Максим Жашкевич

Язык Go
Для Начинающих

От базовых концепций до построения REST API

Об авторе

Привет, меня зовут Максим. Сейчас ты читаешь мою книгу, которая поможет тебе начать разрабатывать современные приложения на языке программирования Go.

С этим языком я активно работаю последних 2.5 года. За это время я успел применить его на практике в проектах различной сложности, как личных так и на работе и фрилансе. Чем больше я разрабатываю на этом языке, тем больше я влюбляюсь в его простоту, производительность и мощность.

Язык я начал изучать в полевых условиях, так как мне нужно было буквально за 1-2 дня сделать тестовое задание на Go при приеме на новую работу. Тогда было совсем мало хорошего материала по этому языку, особенно в рунете.

Недавно я решил заново повторить основы и заполнить пробелы в фундаментальных знаниях Go. Тут мне пришла идея структурировать все знания и опыт для ребят, которые только начинают свой путь с этим языком.

Надеюсь, эта книга будет тебе полезной и послужит хорошим стартом в мир back-end разработки на этом замечательном языке.

Написать мне: *@zhashkevych (Telegram) / zhashkevychmaksim@gmail.com*
Также я веду блог в телеграмм канале: *https://t.me/zhashkevychdev*
По настроению пишу статьи на медиум: *https://medium.com/@zhashkevych*
И выкладываю разные проекты на GitHub: *https://github.com/zhashkevych*

Эту книгу я написал чтобы помочь другим в освоении нового языка и поделиться своими знаниями. Если вам эта книга поможет и вы захотите меня отблагодарить, можете скинуть мне деньги на кофе: 5375 4141 0101 1855, мне будет очень приятно!

Спасибо что вы тут со мной, читаете эту книгу! Поехали!

Как читать эту книгу?

Книга состоит из 12 разделов. В разделах 1-10 идет разбор концепций языка с практическими примерами.

В каждом разделе в конце есть домашнее задание. Для закрепления материала и лучшего понимания последующих разделов я рекомендую вам их выполнять.

Также после каждого раздела предлагаю вам самим пописать код и на практике закрепить полученные знания.

Вы всегда можете обращаться в эту книгу как в справочник, чтобы освежать знания по тем или иным темам.

Конечно вы можете перепрыгивать разделы. Однако весь материал подан именно в такой очереди специально, чтобы вы смогли понять более сложный материал имея понимание базовых концепций.

Исходный код примеров вы можете найти тут:
https://github.com/zhashkevych/go-basics

Оглавление

Раздел 00:

Почему Go?

На дворе начало третьего десятилетия XXI века. Огромная часть населения земли имеет доступ к скоростному интернету. Такими сервисами как YouTube, Facebook, Instagram, Netflix и тд. ежедневно пользуется сотни миллионов людей по всему миру.

Вся разработка уходит в веб и эта тенденция будет только развиваться.

Существует огромное количество языков программирования на которых разрабатываются современные приложения. И на данный момент язык Go уже хорошо зарекомендовал себя на практике и становится стандартом индустрии.

Все больше вакансий для Go разработчиков появляется на рынке. Все больше проектов и стартапов выбирают Go в качестве основного языка для бекенда. Появляется все больше Open-Source репозиториев в которых используется Go.

Если вы только начинаете свой путь в программировании или вы уже имеете опыт с другими технологиями и хотите изучить новый язык, то Go сейчас – отличный выбор.

Немного истории

Go был разработан в 2007 году в стенах компании Google при участии Роба Пайка и Кена Томпсона.

Изначально язык разрабатывался для внутреннего использования в гугле с целью решения широкого спектра задач в эпоху распределенных систем и многоядерных процессоров, однако в 2009 году состоялся релиз и язык стал доступен для широкой публики.

Сам Роб Пайк говорит, что:

> “Go был разработан для решения реальных проблем, возникающих при разработке программного обеспечения в Google”

За чуть более чем 10 лет своего существования, Go успел завоевать любовь множества разработчиков по всему миру. Так чем же так хорош этот язык на сегодняшний день?

Простой и понятный синтаксис

У языка лаконичный и простой синтаксис, основанный на C, но существенно доработанный, с большим количеством синтаксического сахара.

Читать код, написаный на Go, достаточно приятно, что является одним из существенных плюсов данного языка.

Статическая типизация

Позволяет избежать ошибок, допущенных по невнимательности разработчика, упрощает чтение и понимание кода, делает код однозначным.

Быстродействие и производительность

Go разрабатывался с учетом широкого распространения многоядерных процессоров. Из коробки поддерживает конкурентность и параллелизм за счет встроенных потоков (горутин).

Также, Go – компилируемый язык, что делает Go достаточно быстрым языком. Это значит что программы транслируются сразу в машинный код. Это убирает прослойку виртуальной машины, как в случае с интерпретируемыми языками (Python, JavaScript, Ruby и т.д), или семейством JVM (Java, Kotlin, Scala).

Достаточно большое комьюнити, которое стремительно развивается

В сети уже существует огромное множество видео, статьей и книг по данному языку, что облегчает процесс обучения и поиска информации.

Также на GitHub можно найти большое количество Open-Source проектов, библиотек и фреймворков, что дает возможность использовать готовые, качественные инструменты в своем коде, а также изучать кодовые базы на других проектах.

Появляется все больше вакансий на рынке

Как я уже говорил, все больше современных стартапов и проектов выбирают Go, поэтому хорошо изучив данный язык вы сможете работать над интересными продуктами в хороших компаниях и достаточно хорошо зарабатывать.

Идеально подходит для разработки современных веб-сервисов и высоконагруженных распределенных систем.

Такие гиганты как Facebook, Google, Netflix, Uber, Twitch, Medium и тд. используют язык для своих разработки своих продуктов и решения внутренних проблем, о чем кстати часто делятся в своих блогах для девелоперов.

Уверен, все сказанное выше заинтересовало вас освоить этот язык, а в этом вам поможет данная книга.

Она подойдет вам, если:

- У вас есть желание освоить новую технологию и back-end разработку в целом.
- Есть базовое понимание основ программирования
- Уже имеется опыт в других языках программирования (необязательно, но будет большим плюсом)
- Вы полны энтузиазма и готовы самостоятельно закреплять пройденный материал на практике

Эта книга – не исчерпывающее пособие по Go. Скорее это разбор всех основных концепций языка, структурированное пособие для быстрого и уверенного старта разработки.

Чтобы более детально и глубоко изучить данную технологию рекомендую:

- Язык программирования Go (Алан А. А. Донован, Брайан У. Керниган)
- Go на практике (Батчер, Фарина)
- Практика, практика и еще раз практика :)

Я сторонник изучения новых навыков на практике. Поэтому на протяжении всей книги мы будем закреплять все знания на практике, а в конце разработаем небольшое веб приложение.

Раздел 01:

Базовые Концепции. Переменные.

Надеюсь, у вас уже установлены инструменты Go на вашей машине, а если нет, то следуйте этим инструкциям. Также нам понадобится текстовый редактор, я рекомендую VS Code, но вы можете использовать любую другую удобную для вас среду.

На момент написания статьи используется Go версии 1.14.

Структура приложения

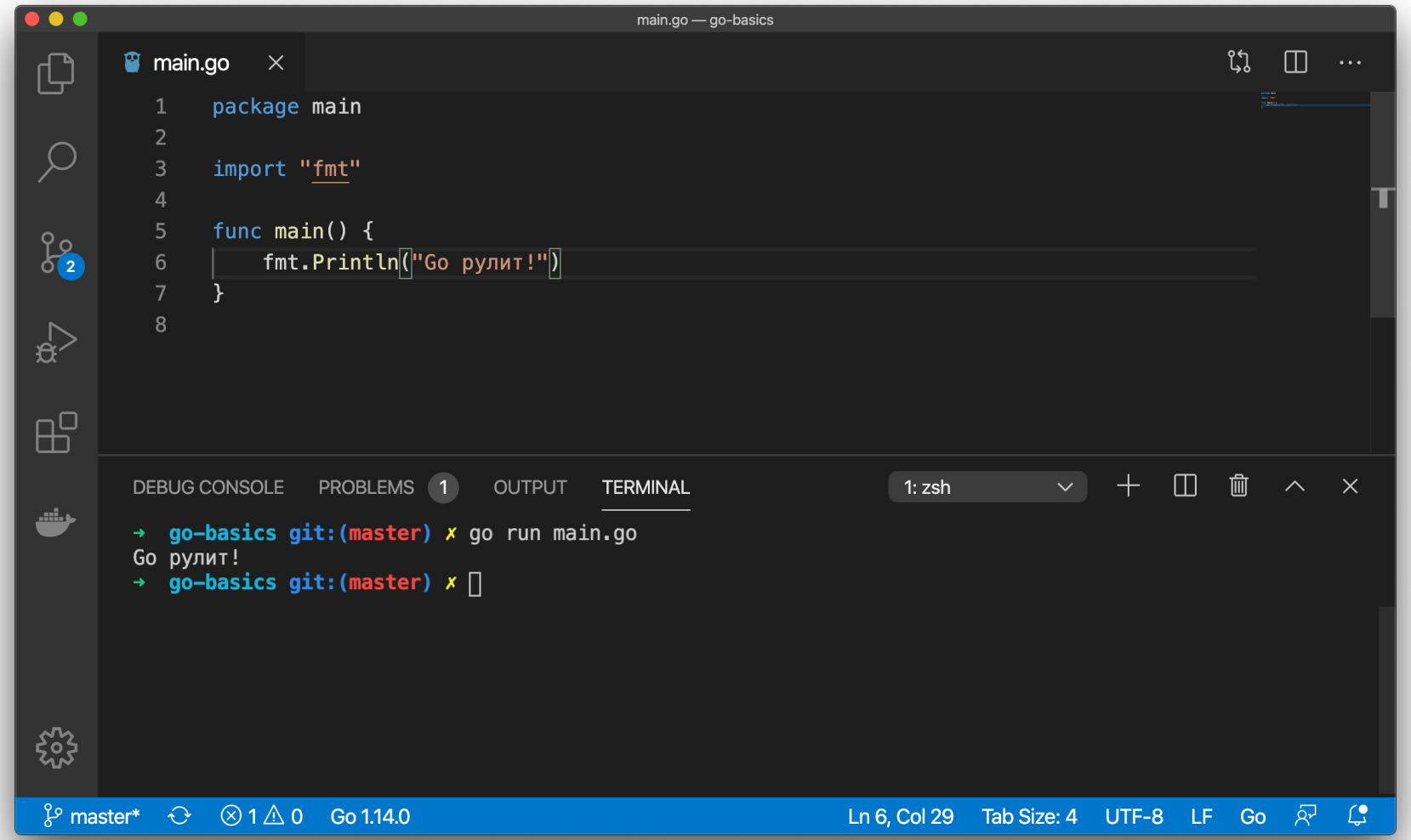
Давайте сразу рассмотрим самую простую программу. Все что она делает, это выводит в консоль строку “Go рулит!”.

Чтобы запустить этот код на своей машине, создайте файл main.go и перепишите в него код ниже. Далее для запуска воспользуйтесь командой **go run main.go** из консоли, в той же директории где находится ваш **main.go** файл.

На примере ниже вы можете увидеть 3 основных элемента:

- **package main –** Объявление название пакета
- **import “fmt” –** Импорт пакета fmt из стандартной библиотеки
- **func main() –** Объявление функции main() которая является точкой входа для любой программы

Программы на Go хранятся в одном или нескольких файлах, имена которых оканчиваются на .go. Каждый файл начинается с объявления `package`, который говорит, частью какого пакета является данный файл.



Вы можете писать простые приложения лишь в одном пакете *main* и этого будет достаточно. Однако в больших приложениях разбиение на разные пакеты позволяет структурировать файлы и разделять их по зонам ответственности, что значительно упрощает написание и чтение исходного кода.

Следом за объявлением пакета следует строка с импортированием пакетов `import`. В данном конкретном примере мы импортируем пакет `"fmt"` из стандартной библиотеки Go.

С помощью `import` мы также можем импортировать в наш проект сторонние библиотеки от других разработчиков, предварительно их установив, или же другие пакеты из этого же проекта.

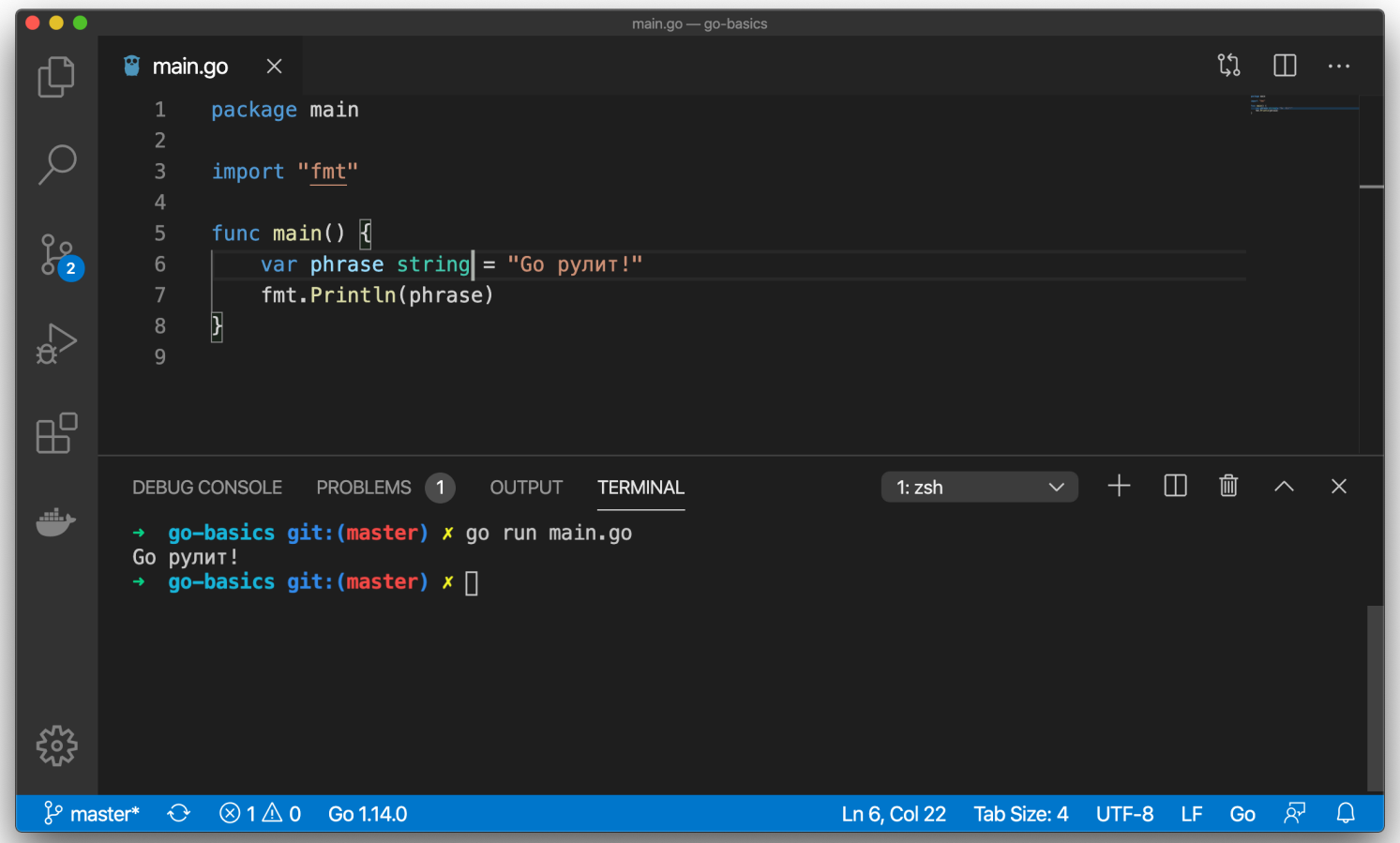
Если сейчас у вас возникают трудности с пониманием этих концепций – не переживайте, дальше мы еще более детально все это рассмотрим.

Следом за `package` и `import` идет последовательность объявления переменных, типов, констант и функций *уровня пакета* в произвольном порядке.

О переменных, константах, типах и функциях мы поговорим далее.

Переменные

Как мы уже говорили в прошлом разделе, Go – статически типизированный язык, по этому при объявлении переменной мы сразу задаем ее тип, который в последствии не может измениться.



Ключевое слово `var` создает переменную определенного типа, назначает ей имя и присваивает начальное значение. Каждое объявление имеет общий вид.

```
var имяПеременной типПеременной = выражение
```

Любая из частей **тип_переменной** и **= выражение** может быть опущена, но не обе вместе. Если опущен тип, он определяется из инициализирующего выражения.

```
var pharse = "Go рулит!" // на этапе компиляции, тип переменной определяется как string автоматически
```

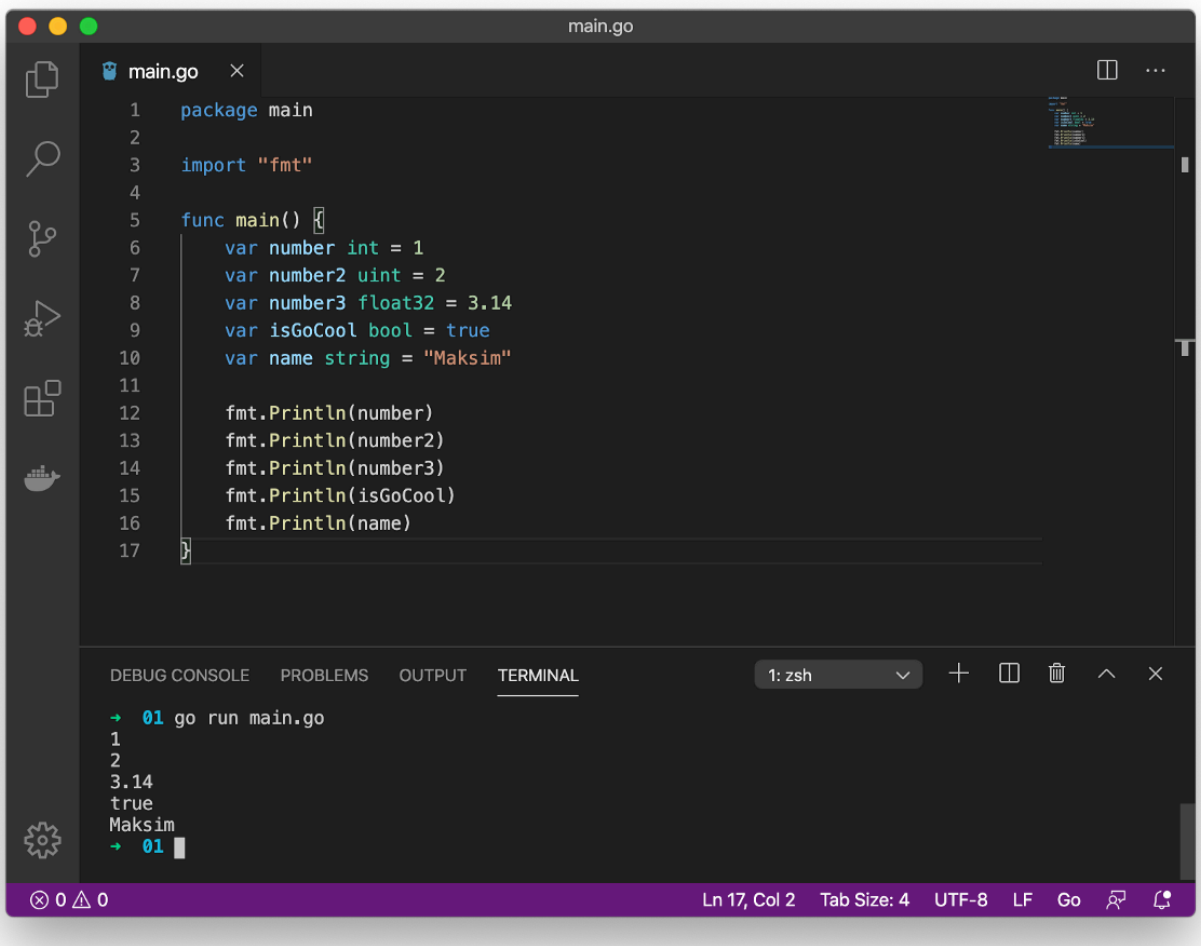
Если же опущено выражение, то начальным значением является нулевое значение для данного типа, равное 0 для чисел, false – для булевых переменных, “ “ – для строк и nil – для интерфейсов и ссылочных типов (срезов, указателей, отображений, каналов, функций).

Не пугайтесь, об этих вещах мы поговорим далее.

```
var phrase string
fmt.Println(phrase) // выведет пустую строку ""
```

В одном объявлении можно объявлять и (необязательно) инициализировать несколько переменных, используя соответствующий список выражений. Пропущенный тип позволяет объявлять несколько переменных разных типов.

```
var i, j, k int // int, int, int
var b, f, s = true, 2.3, "four” // bool, float64, string
```

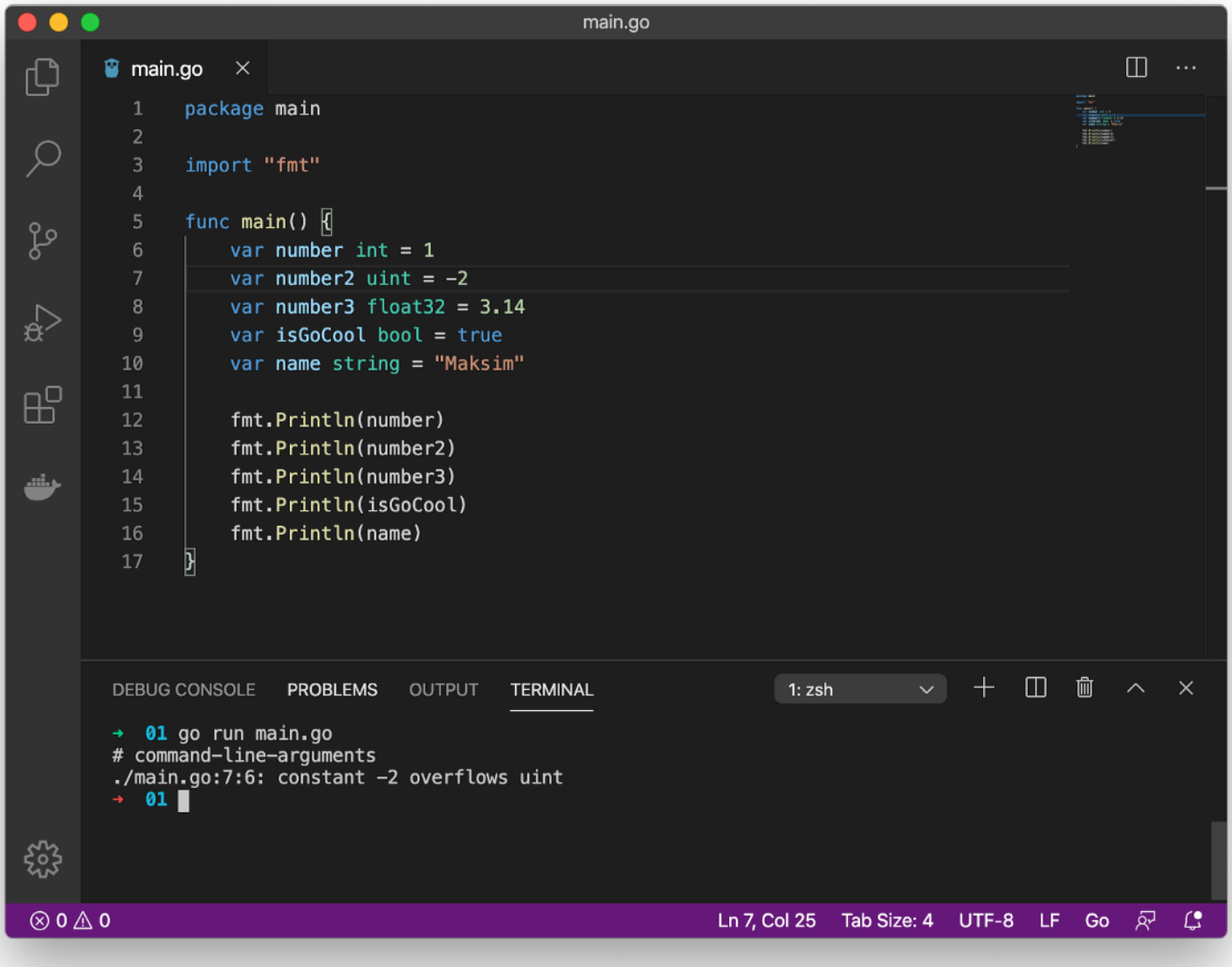


Основные фундаментальные типы данных в Go это:

- **Целочисленные типы** (int, int8, intl6, int32, int64, uint, uint8, uintl6, uint32, uint64, uintptr, byte)
- Числа с плавающей точкой (float32, float64)
- **Комплексные числа** (complex64, complex128)
- Тип **bool** (имеет лишь два возможных значения: true или false)
- Строки **string** (в примере выше, переменная phrase – строка, которая хранит значение “Go рулит!”

Тип int – целочисленное число, приставка в конце (8, 16, 32, 64) определяет разрядность числа, соотвественно максимально допустимое число, которое можно поместить в переменную данного типа.

Тип uint – тоже целочисленное число, которое не может быть отрицательным (меньше 0). Если мы поменяем код выше и изменим `number2 = -2`, при запуске программы получим ошибку.



Область видимости и Краткое объявление переменных

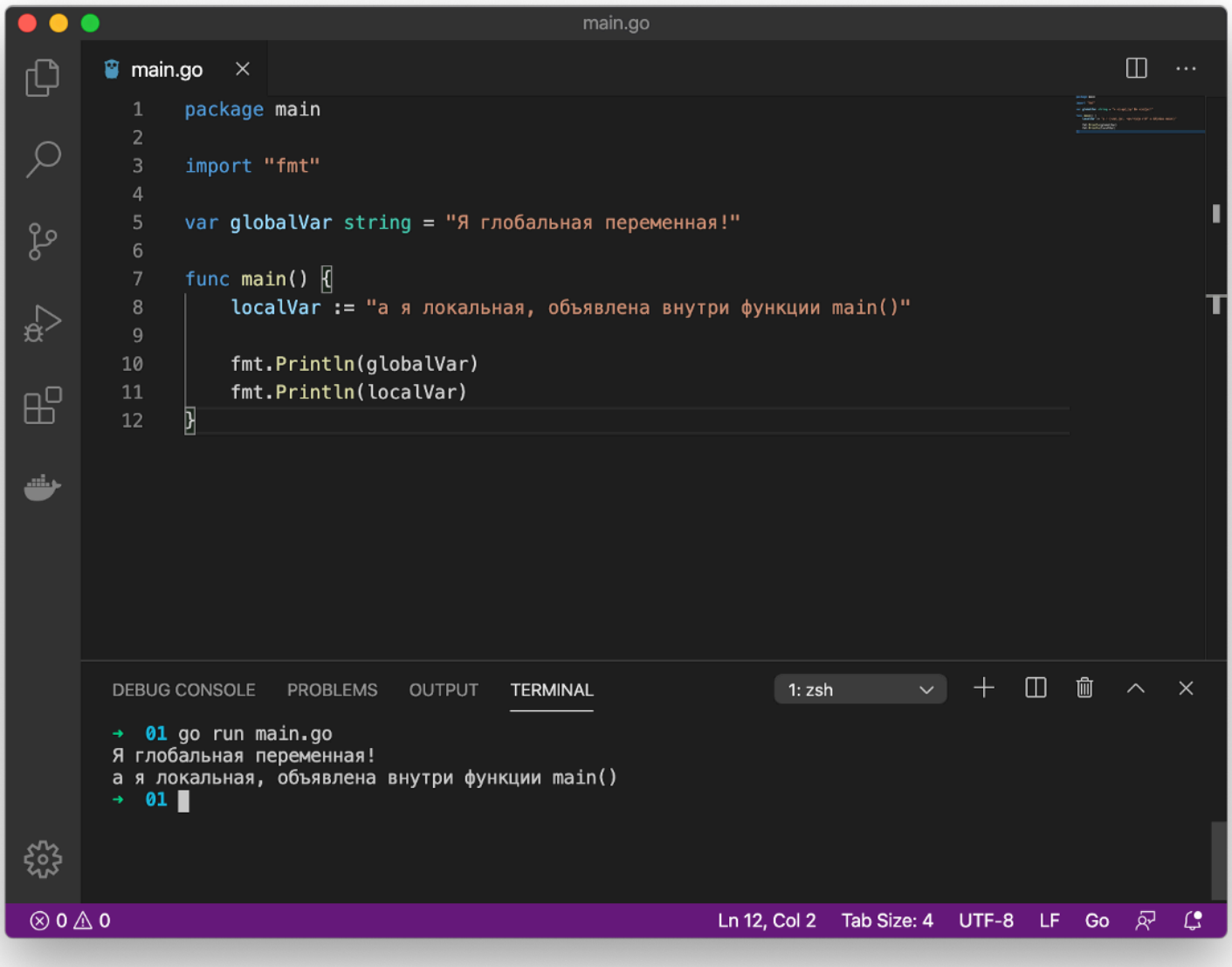
Существует два типа переменных:

- Глобальные
- Локальные

Локальные переменные объявляются внутри фигурных скобок `{}` (например в теле функции, конструкций `if` , `for`) и доступны лишь в рамках этих скобок.

Для локальных переменных доступен краткий формат инициализации

`имя_переменной := выражение`



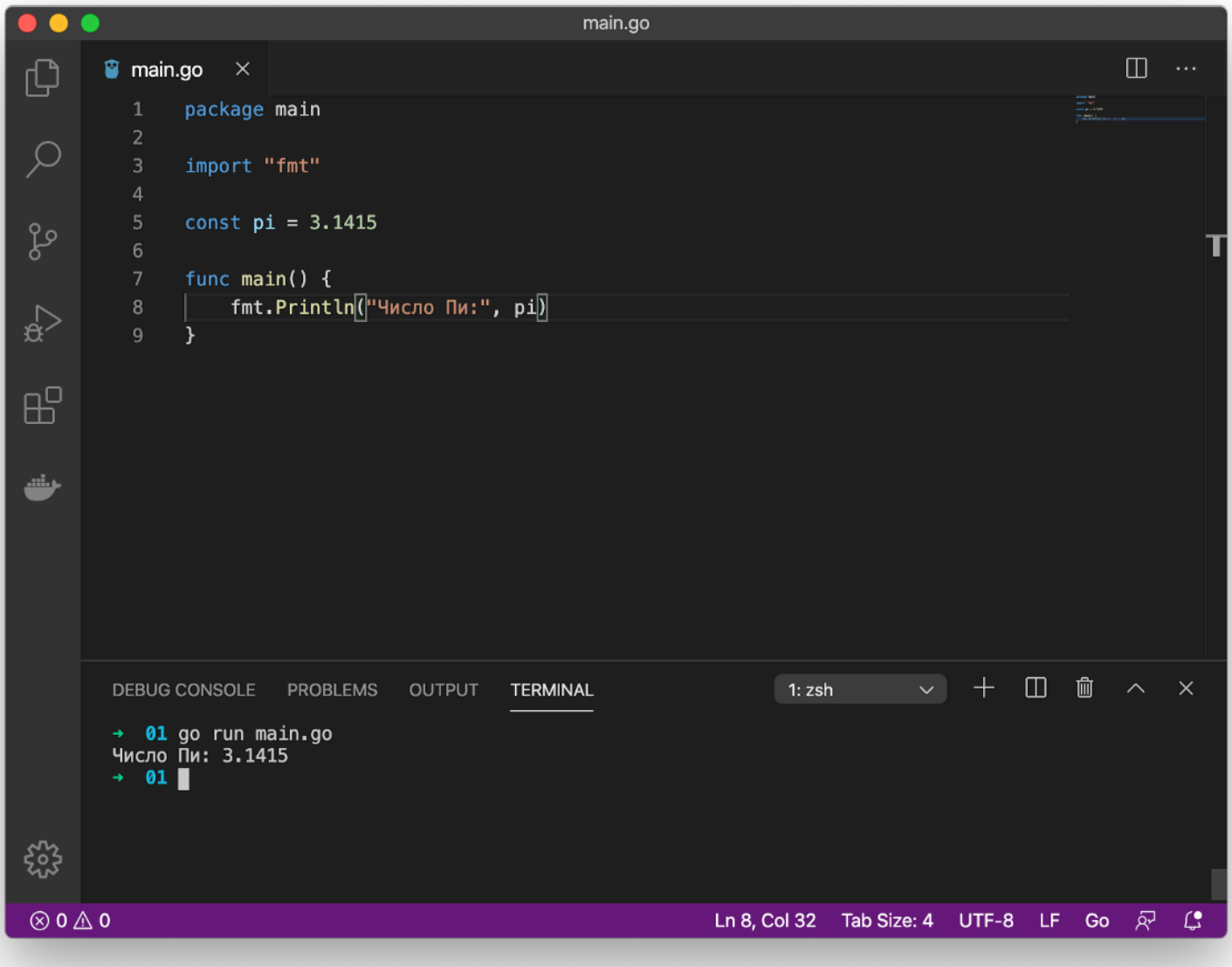
Глобальные переменные в свою очередь объявляются вне тела функций, и могут быть доступными в любом месте внутри пакета или вне его (об этом мы поговорим позже).

При кратком объявлении, тип переменной определяется типом выражения после := при инициализации.

Обратите внимание на символ := . Он означает инициализацию переменной. Если мы хотим присвоить новое значение в уже инициализированную переменную, нужно использовать обычный знак ровно =, иначе компилятор выдаст ошибку.

```
newVariable := 1
newVariable = 2 // хорошо
newVariable := 2 // плохо, программа не скомпилируется
```

Константы



Константы очень схожи с переменными, но их значение не может быть перезаписано после инициализации. Они используются для того, чтобы хранить в программе какие либо неизменные значения.

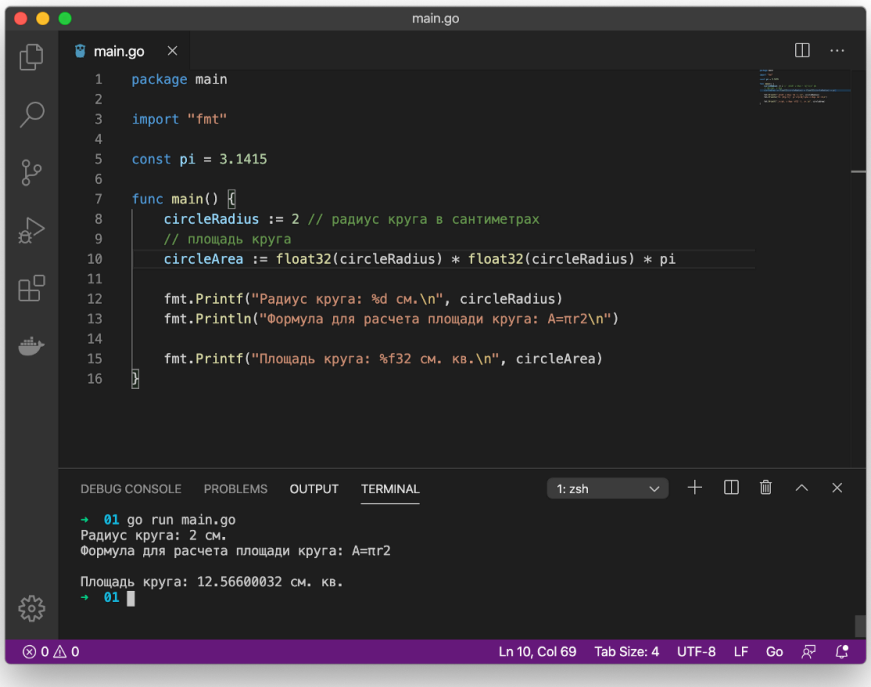
Например, число Пи – константа, т. к его значение всегда будет равно 3.1415… По этому если вы хотите иметь подобные значения в своей программе, стоит использовать именно константы для их хранения.

Инициализация констант имеет схожий синтаксис с инициализацией переменных, однако вместо ключевого слова `var` используется `const`.

Как вы можете увидеть, при компиляции программы происходит ошибка при попытке перезаписать значение в константу.

Константы также бывают локальными и глобальными, но зачастую они используются в качестве глобальной ячейки хранения данных.

Основные операции



К базовым операциям над переменными можно отнести сложение, вычитание, умножение и деление чисел.

Пока что при инициализации переменных и констант мы присваивали им фиксированные значения.

Однако нам никто не запрещает использовать математические операции над числами и другими переменными при инициализации новых констант и переменных.

Хочу обратить ваше внимание на `float32(circleRadius)` .

Дело в том, что, поскольку Go является строго типизированным ЯП, всевозможные математические операции возможны только над переменными одного и того же типа.

При инициализации `circleRadius` , посмотрев на значение 2, компилятор определил тип этой переменной как `int`. Но константа `pi` является числом с плавающей точкой, поэтому ее компилятор определил как `float32`.

Поскольку у этих двух переменных разные типы, мы не можем производить над ними базовые операции. В таких случаях используется **приведение типов.**

Мы приводим `int` к `float32` с помощью данного синтаксиса. Также можно делать противоположное приведение (`float32` к `int`). Тогда значение округлится в меньшую сторону.

Также в данном примере используются более продвинутые техники форматирования и функцию `fmt.Printf()` .

Итак, давайте резюмируем.

Все программы на Go пишутся в файлах с именем .go. Каждый файл начинается с объявления имени пакета, к которому он принадлежит, а следом за этим следует импортирование сторонних пакетов.

Каждая программа должна иметь функцию `main()` которая является точкой входа и с которой начинается исполнение кода.

Существует несколько вариантов создания новых переменных, для локальных переменных доступна краткая форма записи.

Мы используем константы для хранения неизменных значений.

Надеюсь вы усвоили данный материал, предлагаю самим поиграться с переменными, типами, и выводом их в консоль с помощью пакета `fmt`.

Домашнее задание

Пакет `fmt` является частью стандартной библиотеки Go, и очень часто используется для форматирования строк и вывода текста в консоль.

Go славится своей документацией стандартной библиотеки, по этому в качестве домашнего задания предлагаю вам более детально разобраться с возможностями *пакета “fmt”* самостоятельно перед тем, как перейти к следующему разделу данной книги.

Раздел 02:

Функции и Указатели. Обработка Ошибок.

Функция – это именованная часть кода, набор инструкций, к которому можно обращаться из других участков программы.

Функция позволяет “завернуть” последовательность операторов в единое целое, которое может вызываться из других мест в программе, возможно, многократно.

Функции позволяют разбить большую работу на более мелкие фрагменты, которые могут быть написаны разными программистами, разделенными во времени и пространстве.

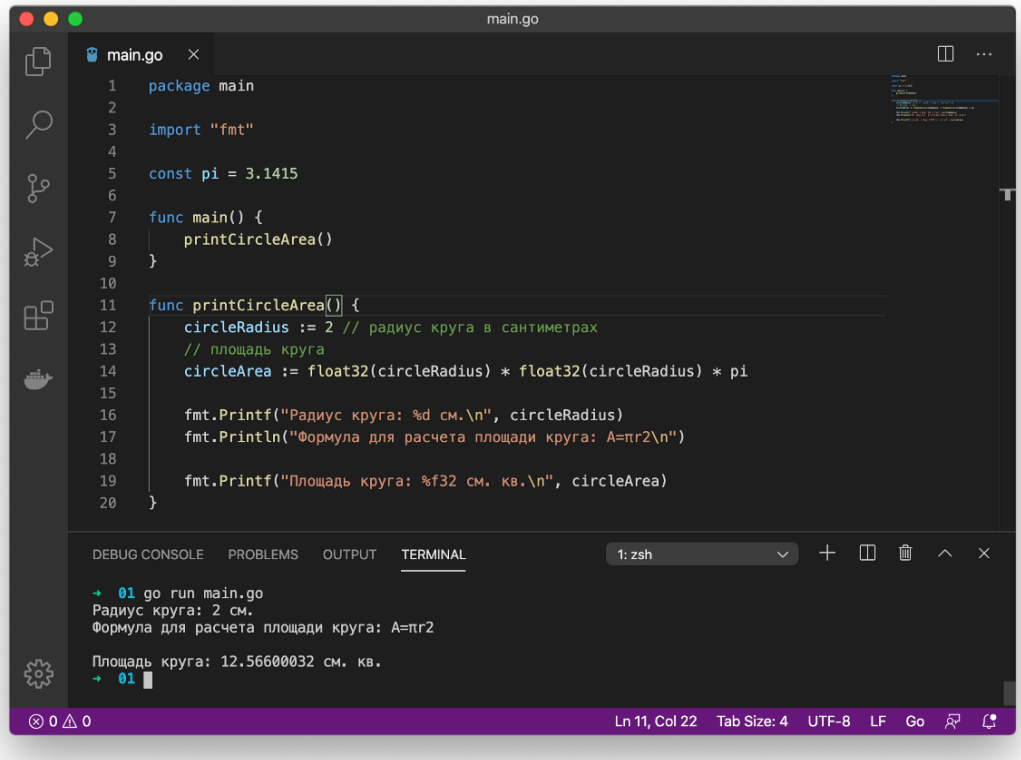
Функция скрывает подробности реализации от своих пользователей. По всем этим причинам функции являются важной частью любого языка программирования.

Синтаксис объявления функций имеет следующий синтаксис:

```
func имяФункции(аргумент1 тип, аргумент2 тип, ...)
возвращаемоеЗначение {}
```

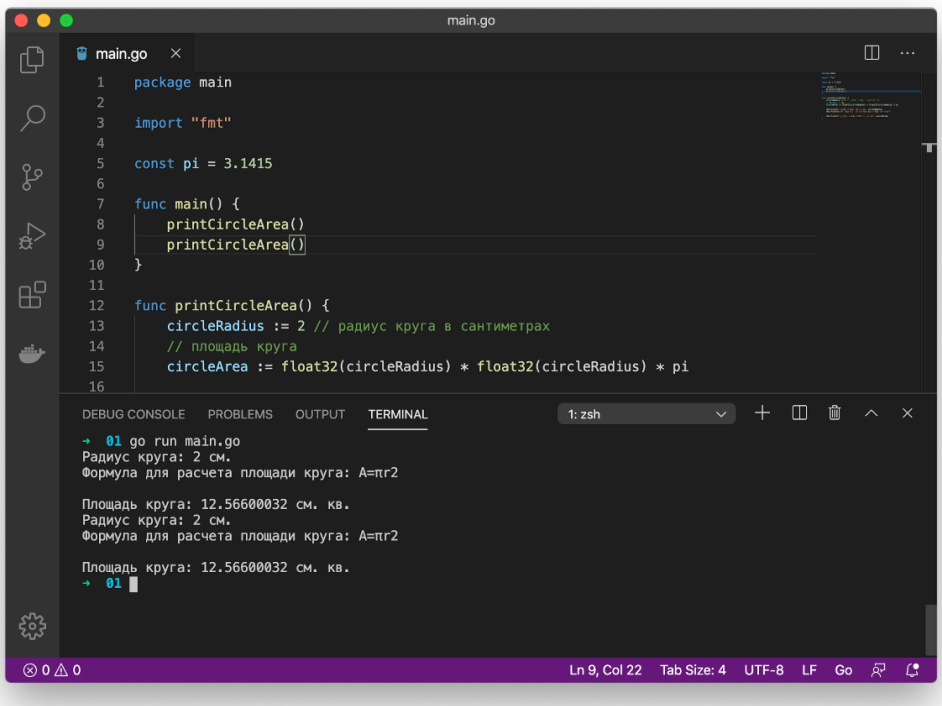
Функция может принимать значения в качестве аргументов, но это совсем не обязательно. Также функция может возвращать какие либо значения, что тоже не является обязательным условием.

Давайте перепишем пример с подсчетом площади круга с использованием функции.



В данном примере мы определили функцию `printCircleArea()`, которая делает все те же действия, что и раньше. Но теперь, мы имеем возможность вызывать эту функцию по несколько раз в разных местах, без необходимости дублировать уже написаный код.

За счет этого функции предоставляют гибкость в написании программ.



Вызвав данную функцию 2 раза, мы увидем что кусок кода, описаный в теле функции выполнился дважды.

Но эта функция далеко не идеальна. Ей не хватает гибкости. Мы уже с вами говорили что функция может иметь входящие параметры (аргументы) и возвращаемое значение.

Есть идеи как мы можем переделать эту функцию, чтобы считать площадь круга в зависимости от радиуса?

Мы можем принимать радиус в качестве аргумента, и использовать это значение для подсчетов. А при вызове функции уже передавать нужный радиус.

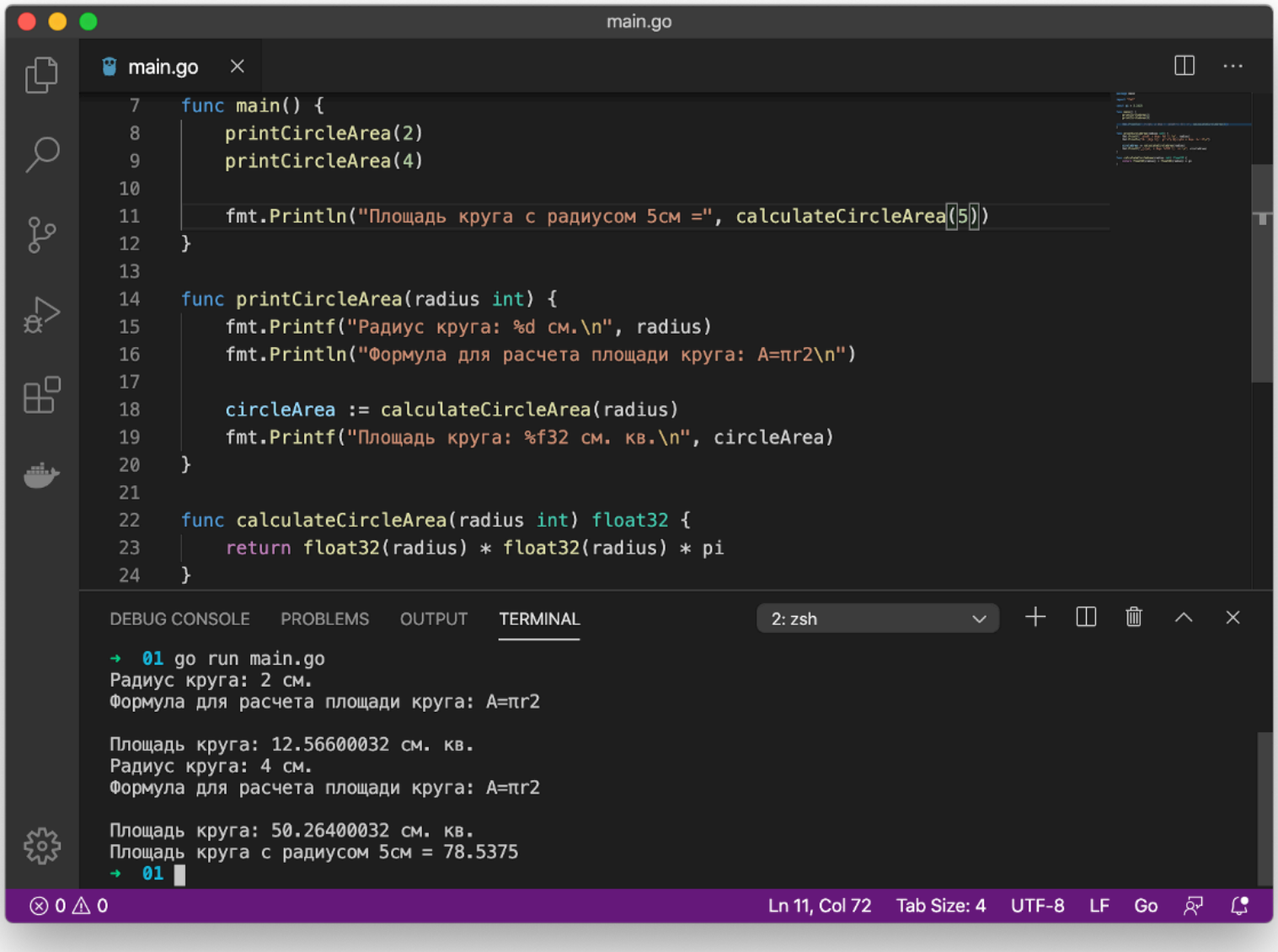
И все равно, тут еще есть пространство для оптимизаций.

При написании функций стоит держать в голове то, что каждая функция должна выполнять минимально возможную единицу работы.

В данном примере мы выполняем 2 задачи:

- Считаем площадь.
- Выводим на экран дополнительный текст.

Давайте разобьем этот функционал на более мелкие части.



Теперь мы с вами разбили код на самые мелкие участки функционала и можем использовать их независимо друг от друга в коде нашей программы.

Хочу обратить ваше внимание на ключевое слово `return`.

Мы определили возвращаемый тип функции как `float32`. А это значит что по завершению выполнения данной функции она обязательно должна вернуть значение типа `float32`.

Для того чтобы указать необходимое для возврата из функции значение необходимо использовать ключевое слово `return` и вслед за ним само значение.

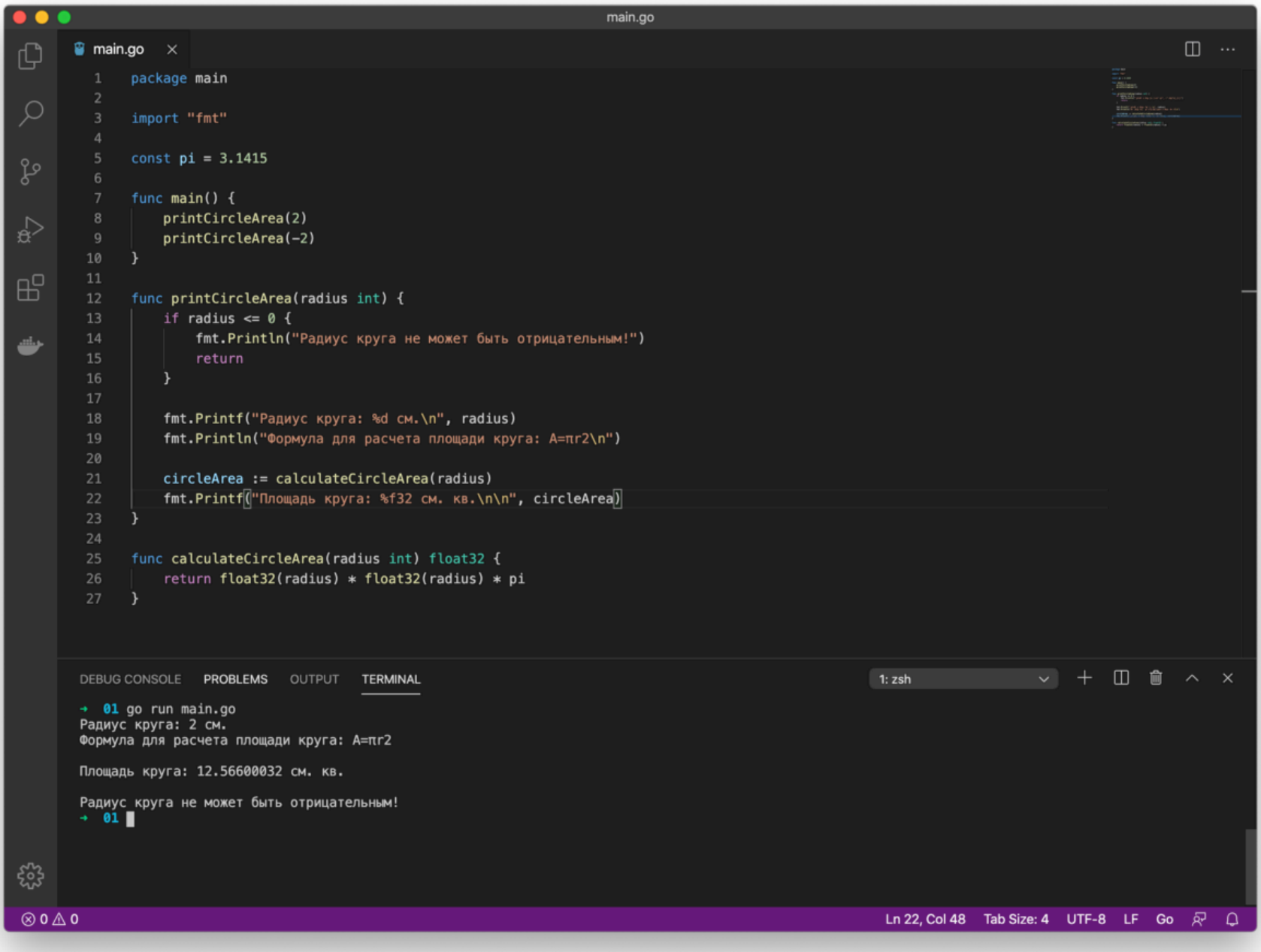
Если в функции нету возвращаемого значение, `return` используется без дополнительных аргументов и вызывается для досрочного прерывание исполнения функции.

Условный оператор if и обработка ошибок

Для разного рода проверок в Go используется условный оператор `if выражение {}`.

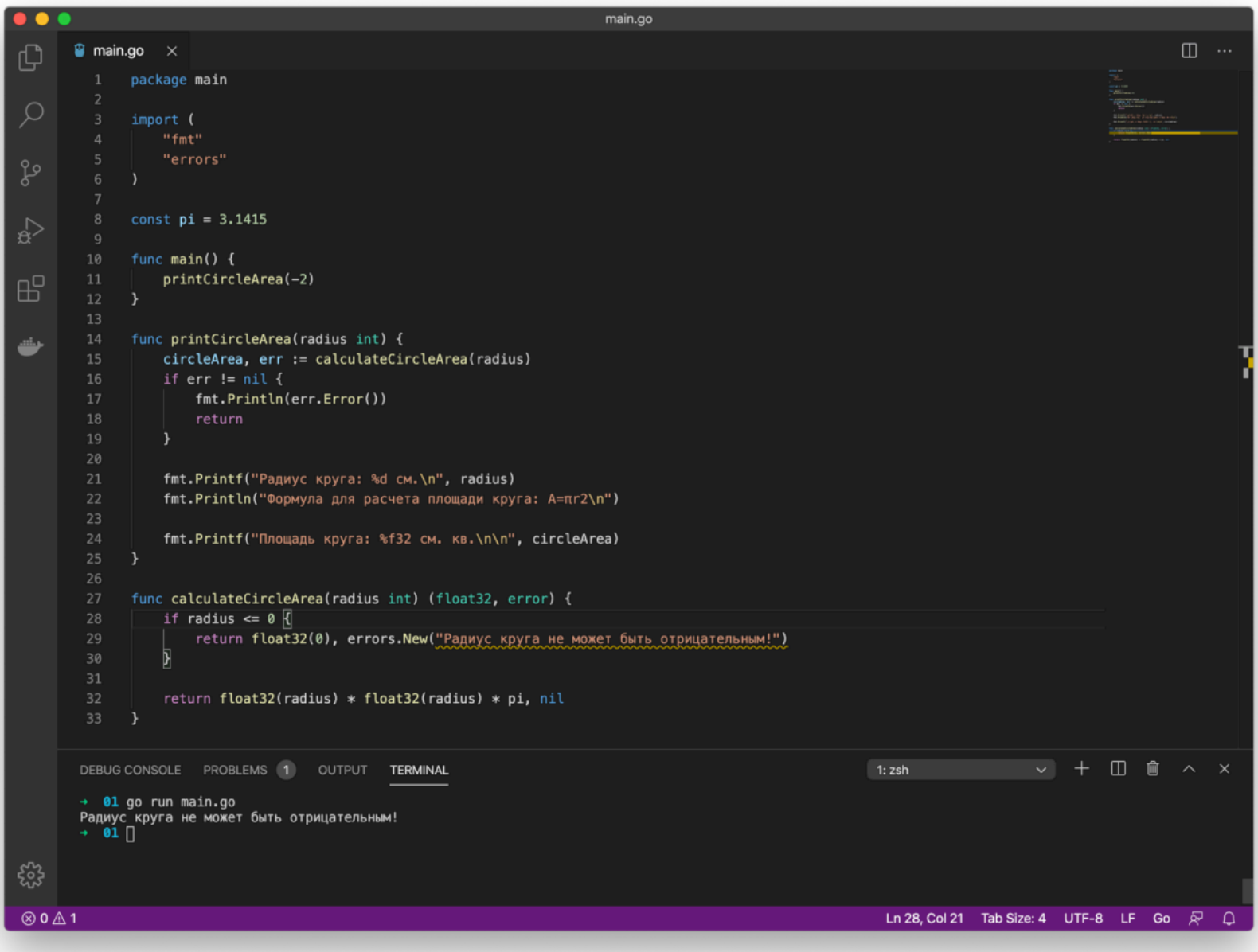
Вернемся к нашей программе. Сейчас она может считать площадь для всех значений радиуса, в том числе и для отрицательных.

Добавим проверку значения передаваемого в функцию радиуса на то, что он больше 0. Пустой `return` без дополнительных аргументов прерывает выполнение функции.



Такой вариант работает, но он далеко не идеален. Опять таки, функция, которая отвечает за вывод текста на экран не должна проверять радиус на валидность. За это должна отвечать функция подсчета площади, потому что именно в ней находиться логика работы с числами.

Давайте перепишем этот кусок кода, используя встроенную систему ошибок.



Что мы тут сделали:

- Вынесли проверку на корректность радиуса в `calculateCircleArea()`
- Добавили этой функции еще одно возвращаемое значение типа `error`, импортировали пакет `errors` и генерируем ошибку при помощи `errors.New()`

- Добавили возвращение `nil` в `return` после подсчета площади
- Перенесли присваивание площади с ошибкой в начало функции `printCircleArea()` и добавили проверку на `nil`
- Выводим текст ошибки с помощью `err.Error()`

В этом примере появилось сразу несколько новых для нас конструкций.

Во первых, язык Go поддерживает неограниченное количество возвращаемых значение в функций, типы которых должны быть в скобках и перечислены через запятую.

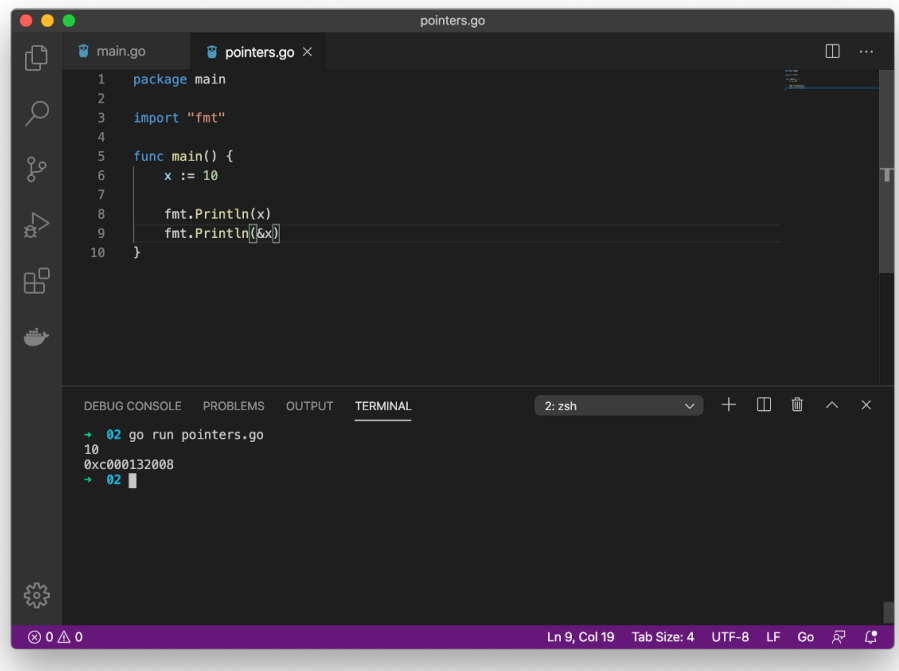
Во вторых, мы добавили новый тип `error` . Это встроенный тип, который является элегантным решением для обработки ошибок при вызове различных функций.

Значение ошибки может быть `nil` – это означает что ошибка пустая, т.е все хорошо и мы можем продолжать выполнение программы.

Проверка `if err != nil {}` выполняется, когда произошла ошибка и мы должны ее как-то обработать в нашей программе (залогировать ошибку и/или прервать выполнение текущей функции).

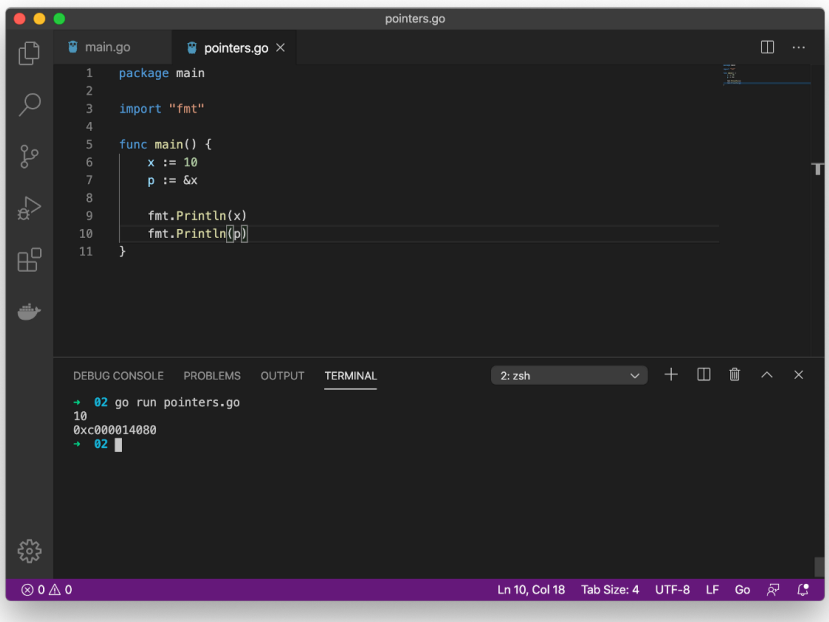
Указатели

Любая переменная в нашей программе храниться в выделенном блоке памяти компьютера. Каждый блок памяти имеет свой адрес, соответственно, у каждой переменной есть адрес в памяти, по которому хранится ее значение.



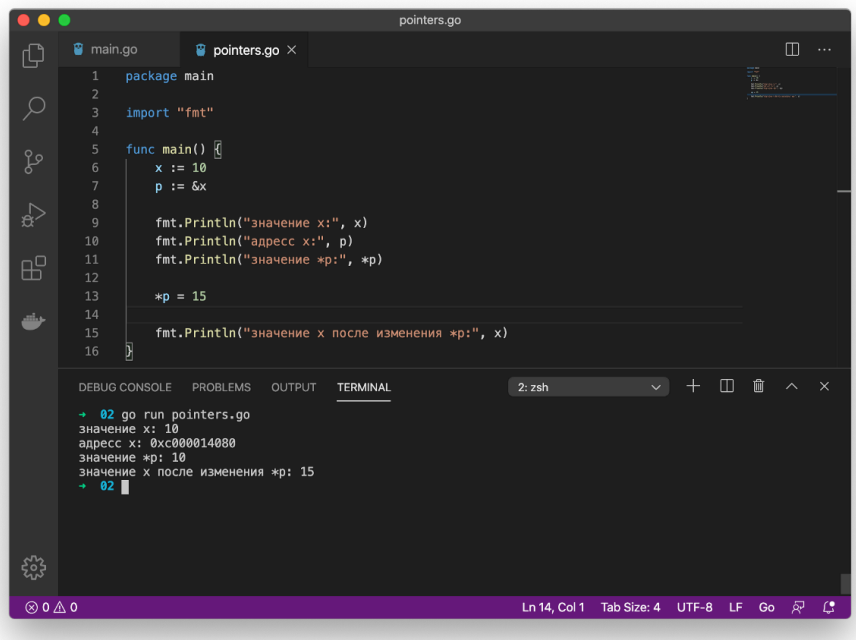
Используя символ `&` перед именем переменной `x`, мы обращаемся к адресу памяти, в котором хранится значение этой переменной.

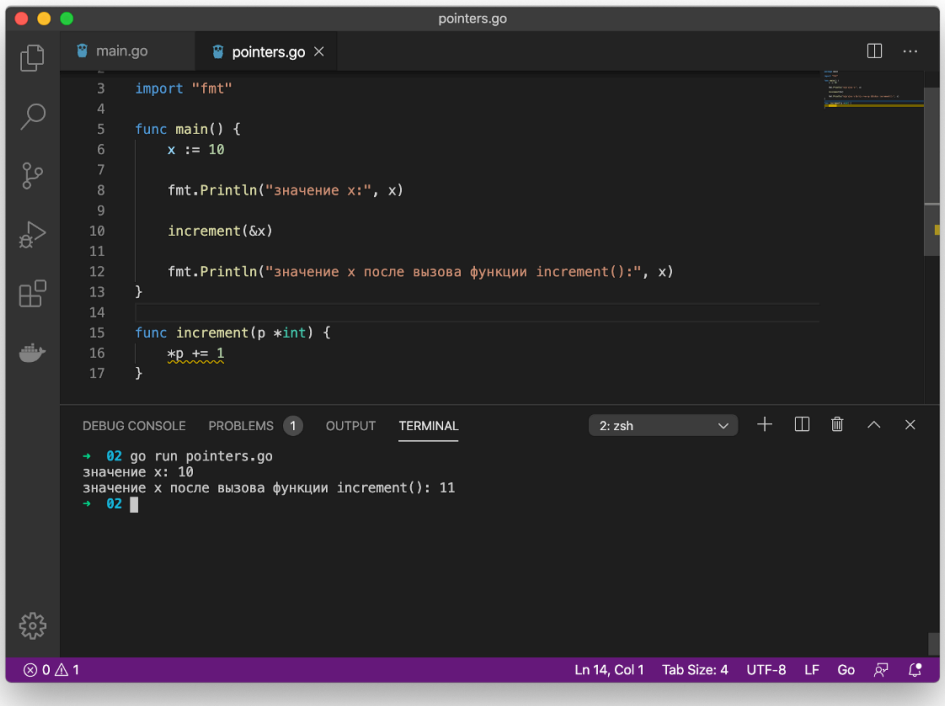
Также `&x` называют ссылкой на значение переменной `x` .



Мы можем создавать новые переменные которые будут указателями, присваивая им ссылки на другие переменные. В данном конкретном примере `p` является указателем на переменную `x` и хранит в себе ее адрес.

Используя синтаксис `*p` , мы обращаемся к значению, которое находиться по адресу `p` , т.е в данном случае это значение переменной `x` . Используя оператор присваивания, мы можем изменить значение, которое храниться в переменной `x` , т.к **мы изменили значение, которое находится по этому адресу в памяти.**

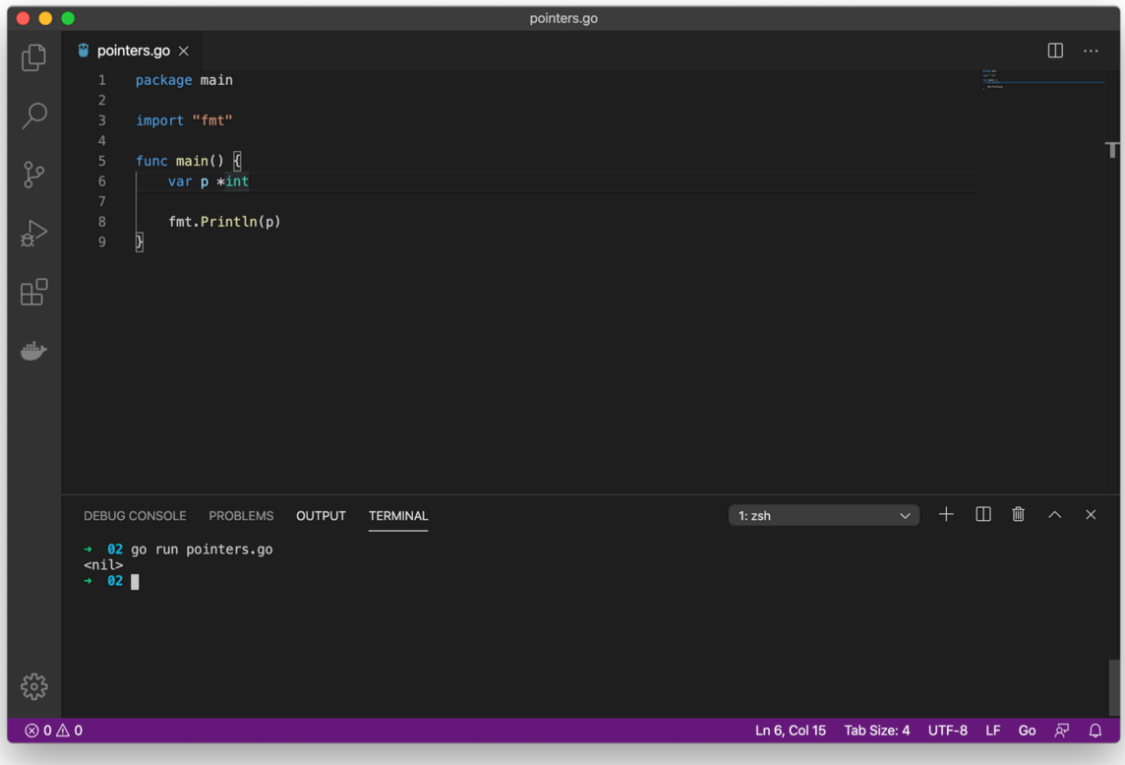




Передавая ссылки в функции, мы можем изменять значение переменных внутри тела функции.

Если же мы передаем не ссылку, а значение, то оно копируется в новую локальную переменную в рамках этой функции. Соотвественно, все операции над этой переменной не отображаются на передаваемом значении.

Существуют также пустые указатели. В примере мы инициализировали указатель, которые не ссылается ни на какую область в памяти, по этому его значение равно `nil` .



С этим ключевым словом мы уже сталкивались при работе с ошибками. Если ошибка не инициализирована, то она равна `nil` .

Именование (Нейминг)

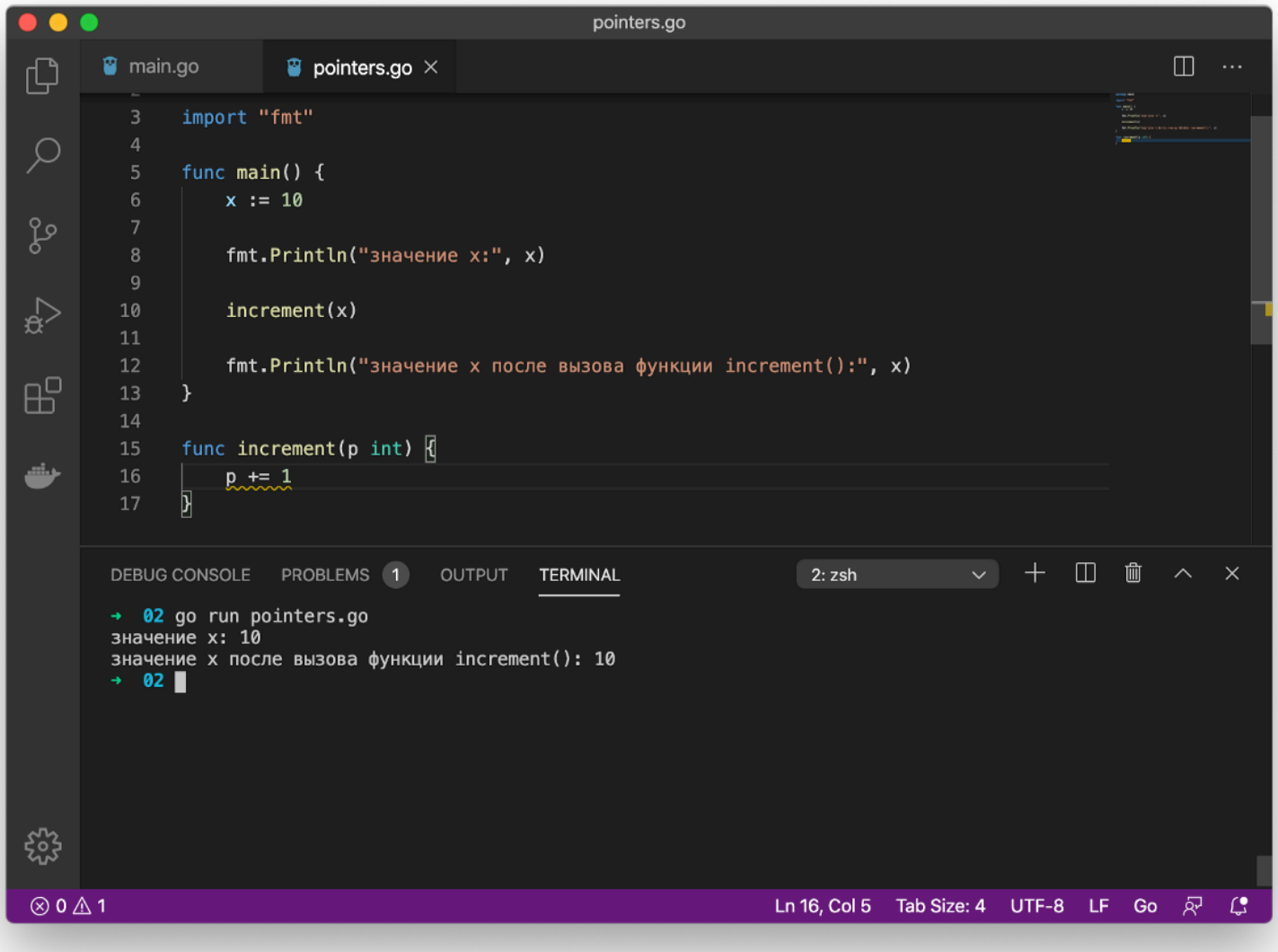
Go – чувствителен к реестру язык. Это означает что `name` и `Name` – это две разных переменных.

```
var name = "Вася"

var Name = "Петя"
```

Название переменных с маленькой и большой буквы также имеет функциональное назначение в языке. Эту тему мы подробнее разберем при детальном разборе пакетов.

Придерживаясь лучших практик языка, *следует использовать* ***camel case*** *при именовании переменных* т.е каждое новое слово в названии начинается с большой буквы, в отличии от **snake case,** когда слова разделяются нижним подчеркиванием.

```
var newVariable int // хорошо
```




```
var
new_variable int // оставьте такой синтаксис питонистам
```

Давайте резюмируем пройденный материал.

Для того чтобы заново использовать в своей программе уже написаные кусочки кода, используйте именованные функции. Они могут принимать и возвращать неограниченное количество аргументов.

Для проверок используется условный оператор `if` . Довольно часто вы можете встретить в программах на Go конструкцию `if err != nil {}` которая используется для обработки ошибок, возвращаемых другими функциями.

Каждая переменная хранится в выделенном блоке памяти, у которого есть свой уникальный адрес.

Мы можем создавать ссылки на переменные, которые будут хранить в себе этот адрес, и изменяя значения по адресу, менять значение самой переменной, которая в нем хранится.

Помните, что хорошей практикой является именование переменных и функций в ***camelCase***.

Посмотреть исходный код вы можете в этом репозитории, где собраны все примеры из данного курса.

Домашнее задание

Ознакомьтесь самостоятельно с библиотеками `errors` и `math` .

- https://golang.org/pkg/math/
- https://golang.org/pkg/errors/

Перепишите программу для подсчета площади круга, используя число Пи из библиотеки `math` . Напишите функции для подсчета площади и периметра прямоугольника и треугольника.

Также поищите дополнительные материалы про условные операторы. Ознакомьтесь с конструкциями `if {} else {}` и `if {} else if {} else {}` .

Найдите материал про функцию `new()` в Go, и какое оно имеет отношение к указателям.

Раздел 03:

Массивы и Срезы. Цикл for.

Мы прошлись по фундаментальным типам данных, которые являются атомами в наших программах. В этой главемы с вами рассмотрим уже более сложные структуры данных, молекулы, созданные путем объединения фундаментальных типов различными способами.

К составным типам данных можно отнести массивы, срезы, структуры и мапы.

В этом разделе мы остановимся подробнее на массивах и срезах, а также рассмотрим техники итерации в Go.

Массивы

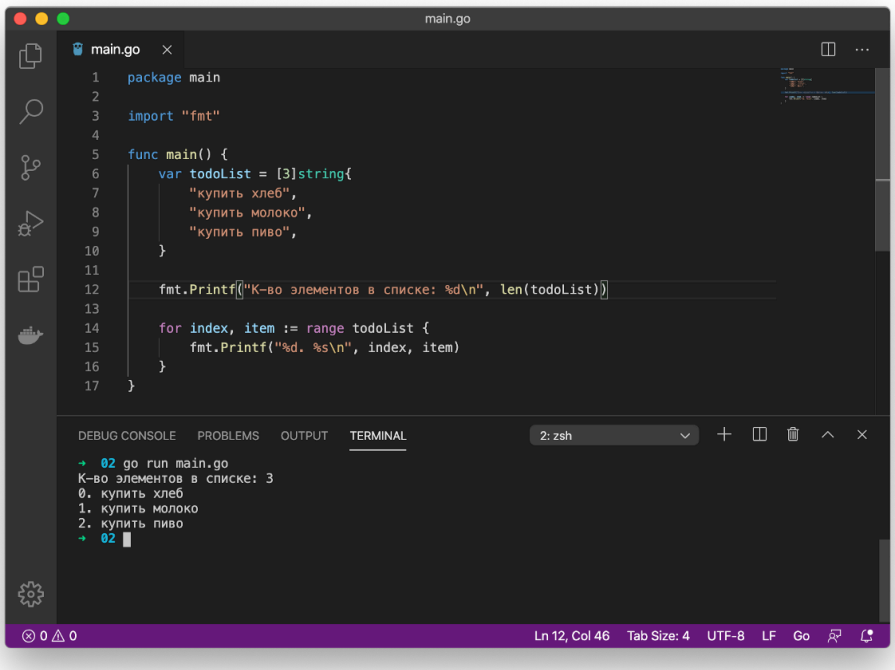
Очень часто нам нужно хранить в программе последовательность значений определенного типа. Для этой задачи обычные переменные не подходят т.к. они могут хранить в себе только одно значение. Тут нам на помощь приходят массивы.

Массив представляет собой *последовательность фиксированной длинны* из нуля или более элементов определенного типа.

Из-за фиксированной длинны массивы редко используются в Go непосредственно. Срезы, которые могут увеличиваться и уменьшаться, являются гораздо более гибкими, но, чтобы понять срезы, сначала следует разобраться в массивах.

Доступ к отдельным элементам массива осуществляется с помощью обычных обозначений индексирования в квадратных скобках `[]`, значения индексов в которых должны иметь значения от нуля до значения, на единицу меньшего длинны массива.

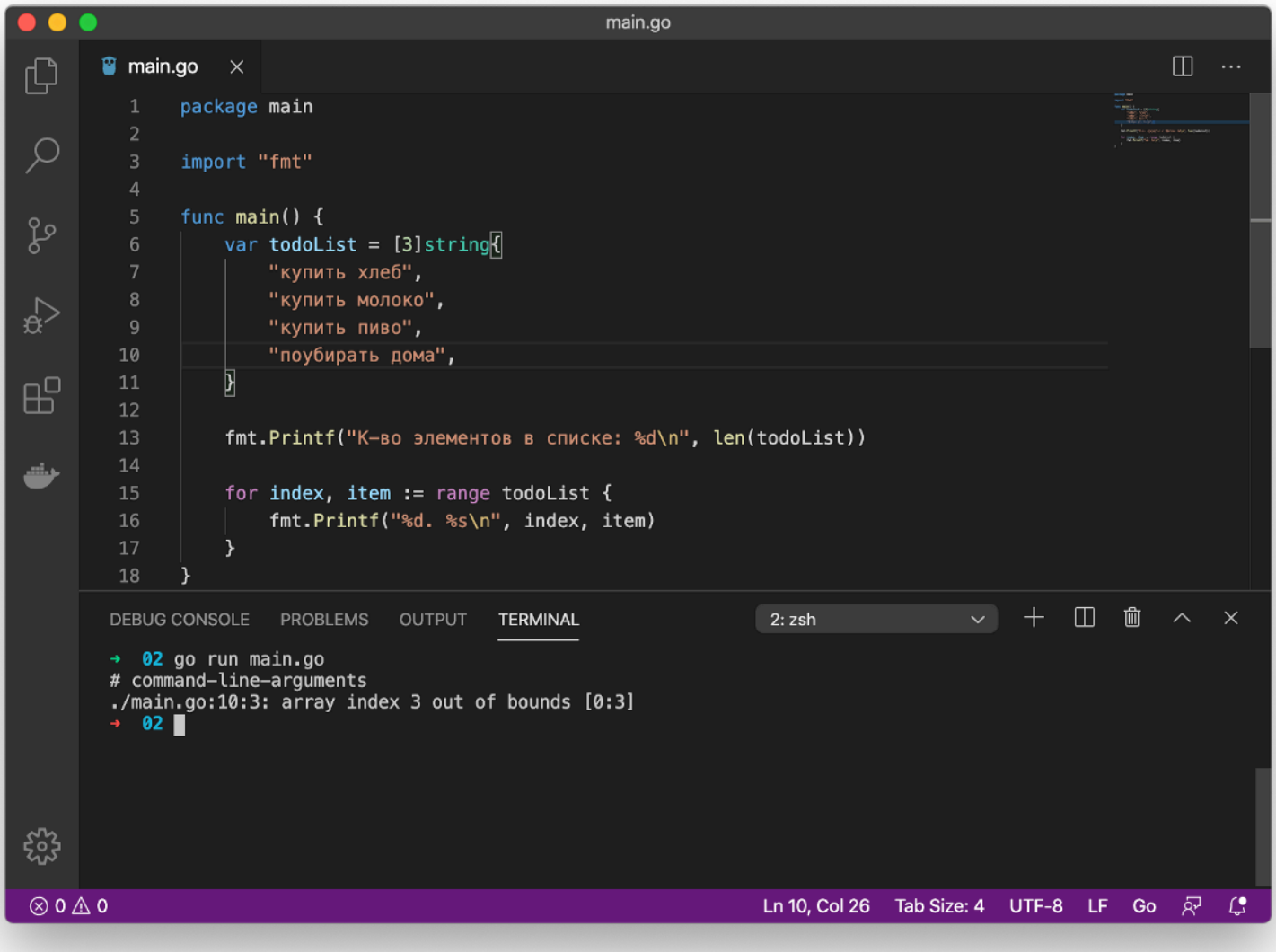
Встроенная функция `len()` возвращает количество элементов в массиве (его длинну).

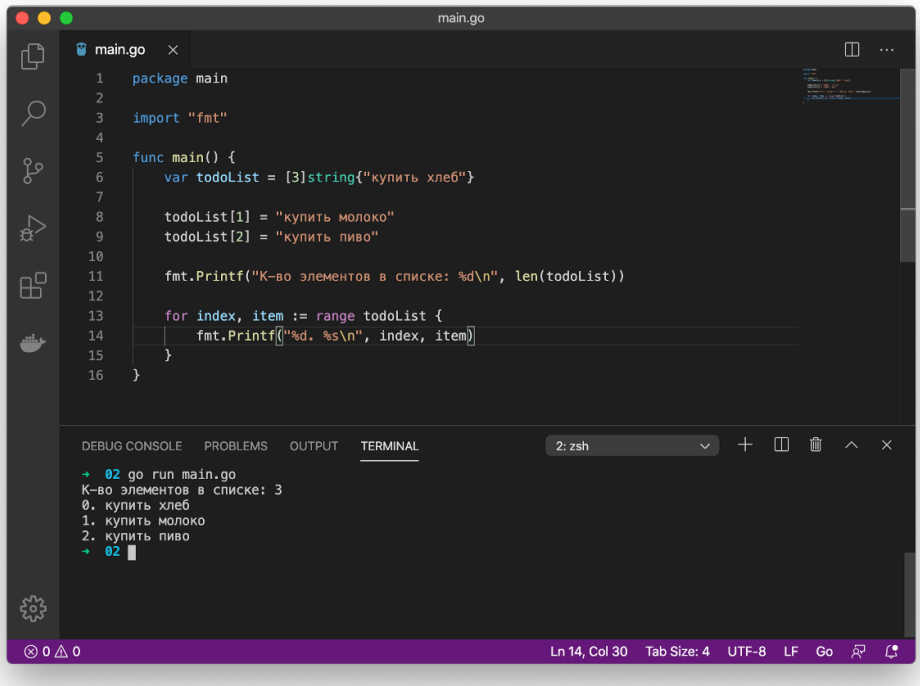


В примере мы определили массив `todoList` типа `[3]string{}`. Число в квадратных скобках указывает на длинну массива, а тип после скобок, соответственно, на типы элементов в массиве.

В фигурных скобках после объявления типа можно сразу перечислить элементы через запятую. Поскольку массив — последовательность элементов ***фиксированной*** длинны, мы не можем добавить в него больше элементов.

Если мы добавим 4-ый элемент, компилятор Go скажет нам, что мы не правы.

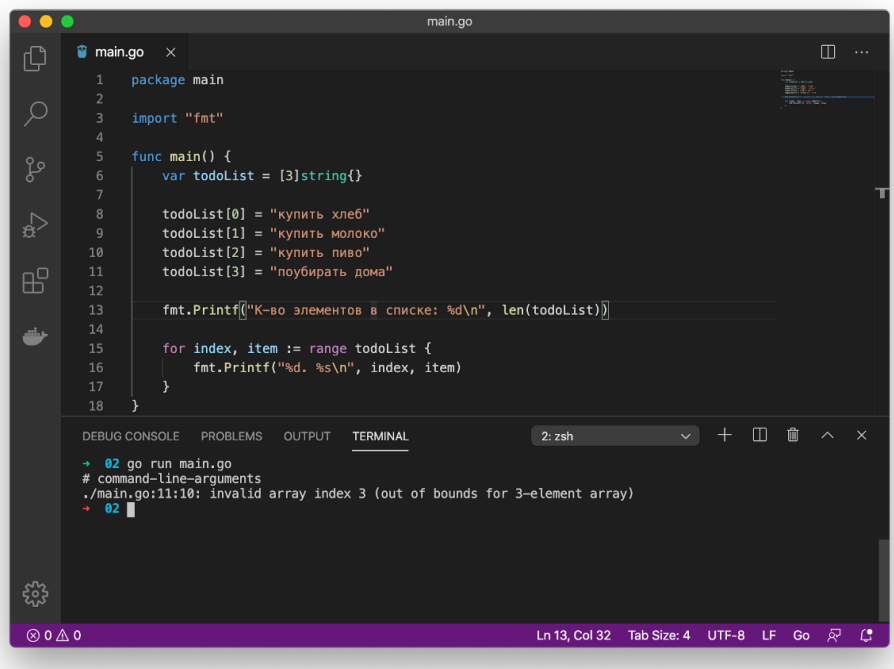


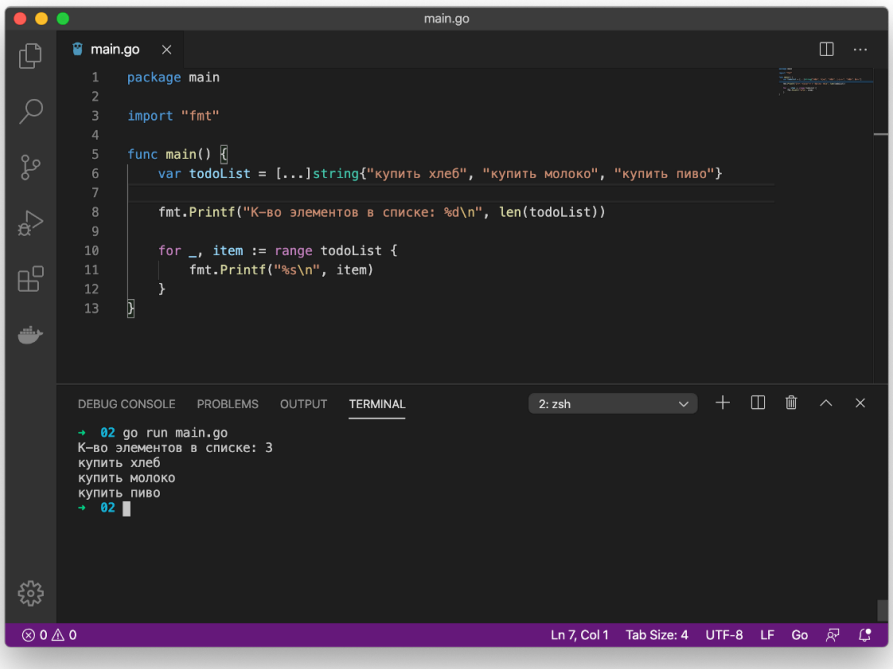


Однако не обязательно сразу заполнять все элементы массива. Скобки можно оставить пустыми, или заполнить только первый элемент.

Обращаясь по индексу в квадратных скобках `[1]` мы можем записывать и получать значения из массива.

Также, при обращении к индексу, который выходит за пределы длинны массива, компилятор будет ругаться.





Если в литерале массива на месте длинны находится троеточие “ . . . ”, то длинна массива определяется количеством инициализаторов.

Сейчас стоит подробнее рассмотреть конструкцию `for` .

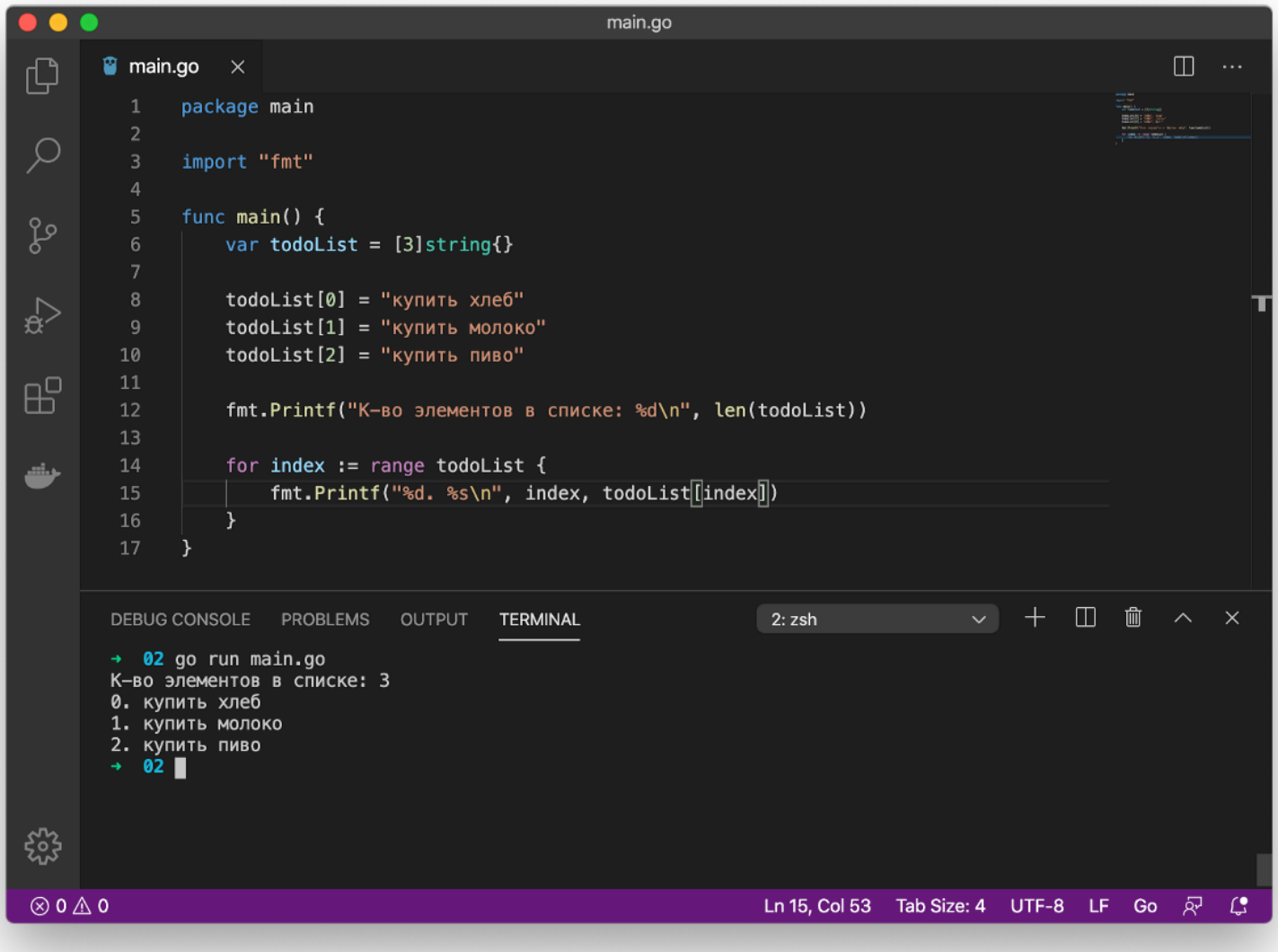
Цикл for

В языке Go есть только одна единственная конструкция `for` которая реализует возможность цикла, в отличии от того же C++, в котором также есть конструкции `while() {}` и `do {} while()` .

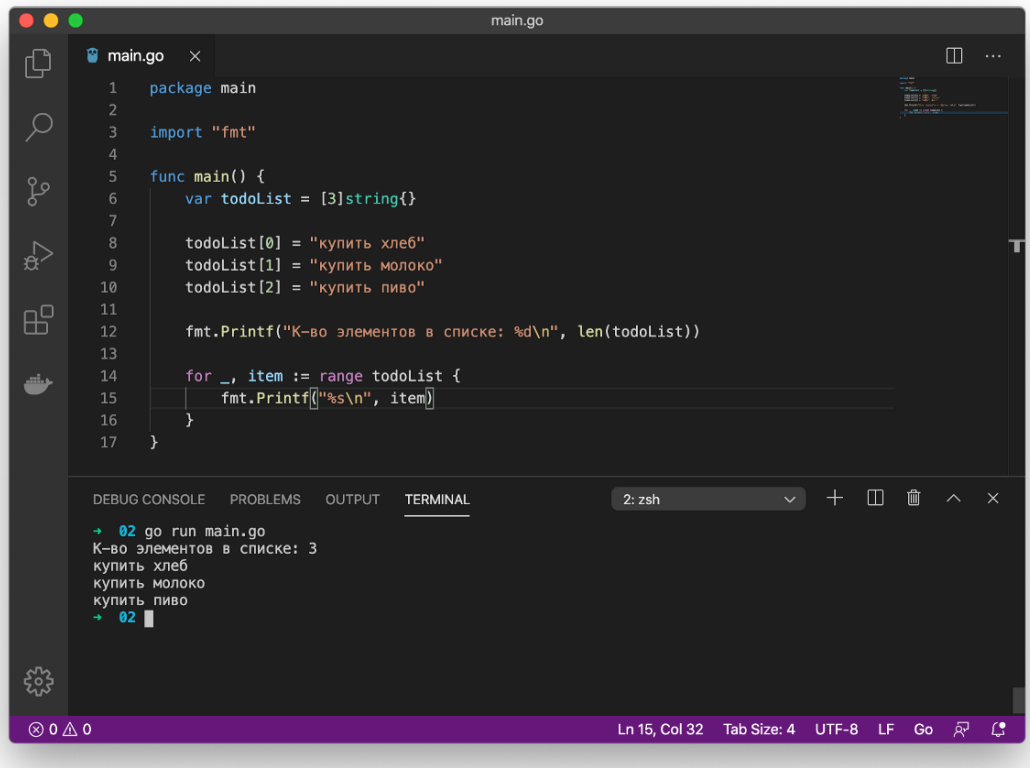
Для итерации по массивам, срезам и мапам используется следующая конструкция `for index, value := range array {}` , где `index` будет указывать на текущий индекс элемента при каждой итерации, а `value` – значение элемента по этому же индексу.

`index` и `value` – это обычные переменные, и вы вольны называть их как угодно в вашей программе. Также их можно опускать.

Например, если мы хотим иметь только индекс, а значение элемента нам не нужно, то мы можем воспользоваться следующей конструкцией `for index := range array {}`

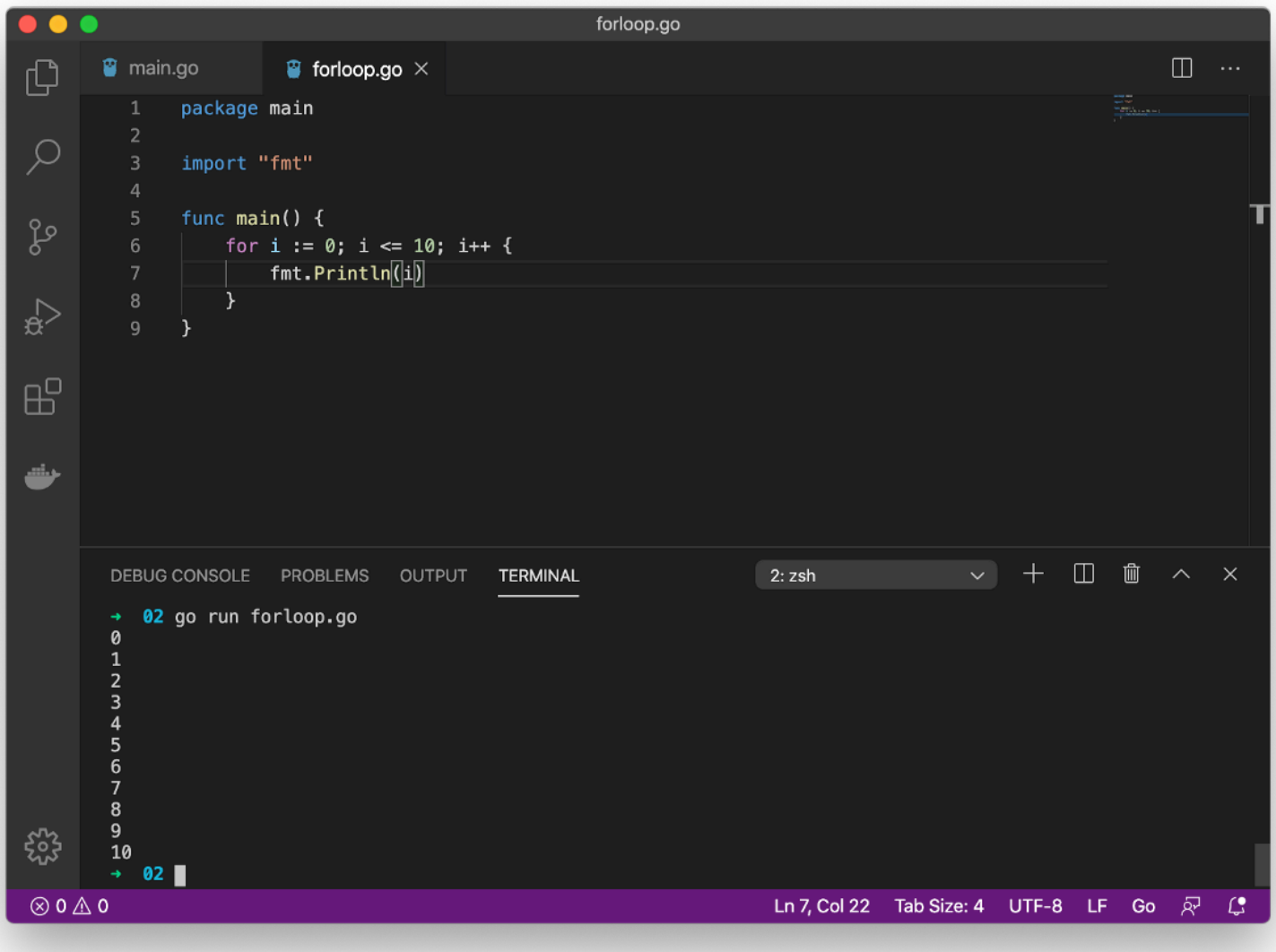


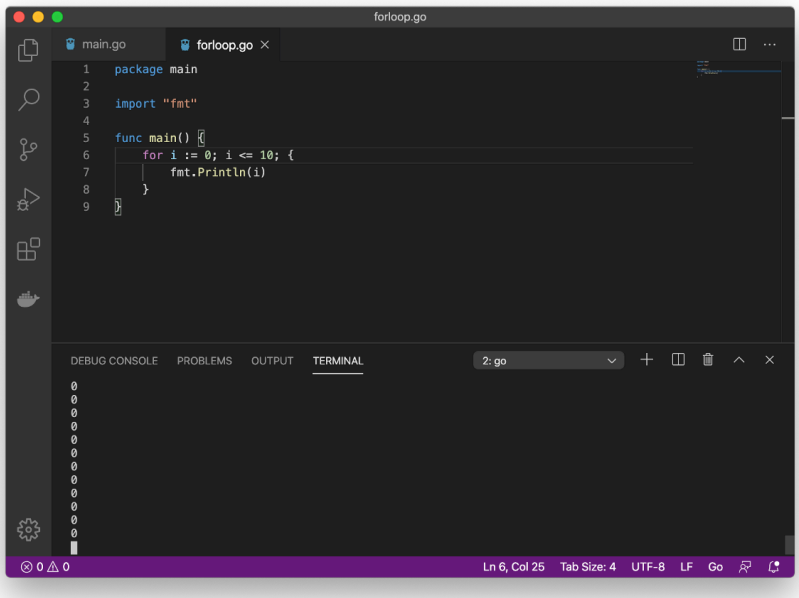
Или же, если в теле цикла мы не будем использовать индекс, то вместо него можно поставить нижнее подчеркивание.



Также цикл for используется для обычных итераций и имеет следующий синтаксис

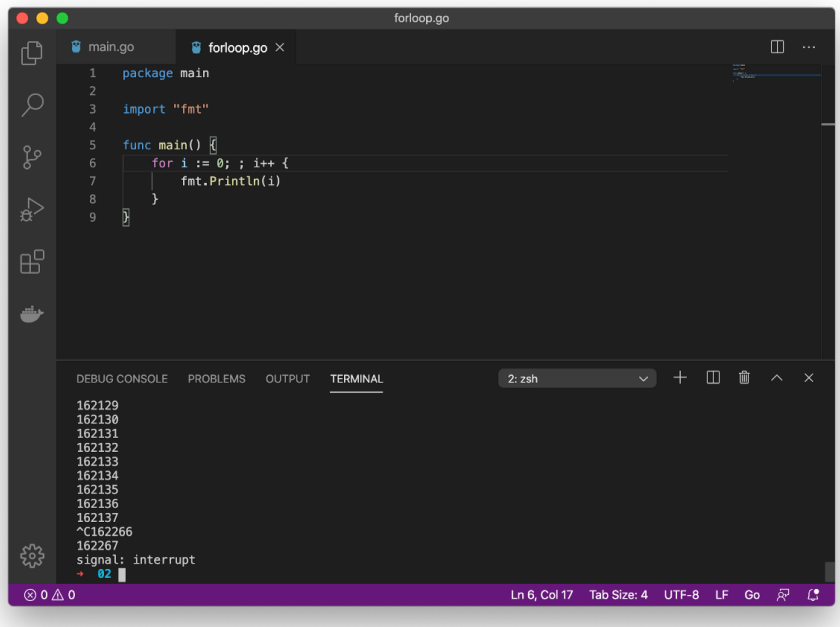
```
for инициализация переменной; условия, при котором цикл должен выполняться; операция, которую нужно выполнить после каждой итерации {}.
```

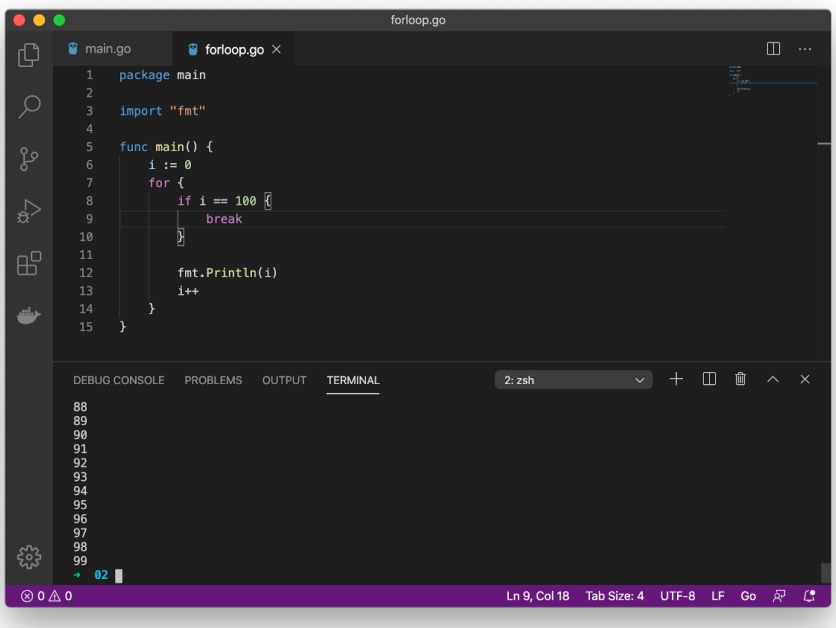




К тому же, некоторые блоки можно опускать. Например, опустив последнюю операцию инкремента, мы получим цикл, который будет бесконечно выполняться, т.к условие из второго блока никогда не выполнится.

Или же мы можем опустить условие, при котором цикл должен выполняться, и тогда он будет выполняться весь жизненный цикл программы.





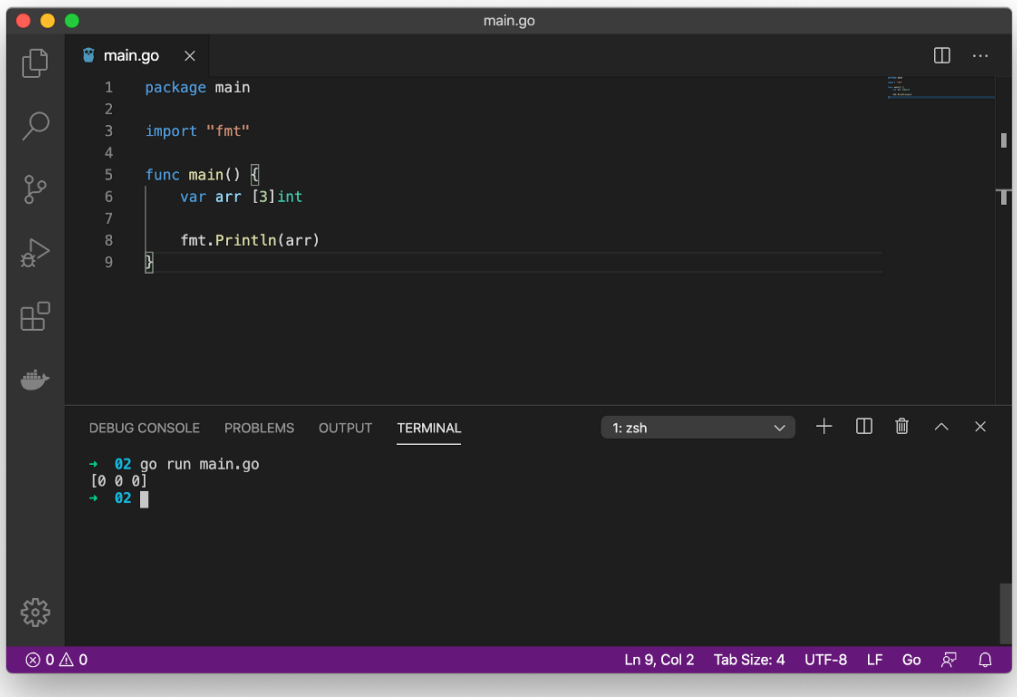
Для запуска бесконечного цикла используется конструкция `for {}`. Такие циклы находят свое применение в программах на Go, но это уже более продвинутые техники и тема для последующих уроков.

Для досрочного прерывания цикла используется ключевое слово `break`. А чтобы пропустить текущую итерацию и перейти к следующей, в Go есть ключевое слово `continue`.

Нулевое значение массива

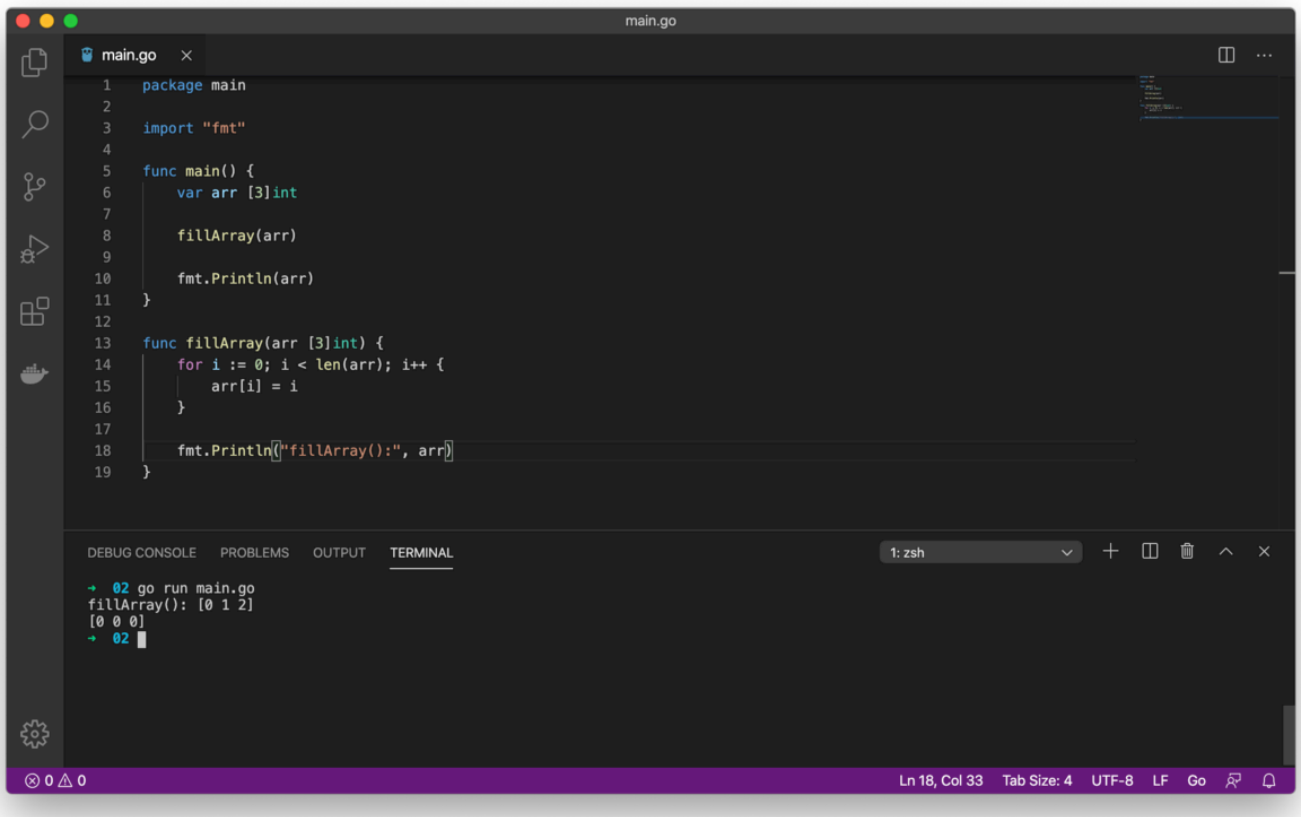
А давайте создадим пустой массив типа `int` длинною 3 элемента .

Как вы можете увидеть, при инициализации пустого массива, все его элементы по умолчанию равны своему нулевому значению.

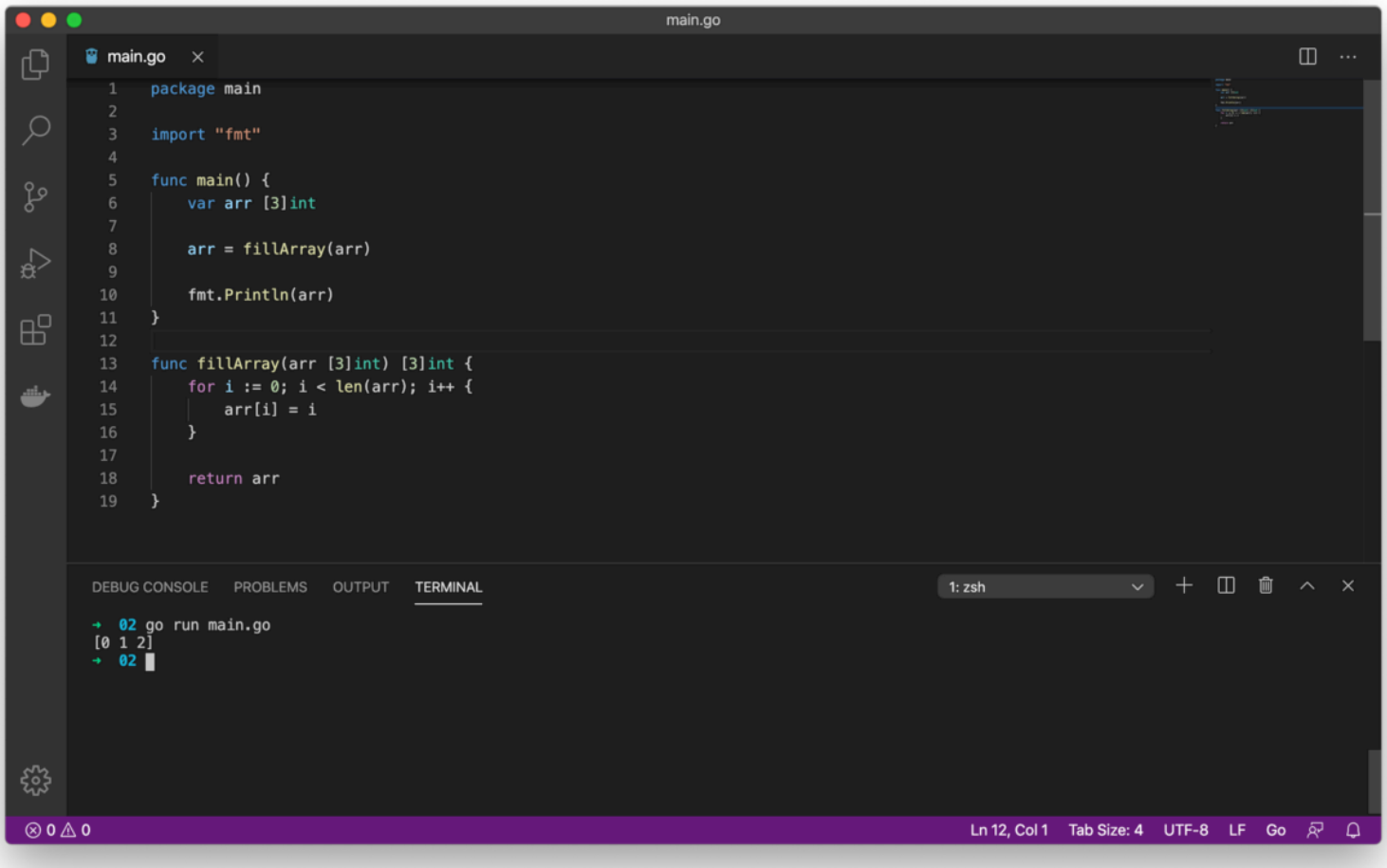


А что происходит когда мы передаем массив в функцию в качестве аргумента?

Массивы, по аналогии с обычными переменными, передаются в функцию по значению. Это означает, что его значение копируется в новую переменную внутри функции, и любые манипуляции с этим массивом никак не отображаются на нем вне тела этой функции.



На примере выше, после заполнения всех элементов массива, его значения остались по прежнему нулевыми внутри `main()` .



Если же мы перепишем нашу функцию так, что бы она возвращала массив, и будем присваивать этот результат в уже созданный массив, тогда его элементы изменятся, т.к. мы *переприсвоили* ему новое значение.

Срезы

Срезы (слайсы) очень схожи с массивами – это та же последовательность элементов одинакового типа, однако его длинна не фиксирована, а динамична.

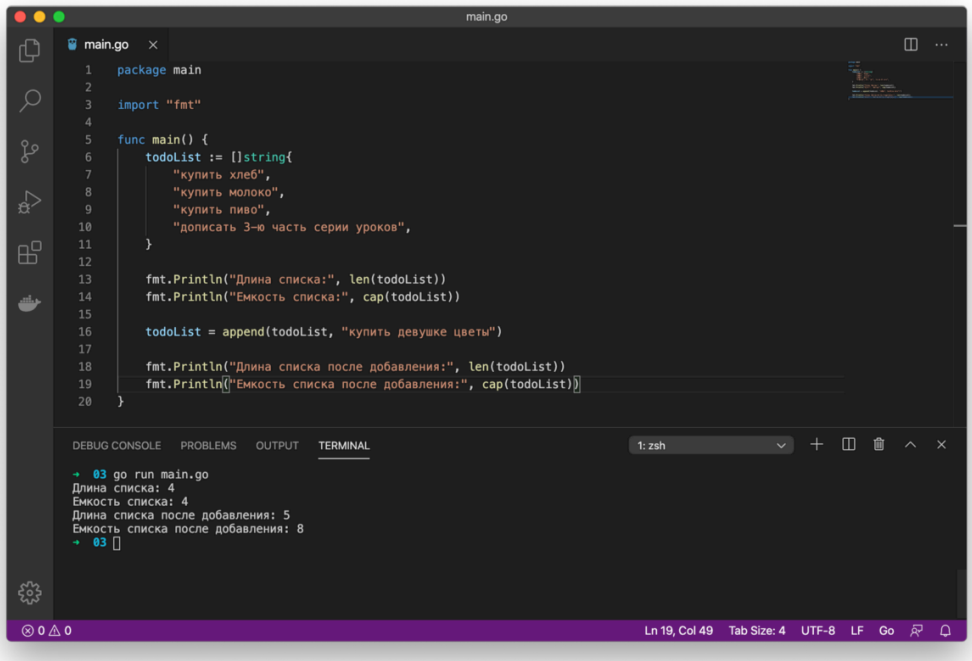
При инициализации срезов мы используем пустые квадратные скобки `[]` без явного указания размера.

Также мы можем добавлять элементы к срезу с помощью функции `append()` , которую мы рассмотрим далее.

У срезов есть такие параметры, как длинна и емкость. На примере ниже, после инициализации среза длинна и емкость равна 4.

Длинна – количество заполненных элементов в срезе. Емкость – количество всех доступных для заполнения элементов в срезе.

Давайте добавим еще один элемент к нашему срезу, и посмотрим как изменятся эти значения.



Как вы можете увидеть, длинна теперь равна 5, а емкость – 8.

Почему так? Сейчас разберемся, это также поможет нам понять как работает функция `append()` .

Все дело в том, что срез, по сути, является указателем на уже знакомый нам массив фиксированной длинны. Во время компиляции анализируется количество элементов среза и создается массив с длинной, равной

их количеству.

Если при добавлении нового элемента в срез с помощью функции `append()` у массива, который был выделен при инициализации среза, нету места, создается новый массив, с длинною вдвое больше. Все элементы старого массива копируются в новый+ записывается новое значение.

Соответственно, при инициализации нашего среза, его длинна была 4, и емкость тоже 4, т.к массив, на который ссылался срез был длинной в 4 элемента.

Когда же мы выполнили функцию `append()` , то:

- Новый элемент вышел за рамки длинны массива
- Создался новый массив, с длинною вдвое больше (4 -> 8)
- Срез начал ссылаться на новый массив
- Все значения из старого массива скопировались в новый + добавился еще один элемент

Теперь основанный на этом массиве срез имеет ёмкость 8 и длину 5.

Функция `append()` имеет следующий вид:

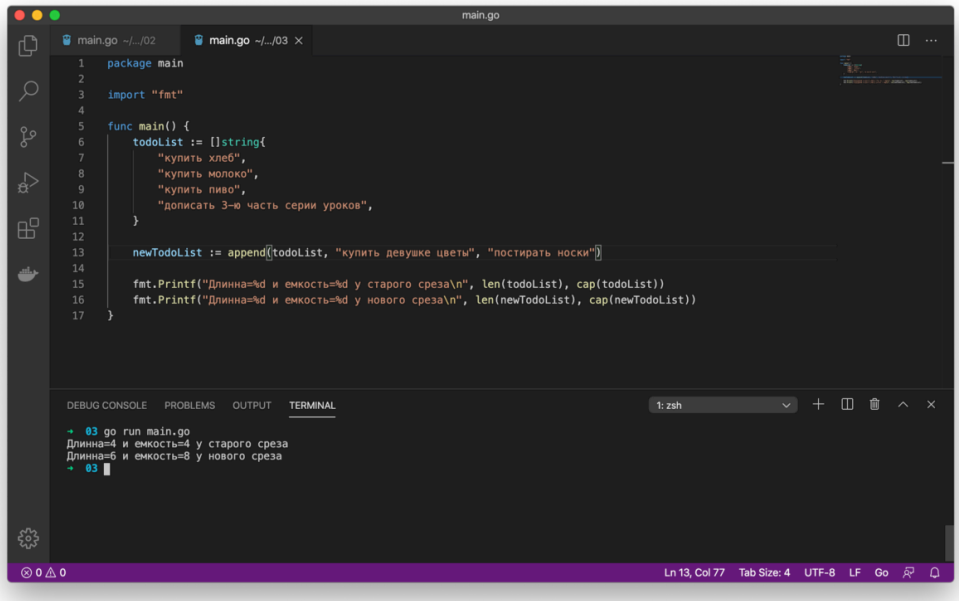
func append(slice []Type, elems ...Type) []Type

Она принимает срез, и один/несколько элементов того же типа что и сам срез.

Хочу обратить ваше внимание на `...Type` , такая запись в аргументах функции означает, что можно передать один и более элементов этого типа через запятую.

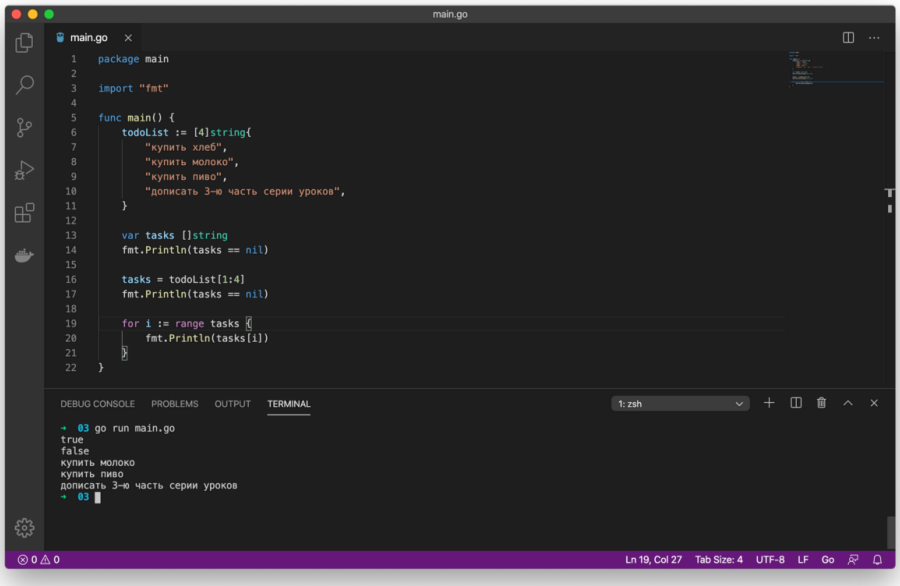
Функция `append()` не изменяет переданный в аргументах срез, а возвращает новый.

Если мы присвоим результат выполнения этой функции в новый срез, то увидим что старый не изменился.



Все это время мы с вами работали с *анонимными срезами*.

Теперь, чтобы лучше понять как срез ссылается на массив, давайте рассмотрим следующий пример.

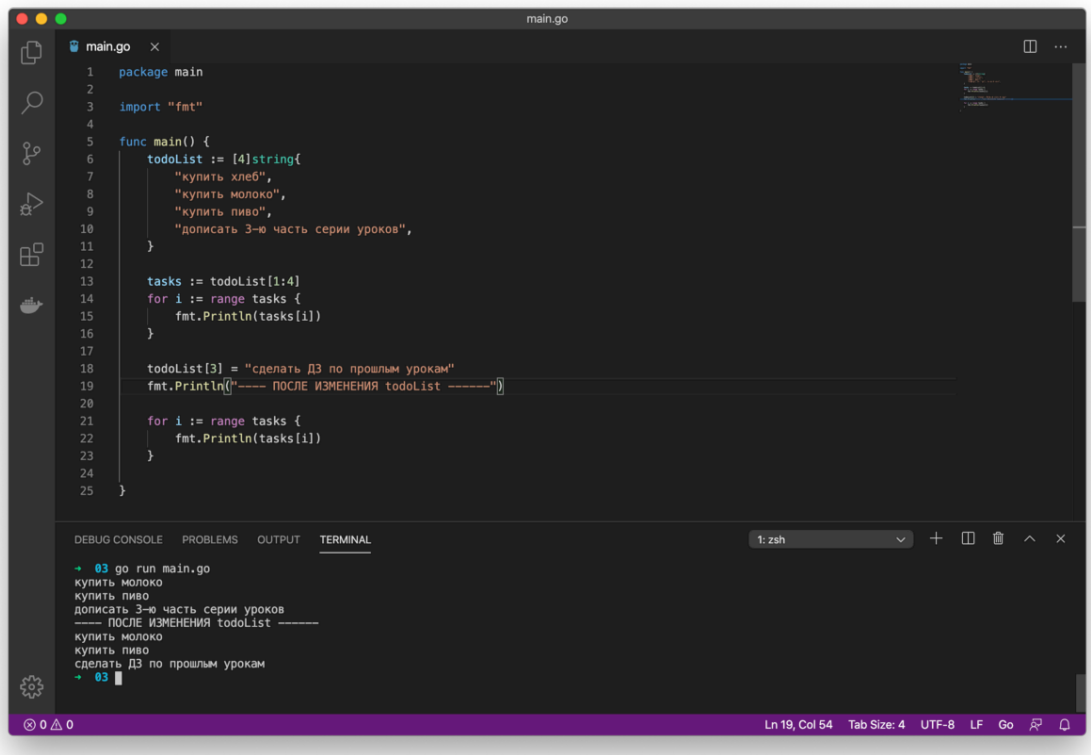


Мы создали пустой срез. Когда он не ссылается ни на какой массив, его значение по умолчанию равно `nil`.

Далее мы присвоили ему срез нашего массива с 1 по 4 элемент. Синтаксис `[m:n]` позволяет делать выборку элементов массива или среза от индекса `m` до `n`. Пустые скобки `[:]` означают выборку всех элементов.

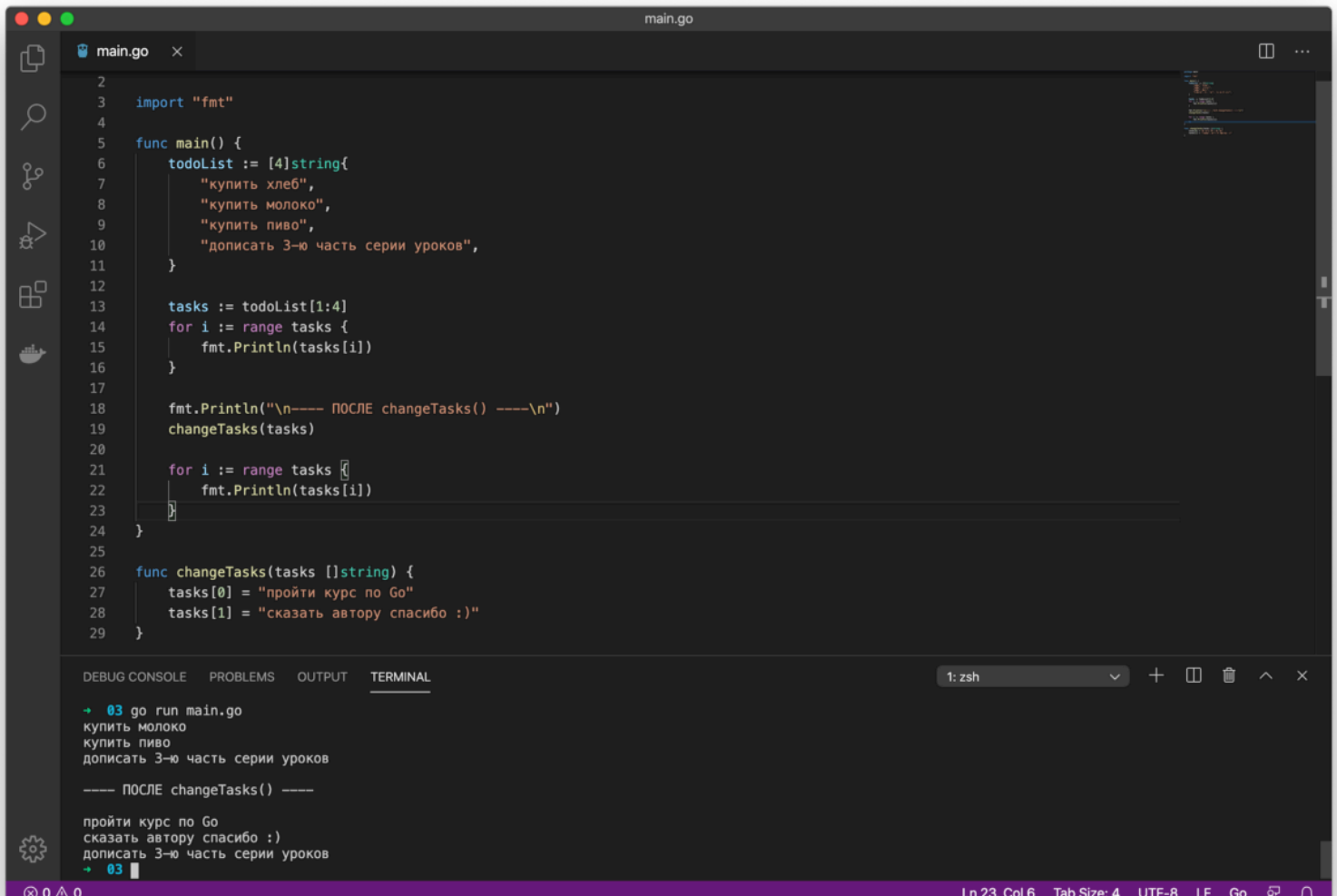
После присвоение у среза его указатель ссылается на массив `todoList`, по этому значение среза теперь не равно `nil`.

Если мы изменяем элементы самого массива, на который ссылается срез, то это отображается и на самом срезе.



Помните мы с вами разбирали передачу массива в функцию в качестве аргумента. Массив передается по значению, по этому его изменения внутри функции никак не отображаются на самом массиве.

Давайте теперь сделаем то же самое со срезом.



Поскольку срез – это указатель, при передаче его в функцию он передается по ссылке. То есть, мы работаем с той же областью в памяти, в которой хранятся значения массива, на который ссылается срез.

Изменяя внутри функции переданный срез, мы тем самым изменяем исходный срез.

И еще. Поскольку мы изменяем указатель на массив внутри функции, то и сам этот массив должен измениться.

Давайте резюмируем.

Массив – это последовательность элементов одинакового типа фиксированной длинны.

Срез в свою очередь такой же массив, только его длинна динамична.

Срез является указателем на массив. Изменения элементов среза изменяют массив, на который он ссылается, и наоборот.

Для итерации по срезам и массивам мы используем цикл `for` .

У пустого массива все его элементы равны нулевому значению. Пустой срез равен `nil` , т.к указатель на массив среза пустой.

Домашнее задание

Поиграйтесь самостоятельно с массивами и срезами.

Как бы вы написали функцию, которая переворачивает все элементы массива или среза местами?

Как бы вы реализовали функцию, которая в качестве аргумента принимает число `n`, и выводит в консоль все числа от 2^1 до 2^n.

Поищите дополнительный материал про функции `make()` и `copy()` . Для чего они нужны и как используются ?

Ознакомьтесь с набором трюков со срезами в репозитории самого языка Go: https://github.com/golang/go/wiki/SliceTricks .

Раздел 04:

Структуры и Мапы. Кастомные Типы.

В данном разделе мы продолжим разбирать составные типы данных, а конкретно поговорим про структуры и мапы. Эти типы очень часто используются в программах на Go, и усвоив данный материал можно продолжать изучение более сложных техник и концепций.

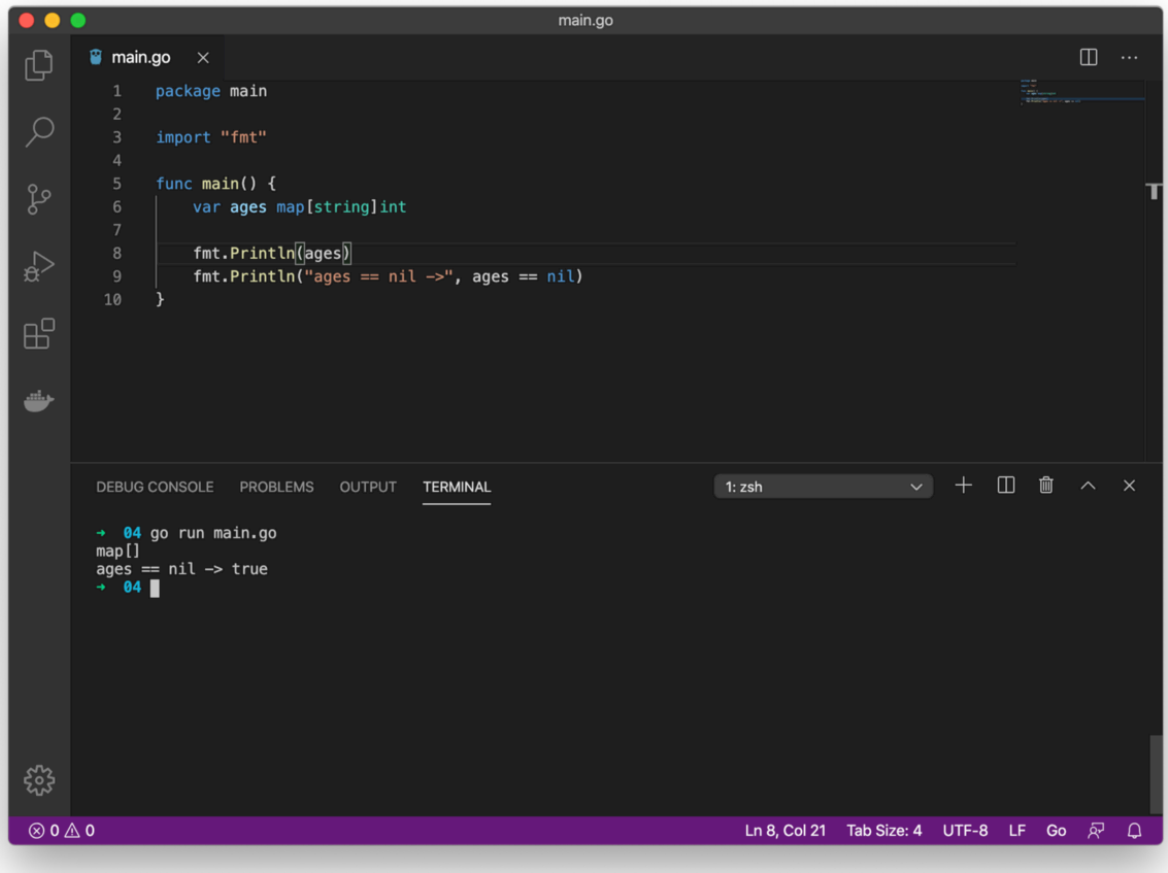
В данной книге мы с вами двигаемся от простых концепций к более сложным. Чтобы понимать как устроены срезы, для начала нужно было пройтись по переменным, указателям и массивам.

Надеюсь у вас не возникает трудностей с восприятием материала, а после изучения новой темы вы берете инициативу на самостоятельное закрепление знаний на практике.

Мапы (или отображения)

Мапы очень схожи с массивами – они также хранят последовательность элементов, однако у массива каждый элемент находится по индексу, который является целочисленным числом `int`.

Мапа же, в свою очередь, вместо индекса имеет ключ, тип которого мы



вольны сами определять.

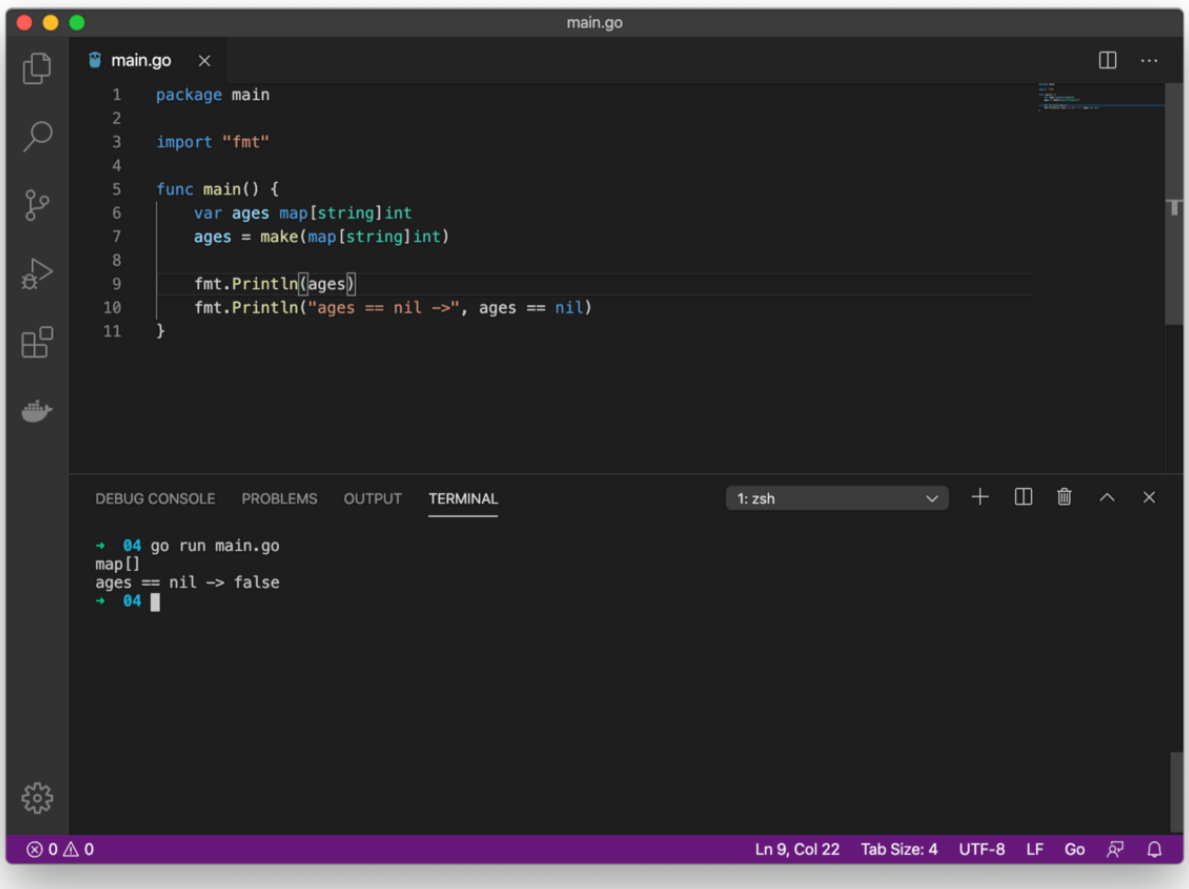
Мапа хранит себе данные в формате `ключ:значение` , и выглядит это примерно следующим образом:

```
{
    "ключ1": значение1,
    "ключ2": значение2,
    ...
}
```

Синтаксис объявления мапы следующий:

```
var mapName map[keyType]valueType
```

Где `keyType` – тип ключа, а `valueType` – тип значения.

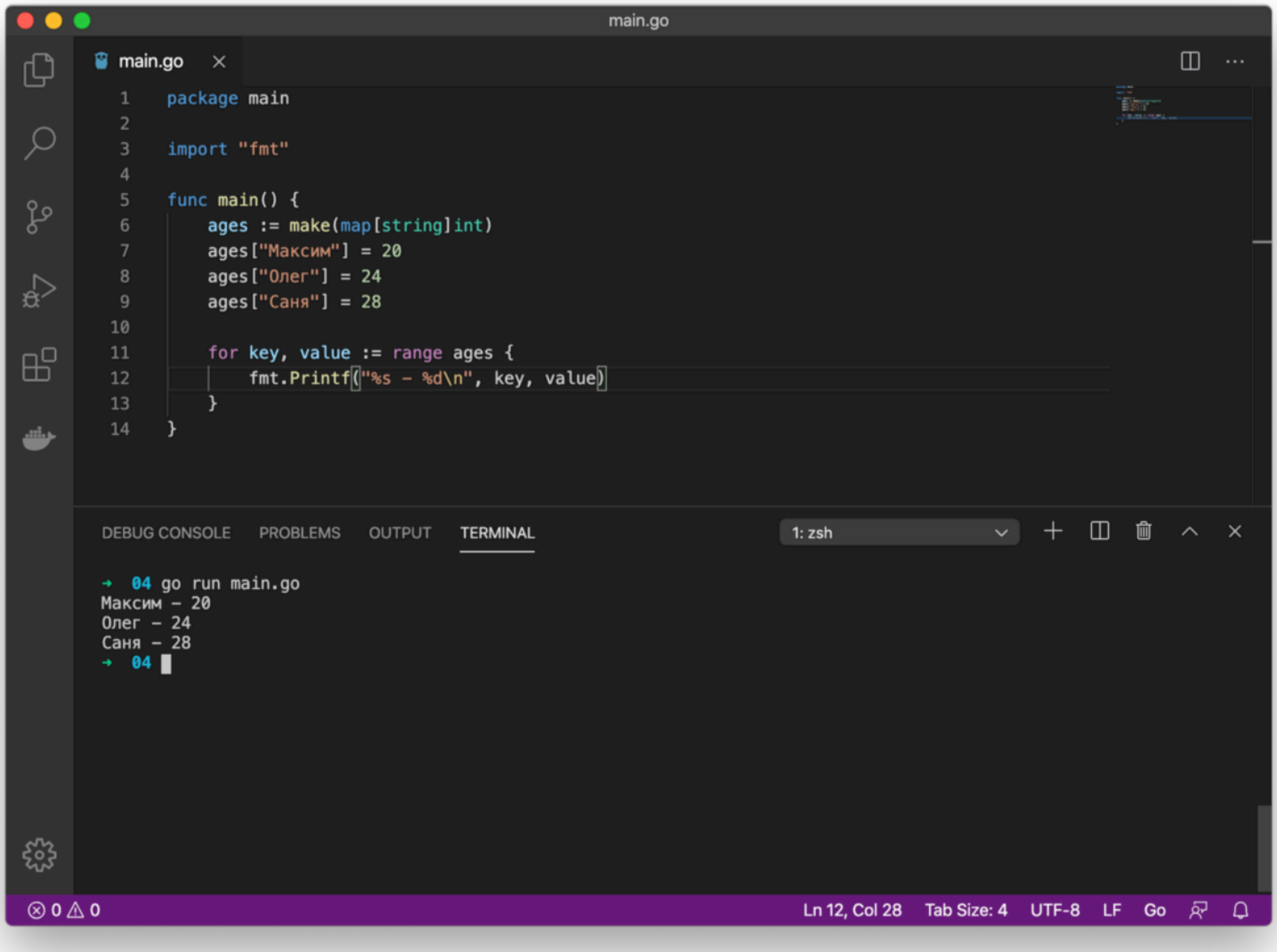


При инициализации пустой мапы, по аналогии со срезом, ее значение равно `nil`. Так происходит потому что мапы в Go также являются по своей сути указателями на базовые типы данных.

Чтобы инициализировать пустую мапу, воспользуемся встроенной функцией `make()` .

Кстати, совсем не обязательно указывать длину вторым аргументом, как в случае со срезами, хотя и есть такая возможность.

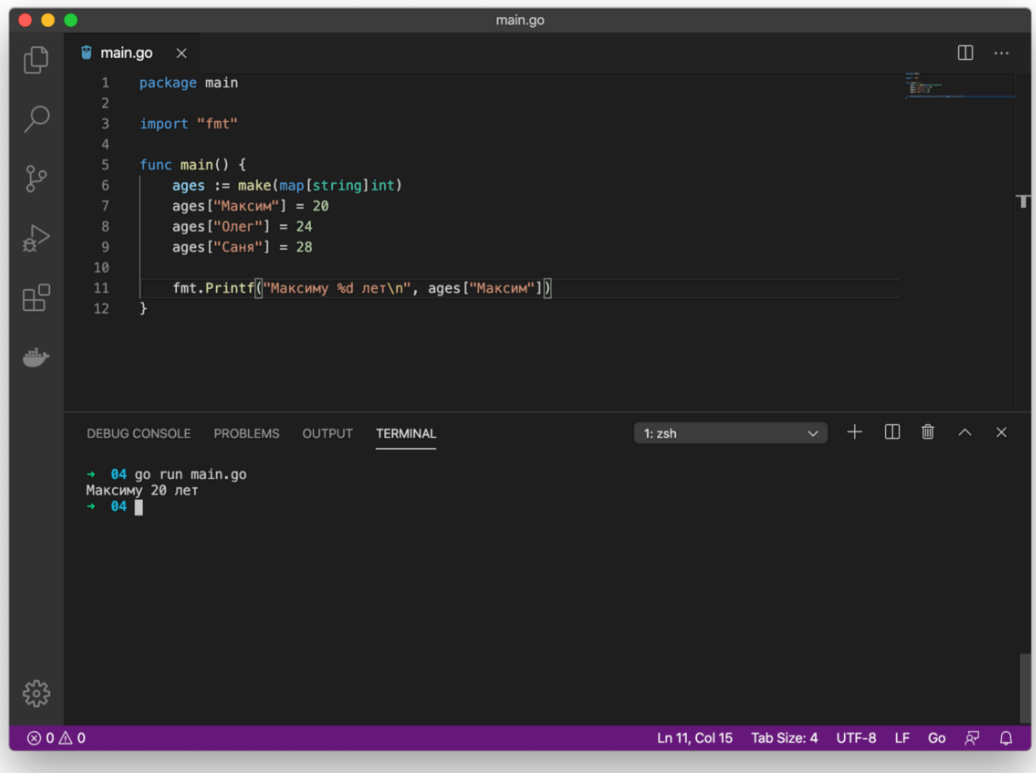
В примере мы инициализировали пустую мапу и теперь можем записывать в нее значения.



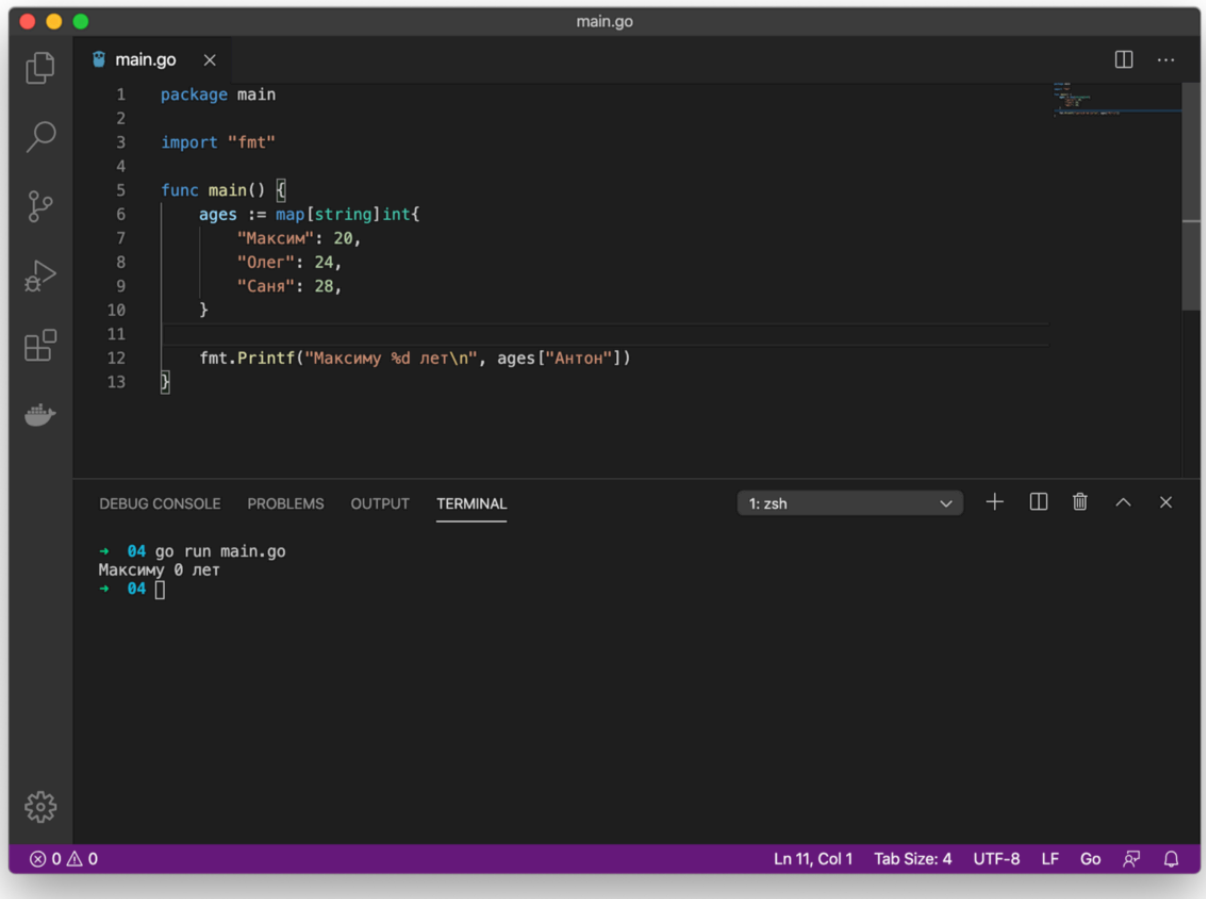
Запись значений в мапу схожа с массивом, только вместо индекса мы указываем ключ. Итерация по мапе аналогична со срезами и массивами.

Получение значения из мапы происходит по обращению в квадратных скобках к нужному ключу.

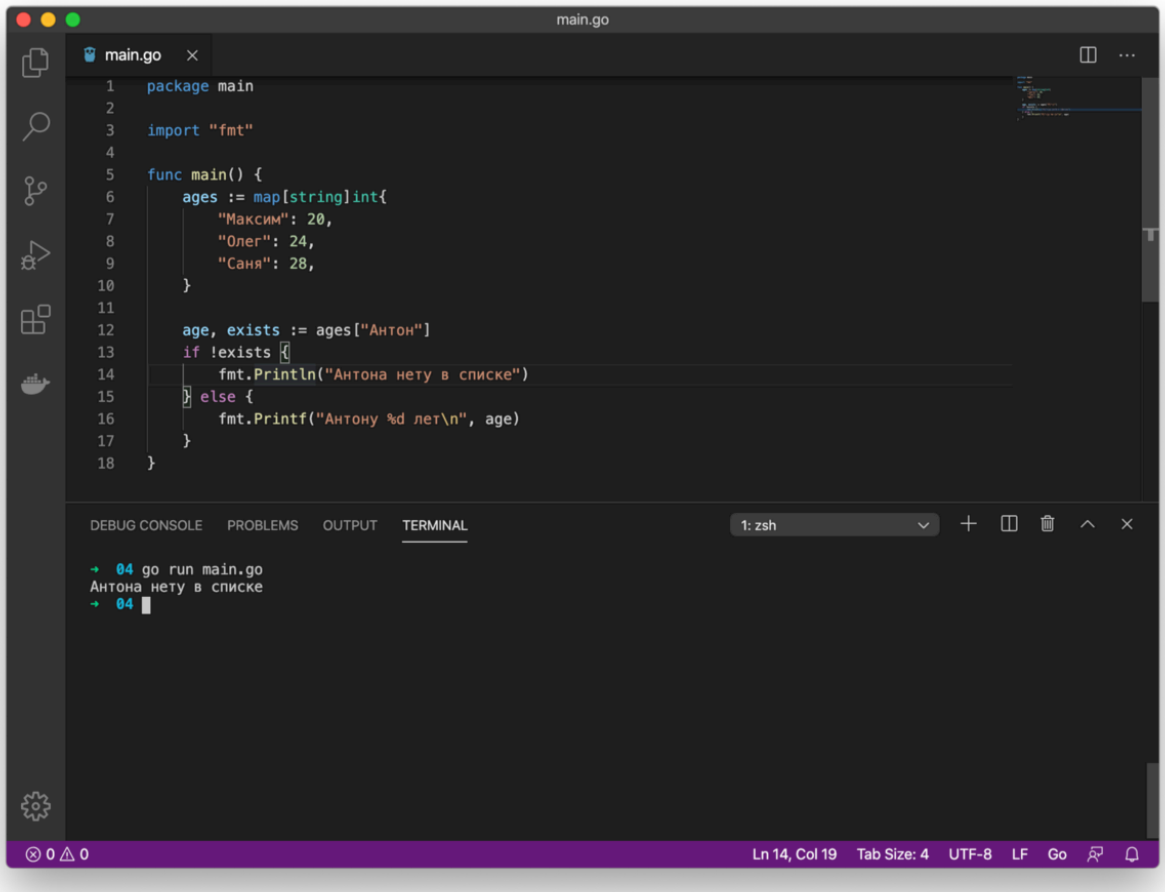
Также, по аналогии со срезами и массивами, можно инициализировать мапу сразу с неким набором значений.



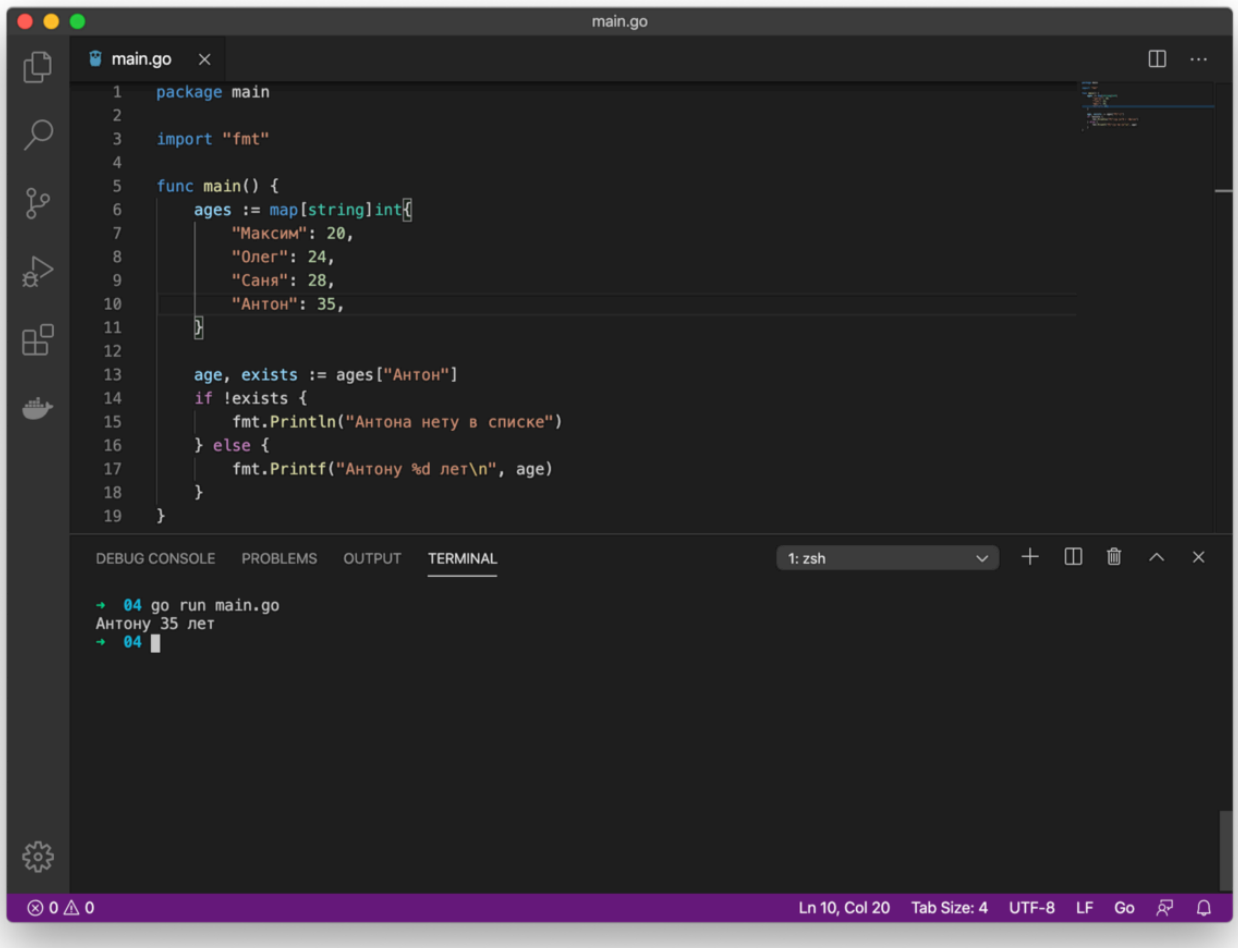
Обращение к несуществующему элементу мапы вполне корректно, и будет содержать в себе нулевое значение типа.



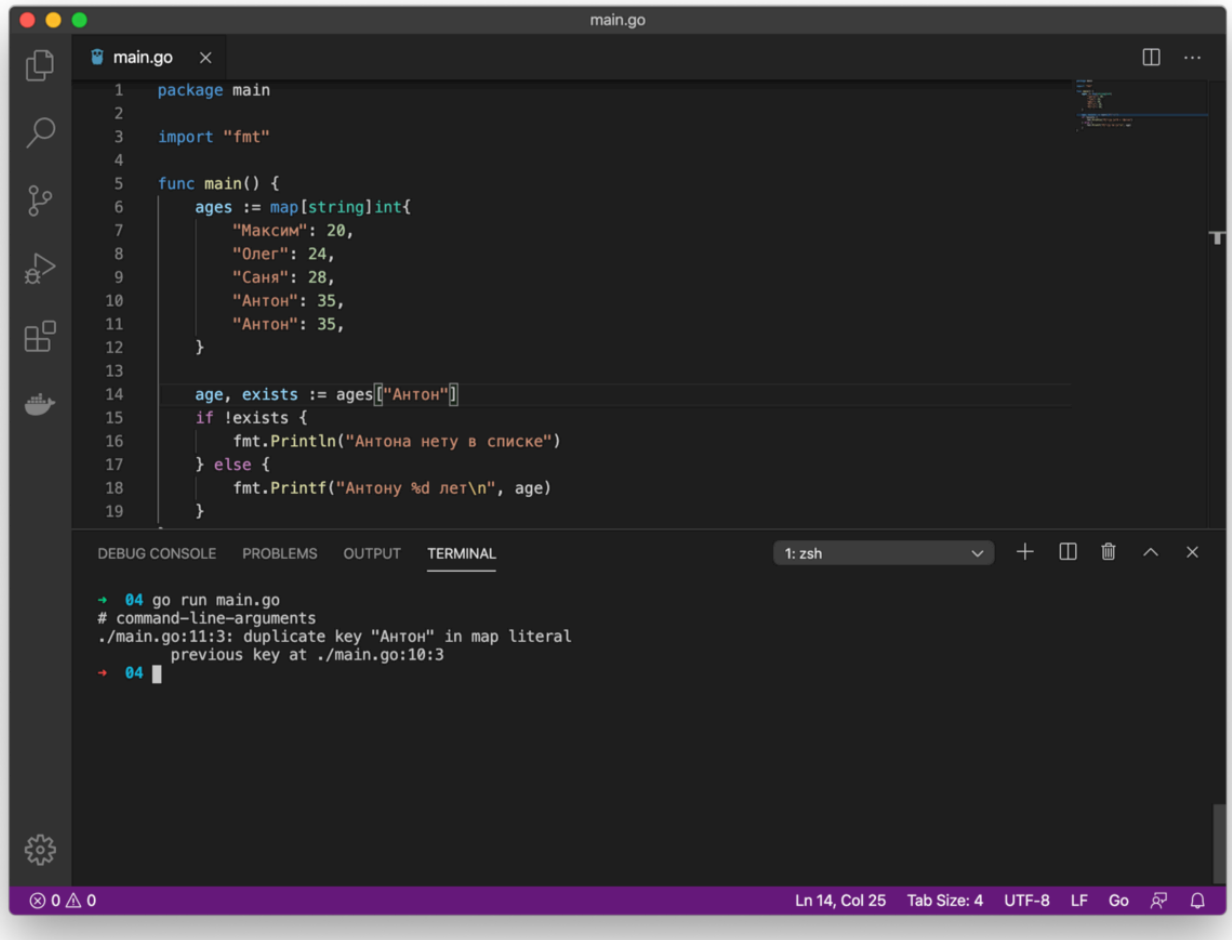
Для того, чтобы проверить, содержится ли данный ключ в нашей мапе, Go может возвращать 2 значения по обращению к элементу мапы по ключу: само значение и `bool`, который `true` если такой элемент есть и наоборот.



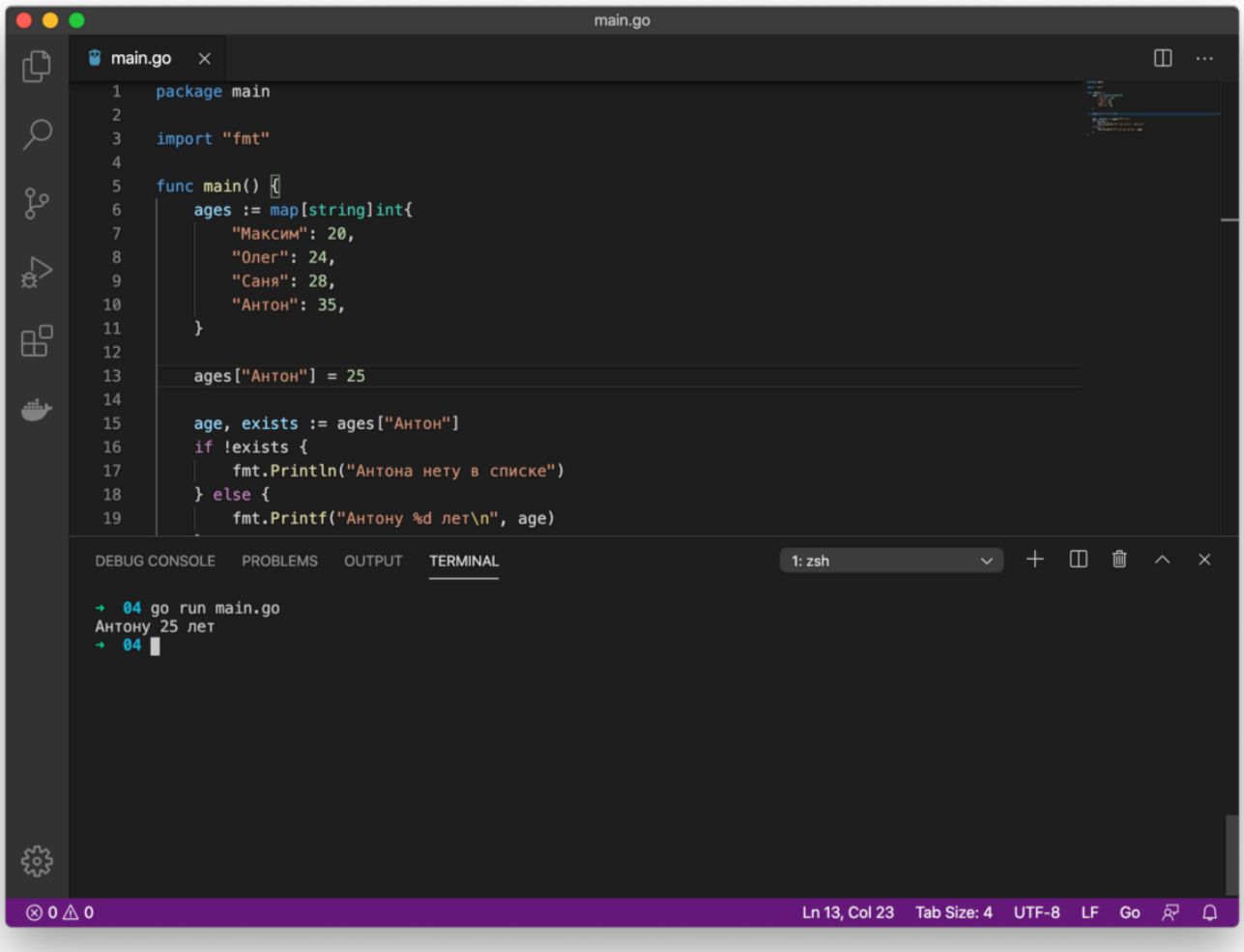
В данном случае `exists == false`, т.к. Антона нету в нашем списке. Но если мы его добавим, то сработает условие `else` потому что в таком случае будет `exists == true`.



Нельзя дублировать ключи в мапе, это приведет к ошибке компиляции.

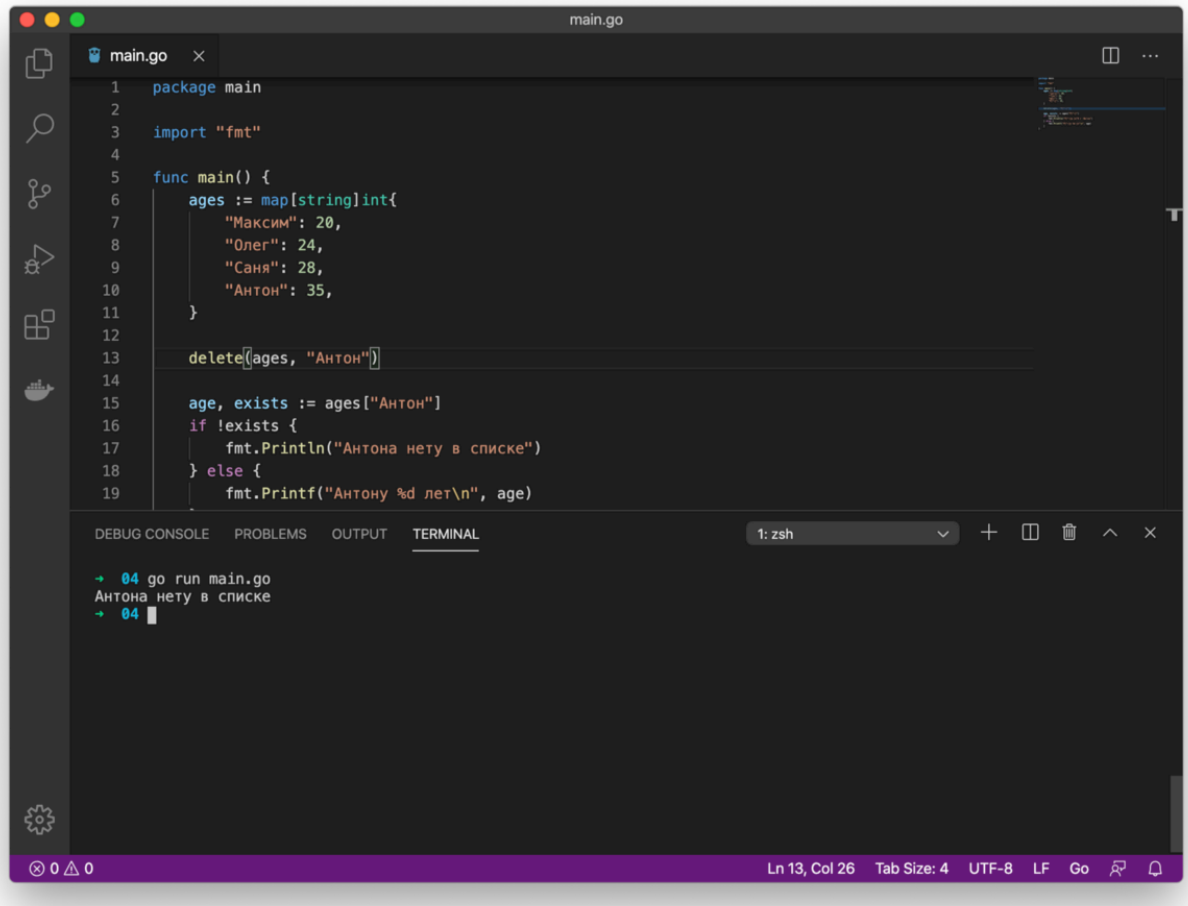


Однако значение по ключам можно свободно перезаписывать.

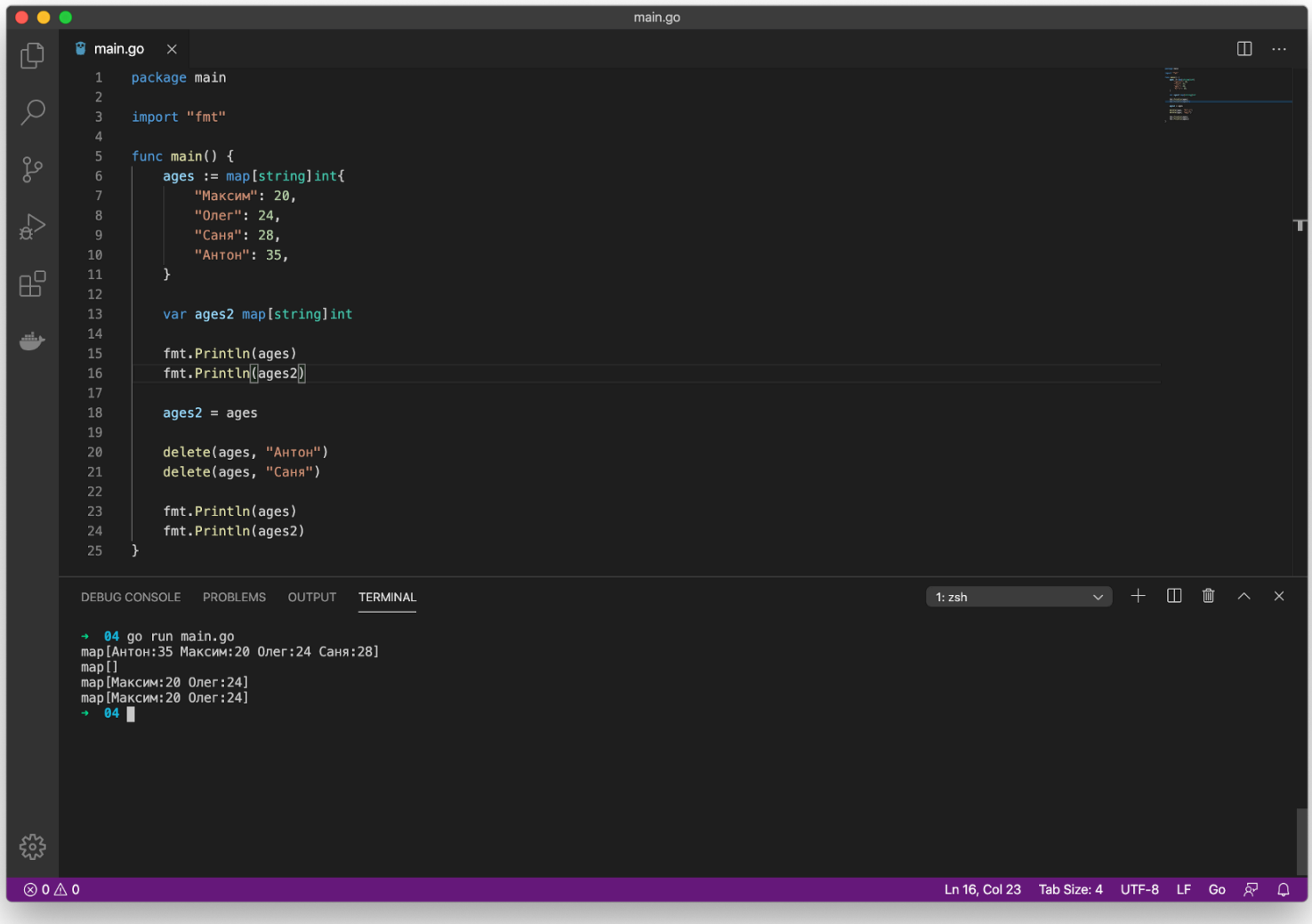


Функция delete()

Используя встроенную функцию `delete()`, можно удалять ключи из мапы, указав в качестве аргумента саму мапу и ключ.



Как мы уже говорили, мапы, как и срезы, являются указателями на область в памяти. Поэтому скопировав мапу в новую переменную, и удалив из нее элементы, это отобразиться также и на новой мапе.



Из этого так же выплывает и то, что мапы, как и срезы, при передаче в качестве аргумента функции передаются по ссылке. Это значит что любые изменения внутри тела функции также изменят саму мапу.

Структуры

Go можно назвать объектно-ориентированным языком, хоть в нем и нету привычных для такого семейства языков классов. Вместо них в Go используются структуры, которые умеют в себе хранить набор разных типов данных.

Если вы не знакомы с концепцией ООП (Объектно Ориентированным Программированием), в конце урока я поделюсь с вами ссылкой на статью. В следующих частях мы будем разбирать как в Go реализуются принципы объектно ориентированной парадигмы, поэтому желательно иметь некое представление.

Структура имеет возможность хранить в себе множество различных типов данных и может описывать какую-либо сущность из реального мира, ее характеристики (поля) и поведение (методы).

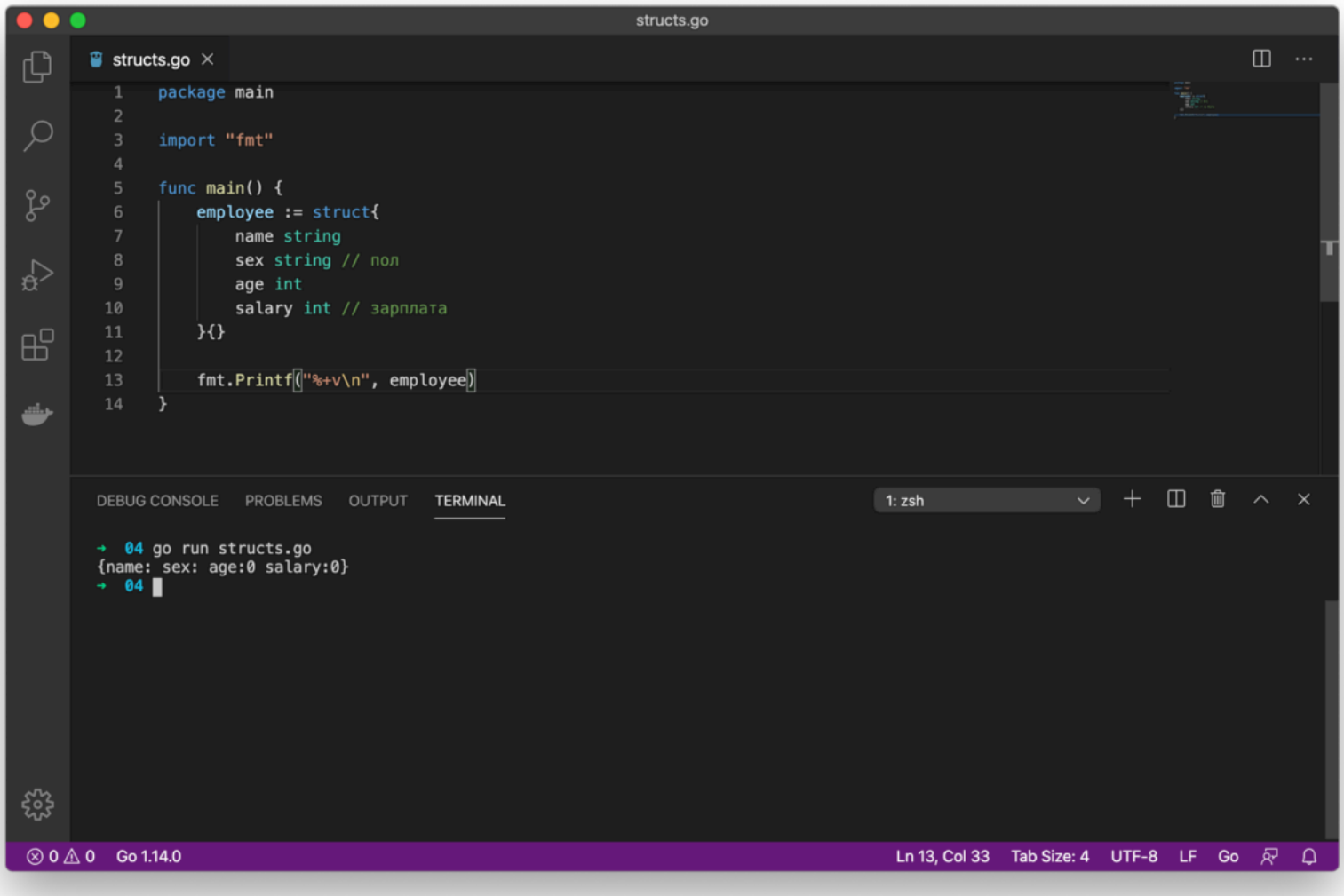
Давайте представим что мы разрабатываем программу для учета сотрудников фирмы. У каждого сотрудника есть имя, пол, возраст, должность и зарплата. Чтобы изобразить подобного рода сущность, ни переменные, ни массивы со срезами нам не подойдут. Тут то на помощь и приходят структуры.

Самый базовый синтаксис объявление структур имеет следующий синтаксис

`varName := struct{}{}`.

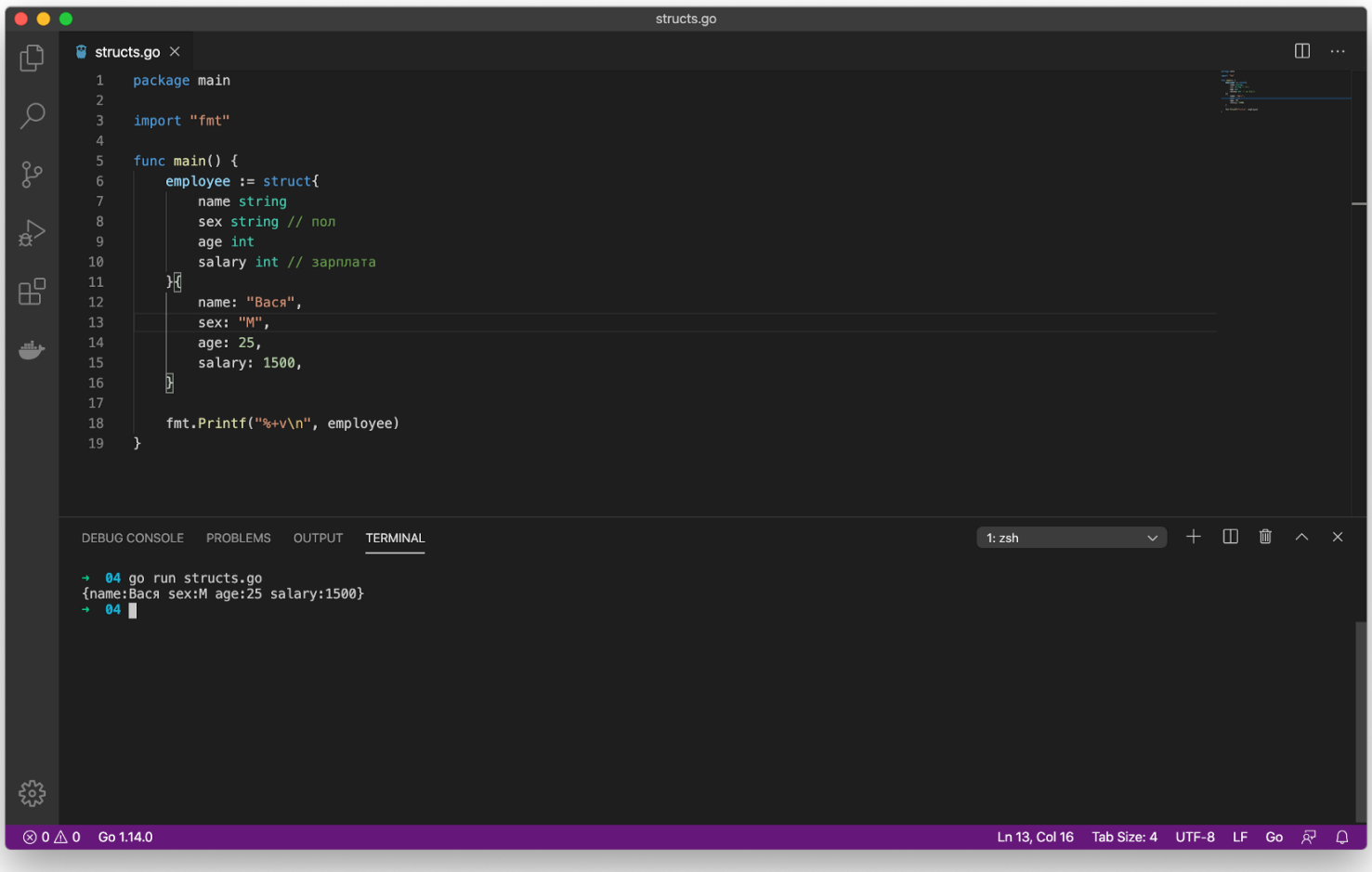
Как вы можете заметить, после ключевого слова `struct` идет 2 пары фигурных скобок. В первой паре необходимо описать саму структуру, ее поля. Во второй паре скобок необходимо уже записать в эти поля значения.

Давайте создадим свою первую структуру, которая описывает сущность сотрудника.



В теле структуры мы объявили имена полей (параметров сущности) и их типы.

Вторые скобки мы оставили пустыми. Как вы можете увидеть в консоле, у переменной `employee` теперь есть поля, но все они заполнены нулевыми значениями.



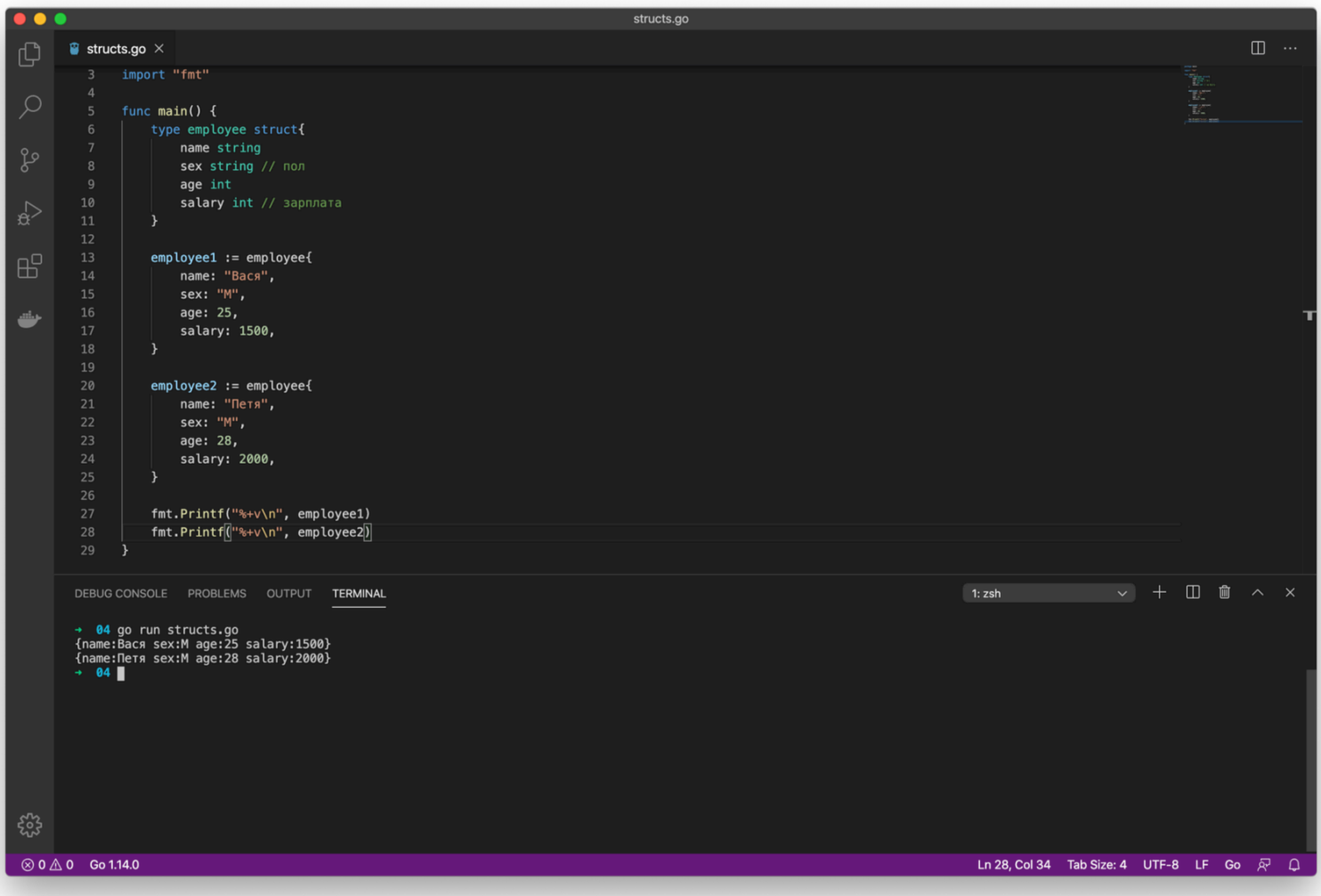
Теперь же мы присвоили значения полям во вторых скобках, и наша переменная `employee` при выводе в консоль отображает это.

Поздравляю, мы создали свою первую структуру! Но это только начало, так что поехали дальше.

Такое объявление структуры как в примере выше на практике используется достаточно редко. Дело в том, что зачастую мы хотим один раз описать структуру, а дальше создавать в нашей программе неограниченное количество ее экземпляров.

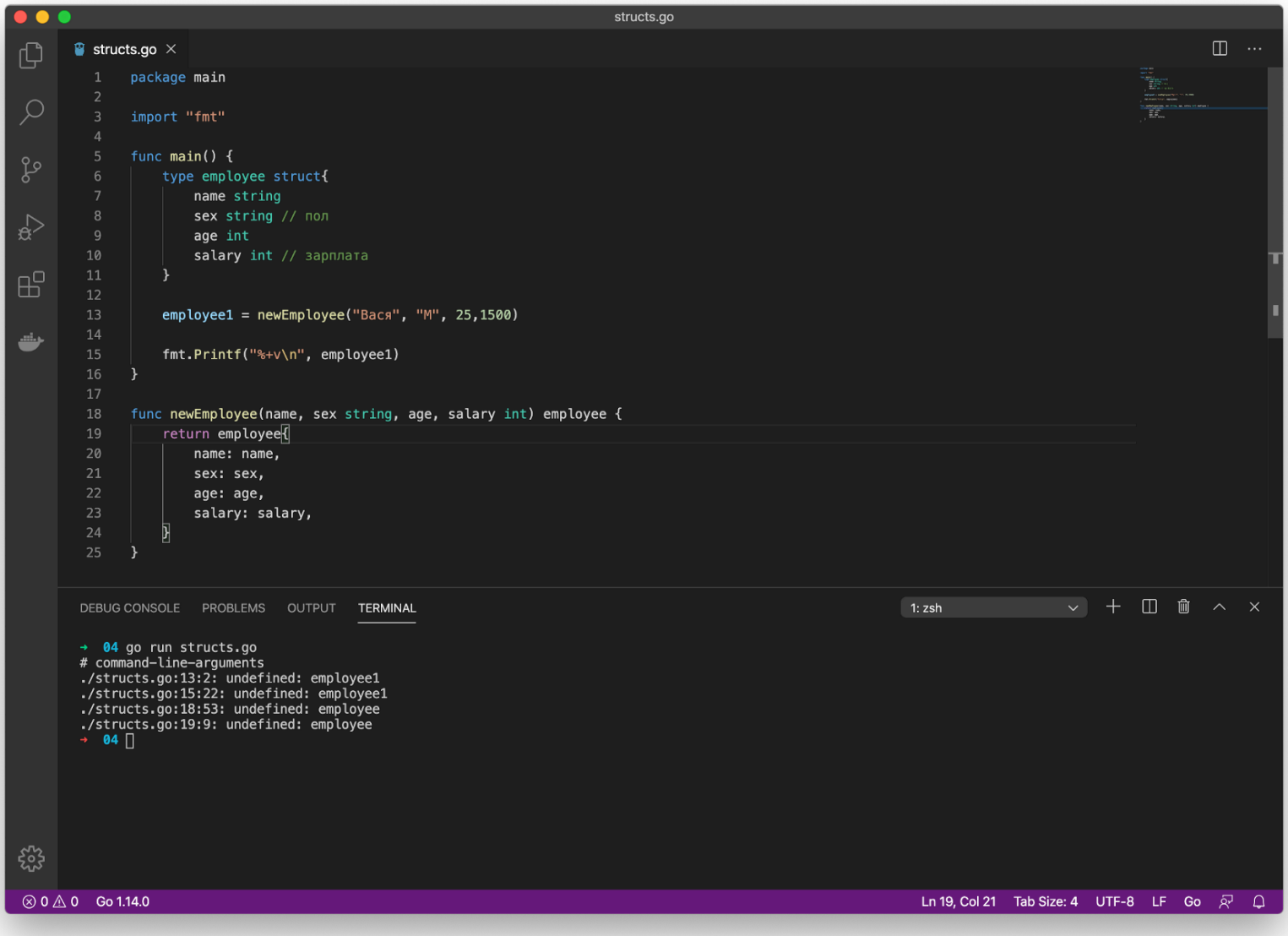
Для этого в Go существует ключевое слово `type` .

Если мы хотим описать структуру, при этом не создавая новой переменной, то нам стоит воспользоваться следующей конструкцией объявления `type Name struct{}` .

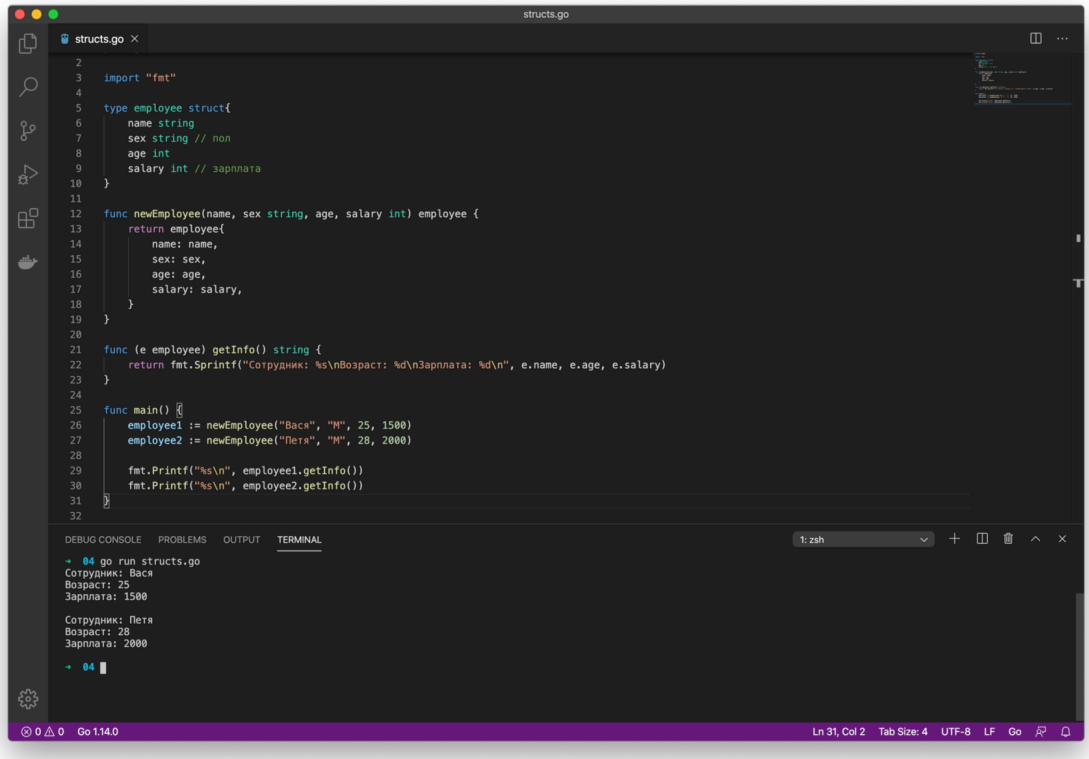


В этом примере мы создали новую структуру `employee`и описали в ней сущность сотрудника. Далее мы создали 2 *экземпляра (или объекта)* этой структуры, и присвоили полям разные значения.

Также, хочу обратить внимание, что мы объявили структуру внутри функции `main()`. Это означает, что за границами этой функции мы не сможем создать экземпляр структуры, потому что она находиться только в области видимости `main()`.



Как вы можете увидеть, код выше не скомпилируется, потому что в области видимости функции `newEmployee()` не доступна структура `employee`. Давайте исправим это, вынеся нашу структуру на верхний уровень кода, из функции `main()`.

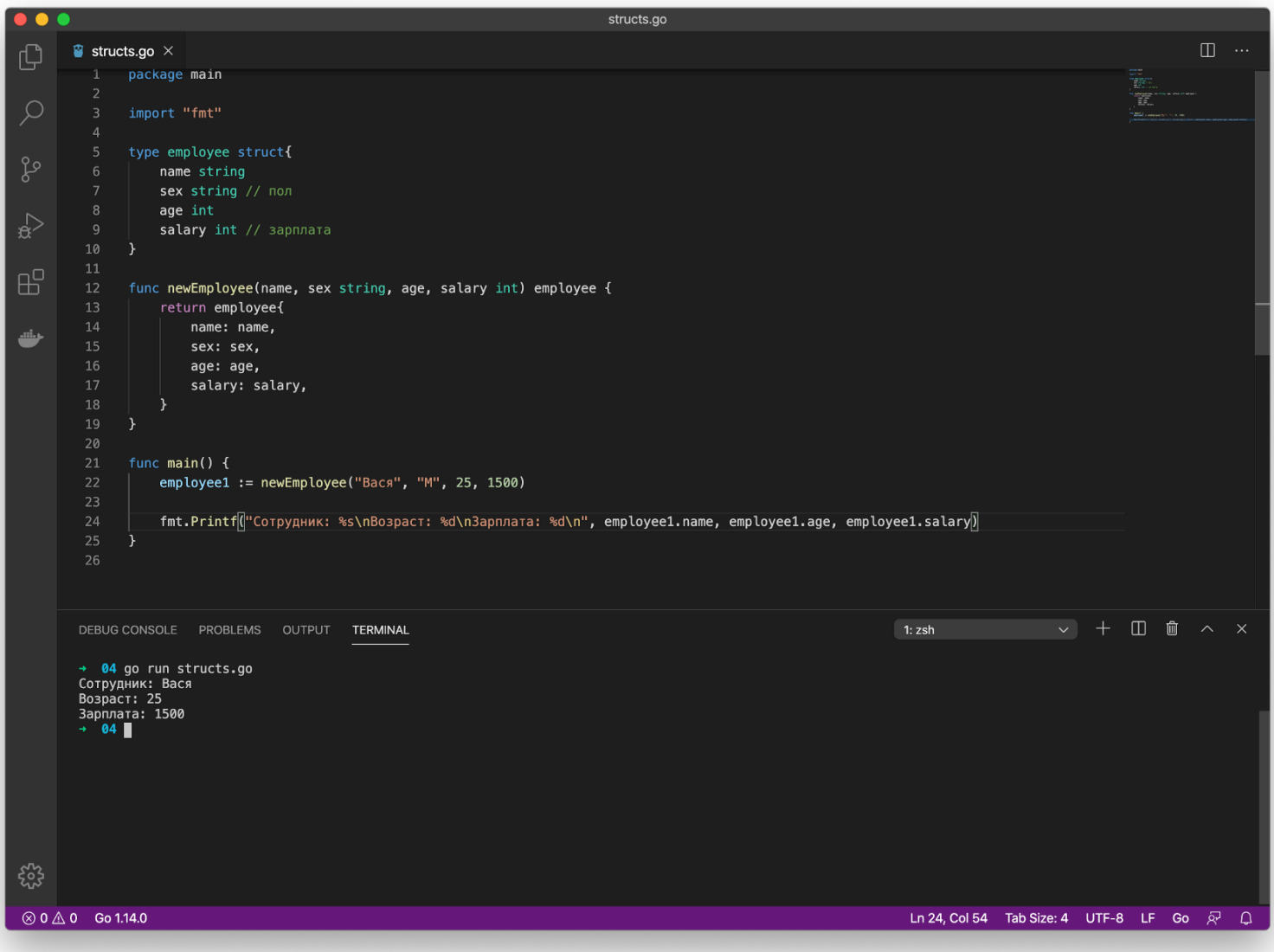


Зачастую структуры объявляются вне тела функций, чтобы можно было создать экземпляр в любом месте программы.

Теперь наша программа работает корректно, и мы можем создавать сколько угодно экземпляров нашей структуры в любом месте, а не только в `main()` .

Функция `newEmployee` принимает в качестве аргументов параметры структуры и создает новый экземпляр. Такие функции называются ***конструкторами*** или ***инициализаторами*** и зачастую записываются как `newНазваниеСтруктуры()` и возвращают новый объект самой структуры.

Конструкторы пришли к нам из мира ООП, и по своей сути, не являются обязательной частью программы на Go. Однако, такой подход можно назвать хорошей практикой, и вы точно столкнетесь с такой конструкций в большинстве проектов.



К отдельным полям объекта можно обращаться по имени поля после точки, как на примере выше.

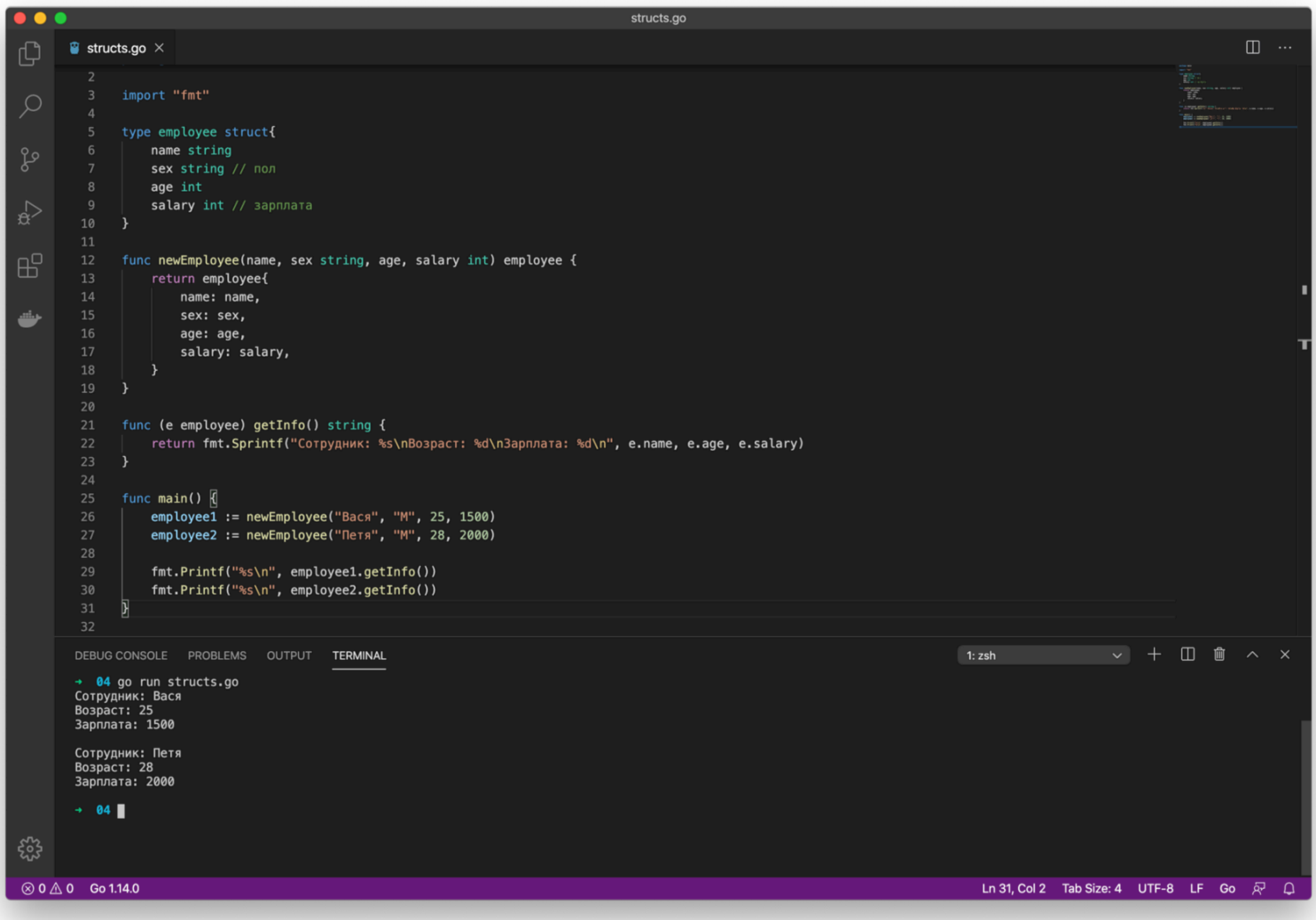
Методы структур

Как мы уже говорили, структура может иметь параметры (поля) и поведение (методы).

Методы – такие же функции, но которые имеют доступ к полям конкретного объекта и доступны, по аналогии с полями, у самого объекта при обращении через точку.

В последне примере мы выводим отформатированную информацию по сотруднику, обращаясь к каждому полю. Однако если в нашей программе будет 1000 сотрудников, будет неудобно писать `fmt.Printf()` для каждого из них.

Давайте объявим метод структуры, которая будет отдавать отформатированную информацию по каждому объекту.



Объявление методов структур имеет немного различный синтаксис с функциями и выглядит он следующим образом:

```
func (s structName) methodName(arg1 type, arg2 type ...) returnType {}
```

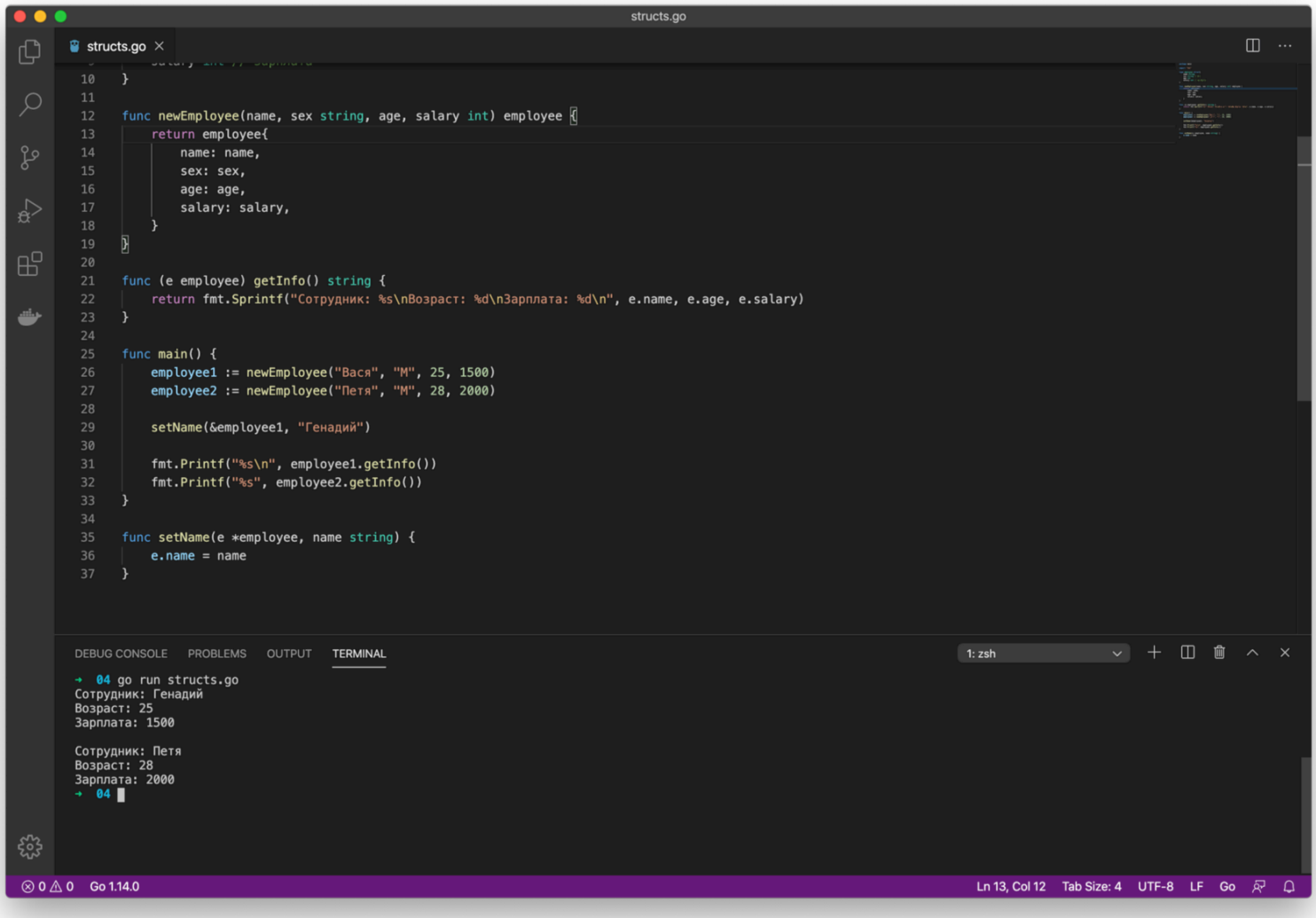
Где:

- `structName` – имя структуры для которой объявляется метод
- `s` – так называемый *ресивер* (или *получатель,* но я больше люблю использовать английскую терминологию). Вы можете называть его как угодно, но хорошей практикой будет использовать 1-2 символа, которые являются первыми символами названия структуры.
- Дальше все аналогично объявлению обычной функции.

Как вы можете увидеть, внутри тела метода обращение к полям происходит через ресивер.

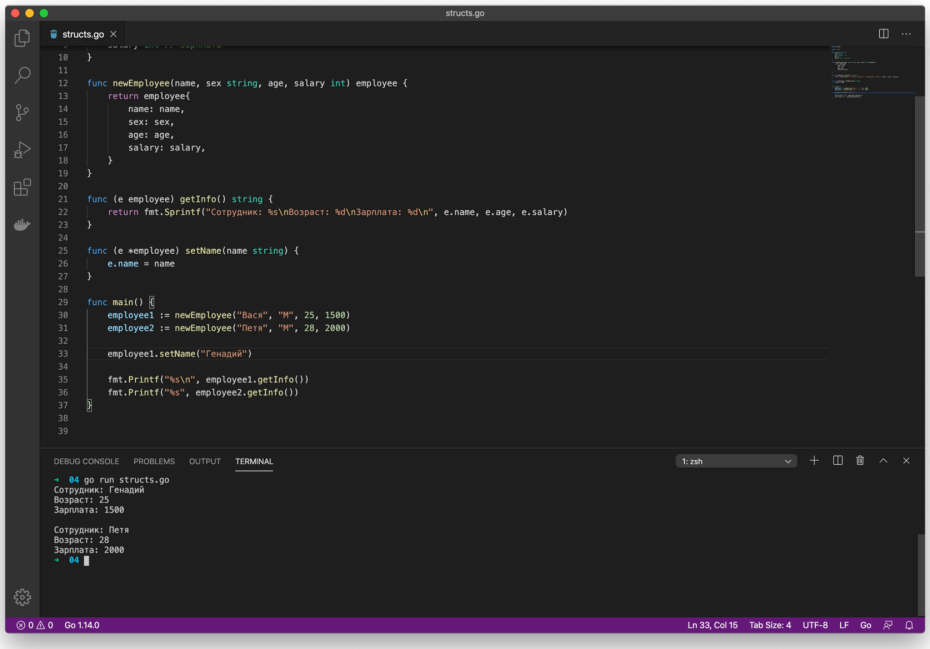
Указатель на структуру

Объекты структур также могут быть указателями. В таком случае, передавая наш объект в другие функции по ссылке, мы сможем менять значения внутри самого объекта, по аналогии со срезами и мапами.



Передав в `setName()` объект структуры по ссылке, мы получили возможность изменять значение полей самого объекта внутри функции.

Указатель в роли ресивера

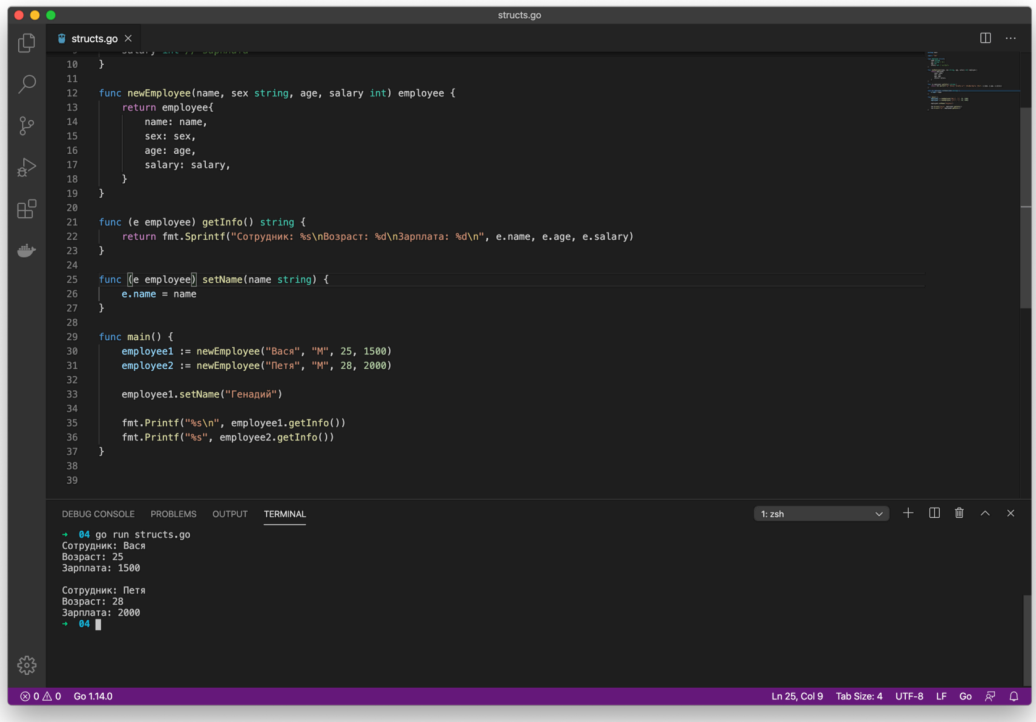


А теперь давайте рассмотрим отдельный тип методов, который в качестве ресивера принимает указатель, что позволяет внутри метода изменять сами поля объекта.

Теперь `setName()` является методом структуры `employee`, а в качестве ресивера принимает указатель `(e *employee)`.

Методы, которые принимают в качестве ресивера значение (без звездочки), работают внутри тела метода с копией объекта. По этому любые изменения на этой копии не скажутся на самом объекте.

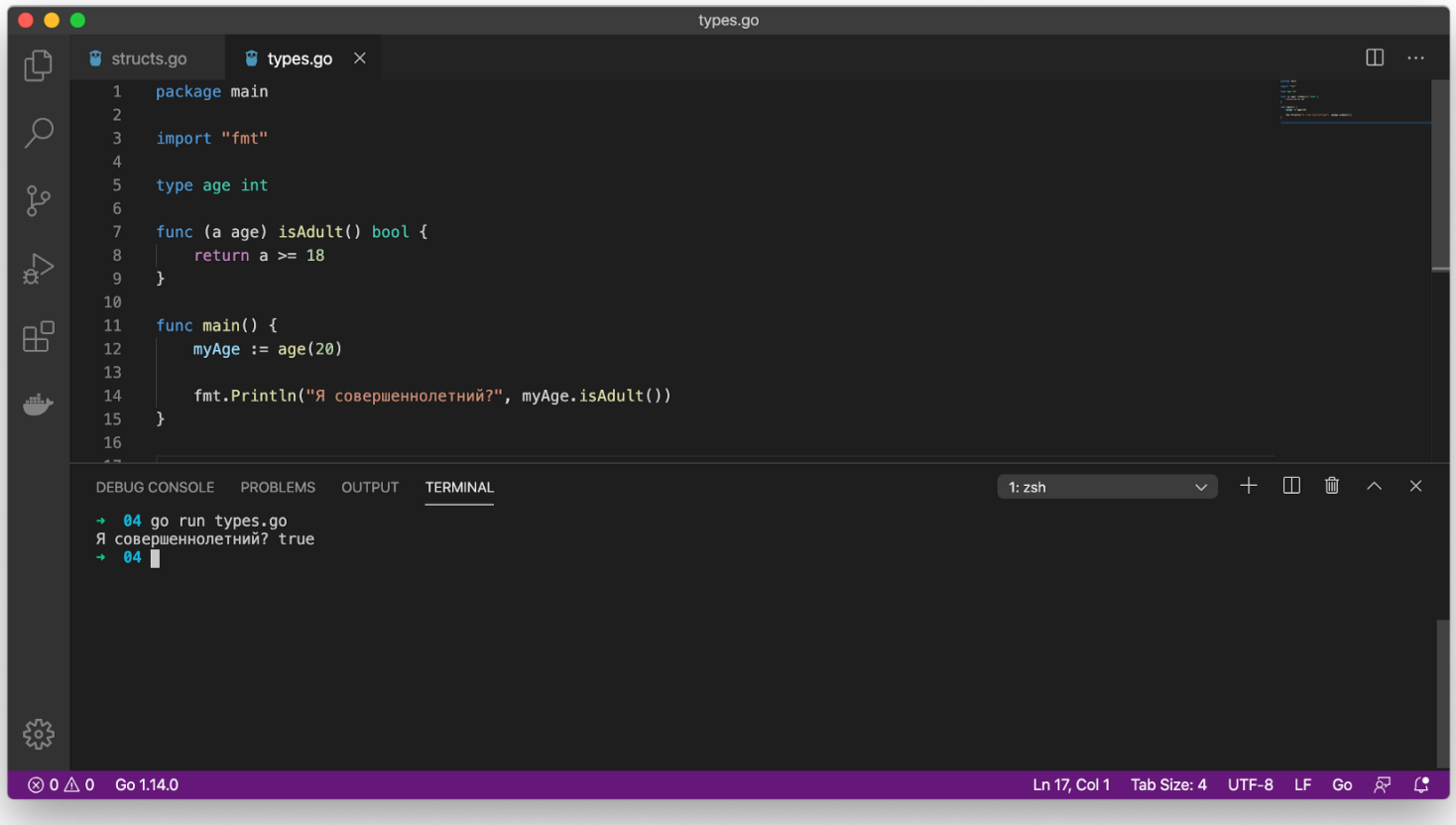
Как вы можете увидеть, поменяв ресивер с указателя на значение, наш метод `setName()` перестал работать как было задумано изначально.



Кастомные типы

Раз уж мы разобрали синтаксис объявления структур с помощью `type` , предлагаю напоследок познакомится с концепцией кастомных типов в Go.

Принцип и синтаксис аналогичный объявлению структуры.



Мы объявляем название нового типа и определяем тип данных этого нового типа.

Для своих типов мы также может объявлять методы с ресивером.

В основном свои кастомные типы используются для того, чтобы вешать на различные базовые типы свою логику, как в нашем случае с `age` .

Наш тип – обычный `int`, но у которого есть метод проверки возраста на совершеннолетие.

В стандартной библиотеке Go можно найти много примеров использования кастомных типов.

Давайте резюмируем.

В этом уроке мы рассмотрели мапы и структуры. Мапы используются для хранения данных в формате ключ:значение, а структуры для описания более сложные типов данных, с множеством полей и всевозможными методами.

Мапы также являются указателями, пустая мапа равна `nil` .

Go можно назвать объектно-ориентированным языком т.к он реализует все 3 принципа данной парадигмы (наследование, инкапсуляция и полиморфизм). Однако реализация ООП в отличается от более классических ООП языков типа C++/Java/C# в которых есть классы.

По аналогии со структурами, в Go можно объявлять кастомные типы данных, которые основаны на других типах, и могут иметь свои собственные методы.

Домашнее задание

Как я и обещал, рекомендую вам почитать материалы по ООП, т.к в в следующем уроке мы затронем эту тему более подробно.

Поиграйтесь с мапами, подумайте для каких сценариев стоит ее использовать.

Попробуйте записать значение в *не инициализированную мапу*. Что с этого получиться? Как вы думаете, почему так происходит?

Найдите материал про *анонимные поля* и *вложенные структуры*.

Подумайте, как вложенность структур реализовывает один из 4-ох основных принципов в Go – *наследование*?

Вспомните программу для подсчета площади круга из предыдущих уроков. Как бы вы ее переписали, используя структуры и методы?

В стандартной библиотеке Go можно найти примеры кастомных структур. Изучите более детально пакет time, найдите там кастомные типы, подумайте, зачем они там реализованы таким способом.

Раздел 05:

Интерфейсы. ООП.

Приветствую вас в 5-ом разделе. Надеюсь вы успешно осваиваете материал и без проблем разобрались с мапами и структурами с прошлого урока.

В предыдущем разделе мы немножко затронули тему ООП в Go. И в правду, Go можно считать объектно-ориентированным, хоть и эта парадигма в нем реализована без привычных для таких языков классов.

Сегодня мы с вами поговорим про интерфейсы и почему это важная часть для достижения объектно-ориентированной парадигмы в Go.

Для разбора этой темы я решил реализовать небольшой практичный модуль для нашего приложения, которое поможет также закрепить темы прошлого урока.

Интересно? Поехали!

Интерфейсы

В прошлом разделе мы с вами разобрали структуры и кастомные типы, и говорили что у структуры (или типа) можно определить методы (поведение).

А при чем тут интерфейсы? Я люблю давать следующее определение:

Интерфейс – это абстрактный тип, который описывает поведение, но не реализовывает его.

Интерфейсы описывают абстракцию (обобщают) поведение других типов. С помощью обобщения интерфейсы позволяют писать более гибкие и адаптируемые функции, не привязанные к деталям одной конкретной реализации.

И в правду, все что делает интерфейс – это описывает поведение какой-либо сущности. Он не реализовывает, а лишь определяет набор методов, какие сущность данного типа должна реализовать.

Структуры и типы могут реализовывать (или имплементировать) некий интерфейс. Тип ***соответствует*** (удовлетворяет) интерфейсу, если он обладает всеми методами, которые требует интерфейс.

Скорее всего вы пока мало что поняли, поэтому давайте разбирать интерфейсы на примерах.

Пишем собственное хранилище

Допустим мы разрабатываем очень простую систему хранения данных для учета сотрудников. Нам нужно уметь сохранять, получать и удалять информацию из этого хранилища по нашим, уже знакомым с предыдущего урока, сотрудникам.

Давайте определим интерфейс `storage`, который описывает поведение хранилища данных, а также сущность сотрудника.

```
1   package main
2
3   type employee struct {
4       id     int
5       name   string
6       age    string
7       salary int
8   }
9
10  type storage interface {
11      insert(e employee) error
12      get(id int) (employee, error)
13      delete(id int) error
14  }
15
16  func main() {
17
18  }
19
```

В теле интерфейса мы описали поведение (определили методы) нашего `storage` . Он должен уметь делать вставку, получение и удаление сотрудников.

На данном этапе нас вообще не волнует как это будет реализовано: будем ли мы хранить эти данные в мапе, записывать в файл или использовать стороннюю базу данных. Нам лишь важно чтобы наше хранилище умело делать эти 3 вещи, а как – это уже вопрос реализации.

Теперь давайте создадим структуру, которая будет реализовывать данный интерфейс.

В данном примере мы описали структуру `memoryStorage` , которая имеет лишь одно поле типа`map[int]employee` и реализовывает все 3 метода нашего интерфейса.

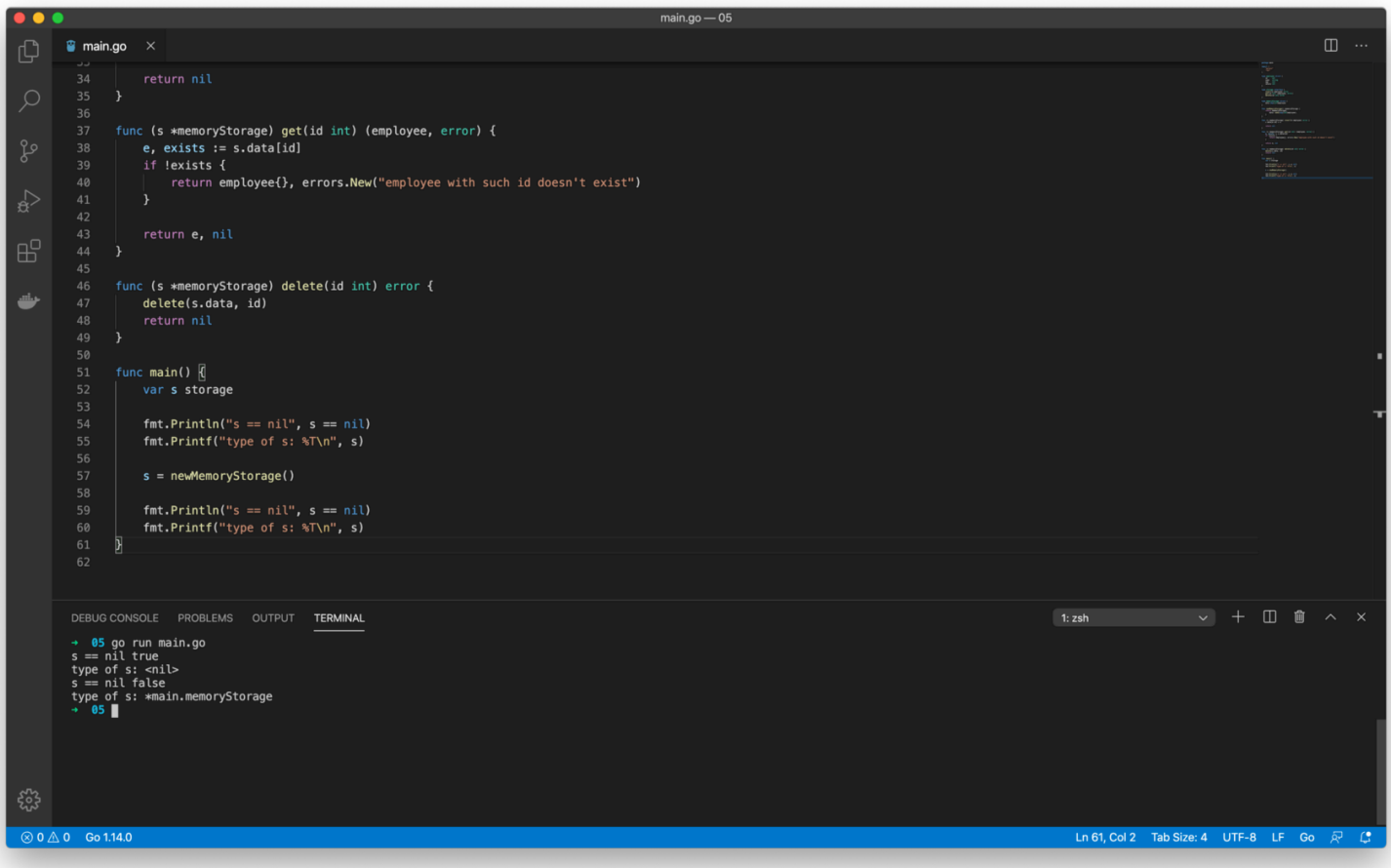
Эта структура будет реализовывать интерфейс `storage` посредством хранения всей информации в памяти (поскольку мы храним данные в мапе, вся информация будет находиться в оперативной памяти компьютера).

При записи и удалении элементов из мапы у нас не может возникать ошибок, поэтому мы просто возвращаем `nil` в качестве ошибки.

```go
type employee struct {
    id     int
    name   string
    age    int
    salary int
}

type storage interface {
    insert(e employee) error
    get(id int) (employee, error)
    delete(id int) error
}

type memoryStorage struct {
    data map[int]employee
}

func newMemoryStorage() *memoryStorage {
    return &memoryStorage{
        data: make(map[int]employee),
    }
}

func (s *memoryStorage) insert(e employee) error {
    s.data[e.id] = e

    return nil
}

func (s *memoryStorage) get(id int) (employee, error) {
    e, exists := s.data[id]
    if !exists {
        return employee{}, errors.New("employee with such id doesn't exist")
    }

    return e, nil
}

func (s *memoryStorage) delete(id int) error {
    delete(s.data, id)
    return nil
}
```

Также мы описали конструктор, который возвращает указатель на структуру и инициализирует мапу при создании объекта. Если этого не сделать, мы получим ошибку при попытке записи в не инициализированную мапу.

За уникальное поле сотрудника мы взяли `id`, по этому это будет служить нашим ключем в мапе.



В примере выше много чего происходит, по этому давайте по порядку.

Мы с вами создали переменную типа `storage` . Поскольку интерфейсы, это такие же кастомные типы, как и структуры, мы можем создавать переменные данного типа.

После создание пустой переменной типа нашего интерфейса, ее значение равно `nil` и **ее тип** также равен `nil` . Сначала это может показаться вам немного сложным для понимания.

Концептуально значение интерфейсного типа, или просто ***значение интерфейса***, имеет два компонента – конкретный тип и значение этого типа. Они называются ***динамическим типом*** и ***динамическим значением*** интерфейса.

Почему динамическим?

В Go мы можем присваивать в переменную интерфейса другие типы, которые соответствуют данному интерфейсу. Как только значение интерфейса становиться не `nil` , его *динамический тип* становится типом нового значения.

Именно поэтому мы без проблемы присвоили в переменную `s` типа `storage` объект типа `*memoryStorage` – этот тип **соответствует** интерфейсу, потому что **обладает всеми необходимыми методами.**

И поэтому после инициализации интерфейса, его значение и тип равны `nil` , а после присвоения нашей структуры, значение не равно `nil`, а тип равен `*memoryStorage` .

Давайте немного изменим нашу программу.

Метод `delete()` мы переименовали на `remove()` и теперь наш компилятор ругается, а редактор подчеркивает ошибку красным.

Как только мы изменили имя метода, наша структура `memoryStorage` уже не реализовывает интерфейс `storage`. По этому присвоение данного объекта в переменную типа интерфейса стало невозможным.

```
func (s *memoryStorage) get(id int) (employee, error) {
    e, exists := s.data[id]
    if !exists {
        return employee{}, errors.New("employee with such id doesn't exist")
    }

    return e, nil
}

func (s *memoryStorage) remove(id int) error {
    delete(s.data, id)
    return nil
}

func main() {
    var s storage

    fmt.Println("s == nil", s == nil)
    fmt.Printf("type of s: %T\n", s)

    s = newMemoryStorage()

    fmt.Println("s == nil", s == nil)
    fmt.Printf("type of s: %T\n", s)
}
```

```
EBUG CONSOLE  PROBLEMS 1  OUTPUT  TERMINAL
 05 go run main.go
command-line-arguments
/main.go:57:4: cannot use newMemoryStorage() (type *memoryStorage) as type storage in assignment:
        *memoryStorage does not implement storage (missing delete method)
 05
```

Давайте добавим еще один тип, который будет реализовывать наш интерфейс.

```go
type dumbStorage struct{}

func newDumbStorage() *dumbStorage {
    return &dumbStorage{}
}

func (s *dumbStorage) insert(e employee) error {
    fmt.Printf("вставка пользователя с id: %d прошла успешно\n", e.id)
    return nil
}

func (s *dumbStorage) get(id int) (employee, error) {
    e := employee{
        id: id,
    }

    return e, nil
}

func (s *dumbStorage) delete(id int) error {
    fmt.Printf("удаление пользователя с id: %d прошло успешно\n", id)
    return nil
}

func main() {
    var s storage

    fmt.Println("s == nil", s == nil)
    fmt.Printf("type of s: %T\n", s)
```

В идеале было бы круто добавить структуру, которая будет уметь работать со сторонним хранилищем (например Postgres или MongoDB). Однако работа со сторонними БД выходит далеко за рамки нашего урока, поэтому мы сделаем “глупое хранилище”.

В примере мы создали новый тип `dumbStorage` , который также реализовывает наш интерфейс, а в своей реализации ничего не сохраняет – лишь выводит на экран текст об удачных операциях.

Давайте теперь будем выводить значение и тип интерфейса в консоль.

Как вы можете увидеть, мы успешно можем переприсвоить интерфейсу значение другого типа, который также имплементирует этот интерфейс. При этом динамический тип и значение интерфейса изменятся, что вы можете увидеть на примере выше.

```go
74
75  func main() {
76      var s storage
77
78      fmt.Println("s:", s)
79      fmt.Printf("type of s: %T\n\n", s)
80
81      s = newMemoryStorage()
82
83      fmt.Println("s:", s)
84      fmt.Printf("type of s: %T\n\n", s)
85
86      s = newDumbStorage()
87
88      fmt.Println("s:", s)
89      fmt.Printf("type of s: %T\n\n", s)
90
91      s = nil
92
93      fmt.Println("s:", s)
94      fmt.Printf("type of s: %T\n\n", s)
95  }
96
```

```
DEBUG CONSOLE  PROBLEMS  OUTPUT  TERMINAL

→  05 go run main.go
s: <nil>
type of s: <nil>

s: &{map[]}
type of s: *main.memoryStorage

s: &{}
type of s: *main.dumbStorage

s: <nil>
type of s: <nil>

→  05
```

```go
func main() {
    var s storage

    fmt.Println("s:", s)
    fmt.Printf("type of s: %T\n\n", s)

    s = newMemoryStorage()

    fmt.Println("s:", s)
    fmt.Printf("type of s: %T\n\n", s)

    s = newDumbStorage()

    fmt.Println("s:", s)
    fmt.Printf("type of s: %T\n\n", s)

    s = nil

    fmt.Println("s:", s)
    fmt.Printf("type of s: %T\n\n", s)

    s = 15
}
```

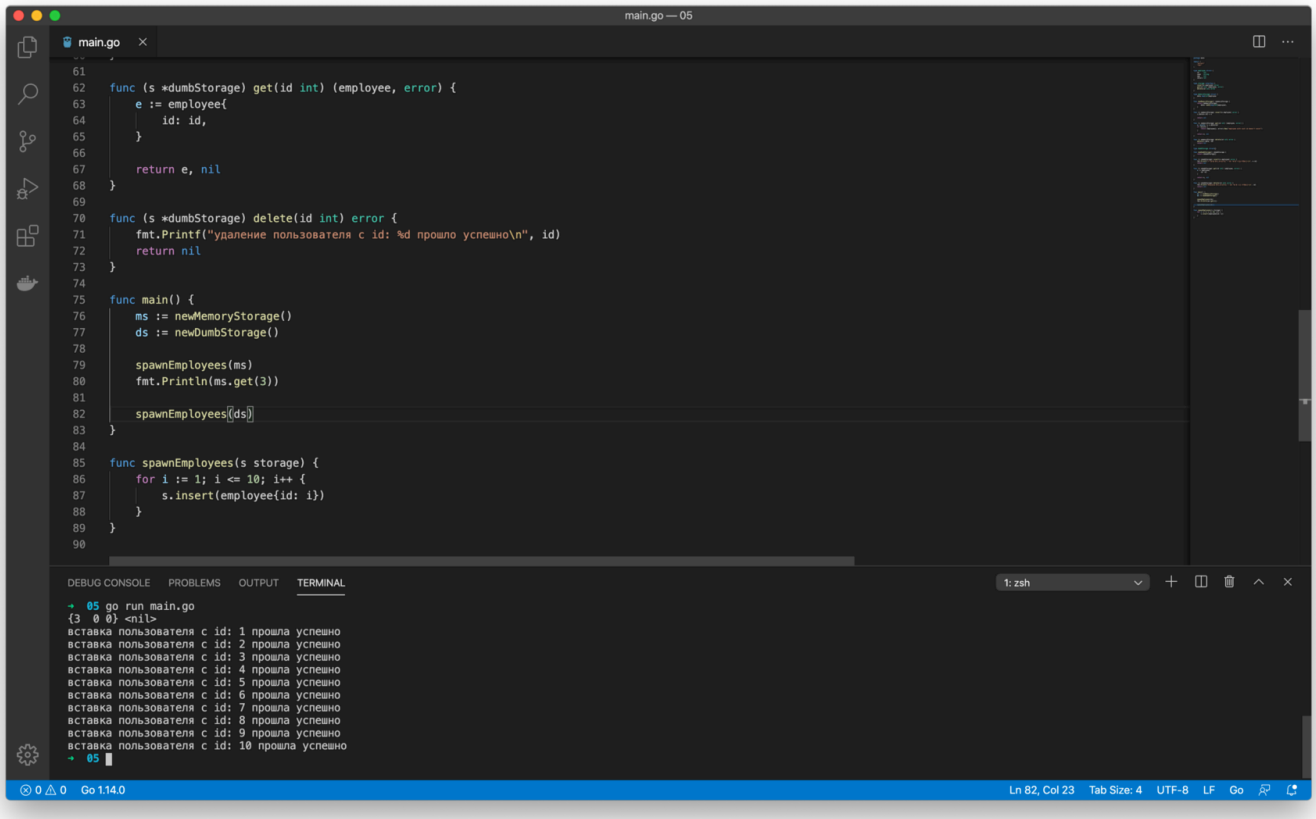
```
DEBUG CONSOLE  PROBLEMS 1  OUTPUT  TERMINAL

→ 05 go run main.go
s: <nil>
type of s: <nil>

s: &{map[]}
type of s: *main.memoryStorage

s: &{}
type of s: *main.dumbStorage

s: <nil>
type of s: <nil>

→ 05
```

Но если мы захотим присвоить нашему интерфейсу тип, который не удовлетворяет наш интерфейс, произойдет ошибка компиляции.

На примере вы можете увидеть, что при присвоении числа (тип `int`), редактор подчеркивает это действие красным.

Как мы уже говорили в самом начале, интерфейсы позволяют писать более гибкие и адаптируемые функции, не привязанные к деталям одной конкретной реализации.

Свобода замены одного типа другим, который соответствует тому же интерфейсу, называется ***взаимозаменяемостью*** (substitutability) и является отличительной особенностью объектно-ориентированного программирования.



К примеру, есть функция `spawnEmployees` которая принимает в качестве аргумента интерфейс. Это позволяет нам передавать в данную функцию разные типы, которые по разному реализовывают *одно и то же поведение*.

То чего мы достигли, создав 2 структуры, которые реализовывают один и тот же интерфейс, называется ***полиморфизмом***.

Полиморфизм – возможность объектов с одинаковой спецификацией иметь различную реализацию. Его смысл можно выразить фразой: «Один интерфейс, множество реализаций».

Полиморфизм – один из четырёх важнейших механизмов объектно-ориентированного программирования (наряду с абстракцией, инкапсуляцией и наследованием).

Пустые интерфейсы

Как мы уже говорили, интерфейс также может быть `nil` . Это интерфейс, который вообще не описывает никакого поведения.

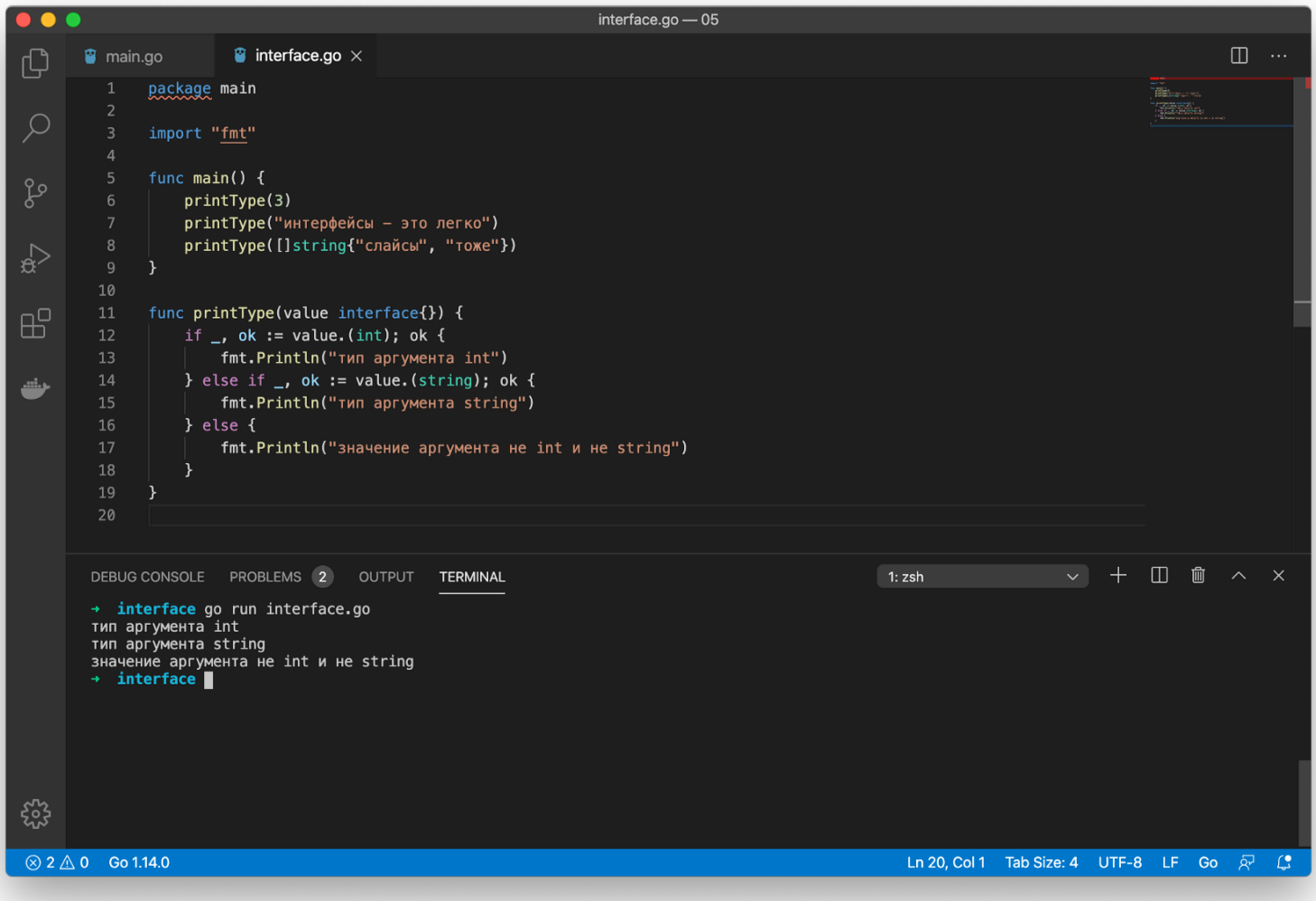
Тогда зачем нам это? Суть в том, что любой тип в программе на Go по умолчанию удовлетворяет пустой интерфейс.

Возьмем для примера уже знакомую нам функцию `fmt.Println()` .

```
270  // Println formats using the default formats for its operands and writes to standard output.
271  // Spaces are always added between operands and a newline is appended.
272  // It returns the number of bytes written and any write error encountered.
273  func Println(a ...interface{}) (n int, err error) {
274      return Fprintln(os.Stdout, a...)
275  }
276
```

Если посмотреть на ее реализацию, то мы можем увидеть что в качестве аргументов она принимает неограниченное количество пустых интерфейсов – именно по этому мы можем в нее передавать любые типы: числа, интерфейсы, мапы, срезы, структуры и тд.

Работая с пустыми интерфейсами, мы имеем возможность проверять интерфейс на его динамический тип. Давайте разберем следующий пример.



Наша функция принимает в качестве аргумента пустой интерфейс. В теле функции мы делаем серию проверок, проверяя это значение на конкретный тип, и выполняя в зависимости от этого разные действия.

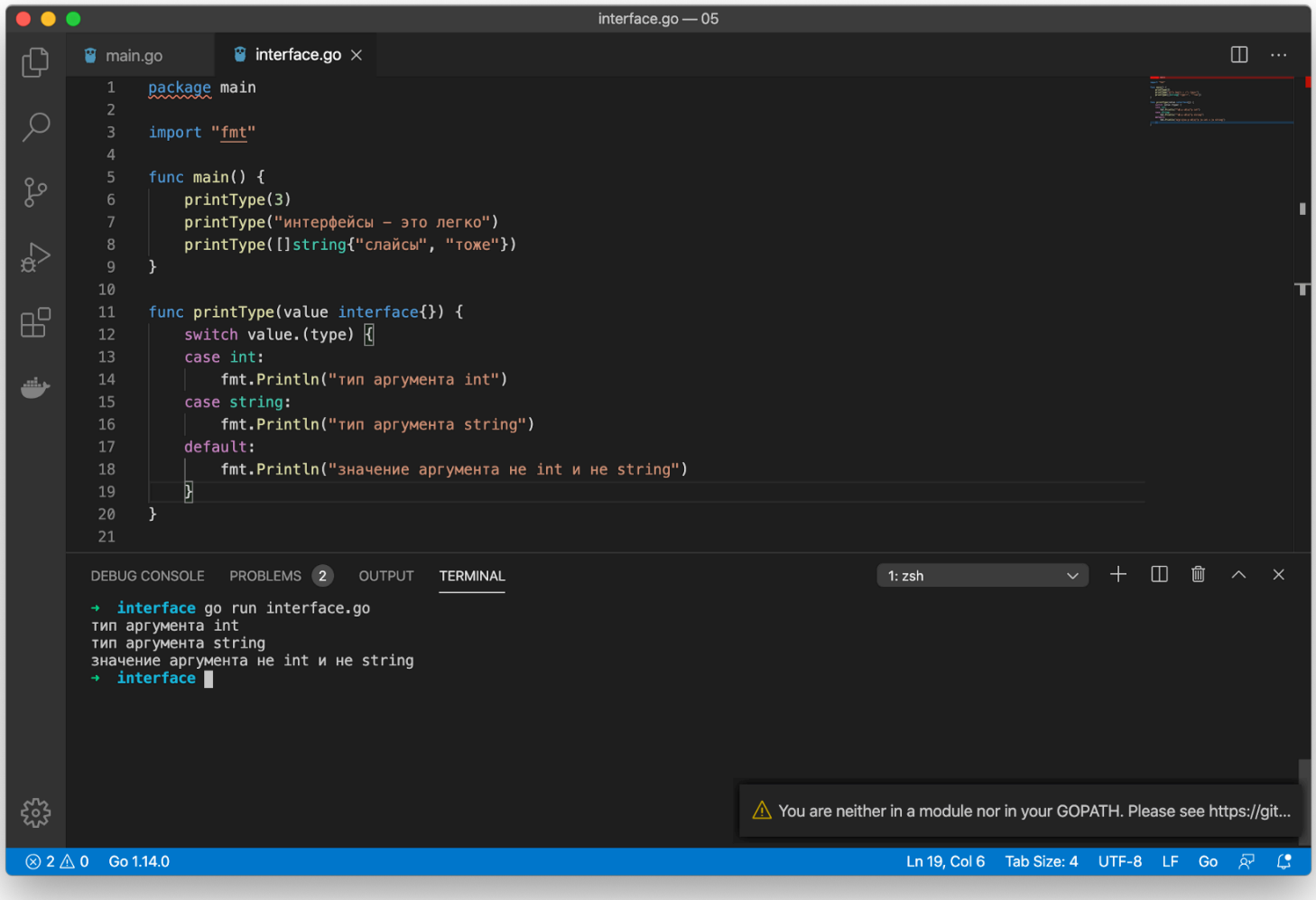
Для проверки типа используется следующая конструкция:

```
value, ok := variable.(type)
```

При проверке типа мы получаем 2 переменных: `bool`, который равен `true` если значение удовлетворяет данному типу и само значение, приведенное к нужному типу.

В таких случаях, при проверке большого количества возможных значений одной переменной, чаще всего используется еще один условный оператор `switch case` .

Давайте перепишем пример выше, используя эту конструкцию.



Как вы можете увидеть, использование конструкции `switch case` идеально подходит для ситуаций, когда нужно проверять тип интерфейса на несколько значений.

Давайте резюмируем.

Интерфейсы описывают абстракцию (обобщают) поведение других типов. С помощью обобщения интерфейсы позволяют писать более гибкие и адаптируемые функции, не привязанные к деталям одной конкретной реализации.

Структуры и типы могут реализовывать (или имплементировать) некий интерфейс. Тип ***соответствует*** (удовлетворяет) интерфейсу, если он обладает всеми методами, которые требует интерфейс.

Интерфейсы приближают Go к парадигме ООП, потому что реализовывают одни из ключевых его принципов: абстракцию и полиморфизм.

В Go есть понятие пустого интерфейса. Любой тип удовлетворяет пустому интерфейсу.

Домашнее задание

Поиграйтесь с конструкцией `switch case`. В каких случаях она будет предпочтительней `if`?

Поищите информацию про внедрение интерфейсов, зачем это нужно и как реализовывается.

Создайте интерфейс геометрической фигуры. Каким поведением она должна обладать? Подсчитывать площадь фигуры? Какие типы смогут удовлетворить данному интерфейсу? Круг, квадрат, прямоугольник? Как вы их реализуете?

Раздел 06:

Go Modules. Пакеты.

Рад видеть вас в 6-ом разделе. Вы уже на половине пути в изучении основных концепций этого языка! Надеюсь, материал дается вам легко и вы успешно справляетесь с выполнением домашних заданий.

В прошлой главе мы изучили тему интерфейсов и ООП.

Сегодня мы с вами будем разбирать тему модулей и пакетов в Go, а также поговорим, как пакеты позволяют достигать одного из главных принципов в ООП – инкапсуляции.

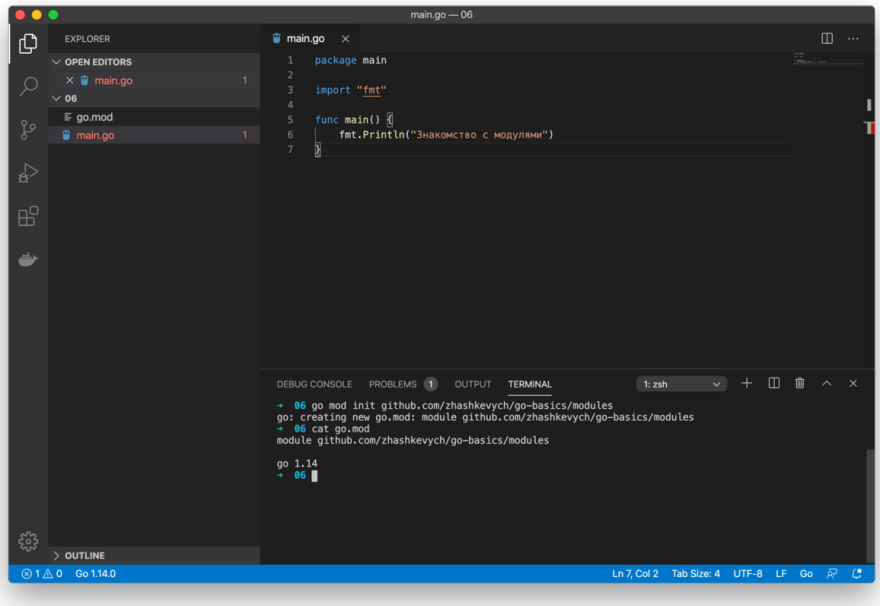
Система модулей

Начиная с версии 1.11, Go внедрил новую систему управления зависимости Go Modules, которая помогает явно хранить информацию о версиях зависимостей и делает процесс их управления проще.

Модуль – это коллекция пакетов записанная в файле `go.mod` , который должен находится в корне проекта. Этот файл также определяет путь для этого модуля, который используется внутри проекта.

Создаем свой модуль

Давайте создадим в пустой директории файл `main.go` с базовой структурой. После этого, находясь в этой же директории запустим команду



`go mod init <название вашего модуля>`.

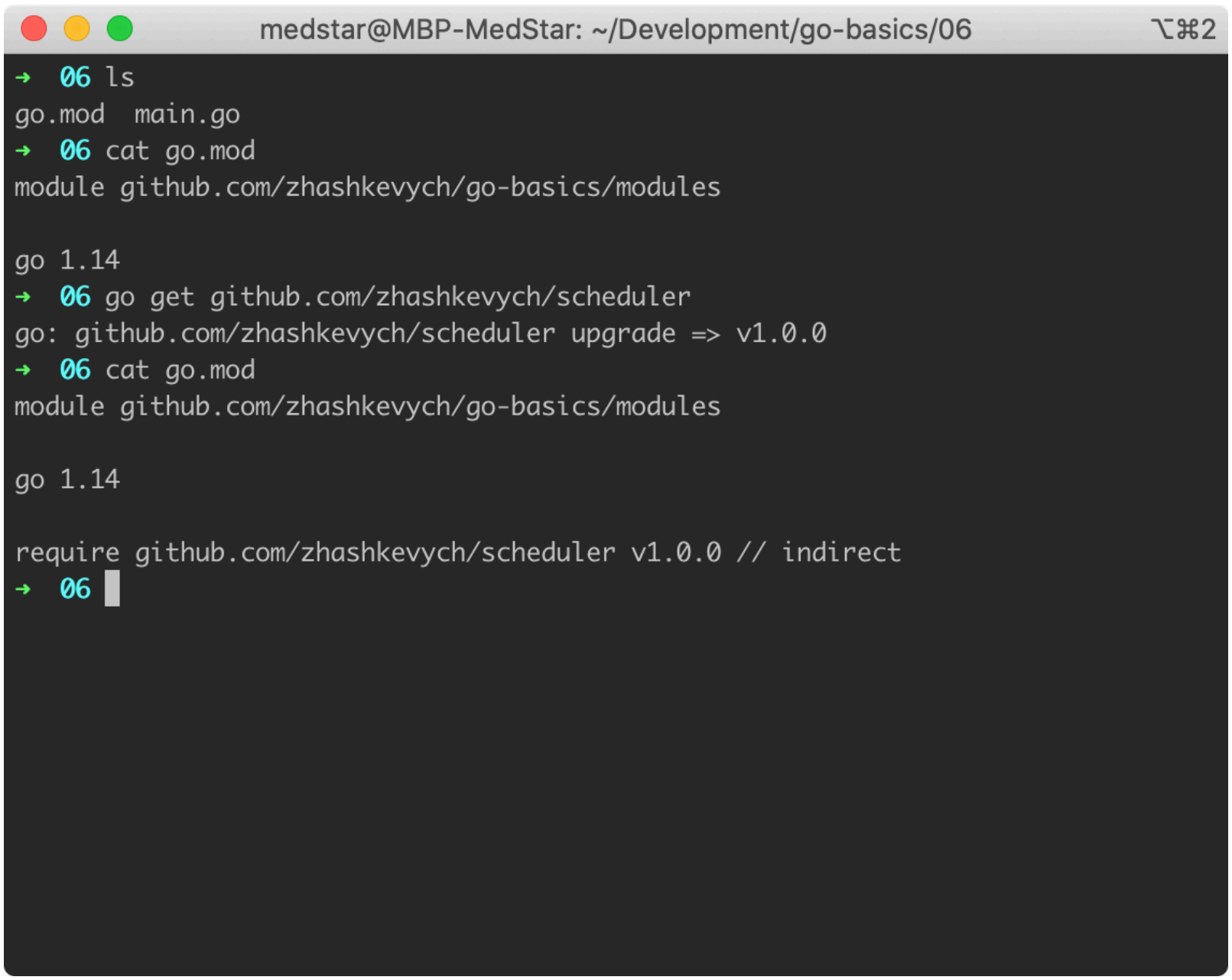
Как вы можете увидеть, после запуска этой команды в нашей директории появился файл `go.mod` , в котором записано название модуля и наша версия Go.

Поздравляю, мы создали свой первый модуль!

Импортируем зависимости

Главной мотивацией создания Go Modules являлось улучшение процесса работы с сторонним кодом, написаного сторонними разработчиками.

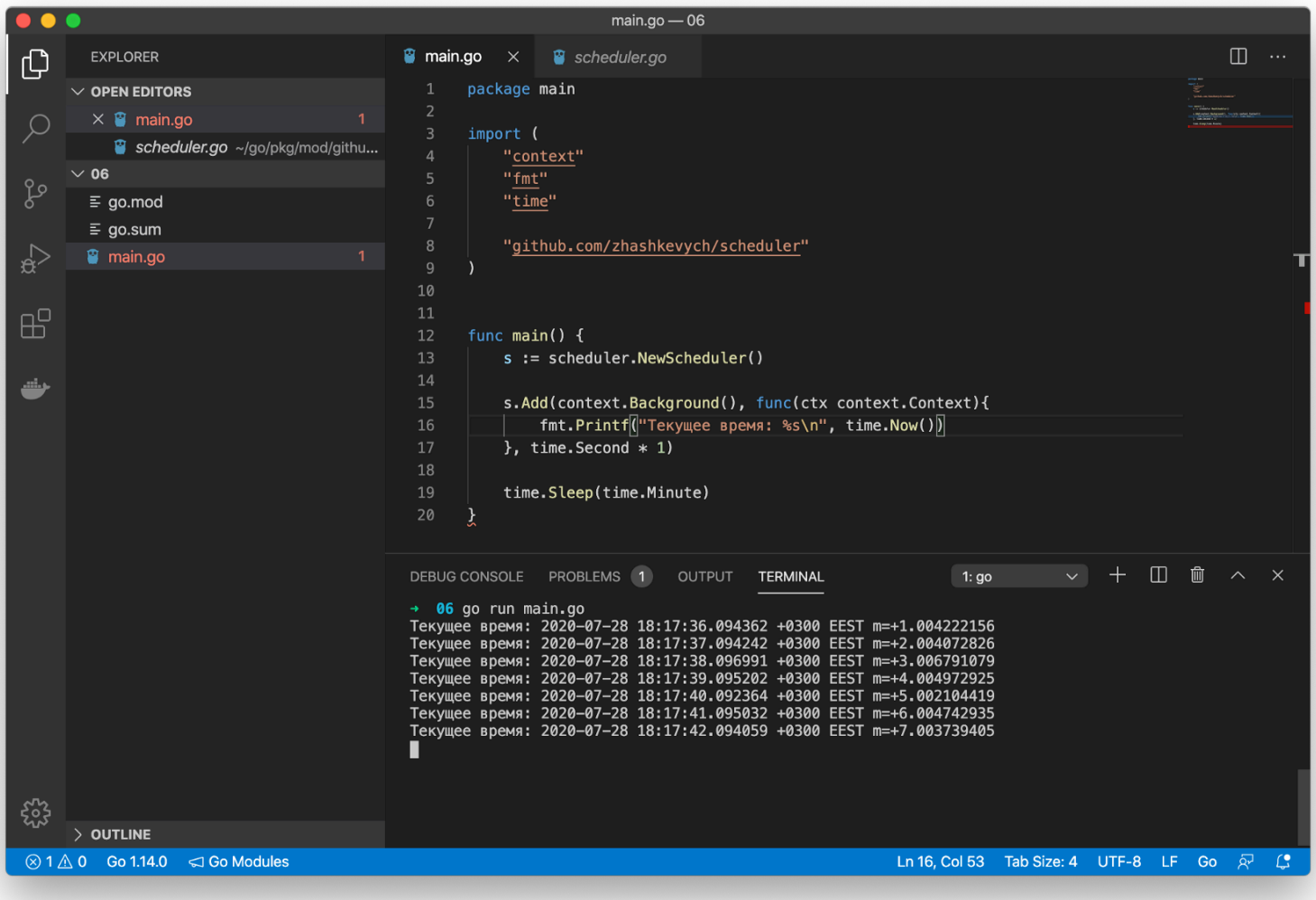
Давайте импортируем в наш проект стороннюю библиотеку с GitHub. Для импорта различных модулей и библиотек используется команда `go get` .



Запустив команду `go get github.com/zhashkevych/scheduler` мы импортировали в наш проект библиотеку, которая лежит на GitHub по этому адресу.

После импорта зависимости вы можете увидеть что файл `go.mod` также изменился, в нем появилась строка, которая добавляет в проект новую зависимость конкретной версии.

Теперь мы можем использовать только что импортированную библиотеку в нашем проекте.



В данном примере мы создали функцию которая выводит текущее время на экран и задали ей интервал выполнения в 1 секунду, используя библиотеку которую мы недавно импортировали в наш проект.

Если вы захотите скомпилировать свой проект на другом компьютере, для импорта всех необходимых зависимостей нужно будет лишь запустить команду `go mod download`.

Работа с пакетами

Сегодня программа скромного размера может содержать десятки тысяч функций. Тем не менее ее автору нужно думать лишь о некоторых из них, а разрабатывать – и того меньше, потому что подавляющее большинство функций написано другими программистами и доступно для повторного использования в виде ***пакетов.***

Цель любой системы пакетов – сделать дизайн и поддержку больших программ практичными путем группирования связанных функций в модули, которые легко понимать и изменять независимо от других пакетов программы.
Такая ***модульность*** позволяет совместно использовать пакеты разными проектами, распространять их в пределах организации или делать доступными всему миру.

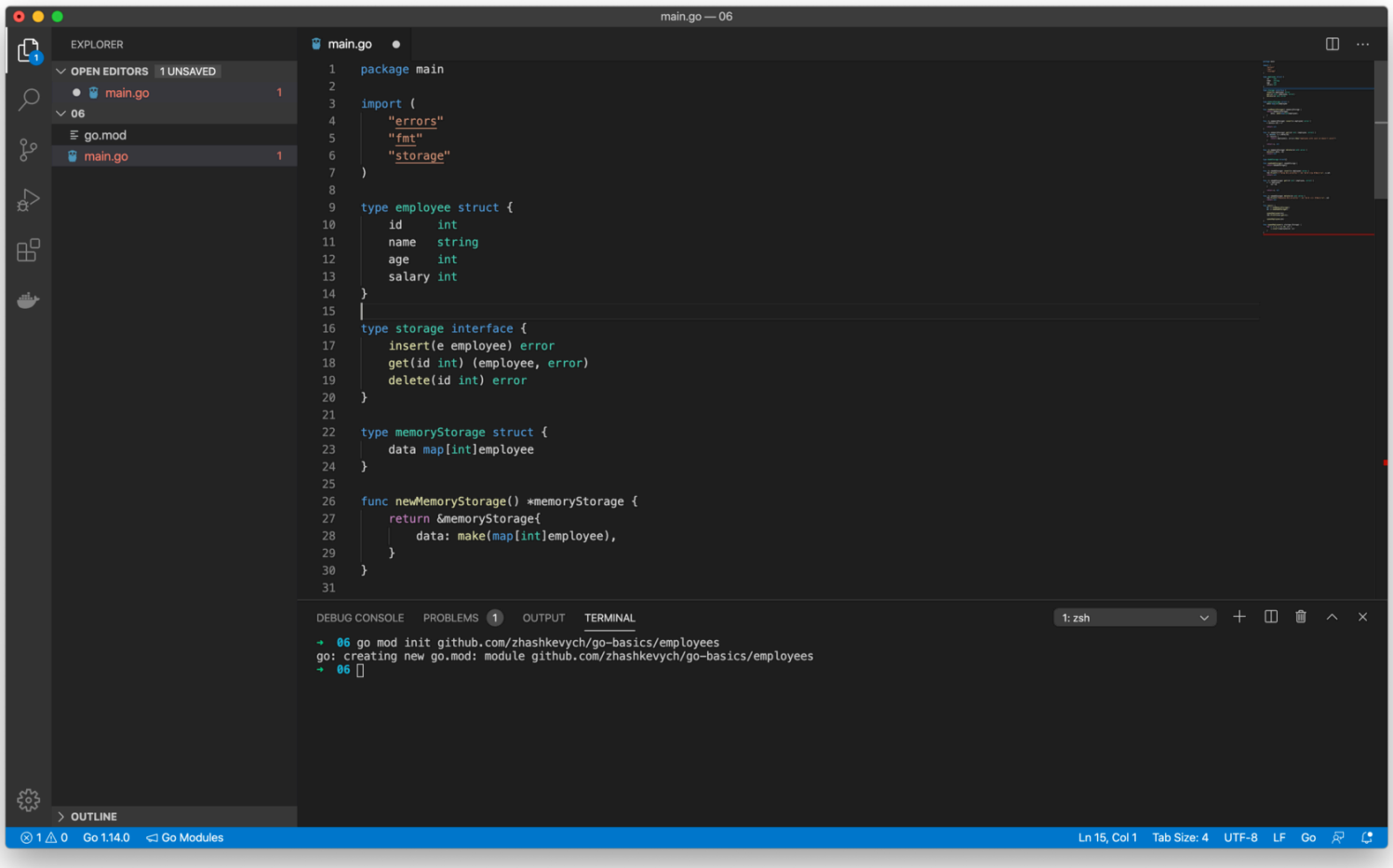
Каждый пакет определяет уникальное пространство имен, охватывающее все его идентификаторы. Каждое имя связано с конкретным пакетом, что позволяет нам выбирать короткие, ясные имена для наиболее часто используемых типов, функций и так далее, не создавая при этом конфликтов с другими частями программы.

Пакеты также обеспечивают ***инкапсуляцию***, управляя тем, какие имена видны, или экспортируемы, вне пакета.

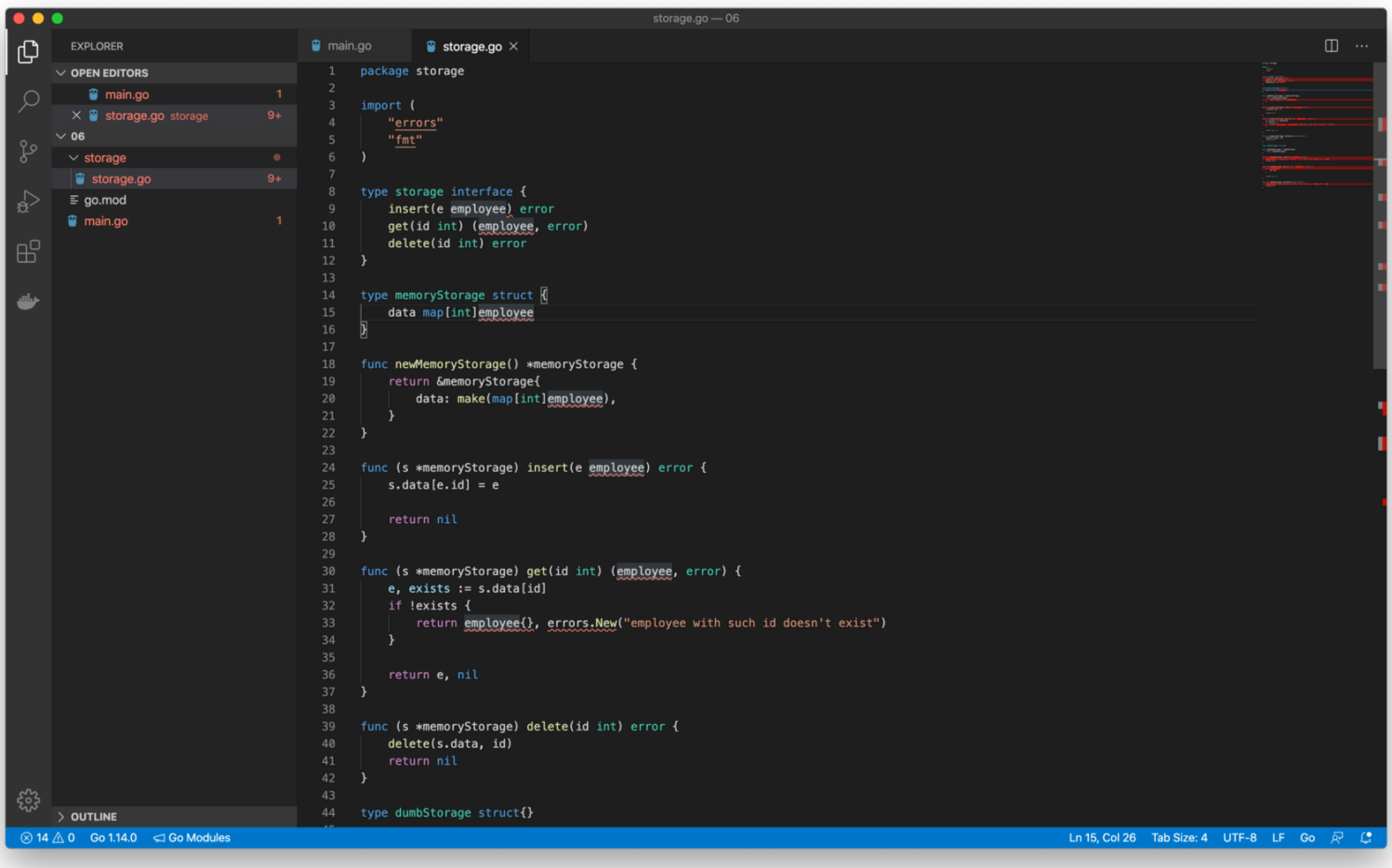
Инкапсуляция – свойство языка программирования, позволяющее пользователю не задумываться о сложности реализации используемого программного компонента (что у него внутри?), а взаимодействовать с ним посредством предоставляемого интерфейса (публичных методов и членов), а также объединить и защитить жизненно важные для компонента данные. При этом пользователю предоставляется только спецификация (интерфейс) объекта.

Инкапсуляция – один из четырёх важнейших механизмов объектно-ориентированного программирования (наряду с абстракцией, полиморфизмом и наследованием).

Давайте вернемся к нашему хранилищу. Создадим новый модуль внутри корневой папки.



А теперь давайте создадим пакет `storage`, в который вынесем весь код, связанный с нашим хранилищем. Создадим папку `storage` , в которой создадим файл `storage.go` . В самом файле названия пакета запишем `package storage` .



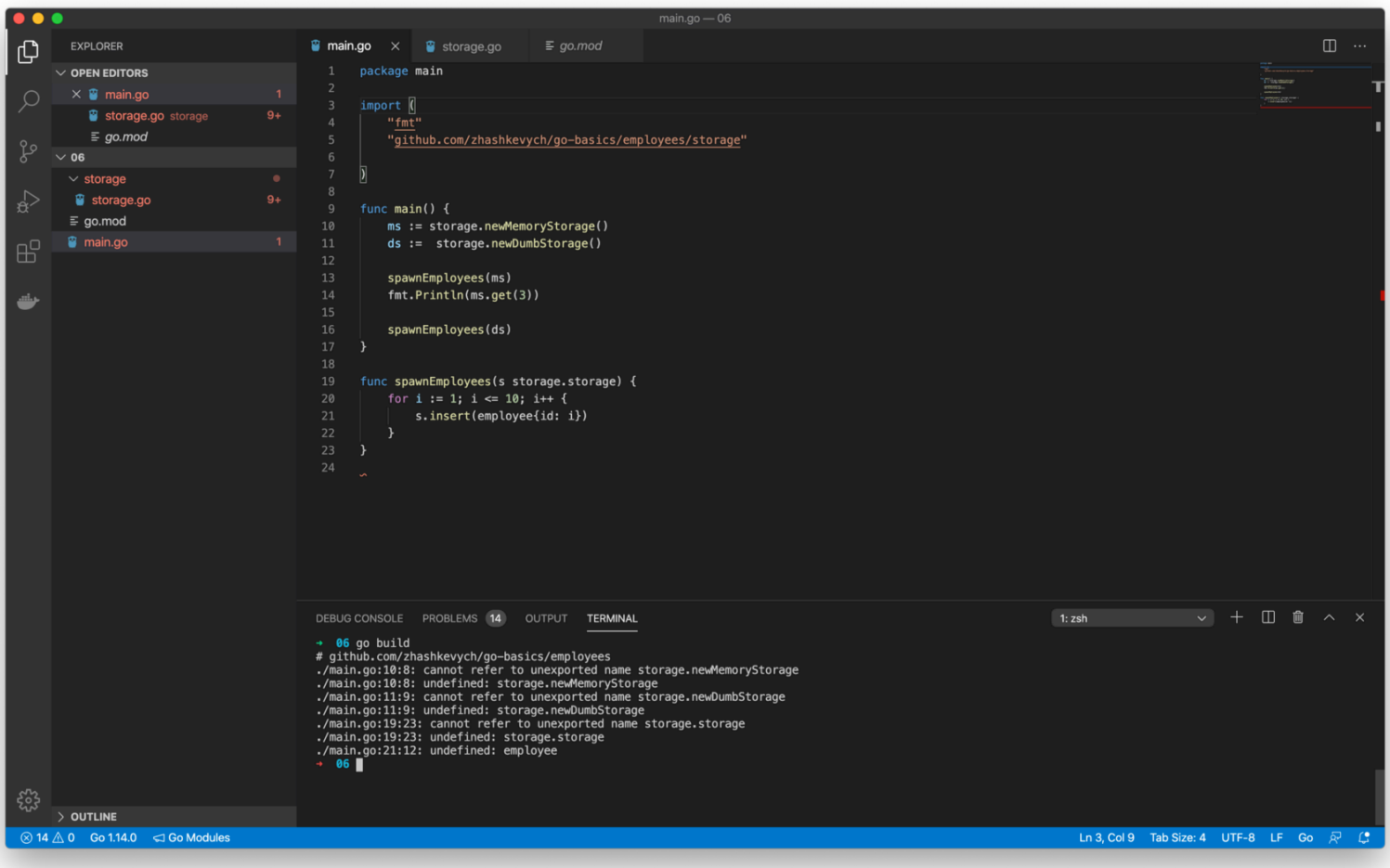
Как видите, наш редактор сразу подчеркнул красным все места, где мы обращаемся к структуре `employee` . Происходит это потому что данная структура лежит вне данного пакета и является ***не экспортируемой***.

В Go все переменные, константы, функции, типы и их методы бывают двух типов: *экспортируемые* и *не экспортируемые*. Достигается это с помощью регистра первой буквы из названия сущности.

Если у сущности название начинается с маленькой буквы, она доступна лишь в рамках того пакета в котором объявлена, т.е является не экспортируемой.

Если с большой – она доступна в других местах программы при импорте пакета, в котором она объявлена, т.е является экспортируемой.

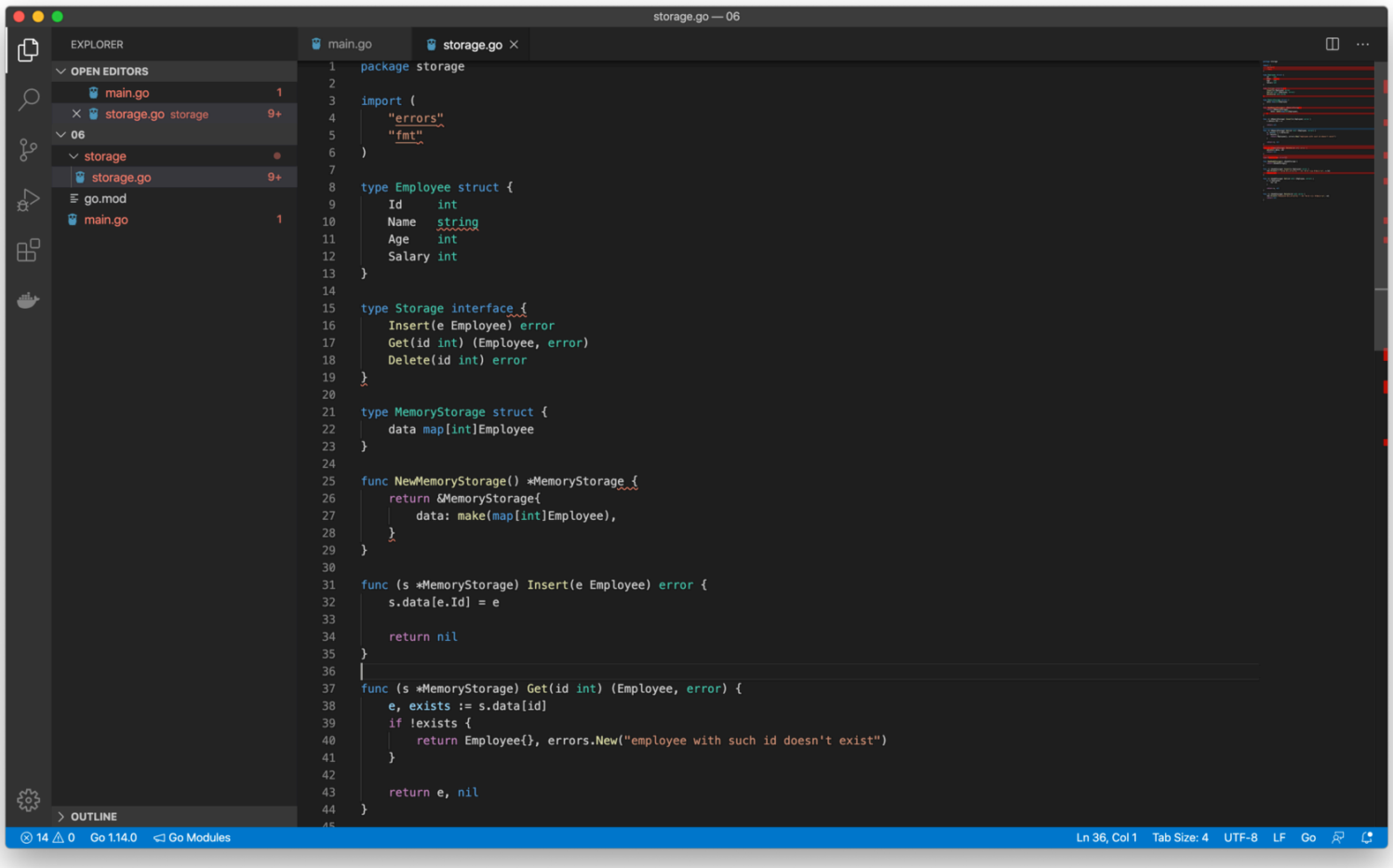
Давайте перенесем структуру `employee` , переименовав ее с большой буквы, в наш пакет `storage` и импортируем его в `main.go` .



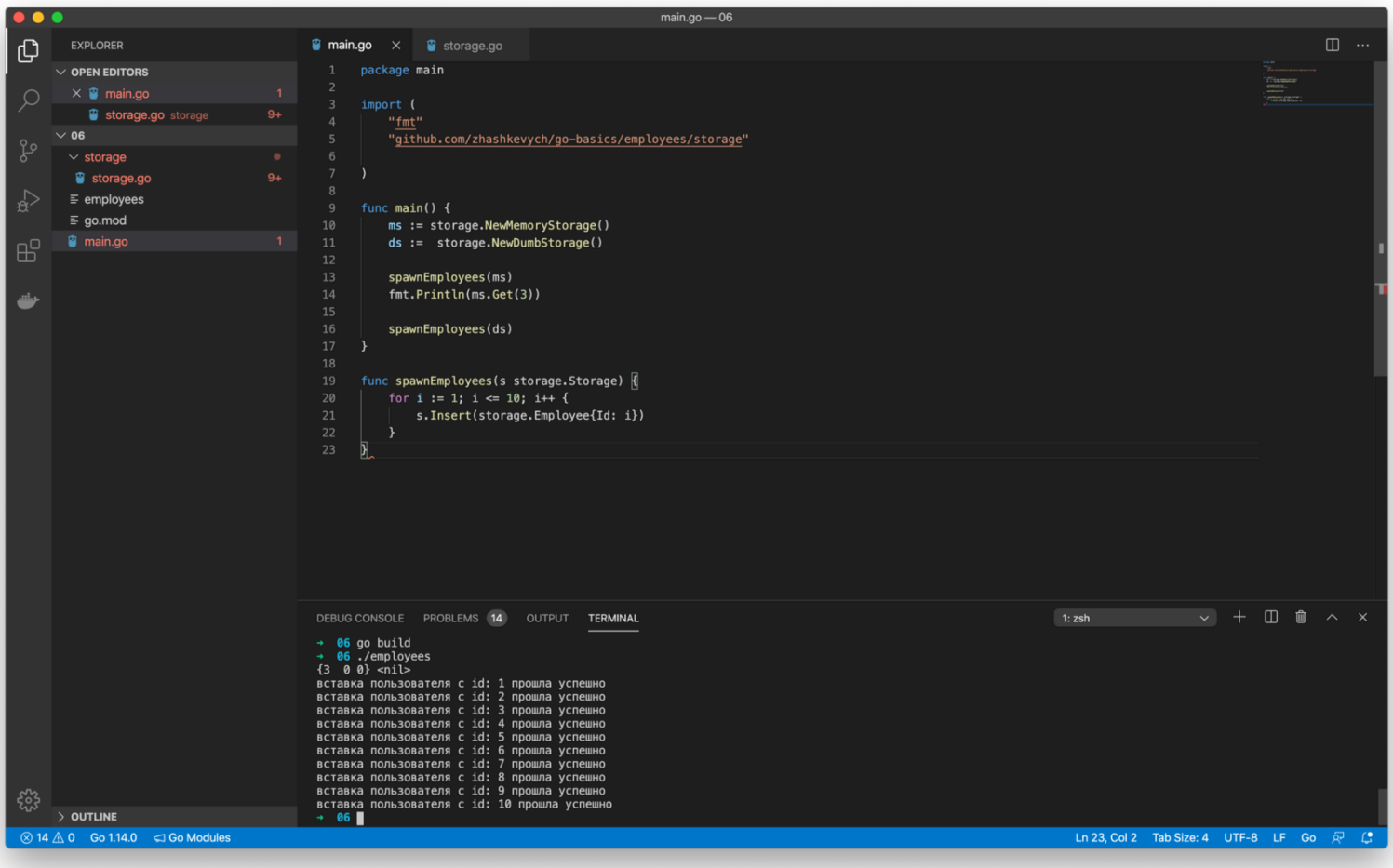
Как вы можете увидеть, мы импортировали наш новый пакет с помощью обычного `import` указав название нашего модуля и добавив название пакета в конце.

Теперь мы можем обращаться к экспортируемым типам и функциям из этого пакета, но вот ведь не задача – сейчас все типы, их поля и методы в

пакете `storage` объявлены с маленькой буквы и мы не можем к ним обратиться. Давайте исправим это.

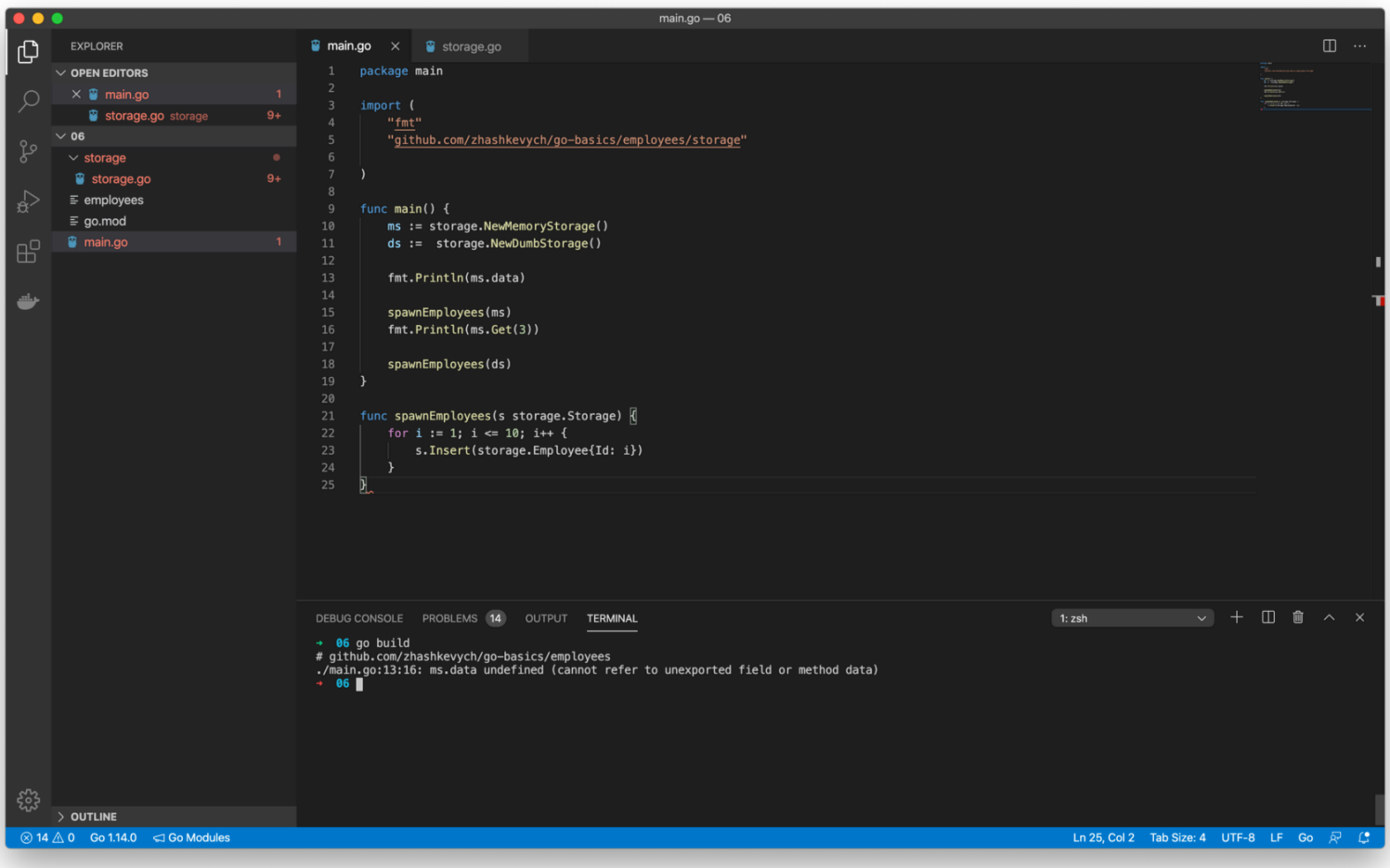


И теперь также изменим наш файл `main.go` .



Как видите, все успешно работает. Мы создали свой модуль, объявили в нем отдельный пакет и импортировали в пакет `main.go` . Также мы переименовали интерфейс, структуры, их поля и методы с большой буквы, после чего они стали доступны вне области видимости пакета `storage`.

А если мы захотим обратиться напрямую к полю `data` структуры `MemoryStorage` ?



У нас это не получится. Поскольку поле объявлено с маленькой буквы, то оно является не экспортируемым, о чем нам говорит компилятор при попытке компиляции.

Давайте подведем итоги.

Начиная с версии 1.11, в Go по умолчанию используется система управления зависимости Go Modules.

Для того чтобы создать свой модуль, используйте команду `go mod init <название вашего модуля>` .

Для импортирования сторонних зависимостей используйте команду `go get` .

Пакеты в Go помогают структурировать исходный код программ, а также помогают достичь *инкапсулиции*. Достигается это за счет верхнего и нижнего регистра записи всех переменных, констант, типов и их методов внутри пакета.

Все, что записано с маленькой буквы доступно лишь внутри пакета. Все что с большой – экспортируется и может быть доступно в других местах приложения при импорте данного пакета.

Домашнее задание

В примере выше мы использовали `go build` для компиляции программы в бинарный файл с последующим его запуском вместо уже привычного `go run` .

Почитайте про эти 2 команды, подумайте почему я выбрал `go build` для нашей программы с пакетами. Поиграйтесь сами с этими командами.

Создайте свой модуль и новый пакет в нем. Пускай это будет `shape` , в котором объявите интерфейс фигуры с методом подсчета площади. Реализуйте несколько структур, которые будут удовлетворять данному интерфейсу. Поэкспериментируйте сами с экспортируемыми и не экспортируемыми полями.

Раздел 07:

Конкурентность и Параллелизм.

Приветствую вас 7-ом разделе. Мы уже успели пройти большинство основных примитивов языка, а в предыдущей главе мы поговорили с вами про модули и пакеты. Сегодня же мы приблизились к теме конкурентности в Go.

Как вы можете помните из вводной части, одной из основных особенностей языка является встроенная поддержка конкурентного программирования.

Сейчас мы подробнее разберем эту тему. Поговорим про то, что такое конкурентность и параллелизм, зачем они нужны. Также мы рассмотрим модель конкурентности в Go и ее особенности с более привычными моделями в других языках.

В этом разделе не будет практики, он по большей части теоретический. Однако понимание теории поможет вам лучше понять подходы конкурентного программирования в Go.

Тема достаточно комплексная и интересная, так что давайте приступим!

Что такое конкурентность?

В компьютерном программировании **конкурентность** – способность компьютера справляться с множеством разных задач одновременно.

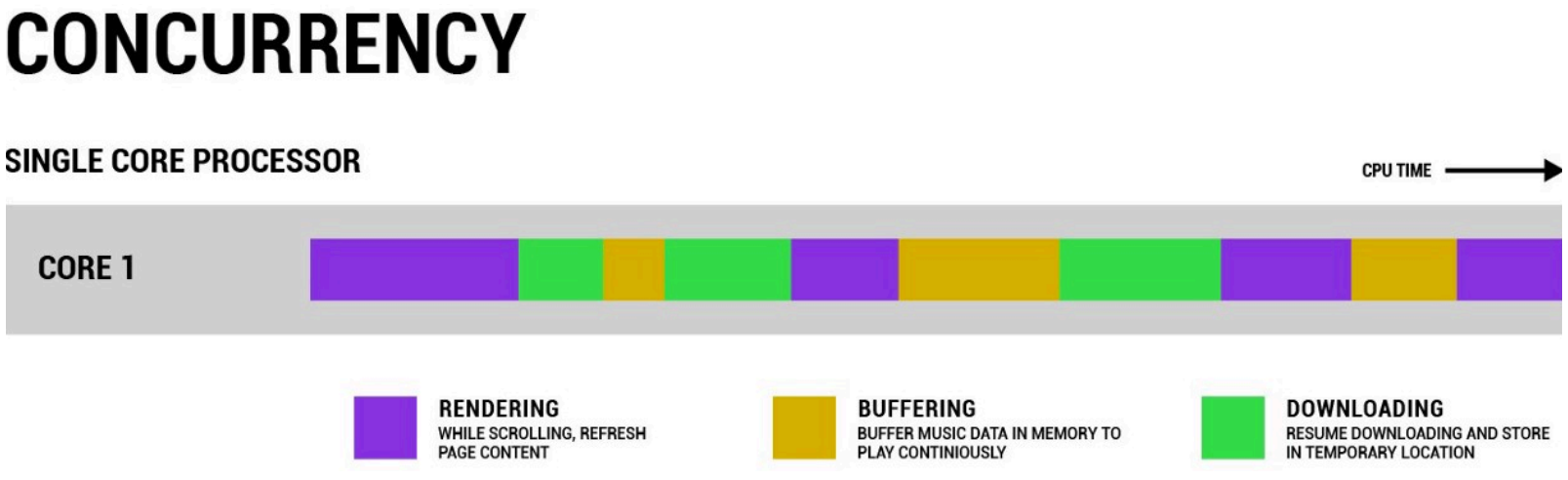
Давайте возьмем для примера использование браузера. Если вы гуляете по сети, происходит много вещей одновременно.

Например идет скачивание файла в то время как вы слушаете музыку и скролите веб сайт в другой в кладке. Можно сказать что компьютер выполняет несколько задач одновременно. Если бы компьютеры так не умели, нам бы пришлось ждать окончание скачивания файла, чтобы продолжить слушать музыку или скролить сайт.

Архитектура ЦПУ устроена таким образом, что одно ядро может выполнять лишь одно действие в единицу времени. В наши дни на рынке все еще есть множество одноядерных процессоров. Но если у вас компьютер с одноядерным процессором, то неужели вы не сможете скачивать файл, слушать музыку и листать мемы в другой вкладке одновременно?

Конечно сможете, а достигается это с помощью *конкурентности*.

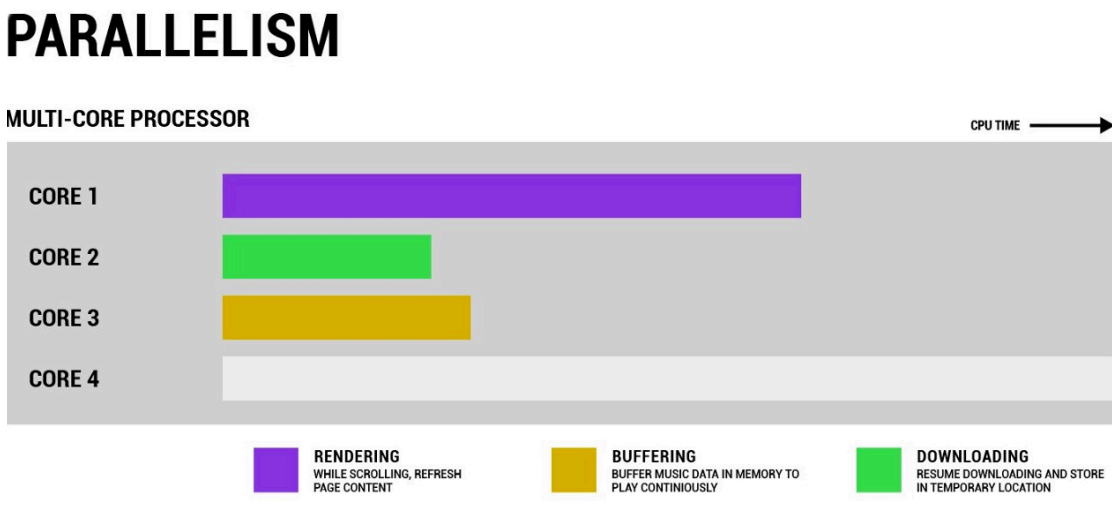
Давайте посмотрим диаграмму, которая демонстрирует как одноядерный ЦПУ справляется с примером с браузером.



Как вы можете увидеть, все задачи разбиваются на кусочки, которые делятся по приоритетности, а ЦПУ постоянно переключается между ними, *создавая иллюзию одновременного исполнения*.

Что такое параллелизм?

А что если у нас многоядерный процессор? Как в таком случае распределяются задачи?



Одно ядро может выполнять одну задачу в единицу времени. Когда у нашего ЦПУ многоядерная архитектура, он может разделять задачи между ядрами, которые будут выполнятся одновременно т.е. *параллельно*.

Обработка нескольких задач одновременно и есть **параллелизм**.

Устройство компьютерной программы

Для того чтобы понять модель конкурентности в Go, для начала нам стоит углубится в более низкоуровневые концепции и понять как на нашем компьютере выполняются программы.

Когда вы пишите программы на разных языках типо `C` , `Java`, `Go`, `Python`, `Scala`, `Ruby` , `Erlang` и тд. ваши программы – просто набор символов в текстовом файле. После компиляции (или интерпретации в случае скриптовых языков), код языка превращается в набор инструкций в формате нулей и единиц, которые отправляются в ОС для выполнения.

Процессы

Когда инструкции после компиляции передаются в операционную систему, ОС создает *процесс* и аллоцирует (выделяет) под него адресное пространство, в котором находятся стэк (stack) и куча (heap), присваивает PID (process id) и делает другие важные вещи.

Любой процесс обладает как минимум одним *потоком,* который называется *главным потоком программы*. Главный поток имеет возможность создавать другие потоки внутри этого процесса.

По сути, процесс можно назвать контейнером, который собирает и аллоцирует все необходимые элементы и ресурсы ОС, которые в последствии могут быть переданы в потоки.

А что же тогда такое потоки? Зачем они нужны и какова их роль?

Потоки

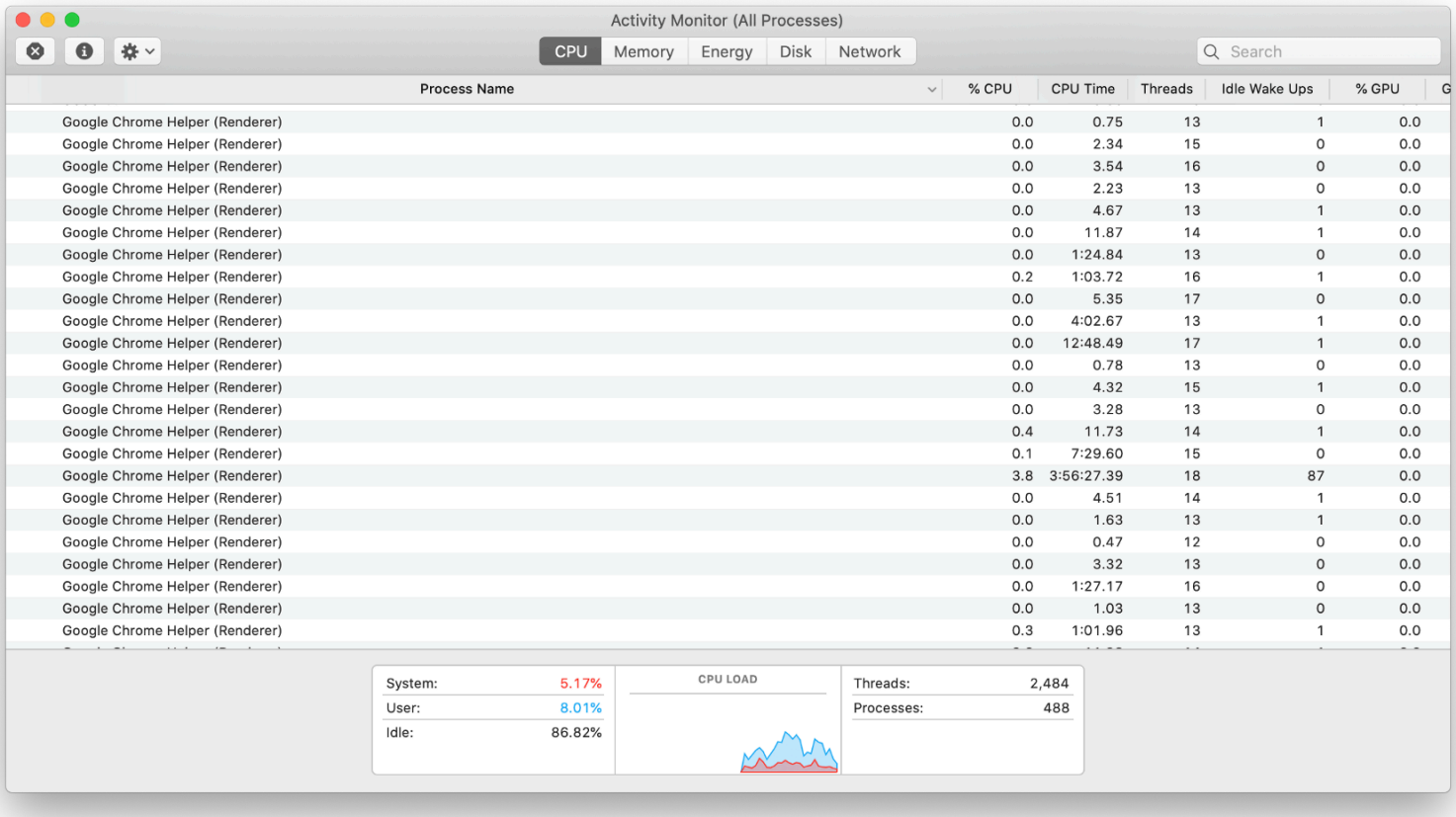
Поток – легковесный процесс внутри самого процесса. Потоки делят между собой общую память, в то время как процессы не могут так просто обращаться к адресному пространству друг друга.

Во время выполнения поток хранит все данные в области памяти, называемой стэком. Стэк создается во время “рантайма” процесса (время выполнения процесса) и зачастую имеет фиксированный размер 1-2 мб. Стэк выделяется под каждый поток отдельно и эта область памяти не делится между несколькими потоками.

Существует также другой тип в области памяти, именуемый кучей (heap). Куча является характеристикой самого процесса, поэтому она делится между потоками.

Приводя аналогию, можно сказать, что кухня в ресторане – это процесс. Задача кухни готовить различные блюда. А внутри этой кухни имеется много поваров и других рабочих – они являются потоками. Все они занимаются разными вещами, однако делают это на одной кухне и читают одну и ту же книгу с рецептами.

На практике все это означает, что когда мы запускаем, к примеру, веб браузер – ОС создает новый процесс. Когда же мы начинаем скачивать файлы, проигрывать музыку и открывать новые вкладки, этот процесс создает множество разных потоков. А убедится в этом можно, запустив Activity Monitor.



Мы с вами узнали, что конкурентность –справляться с несколькими задачами одновременно, а параллелизм – выполнять несколько задач одновременно.

Задача здесь – это поток. Соответственно, когда несколько задач запускаются в конкурентной или параллельной манере – мы работаем с несколькими потоками. Это называется *многопоточностью*.

Для того чтобы работать с несколькими потоками, ОС занимается *планировкой потоков (или thread scheduling)*.

Когда у нас есть несколько потоков, которые работают с одним и тем же участком памяти, у нас может возникать такое явление как *race condition* (состояние гонки). Это нужно постоянно держать в голове при разработке конкурентных приложений.

Конкурентность в Go

Наконец-то мы добрались до конкурентности в Go. В отличии от таких классических ООП языков, как, например,`Java` , в котором существует отдельный класс `Thread` , предоставляющий возможность создавать потоки внутри процессов, Go предоставляет ключевое слово `go` .

Использование `go` перед вызовом функций или методов создает новую *горутину*. По своей сути горутина выполняется как поток, но по факту, является *абстракцией над потоками*.

Когда мы запускаем программу на Go, райнтайм самого Go создает несколько потоков на ядре, на которых будут помещаться все горутины. Когда на одном потоке процесса происходит блокировка горутины, рантайм переключает контекст на другую горутину. По своей сути, это та же планировка потоков, только управляемая не ОС, а самим языком и является существенно быстрее.

Для большинства случаев, советуется запускать программы, написаны на Go, на одном ядре. Однако в языке есть возможность распределить горутины между доступными ядрами системы с помощью `runtime.GOMAXPROCS(n)`, где n– количество ядер.

Потоки и Горутины

Разобрав тему потоков и горутин, вы уже могли понять что между ними есть существенная разница. Перед вами небольшой список основных различий, однако если углубляться, их намного больше.

Потоки	Горутины
Потоки ОС управляются ядром ОС	Горутины управляются "рантаймом" Go
Потоки ОС в основном имееют фиксированый размер в 1-2MB	Горутины обычно имеют размер стэка 2KB
Размер стэка определяется во время компиляции и не может увеличиватся	Размер стэка определяется во время рантайма и может расти вплоть до 1GB что возможно благодаря аллокации и освобождения места из хипа
У потоков нету простого способа коммуникации между собой. Такая коммуникация имеет большую задержку	Горутины используют "каналы" для быстрого общения между собой с маленькой задержкой

Чтобы выделить на примере основные преимущества модели конкурентности в Go, представьте веб сервер, который обрабатывает 10000 запросов в минуту. Если вам нужно обрабатывать все запросы конкурентно, вам нужно создать 10000 потоков. Именно так и делает **Apache Server.** Если поток расходует 1MB памяти, то для обработки 10000 запросов нужно будет выделить ~10GB памяти!

В Go вы также можете без проблем создать 10000 горутин для обработки запросов, под каждую с которых будет выделено 2KB памяти. Сами понимаете существенную разницу и возможности такой модели.

Итак, давайте резюмируем.

Конкурентность – способность компьютера справляться с множеством разных задач одновременно, в то время как параллелизм – выполнять несколько задач одновременно.

Когда мы запускаем программу на компьютере, создается новый процесс, для которого ОС аллоцирует свои ресурсы. Процесс состоит из потоков, которые выполняют разные задачи.

В Go конкурентность достигается с помощью горутин, которые являются абстракцией над потоками, управляются самим рантаймом Go и очень легковесны.

Конкурентное программирование никогда не было столь важным, как сегодня. Веб-серверы одновременно обрабатывают запросы тысяч клиентов. Приложения на планшетах и смартфонах визуализируют анимацию интерфейса пользователя и одновременно в фоновом режиме выполняют вычисления и сетевые запросы. Такой подход позволяет выжимать максимум из ресурсов и улучшать производительность современных систем.

Примечание. В книге “Язык программирования Go” Алана Донована и Брайана Кернигана (в русской адаптации) допущена ошибка. В 8-ой главе, посвященной горутинам и каналам, переводчики называют конкурентность параллелизмом, хотя это две разные концепции.

Перевод гласит о преимуществах Go для параллельного программирования, хотя, как мы сегодня уже говорили, зачастую рекомендуется выполнять программы на одном ядре ЦПУ.

Если вы когда нибудь захотите ознакомится с данной книгой, пусть эта ошибка не сбивает вас с толку. А лучше читайте книгу в оригинале.

Домашнее задание

Для закрепления пройденного материала, поищите дополнительный материал про модель памяти компьютера, стэк и кучу. Почитайте про потоки и процессы, а также про race condition. Также, если вы дружите с английским, рекомендую доклад от одного из создателей языка, Роба Пайка, про конкурентность и параллелизм.

А на этом все. Тема достаточно комплексная, но я надеюсь вы усвоили ключевые моменты.

В следующем разделе мы уже на практике разберем *горутины* и начнем писать конкурентные программы.

Раздел 08:

Горутины и Каналы.

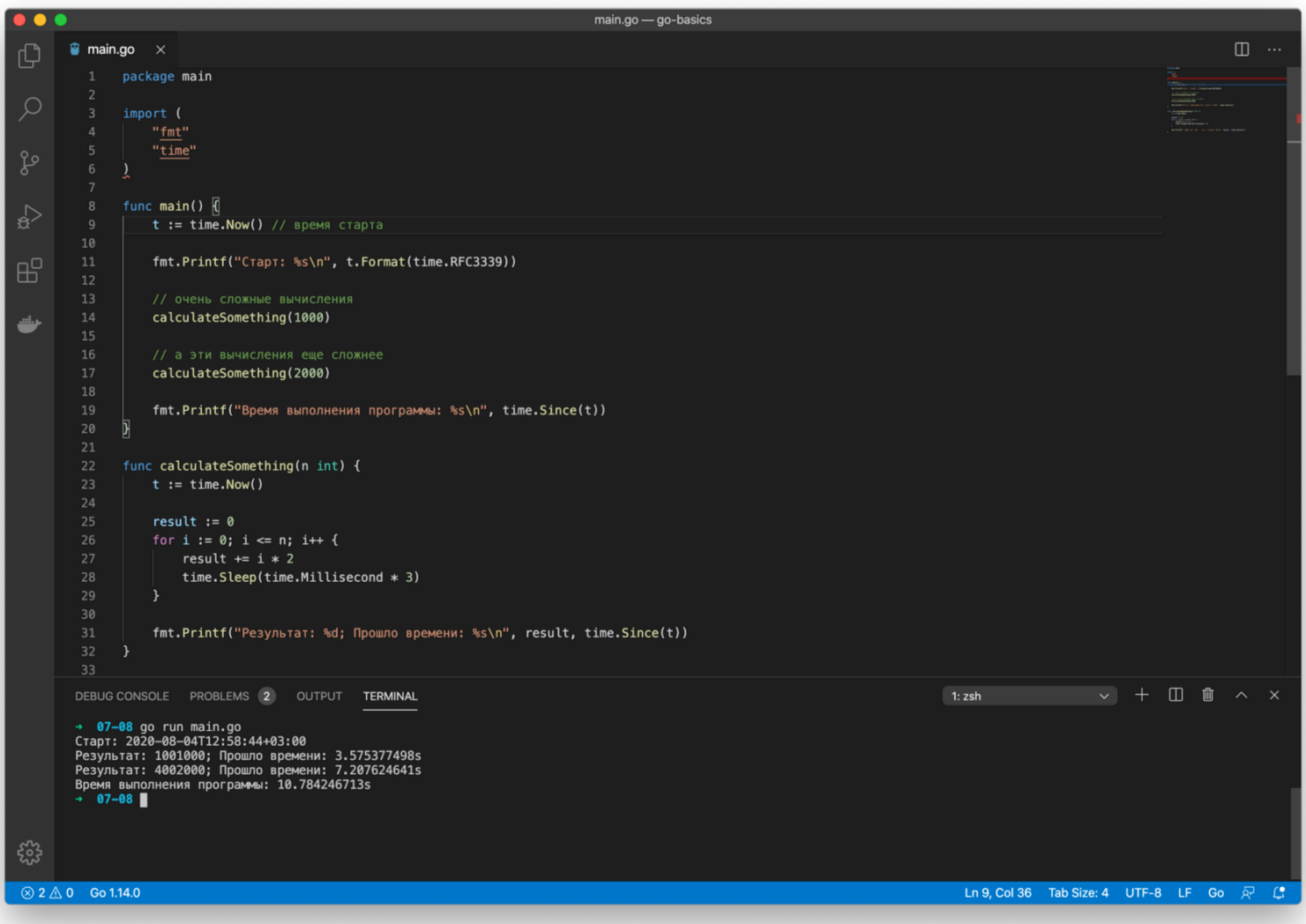
В прошлом разделе мы затронули с вами очень интересную тему конкурентности и параллелизма, а также обсудили реализацию этой модели в Go.

Сейчас же мы с вами на практике поработаем с горутинами и разберем особенности конкурентного программирования.

Поехали!

Горутины

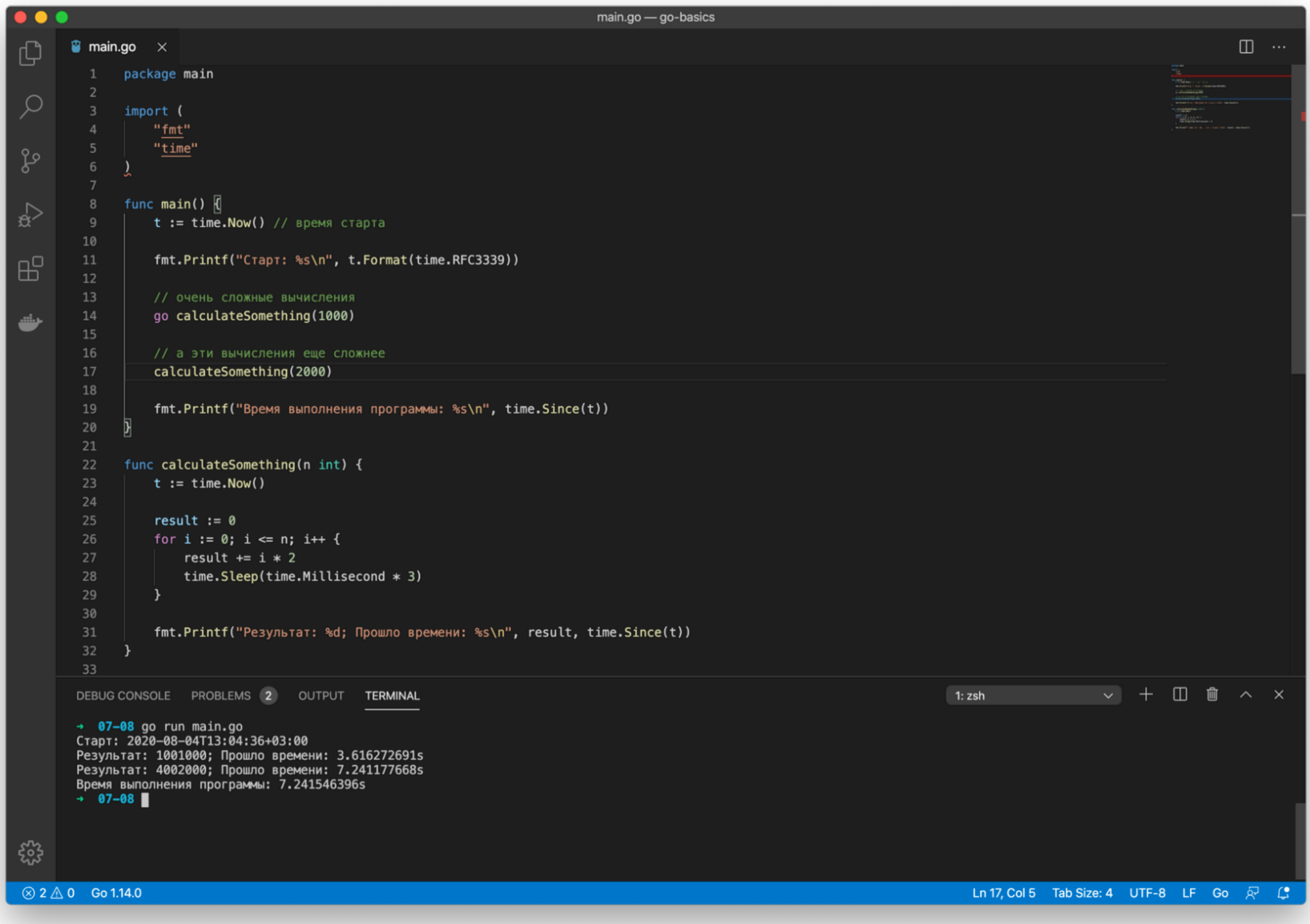
Горутина – обычная функция, которая выполняется конкурентно на заднем фоне вместе с другими горутинами. Чтобы создать новую горутину, необходимо использовать ключевое слово `go` перед вызовом функции. Давайте разберем пример.



Допустим у нас есть функция, которая делает какие-то сложные вычисления. Время выполнение этой функции зависит от количества входящих данных. На примере выше, первый вызов функции занял 3.5 секунды, а второй – 7.2 секунды.

Мы вызвали две функции подряд в синхронной манере. Это означает что выполнение функции `main()` блокируется при вызове `calculateSomething(1000)` , и продолжится только тогда, когда функция закончит свое выполнение.

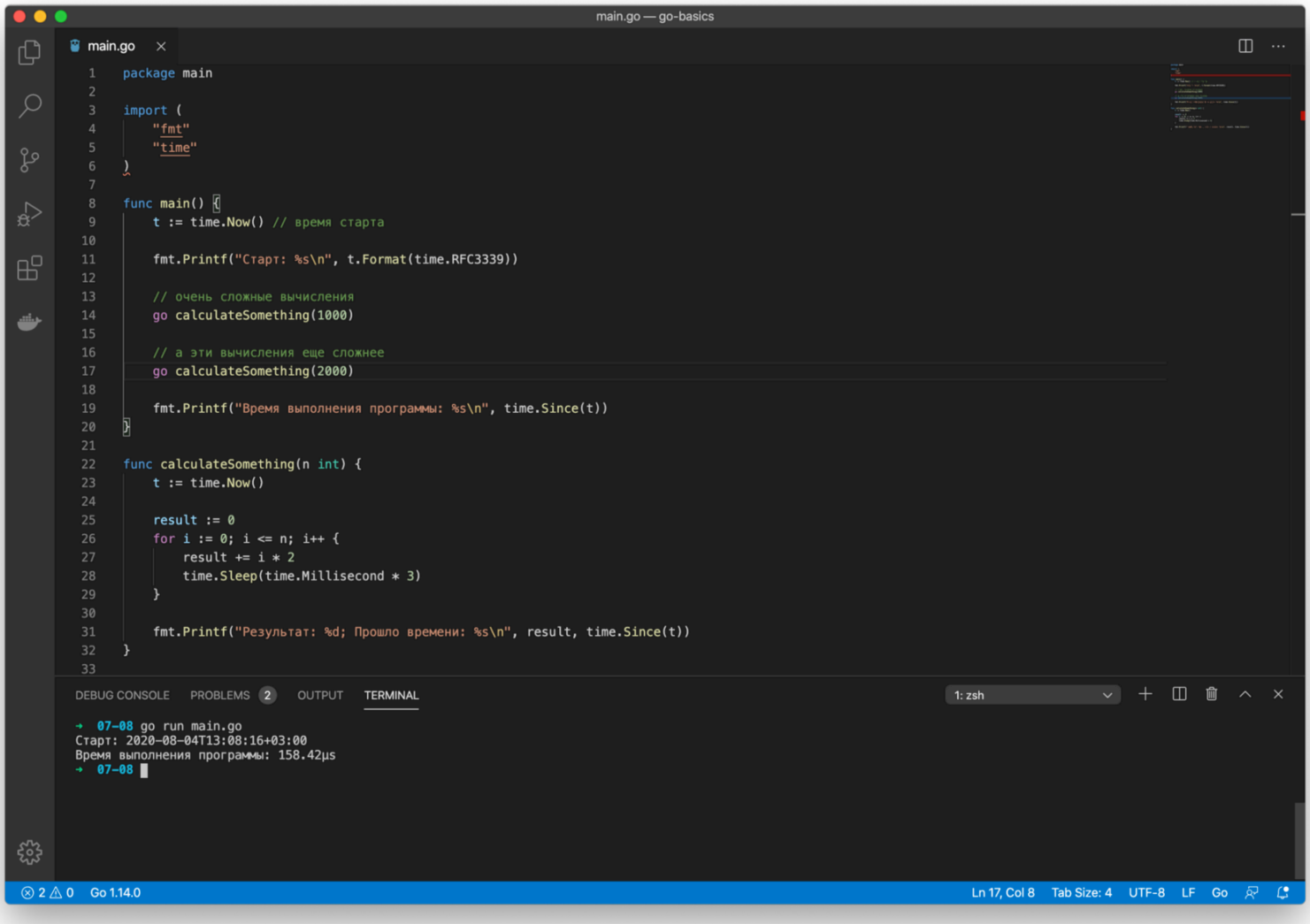
Общее время выполнение программы сейчас получается 10.7 секунд. Давайте теперь ускорим эту программу, добавив лишь одно ключевое слово `go` , создав тем самым горутину.



Теперь наша программа выполняется за 7.2 секунды, почти столько, сколько занимает времени второй вызов `calculateSomething(2000)` .

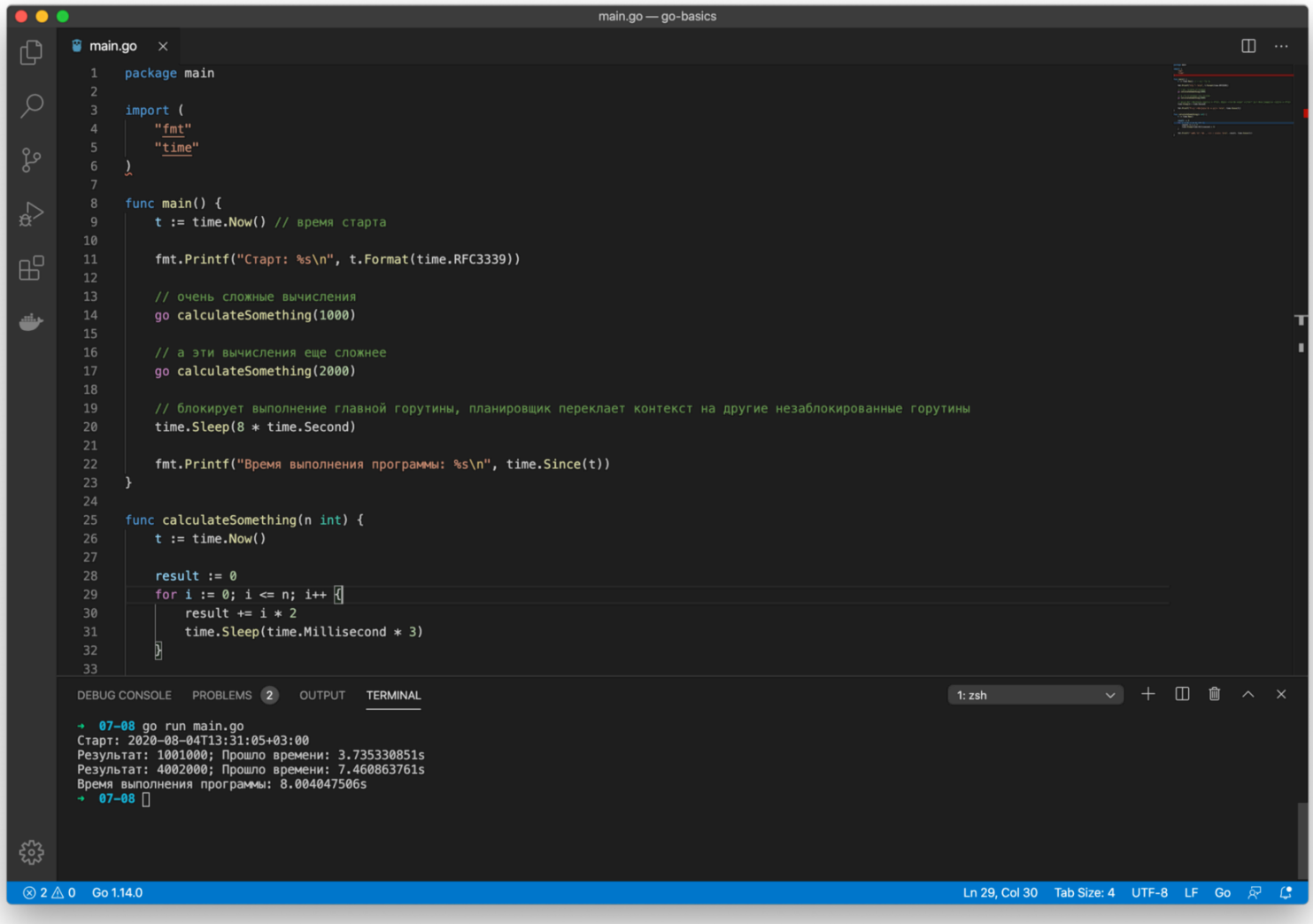
При создании горутины, функция `main()` не блокируется и продолжает выполнение последующего кода. Пока происходили калькуляции из второго вызова функции, наша горутина в заднем фоне также выполняла подсчеты.

А что будет если мы создадим еще одну горутину?



А получается следующей. Главная функция не заблокировалась при создании горутин, и сразу же перешла к последней строчке кода, после чего программа закончила свое выполнение.

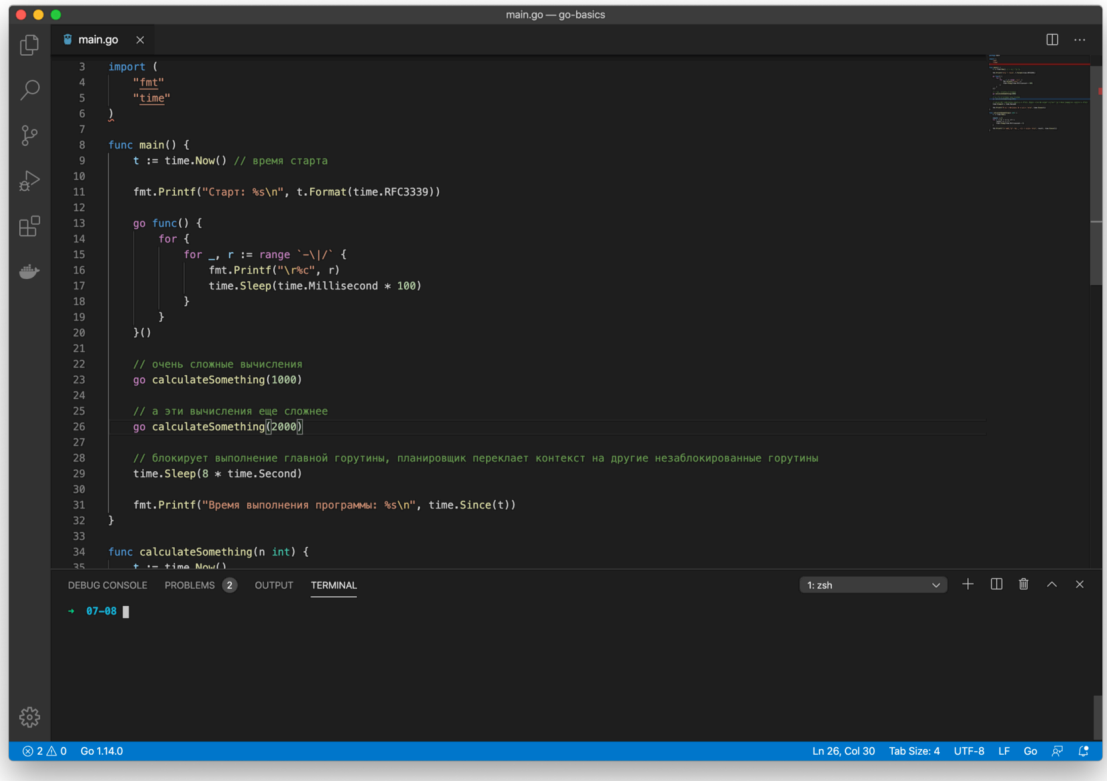
Каждая программа на Go по умолчанию создает одну главную горутину – `main()` .
Горутины выполняются поочередно, за их планирование отвечает планировщик Go.
Переключение контекста между горутинами происходит в момент блокировки выполнения текущей горутины. Блокировки могут происходить при использовании примитивов для блокировок или при чтении/записи в канал.
Функция `time.Sleep()` также блокирует выполнение функции, по этому при ее вызове происходит переключение контекста на другие горутины.



Анонимные горутины

Анонимные горутины – те же анонимные функции, только вызваны используя ключевое слово `go` . Давайте добавим спинер ожидания, чтобы разбавить нашу програму.

Запустите этот код у себя чтобы увидеть анимацию спинера в терминале



Каналы

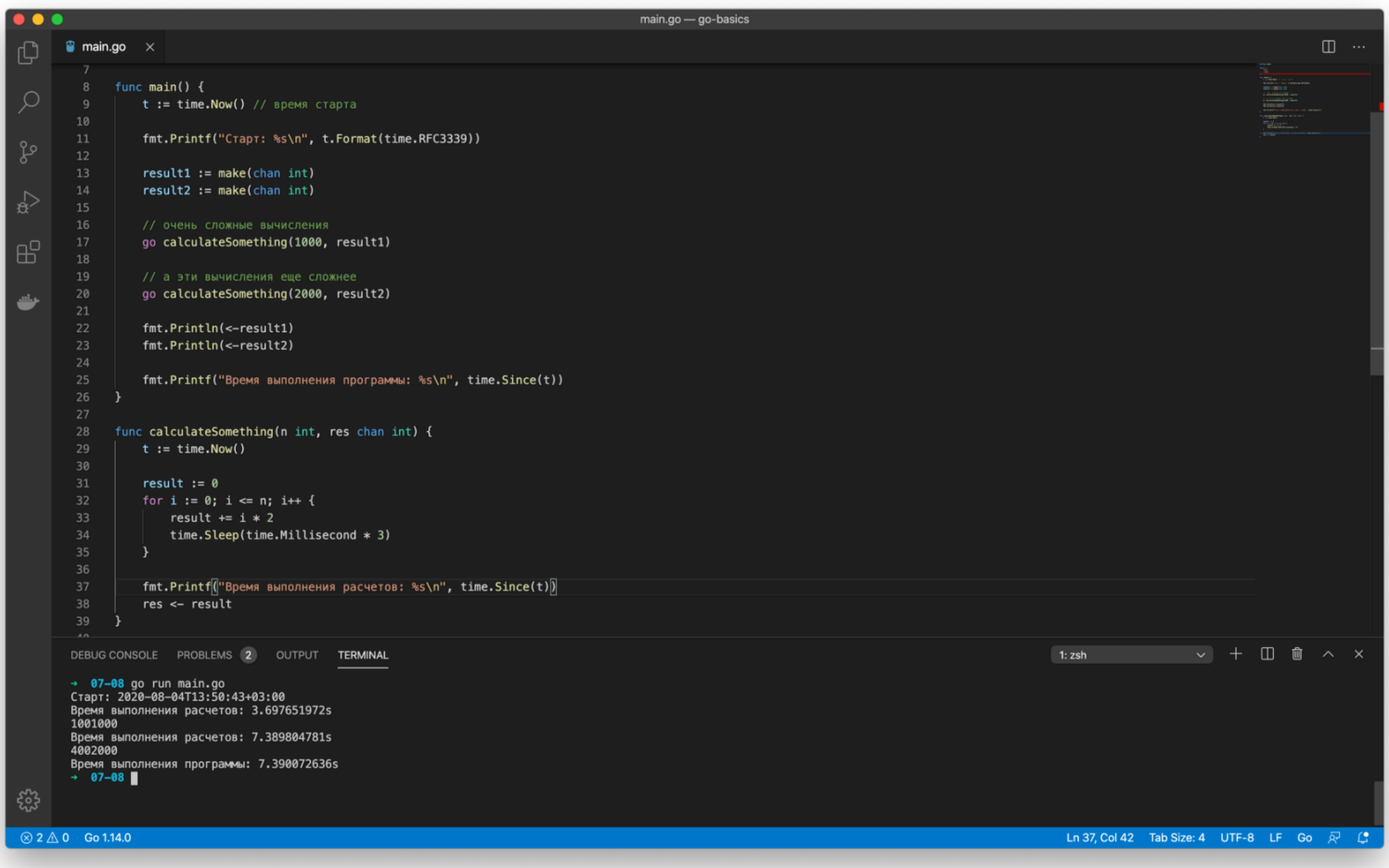
Каналы – это механизм коммуникации между горутинами. Одна горутина может записывать в канал данные в то время, когда другая горутина читает эти данные из этого же канала.

У каждого канала есть тип элемента, который является типом передаваемого по каналу значения. Канал по своей структуре схож со срезом или мапой и тоже является ссылочным типом. Это означает что объявив переменную типа канала, она по умолчанию будет `nil` . А при передаче канала в функцию как аргумент, мы передаем ссылку.

Для записи и чтения данных в/из канала используется `<-` .

```
ch := make(chan int) // инициализация канала

ch <- 1 // запись в канал

number := <-ch // чтение из канала
```

Для того чтобы инициализировать не пустой канал используется функция `make()` .

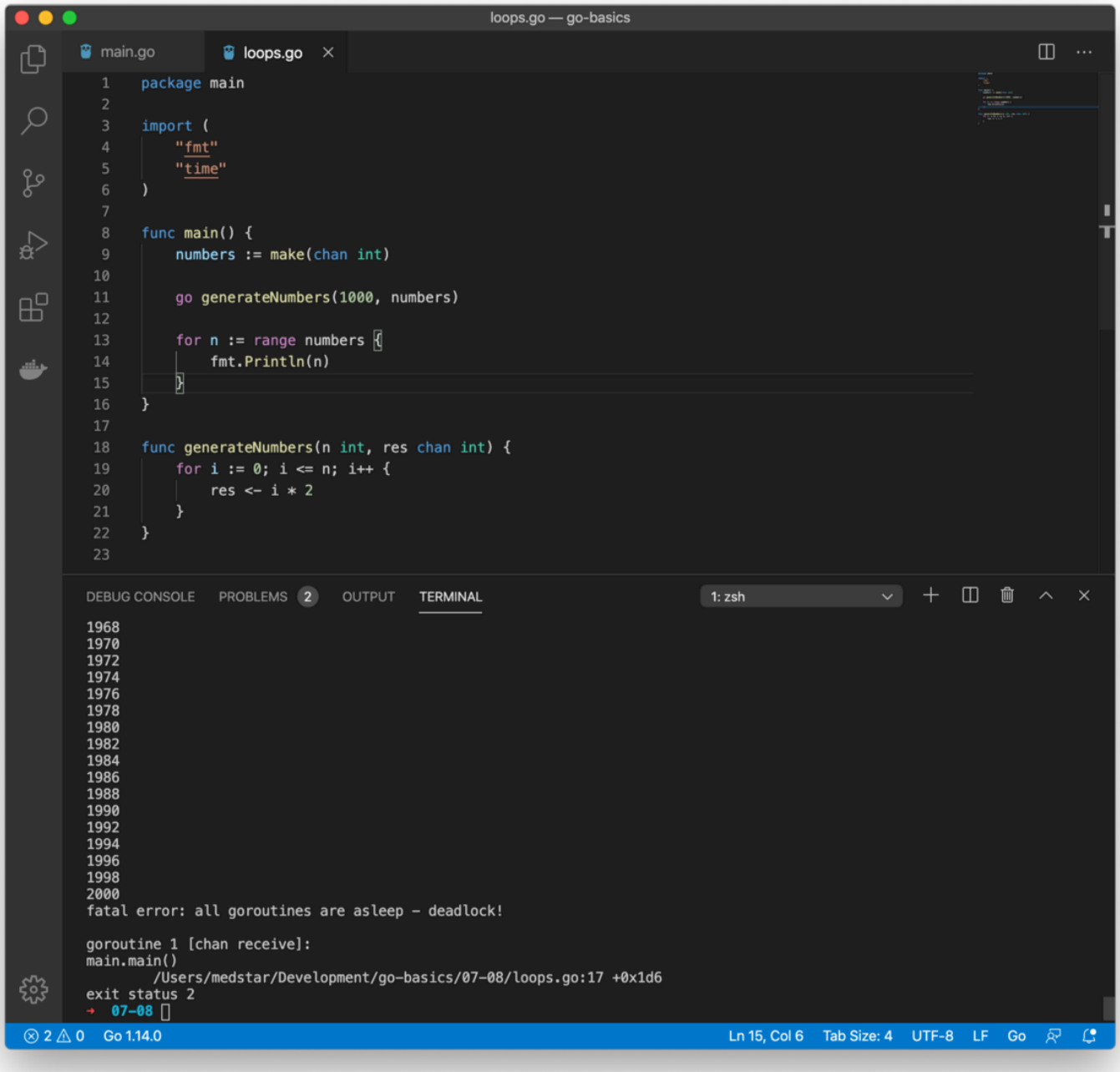


Теперь перепишем нашу программу так, чтобы результат выполнения функции не выводился на экран, а записывался в канал. А в главной функции будем читать эти данные из канала, использовав `fmt.Println(<-result)` .

Как мы уже говорили ранее, чтение с канала блокирует выполнение текущей горутины, по этому все наши горутины успешно выполняются, а функция `main()` блокируется на каждом из вызовов `fmt.Println()` , в ожидании прочитать данные с канала.

Чтение из канала в цикле и deadlock

В канал можно записывать множество значений подряд. В таких случаях целесообразным будет использовать цикл `for` чтобы извлечь все значения. Давайте рассмотрим следующий пример.

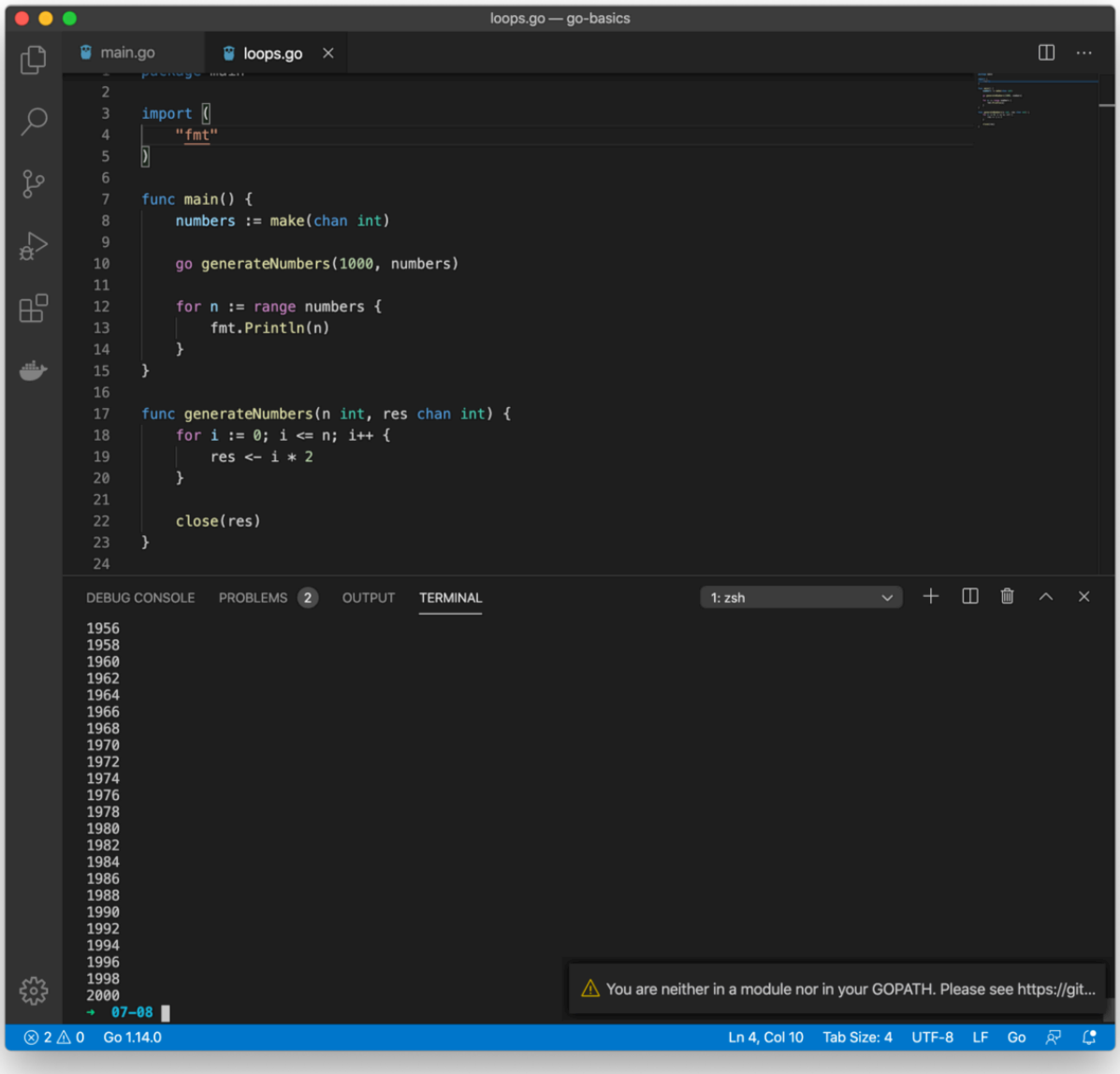


С помощью одного цикла мы записываем в канал поочередном значения, а потом с помощью другого цикла читаем из него эти значения. Однако по достижению последнего числа мы получили ошибку исполнения `fatal error: all goroutines are asleep - deadlock!` .

То что произошло выше называется `deadlock` – когда горутина пытается прочитать значение из канала, но в канале их нету и записи в него не происходит. Это блокирует горутину, и планировщик пробует переключить контекст выполнения на другую. Однако если в данный момент не доступно ни одной не заблокированой горутины, происходит взаимная блокировка или `deadlock`.

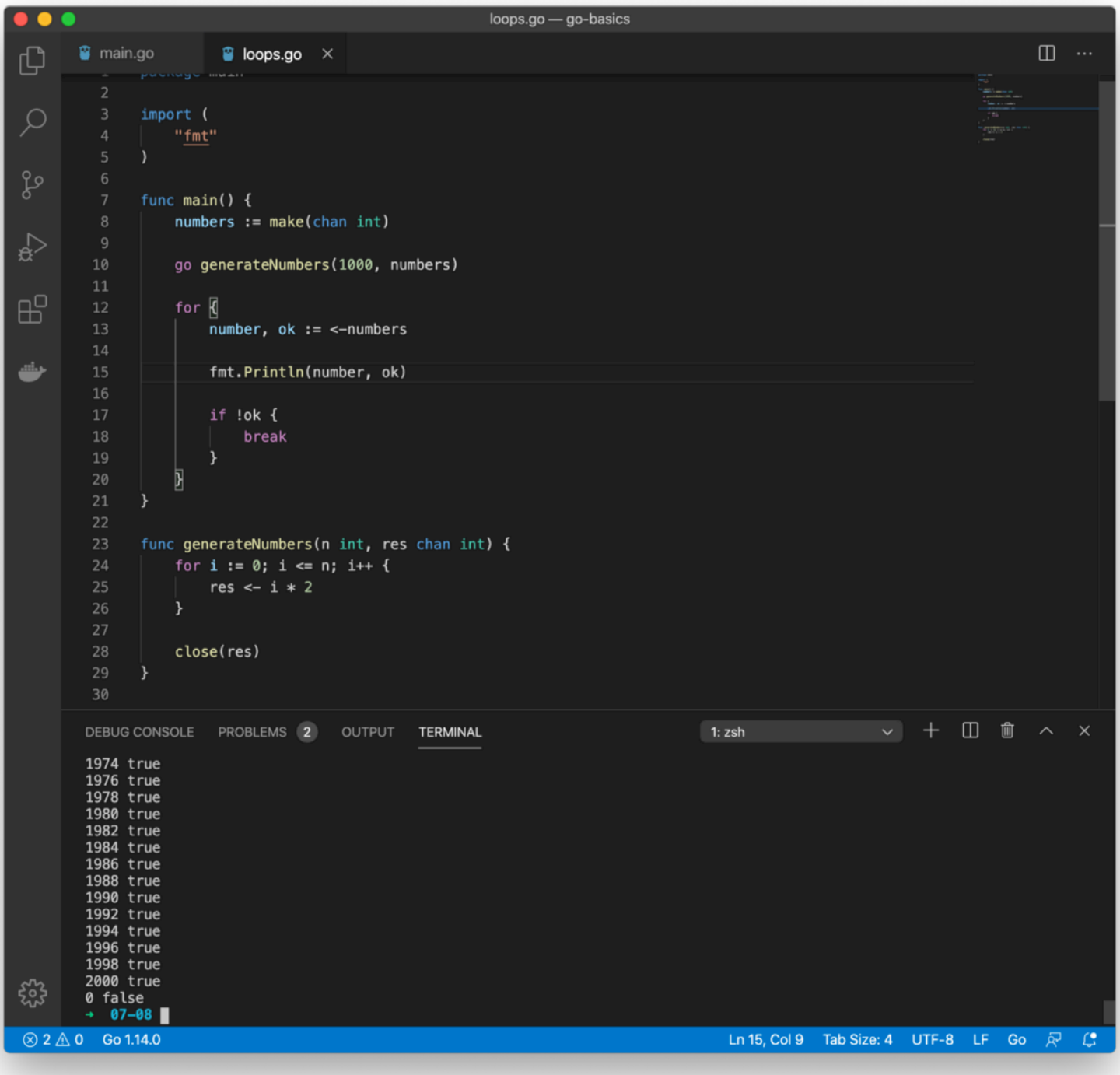
В нашем примере цикл закончил запись в канал, а другой цикл продолжает слушать из этого канала. Но поскольку в него уже больше никогда не произойдет запись, произошел `deadlock` .

Каналы в Go можно закрывать. Если вы записали все значения в канал и больше не планируете в него писать, вы можете его закрыть с помощью встроенной функции `close()` . После этого чтение из канала в цикле `for range` прекратиться.



Теперь после записи всех значений мы закрыли канал, тем самым выйдя из цикла `for` и избежали мертвой блокировки.

На самом деле цикл `for range` делает невидимую магию, поэтому давайте используем другую конструкцию чтобы разобрать этот пример более наглядно.



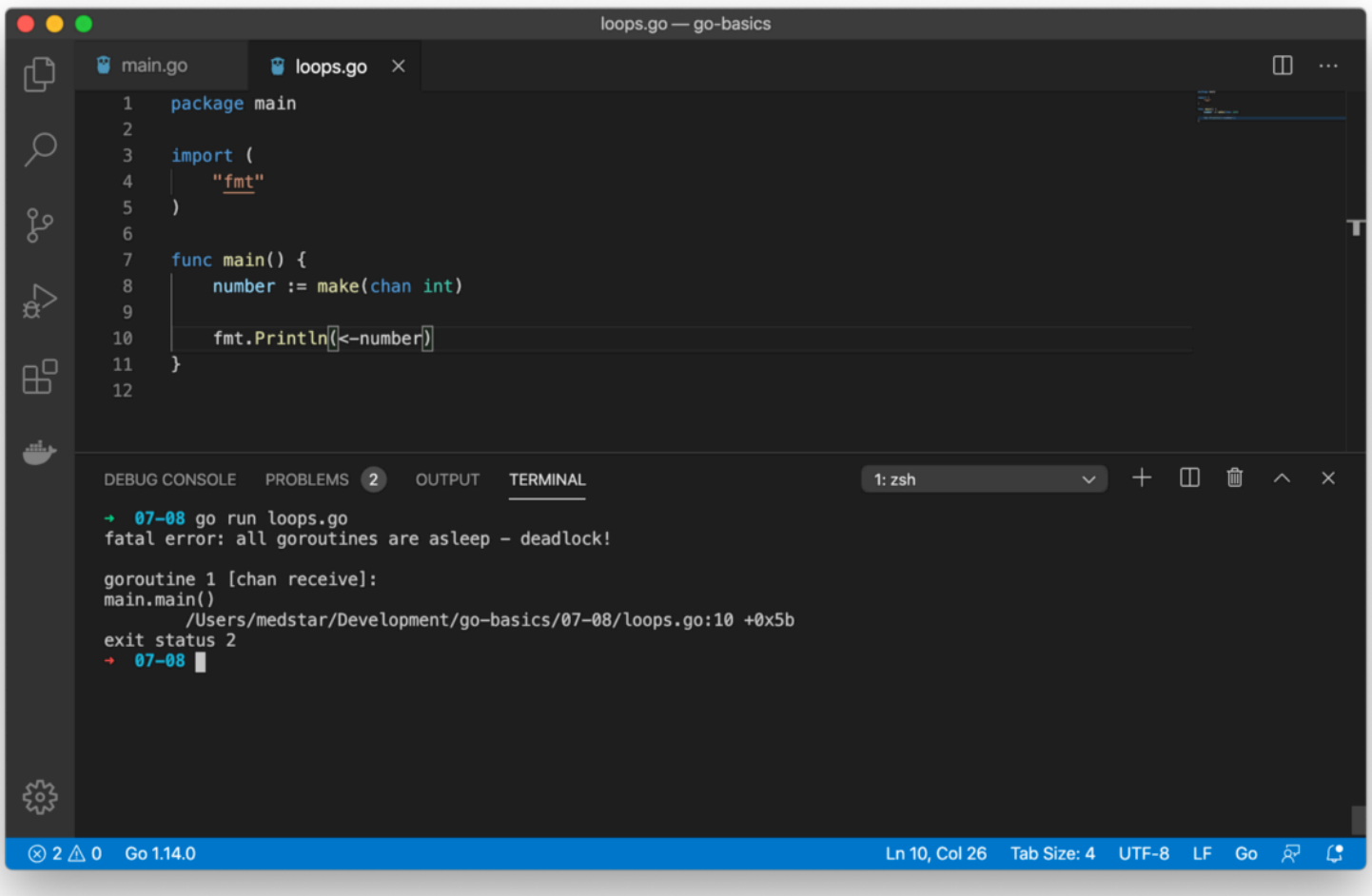
Аналогично с мапами, при попытке чтение из канала мы можем присваивать 2 переменных: одна будет содержать значение, а другая будет типа `bool` и означает, закрыт канал или нет.

В примере выше мы запустили бесконечный цикл `for` , который блокирует выполнение функции `main()` . Дальше мы читаем значения из канала, и проверяем закрыт канал или нет. Если да то, соответственно, выходим из цикла.

Не блокирующее чтение

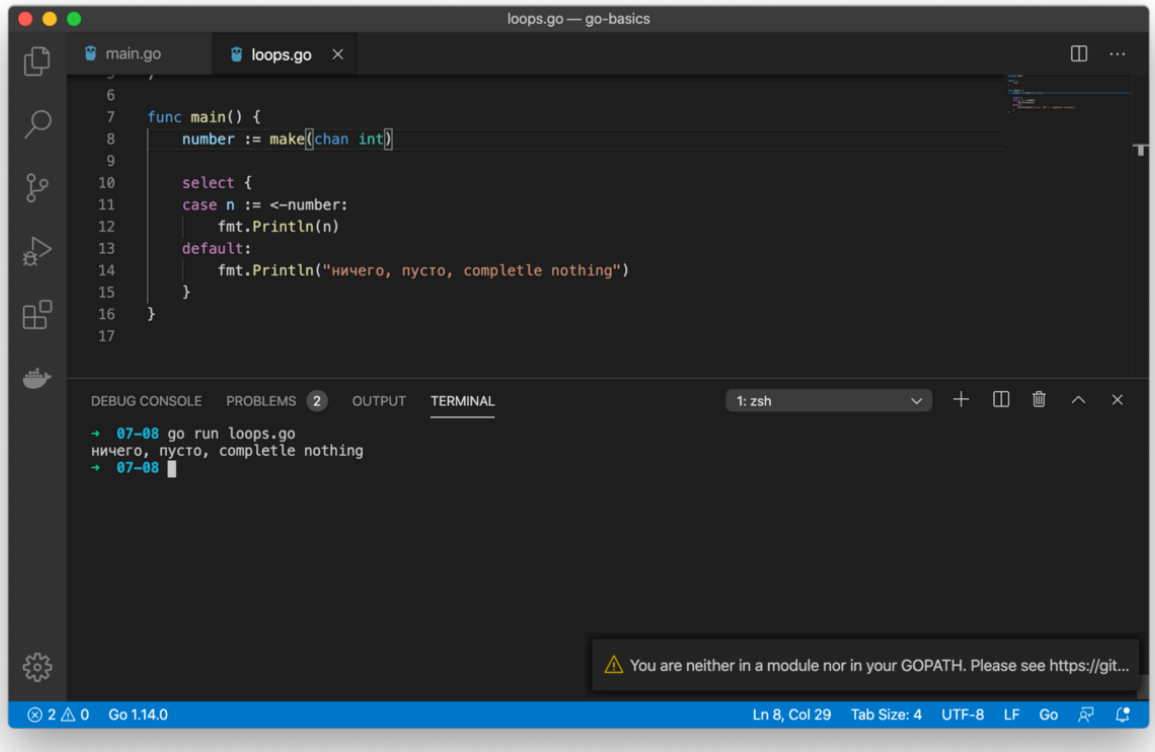
В Go существует конструкция `select` , которая позволяет не блокировать выполнение горутины при чтении\записи в канал.

Давайте рассмотрим ее на примере.

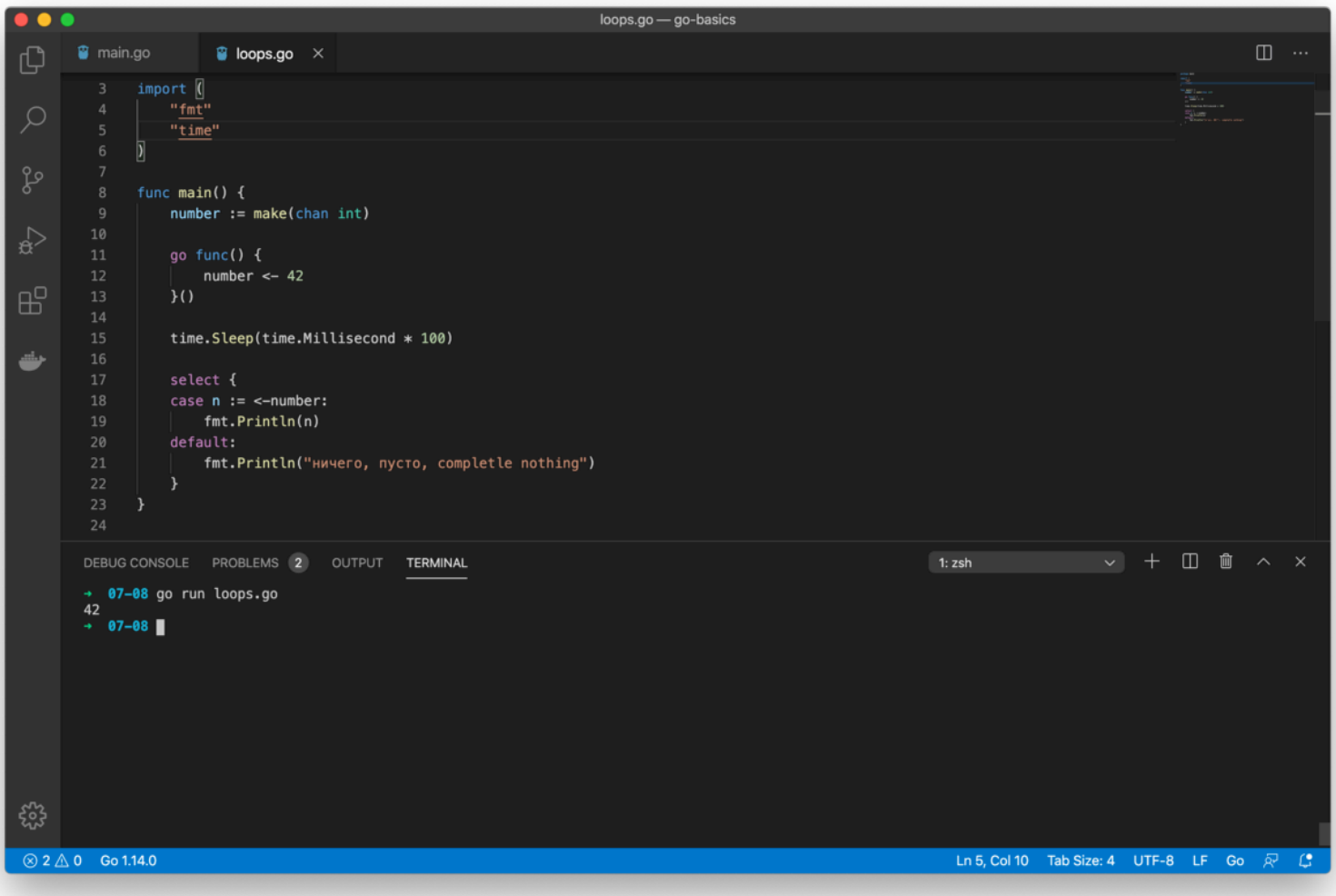


Инициализируем канал и будем читать из него без предыдущей записи. Поскольку операция чтения блокирует выполнение функции `main()` то случится дедлок.

Однако если мы используем конструкцию `select` для чтения из канала, оно не заблокирует выполнение нашей функции.



В случае, если значение не будет получено, будет выполнен код из блока `default:` , а если же мы запишем значение, то сработает `case:` .



Правда если не вызывать `time.Sleep()` то сработает условие`default:` . Как вы думаете, почему?

Давайте резюмируем.

Горутина – обычная функция, которая выполняется конкурентно на заднем фоне вместе с другими горутинами. Для создания новой горутины используется ключевое слово `go` перед вызовом функции.

Все программы на Go по умолчанию имеют одну главную горутину – функцию `main()`.

Переключением между горутинами занимается планировщик Go. Переключение происходит при блокировке выполнения текущей горутины.

Каналы– это механизм коммуникации между горутинами.

Если горутина пытается прочитать из канала, в который никогда не будет произведена запись – случится deadlock.

Домашнее задание

Напишите программу, которая будет считать рекурсивно факториал числа 45. Если вы не знакомы с рекурсией и факториалом, поищите дополнительную информацию самостоятельно. Подсчет факториала займет несколько секунд, по этому выводите в консоль спинер ожидания во время подсчетов.

Попробуйте сделать запись в закрытый канал. Что случится? Почему?

Поищите дополнительную информацию про буферизированные и однонаправленные каналы.

Изучите самостоятельно структуры для блокировок и синхронизации горутин `sync.Mutex` и `sync.WaitGroup` из стандартной библиотеки. Поищите примеры их применения на практике.

Если вы не знакомы с основами HTTP протокола, и не знаете, что такое REST API – настоятельно рекомендую ознакомится с данными концепциями!

Раздел 09:

Работа с HTTP.

Мы с вами проделали уже немалый путь и изучили много интересного. В прошлом разделе мы разобрали конкурентность на практике, научились работать с горутинами и каналами.

Как я говорил в самом начале данной книги, язык Go идеально подходит для разработки современных веб приложений. Легковесная модель конкурентности и стандартные библиотеки для работы с TCP/IP & HTTP протоколами делают Go хорошим выбором для разработки приложений, которые коммуницируют по сети.

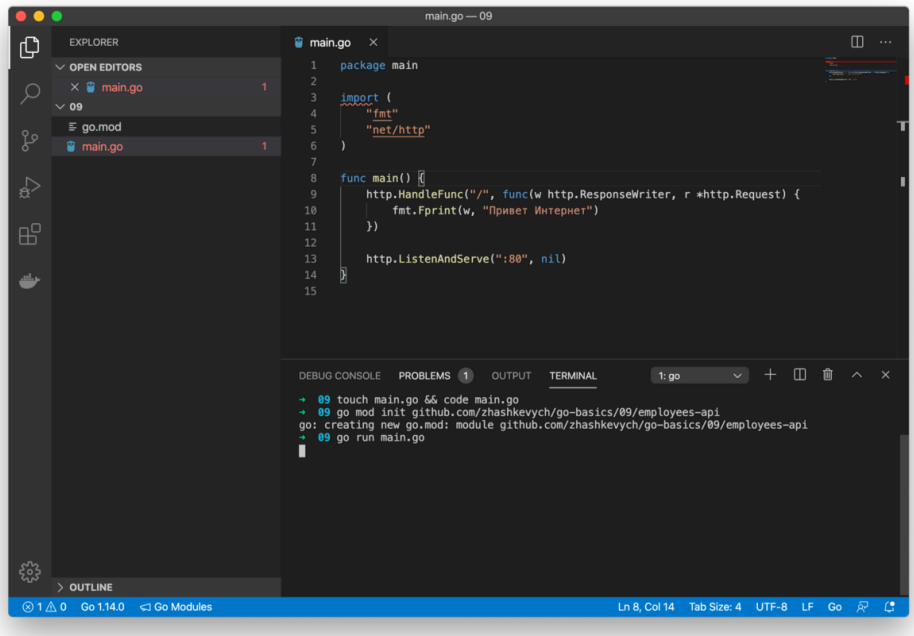
В этом разделе мы посмотрим на пакет `net/http` из стандартной библиотеки и разработаем наш первый HTTP сервер.

Ну что ж, поехали!

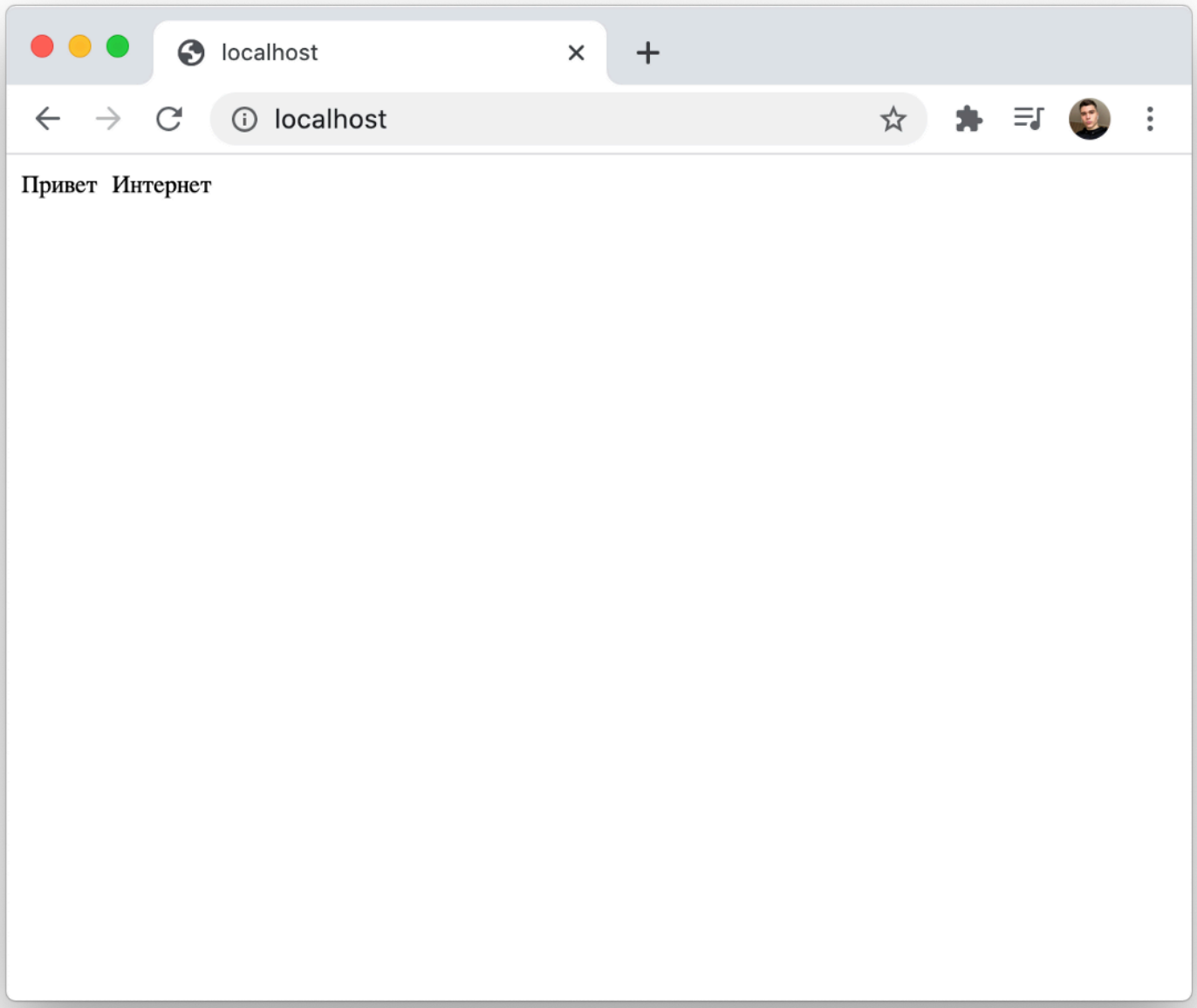
HTTP сервер

Давайте создадим новый проект, инициализируем модуль внутри корневой папки и добавим файл `main.go` .

Для запуска веб сервера в Go необходимо написать лишь пару строк кода. Импортируем пакет `net/http` , а в функции `main()` инициализируем роутер и начнем “слушать” входящие запросы на порту 80.



Теперь если мы запустим наш проект и перейдем в браузер по адресу http://localhost/ то увидим ответ от нашего веб сервера.



Поздравляю, мы написали наше первое веб приложение!

Ну что ж на этом урок окон… Ладно, шучу :)

Давайте разберем, что происходит в этом примере.

Роутинг

Фундаментальная концепция веб сервера – роуты и хендлеры.

В каждом приложении существует набор роутов (маршрутов), по которым мы можем обращаться с помощью клиента (например веб браузера, другого приложения, консольной утилиты `curl` , программы Postman и тд). При обращении на конкретный роут, запрос обрабатывается с помощью хендлера (обработчика).

Роут – это путь в URL после имени хоста.

Для примера, https://golang.org/doc/ – ссылка на документацию по Go, где:

- `https` – название протокола коммуникации
- `golang.org` – название хоста
- `/doc` – роут на страничку документации

Хендлер – это функция, которая умеет работать с запросом и его метаданными (HTTP заголовками, телом запроса и тд.), а также генерировать респонс (ответ).

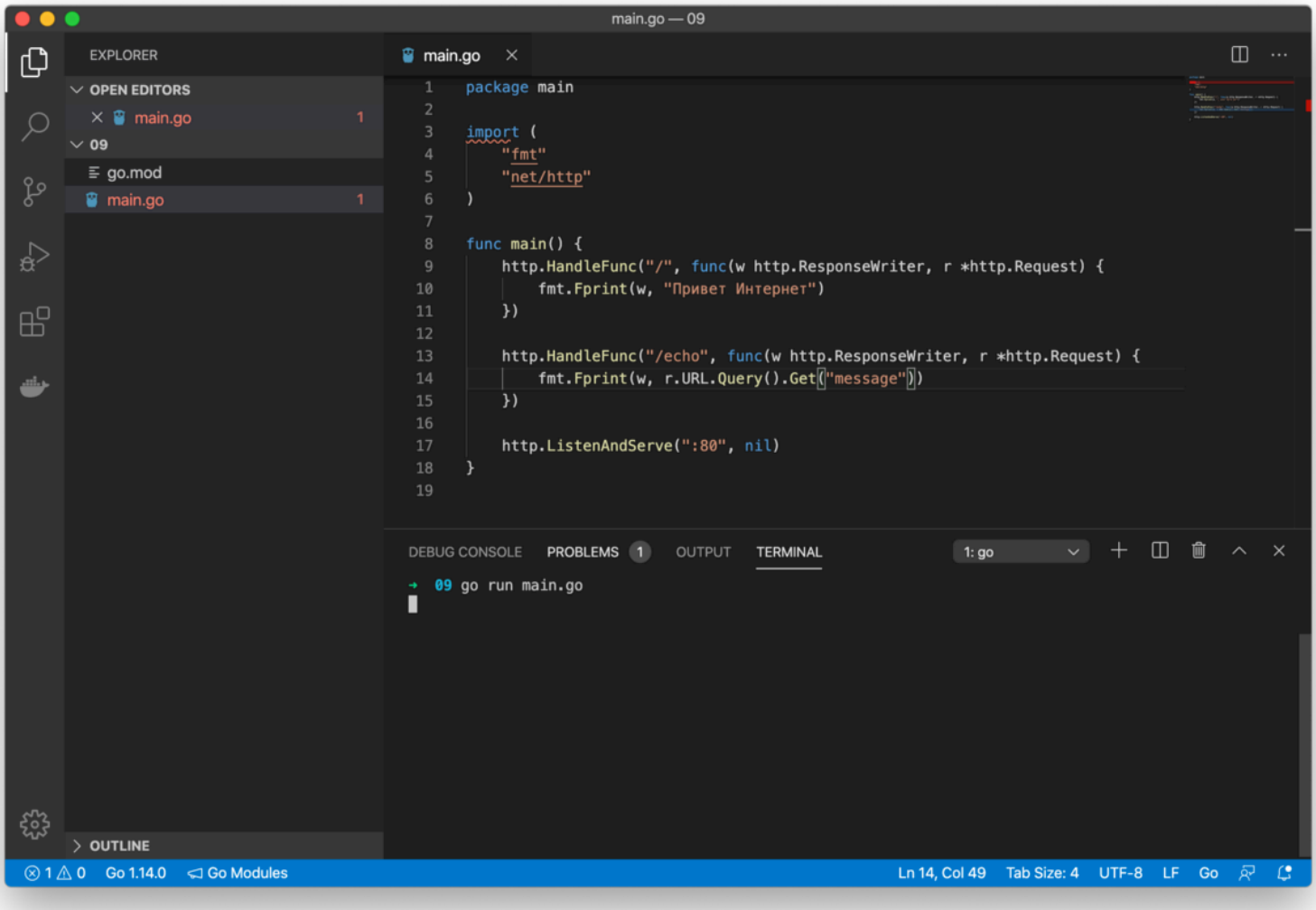
В библиотеке `net/http` хендлер – функция вида:

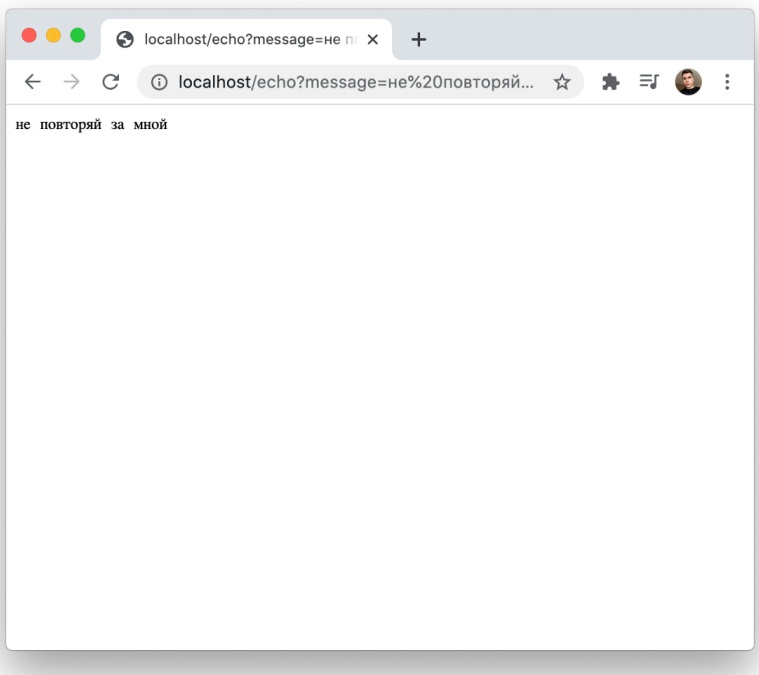
```
handlerFunc(w http.ResponseWriter, r *http.Request) {}
```

Где `r` – объект запроса, а `w` – интерфейс для записи респонса (ответа), который также удовлетворяет интерфейсу `io.Writer` .

В нашем примере мы объявили корневой роут `/` , и добавили для него функцию-обработчик, которая пишет в тело респонса текст.

Давайте создадим еще один роут.





На примере выше мы создали для нашего приложения еще один роут `/echo`, который пишет в респонс значение из параметра URL `?message=`.

Как видите, объект `*http.Request` хранит в себе информацию о нашем запросе. Мы с легкостью смогли получить значение из параметра URL.

В конце нашего приложения мы вызвали функцию `http.ListenAndServe(":80", nil)`, где указали порт, на котором слушать все входящие запросы. В качестве второго аргумента эта функция принимает хендлер (объект интерфейса `http.Handler`). Если указать `nil`, будет использован стандартный хендлер, роуты и обработчики для которого мы объявили выше.

Почему порт 80? В компьютерных системах множество портов уже зарезервированы под конкретные протоколы. С исчерпывающим списком вы можете ознакомиться тут.

Как вы можете увидеть, 80 – стандартный порт для протокола http. Когда вы в браузере пишите домен сайта, например medium.com – ваш браузер находит IP адрес в DNS таблице, а потом делает запрос на этот IP на 80 порт если это http и на 443 – если https. Но это уже тема для отдельного разговора.

Нам совсем не обязательно указывать порт 80. На самом деле мы можем указать любой другой порт.

Давайте подведем итоги.

Для работы с протоколом http, стандартная библиотека Go предоставляет пакет `net/http`.

Фундаментальные составляющие веб сервера – роуты и хендлеры. Роуты это маршруты, обращаясь к которым запрос обрабатывается специальным обработчиком (хендлером).

Домашнее задание

Напишите веб приложение, которое будет принимать в качестве параметра URL значение радиуса круга `?radius=` и записывать в респонс площадь круга.

Почитайте документацию по пакету `net/http` .

Почитайте самостоятельно документацию по сторонней библиотеке https://github.com/gin-gonic/gin . Это достаточно популярная библиотека, своего рода “обертка” над стандартным `net/http` для разработки веб приложений на Go.

На этом все. В следующем разделе мы сделаем финальный проект книги, а именно разработаем небольшое REST API, используя библиотеку `gin-gonic/gin` , тем самым закрепив весь пройденный материал.

Раздел 10:

Пишем REST API.

В предыдущем разделе мы с вами написали самый простой HTTP сервер и познакомились с пакетом `net/http` .

Сейчас же мы закрепим весь пройденный в этой книге материал, разработав финальный проект. Цель данного проекта – применить на практике полученные знания, научится работать со сторонними библиотеками и понять основы разработки REST API на Go.

Для данного проекта мы будем использовать стороннюю библиотеку `github.com/gin-gonic/gin` , которая на данный момент является очень популярной в индустрии.

Ну что же, поехали!

Проект

Это будет АПИ для работы с уже привычными нам сотрудниками из предыдущих уроков. Оно позволит создавать, просматривать, редактировать и удалять сотрудников и имеет следующий вид:

POST /employee/ - создание сотрудника

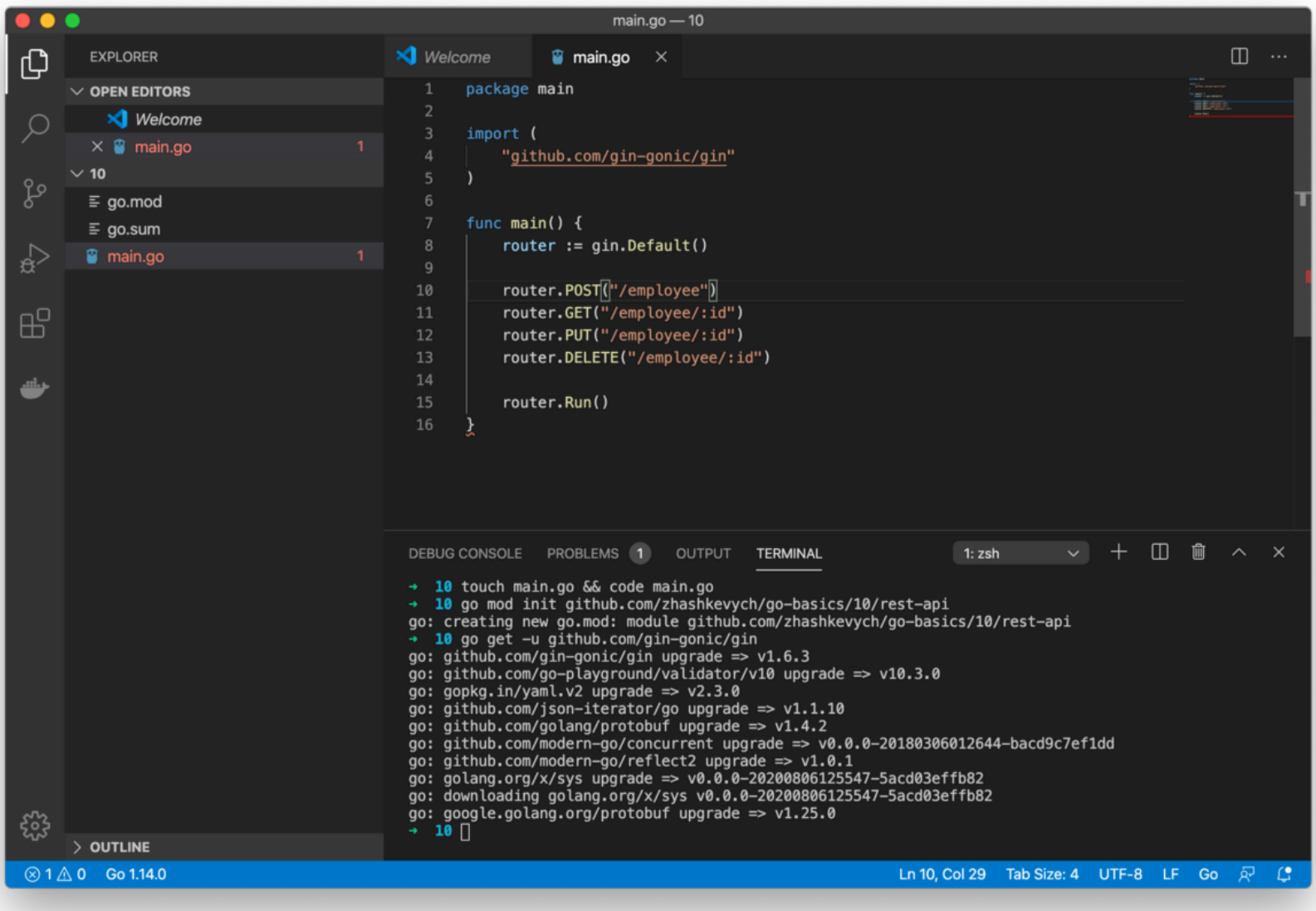
GET /employee/<id> - получение информации о сотруднике

PUT /employee/<id> - редактирование информации о сотруднике

DELETE /employee/<id> - удаление сотрудника

Хранить сотрудников мы будем в мапе, используя слегка модифицированное хранилище с урока, посвященного структурам и мапам.

Давайте создадим новый проект, инициализируем модуль, добавим файл `main.go` и импортируем библиотеку `gin-gonic/gin` .

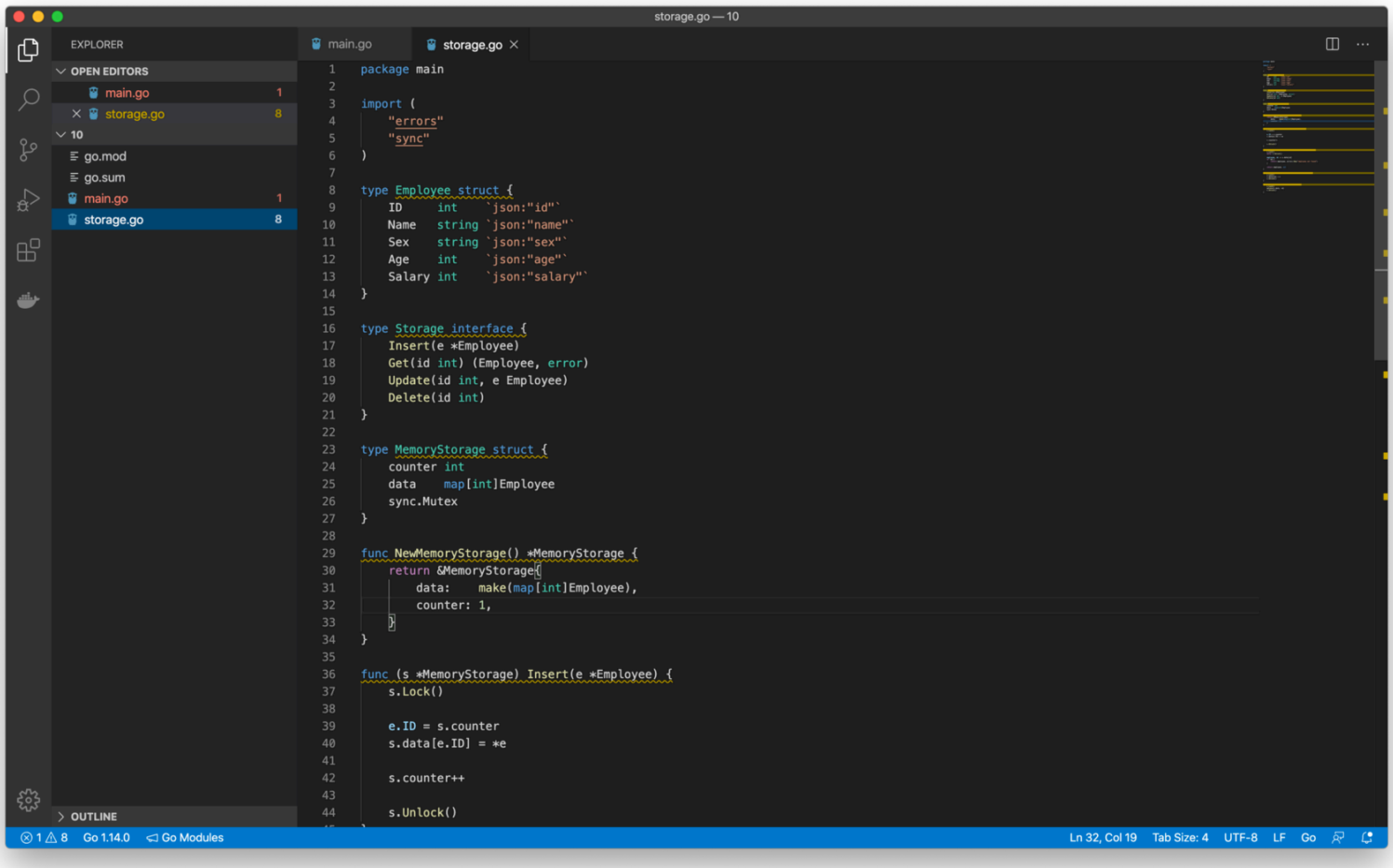


В функции `main()` создадим объект типа `*gin.Engine` . Для этого объекта можно задавать роуты, а при вызове метода `router.Run()` , запуститься HTTP сервер на порту 8080. В качестве аргумента `Run()` можно передать строку с указанием порта, например `Run(":8000")` , но оставим как есть для простоты.

Библиотека `gin`имеет удобную работу с параметрами из URL. Объявив в роуте параметр `:id` , мы сможем легко обратиться к нему в обработчике.

Хранилище

Давайте создадим теперь отдельный файл `storage.go` , в котором опишем наше хранилище и структуру сотрудника.



Итак, мы создали структуру нашего сотрудника, объявили интерфейс хранилища и создали структуру, реализующую его.

Давайте рассмотрим строки в структуре после указания типов. Эти строки – это теги, которые могут хранить в себе абсолютно любую информацию. Относитесь к тегам как к метаинформации о поле.

Конкретно в данном примере мы использовали теги `json` , которые позволяют нам указывать имена полей структуры в формате json для кодирование/декодирования структуры.

Также в нашем хранилище мы добавили к мапе поля `sync.Mutex` и `counter int`.

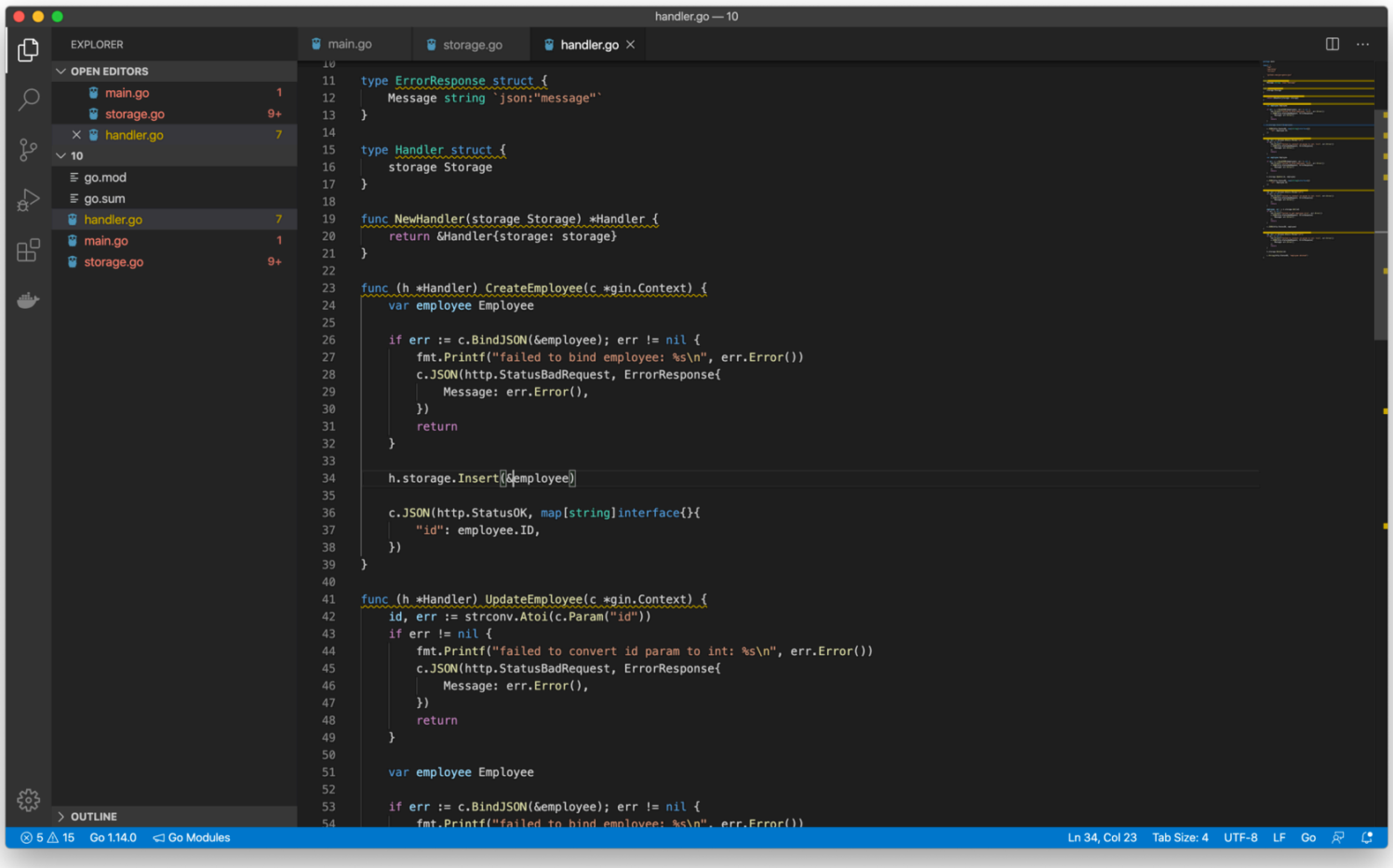
Я надеюсь вы уже имеете примерное представление о мьютексах после выполнения одного из прошлых домашних заданий. Поскольку методы нашего хранилища могут выполнятся конкурентно, нам нужно добавить блокировки чтобы не столкнутся с `race condition`. А `counter` нам нужен чтобы автоинкрементить ID для наших сотрудников.

Почему методы хранилища могут выполнятся конкурентно? При каждом новом запросе создается новая горутина, которая запускает обработчик. Поэтому все запросы в HTTP сервере обрабатываются конкурентно.

Также мы использовали встроенную конструкцию `defer` . Использование этого ключевого слова перед вызовом функции откладывает вызов этой функции в самый конец выполнения. Т.е функция, вызванная через `defer` , выполнится перед запуском команды `return` .

Обработчики

Теперь давайте создадим новый файл `handler.go` , в котором опишем логику для наших хендлеров.

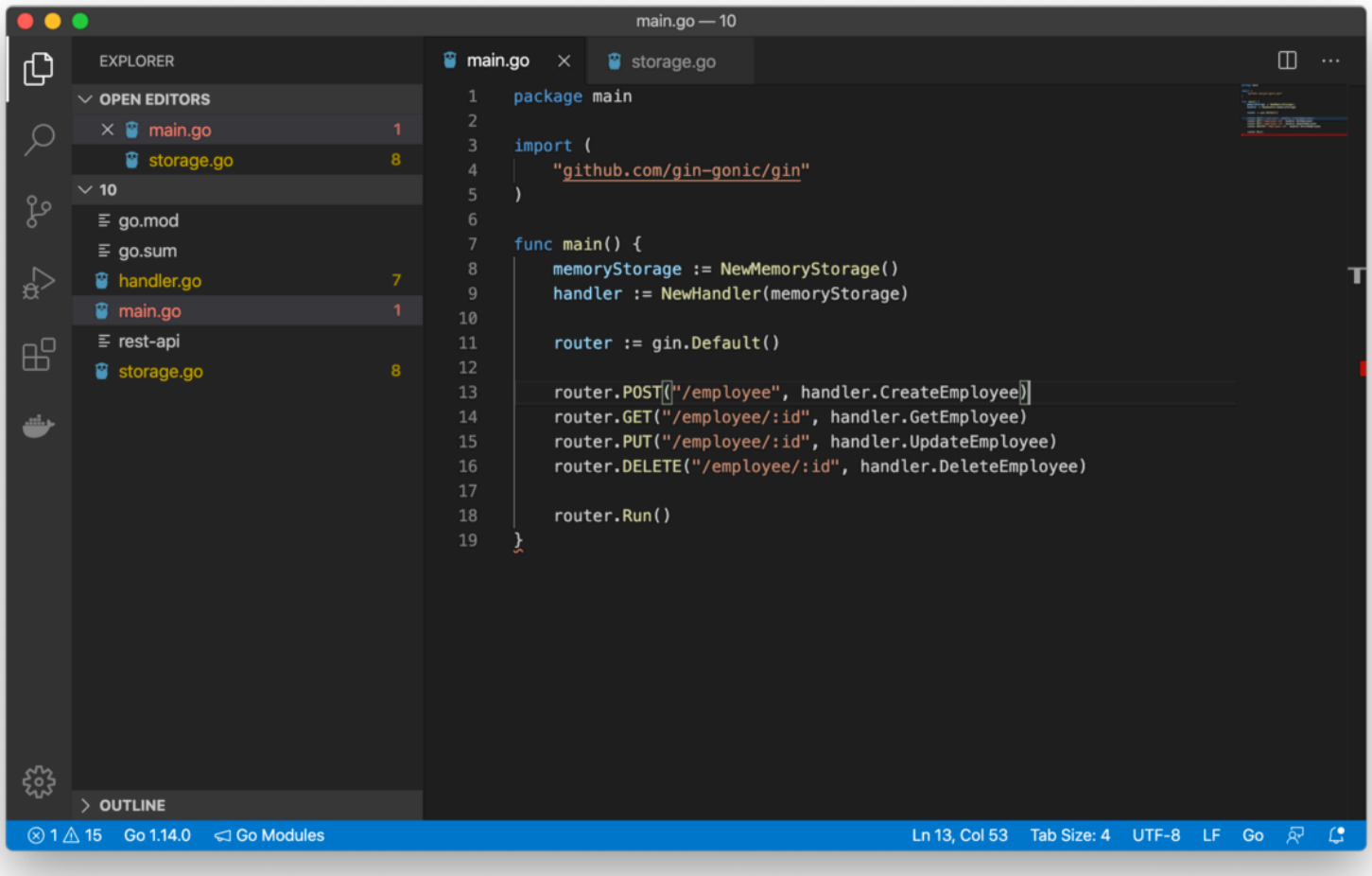


Создадим структуру `Handler` , которая будет принимать в конструкторе объект, имплементирующий интерфейс хранилища. А дальше реализуем хендлеры под все 4 наших эндпоинта (роута).

В библиотеке `gin` функция обработчик имеет вид `func(c *gin.Context) {}` . Объект`*gin.Context` хранит в себе всю информацию про входящий запрос и умеет также записывать ответы.

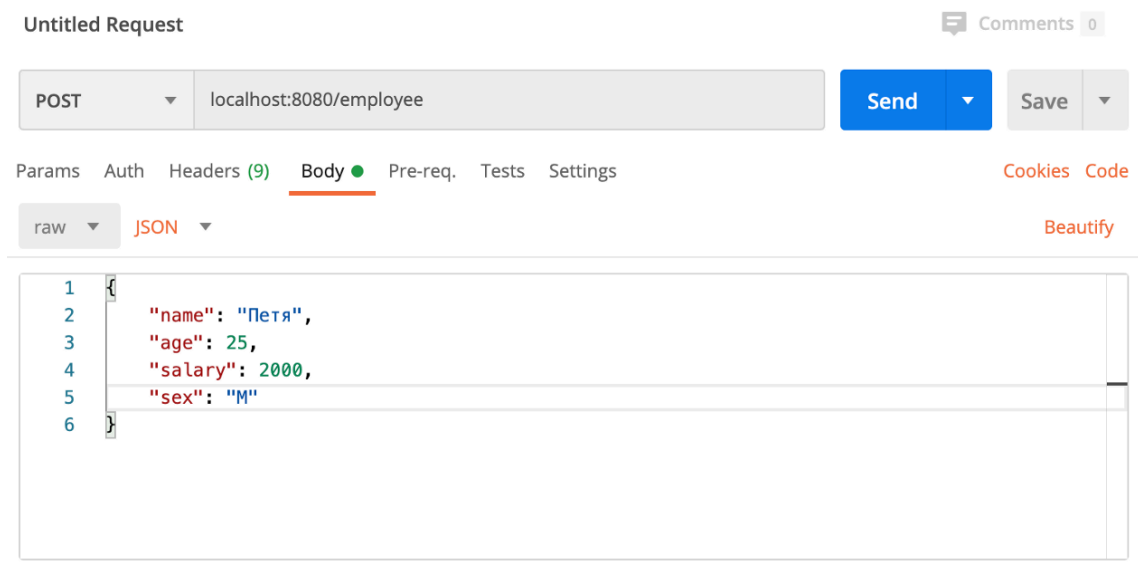
Собираем все вместе

Теперь давайте вернемся к нашей функции `main()` и соберем все элементы вместе.

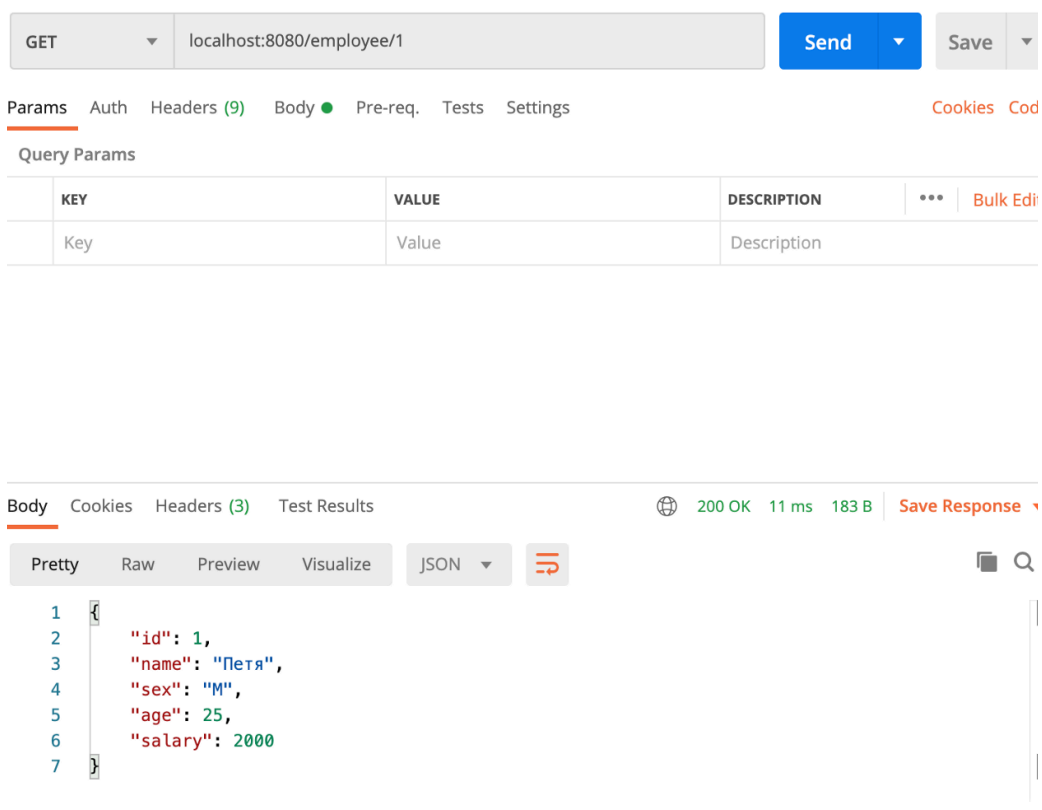


Как вы можете увидеть, сначала мы создали объект хранилища, а вслед за ним хендлер, передав хранилище в конструктор. Это называется *внедрением зависимостей (Dependency Injection)* и используется для достижения разделения ответственности и слабой связности модулей в программе.

Давайте теперь скомпилируем наш код и запустим приложение.



Создание пользователя



Получение пользователя

Как вы можете увидеть, мы успешно смогли создать пользователя, а потом получить о нем информацию по айди 1.

Для составление запросов я использую приложение Postman. Однако можно использовать любую другую утилиту, или тот же `curl` прямо из консоли.

Подведем итоги.

Мы рассмотрели создание простого REST API на Go с использованием библиотеки `github.com/gin-gonic/gin` . Как вы могли увидеть, на разработку такого веб приложения ушло совсем немного времени.

Домашнее задание

Я хочу чтобы вы разобрались с кодом из этого урока самостоятельно. Найти его вы можете тут.

Реализуйте эндпоинт `GET /employee/` , который будет возвращать всех зарегистрированных сотрудников в системе в форме массива.

Теперь даю вам задание поинтересней. Поищите информацию про базу данных MongoDB. Попробуйте установить ее и настроить у себя локально. Поищите в сети уроки по работе с MongoDB в Go. Добавьте еще одну реализацию интерфейса `Storage` , которая будет работать с БД. Возможно вам прийдется немного изменить сигнатуры методов интерфейса, добавив возвращение ошибок.

Создайте еще одну сущность “Отделы”. Добавьте эндпоинты для создания, удаления и получения информации по отделам, а также возможность добавления в них сотрудников.

Раздел 11:

Что Дальше?

Приветствую вас в заключающем разделе книги, посвященной разработке на языке программирования Go.

Мы проделали с вами немалый путь. Начиная с разбора базовой структуры проекта и таких примитивов как функций и переменных, дошли до написания собственного REST API в конце. Надеюсь вам было интересно, материал давался легко, вы успешно справлялись с домашним заданием и самостоятельно искали дополнительный материал.

Цель этой книги– структурировать весь необходимый набор знаний для начала разработки приложений на данном языке. Разобрать все нюансы языка в 10 разделах точно не получится, но я и не преследовал этого.

Закончить книгу я хочу подборкой тем для последующего самостоятельного изучения и советами от себя.

Поищите информацию на следующие темы:

- Тестирование
- Рефлексия
- Дебаг (Отладка)
- Модель памяти и сборщик мусора
- Работа с БД

На данный момент существует множество Open-Source фреймворков и библиотек, которые стали де-факто стандартом индустрии. Подробный список и последующий путь развития с этими технологиями описан в этом репозитории.

Также вопрос структуризации проектов на Go за 10 лет существования этого языка до сих пор остается актуальным. Советую ознакомится с репозиторием, в котором собраны лучшие практики по организации кода. Также предлагаю вам ознакомится с подходом Чистой Архитектуры и ее реализацией на Go.

Ну и самое главное, постоянно практикуйтесь, пробуйте новые технологии и подходы. Находите для себя интересные проекты и реализуйте их. Только сотни часов практики помогут вам стать крутым инженером и специалистом в своей области!

А чтобы не пропускать интересный материал о разработке и профессиональном росте программиста, подписывайтесь на мой Telegram канал, в котором я делюсь свои знаниями и опытом разработки и запуска проектов, а также всем, что считаю интересным.

Никогда не прекращайте обучение! Чести и удачи!