```
// Ключевое слово package присутствует в начале каждого файла.
// Main – это специальное имя, обозначающее исполняемый файл, нежели библиотеку.
package main
// Import предназначен для указания зависимостей этого файла.
import (
    "fmt"      // Пакет в стандартной библиотеке Go
    "io/ioutil" // Реализация функций ввод/ввывода.
    "net/http" // Да, это веб-сервер!
    "strconv"  // Конверт. типов в строки и обратно
    m "math"   // Импортировать math под локальным именем m.
)

// Объявление функции. Main – это специальная функция, служащая точкой входа для исполняемой программы.
func main() {
    // Println выводит строку в stdout.
    // Данная функция находится в пакете fmt.
    fmt.Println("Hello world!")
    // Вызов другой функции из текущего пакета.
    beyondHello()
}
// Функции содержат входные параметры в круглых скобках. Пустые скобки все равно обязательны, даже если параметров нет.
func beyondHello() {
    var x int // Переменные должны быть объявлены до их использования.
    x = 3     // Присвоение значения переменной.
    // Краткое определение := позволяет объявить переменную с автоматической
    // подстановкой типа из значения.
    y := 4
    sum, prod := learnMultiple(x, y)        // Функция возвращает два значения.
    fmt.Println("sum:", sum, "prod:", prod) // Простой вывод.
    learnTypes()  // < y minutes, learn more!
}
// Функция, имеющая входные параметры и возвращающая несколько значений.
func learnMultiple(x, y int) (sum, prod int) {
    return x + y, x * y //Возврат двух значений.
}
// Некоторые встроенные типы и литералы.
func learnTypes() {
    // Краткое определение переменной
    s := "Learn Go!" // Тип string.
    s2 := `"Чистый" строковой литерал может содержать переносы строк` // Тоже тип данных string
    // Символ не из ASCII. Исходный код Go в кодировке UTF-8.
    g := 'Σ' // тип rune, это алиас для типа int32, содержит символ юникода.
    f := 3.14195 // float64, 64-х битное число с плавающей точкой (IEEE-754).
    c := 3 + 4i  // complex128, внутри себя содержит два float64.
    // Синтаксис var с инициализациями.
    var u uint = 7 // Беззнаковое, но размер зависит от реализации, как и у int.
    var pi float32 = 22. / 7
    // Синтаксис приведения типа с кратким определением
    n := byte('\n') // byte – это алиас для uint8.
    // Массивы имеют фиксированный размер на момент компиляции.
    var a4 [4]int           // массив из 4-х int, инициализирован нулями.
    a3 := [...]int{3, 1, 5} // массив из 3-х int, ручная инициализация.
    // Слайсы (slices) имеют динамическую длину. И массивы, и слайсы имеют свои
    // преимущества, но слайсы используются гораздо чаще.
    s3 := []int{4, 5, 9}    // Сравните с a3, тут нет троеточия.
    s4 := make([]int, 4)    // Выделение памяти для слайса из 4-х int (нули).
    var d2 [][]float64      // Только объявление, память не выделяется.
    bs := []byte("a slice") // Синтаксис приведения типов.
    p, q := learnMemory() // Объявление p и q как указателей на int.
    fmt.Println(*p, *q)   // * извлекает
```

```
указатель. Печатает два int-а.


  // Map, также как и словарь или хеш из некоторых других языков, является
  // ассоциативным массивом с динамически изменяемым размером.
  m := map[string]int{"three": 3, "four": 4}
  m["one"] = 1
  delete(m, "three") // Встроенная функция, удаляет элемент из map-а.
  // Неиспользуемые переменные в Go являются ошибкой. Нижнее подчёркивание позволяет игнорировать такие переменные.
  _, _, _, _, _, _, _, _ = s2, g, f, u, pi, n, a3, s4, bs // Вывод считается использованием переменной.
  fmt.Println(s, c, a4, s3, d2, m)
  learnFlowControl() // Идем дальше.
}
// У Go есть полноценный сборщик мусора. В нем есть указатели, но нет арифметики
// указателей. Вы можете допустить ошибку с указателем на nil, но не с инкрементацией указателя.
func learnMemory() (p, q *int) {
  // Именованные возвращаемые значения p и q являются указателями на int.
  p = new(int) // Встроенная функция new выделяет память. Выделенный int проинициализирован нулём, p больше не содержит nil.
  s := make([]int, 20) // Выделение единого блока памяти под 20 int-ов.
  s[3] = 7// Присвоить значение.
  r := -2 // Определить ещё одну локальную переменную.
  return &s[3], &r     // Амперсанд(&) обозначает получение адреса переменной.
}
func expensiveComputation() float64 {
  return m.Exp(10)
}
func learnFlowControl() {
  // If-ы всегда требуют наличие фигурных скобок, но не круглых.
  if true {
    fmt.Println("told ya")
  }
  // Форматирование кода стандартизировано утилитой "go fmt".
  if false {
    // Будущего нет.
  } else {
    // Жизнь прекрасна.
  }
  // Используйте switch вместо нескольких if-else.
  x := 42.0
  switch x {
  case 0:
  case 1:
  case 42:
    // Case-ы в Go не "проваливаются" (неявный break).
  case 43:
    // Не выполнится.
  }
  // For, как и if не требует круглых скобок
  // Переменные, объявленные в for и if являются локальными.
  for x := 0; x < 3; x++ { // ++ – это операция.
    fmt.Println("итерация", x)
  }
  // Здесь x == 42.
  // For – это единственный цикл в Go, но у него есть альтернативные формы.
  for { // Бесконечный цикл.
    break    // Не такой уж и бесконечный.
    continue // Не выполнится.
  }
  // Как и в for, := в if-е означает объявление и присвоение значения y,
  // проверка y > x происходит после.
  if y := expensiveComputation(); y > x {
    x = y
  }
  // Функции являются замыканиями.
  xBig := func() bool {
    return x > 10000 // Ссылается на x, объявленный выше switch.
  }
  fmt.Println("xBig:", xBig()) // true (т.к. мы присвоили x = e^10).
  x = 1.3e3                    // Тут x == 1300
  fmt.Println("xBig:", xBig()) // Теперь false.
  // Метки, куда же без них, их все любят.
  goto love
love:
  learnDefer()      // Быстрый обзор важного ключевого слова.
  learnInterfaces() // О! Интерфейсы, идём
```

```
далее.
}


func learnDefer() (ok bool) {
    // Отложенные(deferred) выражения выполняются сразу перед тем, как функция
    // возвратит значение.
    defer fmt.Println("deferred statements execute in reverse (LIFO) order.")
    defer fmt.Println("\nThis line is being printed first because")
    // defer широко используется для закрытия файлов, чтобы закрывающая файл
    // функция находилась близко к открывающей.
    return true
}
// Объявление Stringer как интерфейса с одним методом, String.
type Stringer interface {
    String() string
}
// Объявление pair как структуры с двумя полями x и y типа int.
type pair struct {
    x, y int
}
// Объявление метода для типа pair. Теперь pair реализует интерфейс Stringer.
func (p pair) String() string { // p в данном случае называют receiver-ом.
    // Sprintf – ещё одна функция из пакета fmt.
    // Обращение к полям p через точку.
    return fmt.Sprintf("(%d, %d)", p.x, p.y)
}
func learnInterfaces() {
    // Синтаксис с фигурными скобками это "литерал структуры". Он возвращает
    // проинициализированную структуру, а оператор := присваивает её p.
    p := pair{3, 4}
    fmt.Println(p.String()) // Вызов метода String у переменной p типа pair.
    var i Stringer          // Объявление i как типа с интерфейсом Stringer.
    i = p                   // Валидно, т.к. pair реализует Stringer.
    // Вызов метода String у i типа Stringer. Вывод такой же, что и выше.
    fmt.Println(i.String())
    // Функции в пакете fmt сами всегда вызывают метод String у объектов для
    // получения строкового представления о них.
    fmt.Println(p) // Вывод такой же, что и выше. Println вызывает метод String.
    fmt.Println(i) // Вывод такой же, что и выше.
    learnVariadicParams("Учиться", "учиться", "и ещё раз учиться!")
}
// Функции могут иметь варьируемое количество параметров.
func learnVariadicParams(myStrings ...interface{}) {
    // Вывести все параметры с помощью итерации.
    for _, param := range myStrings {
        fmt.Println("param:", param)
    }
    // Передать все варьируемые параметры.
    fmt.Println("params:", fmt.Sprintln(myStrings...))
    learnErrorHandling()
}
func learnErrorHandling() {
    // Идиома ", ok" служит для обозначения корректного срабатывания чего-либо.
    m := map[int]string{3: "three", 4: "four"}
    if x, ok := m[1]; !ok { // ok будет false, потому что 1 нет в map-е.
        fmt.Println("тут никого нет")
    } else {
        fmt.Print(x) // x содержал бы значение, если бы 1 был в map-е.
    }
    // Идиома ", err" служит для обозначения была ли ошибка или нет.
    if _, err := strconv.Atoi("non-int"); err != nil { // _ игнорирует значение
        // выведет "strconv.ParseInt: parsing "non-int": invalid syntax"
        fmt.Println(err)
    }
```

```
    // Мы ещё обратимся к интерфейсам чуть позже, а пока...
    learnConcurrency()
}

// c – это тип данных channel (канал), объект для конкурентного взаимодействия.
func inc(i int, c chan int) {
    c <- i + 1 // когда channel слева, <- являтся оператором "отправки".
}

// Будем использовать функцию inc для конкурентной инкрементации чисел.
func learnConcurrency() {
    // Тот же make, что и в случае со slice. Он предназначен для выделения
    // памяти и инициализации типов slice, map и channel.
    c := make(chan int)
    // Старт трех конкурентных goroutine. Числа будут инкрементированы
    // конкурентно и, может быть параллельно, если машина правильно
    // сконфигурирована и позволяет это делать. Все они будут отправлены в один
    // и тот же канал.
    go inc(0, c) // go начинает новую горутину.
    go inc(10, c)
    go inc(-805, c)
    // Считывание всех трех результатов из канала и вывод на экран.
    // Нет никакой гарантии в каком порядке они будут выведены.
    fmt.Println(<-c, <-c, <-c) // канал справа, <- обозначает "получение".

    cs := make(chan string)       // другой канал, содержит строки.
    cc := make(chan chan string)  // канал каналов со строками.
    go func() { c <- 84 }()       // пуск новой горутины для отправки значения
    go func() { cs <- "wordy" }() // ещё раз, теперь для cs
    // Select тоже что и switch, но работает с каналами. Он случайно выбирает
    // готовый для взаимодействия канал.
    select {
    case i := <-c: // полученное значение можно присвоить переменной
        fmt.Printf("это %T", i)
    case <-cs: // либо значение можно игнорировать
        fmt.Println("это строка")
    case <-cc: // пустой канал, не готов для коммуникации.
        fmt.Println("это не выполнится.")
    }
    // В этой точке значение будет получено из c или cs. Одна горутина будет
    // завершена, другая останется заблокированной.

    learnWebProgramming() // Да, Go это может.
}

// Всего одна функция из пакета http запускает web-сервер.
func learnWebProgramming() {
    // У ListenAndServe первый параметр это TCP адрес, который нужно слушать.
    // Второй параметр это интерфейс типа http.Handler.
    err := http.ListenAndServe(":8080", pair{})
    fmt.Println(err) // не игнорируйте сообщения об ошибках
}

// Реализация интерфейса http.Handler для pair, только один метод ServeHTTP.
func (p pair) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    // Обработка запроса и отправка данных методом из http.ResponseWriter
    w.Write([]byte("You learned Go in Y minutes!"))
}

func requestServer() {
    resp, err := http.Get("http://localhost:8080")
    fmt.Println(err)
    defer resp.Body.Close()
    body, err := ioutil.ReadAll(resp.Body)
    fmt.Printf("\nWebserver said: `%s`", string(body))
}
```