

# Go

Начальный курс по Go

**ПЕРЕМЕННЫЕ**

# Переменные

Ключевые понятия:

1. Имя
2. Значение
3. Тип

Пример с файлами

# Переменные

Синтаксис:

имя переменной

`var count int`

ключевое слово

тип

# Переменные

Синтаксис (объявление + присваивание):

```
var count int = 10
```

```
var count = 10
```

```
count := 10
```

} стараться использовать этот вариант

# Переменные

Важно: неинициализированные переменные  
всегда инициализированы в нулевые  
значения (например, для целых чисел – 0)

# Переменные

```
func main() {  
    count := 10  
    fmt.Println(count)  
}
```

# Переменные

```
func main() {  
    fmt.Println(count)  
    count := 10  
    // ниже можно использовать имя count, выше - нет  
    fmt.Println(count)  
}
```



# Имя

ИСПОЛЬЗОВАТЬ

1. Должно быть осмысленным
2. Начинается с буквы (\_, \$ - не рекомендуется)
3. Содержит буквы (цифры, \_, \$ - не рекомендуется)
4. Если состоит из двух и более слов пишется в нотации camelCase\*
5. Имя на английском языке (без транслита – никаких summaOperacii и подобных!)

ИСПОЛЬЗОВАТЬ

—

\_ - don't care (специальное имя,  
используемое тогда, когда нужно что-то  
написать, но значение нас не интересует)

# Quality Gate

ИСПОЛЬЗОВАТЬ

В большинстве требований приняты стандарты по оформлению кода. Ваш код не принимается, если вы не соблюдаете стандарты.

Поэтому, домашние работы будут отправляться на доработку, если:

- ✓ 1. Не выполняются правила именования
- 2. Код не хранится в системе контроля версий (позже)
- 3. Не выполняются статические проверки (позже)
- 4. Нет авто-тестов, либо недостаточное покрытие (позже)
- 5. Не выполнены требования (подробнее см. в требованиях к задаче)

# **ЦЕЛОЧИСЛЕННЫЕ ТИПЫ**

# Целочисленные типы

1. `int8` – 1 байт (-128 до 127)
2. `int16` – 2 байта (-32768 до 32767)
3. `int32 (rune)` – 4 байта (-2147483648 до 2147483647)
4. `int64` – 8 байт (-9223372036854775808 до 9223372036854775807)

`int` - по умолчанию для целых чисел будет `int32` для 32 битных систем и `int64` для 64 битных систем

# Целочисленные типы

1. `uint8 (byte)` – 1 байт
2. `int16` – 2 байта
3. `int32` – 4 байта
4. `int64` – 8 байт

Те же типы, только без знака (т.е. отрицательные числа там не хранятся)

# ПЗ: Посмотреть на ошибки

```
func main() {  
    var overflow int8 = 256  
    fmt.Println(overflow)  
    var invalidType int = 10.8  
    fmt.Println(invalidType)  
    var uninitialized int  
    fmt.Println(uninitialized)  
    fmt.Println(undefined)  
}
```

# Обязательно смотрите на ошибки!

```
▶ func main() {  
    var overflow int8 = 256  
    fmt.Println(overflow)  
    var invalidType int = 10.8  
    fmt.Println(invalidType)  
    var uninitialized int  
    ! fmt.Println(uninitialized)  
    fmt.Println(undefined)  
}
```

Unresolved reference 'undefined'

Create variable 'undefined' ↶ ↷ ↻ More actions... ↶ ↷ ↻



# Форматы записи

1. По умолчанию – в десятичной системе
2. 0x..., 0X... – в шестнадцатеричной
3. 0o... (или просто 0...) - в восьмеричной
4. 0b... - в двоичной

Можно использовать \_ для разделения символов,  
например: 0b0000\_1111

**ХРАНЕНИЕ ИНФОРМАЦИИ**

# Системы счисления

Двоичная – 0 и 1

Восьмеричная – 0-7

Десятичная – 0-9

Шестнадцатеричная – 0-F

# Позиционная система счисления

123 -> вес цифры зависит от её позиции в числе (чем левее, тем весомее цифра)

Т.е.  $1 * 10^2 + 2 * 10^1 + 3 * 10^0$

В двоичной (а также 8-ой и 16-ой) всё так же:

$$101_2 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0$$

# Шестнадцатеричная СС

1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

$$10_{16} = 1 * 16^1 + 0 * 16^0$$

# Байты и биты

Бит – единица измерения количества информации

Байт – 8 бит

# Разминка

Переведите в десятичную систему счисления:

01000110

10111001

Переведите в двоичную систему счисления:

128

А как же буквы, картинки и  
т.д.?

Если всё в компьютере – 0 и 1 как же мы  
видим буквы, картинки и прочее?



# ASCII & UTF

## ASCII Code: Character to Binary

0	0011 0000	O	0100 1111	m	0110 1101
1	0011 0001	P	0101 0000	n	0110 1110
2	0011 0010	Q	0101 0001	o	0110 1111
3	0011 0011	R	0101 0010	p	0111 0000
4	0011 0100	S	0101 0011	q	0111 0001
5	0011 0101	T	0101 0100	r	0111 0010
6	0011 0110	U	0101 0101	s	0111 0011
7	0011 0111	V	0101 0110	t	0111 0100
8	0011 1000	W	0101 0111	u	0111 0101

**ТИПЫ С ПЛАВАЮЩЕЙ ТОЧКОЙ**

# Типы с плавающей точкой

1. float32 – 4 байта (6-7 значащих знаков)
2. float64 – 8 байт (15 значащих знаков)

# Типы с плавающей точкой

$$1.234e+2 = 1.234 * 10^2$$

$$1.234e-2 = 1.234 * 10^{-2}$$

**RUNE**

# rune

rune - символы Unicode

```
var symbol rune = 'Ы'
```

**СТРОЧНЫЙ ТИП**

# Строчный тип

string:

- `"""` - в двойных кавычках не могут содержать перенос строки
- ```` - могут быть многострочными



# Escape-символы

Есть ряд символов, которые не имеют печатного аналога либо используются в служебных целях, для них используются специальные escape-символы:

`\t` – табуляция

`\n` – перенос строки

`\r` – возврат каретки

`\"` – двойная кавычка

`\'` – одинарная кавычка

`\\` – обратный слэш

**ЛОГИЧЕСКИЙ ТИП**

# Логический тип

bool – true или false

**ОПЕРАТОРЫ**

Операторы	Ассоциативность
-> ++postfix --postfix [] . ()вызов метода	->
! ~ ++prefix --prefix +унарный _унарный ()cast new	<-
* / %	->
+ -	->
<< >> >>	->
< <= > >=	->
== !=	->
&	->
^	->
	->
&&	->
	->
?:	<-
= += -= *= /= %= &=  = ^= <<= >>= >>=	<-
,	->

# ПЗ: Приоритеты операторов

```
count := 3
```

```
fmt.Println(10 % count + count ^ 2 + 22 * count)
```

# ПЗ: Операторы

```
var count byte = 255
```

```
count++
```

```
fmt.Println(count) // понять, почему выводится такой результат
```

Самостоятельное изучение

# ОТРИЦАТЕЛЬНЫЕ ЧИСЛА



# Важно

Данный раздел предназначен только для  
ознакомления, детально разбираться в нём  
не нужно

# Числа со знаком

Старший бит – для хранения знака

$$1_{10} = 0000\ 0001_2$$

А как же хранить отрицательные?

$$-1_{10} = 1000\ 0001_2$$

Проблема:

$$\begin{array}{r} 0000\ 0001_2 \\ + \\ 1000\ 0001_2 \\ = \\ 1000\ 0010_2 \end{array}$$

# Альтернатива

Старший бит – для хранения знака

$$1_{10} = 0000\ 0001_2$$

А как же хранить отрицательные?

$$-1_{10} = 1111\ 1111_2$$

Проблемы нет:

$$\begin{array}{r} 0000\ 0001_2 \\ + \\ 1111\ 1111_2 \\ = \\ 0000\ 0000_2 \end{array}$$

# Дополнительный код (-1)

Алгоритм:

1. Модуль числа  $-1_{10} \rightarrow 0000\ 0001_2$
2. Инвертируем все биты  $\rightarrow 1111\ 1110_2$
3. Прибавляем 1  $\rightarrow 1111\ 1111_2$

# Дополнительный код (-128)

Алгоритм:

1. Модуль числа  $128_{10} \rightarrow 1000\ 0000_2$
2. Инвертируем все биты  $\rightarrow 0111\ 1111_2$
3. Прибавляем 1  $\rightarrow 1000\ 0000_2$

# Обратное преобразование

Алгоритм:

1. Запись в доп.коде  $-127_{10} \rightarrow 1000\ 0001_2$
2. Инвертируем все биты  $\rightarrow 0111\ 1110_2$
3. Прибавляем 1  $\rightarrow 0111\ 1111_2$  (модуль)

**ВЕЩЕСТВЕННЫЕ ЧИСЛА**

# Плавающая точка

Хранится:

1. Знак
2. Порядок
3. Мантисса

4 байта

31 – знак	30 – 23 порядок	22-0 мантисса
-----------	-----------------	---------------

Для самостоятельного чтения: <https://habrahabr.ru/post/112953/>



# КОМПЛЕКСНЫЕ ЧИСЛА

# Комплексные числа

1. `complex64`
2. `Complex128`

Мы проходить не будем, нужно для математики.

**ОПЕРАТОРЫ**

# Операторы

Набор операторов фиксирован:

- нельзя создать новый оператор
- нельзя переопределить поведение существующего

# **АРИФМЕТИЧЕСКИЕ ОПЕРАТОРЫ**

# Арифметические

$a + b$  – сложение

$a - b$  – вычитание

$a * b$  – умножение

$a / b$  – деление (  $5 / 2 = 2$  )

$a \% b$  – остаток от деления (  $5 \% 2 = 1$  )

# Операторы

Важно: в зависимости от типа данных,  
семантика может быть разная!

**ЛИТЕРАЛЫ**



# Литералы

Литерал – непосредственное значение в коде

var percent float = 0.27

var count int = 27



Литералы

запомнить

# Литералы

`int` – по умолчанию для целых

`float64` – по умолчанию для вещественных

# **ПРИВЕДЕНИЕ ТИПОВ**

# Приведение типов

запомнить

В Go нет неявного приведения типов, если встречаются в одном выражении два разных типа, то приводить нужно вручную через выражения `int8(x)`, `int16(x)` и т.д.

**УСЛОВИЯ**

# if, else

```
if <expr> { ... }
```

```
if <expr> { ... } else { ... }
```

Важно:

1. <expr> всегда должно быть boolean!
2. Всегда ставьте { }\* (кроме одного случая)

# Ключевые операторы

`==` - проверка на равенство

`!=` - проверка на неравенство

`>` - больше

`<` - меньше

`>=` - больше или равно

`<=` - меньше или равно

Важно, результат всегда **boolean**!

**МАССИВЫ**



# Массивы

Упорядоченный набор элементов одного типа\*

```
var <name>[<length>]<type>
```

```
var data[10]int
```


- Имеет фиксированную длину (определяется `len()`)
- Элементы индексируются с нулевого индекса
- Доступ к элементу по индексу с помощью `[index]`
- Значения инициализируются в нулевые

# Массивы

`<name> = [<length>]<type>{a, b};` // a, b - значения

Сокращённый вариант:

`data := [2]int{1, 2}`



Сразу инициализировать  
(записать туда значения)

`data := [...]int{1, 2, 3}` - ... (просим go сосчитать  
самому кол-во элементов)

# Массивы

```
scores := [...]int{8, 10, 0, 6};  
fmt.Println(scores[2]);
```

**ЦИКЛЫ**

# Цикл

Многократно повторяющийся набор инструкций

# Цикл for

```
for {
```

```
    ... // здесь инструкции
```

```
}
```

Бесконечный цикл

# Цикл for

```
for <check> {  
    ... // здесь инструкции  
}
```

check – условие продолжения

# Цикл for

```
for <init>; <check>; <post> {
```

```
    ... // здесь инструкции
```

```
}
```

- init – выражение инициализации
- check – условие продолжения
- post – подготовка к следующей итерации



# Цикл for

```
int[] scores = {8, 7, 4, 2, 10};  
for (int i = 0; i < 10; i++) {  
    System.out.println(i);  
}  
  
for (int i = 0; i < scores.length; i++) {  
    System.out.println(scores[i]);  
    // TODO: do something  
}
```

# Хардкодинг

Константы в коде (magic constants) - почти всегда плохо! Используйте соответствующие свойства и переменные

```
scores = [...]int{8, 7, 4, 2, 10};
```

```
for i := 0; i < 5; i++ {
```

```
    fmt.Println(scores[i])
```

Плохо!

```
}
```

```
for i := 0; i < len(scores); i++ {
```

```
    fmt.Println(scores[i])
```


Хорошо!

```
    // TODO: do something
```

```
}
```

# Цикл for

```
func main() {  
    scores := [...]int{8, 7, 4, 2, 10}  
    sum := 0  
    for i := 0; i < 5; i++ {  
        sum += scores[i]  
    }  
}
```



Плохо!

# Цикл for + range

```
for index, value := range data {
```

```
    ... // здесь инструкции
```

массив



```
}
```

Сам следит за перебором

**ПЗ: МАССИВЫ И ЦИКЛЫ**

# Сумма продаж

Задача: в массиве хранятся данные о продажах. Вычислить сумму всех продаж

```
sales := [...]int{1_000, 2_000, 500, 5_000};
```

# Средний рейтинг

```
scores := [...]int{8, 7, 4, 2, 10};
```

В массиве `scores` хранятся оценки, выставленные пользователями оператору колл-центра.

Необходимо вычислить среднюю оценку и вывести её на экран

# Лучший сотрудник

```
scores := [...]int{8, 7, 4, 2, 10};
```

В массиве `scores` хранятся средние оценки разных менеджеров, найдите:

1. Максимальную оценку
2. Индекс максимальной оценки



# Худший сотрудник

```
scores := [...]int{8, 7, 4, 2, 10};
```

В массиве `scores` хранятся средние оценки разных менеджеров, найдите:

1. Минимальную оценку
2. Индекс минимальной оценки

# GoLand

ИСПОЛЬЗОВАТЬ

main + tab -> разворачивает main

shift + f6 -> переименование имени

ctrl + alt + v -> создание локальной переменной

shift + ctrl + alt + левый клик -> много курсоров

ctrl + alt + l -> отформатировать весь файл

**GIT**

# VCS

Version Control System (VCS) - система контроля версий

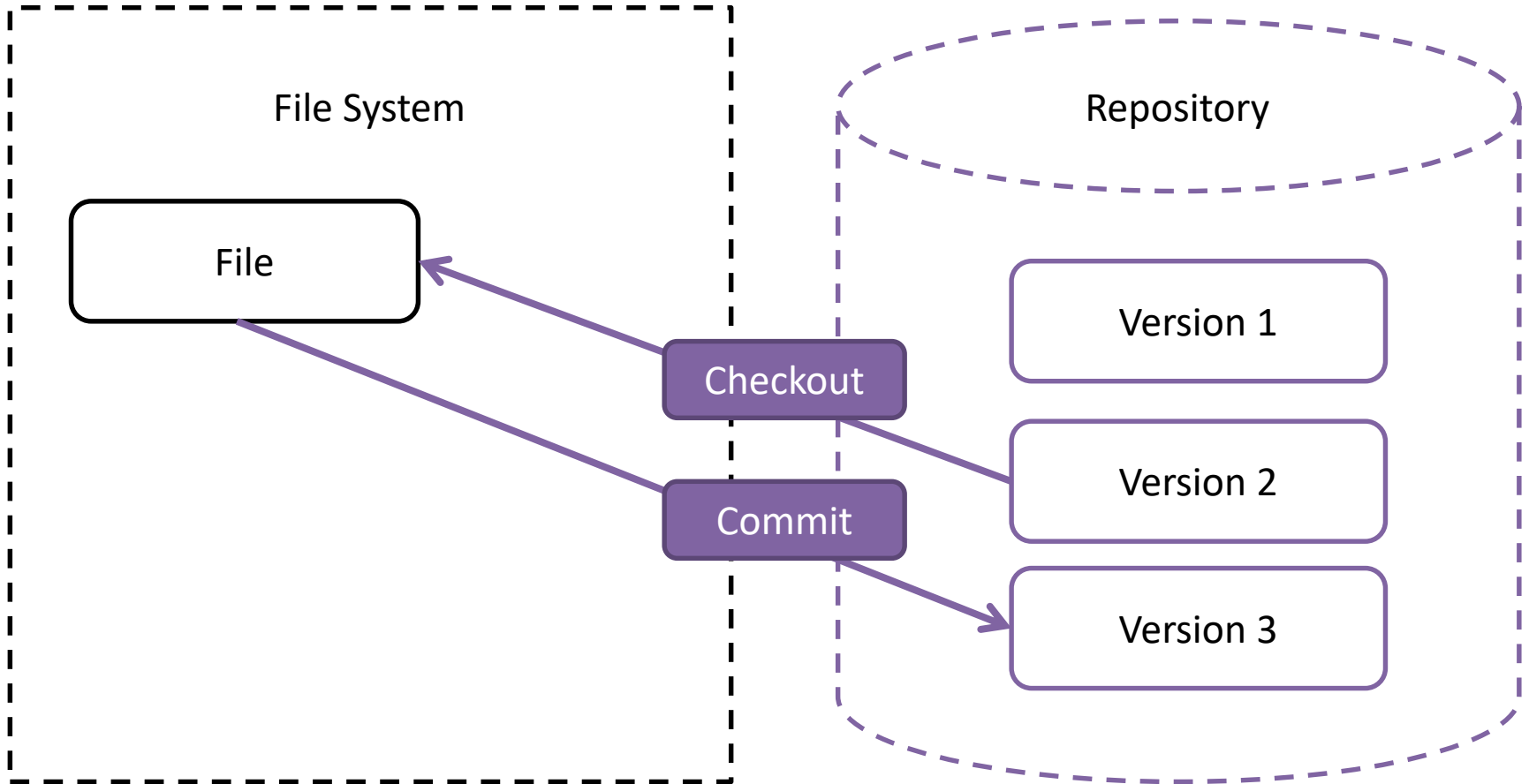
Предназначение – отслеживание изменений и  
коллективная работа

Git, Subversion (SVN), Mercurial и другие

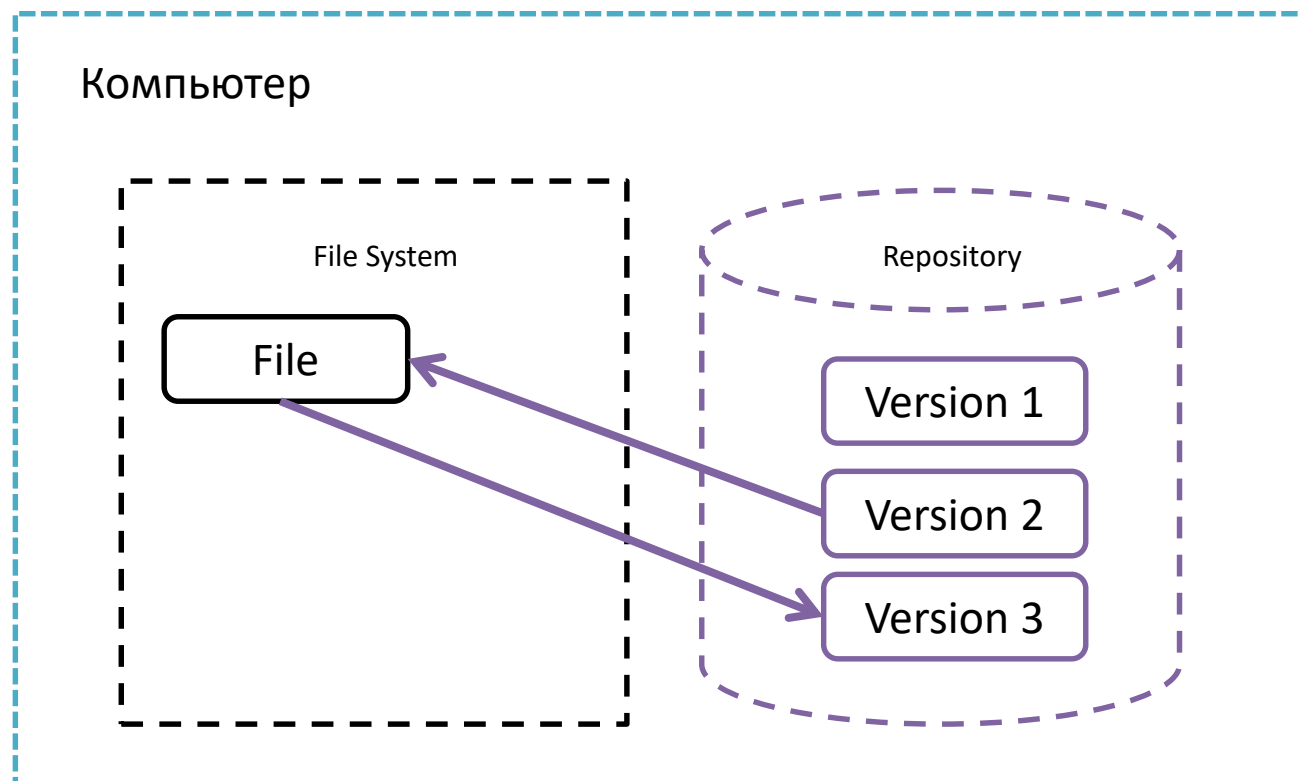
# VCS

- локальные
- централизованные
- распределённые

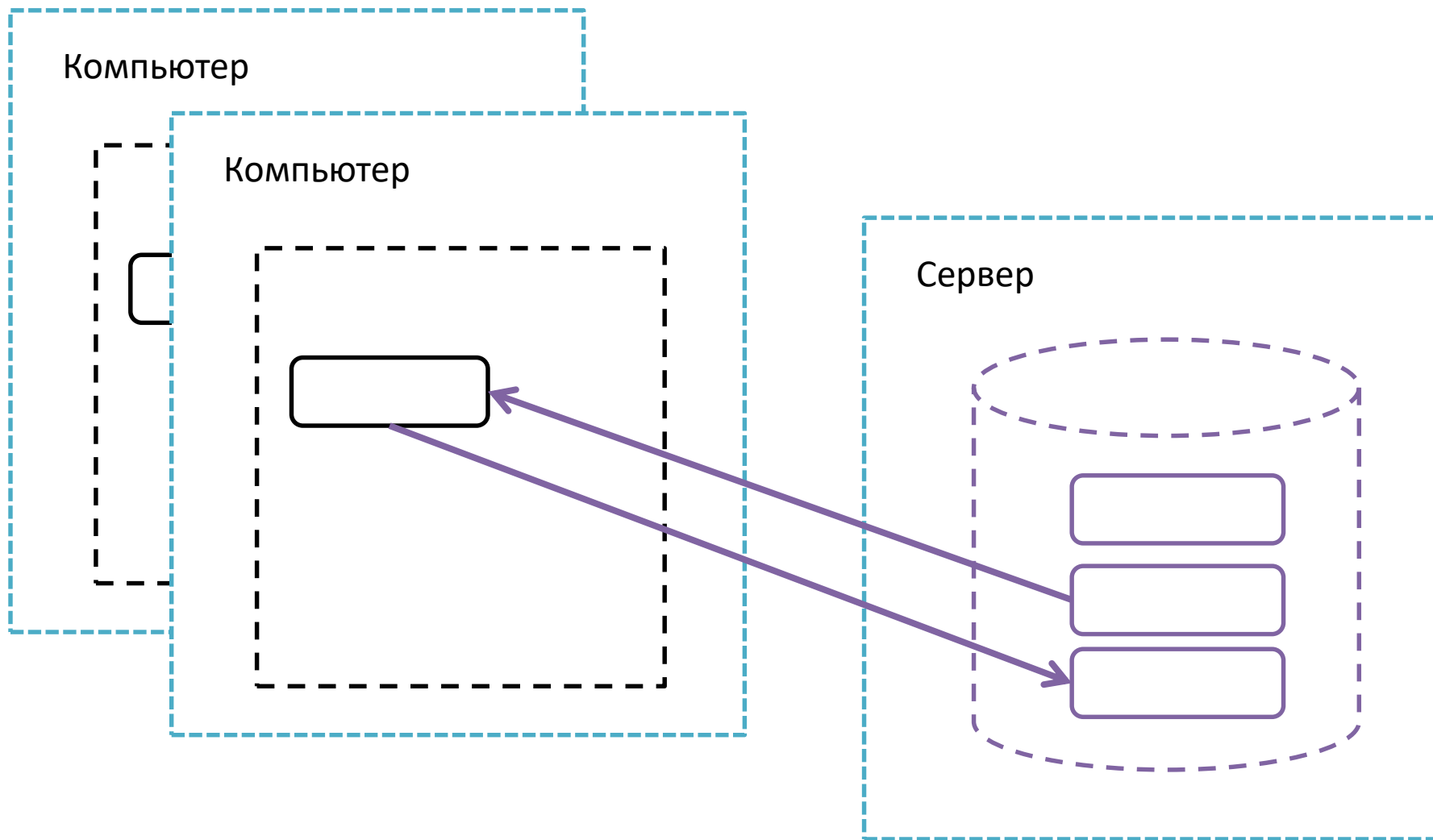
# VCS



# Local VCS

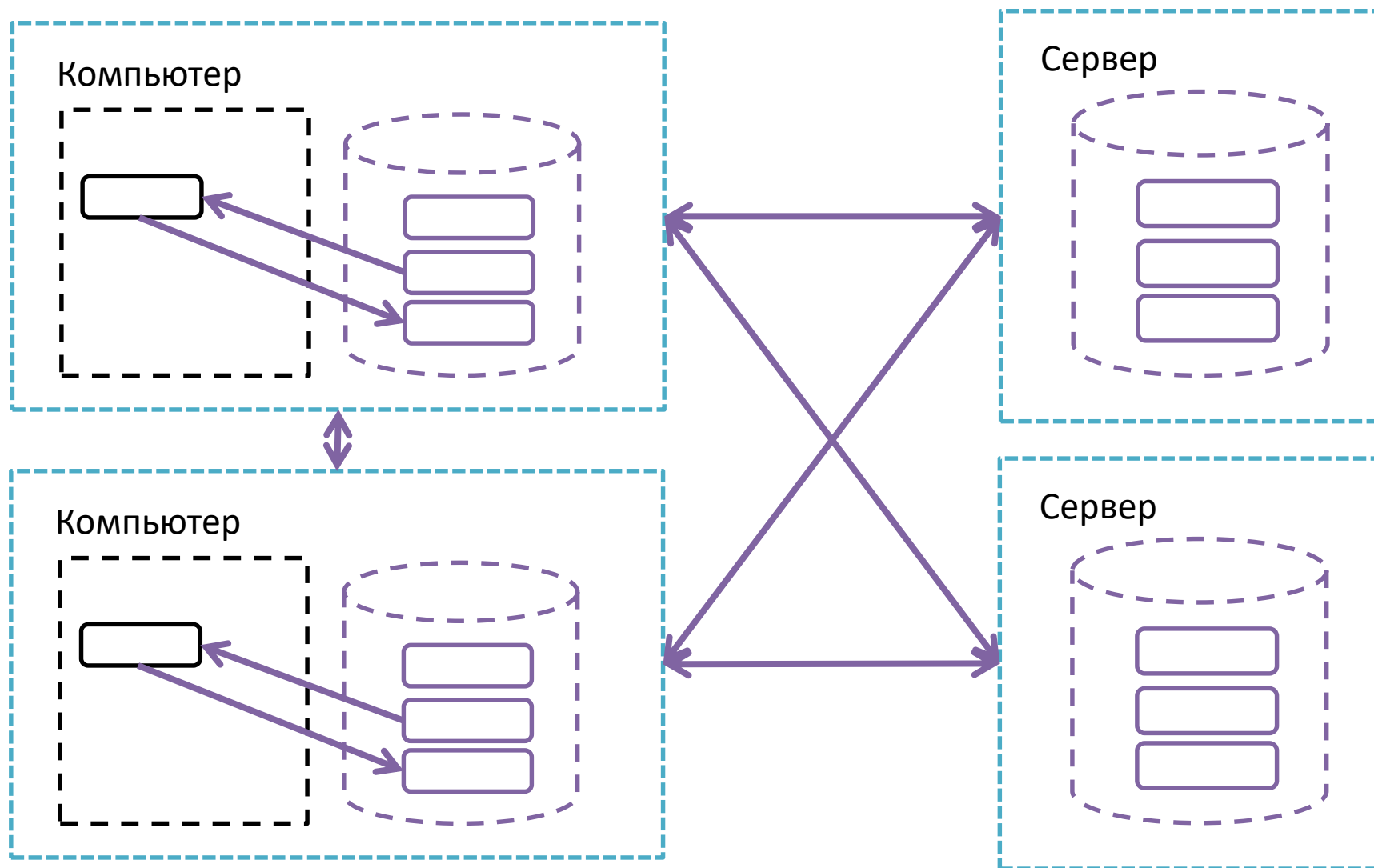


# Centralized VCS

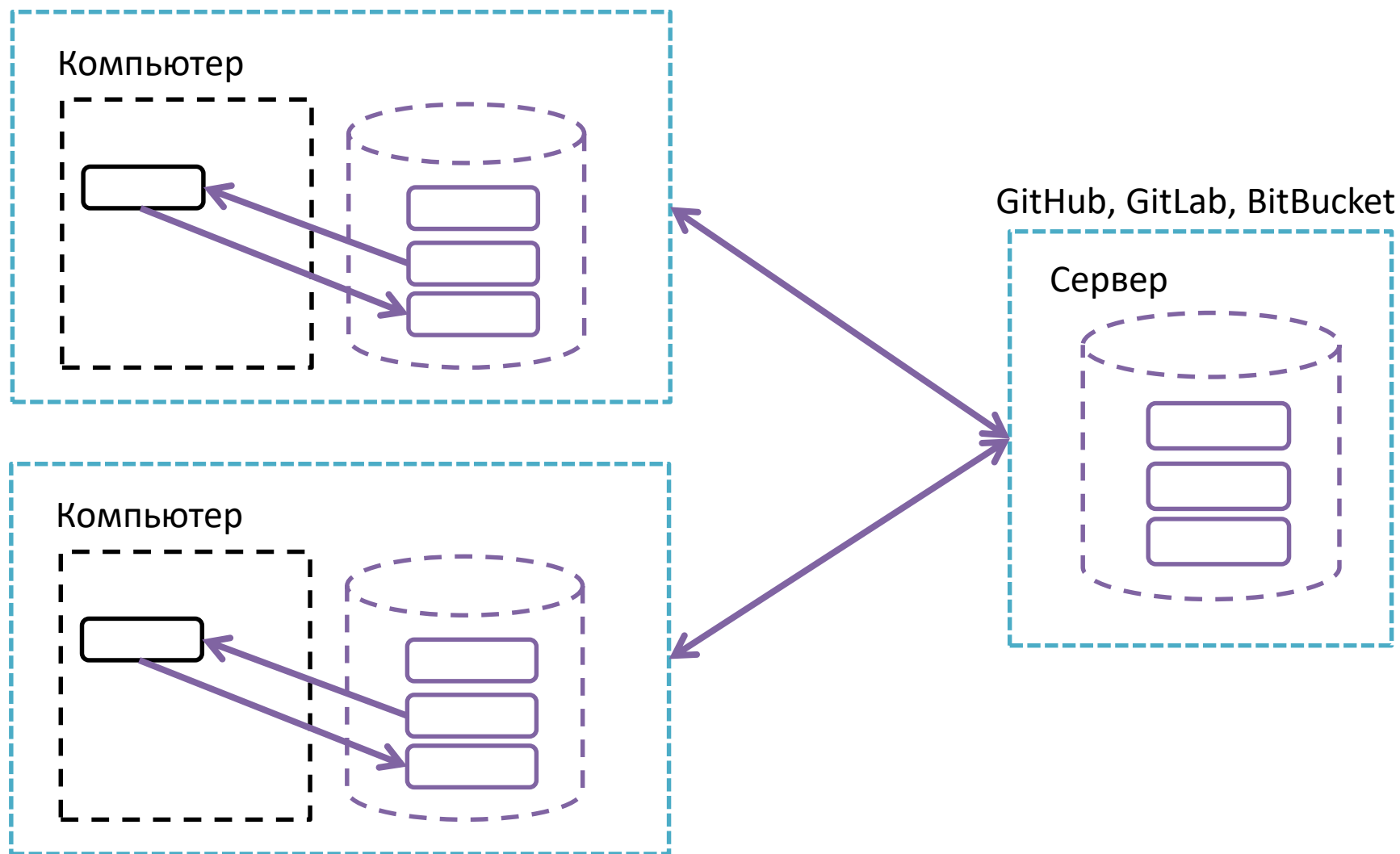




# Decentralized VCS



# Самый частый сценарий



# Git

Распределённая система контроля версий,  
одна из самых популярных на текущий  
момент

<https://git-scm.com/downloads>

# Особенности

Важно: Git хранит практически полную копию репозитория на локальном компьютере, т.е. у каждого участника есть полная история разработки

Это позволяет работать без связи с сервером

# Идентификация пользователя

Git идентифицирует пользователя по двум параметрам:

1. Имя пользователя
2. Email

# Git

Уровни настроек:

- системные
- глобальные
- локальные

# Git

## Глобальные настройки

```
git config --global user.name 'Student'
```

```
git config --global user.email student@itpark.ru
```

Без этого флага настройки будут локальными

# Создание репозитория

Создание репозитория выполняется командой:

```
git init
```

В результате выполнения этой команды появится каталог `.git`, в котором и будет храниться репозиторий и конфигурационные файлы



# Git

Локальные настройки (для конкретного проекта)

```
git config user.name "Student"
```

```
git config user.email student@localhost
```

# Схема работы

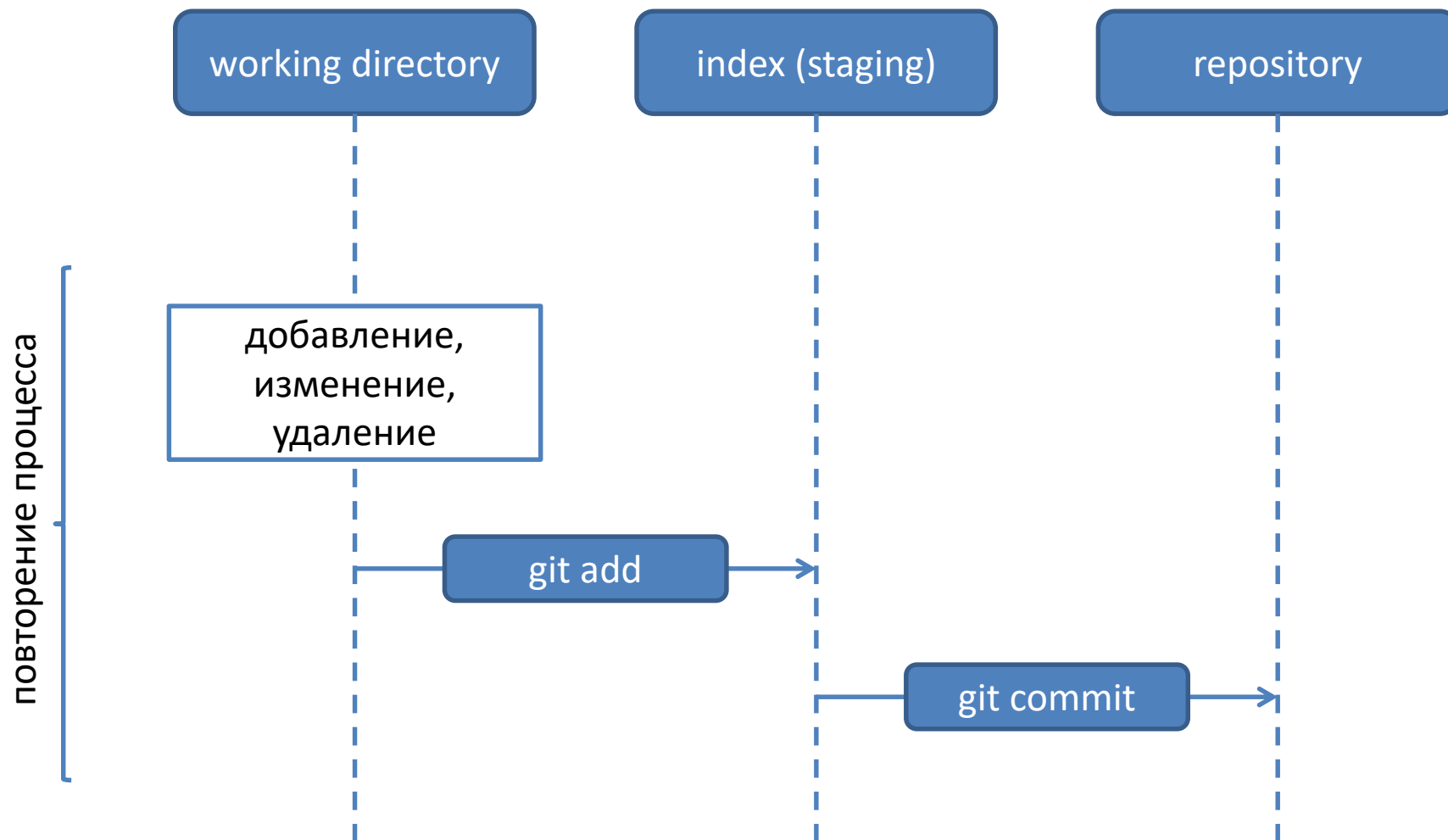
1. Git отслеживает только те файлы, которые мы ему укажем (нужно их добавить в отслеживаемые)
2. Git фиксирует состояния только тогда, когда мы это явно укажем (нужно явно зафиксировать состояния файлов)

# Git

## Состояния:

- **modified** (файл добавлен/удалён/изменён)
- **staged** (изменения добавлены в список для фиксации)
- **committed** (изменения зафиксированы)

# Упрощенная схема работы



# Упрощённая схема работы

`git status` # просмотр статуса (не обязательно)

# создаём файл README.md

`git status` # просмотр статуса (не обязательно)

`git add README.md` # добавляем в index

`git status` # просмотр статуса (не обязательно)

`git commit -m "Добавлен README"` # фиксируем

`git status` # просмотр статуса (не обязательно)

`git log` # история

# Сокращённый формат

`git commit -a -m "Сообщение коммита"`

Делает `git add` перед коммитом, но при этом не добавляет новые файлы (их нужно добавлять отдельно через `git add`)

**IGNORE**

# Игнорирование файлов

`.gitignore` – файл, в котором прописываются игнорируемые пути

`*.log` – все файлы с расширением log

`tmp/` - все файлы из каталога (`tmp` может быть в любом подкаталоге)

`/build/` - относительно `.gitignore`



# .gitignore

**.gitignore** нужно тоже хранить в репозитории

Берите уже готовый из (см. Go Courses):

<https://gist.github.com/coursar>

# .gitignore



coursar / [.gitignore](#)

Created 21 seconds ago

[Go Courses](#)

1 file

0 forks

0 comments

0 stars

```
1  .idea
2
3  # Binaries for programs and plugins
4  *.exe
5  *.exe~
6  *.dll
7  *.so
8  *.dylib
9
10 # Test binary, built with `go test -c`
```

**REMOTES**

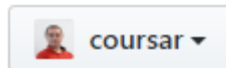
# Git remote

GitHub – бесплатный сервис, позволяет хранить публичные и приватные (с 2019 года) репозитории

См. видео по работе

# Git remote

Owner



Repository name \*

/ go-demo ✓

Задаёте только имя  
(должно соответствовать проекту)

Great repository names are short and memorable. Need inspiration? How about **ideal-happiness**?

Description (optional)



**Public**

Anyone can see this repository. You choose who can commit.



**Private**

You choose who can see and commit to this repository.

Все настройки оставляете как здесь

Skip this step if you're importing an existing repository.



**Initialize this repository with a README**

This will let you immediately clone the repository to your computer.

Add .gitignore: None ▾

Add a license: None ▾



Create repository

# Git remote

...or push an existing repository from the command line

```
git remote add origin https://github.com/coursar/go-demo.git  
git push -u origin master
```



Важно:

1. push можно делать только если у вас есть хотя бы один commit
2. **push -u origin master** нужно делать только один раз (дальше можно писать просто **push**)

**ДЗ**

# ДЗ 1: Расход топлива

Напишите программу, которая запрашивает расход топлива на 100 км и объём имеющегося топлива.

На выходе - приблизительная оценка расстояния в км, на которое хватит топлива.



## ДЗ №2 Бонусы

```
sales :=[...]int{12_000, 8_000, 15_000, 8_000};
```

%5 сверх суммы каждой продажи, которая больше 10\_000 рублей (для 12\_000 5% должно быть от 2\_000)

Сосчитать общую сумму бонуса

# Важно

1. Сдавать можно только через GitHub
2. Завтра пройдём авто-тесты (сдавать  
МОЖНО ТОЛЬКО С НИМИ)

**Спасибо за внимание**

Ильназ Гильязов

2019г.