Основы. Знакомство с языком

Установка и настройка компилятора Go, текстового редактора Visual Studio Code, пакеты и импорт, переменные, фундаментальные типы данных.

## **Оглавление**

[Установка Golang. Утилиты](#_6cvo062wtmvz)

[Скачиваем и устанавливаем Go](#_k5kzv6461oh4)

[Для Windows](#_mk70ne6lb3mc)

[Для Mac OS](#_36r3lc80s3mc)

[Для Linux](#_rexal5gjxzmt)

[Настраиваем GOPATH. Обсуждаем организацию кода](#_wbd03g31b6t9)

[На Windows](#_2ujo2tp2ao6)

[Для Mac OS и Linux](#_9ryo9ygaqx07)

[Проверка установки](#_2cwbr5zhwkyn)

[Обзор инструментов командной строки go: run, build, get, install](#_5bxcxix2ncr2)

[Установка и настройка Visual Studio Code](#_lgzrgg4oaiut)

[Обзор основных настроек и панелей редактора](#_f3csrmx4r2sf)

[Устанавливаем и настраиваем плагин Go](#_jh6kfv3w0q77)

[Запускаем Hello, world!](#_zbekasep7hw7)

[Запуск первой программы](#_ayaw4a3p4unn)

[Создаем проект курса](#_4n9hzb1b4a8x)

[Пакеты и импорт](#_ces88anhkvte)

[Импортируем пакеты из стандартной библиотеки и свои пакеты](#_tdxwx6vdsmq2)

[Пакет fmt](#_gtzaw27vebtc)

[func Println(a …interface{}) (n int, err error)](#_he1p9mgm2fe0)

[func Scan(a ...interface{}) (n int, err error)](#_t2c4esabm0f2)

[Пакет strconv](#_bpmm6z3dbkja)

[func Atoi(s string) (int, error)](#_i25wwby7kvvy)

[Пакет math](#_qjew3dwv27ds)

[func Pow(x,y float64) float64](#_a1c7f9fumdoj)

[func Sqrt(x float64) float64](#_wwi2ht1zz36t)

[Создание собственного пакета](#_zok2byymxj8)

[Инициализация пакета: разбираем метод init](#_gz483k3ub2i0)

[Обсуждаем область видимости внутри и вне пакета](#_98xic6yelo5m)

[Переменные и константы](#_5l4pdxa5nytv)

[Переменные](#_di3e9pe93dji)

[Именование](#_rgi5w5u4wpto)

[Области видимости](#_63vv2ek6drns)

[Зарезервированные слова](#_dfuptjx6alm0)

[Разбираем способы объявления переменных](#_yj5r3bcdui9i)

[Обсуждаем указатели и функцию new](#_6uosw4xgol5f)

[Время жизни переменных](#_a2oyym3fq3zr)

[Разбираем константы](#_q66we5tt0rk)

[Фундаментальные типы данных](#_59p2fjsiviml)

[Разбираем целые числа, типы, операции над целыми числами](#_db8cwfi8ii3r)

[Разбираем числа с плавающей точкой](#_j23hvd4chlpa)

[Разбираем логические значения и сравнения](#_q2q67f70pjda)

[Разбираем строки](#_st0jbpgvxn3q)

[Операторы для работы с данными](#_1vfyov2uo9bw)

[Арифметические операторы](#_657djbdbe160)

[Реляционные операторы](#_icy4mrx6rpqk)

[Логические операторы](#_2gsl37duyvlj)

[Побитовые операторы](#_x46xq0nz9s3f)

[Операторы присваивания](#_5ukz31gdgm6)

[Практическое задание](#_4i7ojhp)

[Дополнительные материалы](#_2xcytpi)

[Используемая литература](#_1ci93xb)

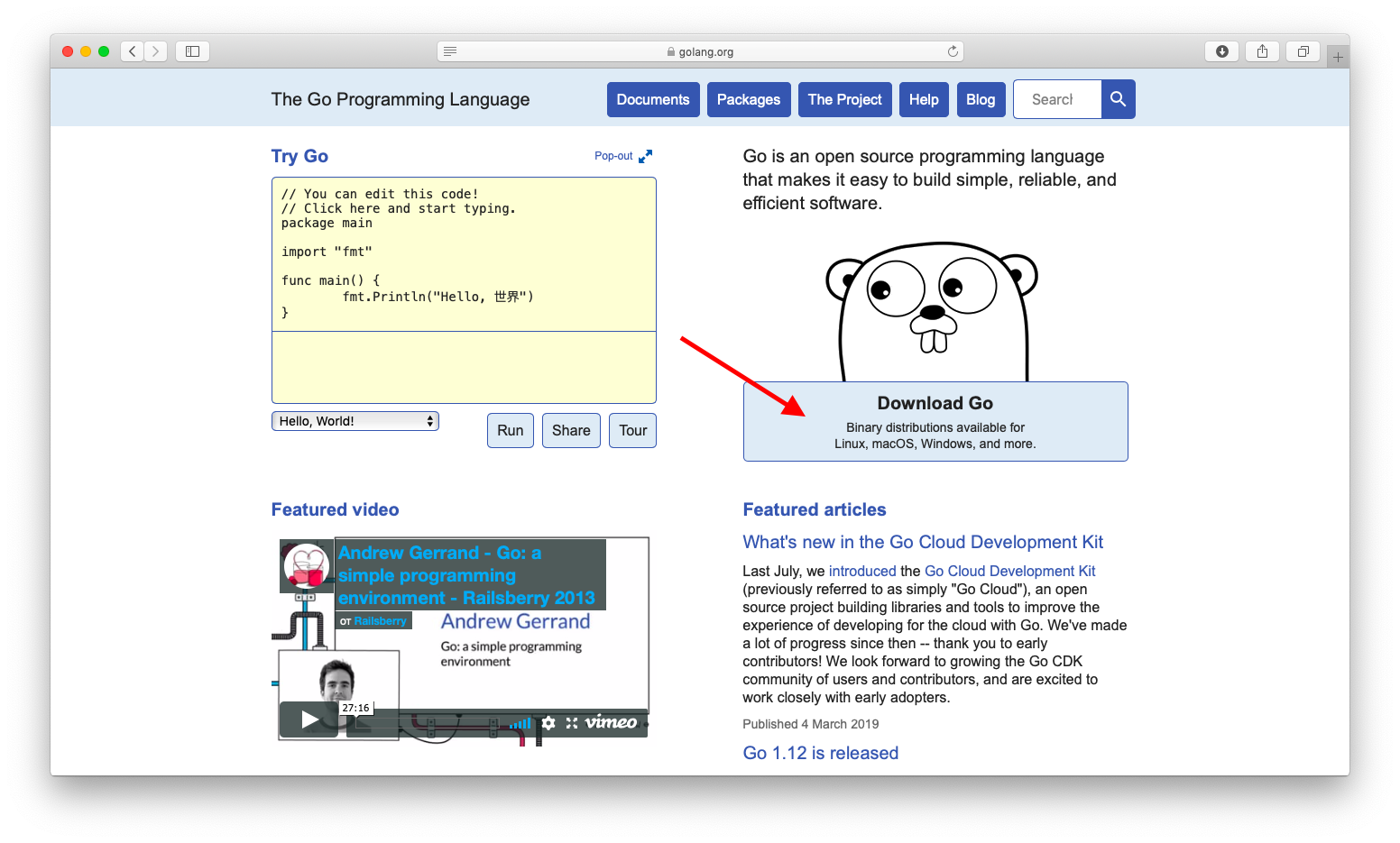
# 

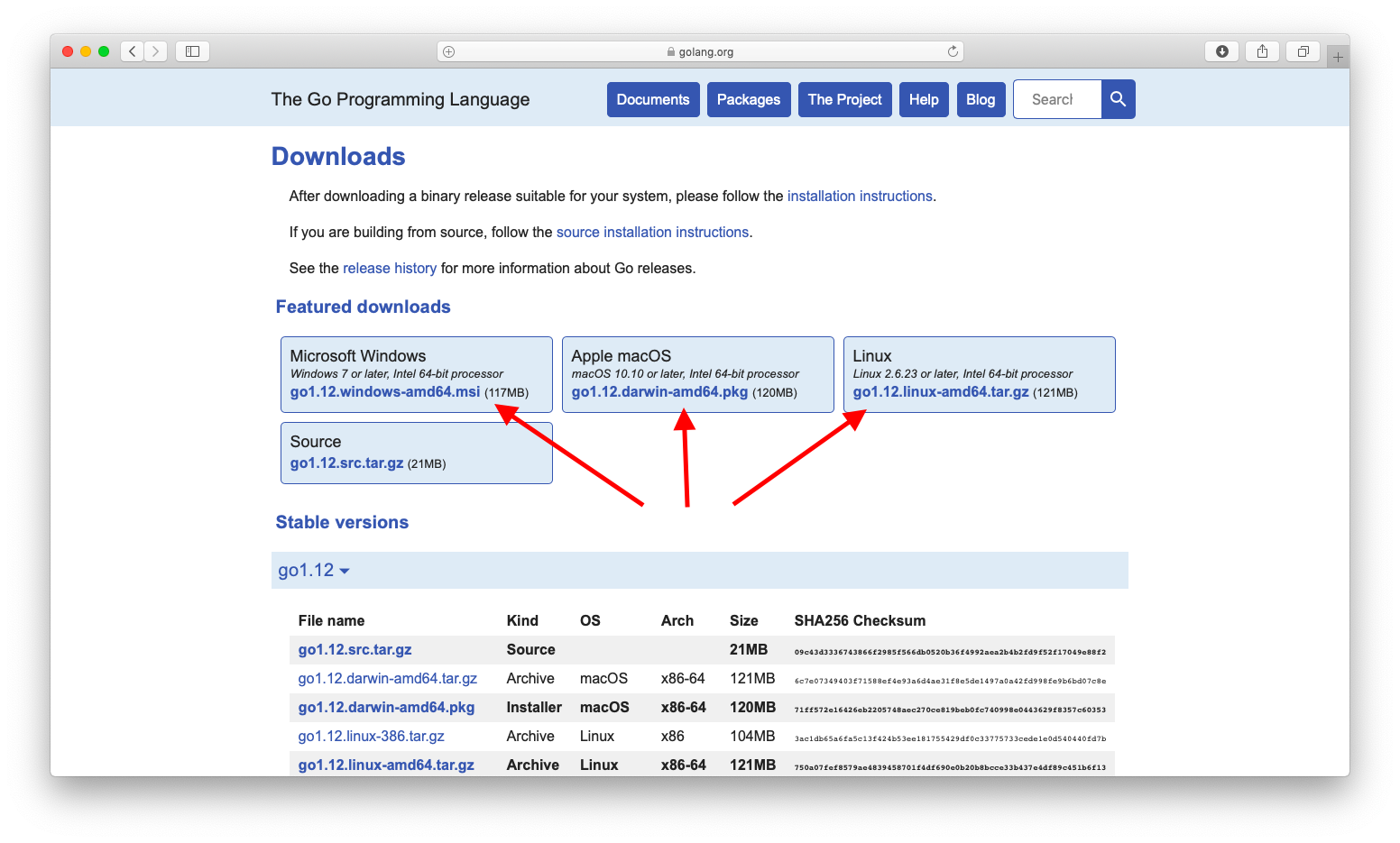
# Установка Golang. Утилиты

Go — это компилируемый язык программирования, поэтому работа с ним начинается со скачивания и настройки компилятора.

## Скачиваем и устанавливаем Go

Прежде всего необходимо зайти на [сайт языка Go](https://golang.org/) и скачать установочный файл для вашей операционной системы.





### Для Windows

В случае Windows установка проходит так же, как и для любого компонента операционной системы:

* после скачивания вы получите файл **go1.12.linux-amd64.msi**;
* запустите его и пройдите этап установки языка — местоположение для нее не имеет значения.

Для корректной работы всех инструментов Go также потребуется установить консольный клиент для Git.

Скачать его можно [на сайте git-scm.com](https://git-scm.com/download/win). Установка со стандартными параметрами вполне подойдет.

### Для Mac OS

Установка будет даже проще, чем на Windows:

* после скачивания запустите файл **go1.12.linux-amd64.pkg**;
* пройдите установку.

Как и в случае с Windows, Git-клиент тоже потребуется, но Mac OS сама предложит его установить при первой необходимости.

### Для Linux

|  |
| --- |
| cd ./Downloads tar -C /usr/local -xzf go1.12.linux-amd64.tar.gz echo "export PATH=$PATH:/usr/local/go/bin" >> $HOME/.bash\_profile echo "export GOPATH=$HOME/go" >> $HOME/.bash\_profile sudo apt-get install git |

На Linux необходимо открыть терминал и ввести набор команд, представленный ниже. Разберем их по порядку:

1. **cd Downloads** — переведет вас в каталог с загрузками.
2. **tar -C /usr/local -xzf go1.12.linux-amd64.tar.gz** — распакует содержимое архива в **/usr/local**.
3. **echo "export PATH=$PATH:/usr/local/go/bin" >> $HOME/.bash\_profile** — добавит папку с компилятором Go в список директорий с исполняемыми файлами приложений.
4. **echo "export GOPATH=$HOME/go" >> $HOME/.bash\_profile** — создаст путь для сохранения исходных файлов программ и пакетов, а также скомпилированных из них файлов.
5. **sudo apt-get install git** — установка инструментов для работы с Git. Во время установки утилита **apt-get** составит список необходимых пакетов и попросит подтверждения для их установки.

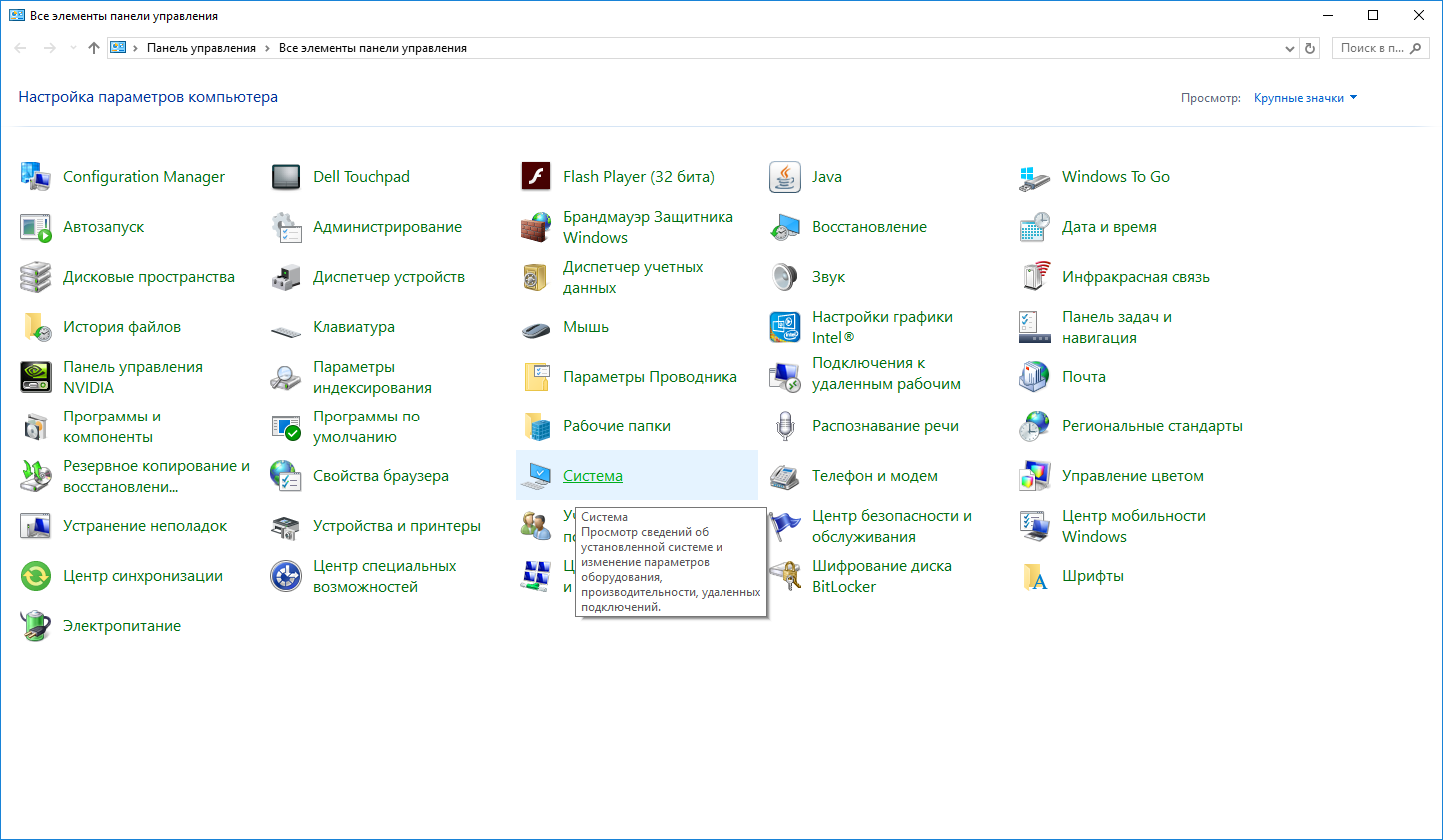
## Настраиваем GOPATH. Обсуждаем организацию кода

GOPATH — это переменная окружения, которая ведет к каталогу. В нем будут храниться скачанные вами в процессе работы с Go исходные коды пакетов, а также скомпилированные библиотеки и исполняемые файлы.

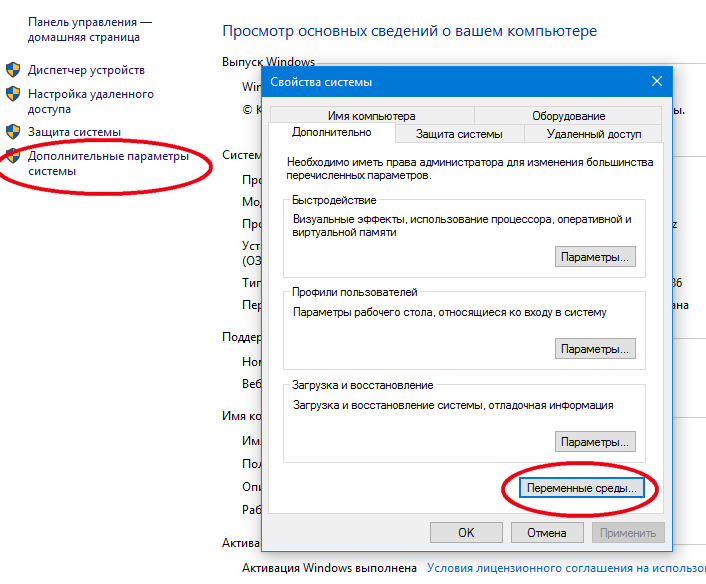
По умолчанию Go будет использовать для этих целей каталог **go** в корневой папке пользователя системы, из которого ведется работа. Можно изменить его местоположение на удобное вам.

## На Windows

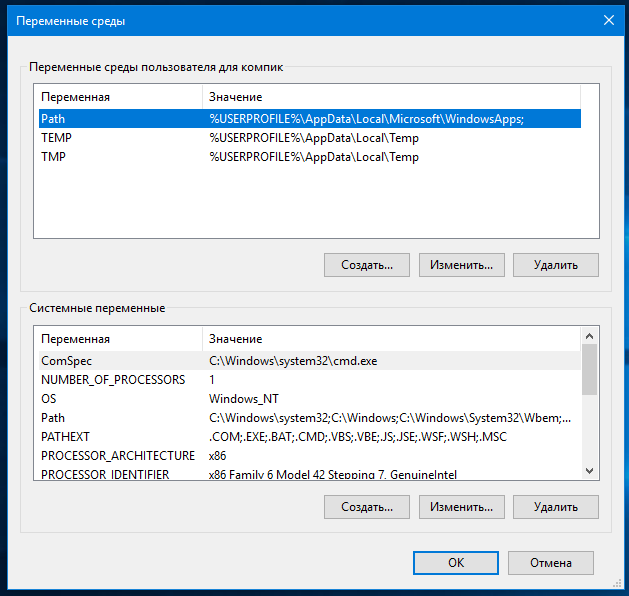
Необходимо открыть **панель управления** и зайти в раздел **«Система»**:



Далее — перейти раздел «Дополнительные параметры системы», в открывшемся окне — на вкладку «Дополнительно» и нажать на кнопку «Переменные среды»:



В переменных средах для вашего пользователя создайте GOPATH и настройте путь к папке, в которой хотите хранить данные. Можно выбрать любое местоположение для директории GOPATH. Главное, чтобы запись и чтение могли проходить без прав администратора. К примеру, можно создать папку в директории «Мои документы».



### Для Mac OS и Linux

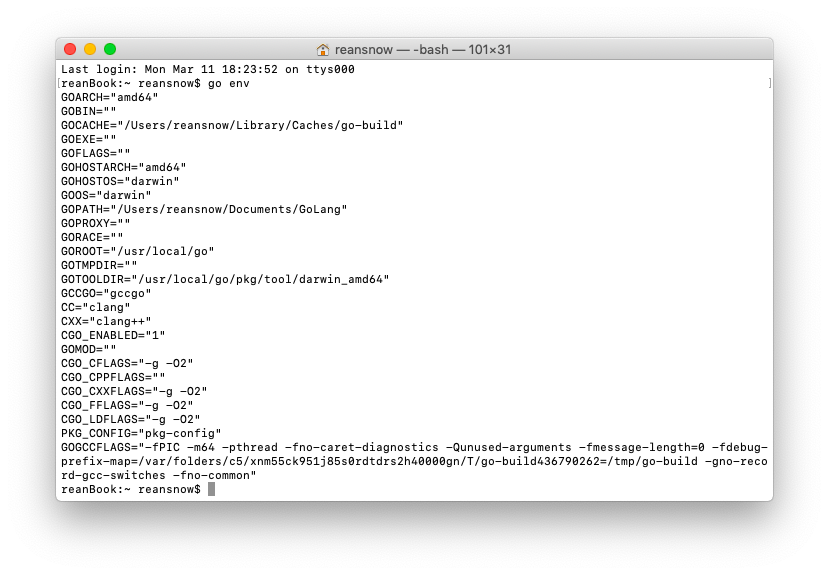
|  |
| --- |
| echo "export GOPATH=$HOME/go" >> $HOME/.bash\_profile |

Изменение директории для GOPATH делается проще. Нужно зайти в терминал и ввести команду, заменив в ней стандартное положение «$HOME/go» на нужное вам. Не забудьте перезапустить терминал, чтобы применить изменения.

### Проверка установки

Для проверки установки откройте терминал или, в случае с Windows, командную строку (можно и powershell). Также подойдут любые интерпретаторы bash (например, git-bash). Затем наберите следующую команду:

|  |
| --- |
| go env |

Результат выполнения — на скриншоте. В случае ошибки следует проверить, правильно ли выполнены предыдущие шаги.

## Обзор инструментов командной строки go: run, build, get, install

* **Run** — команда, которая запустит ваш файл без сохранения скомпилированного результата. Пример использования:

|  |
| --- |
| go run myfile.go |

* **Build** — скомпилирует программу, полученный файл по умолчанию разместит в рабочей папке:

|  |
| --- |
| go build myfile.go |

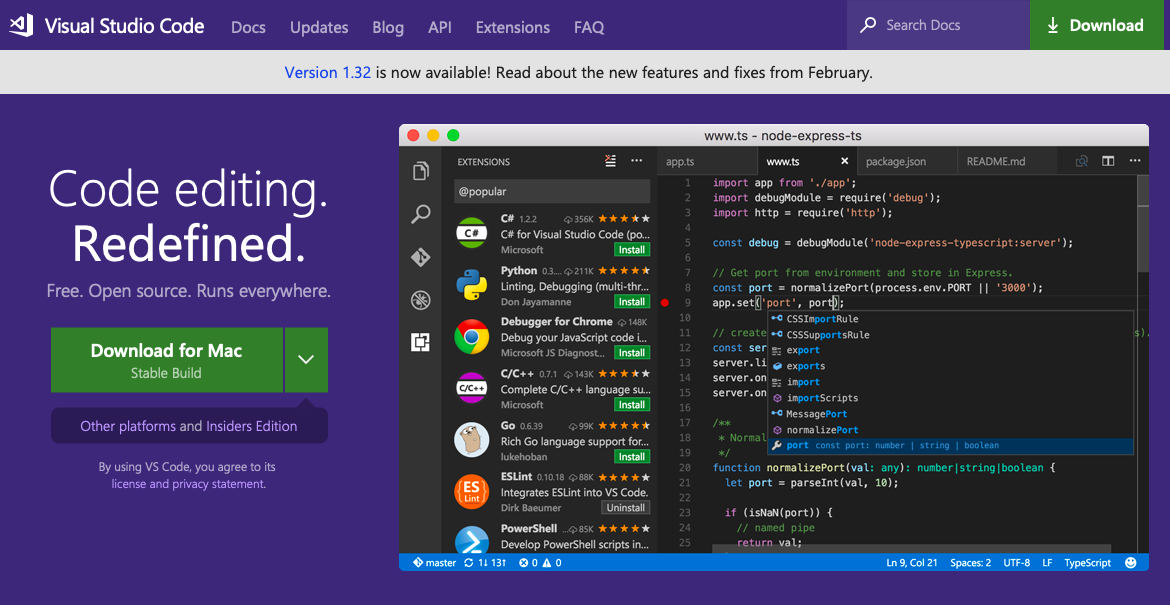
* **Get** — загрузит пакет и зависимости для его работы, разместит файлы в папке **src** внутри GOPATH:

|  |
| --- |
| go get github.com/go-chi/chi |

* **Install** — произведет компилирование программы или пакета. Чтобы команда сработала, необходимо перейти в папку проекта внутри GOPATH. Скомпилированный пакет будет расположен в папке **pkg** в GOPATH, а исполняемый файл — в папке **bin** в GOPATH:

|  |
| --- |
| go install |

# Установка и настройка Visual Studio Code

Чтобы удобно работать с языком Go, рекомендуем текстовый редактор Visual Studio Code. Есть вариант и лучше, например intellij GoLand, а так же множество намного более посредственных, но преимущество Visual Studio Code — в бесплатности.

Скачать Visual Studio Code можно с [официального сайта](https://code.visualstudio.com/). Ваша ОС будет определена автоматически: жмем Download и производим установку.

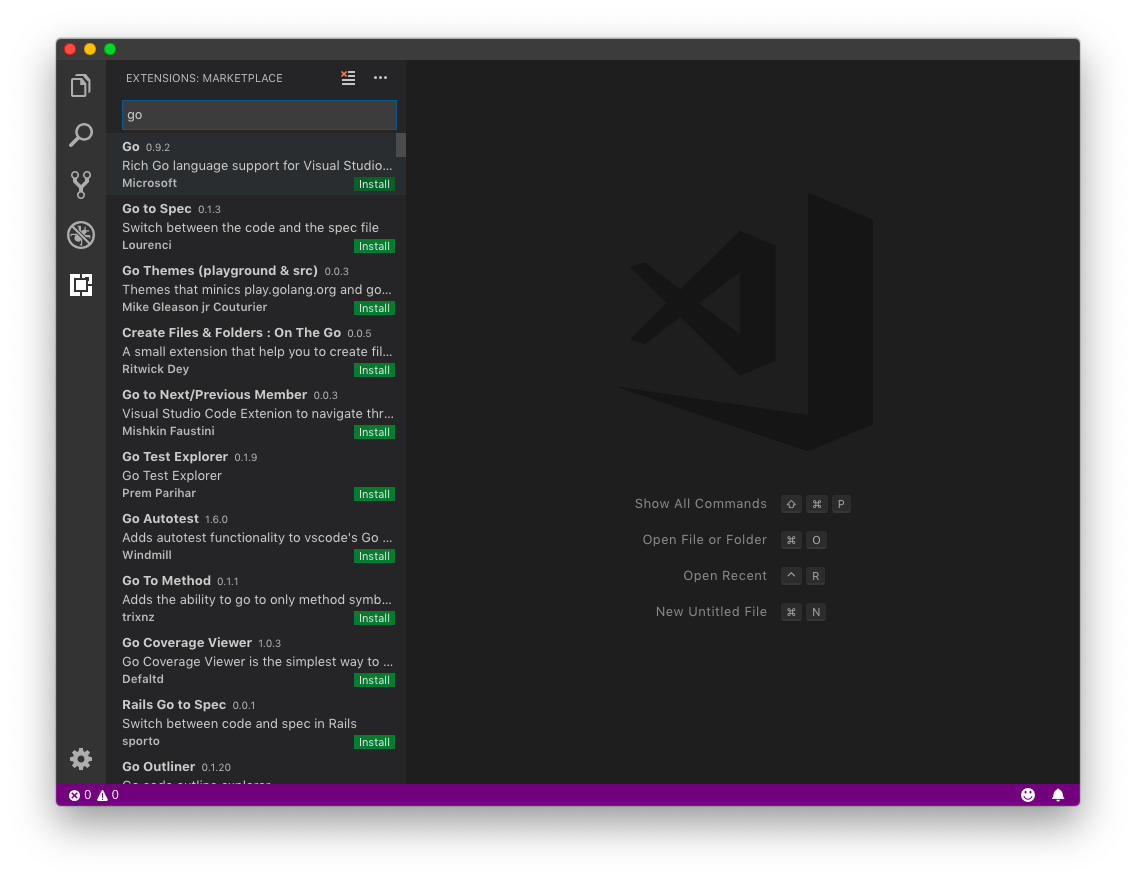
Для Mac файл скачивается в архиве .zip, автоматически разархивируется — и его можно использовать. Но чтобы с приложением было удобно работать, рекомендуем переместить Visual Studio Code в Finder > Программы.

## Обзор основных настроек и панелей редактора

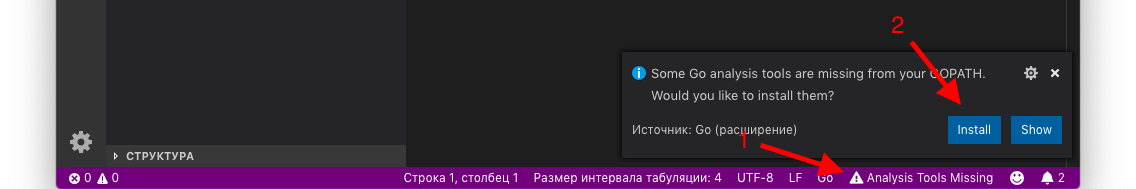
Главное окно редактора выглядит, как на скриншоте, — с небольшими отличиями в зависимости от операционной системы.

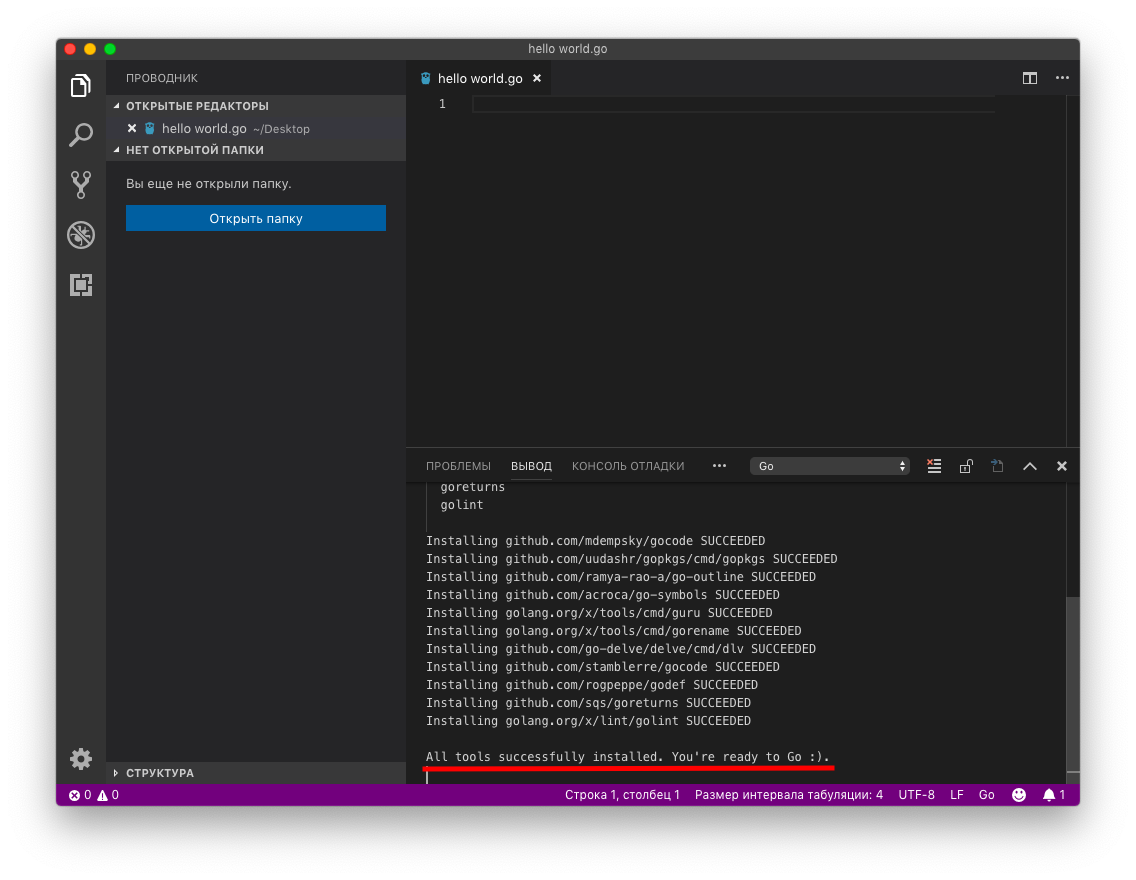
* Первый пункт бокового меню — это проводник. Он позволит работать с директориями, а также переключаться между файлами.
* Далее — поиск, чтобы искать отрывки кода и быстро их заменять при необходимости.
* Контроль версий поможет при работе с Git.
* Отладка позволяет произвести быстрый запуск приложения, провести тесты и найти ошибки.
* С расширениями легко дополнить редактор необходимой функциональностью, изменить внешний вид и добавить сочетания клавиш из других редакторов.
* Настройки позволят сконфигурировать редактор так, как вам потребуется.

## Устанавливаем и настраиваем плагин Go

1. Для этого нужно открыть редактор.
2. Затем перейти на вкладку расширений и в появившемся поисковом поле набрать «go».
3. Нажать кнопку Install возле дополнения «Go» и дождаться завершения установки.
4. По желанию можно установить пакет Russian Language Pack.
5. После установки следует перезагрузить редактор.

Осталось только завершить установку дополнительных инструментов.

* Откроем редактор и создадим новый файл комбинацией клавиш ctrl+n или cmd+n.
* Сохраним файл в произвольное местоположение комбинацией клавиш ctrl+s или cmd+s под названием **hello world.go**.
* После сохранения файла редактор начнет относиться к файлу как к исходному коду на языке Go, а справа внизу появится уведомление о том, что отсутствуют инструменты для анализа кода.
* Нажимаем на уведомление и в открывшемся диалоговом окне жмем **install**.
* Ждем, пока закончится загрузка инструментов.

Видим надпись о том, что все установлено удачно.

## 

## Запускаем Hello, world!

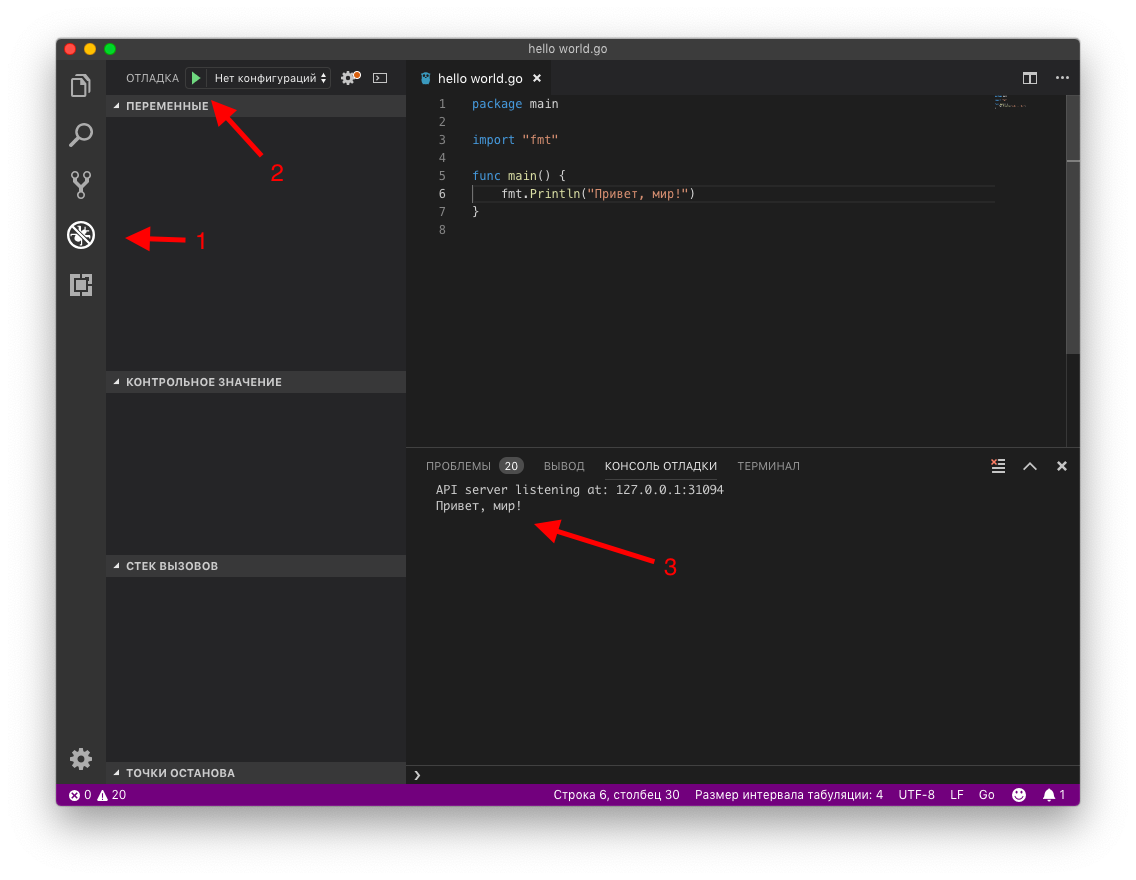
В файле, который был создан ранее, напишем следующий код, сохраним его и разберем, из чего он состоит:

|  |
| --- |
| package main  import "fmt"  func main() {  fmt.Println("Привет, мир!") } |

1. **package main** — это объявление имени пакета. Для исполняемых файлов оно всегда будет «main», а для библиотек будет соответствовать названию директории, в которой находится файл.
2. **import "fmt"**. import — это ключевое слово для включения сторонних библиотек в вашу программу. fmt — название пакета, который будет использоваться в программе.
3. **func main () {}** — точка входа в программу. Здесь начинается ее выполнение.
4. **fmt.Println("Привет, мир!")** — вызов функции Println из пакета fmt. Она выведет в консоль сообщение «Привет, мир!».

## Запуск первой программы

Теперь стоит запустить первую программу, выполнив три простых действия.

1. Откроем вкладку Debug в VS Code:

2. Нажмем на кнопку Start Debugging (маленький зеленый треугольник).

3. Посмотрим на результат выполнения в открывшемся окне.

# Создаем проект курса

После успешного запуска первой программы пришло время подготовить место для работ по курсу.

Зайдем в директорию GOPATH (стандартное местоположение папки GOPATH — это «путь\_к\_папке\_вашего\_пользователя/go») и посмотрим на ее содержимое. Сейчас внутри две директории, но в дальнейшем появится и третья.

* **bin** — это папка, в которой будут храниться скомпилированные экземпляры программ;
* **pkg** — папка, которая служит для хранения скомпилированных пакетов;
* **src** — место для хранения ваших исходных кодов, а также исходных кодов, скачанных посредством go get.

Чтобы начать работу, потребуется создать рабочую директорию в папке **src** — назовем ее **GoCourse**.

Необходимо перейти в созданную папку проекта и инициализировать репозиторий.

|  |
| --- |
| cd ~/go/src  mkdir GoCourse  cd GoCourse  git init git remote add origin <Ссылка\_на\_созданный\_репозиторий\_в\_Github>.git  git push -u origin master |

Важно! Последняя команда не сработает, если в локальной ветке **master** будет пусто. Так что стоит создать первый коммит: пусть в нем будет файл **README.md**, можно оставить его пустым.

Откроем созданную папку в редакторе.

Практические задания будем сдавать через Pull Request в GitHub, так что в директории нужно будет инициализировать репозиторий, а также подключить его к удаленному репозиторию в GitHub набором команд.

В дальнейшем сдача практически будет проходить так:

* Создается ответвление из **master** в ветвь с названием **homework-\*** (вместо звездочки — номер урока).
* В новой ветке создается папка с вашим заданием **homework-\***.
* В этой папке размещается практическая работа.
* После выполнения задания создается коммит в ветку **homework-\*.**
* Создается pull request из ветки **homework-\*** в **master**. Ссылка на него сдается на странице курса на сайте GeekBrains.

# Пакеты и импорт

Язык Go способствует применению инженерной практики DRY, или Don't Repeat Yourself (не повторяйтесь). В этом помогают пакеты. Любой файл с исходным кодом на языке Go содержит имя пакета, к которому он принадлежит, а также список пакетов, которые он будет использовать для своей работы.

|  |
| --- |
| package main |

При создании **Hello, world!** код имел название пакета **main** — так именуются пакеты, которые могут быть скомпилированы в конечную программу. Пакет с любым другим именем будет скомпилирован в байт-код для последующего переиспользования в приложениях.

**Важно!** Внутри одной директории может находиться несколько файлов с исходным кодом. Каждый из них должен иметь одно и то же имя пакета, иначе программа не скомпилируется.

|  |
| --- |
| github.com/peterbourgon/foo/  main.go *// package main*  main\_test.go *// package main*  foo/  foo.go *// package foo*  foo\_test.go *// package foo*  *github.com/peterbourgon/foo*  *foo.go // package foo*  *foo\_test.go // package foo*  *cmd/*  *foo/*  *main.go // package main*  *main\_test.go // package main* |

Два варианта, как может выглядеть импорт сторонних пакетов:

|  |
| --- |
| *// Импорт пакетов по одному* import "fmt" import "log"  *// Импорт нескольких пакетов* import (  "fmt"  "log" ) |

## Импортируем пакеты из стандартной библиотеки и свои пакеты

Стандартная библиотека Go содержит множество пакетов — весь список можно посмотреть на сайте языка в [разделе Packages](https://golang.org/pkg/). Сейчас мы разберем два пакета, которые понадобятся для выполнения практического задания.

## Пакет fmt

Пакет fmt служит для выведения информации в консоль, а также чтения из нее. Рассмотрим несколько функций из него.

### func Println(a …interface{}) (n int, err error)

**Println** служит для вывода в консоль переданных в нее аргументов — их может быть неограниченное количество. Между переданными аргументами будут автоматически проставлены пробелы. Вернет число байт, записанных в консоль, а также ошибку в случае ее возникновения.

|  |
| --- |
| package main  import "fmt"  func main() {  var name = "Юля"  var age = 22  fmt.Println("Имя:", name, "Возраст:", age) } |

### func Scan(a ...interface{}) (n int, err error)

**Scan** сканирует текст из стандартного ввода (командной строки, терминала), а также сохраняет значение в переданные в функцию аргументы. Возвращает количество успешных сканирований, а также ошибку, если она возникнет.

|  |
| --- |
| package main  import "fmt"  func main() {  var name string  fmt.Println("Как тебя зовут?")  fmt.Scanln(&name)  fmt.Println("Привет,", name) } |

Обратите внимание, что в этом примере в восьмой строке перед именем переменной стоит символ & — это передача указателя в качестве аргумента. В языке Go передача аргументов в функцию может происходить по значению и по указателю. В чем разница, вы узнаете позже, а пока просто запомните.

## Пакет strconv

### func Atoi(s string) (int, error)

**Atoi** сконвертирует строку в десятичное число.

|  |
| --- |
| package main  import (  "fmt"  "log"  "strconv" )  func main() {  str := "12"  num, err := strconv.Atoi(str)  if err != nil {  log.Fatalln(err)  }  fmt.Println("В переменной num хранится число", num) } |

В данном примере мы впервые получили возвращаемые функцией значения, а также обработали ошибку.

## Пакет math

Пакет **math** содержит базовые математические функции и константы.

### func Pow(x,y float64) float64

Служит для возведения числа в степень.

|  |
| --- |
| package main  import (  "fmt"  "math" )  func main() {  c := math.Pow(2, 3)  fmt.Printf("%.1f", c) } |

В примере число 2 возведено в 3 степень, функция возвращает ответ в переменную **c**.

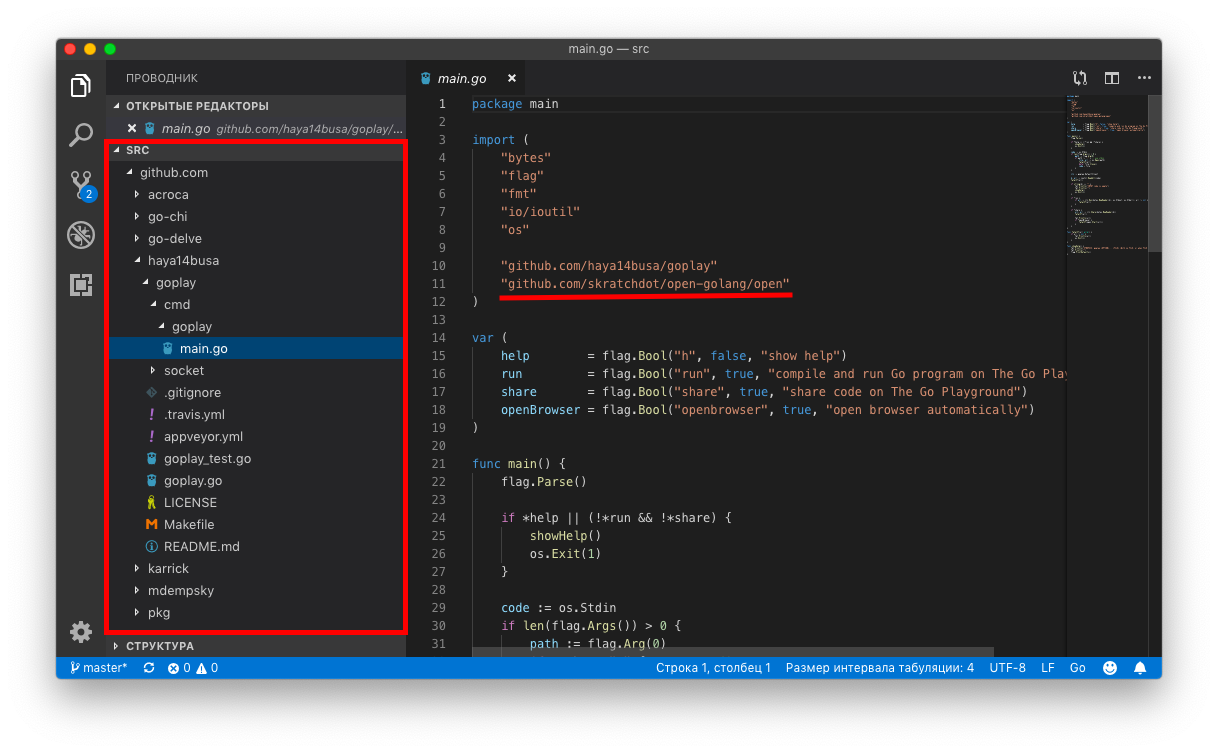
### func Sqrt(x float64) float64

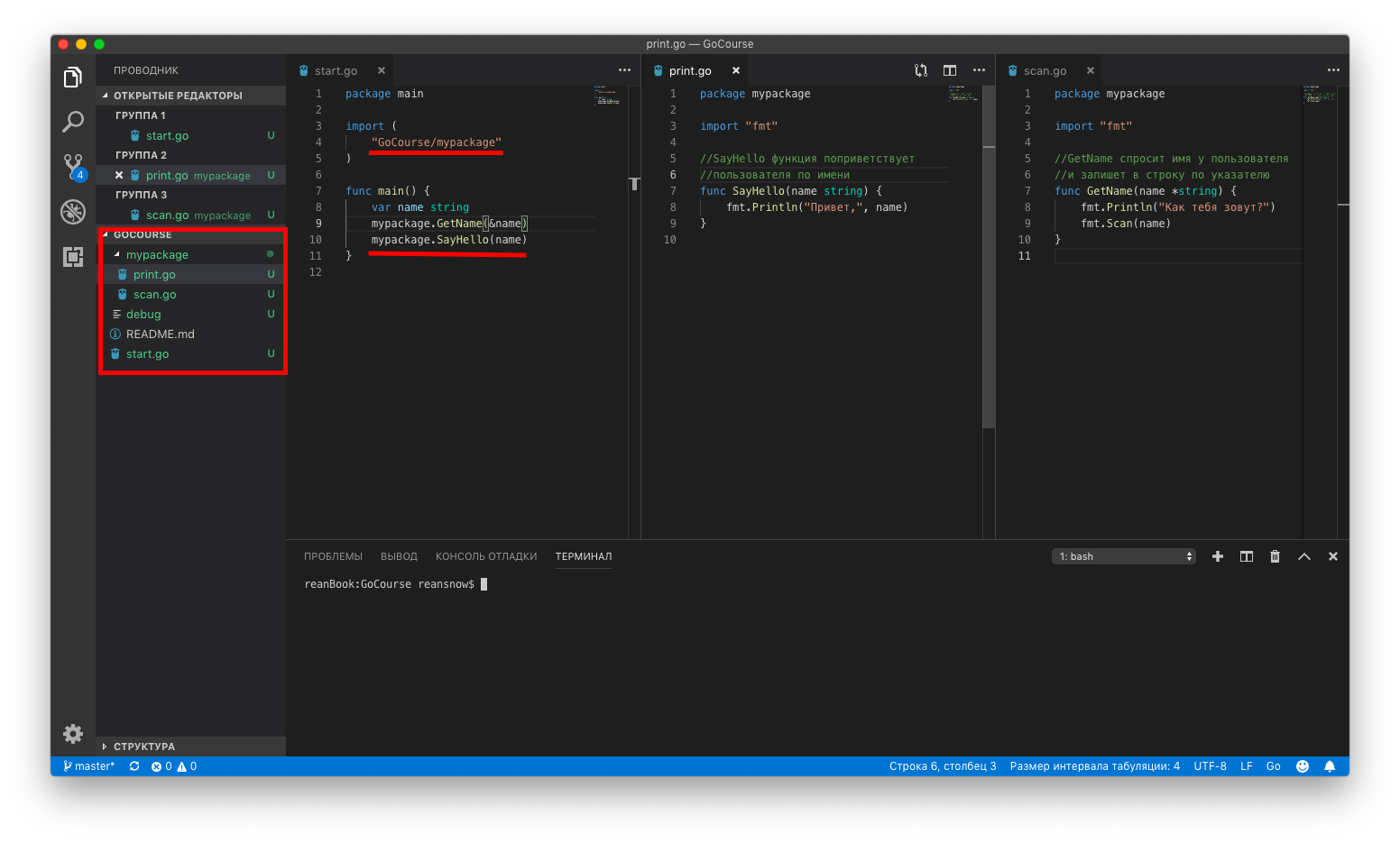
Функция **Sqrt** вернет квадратный корень из числа.

|  |
| --- |
| package main  import (  "fmt"  "math" )  func main() {  c := math.Sqrt(9)  fmt.Printf("%.1f", c) } |

В примере был извлечен квадратный корень из числа 9, результат работы записан в переменную **c**.

## Создание собственного пакета

Создавая директории в GOPATH/src и размещая код в них, вы создаете пакеты, которые потом сможете использовать при написании программ. Эти пакеты могут содержать и другие пакеты внутри себя.

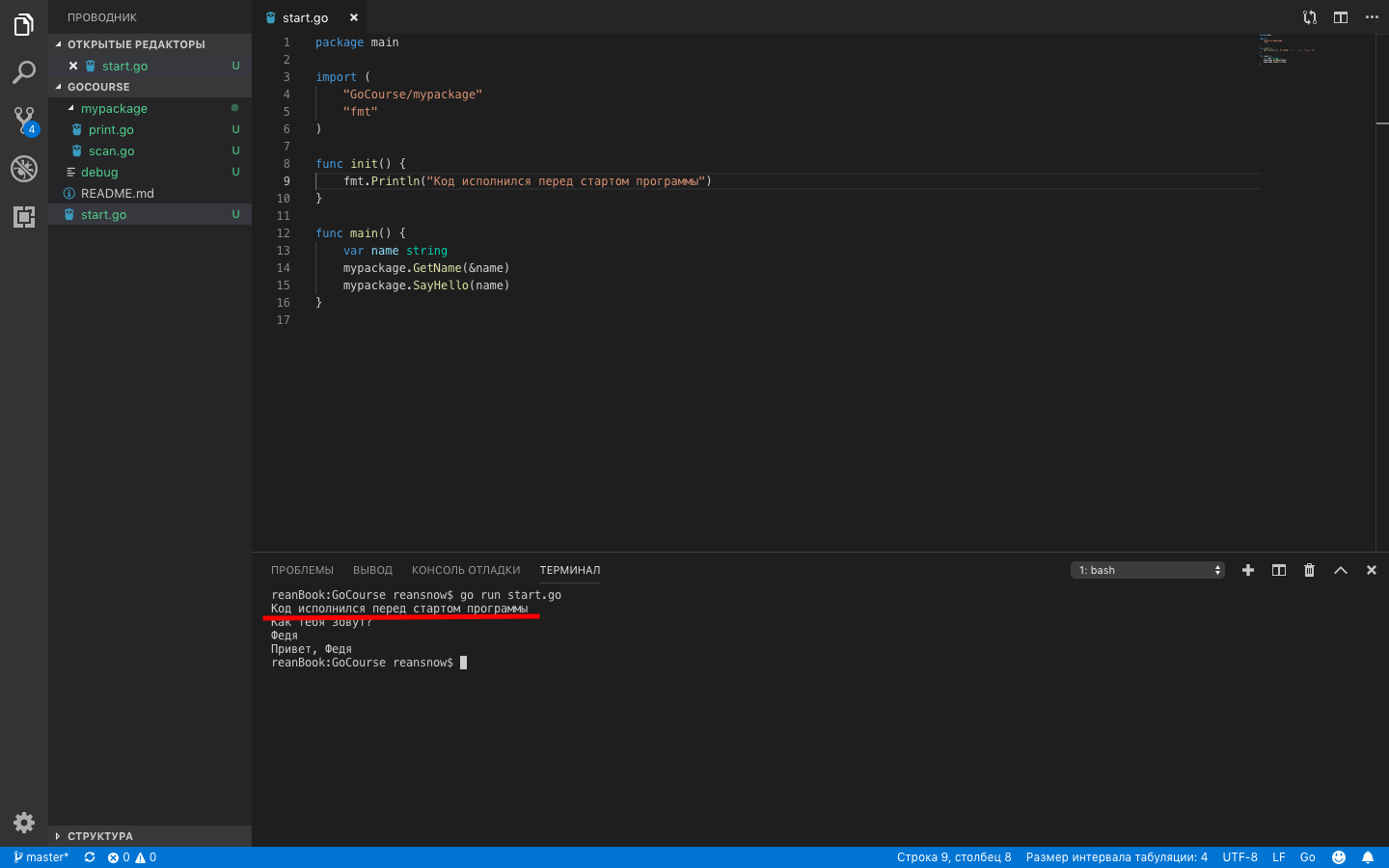
Как видно на снимке, вложенность пакетов может быть глубокой. Если пакет находится глубже директории **src**, для его импорта указывается относительный путь.

В директории GoCourse был создан пакет **mypackage**, а также описаны две функции: **SayHello** и **GetName**, — каждая в своем файле, но имя пакета у них одинаковое. Дело в том, что одна директория может содержать только один пакет, но сам он может быть описан в нескольких файлах.

**Итоги**

* Создание пакета — это создание директории внутри папки **src** в вашем GOPATH.
* Пакет может содержать несколько файлов с исходным кодом, но имя пакета у каждого должно быть одинаковым.
* Обращение из кода к вложенным в пакет функциям и переменным происходит по названию пакета без указания полного пути.

## Инициализация пакета: разбираем метод init

Каждый из пакетов может содержать метод **init()**. Его выполнение происходит перед вызовом функции **main()**. Данный метод служит для начального конфигурирования пакета.

## Обсуждаем область видимости внутри и вне пакета

Область видимости — это совокупность переменных, функций и методов, к которым есть доступ из выбранной точки кода.

Надо запомнить, что наименования (констант, переменных, структур и функций) с большой и маленькой буквы будут существенно влиять на сами сущности.

Названные с большой буквы сущности будут доступны снаружи при импортировании пакета, а с маленькой — только внутри пакета.

Например, названия функций из пакета **fmt** начинаются с большой буквы. Сделано это для того, чтобы дать возможность пользоваться функциями пакета из других программ.

**У функций есть доступ к:**

* переменным пакета (приватным и публичным);
* функциям пакета (приватным и публичным);
* структурам пакета (приватным и публичным);
* интерфейсам пакета (приватным и публичным);
* импортированным переменным (публичным);
* импортированным функциям (публичным);
* импортированным структурам (публичным);
* импортированным интерфейсам (публичным).

**У методов есть доступ к:**

* тому же, что и у функций;
* а также к приватным полям структуры, для которой описан метод.

На деле это будет достаточно просто, стоит только войти во вкус.

# Переменные и константы

**Переменная** — это именованное место хранения типа данных.

**Константа** — это переменная, значения которой не могут быть изменены после инициализации.

**Указатель** — это переменная, значение которой является адресом другой переменной.

## Переменные

Изменим программу, которую мы написали ранее, чтобы в ней использовались переменные:

|  |
| --- |
| package main  import "fmt"  func main() {  var x string = "Привет, мир!"  fmt.Println(x) } |

Обратите внимание, что мы создаем переменную и присваиваем ей текст, — вместо того, чтобы напрямую передать его в функцию **Println**. Переменные в Go создаются с помощью ключевого слова **var**, за которым следуют имя переменной **x**, тип **string** и присваиваемое значение «Привет, мир!».

## Именование

Правильное именование переменных — важная часть разработки ПО. Имена должны начинаться с буквы и могут содержать буквы, цифры и знак подчеркивания \_. Компилятору Go, в принципе, все равно, как вы назовете переменную, но не забудьте, что вам (может быть, и кому-то еще) потом это придется читать. Предположим, у нас есть:

|  |
| --- |
| x := "Max" fmt.Println("My dog's name is", x) |

В этом случае **x** — не идеальное имя переменной. Лучше было бы так:

|  |
| --- |
| name := "Max" fmt.Println("My dog's name is", name) |

Или даже так:

|  |
| --- |
| dogsName := "Max" fmt.Println("My dog's name is", dogsName) |

Главное — помнить: когда переменная должна быть доступна при импортировании пакета, ее название должно начинаться с большой буквы. И для всех случаев правильно начинать каждое последующее слово с большой буквы.

### Области видимости

В языке Go область видимости регулируется с помощью имени переменной, функции, метода, интерфейса.

|  |
| --- |
| package foo  *// Это глобальная переменная, она будет видна в пакетах* *// в которые будет импортирован foo* var GlobalVar := "Я глобальная константа"  *// Это переменная пакета, она будет видна во всех функциях и структурах* *// внутри foo* var variable := 0  *// Так как данная функция видна из других пакетов,* *// вы сможете получить следующее ее значение с помощью* *// NextNumber* func NextNumber() int {  variable++  return variable }  *// Функция updateIterator может быть вызвана только из пакета foo* *// в качестве аргумента она получает переменную num, которая может быть* *// использована только внутри функции* func updateIterator(num int) {  variable = num }  *// Функции имеют свою область видимости, если переменную объявить внутри функции,*  *// она не будет* *видна из других функций пакета и из других пакетов при*  *// импортировании* |

### Зарезервированные слова

Зарезервированные слова — это список слов, которые нельзя использовать при создании переменных. Дело в том, что надо объяснить компилятору, какая часть полученного исходного текста описывает алгоритм.

Вот список зарезервированных слов:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| break | default | func | interface | select |
| case | defer | go | map | struct |
| chan | else | goto | package | switch |
| const | fallthrough | if | range | type |
| continue | for | import | return | var |
| new |  |  |  |  |

## Разбираем способы объявления переменных

Возможен вариант, когда значение переменной будет присвоено после ее создания:

|  |
| --- |
| package main import "fmt" func main() {  var x string  x = "Привет, мир!"  fmt.Println(x) } |

Символ **=** в Go является оператором присваивания, поэтому читать строку следует как «переменной **x** присваивается значение „Привет, мир!“». Это важно потому, что переменные могут менять свои значения во время выполнения программы.

Рассмотрим следующий пример:

|  |
| --- |
| package main import "fmt" func main() {  var x string  x = "первый"  fmt.Println(x)  x = "второй"  fmt.Println(x) } |

Можно сделать даже так:

|  |
| --- |
| var x string x = "first " fmt.Println(x) x = x + "second" fmt.Println(x) |

Эта программа будет бессмысленной, если читать ее как теорему из алгебры. Но она обретет смысл, если вы будете внимательно читать ее как список команд. Когда мы видим **x = x + "second"**, то должны воспринимать это так: «Присвоить конкатенацию значения переменной **x** и литерала строки переменной **x**». Операции справа от **=** выполняются первыми, а результат присваивается левой части.

Если хотим присвоить значение переменной при ее создании, то можем использовать сокращенную запись:

|  |
| --- |
| x := "Hello World" |

Обратите внимание, что **:** стоит перед **=**, а тип отсутствует. Тип в данном случае указывать не обязательно, так как компилятор Go способен определить его по значению, которым мы инициализируем переменную. Тут мы присваиваем строку, поэтому **x** будет иметь тип **string**. Компилятор может определить тип и при использовании **var**:

|  |
| --- |
| var x = "Hello World" |

## Обсуждаем указатели и функцию new

Как вы знаете, каждая переменная является местом памяти и каждая ячейка памяти имеет свой адрес, который можно получить, используя оператор **ampersand** (&). Рассмотрим пример, где будет печататься адрес определяемых переменных:

|  |
| --- |
| package main import "fmt" func main() {  var a int = 10  fmt.Println("Адрес переменной:", &a) } |

Мы уже видели, как в функцию **Scanln** вместо переменной мы передавали указатель на нее. Дело в том, что значения, которые мы передаем в функцию, копируются в новые переменные. Значит, и место хранения в памяти станет новым, и мы не получим результат из функции. Но сколько бы ни происходило копирование адреса, он все равно будет вести лишь в одну ячейку в памяти.

Рассмотрим на примере:

|  |
| --- |
| package main import "fmt" func main() {  a := 1  b := 2  fmt.Printf("Значение переменной a: %x, b: %x \n", a, b)  a = b  fmt.Printf("Значение переменной a: %x, b: %x \n", a, b)  a = a + 1  b = b + 1  fmt.Printf("Значение переменной a: %x, b: %x \n", a, b) } ------ Результат выполнения Значение переменной a: 1, b: 2  Значение переменной a: 2, b: 2  Значение переменной a: 3, b: 3 |

Создали две переменные, присвоили одной значение другой, а потом увеличили каждую на 1:

|  |
| --- |
| package main import "fmt" func main() {  a := new(int)  b := new(int)  \*a = 1  \*b = 2  fmt.Printf("Значение переменной a: %x, b: %x \n", \*a, \*b)  a = b  fmt.Printf("Значение переменной a: %x, b: %x \n", \*a, \*b)  \*a = \*a + 1  \*b = \*b + 1  fmt.Printf("Значение переменной a: %x, b: %x \n", \*a, \*b) } ------Результат выполнения Значение переменной a: 1, b: 2  Значение переменной a: 2, b: 2  Значение переменной a: 4, b: 4 |

Разберем второй пример по порядку.

Указатели созданы с ключевым словом **new**, а также указанием типа данных в круглых скобках. Создать указатель можно и с помощью & перед типом, но в отличие от случая с **new**, он не будет ни на что ссылаться, а попытка изменить данные по указателю приведет к ошибке.

Чтобы изменить значения по указателю, используется обращение со звездочкой \*. Так мы даем компилятору понять, что изменять будем именно значение, а не сам указатель.

Значение **a** и **b** изменилось на 2, а не 1, потому что мы присвоили указатель из переменной **b** в **a**. С этого момента они начали указывать на одну область в памяти, а значит изменения значений по указателю **a** влекут за собой изменения и по указателю **b**.

## Время жизни переменных

Жизнь переменной — это время, пока к ней можно обращаться из кода. Это время зависит от места, в котором они были вызваны. Глобальные переменные считаются «живыми» до тех пор, пока выполняется программа, а остальные — пока выполняется контекст, который их создал (цикл, ветвление, структура, функция или метод).

## Разбираем константы

Go также поддерживает константы. Они создаются так же, как и переменные, только вместо **var** используется ключевое слово **const**:

|  |
| --- |
| package main  import "fmt"  func main() {  const x string = "Hello World"  fmt.Println(x) } |

А вот этот код:

|  |
| --- |
| const x string = "Hello World" x = "Some other string" |

вызовет ошибку компиляции: **.\main.go:7: cannot assign to x**

Константы — хороший способ использовать определенные значения в программе без необходимости писать их каждый раз. Например, константа **Pi** из пакета **math**.

# Фундаментальные типы данных

Типы данных определяют множество принимаемых значений, описывают, какие операции могут быть применены к ним, и определяют, как данные будут храниться.

## Разбираем целые числа, типы, операции над целыми числами

Целые числа, как и их математические коллеги, — это числа без вещественной части. В отличие от десятичного представления чисел, которое используем мы, компьютеры применяют двоичное представление.

Десятичная система счисления строится на 10 цифрах. Когда мы исчерпываем доступные, представляем большое число, используя новую цифру 2 (а затем 3, 4, 5 и далее). Числа следуют одно за другим. Например, число, следующее за 9, — это 10, а за 99 — 100 и так далее. Компьютеры делают то же самое, но имеют только две цифры вместо десяти. Поэтому подсчет выглядит так: 0, 1, 10, 11, 100, 101, 110, 111 и далее. Другое отличие между системой счисления, которую используем мы, и «компьютерной» — все типы чисел имеют строго определенный размер. У них есть ограниченное количество цифр. Поэтому четырехразрядное число может выглядеть так: 0000, 0001, 0010, 0011, 0100. В конце концов мы можем выйти за лимит разрядности, и большинство компьютеров просто вернутся к самому началу (что может стать причиной очень странного поведения программы).

В Go существуют следующие типы целых чисел: **uint8**, **uint16**, **uint32**, **uint64**, **int8**, **int16**, **int32** и **int64**. Значения 8, 16, 32 и 64 говорят нам, сколько бит использует каждый тип. **uint** означает «unsigned integer» (беззнаковое целое), а **int** — «signed integer» (знаковое целое). Беззнаковое целое может принимать только положительные значения (или ноль). Еще есть два типа-псевдонима: **byte** (то же самое, что uint8) и **rune** (то же самое, что int32). Байты — очень распространенная единица измерения в компьютерах (1 байт = 8 бит, 1024 байта = 1 килобайт, 1024 килобайта = 1 мегабайт), и именно поэтому тип **byte** в Go часто используется для определения других типов. Также существует три машинно-зависимых целочисленных типа: **uint**, **int** и **uintptr**, — их размер зависит от архитектуры используемого компьютера.

В общем, если вы работаете с целыми числами, пока можно просто использовать тип **int**.

## Разбираем числа с плавающей точкой

Числа с плавающей точкой содержат вещественную часть (вещественные числа) (1.234, 123.4, 0.00001234, 12340000). Их представление в компьютере довольно сложно и знание о нем не критично для их использования. Так что мы просто должны помнить, что:

* Числа с плавающей точкой неточны. Бывают случаи, когда число вообще нельзя представить. Например, результатом вычисления 1.01 – 0.99 будет 0.020000000000000018 — число очень близкое к ожидаемому, но не то же самое.
* Как и целые числа, числа с плавающей точкой имеют определенный размер (32 или 64 бита). Использование большего размера увеличивает точность (сколько цифр мы можем использовать для вычисления)
* В дополнение к числам существуют несколько других значений: «not a number» (не число) (NaN — для вещей наподобие 0/0), а также положительная и отрицательная бесконечность (+∞ и −∞).

В Go есть два вещественных типа: **float32** и **float64** (часто их называют вещественными числами с одинарной и двойной точностью). А также два дополнительных типа для представления комплексных чисел (чисел с мнимой частью): **complex64** и **complex128**. Как правило, мы должны придерживаться типа **float64**, когда работаем с числами с плавающей точкой.

## Разбираем логические значения и сравнения

Булевский тип — это специальный однобитный целочисленный тип, используемый для представления истинности и ложности. С ним используются три логических оператора:

|  |  |
| --- | --- |
| **Литерал** | **Пояснение** |
| && | И |
| || | ИЛИ |
| ! | НЕ |

Вот пример программы, показывающей их использование:

|  |
| --- |
| package main import "fmt" func main() {  fmt.Println(true && true)  fmt.Println(true && false)  fmt.Println(true || true)  fmt.Println(true || false)  fmt.Println(!true) } ------ Результат работы программы true false true true false |

## Разбираем строки

**Строка** — это последовательность символов определенной длины, используемая для представления текста. Строки в Go состоят из независимых байтов, обычно по одному на каждый символ (символы из других языков, например китайского, представляются несколькими байтами).

Строковые литералы могут быть созданы с помощью двойных кавычек "Hello World" или с помощью апострофов `Hello World`. Различие между ними в том, что строки в двойных кавычках не могут содержать новые строки и позволяют использовать особые управляющие последовательности символов. Например, **\n** будет заменена символом новой строки, а **\t** — символом табуляции.

Распространенные операции над строками включают нахождение длины строки **len("Hello World")**, доступ к отдельному символу в строке **"Hello World"[1]** и конкатенацию двух строк **"Hello " + "World"**. Модифицируем созданную ранее программу, чтобы проверить это:

|  |
| --- |
| package main  import "fmt"  func main() {  fmt.Println(len("Hello World"))  fmt.Println("Hello World"[1])  fmt.Println("Hello " + "World") } |

* Пробел тоже считается символом, поэтому длина строки — 11 символов, а не 10, и третья строка содержит "Hello " вместо "Hello".
* Символы в строках считаются с 0, а не с 1. [1] даст вам второй элемент, а не первый. Также заметьте, что вы видите 101 вместо **e**, когда выполняете программу. Это происходит из-за того, что символ представляется байтом (помните, байт — это целое число).
* Конкатенация использует тот же символ, что и сложение.

# Операторы для работы с данными

### Арифметические операторы

|  |  |
| --- | --- |
| **Оператор** | **Описание** |
| + | Добавляет два операнда |
| - | Вычитает второй операнд из первого |
| \* | Умножает оба операнда |
| / | Делит числитель на знаменатель |
| % | Оператор модуля — дает остаток после целочисленного деления |
| ++ | Оператор приращения — увеличивает целочисленное значение на единицу |
| -- | Оператор сокращения — уменьшает целочисленное значение на единицу |

### Реляционные операторы

|  |  |
| --- | --- |
| **Оператор** | **Описание** |
| == | Проверяет, равны ли значения двух операндов; если да, условие становится истинным |
| != | Проверяет, равны ли значения двух операндов; если значения не равны, то условие становится истинным |
| > | Проверяет, превышает ли значение левого операнда значение правого операнда; если да, условие становится истинным |
| < | Проверяет, меньше ли значение левого операнда значения правого операнда; если да, условие становится истинным |
| >= | Проверяет, больше ли значение левого операнда значения правого операнда или равно ему; если да, условие становится истинным |
| <= | Проверяет, является ли значение левого операнда меньшим или равным значению правого операнда; если да, условие становится истинным |

### Логические операторы

|  |  |
| --- | --- |
| **Оператор** | **Описание** |
| & | Вызывается логическим оператором AND. Если оба операнда отличны от нуля, условие становится истинным |
| | | Вызывается логическим оператором ИЛИ. Если любой из двух операндов отличен от нуля, условие становится истинным |
| ! | Вызывается логическим оператором NOT. Используется для изменения логического состояния операнда. Если условие истинно, то логический оператор NOT сделает его ложным |

### Побитовые операторы

Побитовые операторы работают с битами и выполняют побитовые операции. Таблицы истинности для &, |, и ^ заключаются в следующем:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **q** | **p & q** | **p | q** | **p ^ q** |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

Побитовые операторы, поддерживаемые языком Go, перечислены в следующей таблице.

|  |  |
| --- | --- |
| **Оператор** | **Описание** |
| & | Двоичный оператор AND копирует бит в результат, если он существует в обоих операндах |
| | | Двоичный оператор OR копирует бит, если он существует в любом из операндов |
| ^ | Оператор двоичного XOR копирует бит, если он установлен в один операнд, но не тот и другой |
| << | Двойной левый оператор сдвига. Значение левых операндов перемещается влево на количество бит, заданных правым операндом |
| >> | Двоичный оператор правого сдвига. Значение левых операндов перемещается вправо на количество бит, заданных правым операндом |

### Операторы присваивания

|  |  |
| --- | --- |
| **Оператор** | **Описание** |
| = | Простой оператор присваивания, присваивает значения из правых операндов в левый операнд |
| += | Добавить AND присваивания. Добавляет правый операнд в левый операнд и присваивает результат левому операнду |
| -= | Вычитание и оператор присваивания. Вычитает правый операнд из левого и присваивает результат левому операнду |
| \*= | Оператор умножения и присваивания. Умножает правый операнд на левый и присваивает результат левому операнду |
| /= | Оператор присваивания и деления. Делит левый операнд на правый, и присваивает результат левому операнду |
| %= | Оператор деление по модулю и присваивания. Делит по модулю левый операнд на правы, присваивает результат левому операнду |
| <<= | Оператор побитового сдвига влево и присваивания |
| >>= | Оператор побитового сдвига вправо и присваивания |
| &= | Оператор побитового И и присваивания |
| |= | Оператор побитового ИЛИ и присваивания |
| ^= | Оператор побитового XOR и присваивания |

# Практическое задание

1. Написать программу для конвертации рублей в доллары. Программа запрашивает сумму в рублях и выводит сумму в долларах. Курс доллара задайте константой.
2. Даны катеты прямоугольного треугольника. Найти его площадь, периметр и гипотенузу. Используйте тип данных **float64** и функции из пакета [math](#_qjew3dwv27ds).
3. \* Пользователь вводит сумму вклада в банк и годовой процент. Найти сумму вклада через 5 лет.

Задача со \* — дополнительная, для продвинутого уровня.

# Дополнительные материалы

1. [A Tour of Go](https://go-tour-ru-ru.appspot.com/welcome/1).
2. [Перевод книги An introduction programming in Go](http://golang-book.ru).
3. [Бесплатный курс Git. Быстрый старт](https://geekbrains.ru/courses/66).
4. [Пакет fmt](https://golang.org/pkg/fmt/).
5. [Пакет strconv](https://golang.org/pkg/strconv/).

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Описание стандартной библиотеки Go](https://golang.org/pkg/).
2. [Книга An introduction programming in Go](https://www.golang-book.com/books/intro).