

# *Язык программирования*



*Обзор*

*расскажет Михаил В. Шохирев*

Клуб программистов  
Шадринск  
2024-2025

# О чём поGOворим



**История:** кем, когда, где и как создавался язык.

**Цели:** зачем создавался язык. Технология разработки.

**Особенности:** чем Go отличается от других языков. Корни языка.

**Компиляция** (для разных платформ) и выполнение.

**Синтаксис:** правописание и стиль. Управляющие конструкции. Данные.

**Модульность:** функции, методы. Пакеты. Объектное программирование.

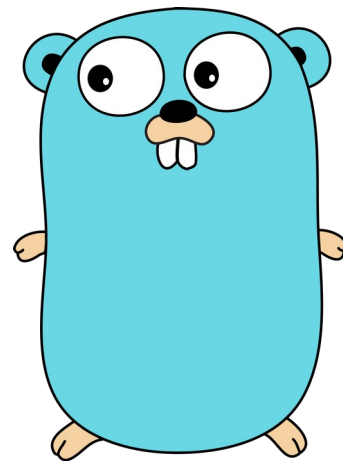
**Интерфейсы:** типы для действий (contracts), ограничения (constraints) для generics.

**МноGозадачность:** concurrency (goroutines, channels).

**Инструменты:** gofmt, go command. Go tools. IDE и редакторы.

**Применение:** где, как и почему лучше использовать Go.

**Критика:** недостатки Go и альтернативы ему.



Шустрый суслик  
**Gopher**

(автор: Renée French)

# Go = "C for the 21st century"



**Go** — простой быстро компилируемый многопоточный язык программирования со статической типизацией, ориентированный на высокопроизводительную работу в сети и эффективное многопоточное выполнение (в виде “родных” исполняемых файлов), легко осваиваемый, с многочисленными надёжными *стандартными* библиотеками, удобный для сопровождения.

## Проектировщики:

Robert Griesemer (*разработчик V8 JS engine, Java HotSpot VM, Sawzall; его учитель – Niklaus Wirth*)

Rob Pike (*разработчик window system for Unix; Plan 9, Inferno, UTF-8, языка Sawzall*)

Ken Thompson (*разработчик Unix, C, grep, ed, Plan 9, Inferno, UTF-8*)

## Разработчики:

Команда в Google + Go community.

Цель: система программирования для разработки больших надёжных высоконагруженных быстро работающих серверных программных комплексов с распараллеливанием выполнения, которые будут развиваться в течение длительного времени большой командой разработчиков.

На проектирование Go **повлияли языки** C, Oberon-2, Active Oberon, Oberon, Modula-2, Modula, Pascal, Alef, Newsqueak, Squeak, CSP, Smalltalk, Limbo, APL, BCPL, occam.

Разработчиков первоначально объединило их общее недовольство языком C++. Кроме того, они хотели сделать язык с простым синтаксисом, но отвечающий современным требованиям к разработке программ.

# Go: creators



**Ken Thompson,**  
*США*  
(«старая инженерная школа»):  
разработчик языка C, ОС Unix, Plan 9, Inferno, grep, ed, QED, UTF-8.



**Rob Pike,** *Канада*  
(«следующее поколение», специалист по concurrency):  
разработчик window system for Unix; ОС Plan 9, Inferno, UTF-8, sam, асте, языков Sawzall, Limbo, Newsqueak.



Go = C tokens +  
Oberon structure  
(& strictness)



**Robert Griesemer,**  
*Швейцария*  
(«ученик Никласа Вирта (Pascal, Modula, Oberon), европейская школа»):  
разработчик V8 JS engine, Java HotSpot VM, языка Sawzall, a programming language for vector computers, системы Strongtalk.

«Our original goal was not to create a new programming language, it was to create a better way to write programs.»

Rob Pike

Разработчики хотели не просто создать новый язык, но разработать лучшую технология разработки программ (большой командой разработчиков в течение длительного времени):

- язык спроектирован для надёжного программирования больших программных комплексов;
- *стабильная спецификация языка*: совместимость с предыдущими и последующими версиями;
- краткий и логичный *синтаксис*: легко освоить и однозначно понимать в команде;
- *строгая типизация, объявления, импорты*: компилятор контролирует программистов;
- *быстрый компилятор*: минимизирует время сборки больших программных систем;
- *легко компилировать для разных ОС и архитектур* одни и те же исходники;
- 
- *интерфейсы* с неявным соответствием: позволят расширять готовые системы;
- *композиция* вместо наследования: обеспечит независимое развитие компонентов;
- *легковесные goroutine-ы*: структурируют программу для параллельного выполнения;
- *каналы* обеспечат удобную синхронизацию и обмен данными между процессами;
- тип данных *error*: даёт все средства языка для явной обработки ошибок;
- *единый стиль* оформления исходников: задаётся утилитой **go fmt**;
- 
- богатая и надёжно работающая *стандартная библиотека*: предоставит готовые компоненты;
- *удобная система управления внешними пакетами* с идентификацией по URL;
- мощный набор стандартных инструментов: *go command*, *go tool command*;
- *открытая возможность автоматизации* за счёт расширения набора инструментов;
- 
- *открытый исходный код*: привлекает сообщество для развития системы программирования.

2007-09	началась <b>разработка</b> Go в компании Google; проектированием занимались: Robert Griesemer, Rob Pike и Ken Thompson (~ в течение 1 года).
2008-03	1-й проект (draft) спецификации языка.
2009-11-10	был официально <b>представлен</b> язык Go.
2011-03-16	go r56: based on release weekly.2011-03-07.1
2012-03-28	<b>go1.0</b> : language & a set of core libraries.
2013-05-13	go1.1: ~30%-40% performance improvement of compiled code.
2015-08-19	go1.5: compiler & runtime written entirely in Go (with a little assembler).
2017-08-24	go1.9: type aliases.
2020-02-25	go1.14: Go modules.
2022-03-15	go1.18: generics. Built-in fuzz testing.
2023-08-08	go1.21: min, max, clear built-in functions.
2024-02-06	go1.22: <i>math/rand/v2</i> package; PGO (Profile-guided Optimization) in compiler.
2024-08-13	go1.23: range over function types; <i>iter</i> , <i>unique</i> , <i>structs</i> packages.
2025-02-11	go1.24: generic type aliases; weak pointers; post-quantum cryptography; FIPS mode.
2025-08	go1.25 ( <i>upcoming release</i> ).

**Спецификация языка и стандартной библиотеки обратно совместимы с версиями Go 1.x.**

Поэтому многие крупные компании, выждав время, убедились в долговременной поддержке языка и стали применять его в своих важных проектах.

## Реализации:

1. Официальный компилятор (Google) для ОС AIX, Android, \*BSD, iOS, Linux, macOS, Plan 9, Solaris, Windows (на разных аппаратных архитектурах) и для WebAssembly (WASM).
2. gofrontend + libgo для GCC и других компиляторов.
3. **TinyGo** для embedded systems и WebAssembly.
4. **GopherJS** – кросс-компилятор из Go в JavaScript.

Поддерживаются практически все **архитектуры**: i386, amd64, ARM, RISC-V, MIPS, ppc64, ...

```
go tool dist list
```

*Лёгкая кросс-компиляция!*

**Установка** (описание <https://golang.org/doc/install>):

```
sudo apt-get install golang
```

**Обновление:**

```
go get go@1.24.3 # или go get go@latest
```

The Go **Playground** ~ интерактивное выполнение программ в браузере:

<https://go.dev/play/>

# Пример с приветом



```
package main                                // все программы принадлежат к своему пакету
import (                                    // подключить...
    "fmt"                                  // ... пакет форматированного вывода
    "os"                                  // ... и взаимодействия с ОС
)

const world = "世界"                       // UTF-8 для исходного кода и литералов

func main() {                              // с main() начинается выполнение программы
    var s string = world                  // var переменная тип = значение
                                          // len() – встроенная функция определения размера
    if len(os.Args) > 1 {                 // в os.Args[0] – имя программы
        s = os.Args[1]                   // имена с заглавной буквы экспортируются
    }
    fmt.Printf("Привет, %s!\n", s) // вызов функции из импортированного пакета
}
```

```
$ go run helloWorld.go
Привет, 世界!
$ go build helloWorld.go
$ ./helloWorld мир
Привет, мир!
$ GOOS=windows GOARCH=amd64 go build helloWorld.go
$ ls helloWorld*
helloWorld.exe helloWorld.go
```



# Пример: *web-server*



```
package main                                // на основе примера, который привёл Rob Pike:
                                           // полноценный многопоточный web-сервер
import (                                    // импортировать пакеты:
    "fmt"                                  // форматированная печать
    "log"                                  // протоколирование
    "net/http"                             // сетевая магия – в этом пакете
)

                                           // во всех идентификаторах можно использовать символы UTF-8
func こんにちはМир(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintf(w, "Привет, мир!!!\n")      // вывод ответа прямо в сеть 8-)
}                                           // точнее: в любой Writer

func main() {
    http.HandleFunc("/", こんにちはМир)    // подключить функцию-обработчик
    // слушать порт на хосте
    log.Fatal(http.ListenAndServe("localhost:12345", nil))
    // обработать ошибку и// записать в протокол
}                                           // бесконечный цикл
// аргумент nil означает, что надо использовать HandleFunc
// https://pkg.go.dev/net/http#ListenAndServe
```

Простой синтаксис. Минимум синтаксических конструкций. Однозначное выражение действий (no TIMTOWTDI). Каждое утверждение (statement) начинается с ключевого слова. Исходники в UTF-8.

Ошибки – это тип данных **error**. Нет исключений (exceptions). Есть `panic()` и `recover()`.

Нет классов, но в `struct` можно описывать поля, и для всех типов данных можно определять методы.

Есть интерфейсы (абстрактные наборы действий), и типы могут неявно соответствовать этим интерфейсам, реализуя их.

`;` служит переводом строки (line feed) — автоматически вставляется компилятором, где необходимо.

`,` запятая обязательна в конце строки в списке, если нет `)` как завершителя списка.

`_` “пустая переменная” (blank identifier) для игнорирования значения.

`:=` простое объявление с выводом типа из значения (inferred type) и инициализация переменной в функции.

Все объявленные переменные получают начальное zero value (`0`, `false`, `""`, `nil` для интерфейсов и ссылочных типов).

Имена с заглавной буквы (Capitalized) экспортируются (видны вне пакета). Область видимости имён — пакет (package).

Все имена со строчной буквы видны во всех файлах внутри одного пакета. 1 пакет = 1 каталог.

Функции: multiple return values, named return values, bare return, variadic functions (с переменным списком параметров).

**func** **init()** { } // инициализирующие функции в файлах пакета.

**defer** **f()** // отложенное исполнение функции перед завершением функции: появилось в Go.

**go** **f()** // запустить функцию как goroutine. `channel <- value`; `value <- channel`;

Безтиповые константы (untyped constants) в языке со строгой типизацией!

**rune** // тип данных для “символа” (code point) в кодировке UTF-8.

**iota** // перечисление (enumerator) именованных целых значений.

`import "package"; var declared` // если не используются — программа не скомпилируется!

```
const START, STOP = 10, 16
```

```
type PointerToInt *int
```

```
var i int = START
```

```
var (  
    p1 *int = &i  
    p2 PointerToInt  
)  
p2 = p1
```

```
loop:  
for i := i; i < STOP; i++ {  
    *p1++; *p2 += 2  
    if i < 0 { break loop }  
    fmt.Printf("i=%d outer i=%d \n", i, *p1)  
}
```

*Почти всё в синтаксисе стало понятно, когда прочитал статью про Go в Википедии. 8-)*

Синтаксис похож на языки C и Pascal

тип `*int` – указатель на `int`

`&i` – получить адрес `i`

`for` – единственная форма цикла

`“ ; ”` == перевод строки

в `if`, `for`, `switch` – нет скобок при условии

в `Printf` – много удобных `%verb` и `#adverb`

```
i=10 outer i=13  
i=11 outer i=16  
i=12 outer i=19  
i=13 outer i=22  
i=14 outer i=25  
i=15 outer i=28
```

`()` список: импортов, констант, параметров, возвращаемых значений, ...

`[]` размер массива, показатель среза  
элемент массива, среза, словаря

`[]` тип параметра в generics

`{}` блок определения

`{}` блок начальных значений

`{}` блок кода

`:` отделяет индекс или ключ от значения

`:` ставится после метки

`...` размер массива вычисляется по значениям

`...` список параметров переменной длины

`...` список аргументов переменной длины

`;` разделитель выражений в for и if

`;` перевод строки (разделитель утверждений)

`:=` объявление и присваивание (в функции)

`,` разделитель в списке

`.` разделитель объекта и метода

`<-` запись в канал и чтение из канала

```
import ( "fmt" ); const ( answer = 42 ); var ( five = 42 )
func f(x float64) float64 { return 0.0 }
```

```
array [size]int;      slice []string
array[index];        slice[index];      value = map[key]
func f[T any](a []T) T { return a[0] }; f[int](someSlice))
```

```
type struct Point { x, y int32 }
ipAddress = [4]int{127, 0, 0, 1}
func answer() int { return 42 }
```

```
gender := [2]string{ 0: "Female", 1: "Male" }
language := map[string]int { "Go": 2007 }
label:
```

```
shadrinsk := [...] float32 { 56.05, 63.38 }
func sum(numbers ...int) (sum int) { /* range numbers */ }
integers := []int{1,2,3,4,5}; sum(integers...)
```

```
for i := 0; i < n; i++ { if y := f(i); y > 0 { println(y) } }
version := 1.0; released := 2012; fmt.Println(version)
```

```
site := "https://go.dev/"
```

```
place = findLocation(latitude, longitude)
```

```
result = object.method()
```

```
channel <- value; value = <-channel
```

```
if условие {  
    действие1()  
} else {  
    действие2()  
}
```

```
switch выражение {  
    case значение-1, значение-2:  
        действие1()  
        fallthrough  
    case значение-N:  
        действиеN()  
        break  
    default:  
        действиеПоУмолчанию()  
}
```

```
if выражение; условие {  
    действие1()  
} else {  
    действие2()  
}
```

```
switch {  
    case условие1:  
        действие1()  
    case условиеN:  
        действие1()  
}
```

```
switch выражение; значение := интерфейс(type) {  
    case Тип1:  
        действие1()  
    case ТипN:  
        действие1()  
}
```

# Управление выполнением: циклы



```
// итерационный: (i := 0; i < n; i++)  
for инициализация; условие; изменение {  
    обработка(данных)  
}
```

```
// == while  
for условие {  
    обработка(данных)  
}
```

```
// бесконечный цикл  
метка:  
for {  
    обработка(данных)  
    if условие { continue }  
    if условие { break метка }  
}
```

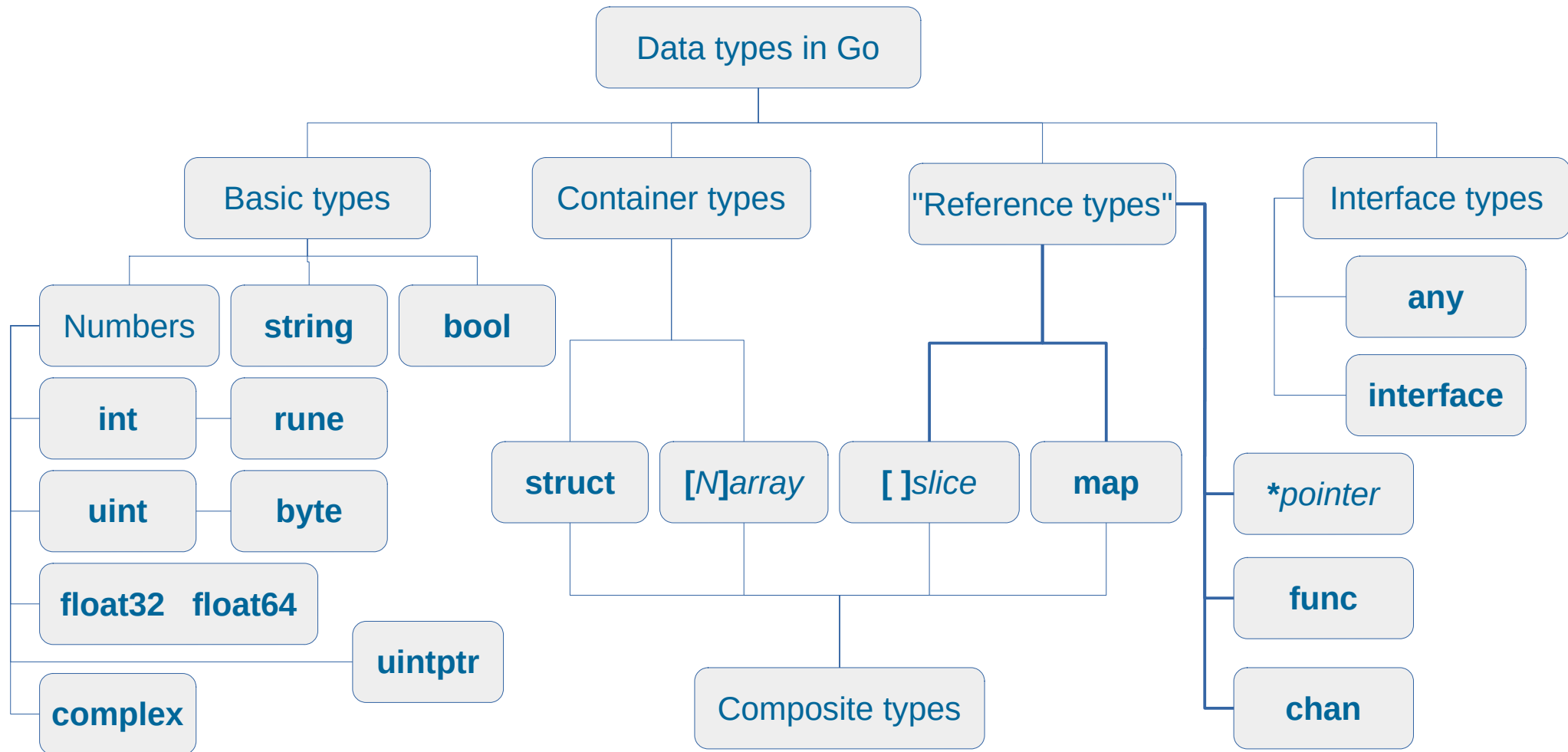
«One **for** to rule them all.»

```
// перебор целых чисел от 0 до < число  
for значение := range число {  
    обработка(данных)  
}
```

```
// перебор array или slice  
for индекс, элемент := range коллекция {  
    обработка(данных)  
}
```

```
// перебор хэша  
for ключ, значение := range map {  
    обработка(данных)  
}
```

```
// получение из channel  
for значение := range канал {  
    обработка(данных)  
}
```



# Константы: `const`



Предопределённые константы (predefined constants):

`true false nil iota`

Безтиповые константы (untyped constants):

`const constant = expression`

`const constant1, constant2 = value1, value2`

`const ( constant1 = value1, constant2 = value2 )`

```
const (  
    e = 2.71828182845904523536028747135266249775724709369995957496696763  
    pi = 3.14159265358979323846264338327950288419716939937510582097494459  
)
```

Константы с заданным типом (typed constants):

`const constant Type = expression`

```
const b byte = 0xf           // байт  
const x complex128 = 2+5i    // комплексное число  
const Big float64 = 1 << 100 // binary 1 with 100 zeros  
const i int32 = -273         // целое  
const Go rune = ' 碁 '      // символ  
const language string = "Go" // строка
```



# Операции: объявление и присваивание



**var** *переменная* **Type**

При объявлении новой переменной **всегда** есть начальное значение (zero value).

```
var i      int      // 0
var G, o   rune     // 0, 0
var s      string   // ""
var (
    tube   chan string // nil
    ok     bool        // false
    f      float64     // 0.0
)
```

```
var answer = 42      // тип int выведен из присвоенного значения
```

// объявление новой переменной и присваивание значения (в функции)

```
j := 0      // тип выводится из присваиваемого значения
t, f := true, false // параллельное присваивание
```

// присваивание значения уже объявленным переменным

```
s = "Go"      //
G, o = 'G', 'o' // параллельное присваивание (tuple assignment)
i, j = j, i    // обмен значений i и j
```

Структура (**struct**) — набор разнотипных полей

```
type Structure1 struct {           // объявление типа
    field1 Type1
    field2, field3 Type2
}

s := Structure1 {                 // объявление переменной и
    field1: initial_value1_of_Type1, // инициализация полей значениями
    field2: initial_value2_of_Type2, // по именам (не всех) полей
}

var s2 Structure1                // объявление переменной
s2 = s1

// объявление переменной и инициализация полей значениями по порядку
s3 := Structure1{s2.field1, s2.field2, s2.field3} // следования полей

var s4 Structure1 = Structure1{} // инициализация пустой структурой
```

# Ссылочные типы: reference types



Когда переменные-указатели передаются в функции, их значения могут быть изменены.

**pointer** ~ указатель:

```
var v BaseType      // переменная типа BaseType  
var p *BaseType     // указатель на переменную типа BaseType  
p = &v              // ссылка на значение переменной типа BaseType  
*p                  // значение переменной типа BaseType по ссылке на v
```

**slice** ~ динамический массив может изменять свой размер:

```
s := []Type
```

**map** ~ хэш | ассоциативный массив | словарь | отображение | «мапа»:

```
m := map[KeyType]ValueType{ key: value }
```

**function** ~ функция:

```
func f(parameter Type) returnValueType { /* код */ }
```

**channel** ~ канал:

```
ch chan Type
```

**[размер]Type** // массив (**array**) определённой длины:

```
var punchCard [80]rune // 80 * 0
localhost := [4]int {127, 0, 0, 1}
gender    := [2]string { 0:"Female", 1:"Male" }
location  := [...]float32 { 56.05, 63.38 }
```

**[]Type** // срез (**slice**) – динамический массив:

```
primes := []int{2, 3, 5, 7, 11, 13, 17, 19, 23}
messages := make([]string, 0, 1024)
messages = append(messages, "OK")
```

**map[KeyType]ValueType** // **map** = хэш = ассоциативный массив = словарь:

```
languages := map[string]int { "Go": 2007 }
languages["Kotlin"] = 2011
```

```
type Coordinates map[[2]float32]string
places := make(Coordinates)
places[location] = "Шадринск"
places[[2]float32{33.54, -118.05}] = "Norwalk"
```

# Функции: `func main()`, `func init()`



// Главная функция в пакете `main`, с которой начинается выполнение программы.

`package main`

**`func main()`** {

`/*`

`*/`

}

// в каждом пакете может быть несколько «инициализирующих» функций,

**`func init()`** {

`/*`

`*/`

}

// которые выполняются при загрузке пакета в порядке их описания

**`func init()`** {

`/*`

`*/`

}

# Функции: func



```
// функция без возвращаемого значения = процедура
func f1(param1 T1) { /* ... */ } // no return value
f1(arg1)
```

```
// функция с одним возвращаемым значением
func f2(p1, p2 T1, p3 Type3) ReturnType1 { /* ... */ return res }
result = f2(a1, a2, a3)
```

```
// функция с одним именованным возвращаемым значением
func f3(p1, p2, p3 T3) (returnValue1 RT1) { /* ... */ return }
result = f3(a1, a2, a3)
```

```
// функция с несколькими возвращаемыми значениями: возможно, именованными
func f4(p1 T1, p2 T2) (rV1 RT1, rV2 RT2) {
    /* ... */ return res1, res2
}
result1, result2 = f4(a1, a2)
```

```
// функция с переменным списком параметров
func f5(p ...T) ReturnType { /* ... */ } // variadic function
arguments := []siceOfValues{v1, v2, v3, v4, v5}
result = f5(arguments...)
result = f5(a1, a2, a3)
```

# Функции: func type



```
// функция как тип
// type FuncType func(ParamerType1, PT2) ReturnValueType
// type F1 func(int, int) int

// у любой функции есть тип, например: func(int, int) int
func add(x, y int) int { return x+y }

// функция как значение переменной
var f0 F1 = add
fn := func() { fmt.Println("anonymous") } // анонимная функция типа func()

// функция как возвращаемое значение
func returnsFunc() F1 { return add }
f1 := returnsFunc()
y := f1(4, 2)

// функция как параметр
func receivesFunc(a, b int, f F1) (r Return_type) { r:= f(a, b); return r }
sum = receivesFunc(21, 21, add)
production = receivesFunc(21, 21, func(x, y int) int { return x*y } )

// определение и вызов анонимной функции
func() { fmt.Println("lambda") }() // lambda типа func()
```

# Методы: func (t T) f()



К любому типу данных можно присоединить поведение с помощью методов:

```
func (object Type) method(parameters) { /* body */ }
```

```
type Temperature float32
func (t Temperature) Celsius() Temperature { return (t-32.0)*(5.0/9.0) }
var f Temperature = 37.0
c := f.Celsius() // 2.777778
```

```
type Album struct { name, artist string; year, length int }
a := Album{"Pink Floyd", "Dark Side of the Moon", 1973, 44}
type TapeRecorder struct {
    // ...
}
func (r TapeRecorder) play(a Album) {
    fmt.Printf("Playing album '%s' by '%s' for %d minutes...\n",
        a.name, a.artist, a.length)
}
recorder := TapeRecorder{/* начальные значения */}
recorder.play(album)
```



# Объектное программирование: type struct + func



// Нет классов, но можно описывать типы объектов на основе struct:

**package user**

```
type User struct {  
    login string  
    email string  
}
```

// К такому типу можно присоединить поведение с помощью методов:

```
func (u User) Login() string { return u.login }
```

```
func (u User) Email() string { return u.email }
```

```
func (u *User) SetEmail(mailbox string ) { u.email = mailbox }
```

```
    // Это не конструктор, а просто обычная функция, которую назвали New  
func New(l, e string) (u User) { u = User{login: l, email: e}; return u }
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

**package main**

```
import "github.com/mike-shock/go/sample/oop/user"
```

```
func main() {
```

```
    mike := user.New("mshock", "mshock@caiman-club.org")
```

```
    mike.SetEmail("librarian@caiman-club.org")
```

```
    fmt.Printf("'%v' '%v'\n", mike.Login(), mike.Email())
```

```
}
```

# Интерфейсные типы: interface types



**interface** – это абстрактный тип данных, у которого может быть набор методов (set of method signatures):

```
type Messenger interface {           // Basic interface
    Send(message string) error
    Receive() string
}
```

```
// объявление корректно: это переменные абстрактного типа,
// которым можно присваивать значения, соответствующие контракту (интерфейсу);
// но пока у них нет конкретного типа (реализации), а значение = nil
var icq, skype, whatsapp, viber, signal Messenger
```

```
// Конкретный тип будет неявно соответствовать ранее описанному интерфейсу,
// если реализует все методы этого интерфейса
type Telegram struct { }
func (t Telegram) Send(m string) error { return nil }
func (t Telegram) Receive() (m string) { return m }
```

```
// у абстрактного типа динамически появляется конкретный тип (underlying type)
// и конкретное значение
var telegram Messenger = Telegram{}
telegram.Send("Интерфейсы в Go")           // вызывается реализованный метод
```

# Интерфейсные типы: interface types



```
// Интерфейс также может включать в себя (embed) другие интерфейсы:
type File interface{
    Reader           // Embedded interfaces
    Writer
    Seeker
    ReaderAt
    WriterAt
    Closer
}

type Reader interface {
    Read(b []byte) (n int, err error)
}

// Конкретный тип может соответствовать (implement) нескольким интерфейсам
func (t Telegram) Read(b []byte) (int, error) { return 0, nil }

// any == пустой интерфейс
type AnyObjectSatisfyMe interface{} // ему соответствует любой объект
```

# Интерфейсы: пример



```
type Flyer interface { fly() string }           // 1-й интерфейс с методом
// все типы, которые реализуют метод fly(), будут соответствовать типу Flyer

type Bird struct { Name string }                // пользовательский тип Bird
func (b Bird)fly() string {                   // соответствует интерфейсу Flyer
    return "flying..."
}

type Swimmer interface { swim() string }      // 2-й интерфейс с методом

type Penguin struct { Name string }            // пользовательский тип Penguin
func (f Penguin)swim() string { return "swimming..." } // соответствует сразу
func (b Penguin)fly() string { return "I can fly under water!" } // двум интерфейсам

var s = Bird{"Sparrow"}
var p = Penguin{"Gentoo"}

birds := []Flyer{s, p, Bird{"Dove"}}          // Flyer – это тип данных
for _, b := range birds {                       // polymorphism
    fmt.Println(b, b.fly())
}
```

# Интерфейсные типы: constraints



Интерфейсные типы также могут описывать ограничения (constraints):

```
type Numeric interface { // non-basic: has a type set (union)
    ~int | ~int8 | ~int16 | ~int32 | ~int64 | ~uint | ~uint8 | ~uint16 | ~uint32 |
    ~uint64 | ~uintptr | ~float64 | ~float32           // General interface
}

// Интерфейс может сочетать ограничения, основные (basic) и
// пользовательские (non-basic) типы и интерфейсы, а также методы
type SpecialNumber interface {
    Number
    IsSpecial() bool
}

type ComparableTypes interface {
    comparable
    SomeOtherType
}
```

# Обобщённые типы: generics



// Generics описываются с помощью ограничений (constraints) на обобщённый тип в функции:

```
func First[T any](a []T) (result T, err error) {
    if len(a) == 0 {
        return result, errors.New("Slice is empty!") // пустой результат и ошибка
    }
    return a[0], err // 1-й элемент и nil (ошибки нет)
}
func Last[T any](a []T) (result T, err error) {
    if len(a) == 0 {
        return result, errors.New("Slice is empty!")
    }
    return a[len(a)-1], err // последний элемент и nil
}

sliceOfIntegers := []int{1, 2, 3, 4, 5}

// если тип параметра можно вывести из переменной, то его можно не указывать
first, err := First[int](sliceOfIntegers)
last, err := Last(sliceOfIntegers)

sliceOfStrings := []string{"Вышел", "зайчик", "погулять"}

fmt.Println(First(sliceOfStrings))
fmt.Println(Last[string](sliceOfStrings))
```

Пакет — это набор (логически связанных) исходных файлов, расположенных в одном каталоге. В начале каждого файла должно описываться его принадлежность к пакету фразой

```
package packageName // site/path/packageName
```

Пакет — единица видимости имён (типов, констант, переменных, полей, функций):

- Все имена видны во всех файлах одного пакета.
- Имена в пакете, начинающиеся с *Заглавной буквы* экспортируются: они видны в программе, которая импортировала пакет фразой

```
import "packageName" // это строка
```

```
var result packageName.Type = packageName.Func(packageName.Const, packageName.Var)
```

Пакет **main** — это специальное имя пакета, которое означает, что этот пакет содержит код, который будет скомпилирован в двоичный исполняемый файл. В одном из файлов (обычно, в `main.go`) этого пакета должна быть функция **main()**.

Пакеты (не из стандартной библиотеки) могут располагаться где угодно, их полные адреса (локальные пути или URL) содержатся в файле

```
go.mod
```

Модуль — это набор пакетов, которые распространяются (с определённым номером версии) как единое целое. Модули могут загружаться прямо из систем управления версиями исходников или с общедоступных серверов.

Модуль идентифицируется путём к модулю (module path), который объявляется в файле **go.mod** вместе с информацией о зависимостях модуля.

```
# создать файл go.mod с именем модуля  
go mod init path/to/module/moduleName
```

Например:

```
go mod init caiman.org/go/mshock/presentation
```

Главный каталог модуля (module root directory) — это каталог, содержащий файл **go.mod**. Когда модуль состоит из нескольких пакетов, они располагаются в подкаталогах главного каталога модуля.



# Модули: `go mod <command>`



Главный каталог модуля (module root directory) — это каталог, содержащий файл **go.mod**. Когда модуль состоит из нескольких пакетов, они располагаются в подкаталогах главного каталога модуля.

```
go mod init path/to/module/moduleName
```

```
module path/to/module/moduleName
```

```
go 1.24.4
```

Файл **go.sum** содержит контрольные суммы для тех версий модулей, от которых прямо или косвенно зависит этот модуль.

```
go mod tidy           #
go mod download       #
go mod verify         #
go mod graph          #
go mod why            #
```

- **concurrency** ~ одновременность = взаимодействие множества процессов, которые (в какие-то моменты) выполняются одновременно

"**Concurrency** is the *composition* of independently execution things." — *Rob Pike*

**Concurrency** — это способ структурировать программу, согласовывая взаимодействие процессов (возможно, выполняющихся одновременно).

**Concurrency** — это о том, как **организовать** одновременную обработку многих вещей («**dealing** with a lot of things at once»).

В программе, спроектированной на основе **concurrency**, процессы не обязательно будут автоматически выполняться параллельно (например, из-за аппаратных ограничений).

- **parallelism** ~ параллелизм = параллельное выполнение множества процессов

"**Parallelism** is the simultaneous *execution* of multiple things." — *Rob Pike*

**Parallelism** — это параллельное выполнение нескольких (независимых, возможно, взаимосвязанных) процессов.

**Parallelism** — это о том, как **выполнить** обработку многих вещей параллельно («**doing** a lot of things at once»).

Программа, спроектированная на основе concurrency, организует взаимодействие процессов, учитывая их возможный **параллелизм**.

Многозадачность в Go реализована на основе CSP (communicating sequential processes, C. A. R. Hoare, 1978).

Для управления многозадачностью в язык встроено несколько механизмов:

- Сопрограммы (goroutines) для распаралелливания выполнения:  
`go f()` // запустить любую функцию как процесс
- Каналы (channel) для обмена данными и синхронизации выполнения:  
`channel <- value` // отправить значение в канал и ждать  
`value = <-channel` // ждать и получить значение из канала
- Выбор (select) для обработки нескольких потоков данных через каналы:  
`select {`  
`case <-ch2: // ожидать, когда что-то появится в канале #1`  
`case v2 := <-ch2: // ждать и получить значение из канала #2`  
`case ch3 <- v3: // отправить в канал #3`  
`default: // обработать другое событие`  
`}`

# МноГозадачность: goroutines



Сопрограммы (goroutines) — это легковесные потоки, которые выполняются одновременно и управляются главным потоком (main go thread).

```
func f(n int) { println(n) }
```

```
func main() {  
    for n := range 5 {  
        go f(n+1) // запустить 5 экземпляров f() параллельно с main()  
  
        // можно запустить анонимные функции  
        go func () { println(n+1) }()  
    }  
  
    time.Sleep(1 * time.Second)  
    println("Вышел зайчик погулять.")  
}
```

```
// объявить переменную для канал обмена данными указанного типа
var channel chan T

// создать канал
channel = make(chan T, размерБуфера)

// отправить значение в канал
channel <- value

// и ждать, пока не будет прочитано значение из канала

// ждать, пока не будет записано значение в канал
// прочитать значение из канала в переменную
value := <-channel

// прочитать из канала, игнорируя значение
<-channel

close(channel) // закрыть канал
```

# Каналы: пример



```
var club chan string // объявить канал для строк, значение nil
club = make(chan string) // выделить память каналу для строк

// club <- "Разговор о Go" // будет deadlock !!!
// отправить в канал значение параллельно
go func() { club <- "Предложен разговор о языке Go" }()
received := <-club // получить значение из канала в переменную

go sendMessage(club, "Разговор о языке Go запланирован.")
go sendMessage(club, "Разговор о языке Go состоялся.")

m2, m1 := <-club, <-club // получить 2 сообщения
close(club)
message, ok := <-club // проверить доступность канала
if !ok { // канал закрыт
    fmt.Println("Разговор завершился.")
}

func sendMessage(ch chan<- string, s string) {
    ch <-s
}
```

# Инструменты: *go command*



**go** *command* [*arguments...*]

The *commands* are:

<b>bug</b>	start a bug report
<b>build</b>	compile packages and dependencies
<b>clean</b>	remove object files and cached files
<b>doc</b>	show documentation for package or symbol
<b>env</b>	print Go environment information
<b>fix</b>	update packages to use new APIs
<b>fmt</b>	gofmt (reformat) package sources
<b>generate</b>	generate Go files by processing source
<b>get</b>	add dependencies to current module and install them
<b>install</b>	compile and install packages and dependencies
<b>list</b>	list packages or modules
<b>mod</b>	module maintenance
<b>work</b>	workspace maintenance
<b>run</b>	compile and run Go program
<b>telemetry</b>	manage telemetry data and settings
<b>test</b>	test packages
<b>tool</b>	run specified go tool
<b>version</b>	print Go version
<b>vet</b>	report likely mistakes in packages

## ▲ IDEs And Text Editor Plugins @ go.dev:

- **Visual Studio Code** + plug-in (Microsoft)
- **GoLand** (IDE by JetBrains)
- **LiteIDE** (open source and cross-platform Go IDE)
- **Komodo IDE** (cross-platform IDE with built-in Go support)
- **Komodo Edit** + plug-in (cross-platform text editor)
- **jEdit** (open-source, cross-platform text editor: Java)
- **Geany** (free cross-platform programmer's text editor)
- **Notepad++** (text & source code editor: Windows)
- **Kate** (cross-platform text editor with Go support out-of-the-box: KDE)
- **Sublime Text** (commercial text editor: macOS, Windows, Linux)
- **TextMate** (commercial text editor: macOS)
- **vim** & **Neovim**+ vim-go plugin (open-source, cross-platform text editor)

... Atom, BBEdit, Chime, CodeLobster IDE, Coding Rooms, emacs, Gitpod, IDEone, Jdoodle, OneComplier, OnlineGDB, Micro, Nova, zed, Zeus IDE, ...



Go **применяется** в (*> 40% IT technology companies worldwide*):

**Alibaba**, Amazon, American Express, **Apple**, Armut (C# → Go), **Baidu**, BBC, bitly, ByteDance, Canonical, Capital One, CERN, **Cloudflare**, Cockroach Labs, Curve, DataDog, Dailymotion, **Docker**, DropBox (Python → Go), GitHub, **Google**, gov.uk, Heroku, **IBM**, InfluxDB, Intel, K8s, Kubernetes, **Meta**, **Microsoft**, Monzo Bank, **Mozilla** (Rust & Go), Netflix (Java → Go), New York Times, **Oracle**, PayPal, Pinterest, **Reddit**, Riot Games, Slack, Salesforce (Python, C → Go), SendGrid, Stream (Python → Go), SoundCloud, Terraform, The Economist, The New York Times, **Twitch**, Uber, Walmart, YouTube, **X**, *многих других организациях и проектах open-source.*

В **России** (всеми крупными компаниями):

**Яндекс**, ЦУМ, УГМК-Телеком, Точка, Т-Банк, Совкомбанк Технологии, **СберТех**, Ростелеком, Онлайн-кинотеатр Иви, **МТС**+MWS, Магнит.Tech, Лаборатория Касперского, ИТ-Холдинг Т1, ИнГосСтрах Банк, Домклик, Группа Астра, ГНИВЦ, АйТи Инновация, YADRO, X5 Digital, **VK**, Viasat Tech, Tutu, **Tele2**, Selectel, S8.Capital, **Ozon**, Okko, **Mail.ru**, Lamoda Tech, iSpring, IBS, Cloud.ru, Boxberry: IT, **Beeline**, **Avito**, 2GIS и *многие другие...*

## Области применения ▲ Go:

- DevOps & SRE ▲ (Development Operations & Site Reliability Engineering)
- Cloud & Distributed Network Services ▲
- Web Development ▲ (frameworks, toolkits, engines, servers)
- System Automation & CLIs ▲, Utilities & Stand-Alone Tools
- ... multi-platform GUI apps (fyne.io)
- ... AI clients via API & libraries (TensorFlow in Go, etc.)
- ... IoT & embedded systems (TinyGo)

## Software на Go:

Allegro (eCommerce), **AmneziaWG**, **AKS** [Azure Container Service] @ Microsoft, AresDB @ Uber, Buffalo (web framework), Caddy (web server), CockroachDB, Digger (IaC), **Docker**, Drone (CD), **DropBox** (backend), ent @ Meta, Genkit, Flamingo (web framework), Gin (web framework), Gitlab Runner, **Google Cloud**, Gorilla (web toolkit), **Grafana**, Hugo (website engine), InfluxDB, JuiceFS, **Kubernetes**, LangChainGo, **LocalAI**, LXD @ Canonical, **Mattermost** (messaging platform), Monzo (banking app), **Ollama** (89%), **PayPal**, **Prometheus** (monitoring & alerting toolkit), Rend (large scale data caching @ Netflix), **SoundCloud**, Soundscape (music streaming server), **Terraform** (IaC), Timesheets (project management), Twitch (live-streaming), VITESS @ YouTube, **Zabbix** agent2, ...

Top 60+ Open-source Apps Written with Golang in 2024 ▲

**TIOBE index ▲** (since 2009):

Now: **#7** (Apr 2025) ← **#13** (Nov 2023)

Highest Position (before): **#7** (Apr 2024)

Lowest Position: **#122** (May 2015)

Language of the Year: 2009, 2016

**Cloudflare Radar ▲** API Client Language popularity: **#1** (2024)

**GitHub Octoverse ▲** Top 10 fastest growing languages in 2024: **#3**

**JetBrains ▲** Language Promise Index: **#4** (2024)

**IEEE Spectrum ▲** Top Programming Languages: **#8** (2024)

**Crossover ▲** Top 10 In-Demand Programming Languages for 2025: **#10**

**ZDnet ▲** The most popular programming languages in 2025: **#10**

**StackOverflow ▲** **#11** Most popular techs: language (professionals) (2024)

**PYPL ▲**: **#12** (May 2025)

**RedMonk ▲** Programming Language Rankings: **#12** (Jun 2024)

**Statista ▲** Most used programming languages among developers (2024): **#12**

**GeeksForGeeks ▲** 20 Best Programming Languages to Learn in 2025: **#13**

Is Golang Still Growing?  
Go Language Popularity  
Trends in 2024  
@ JetBrains

*\* я не учитывал в индексах позицию HTML/CSS*

При разработке и развитии программных комплексов, написанных на Go, выделяют следующие недостатки:

- Синтаксис слишком простой, мало syntactic sugar.
- Синтаксис непривычный: использование [] в типах параметров и в интерфейсах при ограничениях снижает читабельность.
- Нет настоящего ООП.
- Ограниченный вывод типов (inference): явное указание типов параметров: reduces the simplicity and reduces boilerplate code benefits.
- Ограничения (constraints) задаются только интерфейсами и могут ограничивать гибкость generics в определённых сценариях работы.
- Явная обработка ошибок: смущает разработчиков, кто привык к исключениям (многословность, нет прерывания потока выполнения).
- Нет перегрузки функций (function overloading).
- no support operator overloading or keyword extensibility, no support immutability declarations.
- Не хватает значений по умолчанию для параметров функций (default values for arguments).
- Использование nil и weak type safety?
- Диспетчер сопрограмм (goroutines scheduler) управляет выполнением goroutines, что может привести к недетерминированному поведению.
- Сборщик мусора иногда вносит недопустимые задержки при выполнении программ.
- Странный шаблон при форматировании даты и времени.
- Нет проверки значений на соответствие перечислению, объявленному через iota.
- В некоторых случаях требуется более низкоуровневое управление распределением памяти, как в Rust.
- 

Многие компании сочетают применение Go с использованием других новых языков, таких как Rust (Rust vs. Go: Why They're Better Together ▲).

# Мои впечатления от Go



Синтаксис простой, но с некоторыми непривычными конструкциями.  
Логично спроектирован, предсказуем. Исходники хорошо понимаются.  
Непривычно после динамического Ruby: все объявления и преобразования надо делать явно.

Очень строгий: переменная не используется – код компилироваться не будет!  
Strong typing и другие строгости важны для надёжности больших программ.  
Явная работа с ошибками дисциплинирует программиста: о них надо думать постоянно.  
Убедился в преимуществах отказа от традиционного ООП в пользу объектного подхода в Go.  
Интерфейсы в Go — основа динамичности и гибкости при разработке.

Довольно низкоуровневый: напоминает Си – вспомнил молодость!  
Очень быстро компилируется. Удобно сразу выполнить: `go run program.go`  
Легко скомпилировать исполняемую программу для другого «железа» и ОС.  
Действительно очень быстро выполняется: переписывал на Go с Python и Ruby.

Много *стандартных* библиотек – на все случаи жизни.  
Легко подключать сторонние модули.  
Хорошая документация на библиотеки (с исполняемыми примерами).  
Много сайтов с примерами – изучать легко.

*Наверное, это последний язык, разработанный «классиками», которые создали Unix.*



# КНИГ МНОГО (лучше читать на английском)



- [go.dev/](https://go.dev/) // Официальный сайт языка
- [go.dev/play/](https://go.dev/play/) // Go Playground ~ выполнение в браузере
- [go.dev/ref/spec](https://go.dev/ref/spec) // **Спецификация языка (!!!)**
- [go.dev/doc/](https://go.dev/doc/) // Документация
- [go.dev/doc/code](https://go.dev/doc/code) // How to Write Go Code
- [pkg.go.dev/std](https://pkg.go.dev/std) // стандартная библиотека
- [gobyexample.com](https://gobyexample.com) // Go в примерах
- [go.dev/doc/modules/layout](https://go.dev/doc/modules/layout) // Структура каталогов
- [github.com/golang-standards/project-layout](https://github.com/golang-standards/project-layout) // Стандартный макет [большого] Go проекта
  
- [tour.golang.org](https://tour.golang.org) // Экскурсия по возможностям Go
- [golangdocs.com](https://golangdocs.com) // примеры конструкций
- [appliedgo.net/why-go/](https://appliedgo.net/why-go/) // 15 Reasons I Love Go
- [awesome-go](https://awesome-go.com) // libraries for everything...
  
- [go.dev/doc/effective\\_go](https://go.dev/doc/effective_go) // "Effective Go" бесплатная web-книга
- [gopl.io](https://gopl.io) // "The Go Programming Language" by A.A.A.Donovan & B.W.Kernighan
  
- [w3schools.com/go/](https://w3schools.com/go/) @ w3schools // Справочник
- Самоучитель по Go для начинающих @ proglib.io // Самоучитель
- Дорожная карта Go-разработчика @ proglib.io // План изучения
- [lyceum.yandex.ru/go](https://lyceum.yandex.ru/go) // Яндекс-лицей: Программирование на Go
- Книги по Go @codelibs.ru //
  
- [tinygo.org](https://tinygo.org) // TinyGo

# Готов ответить на вопросы



Ссылка на презентацию

# ???





communicating sequential processes ~ взаимодействие последовательных процессов

concurrency ~ свойство программы, допускающее одновременное выполнение нескольких вычислительных процессов

CSP = communicating sequential processes

gopher ~ программист на Go

goroutine ~ подпрограмма, возможно, выполняемая параллельно

multitasking ~ многозадачность

parallelism ~ параллелизм = параллельное выполнение вычислений

process ~ процесс

subprocess ~ подпроцесс

subtask ~ подзадача

task ~ задача