утка, то это утка. То есть за соблюдение контракта этого интерфейса отвечает не сама структура, а метод, в который вы ее передаете. То есть вы передаете в метод, который принимает определенный интерфейс. А уже интерфейс описывает, что всем, кто хочет ему удовлетворять необходимо иметь определенные методы. Давайте посмотрим, как это работает в коде.

```
type Payer interface {
        Pay(int) error
}
```

В примере показано, как объявить тип интерфейса. Сначала нужно указать служебное слово type, затем имя интерфейса и ключевое слово interface, чтобы указать, что этот тип является интерфейсом. В примере объявлен интерфейс Payer, «плательщик», для того чтобы соответствовать этому интерфейсу, нужно иметь метод pay, «заплатить», который принимает int и возвращает ошибку. Реализуем структуру «кошелек», у которой есть какое-то поле cash, то есть количество денег в этом кошельке, и есть метод «заплатить».

```
type Wallet struct {
        Cash int
}

func (w *Wallet) Pay(amount int) error {
        if w.Cash < amount {
                return fmt.Errorf("Не хватает денег в кошельке")
        }
        w.Cash -= amount
        return nil
}
```

Обратите внимание: в реализации кошелька нигде нет упоминания того, что он каким-либо образом реализует интерфейс «плательщик». И теперь напишем функцию Buy, которая принимает в себя интерфейс «плательщик». Она не знает уже, какую именно структуру в неё передадут, но ей важно, чтобы то, что придет в эту функцию, обладало методом Pay.

```
func Buy(p Payer) {
        err := p.Pay(10)
        if err != nil {
                panic(err)
        }
        fmt.Printf("Спасибо за покупку через %T\n\n", p)
}
```

Теперь, если запустим

```
func main() {
        myWallet := &Wallet{Cash: 100}
        Buy(myWallet)
}
```

то получим вывод «Спасибо за покупку через main.Wallet». При помощи %T в форматируемом выводе можно получить тип переданного аргумента.

Это был простой пример, так как в нем была всего одна структура, реализующая интерфейс. Давайте теперь рассмотрим несколько более сложный пример, когда у есть несколько структур, которые реализуют интерфейс.

```
package main

import (
        "fmt"
)

// --------------

type Wallet struct {
        Cash int
}

func (w *Wallet) Pay(amount int) error {
        if w.Cash < amount {
                return fmt.Errorf("Не хватает денег в кошельке")
        }
        w.Cash -= amount
        return nil
}

// --------------

type Card struct {
        Balance    int
        ValidUntil string
        Cardholder string
        CVV        string
        Number     string
}

func (c *Card) Pay(amount int) error {
        if c.Balance < amount {
                return fmt.Errorf("Не хватает денег на карте")
        }
        c.Balance -= amount
        return nil
```

```
}

// --------------

type ApplePay struct {
        Money   int
        AppleID string
}

func (a *ApplePay) Pay(amount int) error {
        if a.Money < amount {
                return fmt.Errorf("Не хватает денег на аккаунте")
        }
        a.Money -= amount
        return nil
}

// --------------

type Payer interface {
        Pay(int) error
}

// --------------

func Buy(p Payer) {
        err := p.Pay(10)
        if err != nil {
                fmt.Printf("Ошибка при оплате %T: %v\n\n", p, err)
                return
        }
        fmt.Printf("Спасибо за покупку через %T\n\n", p)
}

// --------------
```

Реализована структура кошелек, в нем есть какое-то количество денег и реализован метод Pay. Реализована структура карточка, в которой хранится баланс, дата, до которой она валидна, CVV, имя карточки, имя владельца, и она тоже реализует метод Pay. И есть ApplePay. В нем хранится количество денег на аккаунте и AppleID. И он тоже реализует метод Pay. Так же реализова интерфейс «плательщик», который требует, чтобы у типа, подаваемого на вход был метод Pay. И функция «купить». Попробуем этим воспользоваться.

```
func main() {

        myWallet := &Wallet{Cash: 100}
        Buy(myWallet)

        var myMoney Payer
        myMoney = &Card{Balance: 100, Cardholder: "rvasily"}
        Buy(myMoney)

        myMoney = &ApplePay{Money: 9}
        Buy(myMoney)
}
```

В main создаем кошелек, и через него что-то покупаем. Теперь создаем не сразу какой-то объект, который дальше будем передавать, а переменную типа «плательщик». И теперь в эту переменную можно присваивать любые структуры, которые реализуют этот интерфейс. То есть проверка на интерфейс может быть не только при передаче в функцию, но и при присвоении в переменную. Соответственно при создании структуры можно указывать, что какое-то поле должно иметь определенный интерфейс, соответствовать определенному интерфейсу. И, наконец, присваиваем в эту же переменную, которая реализует интерфейс «плательщик», другую реализацию этого плательщика - ApplePay.

Теперь, если это запустить будет выведено:

```
Спасибо за покупку через *main.Wallet

Спасибо за покупку через *main.Card

Ошибка при оплате *main.ApplePay: Не хватает денег на аккаунте
```

Иногда бывает нужно не просто вызывать какие-то методы интерфейса, но и проверять, какая именно структура, удовлетворяющая интерфейсу поступила на вход. Для этих целей у нас есть специальная конструкция type switch. Реализуется она через оператор switch, p и запрос типа в скобочках. Перепишем функцию Buy с использованием type switch.

```
func Buy(p Payer) {
        switch p.(type) {
        case *Wallet:
                fmt.Println("Оплата наличными?")
        case *Card:
                plasticCard, ok := p.(*Card)
                if !ok {
                        fmt.Println("Не удалось преобразовать к типу *Card")
                }
                fmt.Println("Вставляйте карту,", plasticCard.Cardholder)
        default:
                fmt.Println("Что-то новое!")
        }

        err := p.Pay(10)
        if err != nil {
                fmt.Printf("Ошибка при оплате %T: %v\n\n", p, err)
                return
        }
        fmt.Printf("Спасибо за покупку через %T\n\n", p)
}
```

Смотрите, если интерфейс представлен типом «кошелек», выведем «Оплата наличными». Если интерфейс представлен типом Card, то получим доступ к данным этой карточки. Когда в функцию передается интерфейс, нельзя просто так обратиться к полям структуры, которая лежит под этим интерфейсом. Если так сделать, будет вызвана паника. Например,

```
Cardholder undefined (type Payer has no field or method Cardholder).
```

если мы пытаемся получить доступ к имени владельца. Если преобразование прошло успешно, в plasticCard действительно будет лежать тип Card, и теперь можно обращаться к его полям, это уже не интерфейс.

Далее мы рассмотрим пустой интерфейс, который позволяет передать в себя вообще все что угодно.

Пустой интерфейс

В этом разделе мы поговорим про пустые интерфейсы. Пустой интерфейс — это интерфейс, который может принять в себя совершенно любую переменную, потому что у него нет никаких требований к реализации. Начнём рассматривать пустые интерфейсы с демонстрации их работы.

```
func (w *Wallet) String() string {
        return "Кошелёк в котором " + strconv.Itoa(w.Cash) + " денег"
}

// -----

func main() {
        myWallet := &Wallet{Cash: 100}
        fmt.Printf("Raw payment : %#v\n", myWallet)
        fmt.Printf("Способ оплаты: %s\n", myWallet)
}
```

Определим кошелёк, выполним вывод кошелька в двух форматах. Первый формат — это полное Go-шное представление структуры, второй формат — это строка. Запустим. Получим такой вывод:

```
Raw payment : &main.Wallet{Cash:100}
Способ оплаты: Кошелёк в котором 100 денег
```

Итак, первый вывод вывел прямо имя структуры вместе со всеми полями и значениями, которые там есть. Второй вывод вывел нам какой-то текст. Каким образом функция Printf() догадалась, что надо выводить?. Явно, что функция стандартной библиотеки не знает ничего про тип, который мы только что определили. На самом деле функция Printf() принимает в себя пустой интерфейс. Поэтому мы туда можем передавать абсолютно любые параметры, одни за другими, она все из них выведет. Но каким образом получилось, что вывелось то, что надо? Дело в том, что если структура, реализует интерфейс Stringer, то тогда функция Pintf() вызовет функцию String, которая уже отформатирует вывод так, как мы хотим.

Пустые интерфейсы — очень полезная вещь, когда нам нужно делать совсем генерические функции, которые работают со всем, чем угодно. Как внутри происходит преобразование интерфейсов? Перепишем еще раз функцию для оплаты так, чтоб она принимала не интерфейс-плательщик, а пустой интерфейс.

```
func Buy(in interface{}) {
        var p Payer
        var ok bool
        if p, ok = in.(Payer); !ok {
                fmt.Printf("%T не не является платежным средством\n\n", in)
                return
        }

        err := p.Pay(10)
        if err != nil {
                fmt.Printf("Ошибка при оплате %T: %v\n\n", p, err)
                return
        }
        fmt.Printf("Спасибо за покупку через %T\n\n", p)

}
```

В функцию теперь можно передать всё, что угодно, и она должна проверить не на этапе компиляции соответствие интерфейсу, а уже в runtime'е. Поэтому нужно попытаться вход преобразовать к плательщику. Если преобразование не удалось, то выводим, что вход платехным средством не является, иначе обрабатываем, как прежде. Выполним

```
func main() {
        myWallet := &Wallet{Cash: 100}
        Buy(myWallet)
        Buy([]int{1, 2, 3})
        Buy(3.14)
}
```

Кошелёк пройдет успешно, а вот slice интов и float не пройдут, потому что они не реализуют интерфейс «плательщик».
Пустые интерфейсы — очень мощный инструмент, и мы будем неоднократно сталкиваться с применением пустого интерфейса.

Композиция интерфейсов

Теперь обсудим встраивание интерфейсов. С интерфейсами можно поступать подобно структурам, когда вы можете вложить одну структуру в другую и иметь доступ к её полям. Можете встраивать один интерфейс в другой, тем самым образуя более сложные интерфейсы. Давайте рассмотрим пример.

```
type Payer interface {
        Pay(int) error
}

type Ringer interface {
        Ring(string) error
```

```
}

type NFCPhone interface {
        Payer
        Ringer
}
```

Объявляем интерфейс Плательщик, который требует метод Pay. И интерфейс Звонилка, который требует метод позвонить. Теперь можно объявить интерфейс NFCPhone, то есть телефон, который реализует метод бесконтактной оплаты через NFC технологию. Этот интерфейс образован композицией двух других интерфейсов: Плательщик и Звонилка. То есть получается, что на самом деле мой интерфейс выглядит вот так.

```
type NFCPhone interface {
                Pay(int) error
                Ring(string) error
}
```

Но каждый раз полностью объявлять совсем новые типы не очень удобно, поэтому композиция интерфейсов позволяет облегчить жизнь. Не обязательно использовать только интерфейсы, один интерфейс можно встроить, а другой, например, объявить. Это также очень мощный способ композиции интерфейсов, и много в стандартной библиотеке реализовано через него.

Примеры

Написание программы уникализации

Теперь мы напишем простую программу для уникализации строк, которые подаются в стандартный ввод. Нам потребуется несколько стандартных пакетов: «os» для того, чтобы получить доступ к спинартному вводу, «fmt» для того, чтобы делать форматированный вывод, и буферизированный ввод-вывод из пакета io.

В функции для считывания ввода не посимвольно, а сразу всей строкой нам нужен сканер ввода. После того, как мы его создали, будем построчно двигаться по вводу в цикле. Когда сканировать будет больше нечего, то scan вернет false, и мы выйдем из цикла. Каждую строку, которую мы считали, будем выводить на экран в том случае, если она прежде не встречалась. Для этого можно использовать структуру $map[string]bool$, чтобы запоминать строки, которые мы уже видели, но использовать map может быть не очень выгодно в случае, если у нас очень большой массив входных данных — мы можем просто не влезть по памяти. Поэтому сразу напишем программу так, чтобы она была похожа на стандартную команду unick, которая принимает на вход отсортированный набор данных. За счет этого мы можем вместо всех данных хранить только предыдущее значение, но все равно понимать, уникальна ли поданная на вход строка. Если новая строка больше текущей, — будем сравнивать просто по байтам, — то она новая, напечатаем её, если совпадает, то мы её уже выводили, если меньше, то это значит, что файл не отсортирован, и наша программа в этом случае работать не может, мы просто запаникуем. Итак, вот получившаяся программа.

```
package main

import (
        "bufio"
        "fmt"
        "io"
        "os"
)

func main() {
        in := bufio.NewScanner(os.Stdin)
        var prev string
        for in.Scan() {
                txt := in.Text()
                if txt == prev {
                        continue
```

```
                }
                if txt < prev {
                        panic("file not sorted")
                }
                prev = txt
                fmt.Println(txt)
        }
}
```

Попробуйте её запустить и потестировать, верно ли она работает. Вдруг мы сделали какую-то ошибку в ней, и на каких-то условиях она будет работать некорректно. А в следующем разделе мы напишем тесты для программы, чтобы быть уверенными, что она всегда отрабатывает, даже если внести в нее какие-то модификации.

Написание тестов для программы

Для того чтобы написать тесты, нам нужно нашу программу немножко «порефакторить», потому что нужно, чтобы тестируемый код находился в функциях, отличных от main. Вынесем весь наш код в отдельную функцию, которая будет просто принимать на вход поток, из которого мы будем читать, и поток, куда мы будем писать результат. Это все реализуется через интерфейс Reader и Writer. Вместо стандартного вывода будем писать при помощи функции Fprintln, которая первым параметром принимает интерфейс Writer. В случае ошибки больше не будем паниковать, а будем возвращать ошибку и обрабатывать её уже в main или тестах. Не забудем вернуть пустую ошибку при корректном завершении функции. Вот так изменится программа после рефакторинга.

```
package main

import (
        "bufio"
        "fmt"
        "io"
        "os"
)

func uniq(input io.Reader, output io.Writer) error {
        in := bufio.NewScanner(input)
        var prev string
        for in.Scan() {
                txt := in.Text()
                if txt == prev {
                        continue
                }
                if txt < prev {
                        return fmt.Errorf("file not sorted")
                }
                prev = txt
                fmt.Fprintln(output, txt)
        }
        return nil
}

func main() {
        err := uniq(os.Stdin, os.Stdout)
        if err != nil {
                panic(err.Error())
        }
}
```

Теперь давайте приступим к написанию непосредственно тестов. Тесты в go должны лежать в файле, который имеет суффикс test. Например в *main – test.go*. Все тесты в go начинаются с префикса Test и принимают на вход единственный параметр тестирующего модуля *testing.T. Создадим тестовые данные, результат, с которым будем сравнивать работу функции и сам тест.

```
import (
        "bufio"
        "bytes"
        "strings"
        "testing"
)

var testOk = `1
2
3
3
4
5`

var testOkResult = `1
2
3
4
5
`

func TestOk(t *testing.T) {
        in := bufio.NewReader(strings.NewReader(testOk))
        out := new(bytes.Buffer)
        err := uniq(in, out)
        if err != nil {
                t.Errorf("test for OK Failed - error")
        }
        result := out.String()
        if result != testOkResult {
                t.Errorf("test for OK Failed - results not match\n %v %v",
                                                        result, testOkResult)
        }
}
```

В тесте мы запускаем нашу функцию на тестовых данных. Если получаем ошибку, говорим, что тест сфейлился. Если ошибки не было, проверим, что результат работы программы корректный, сверившись с образцом.

Чтобы запустить тестирование, надо ввести в командной строке go test — v. v — это значит verbals, то есть мы будем видеть результаты: какие тесты заработали, а какие закончились неудачей. Убедитесь, что тест проходит корректно.

Но это не все, что можно сделать. Еще остался случай, когда мы должны вернуть ошибку, если поток ввода не отсортирован. Давайте напишем еще один тест.

```
var testFail = `1
2
1`

func TestForError(t *testing.T) {
        in := bufio.NewReader(strings.NewReader(testFail))
        out := new(bytes.Buffer)
        err := uniq(in, out)
        if err == nil {
                t.Errorf("test for OK Failed - error: %v", err)
        }
}
```

В него мы будем подавать заранее неправильные, то есть не отсортированные данные, и говорить, что тест сфейлился, если программа отрабатывает без ошибки, то есть не замечает некорректность ввода.

Запустите тесты еще раз. Теперь, мы можем быть не просто уверены, что программа работает корректно, но и что мы сможем быстро проверить её корректность, даже если будем вносить в неё изменения, потому что у нас есть тесты.