OTUS

# Golang Professional

## Тема: Синтаксис языка

otus.ru


**Начинается запись.**
**Проверка связи**

# Меня хорошо видно && слышно?

Ставим “+”, если все хорошо
“!!!!”, если есть проблемы

Тема вебинара

# Синтаксис языка Golang.


**Рубаха Юрий**

**Опыт разработчиком:**

3 года разработчик Golang
5+ лет предшествующего опыта на других технологиях (преимущественно Vb.Net, C#)

**telegram:**
@Hover2

# Организационные моменты

Прогнозируемая
длительность – 90 минут

Активно
участвуем

Задаем вопрос
в чат или голосом
(предпочтительнее голосом,
т.к. вебинар короткий)

Вопросы вижу в чате,
могу ответить не сразу

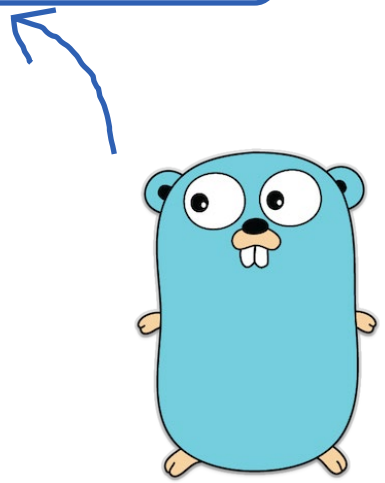
# Содержание занятия

**Общая информация о синтаксисе**

**Переменные**

**Функции**

**Управляющие конструкции**



# Общая информация о синтаксисе

## Сейчас рассмотрим:

- **package**
- **func main**
- **Import**
- **Комментарии // , /* */**
- **Блоки { }**

## Вернемся к этому сегодня позже:

- **«Невидимые точки с запятой»**
- **Управление видимостью**

# package

Ключевое слово **package** встречается в каждом *.go файле ровно 1 раз.

```
package main
```

Пакеты могут иметь разные имена.
В исполнимом приложении должен обязательно быть пакет main. В нём должна находиться точка входа в приложение.

# Func main() {  }

Особенная функция: func main() { }
Эта функция будет запущена рантаймом при запуске приложения.

Она должна находиться в пакете main

```
package main

func main() {
   print("Hello go-world")
}
```

Другие пакеты тоже могут содержать функцию main, но она не будет иметь таких свойств, и не будет вызываться автоматически при запуске приложения.

# import

Для использования пакета его нужно сначала подключить
Для этого используется ключевое слово import
Перед import не должно быть ничего, кроме package

```
package main
import "fmt"
func main() {
  fmt.Println("Hello go-world")
}
```

# Import (продолжение)

```
package main

import (
  "fmt"
  "sync"
)
```

Import может быть однострочным, подключая один
может содержать сразу импорт многих пакетов

```
package main
import (
  f "fmt"
)
func main() {
  f.Println("Hello go-world")
}
```

Пакет можно импортировать с другим удобным именем.
Но стандартные пакеты не принято переименовывать

# Комментарии //, /* */

Есть два типа комментариев:

Строчный //

Блочный /*
*/

Но есть и директивы, например
//nolint
(это уже не комментарий)

```
package main
import (
  "fmt" //это комментарий
)
//комментарий
func main() { //это комментарий
  fmt.Println("Hello go-world")

  /*
    многострочный
    комментарий
  */
}
```

# Блоки { }

Почти весь наш код находится в блоках

Блоки могут содержать объявления переменных, типов, выполнение выражений, вызов функций

Блоки могут быть вложенными. Во вложенном блоке доступны переменные, объявленные в блоках верхних уровне.

Переменные, объявленные в блоках более низкого уровня сверху не видны.

# Блоки { }

Почти весь наш код находится в блоках

Блоки могут содержать объявления переменных, типов, выполнение выражений, вызов функций

Блоки могут быть вложенными. Во вложенном блоке доступны переменные, объявленные в блоках верхних уровне.

Переменные, объявленные в блоках более низкого уровня сверху не видны.

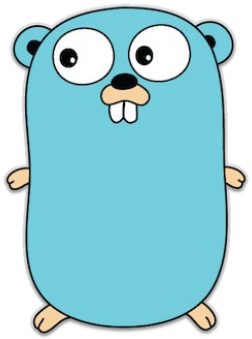
# Содержание занятия

Общая информация о синтаксисе

**Переменные**

Функции

Управляющие конструкции



# Переменные

## Сейчас рассмотрим:

- простейшие типы данных
- способы объявления переменных
- присвоение значений переменным
- множественные присвоения
- Символ нижнего подчеркивания
- Zero-values
- Константы, iota

# Простейшие типы данных

Go – строготипизированный язык. У каждой переменной есть конкретный тип. Он определяется при создании переменной и не меняется в ходе выполнения программы.

Тип переменной указывает компилятору на:

1) Сколько переменная занимает памяти
2) Из чего она состоит
3) Что с ней можно делать

На ближайшие полчаса нам хватит трех типов данных

- int – хранит целые числа со знаком (полож. и отриц.)
- bool – хранит логическое значение. Может быть только в состоянии true или false
- string – хранит строку символов.

# Способы объявления переменных

Можно выделить 6 способов объявления переменных
1. С помощью ключевого слова var, с указанием типа без указания значения.

```
var temperature int
var message string
```

2. С помощью ключевого слова var, с указанием типа и указанием значения.

```
var temperature int =-5
var message string = "Холодно"
```

Во втором случае тип данных иногда можно не указывать, если тип переменной совпадает с типом значения.

# Способы объявления переменных

## Выведение типов

Во многих случаях тип переменной можно и не указывать. Выведение типов – Go определяет тип выражения справа, и наделяет переменную слева этим же типом.

3. С помощью ключевого слова var, БЕЗ указания типа с указанием значения

```
var temperature = -5  //будет выведен тип int
var message = "Холодно" //будет выведен тип string
var ok = true //будет выведен тип bool
```

# Способы объявления переменных

## Выведение типов

4. Без ключевого слова var, БЕЗ указания типа с указанием значения. Очень характерный для Go

```
temperature := -5    //будет выведен тип int
message := "Холодно" //будет выведен тип string
ok := true           //будет выведен тип bool
```

В этом случае переменная и объявляется, и получает начальное значение. Тип переменной будет жестко зависеть от типа значения, которое было в момент задания переменной.

5 и 6 способ связаны с объявлением функций. Увидим сегодня позже.

# Способы объявления переменных

## Области определения переменных

Переменные могут быть объявлены за пределами блоков { }
Тогда это будут глобальные переменные уровня пакета.
Такие переменные можно объявлять только способами 1-3 с помощью слова var

Переменные внутри блоков называются локальными
Их можно объявлять всеми 4 способами

# Способы объявления переменных

## Области определения переменных

```
package main

var price int
var name string = "Автомобиль"
var age =10
// здесь так нельзя
// power:=200

func main() {
  var price int
  var name string = "Грузовик"
  var age =5
  // здесь так можно
  power:=500
}
```

Одноименные локальные переменные «затеняют» глобальные. Об этом будет подробнее на другом занятии

# Присвоение значений переменным

Переменной можно присвоить значение литерала («захардкоженого значения»), другой переменной, выражения, результат вызова функции (о которых мы еще не говорили, но поговорим сегодня)

```
package main

func main() {
  var a = 10       //присвоено значение литерала
  b := a           //присвоено значение другой переменной
  var c int = a + b //присвоено значение выражения

  //и дальше делаем всё, что нам нужно для задачи
  c = 5
  b = 2*a + b
  //и т.д. Полная свобода действий
}
```

# Множественные присвоения

Оператор присвоения = в Go позволяет присваивать несколько значений нескольким переменным. Количество значений слева должно совпадать с количеством значений справа. Переменные могут быть разных типов. Но типы переменных нарушать нельзя.

```
package main

func main() {
  //объявление с множественным присвоением
  var a, b = 10, 20
  //множественное присвоение с выведением типа
  age, name := 20, "Вася"
  //обмен значениями
  a, b= b, a
}
```

Для ясности: сначала вычисляются все значения в правой части, а потом записываются в переменные левой части.

# Символ нижнего подчеркивания _

Все переменные, объявленные в блоках { }
обязательно должны быть использованы для чтения!
Если этого не сделать, будет ошибка компиляции.
К переменным уровня пакета такого требования нет.

_ - это универсальный приемник значений любых типов. В него можно записать любое значение. Применяется для:

1) Временно, чтобы компилятор увидел, что переменная используется
2) Для выравнивания количества принимаемых и передаваемых значений (увидим позже).

```
package main

func main() {
  var a=10 //пока что a не используется для чтения
  _=a //теперь используется
}
```

Этим злоупотреблять не стоит!
Но есть и другие применения _, которыми можно пользоваться сколько угодно.

# Zero-values

В Go все объявленные переменные (независимо от способа) являются инициализированными и могут сразу же использоваться как на чтение, так и на запись.
Даже если переменной не присвоено значение, она содержит не «мусор» – непредсказуемое значение, а т.е. Zero-value. Эти значения разные для разных типов данных. Например

```
var a int //будет равно 0
var b string //будет равно "" (пустая строка)
var c bool //будет равно false
```

# Константы

В Go есть константы – переменные, значения которых нельзя менять в процессе выполнения программы.
Константы определяются с помощью слова const
Константы могут быть уровня пакета и блока { }, как и переменные.
Не все типы данных пригодны для создания констант (об этом мы поговорим позже).

```
const absoluteZero=-273
const myName="Юра"

func main() {

  const temperature = 451
  const hello = "Hello world!"
}
```

# iota

Iota – особое ключевое слово, которое позволяет создать несколько констант со связанными друг с другом значениями.
Iota начинается с 0 и для каждой следующей константы увеличивается на 1

```
const (
  zero =iota //будет иметь значение 0
  one =iota //будет иметь значение 1
  two //iota опустили, но она продолжает работать, 2
  three //3
  _ //задействуем пустой получатель, чтобы пропустить
4
  five //5
)
```

# iota

(продолжение)

Iota можно использовать и в составе более сложных выражений, например

```
const (
  one = iota +1 //задали выражение, результат =1
  two //выражение будет повторяться, результат
=2
  three //повторяется, результат 3

)

const (
  ten =(iota+1)*10 //10
  twenty      //20
  thirty      //30
)
```

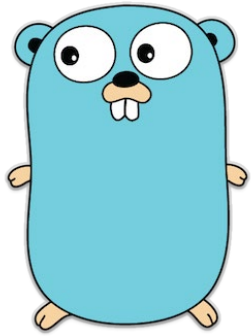
# Содержание занятия

Общая информация о синтаксисе

Переменные

**Функции**

Управляющие конструкции



# Функции

## Сейчас рассмотрим:

- Именованные функции
- Принимаемые и возвращаемые значения
- Сигнатура функции
- Return для разных случаев
- Вызов функции
- Присвоение переменным результата вызова функции
- Обычный набор возвращаемых значений (res, err) или (res, ok)

# Именованые функции

Объявление именованной функции состоит из
- ключевого слова func (обязательно)
- имени функции (обязательно)
- перечня входных параметров (опционально)
- перечня возвращаемых значений (опционально)
- тела функции (обязательно)

```
func sum (a int, b int) int {
  return a+b
}
```

Имена функций в пакете должны быть уникальными. Перегрузок функций в Go нет. Мы не можем сделать две функции с одинаковым именем даже с разными параметрами.

# Принимаемые и возвращаемые значения

В Go нет «процедур». Вместо процедур используются функции без возвращаемых значений.

Если функция принимает подряд несколько параметров одного типа, описание можно сократить так:

```
func sum(a, b int) int {

}
```

То же самое, что и:

```
func sum(a int, b int) int {

}
```

# Принимаемые и возвращаемые значения

Функции могут возвращать более, чем одно значение.
Это выглядит так:

```
func div(a, b int) (int, int) {

}
```

Выше пример функции с именованными входными параметрами, и неименованными выходными параметрами

```
func div(a, b int) (d int, r int) {

}
```

А это пример функции с именованными входными и выходными параметрами.

# Принимаемые и возвращаемые значения

5-й и 6-й способ объявления переменных – это их объявление в качестве входного или выходного параметра функции.
Если параметр функции именован, то на уровне функции он уже существует как переменная, и может быть использован как для чтения так и для записи, как

```
func div(a, b int) (d int, r int) {
  d=a/b
  r=a%b
}
```

Как видим, переменные a, b, d, r используются уже без дополнительных объявлений.

# Сигнатура функции

Сигнатурой функции называется набор типов входных и выходных параметров функции, учитывая их порядок.
При этом имя функции, имена параметров и вообще их именованность, а так же тела (реализации) функций значения не имеют.

Ниже пример двух функций с одинаковой сигнатурой

```
func getWeekDayName(day int) string {
// ... реализация опущена
}

func getMonthName (monthNumber int) (monthName string) {
  //... реализация опущена
}
```

# Ключевое слово return

Return – это команда, выполнение которой завершает выполнение функции. Return – это именно команда, а не «отметка» конца функции, поэтому

- return может в функции быть не один
- return’ов может быть в функции несколько

return завершает выполнение функции безусловно

если функция не имеет выходных параметров, она может не содержать return. Такая функция будет завершаться при достижении последней строки функции.
Для функций с возвращаемыми параметрами наличие хотя бы одного return в теле обязательно.

# Ключевое слово return (продолжение)

Пример функции без return

```
func hello(message string) {
  fmt.Println("Сейчас выведем сообщение")
  fmt.Println(message)
} //функция закончится здесь
```

Но и для таких функций можно в теле использовать слово return, чтобы прекратить выполнение функции в определенном месте.

# Ключевое слово return (продолжение)

Функции, возвращающие неименованные параметры, должны содержать return. При этом все сценарии выполнения функции должны заканчиваться return.
А возвращаемые в return значения должны совпадать по типам с возвращаемыми параметрами функции.
Возвращать можно литералы, значения переменных, или значения выражений

```
// функция возвращает день и месяц Нового года
func newYear() (int, string) {
  return 1, "январь"
}
```

# Ключевое слово return (продолжение)

Еще пример

```
// возвращает день, месяц и год Миллениума
func millenium() (int, string, int) {
  thousand := 1000
  first := 1
  jan := "январь"
  return first, jan, 2 * thousand
} // функция вернет 1, "январь", 2000
```

# Ключевое слово return (продолжение)

Если у функции выходные параметры (возвращаемые параметры) именованы, то они уже создаются в момент вызова функции.
В этом случае мы можем делать return без параметров, тогда из функции вернутся параметры с теми значениями, которые были на момент return, или return с параметрами.

```
func millenium() (day int, month string, year int) {
  day=1
  month="Январь"
  year=1000*2
  return
} // функция вернет 1, "январь", 2000
```

# Вызов функции

Для вызова функции мы указываем её имя и в круглых скобках перечисляем передаваемые в функцию значения. Если передаваемых значений нет, оставляем скобки пустыми.
Передаваемые значения должны соответствовать типам в сигнатуре функции.

В функцию ВСЕГДА передаются копии тех значений, что мы передали при вызове. Поэтому в большинстве случаев из функции нельзя изменить те переменные, которые в неё переданы (если это не указатели).

# Вызов функции (продолжение)

```
package main

import "fmt"

func main() {
   value:=10
   change(value) //вызов функции
   fmt.Println(value)
   //value по-прежнему равняется 10
}

func change(a int){
   a=a+1
}
```

# Результат вызова функции

Если у функции есть возвращаемое значение, то результат вызова функции можно сохранить в переменную или переменные.
Например функцию div можно вызвать несколькими способами

```
func div(a, b int) (int, int) {
  return a/b, a%b
}

func main() {
  div(100, 3)        //игнорируем результат
  d, r := div(100, 3) //сохраняем оба результата
  _, r := div(100, 3) //сохраняем только второй параметр
  d, _ := div(100, 3) //сохраняем только первый параметр
  _, _ = div(100, 3)  //ничего не сохраняем. Заметьте – без :
}
```

# Результат вызова функции (продолжение)

Результаты вызова одних функций могут становиться аргументами для вызова других функций.
Важно сохранять человекочитабельность кода.

```
package main

func main() {
  d, r := div(add(25, 75), 3)       //это еще нормально
  d, r = div(add(25, 75), add(2, 1)) //такое читается уже хуже
}

func add(a, b int) int {
  return a + b
}

func div(a, b int) (int, int) {
  return a / b, a % b
}
```

# Обычный набор возвращаемых значений

Как правило, функции возвращают ни одного, одно либо два значения.
Причем одним из возвращаемых значения является либо булевое ok, являющееся признаком того, что второе значение имеет смысл, либо значение типа error, содержащее в себе информацию о возникшей ошибке.
Подробнее об этом в специальном занятии.

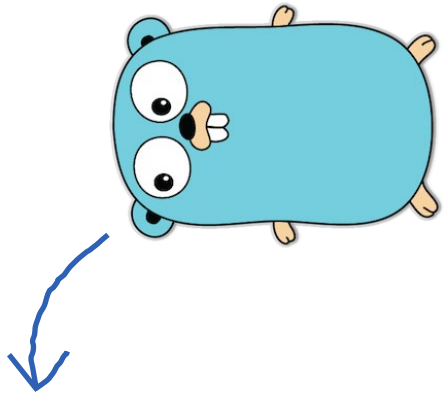
```
// функция возаращает результат деления a/b
//если b=0, ok=false, иначе ok = true
func div(a, b int) (quotient int, ok bool) {
   //пока без реализации
}

//функция save что-то сохраняет.
//Если неудачно это будет видно по err
func save() (err error){
   // это как-то работает
}
```

# Содержание занятия

Общая информация о синтаксисе

Переменные

Функции



**Управляющие конструкции**

# Управляющие конструкции языка

В отсутствие управляющих конструкций команды в блоке выполняются в соответствии с тремя правилами:

- Сверху вниз
- Подряд
- Однократно.

Это «линейный участок кода»

Чтобы у нас появилась возможность выполнять не все команды, либо выполнять какие-то команды более, чем один раз, нам нужны управляющие конструкции.

# Управляющие конструкции языка

## Сейчас рассмотрим:

- If без else
- If с else
- If с вызовом многозначной функции
- Switch по значению
- Switch по условию
- default, fallthrough
- Вечный for
- Традиционный for
- Break, continue
- Необычные варианты for

# If без else

Конструкция If позволяет выполнять или не выполнять фрагмент кода в зависимости от значения управляющего выражения.
Управляющим выражением может быть bool-переменная, выражение, возвращающее значение bool, или функция, возвращающая значение bool.

Выражения, возвращающие bool, это в частности операторы сравнения:
==
!=
>
=>
<
<=

# If без else (продолжение)

В этом примере условие в выражении первого if не выполняется, поэтому в тело первого if вход выполнен не будет.

```
var i int = 3
a := 10    //эта строка выполнится
if i > 5 { //условие не выполняется
   a = a + 10                    //эта строка будет пропущена
   fmt.Println("А ведь i больше, чем 5!") //будет пропущена
}
a = a + 10 //эта строка выполнится, она за пределами тела if

if a<=20 { //это условие выполняется
   a=a+50 //строка будет выполнена
}
```

А в тело второго if мы войдем, т.к. его управляющее условие выполняется.

# If без else (продолжение)

Предположим, у нас есть функция isCold() bool, которая каким-то образом определяет, холодно ли сегодня.
Можем её тоже использовать как управляющее выражение.

```
if isCold() {
  fmt.Println("Сегодня холодно")
}
```

Для управляющих выражений if очень характерно использование логических операций &&, ||, !

```
if isCold() && isRainy() {
  fmt.Println("Гулять не идем")
}
```

# If c else

If без else используется, чтобы «включить» или «выключить» фрагмент кода.
Конструкция else позволяет управлять выполнением сразу двух блоков, из которых выполнится лишь один.

Соответствует дихотомии в логике.

Если управляющее выражение истинно, выполнится первый блок. Его называют true-блок, true-part
Если управляющее выражение ложно, то выполнится else-блок, else-part.

```
if temperature>20 {
  fmt.Println("Тепло или жарко")
} else    {
  fmt.Println("Прохладно или холодно")
}
```

# If c else, else if

После else может идти не блок кода, а сразу еще один if (который будет иметь свою true-part и false-part)
Это позволяет делать каскадные проверки условий.
Читается это не очень хорошо, но встречается часто.

```
if temperature > 20 {
   fmt.Println("Тепло или жарко")
} else if temperature > 10 {
   fmt.Println("Прохладно")
} else if temperature > 0 {
   fmt.Println("Холодно")
} else {
   fmt.Println("Вообще мороз")
}
```

Здесь последний else относится к if temperature>0

# If c else, else if (продолжение)

Но это читается лучше, чем вложенные if одновременно в true-part и false-part. Запишем аналогичный код другим способом.

```
if temperature>10 {
  if temperature>20{
    fmt.Println("Тепло или жарко")
  } else {
    fmt.Println("Прохладно")
  }
} else {
  if temperature>0 {
    fmt.Println("Холодно")
  } else {
    fmt.Println("Мороз")
  }
}
```

Читается не очень хорошо. Возможно даже хуже.

# If с вызовом многозначной функции

Особый синтаксис if – это когда мы сначала вызываем функцию, сохранив результат её вызова, а потом строим управляющее выражение.
Представим, что у нас есть функция div, которая делит два числа, возвращает частное и признак того, удалось ли разделить.

```
func div(a, b int) (quotient int, ok bool)
```

Можем её использовать прямо в if

```
var cakes, kids:=6, 3
if d, ok:=div(cakes, kids); ok {
  fmt.Print("Каждому ребенку достанется пирожных: ")
  fmt.Println(d)
}
```

Вызов от управляющего выражения отделяется ;

# Switch

Конструкция switch призвана использоваться в случаях, когда должна наступить одна из многих ситуаций.
Можно то же самое написать при помощи if и else if, но код со switch выглядит нагляднее.

Конструкция switch содержит несколько случаев, case’ов, которые проверяются подряд сверху вниз. Как только будет найден наступивший случай, остальные уже не будут проверяться (но у этого утверждения есть свои особенности)

Существует три вида сильно отличающихся switch’ей, из которых мы сегодня рассмотрим два

# Switch по значению

В этом случае в самом switch’е указывается переменная, или выражение, значение которого мы будем искать в case’ах.
А case’ы будут содержать варианты этих значений.
В зависимости от вида домашнего животного, назначим ему кличку:

```
var pet = "собака"
var petName string
switch pet {
case "кот":
  petName = "Мурзик"
case "собака":
  petName = "Тузик"
case "попугай":
  petName = "Кеша"
}
```

# Switch по значению (продолжение)

Еще один пример.
Мы можем работать с разными, почти любыми типами

```
func main() {
  var day = 3
  var dayName string
  switch day {
  case 1:
    dayName = "Понедельник"
  case 2:
    dayName = "Вторник"
  case 3:
    dayName = "Среда"
  case 7:
    dayName = "Воскресенье"
  }
  fmt.Println(dayName)
}
```

# Switch по значению (продолжение)

Интересно, что можно в одном case, комбинировать несколько случаев:

```
var day = 3
var dayDescription string
switch day {
  case 1, 2, 3, 4, 5 :
    dayDescription = "Будний день, работаем"
  case 6, 7:
    dayDescription = "Выходной, отдыхаем"

}
fmt.Println(dayDescription)
```

# Switch по условию

Другой формат switch, когда каждый case – это самостоятельное bool'условие. То условие, которое выполнится первым, определит case, который выполнится.

```
var childAge1, childAge2=10, 11
switch {
case childAge1==childAge2:
  fmt.Println("Дети ровесники")
case (childAge1==childAge2-1)||(childAge2==childAge1-1):
  fmt.Println("Дети погодки")
}
```

# Switch (продолжение)

Код внутри каждого case может быть и многострочным.
Количество case’ов неограниченно.

Нужно сохранять благоразумие, чтобы не сделать код неподдерживаемым и неудобным.

# Default:

В конце switch может существовать особый случай default: который выполнится в том случае, если не выполнится никакой другой.

```
var pet = "змея"
var petName string
switch pet {
case "кот":
  petName = "Мурзик"
case "собака":
  petName = "Тузик"
case "попугай":
  petName = "Кеша"
default:
  fmt.Println("Такое животное мы заводить не будем")
}
fmt.Println(petName)
```

# fallthrough

Если в конце case’а указать ключевое слово fallthrough, то обработчик этого случая после завершения провалится в обработчик следующего случая. Это применяется довольно редко.

```
var money = 10000
switch {
case money > 50000:
  fmt.Print("можем купить подарок")
  fallthrough
case money > 1000:
  fmt.Println("можем купить торт")
  fallthrough
case money > 100:
  fmt.Println("можем купить открытку")
default:
  fmt.Println("не можем купить ничего")
}
```

Каждый fallthrouth действует ровно на 1 следующий case

# Цикл for

В Go существует только один вид циклов – for
Но у него несколько различных синтаксисов, позволяющих реализовывать вечный цикл, традиционный цикл, аналог цикла while и некоторые экзотические варианты.

Также существует цикл for range, для перебирания элементов составных структур, который будет изучаться на следующем занятии.

```
func main() {
  var a int
  for {
    a++
    fmt.Println(a)
  }
}
```

# Вечный for

Вечный же for выглядит предельно просто:

```
func main() {
  var a int
  for {
    a++
    fmt.Println(a)
  }
}
```

Блок кода, заключенный между { } for называется «тело цикла». Каждое выполнение тела цикла называется «итерацией».

С вечным циклом нужно быть внимательным: его возникновение может быть как следствием ошибки, так и частью правильно реализованной логики программы.

# Традиционный for

Традиционный for похож на аналогичную конструкцию в C.
Он состоит из трех секций:

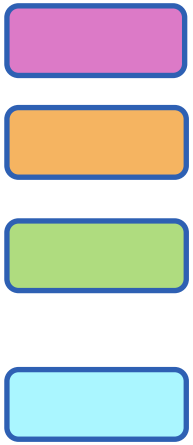
1) действия, которое выполняется однократно перед первой итерацией.
2) Условия, которое проверяется ПЕРЕД каждой итерацией. Если условие нарушится, цикл прекратится.
3) Действия, которое выполняется ПОСЛЕ каждой итерации.

Следовательно, тело цикла for может в разных ситуациях выполняться разное количество раз, в т.ч. бесконечность (если условие не может нарушиться), в т.ч. ни разу (если условие нарушено еще перед первой итерацией)

# Традиционный for (продолжение)

Обычно традиционный for используется для приращения некоего счетчика от минимального значения до максимального. Но это не единственное его применение.

```
func main() {

  fmt.Println("Скоро войдем в цикл")

  for i:=0; i<3; i++ {
    fmt.Println("Мы в цикле")
  }

  fmt.Println("Мы вышли из цикла")
}
```



- Действие перед циклом
- Условие продолжения цикла
- выполняется ПОСЛЕ каждой итерации
- Управляемый блок кода (тело цикла)

# Break, Continue

Конструкция break позволяет немедленно прервать цикл.
Конструкция continue позволяет немедленно прервать текущую итерацию цикла.

Прерывание итерации – это немедленный переход на позицию «после последней строки цикла», как будто выполнение тела дошло до конца

```
for i := 0; i < 10; i++ {

  if i == 3 {
    continue
  }
  if i > 6 {
    break
  }
  fmt.Println(i)
}
```

Попробуем предположить, какие числа будут выведены на экран?

# Необычные варианты for

Каждое из трех составляющих выражения for может отсутствовать. Например отсутствие первого и последнего выражения превращает for в аналог while

```
var i=10
for  i >0 {
  i=i-3
  fmt.Println(i)
}
```

Какие числа мы увидим на экране?

Также существуют варианты for без условия, без приращения, без инициализации и т.д.

# Заключение

- Оказывается, основы синтаксиса GO можно изложить за 1,5-2 часа.
- Для понимания может потребоваться дополнительное время.
- Некоторые элементы синтаксиса неразрывно связаны со специальными понятиями языка и могут быть изучены только вместе с ними позже.

# Самое время для вопросов

Ставим “+”,
если вопросы есть

Ставим “–”,
если вопросов нет

# Минутка рефлексии

# Насколько мы освоили тему?

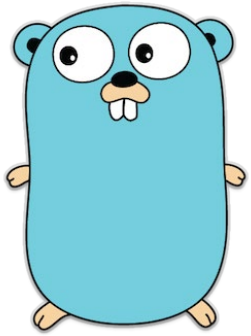
**Общая информация о синтаксисе**

**Переменные**

**Функции**

**Управляющие конструкции**

? ? ?



Спасибо за внимание

# Приходите на следующие вебинары

**Рубаха Юрий**

# Заполните, пожалуйста, опрос о занятии по ссылке в чатботе